UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Runtime Support For Maximizing Performance on Multicore Systems

Permalink
https://escholarship.org/uc/item/99r4j1ws

Author
Pusukuri, Kishore Kumar

Publication Date
2012

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Runtime Support For Maximizing Performance on Multicore Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Kishore Kumar Pusukuri

September 2012

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Laxmi N. Bhuyan
Dr. Walid Najjar

Copyright by
Kishore Kumar Pusukuri
2012

The Dissertation of Kishore Kumar Pusukuri is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First and foremost I would like to sincerely thank my advisor, Dr. Rajiv Gupta, who was always there for me and shaped my research in many ways. His enthusiasm in research and hard working nature were instrumental in enabling my research to make the progress which it has made. I am particularly grateful for all the freedom he gave me in selecting research problems and his seemingly never-ending trust in my potential.

Next, I would like to thank the members of my dissertation committee, Dr. Laxmi N. Bhuyan and Dr. Walid Najjar for reviewing this dissertation. Their extensive and constructive comments have been very helpful in improving this dissertation.

I was fortunate enough to do various internships during the course of my Ph.D. In particular, my internship experience at Sun Microsystem Laboratories, Menlo Park, CA. was very rewarding and helpful for improving this dissertation. I would like to sincerely thank Dr. David Vengerov, Dr. Steve Heller, Rick Weisner, Darrin Johnson, Eric Saxe, and Kuriakose Kurivilla from Sun Microsystems and Dr. Alexandra Fedorova from Simon Fraser University for making my internship as a valuable research experience.

Next, I would like to express my gratitude to all the members of my research group including Min Feng, Changhui Lin, Yan Wang, Li Tan, and Sai Charan for helping me in many ways during these years. I would like to thank the rest of the Department of Computer Science and Engineering for providing a pleasant research environment.

I would also like to thank all my teachers I have had throughout my life. Last but not least, I would like to thank my family and friends who supported me throughout this endeavor. In particular, I wish to thank my wife, Swathi Sandhya, for her love and support.

To my wife, Swathi Sandhya, who has supported me the entire time.

ABSTRACT OF THE DISSERTATION

Runtime Support For Maximizing Performance on Multicore Systems

by

Kishore Kumar Pusukuri

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2012
Dr. Rajiv Gupta, Chairperson

Since multicore systems offer greater performance via parallelism, future computing is progressing towards use of machines with large number of cores. However, due to the complex interaction among characteristics of multithreaded applications, operating system policies, and architectural characteristics of multicore systems, delivering high performance on multicore systems is a challenging task. This dissertation addresses the above challenge by developing runtime techniques to achieve high performance when running a single multithreaded application as well as high system utilization and fairness when running multiple multithreaded applications. The runtime techniques are based on a simple monitoring system that captures important application characteristics and relevant architectural factors with negligible overhead.

To develop runtime techniques for achieving high performance when running a single multithreaded program on a multicore system, important performance limiting factors that impact the scalability of performance are identified. These factors include the threads configuration (i.e., the number of threads for a multithreaded program that provide the

best speedup) and the thread scheduling and memory allocation polices employed. This dissertation presents two runtime techniques *Thread Reinforcer*, for dynamically determining appropriate threads configuration and *Thread Tranquilizer*, for dynamically selecting appropriate scheduling and memory allocation policies. By dynamically determining the appropriate threads configuration, scheduling policy, and memory allocation policy the performance of applications is maximized.

Lock contention is an important performance limiting factor for multithreaded programs on a multicore system. The dissertation presents two techniques *Thread Shuffling* and *FaithFul Scheduling* to limit the performance impact due to locks. *Thread Shuffling* reduces high lock acquisition latencies, resulting from the NUMA nature of a multicore system, via inter-CPU thread migrations. *FaithFul Scheduling* reduces the durations for which threads hold locks by minimizing lock holder thread preemptions through adaptive time-quanta allocations. These techniques significantly enhance the performance of applications in the presence of high lock contention.

Finally, this dissertation presents a coscheduling technique called $ADAPT$ for achieving high system utilization and fairness when running multiple multithreaded applications on multicore systems. $ADAPT$ uses supervised learning techniques for predicting the effects of interference between programs on their performance and adaptively schedules together programs that interfere with each other's performance minimally. It achieves high throughput, high system utilization, and fairness when running multiple multithreaded applications.

Contents

List of Figures xi

List of Tables xiv

1 Introduction 1
- 1.1 Dissertation Overview . . . 3
  - 1.1.1 Selecting Configuration for Delivering Performance . . . 3
  - 1.1.2 Dealing with Performance Impact of Lock Contention . . . 5
  - 1.1.3 Considering Interactions among Multiple Multithreaded Applications 7
- 1.2 Dissertation Organization . . . 10

2 Determining Number of Threads 11
- 2.1 Identifying Important Performance Limiting Factors . . . 12
  - 2.1.1 OPT-Threads > Number of Cores: . . . 15
  - 2.1.2 OPT-Threads < Number of Cores . . . 21
- 2.2 The Thread Reinforcer Framework . . . 23
  - 2.2.1 Algorithm . . . 25
  - 2.2.2 Finding Thresholds . . . 29
- 2.3 Evaluating Thread Reinforcer . . . 30
- 2.4 Summary . . . 31

3 Selecting System Policies 34
- 3.1 Performance Variation Study . . . 37
  - 3.1.1 Thread Migrations and Memory Allocation Policies . . . 37
  - 3.1.2 Dynamic Priorities and Involuntary Context-switches . . . 42
  - 3.1.3 Combination of Memory Allocation and Scheduling Policies . . . 44
- 3.2 The Thread Tranquilizer Framework . . . 49
- 3.3 Evaluating Thread Tranquilizer . . . 52
  - 3.3.1 Improved Performance and Reduced Performance Variation . . . 52
  - 3.3.2 Improved Fairness and Effectiveness Under High Loads . . . 55
- 3.4 Summary . . . 56

4 Reducing Lock Acquisition Overhead **57**
4.1 Performance Degradation due to Locks and ccNUMA 58
4.2 Thread Shuffling 64
4.2.1 Monitoring Threads 69
4.2.2 Forming Thread Groups 70
4.2.3 Performing Thread Shuffling 71
4.3 Evaluating Thread Shuffling 71
4.3.1 Performance Benefits 73
4.3.2 Cost and Efficiency 74
4.3.3 Time Varying Behavior 75
4.3.4 Multiple Applications 78
4.4 Summary 79

5 Reducing Critical Section Delays **81**
5.1 Interaction between OS Scheduling and Contention Management 82
5.2 FaithFul Scheduling (FF) 85
5.2.1 Scaling-factor Table 86
5.2.2 Dealing with Phase Changes 88
5.2.3 Dealing with Pipeline Parallelism 89
5.2.4 Implementation of FF Policy 90
5.3 Evaluating FF policy 93
5.3.1 Benchmarks 93
5.3.2 Against varying contention levels 94
5.3.3 Against phase changes 95
5.3.4 Against dynamic load changes 96
5.3.5 Performance Improvements 97
5.3.6 Multiple Applications 99
5.4 Summary 100

6 Coscheduling Multiple Multithreaded Applications **101**
6.1 Cache Miss-Ratio vs Lock-contention vs Latency 102
6.2 The ADAPT Framework 105
6.2.1 The Cores Allocator 106
6.2.2 The Policy Allocator 115
6.2.3 Implementation of ADAPT 121
6.3 Evaluating ADAPT 124
6.3.1 Performance and System Utilization Improvements 125
6.3.2 Impact on Performance Variation 127
6.4 Summary 128

7 Related Work **129**
7.1 Balancing Parallelism and Resource Usage 129
7.1.1 One Thread Per Core Model 130
7.1.2 Dynamically Determining Number of Threads 130

7.2 Configuring System Policies . . . 132
7.2.1 Study on Performance Variation . . . 132
7.2.2 Reducing Performance Variation . . . 133
7.2.3 NUMA Optimization Techniques . . . 134
7.3 Lock Contention . . . 134
7.3.1 Synchronization Mechanisms . . . 134
7.3.2 Reducing Lock Acquisition Overhead. . . . 137
7.3.3 Reducing Critical Section Delays . . . 138
7.4 Thread Scheduling . . . 138
7.4.1 Operating System Scheduling . . . 138
7.4.2 Work Stealing . . . 139
7.4.3 Thread Coscheduling . . . 140
7.4.4 Other Scheduling Techniques . . . 143

8 Conclusions **144**
8.1 Contributions . . . 144
8.1.1 Selecting Configuration for Delivering Performance . . . 145
8.1.2 Dealing with Performance Impact of Lock Contention . . . 146
8.1.3 Coscheduling Multiple Multithreaded Programs . . . 147
8.2 Future Directions . . . 147
8.2.1 Enhancing Scalability of Resource Usage Monitoring . . . 147
8.2.2 Using Monitoring for Runtime Power Management . . . 148
8.2.3 Monitoring for Fault Isolation and High Availability . . . 148
8.2.4 Monitoring and Coscheduling for Virtualized Systems . . . 149

Bibliography **150**

List of Figures

1.1 Speedup behavior of PARSEC programs *ferret, bodytrack*, and *facesim* on our 24-core machine for varying number of threads. 4
1.2 Execution times of *streamcluster* and *facesim* programs in 10 runs. 5
1.3 Lock times of PARSEC and SPEC OMP programs when 64 threads are run on 64 cores spread across 4 CPUs. 6
1.4 Lock time (% of execution time), Thread Latency (% of execution time), and Last-level Cache Miss Ratio of *equake* when executed with *facesim*. 9
1.5 Lock Time (% of execution time), Thread latency (% of execution time), and Last-level Cache Miss Ratio of *bodytrack* when executed with *applu*. 9

2.1 Breakdown of elapsed time of critical threads. 14
2.2 Speedup behavior of PARSEC workloads for varying number of threads: the graph on the left shows the behavior of applications where maximum speedup was observed for $Number\ of\ Threads > Number\ of\ Cores = 24$; and the graph on the right shows the behavior of applications where maximum speedup was observed for $Number\ of\ Threads < Number\ of\ Cores = 24$. . 15
2.3 *swaptions*: Cause of Erratic Speedup Changes. 17
2.4 *bodytrack*: Cause of Decline in Speedup. 18
2.5 Maximum Speedup When Number of Threads < Number of Cores. 21
2.6 Voluntary Context Switch Rate. 22
2.7 VCX vs. ICX. 22

3.1 The impact of thread migration on CPU-intensive and memory-intensive single threaded micro-benchmarks. Standard deviation (SD) of the running-times and the average number of thread migrations (last column) per run are presented in the table. 39
3.2 Running-times and cache miss-rates of facesim (memory-intensive program) in 10 runs. Table 3.3 lists the configurations. 45
3.3 Running-times and ICX Rates of mgrid (CPU-intensive program) in 10 runs. Table 3.3 lists the configurations. 46

3.4 Performance variation of memory-intensive programs is reduced with the combination of Random memory allocation and FX scheduling policies. . . 47
3.5 Performance variation of SPECjbb2005 and TATP is reduced with the combination of Random and FX policies. Performance (**throughput**) is also improved. 47
3.6 Performance variation of CPU-intensive programs is reduced with FX scheduling policy. There is no significant effect of Random or RR policies on CPU-intensive programs. 48
3.7 State-transition diagram shows one pass of Thread Tranquilizer. 50
3.8 Performance variation is reduced and performance is improved with Thread Tranquilizer. The bar plot shows another view of the reduction in performance variation (*coefficient of variation*) with Thread Tranquilizer. 51
3.9 Thread Tranquilizer is very effective against parallel runs of more than one application. 54

4.1 Lock times of PARSEC and SPEC OMP programs when 64 threads are run on 64 cores spread across 4 CPUs. 58
4.2 (a) ccNUMA machine; (b) Barrier execution times for varying number of CPUs; and (c) Barrier execution times with varying number of threads. . . 60
4.3 (Barrier) The distribution of lock transfers for successful (*Acquire*; *Release*) operations. 64
4.4 Illustration of Thread Shuffling. 66
4.5 Performance of Thread Shuffling. 72
4.6 Reductions in execution time. 72
4.7 The cost and efficiency of thread shuffling. 74
4.8 Time varying behavior of cumulative lock times without thread shuffling and with thread shuffling. 76
4.9 Time varying behavior of degree of thread shuffling. 77

5.1 Frequent changes in thread priority drastically increases thread context-switches. 83
5.2 The interactions between the TS policy and the spin-then-block policy create vicious cycles between priority changes and context-switches. 84
5.3 BS vs Lock Time (24 threads is 100% load). 85
5.4 Phase changes of ammp. Here ammp is run with 24 threads. Lock-contention value 1 means application experiences lock-contention for 100% of the total elapsed time. 89
5.5 FF policy is very effective against varying contention levels. 95
5.6 FF policy effectively deals with phases of ammp program and improves its performance. 96
5.7 FF policy avoids spikes in the load. 97
5.8 FF policy improves performance of a wide variety of programs. 98
5.9 Performance improvement of SPECjbb2005, facesim, and TATP with FF policy. 98

5.10 FF policy is very effective against parallel runs of more than one application. 100

6.1 The machine has four 16-core CPUs and are interconnected with Hyper-Transport. Table shows the number of cores allocated to two programs A and B in different cores-configurations. 103
6.2 Lock time (% of execution time), Thread Latency (% of execution time), and Last-level Cache Miss Ratio of *equake* when executed with *facesim*. 104
6.3 Lock Time (% of execution time), Thread latency (% of execution time), and Last-level Cache Miss Ratio of *bodytrack* when executed with *applu*. 105
6.4 While APSI has steady behavior, FMA shows a significant phase change. . 114
6.5 CPI is high with next policy. Random policy improves memory-bandwidth. 118
6.6 Random policy reduces lock-contention. 119
6.7 FF policy reduces context-switch rate of APSI. 120
6.8 Size of time-interval vs System overhead. 123
6.9 ADAPT improves TTT and system utilization compared to the default Solaris scheduler. Here, improvement in system utilization = (utilization with ADAPT - utilization with Solaris). 126
6.10 ADAPT improves performance of all the four memory-intensive programs. . 127
6.11 ADAPT improves performance of all the four CPU-intensive programs. . . 128

List of Tables

2.1 Maximum speedups observed and corresponding number of threads for PARSEC programs on the 24-core machine. 13
2.2 Behavior of *ferret*. 20
2.3 Factors considered wrt to the number of threads. 23
2.4 Table 2.4: Algorithm vs. Optimal (PARSEC programs). 31
2.5 Algorithm vs. Optimal (Other programs). 32
2.6 Search Overhead (seconds) for PARSEC programs. 33
2.7 Search Overhead (seconds) for Other programs. 33

3.1 Performance variation of the programs. 36
3.2 Memory Allocation Policies. 41
3.3 Configurations 45
3.4 Thread Tranquilizer improves performance and reduces performance variation simultaneously by applying the combination of Random and FX policies. Standard deviation values are used to allow the readers to easily map the boxplots (length of the boxplot) with the standard deviation values. 53

4.1 Thread shuffling multiple applications. 79

5.1 The Scaling-factor Table. The range of the scaling-factor is 0.10. 88

6.1 Initial predictors and the target *usr_ab* of the PAAP model. 108
6.2 VIF values of PAAP predictors. 108
6.3 Models 110
6.4 VIF values of the PACC model predictors. 112
6.5 Models 112
6.6 The actual and predicted *usr_FA* and *usr_SM* values with PAAP and PACC models are shown here. 113

Chapter 1

Introduction

The advent of multicore architectures provides an attractive opportunity for achieving high performance for a wide range of multithreaded applications. However, exploiting the parallelism they offer, to improve performance of multithreaded programs is a challenging task. This is because of the complex interaction between several factors that affect the performance including: application characteristics (e.g., degree of parallelism, lock contention, and memory bandwidth requirements); operating system policies (e.g., scheduling and memory management policies); and architectural characteristics (e.g., cache hierarchy and non-uniform memory access latencies).

Current practices fail to fully consider the above factors and hence do not fully realize the potential of multicore systems in delivering performance for multithreaded applications. Here are some instances of common practices that illustrate the lack of full consideration of above factors. Multithreaded applications are often written such that it is the responsibility of the user to specify the number of threads to be created to exploit paral-

lelism. An uninformed user may select too few or too many threads for execution and thus achieve suboptimal performance. Modern Operating Systems (OSs) such as OpenSolaris and GNU/Linux do not distinguish between threads from multiple single threaded applications and multiple threads corresponding to a single multithreaded application. Therefore they fail to consider the interactions among different multithreaded applications running on the system and effectively coschedule them. Even under the scenario when a single multithreaded application is being run on the system, these OSs exhibit certain drawbacks. Since they are oblivious to lock contention among threads of a multithreaded application, they fail to consider contention when scheduling threads across the CPUs of a cache-coherent Non-Uniform Memory Access (ccNUMA) multicore system. The default OS memory allocation policy aims to exploit data locality by allocating data close to the threads. Though these policies work well for multiple single threaded programs, this is not the case for multithreaded programs. This is because many multithreaded programs involve communication between threads, leading to contention for shared objects and resources.

To alleviate the above problems, this dissertation develops lightweight runtime techniques to monitor important resource-usage characteristics of multithreaded applications, understand the interactions between OS policies and execution behavior of the applications, and then adaptively create appropriate number of threads, assign appropriate number of cores, and select appropriate OS scheduling and memory allocation policies.

1.1 Dissertation Overview

This dissertation consists of three parts. First, it presents runtime monitoring techniques to select the configuration under which an application performance is expected to be high. The configuration includes factors such as, number of threads, the scheduling policy, and the memory management policy. Second, as an application executes under the selected configuration, it presents techniques to minimize the harmful impact of high lock contention on program performance. In particular, it presents techniques to minimize the times threads spend on acquiring locks and durations for which they hold locks. All of the above work is carried out in context of executing a single multithreaded application on a multicore system. The third part of this dissertation develops runtime techniques for effectively coscheduling multiple multithreaded applications being simultaneously run of the system.

1.1.1 Selecting Configuration for Delivering Performance

The performance of a multithreaded program running on a multicore system is sensitive to the number of threads used to run the application (i.e., *threads configuration*), as it impacts the application's resource-usage characteristics. For example, the number of threads that produce maximum speedups vary widely for PARSEC [1] programs on our 24-core machine running OpenSolaris. As Figure 1.1 shows, not only does the maximum speedup achieved by the three PARSEC programs vary widely (from 4.9x for *facesim* to 14x for *ferret*), the number of threads that produce maximum speedups also varies widely (from 16 threads for *facesim* to 63 threads for *ferret*). Therefore, for achieving the best per-

formance for a multithreaded program, it should be run with a suitable number of threads. However, an uninformed user may select too few or too many threads for execution and thus achieve suboptimal performance. An attractive technique for solving this problem is to dynamically determine the suitable number of threads for running the program. However, dynamically finding a suitable number of threads for a multithreaded program running on a multicore system is a challenging problem because it requires identifying important application characteristics. This dissertation presents a runtime technique called Thread Reinforcer, which monitors important application characteristics at runtime to guide the search for determining appropriate number of threads that are expected to yield the best speedup.

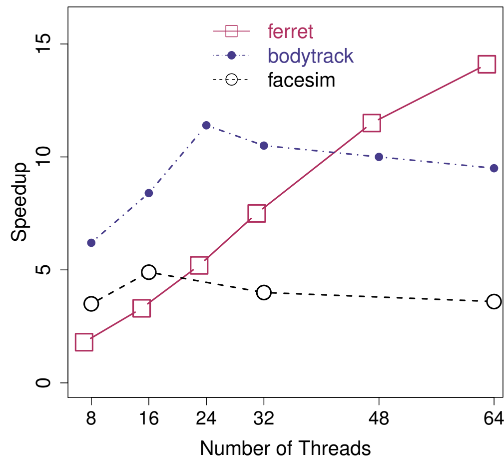


Figure 1.1: Speedup behavior of PARSEC programs *ferret, bodytrack*, and *facesim* on our 24-core machine for varying number of threads.

The performance of a multithreaded program is sensitive to the OS scheduling and memory allocation policies employed. This is because the interactions between program memory reference behavior and the OS scheduling and memory allocation policies

significantly impacts application performance. These interactions make the performance of a program highly sensitive to small changes in resource usage characteristics of the program. In particular, significant variations in the performance are observed from one execution of a program to the next, even when the program input remains unchanged and no other applications are being run on the system. Figure 1.2 shows that the PARSEC programs *streamcluster* and *facesim* exhibit significant variation in their execution times across ten runs even when their inputs remain unchanged. This dissertation presents a runtime technique called Thread Tranquilizer, which simultaneously reduces performance variation and improves performance by adaptively choosing appropriate memory allocation and process scheduling policies based upon the resource usage characteristics of the programs.

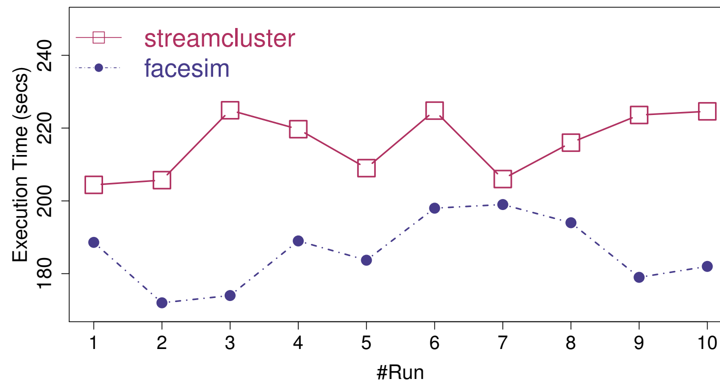
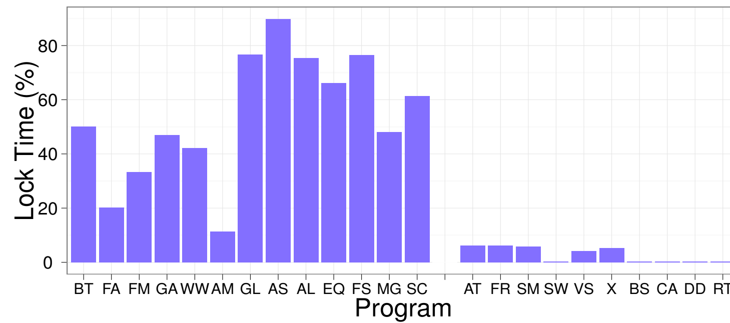


Figure 1.2: Execution times of *streamcluster* and *facesim* programs in 10 runs.

1.1.2 Dealing with Performance Impact of Lock Contention

On a ccNUMA system, the performance of a multithreaded application is often impacted greatly by lock contention. Figure 1.3 presents the lock times of programs from PARSEC and SPEC OMP benchmark suites. Here, the lock time is defined as the percent-

age of elapsed time a program spends on performing operations on locks. As Figure 1.3 shows, the first 13 programs out of a total of 23 programs, exhibit very high lock times when they are run with 64 threads on a 64-core machine.



Lock times.

PARSEC: blackscholes (BS); bodytrack (BT); canneal (CA); dedup (DD); fluidanimate (FA); facesim (FS); ferret (FR); raytrace (RT); streamcluster (SC); swaptions (SW); vips (VS); x264 (X); **SPEC OMP:** applu (AL); ammp (AM); art (AT); apsi (AS); equake (EQ); fma3d (FM); gafort (GA); galgel (GL); mgrid (MG); swim (SM); wupwise (WW)

Figure 1.3: Lock times of PARSEC and SPEC OMP programs when 64 threads are run on 64 cores spread across 4 CPUs.

This dissertation considers the following two reasons for high lock times:

- *High lock acquisition latencies.* On a ccNUMA system the performance of a multithreaded application is highly sensitive to the distribution of application threads across the multiple multicore CPUs. In particular, when multiple threads compete to acquire a lock, due to the NUMA nature of the architecture, the time spent on *acquiring locks* by threads distributed across different CPUs is greatly increased.

- *Prolonged durations for which locks are held.* Under high load conditions, frequent preemption of lock holder threads can slow down the progress of lock holder threads and increase lock times. In particular, negative interaction between the Time Share (TS) thread scheduling policy and the spin-then-block lock-contention management policy dramatically increases lock holder thread preemptions under high loads.

To address the above problems, this dissertation presents two techniques *Thread Shuffling* and *Faithful Scheduling*. *Thread Shuffling* minimizes the times threads spend on acquiring locks through inter-CPU thread migrations and *Faithful Scheduling* minimizes lock holder thread preemptions via adaptive time-quanta allocations.

1.1.3 Considering Interactions among Multiple Multithreaded Applications

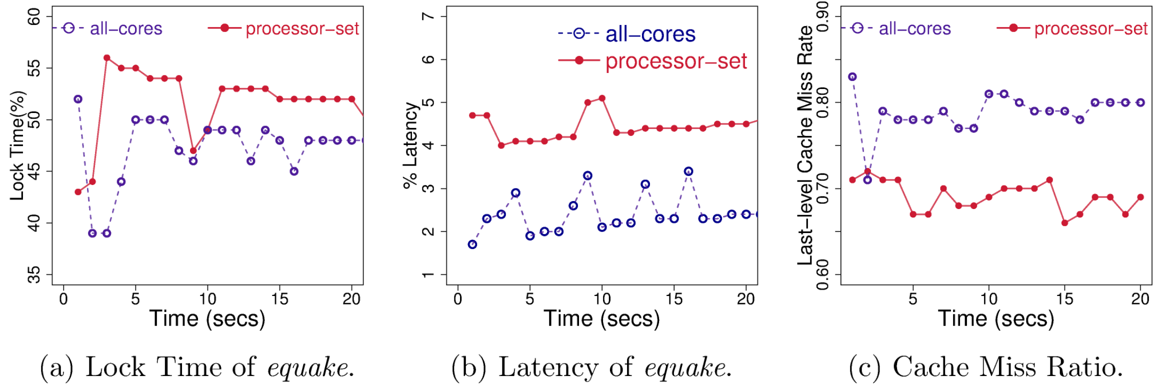
Since the performance of multithreaded applications often does not scale to fully utilize the available cores in a multicore system, simultaneously running multiple multithreaded applications becomes inevitable to fully utilize such machines. However, coscheduling multithreaded programs effectively on such machines is a challenging problem.

There are two different cores configurations that can be used for coscheduling multiple multithreaded programs on a multicore system: *all-cores* configuration; and *processor-set* configuration. In all-cores configuration each program is run using all the cores while in processor-set configuration each program is run on a separate processor-set to minimize interference between the programs. A processor-set is a pool of cores such that if a multithreaded program is assigned to a processor-set, OS migrates the threads of the program

only across the cores belonging to the processor-set for balancing load. Next, it is illustrated how the cores configuration impacts the lock times (time spent on lock operations and in critical sections), latency (time ready threads spend waiting for a core to become available), and last level cache miss ratios (cache misses/accesses). These impacts collectively determine which configuration is the most suitable.

When two memory-intensive and high lock contention multithreaded programs *facesim* and *equake* are coscheduled in the above two configurations, all-cores configuration gives better overall performance. This is even though, due to their memory-intensive nature, these programs suffer from higher last-level cache miss ratios under all-cores configuration. This is because *facesim* and *equake* are also high lock-contention programs. As Figure 1.4 shows, *equake* experiences high lock times and latency in processor-set configuration compared to all-cores configuration. Likewise, although not shown here, *facesim* also experiences high lock times and latency in processor-set configuration compared to all-cores configuration. Thus, the trade-off between lock contention, latency, and last-level cache miss ratio results in the all-cores configuration delivering better performance.

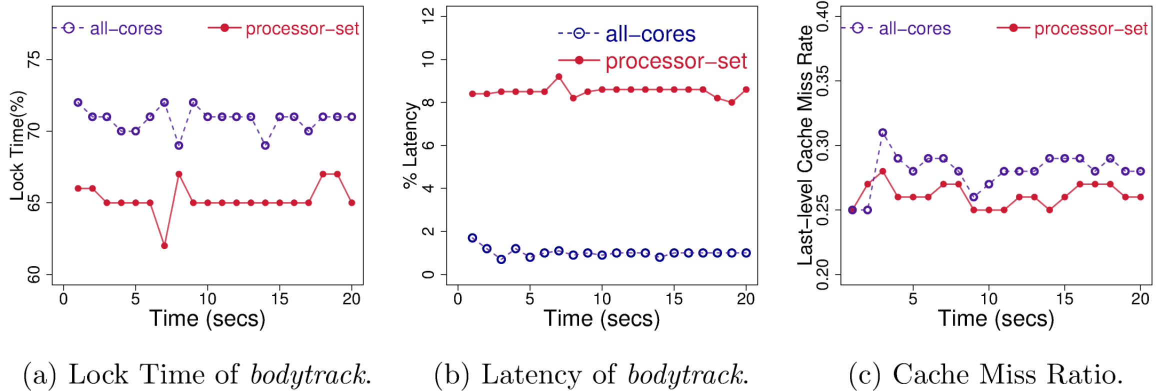
When two CPU intensive and high lock-contention multithreaded programs *bodytrack* and *applu* are co-scheduled, processor-set configuration provides high performance compared to all-cores configuration. As shown in Figure 1.5, thread latency of *bodytrack* is low in all-cores configuration compared to processor-set configuration. Likewise, although not shown here, the thread latency of *applu* is also low in all-cores configuration. However, as shown in Figure 1.5, lock time and last level cache miss ratio for *bodytrack* are higher in all-cores configuration. Likewise, the lock times of *applu* is also high in all-cores configu-



(a) Lock Time of *equake*. (b) Latency of *equake*. (c) Cache Miss Ratio.

Figure 1.4: Lock time (% of execution time), Thread Latency (% of execution time), and Last-level Cache Miss Ratio of *equake* when executed with *facesim*.

ration. Therefore, the trade-off between lock times, last level cache miss ratio, and thread latency results in the processor-set configuration delivering better performance.



(a) Lock Time of *bodytrack*. (b) Latency of *bodytrack*. (c) Cache Miss Ratio.

Figure 1.5: Lock Time (% of execution time), Thread latency (% of execution time), and Last-level Cache Miss Ratio of *bodytrack* when executed with *applu*.

Therefore, for effective coscheduling of multithreaded programs, cores must be allocated adaptively according to the resource-usage characteristics of the multithreaded programs being run simultaneously. To address this problem, this dissertation presents

a runtime technique called ADAPT. It uses supervised learning techniques for predicting the effects of interference between programs on their performance and adaptively schedules together programs that interfere with each other's performance as little as possible. It achieves high throughput, high system utilization, and fairness when running multiple multithreaded applications.

1.2 Dissertation Organization

The contents of this dissertation are organized as follows. Chapter 2 describes the *Thread Reinforcer* for dynamically determining appropriate number of threads for a multithreaded application. Chapter 3 presents the *Thread Tranquilizer* to select appropriate memory allocation and scheduling policies according to the resource-usage characteristics of multithreaded applications. Chapter 4 describes the *Thread Shuffling* technique to minimize lock acquisition latencies of threads of a multithreaded application running on ccNUMA multicore system. Chapter 5 presents the *Faithful Scheduling* to reduce lock holder thread preemptions. In chapter 6, the technique ADAPT, for coscheduling multiple multithreaded applications is described. Related work is given in Chapter 7 and the conclusions of the dissertation are summarized in Chapter 8.

Chapter 2

Determining Number of Threads

The performance of a multithreaded program running on a multicore system is sensitive to the number of threads used to run the multithreaded program (i.e., the *threads configuration*), as it impacts the application's resource-usage characteristics. Using too few threads leads to under exploitation of parallelism in the application and using too many threads degrades application performance because of lock-contention and contention of shared resources. One simple method for finding the appropriate number of threads is to run the application with different number of threads and find the number of threads that gives the best performance. However, this is time-consuming, does not work if the number of threads is input dependent, does not adapt to the system's dynamic behavior, and therefore is not a practical solution. An attractive alternative technique for solving this problem is to dynamically determine the suitable number of threads for running a multithreaded program on a multicore system. This dissertation presents such a technique called *Thread Reinforcer*.

2.1 Identifying Important Performance Limiting Factors

Dynamically finding a suitable number of threads for a multithreaded program running on a multicore system is a challenging problem because it requires identifying important application characteristics. Therefore, first a performance study of eight PARSEC programs for different numbers of threads ranging from a few threads to 128 threads was conducted. Table 2.1 shows the maximum speedup (Max Speedup) for each program on the 24-core machine along with the minimum number of threads (called OPT Threads) that produced this speedup. As shown in Table 2.1, not only does the maximum speedup achieved by these programs vary widely (from 3.6x for *canneal* to 21.9x for *swaptions*), the number of threads that produce maximum speedups also varies widely (from 16 threads for *facesim* to 63 threads for *ferret*). Moreover, for the first five programs the maximum speedup results from creating more threads than the number of cores, i.e. OPT-Threads is greater than 24. For the other three programs OPT-Threads is less than the number of cores.

The above performance study shows that the number of threads used to run a multithreaded application is crucial in achieving high performance on a multicore system. Next, using the *prstat* [2] utility, the following main components of the execution times for threads in each application are considered to help identify the performance limiting factors.

1. **User**: The percentage of time a thread spends in user mode.

2. **System**: The percentage of time a thread spends in processing the following system events: system calls, system traps, text page faults, and data page faults.

Table 2.1: Maximum speedups observed and corresponding number of threads for PARSEC programs on the 24-core machine.

Program	Max Speedup	OPT Threads
swaptions	21.9	33
ferret	14.1	63
bodytrack	11.4	26
blackscholes	4.9	33
canneal	3.6	41
fluidanimate	12.7	21
facesim	4.9	16
streamcluster	4.2	17

3. **Lock-contention**: The percentage of time a thread spends waiting for locks, condition-variables etc.

4. **Latency**: The percentage of time a thread spends waiting for a CPU. In other words, although the thread is ready to run, it is not scheduled on any core.

Next, the above times for all threads are studied to see if changes in these times would explain the changes in speedups observed by varying number of threads. Although the data for all threads was examined, it quickly became apparent that in many programs not all threads were critical to the overall speedup. Based on this study, *critical threads* were identified and then studied in greater detail. The critical threads for each application are

Program	**Critical Threads**
ferret	Rank stage Threads
canneal	Main Thread
swaptions	Worker Threads
blackscholes	Main Thread
bodytrack	All Threads
fluidanimate	Worker Threads
streamcluster	Worker Threads
facesim	All Threads

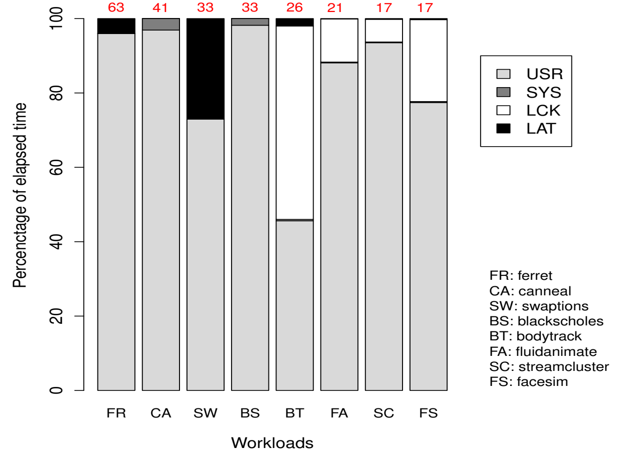


Figure 2.1: Breakdown of elapsed time of critical threads.

listed in the table below. Figure 2.1 provides the breakdown of the time of critical threads in the above four categories – this data is for the OPT-Threads run and is the average across all critical threads. As shown in Figure 2.1, in some programs lock time (LCK) plays a critical role, in others the threads spend significant time waiting for a CPU as latency (LAT) is high, and the system time (SYS) is the highest for *canneal* and *blackscholes*.

In the subsequent sections the above times for each of the programs are analyzed in greater detail to study their relationship with speedup variations that are observed when number of threads is varied. Furthermore, the program characteristics that are the causes for the observed speedup variations are identified.

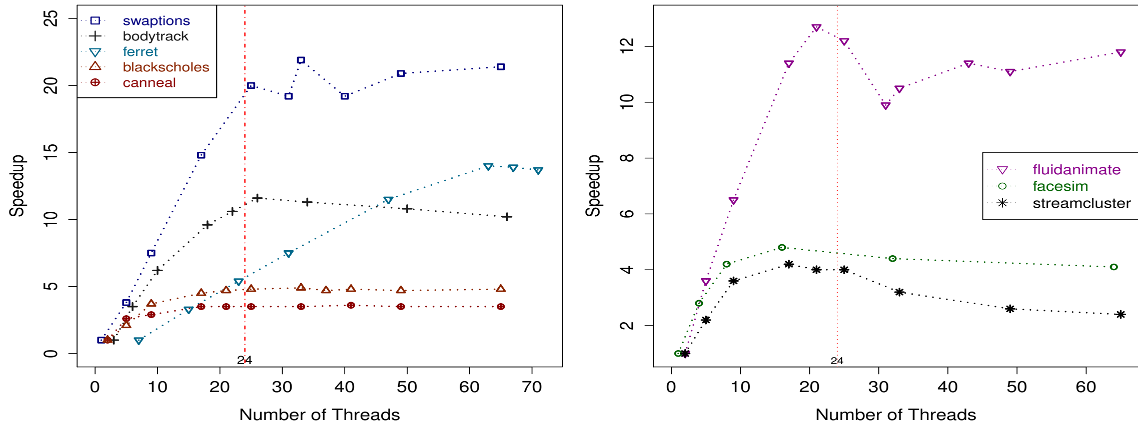


Figure 2.2: Speedup behavior of PARSEC workloads for varying number of threads: the graph on the left shows the behavior of applications where maximum speedup was observed for $Number\ of\ Threads > Number\ of\ Cores = 24$; and the graph on the right shows the behavior of applications where maximum speedup was observed for $Number\ of\ Threads < Number\ of\ Cores = 24$.

2.1.1 OPT-Threads > Number of Cores:

Scalable Performance. As shown in the graph on the left in Figure 2.2, for the three programs (*swaptions*, *bodytrack*, and *ferret*) in this category, the speedups scale quite well. As the number of threads is varied from a few threads to around 24, which is the number of cores, the speedup increases linearly with the number of threads. However, once the number of threads is increased further, the three programs exhibit different trends as described below:

- (Erratic) *swaptions*: Although the speedup for *swaptions* can be significantly increased -- from 20 for 25 threads to 21.9 for 33 threads -- its trend is erratic. Sometimes the addition of more threads increases the speedup while at other times an increase in

number of threads reduces the speedup.

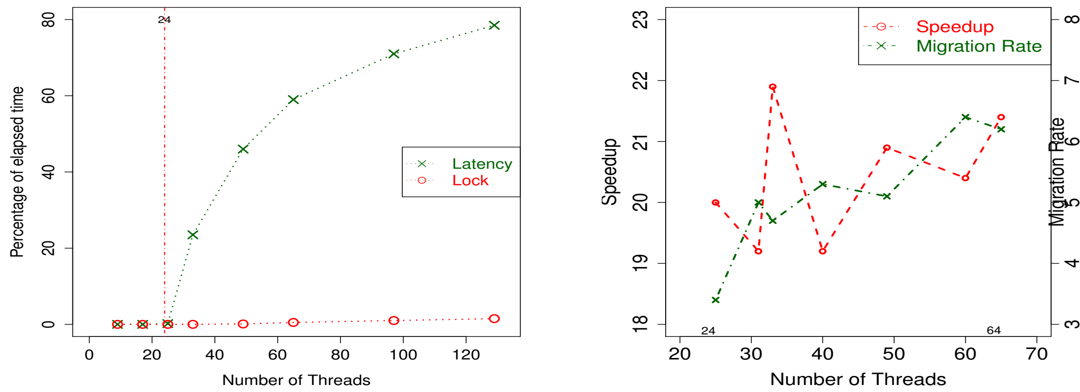
- (Steady Decline) *bodytrack*: The speedup for *bodytrack* decreases as the number of threads is increased beyond 26 threads. The decline in speedup is quite steady.

- (Continued Increase) *ferret*: The speedup for *ferret* continues to increase linearly. In fact the linear increase in speedup is observed from the minimum number of 6 threads all the way up till 63 threads. Interestingly no change in behavior is observed when the number of threads is increased from less than the number of cores to more than the number of cores.

Next the differing behaviors are traced back to specific characteristics of these programs.

swaptions: To understand the erratic behavior of speedups observed in *swaptions*, first the lock contention and latency information are examined. As shown in Figure 2.3(a), the lock contention (LOCK) is very low and remains very low throughout and the latency (LAT) increases steadily which shows that the additional threads created are ready to run but are simply waiting for a CPU (core) to become available. This causes the execution time to remain the same. However, upon further analysis, the correlation between the speedup behavior and thread migration rate was identified as the reason behind the erratic behavior of speedups observed in swaptions. As shown in Figure 2.3(b), when the migration rate goes up, the speedup goes down and vice versa – the migration rate was measured using the *mpstat* [2] utility. Migrations are expensive events as they cause a thread to pull its working set into cold caches, often at the expense of other threads [2]. Thus, the speedup

behavior is a direct consequence of changes in thread migration rate.

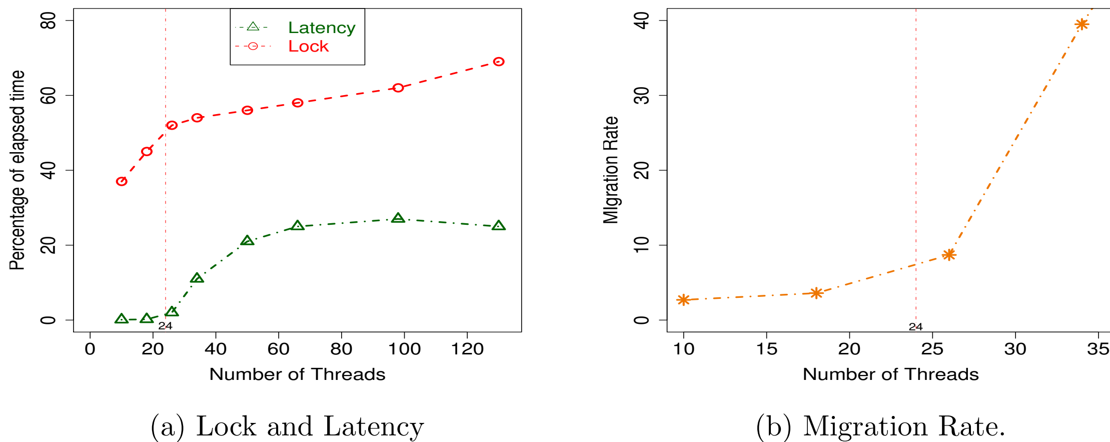


(a) Lock and Latency (b) Speedup vs. Mig. Rate.

Figure 2.3: *swaptions*: Cause of Erratic Speedup Changes.

The OS scheduler plays a significant role here as it is responsible for making migration decisions. When a thread makes a transition from sleep state to a ready-to-run state, if the core on which it last ran is not available, the thread is likely to be migrated to another available core. In general, one would expect more migrations as the number of threads increases beyond the number of cores. However, if the number of threads is divisible by the number of cores, then the likelihood of migrations is less compared to when this is not the case. In the former case, the OS scheduler can allocate equal number of threads to each core, balancing the load, and thus reducing the need for migrations. Thus, *variations in degree of load balancing* across cores causes corresponding variations in thread migration rate and hence the observed speedups. For example, in Figure 2.3(b), the thread migration rate for 48 threads on 24 cores is lower than thread migration rate for 40 threads on 24 cores. Moreover, low thread migration rate can be expected when the input load (128 swaptions) is perfectly divisible by the number of threads (e.g., 16, 32, 64 etc.).

bodytrack: Next consider the steady decline in speedup observed for *bodytrack*. Figure 2.4(a) shows that although the latency (LAT) rises as more threads are created, so does the lock contention (LOCK) which is significant for *bodytrack*. In addition, *bodytrack* is an I/O intensive benchmark where I/O is performed by all the threads and it produces around 350 *ioctl()* calls per second. Both lock contention and I/O have the consequence of increasing the thread migration rate. This is because both lock contention and I/O result in *sleep to wakeup* and *run to sleep* state transitions for the threads involved. When a thread wakes up from the sleep state, the OS scheduler immediately tries to give a core to that thread, if it fails to schedule the thread on the same core that it used last, it migrates the thread to another core. As shown in Figure 2.4(b), the thread migration rate for *bodytrack* rises with the number of threads which causes a steady decline in its speedup.



(a) Lock and Latency

(b) Migration Rate.

Figure 2.4: *bodytrack*: Cause of Decline in Speedup.

ferret: The behavior of this program is interesting as the speedup for it increases linearly starting from 6 threads to all the way up to 63 threads even though only 24 cores are available. To understand this behavior the program was examined in greater detail. The program is divided into six pipeline stages – the results of processing in one stage are

passed on to the next stage. The stages are: Load, Segment, Extract, Vector, Rank, and Out. The first and last stage have a single thread but each of the intermediate stages are a pool of n threads. Between each pair of consecutive stages a queue is provided through which results are communicated and locking is used to control queues accesses.

The reason for the observed behavior is as follows. The *Rank* stage performs most of the work and thus the speedup of the application is determined by the *Rank* stage. Moreover the other stages perform relatively little work and thus their threads together use only a fraction of the compute power of the available cores. Thus, as long as cores are not sufficiently utilized, more speedup can be obtained by creating additional threads for the *Rank* stage. The maximum speedup of 14.1 for *ferret* was observed when the total number of threads created was 63 which actually corresponds to 15 threads for *Rank* stage. That is, the linear rise in speedup is observed from 1 thread to 15 threads for the *Rank* stage which is well under the total of 24 cores available – the remaining cores are sufficient to satisfy the needs of all other threads.

The justification of the above reasoning can be found in the data presented in Table 2.2 where the average percentage of USR and LOCK times for all stages and SYS time for only *Load* stage are shown because all other times are quite small. The threads belonging to *Segment*, *Extract*, and *Out* stages perform very little work and mostly spend their time waiting for results to become available in their incoming queues. While the *Load* and *Vector* stages do perform significant amount of work, they nevertheless perform less work than the *Rank* stage. The performance of the *Rank* stage determines the overall speedup – adding additional threads to the *Rank* stage continues to yield additional speedups

Table 2.2: Behavior of *ferret*.

Total Threads	n	Load (1)			Segment (n)		Extract (n)		Vector (n)		Rank (n)		Out (1)		Speedup
		USR	SYS	LOCK	USR	LOCK	USR	LOCK	USR	LOCK	USR	LOCK	USR	LOCK	
15	3	22	4	74	8	92	1	99	44	56	100	0	0.5	99.3	3.3
31	7	44	7.8	48	6.7	93	1	99	43	57	100	0	0.6	99	7.5
47	11	56	11.3	32	5.4	95	1	99	40	60	100	0	0.7	99	11.5
55	13	64	14	19	5	95	1	99	44	56	98	0	0.7	99	12.5
63	15	79	20	0	5	95	1	99	43	57	96	0	0.7	99	**14.1**
71	17	77	20	0	5	95	1	99	37	63	80	16	0.7	99	13.8
87	21	78	17	0	4	96	1	99	28	72	65	33	0.4	99.3	13.7
103	25	75	17	0	3	97	1	99	24	76	53.5	45	0.4	99.3	13.4
119	29	74	17	0	3	97	1	99	20	80	46	52.5	0.4	99.4	13.2
127	31	70	20	0	3	97	1	99	19	81	40	59	0.4	99.4	13.1

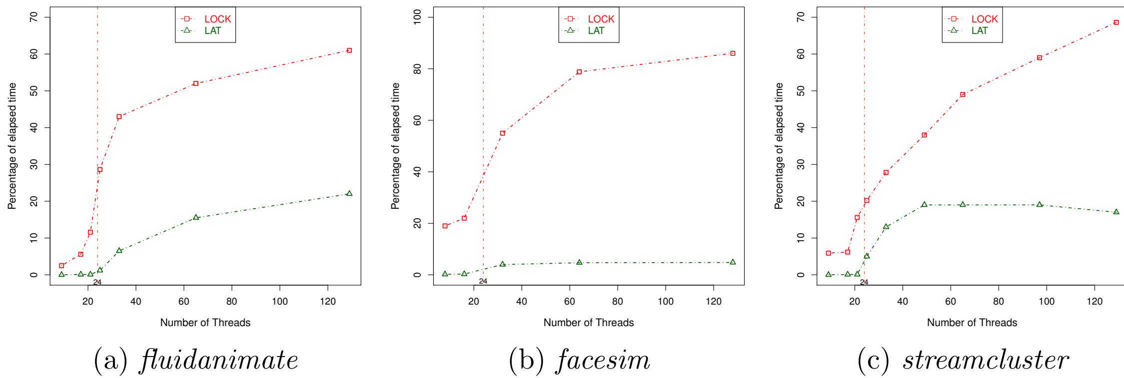
as long as this stage does not experience lock contention. Once lock contention times start to rise (starting at $n = 17$), the speedup begins to fall.

To further confirm the above observations, the number of threads in the *Rank* stage were increased and the number of threads in other intermediate stages was lowered. While the configuration with (1, 10, 10, 10, 16, 1) threads gave a speedup of 13.9, the configuration with (1, 16, 16, 16, 16, 1) threads gave the same the speedup. This further confirms the importance of the *Rank* stage.

Performance Does Not Scale. (*blackscholes* and *canneal*) Although the maximum speedups of these programs (4.9 and 3.6) are observed when 32 and 40 worker threads are created, the speedups of both these programs increase very little beyond 16 worker threads. This is because most of the work is performed by the main thread causing the overall CPU utilization to become low. The main thread takes up 85% and 70% of the time for *blackscholes* and *canneal* respectively. During rest of the time the parallelized part of the program is executed by worker threads. The impact of parallelization of this limited part on the overall speedup diminishes with increasing number of threads.

2.1.2 OPT-Threads < Number of Cores

The three programs where maximum speedup was achieved using fewer threads than number of cores are *fluidanimate*, *facesim*, and *streamcluster*. In these programs the key factor that limits performance is lock contention. Figure 2.5 shows that the time due to lock contention (LOCK) dramatically increases with number of threads while the latency (LAT) shows modest or no increase. The maximum speedups are observed at 21 threads for *fluidanimate*, 16 threads for *facesim*, and 17 threads for *streamcluster*.



(a) *fluidanimate* (b) *facesim* (c) *streamcluster*

Figure 2.5: Maximum Speedup When Number of Threads < Number of Cores.

When the number of threads is less than the number of cores, the load balancing task of the OS scheduler becomes simple and thread migrations become rare. Thus, unlike *swaptions* and *bodytrack* where maximum speedups were observed for greater than 24 threads, thread migration rate does not play any role in the performance of the three programs considered in this section. However, the increased lock contention leads to slowdowns because of increased *context switch rate*. Context-switches are divided into two types: involuntary context-switches (ICX) and voluntary context-switches (VCX). Involuntary context-switches occur when threads are involuntarily taken off a core (e.g., due to

expiration of their time quantum). Voluntary context-switches occur when a thread performs a blocking system call (e.g., for I/O) or when it fails to acquire a lock. In such cases a thread voluntarily releases the core using the *yield()* system call before going to sleep using *lwp_park()* system call. Therefore as more threads are created and lock contention increases, VCX context switch rate rises as shown in Figure 2.6. It is also worth noting that most of the context switches performed by the three programs are in the VCX category. The VCX and ICX data are measured using the *prstat* utility. Figure 2.7 shows that the percentage of VCX ranges from 84% to 97% for the three programs considered here. In contrast, the VCX represents only 11% and 13% of context switches for *swaptions* and *ferret*.

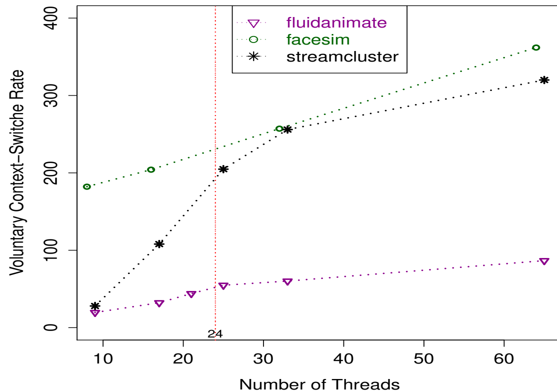


Figure 2.6: Voluntary Context Switch Rate.

Program	**VCX** (%)	**ICX** (%)
fluidanimate	84	16
facesim	97	3
streamcluster	94	6
swaptions	11	89
ferret	13	87

Figure 2.7: VCX vs. ICX.

From the above scalability analysis of PARSEC programs, it is clear that the speedup behavior of an application correlates with variations in lock time (LOCK), migration rate (MIGR_RATE), voluntary context-switch rate (VCX_RATE), and CPU utilization (CPU_UTIL). Moreover, the relationship between above factors and the number of threads

from the above scalability analysis is also derived. Table 2.3 shows the factors that play an important role when the number of threads is no more than the number of cores (i.e., 24) versus when the number of threads is greater than the number of cores. The lock contention is an important factor which must be considered throughout. However, for ≤ 24 threads the voluntary context-switch rate is important while for > 24 threads the thread migration rate is important to consider. In general, the limit of parallelism for a program may reach at any time. Therefore the degree of parallelism (CPU utilization) is an important factor to consider throughout. The above observations are a direct consequence of our observations made during the study presented earlier.

Table 2.3: Factors considered wrt to the number of threads.

Factor	≤ 24 Threads	> 24 Threads
Lock-contention	Yes	Yes
Voluntary Context-switches	Yes	-
Migrations	-	Yes
Degree of parallelism	Yes	Yes

Thus, *Thread Reinforcer* monitors the above characteristics for dynamically determining a suitable number of threads for a multithreaded application.

2.2 The Thread Reinforcer Framework

The applications considered allow the user to control the number of threads created using the command line argument n. Since our experiments show that the number of threads

that yield peak performance varies greatly from one program to another, the selection of n places an added burden on the user. Therefore, in this section, a framework for automatically selecting the number of threads is developed.

The framework runs the application in two steps. In the first step the application is run multiple times for short durations of time during which its behavior is monitored and based upon runtime observations Thread Reinforcer searches for the appropriate number of threads. Once this number is found, in the second step, the application is fully reexecuted with the number of threads determined in the first step. Since the applications do not support varying number of threads online, they need to be rerun for short durations. Thus, Thread Reinforcer does not consider phase changes of the target program. However, out of the 16 programs tested, only the *ammp* program shows two significantly different phases and its first phase dominates the execution. Therefore Thread Reinforcer works well also for the *ammp* program.

Each time an application is to be executed on a new input, Thread Reinforcer is used to determine the appropriate number of threads for that input. This is done in order to handle applications whose runtime behavior is input dependant and thus the optimal number of threads may vary across inputs. Our goal is twofold: to find the appropriate number of threads and to do so quickly so as to minimize runtime overhead. Since the applications considered take from tens of seconds to a few hundred seconds to execute in the OPT-Threads configuration, Thread Reinforcer is designed in such a way that the times it takes to search for appropriate number of threads is only a few seconds. This ensures that the benefits of the algorithm outweigh the runtime overhead of using it.

2.2.1 Algorithm

Based on the above observations, *Thread Reinforcer* searches for appropriate number of threads in the range of T_{min} and T_{max} threads as follows. It runs the application for increasing number of threads for short time durations. Each successive run contains either T_{step} or $T_{step}/2$ additional threads. The decision of whether or not to run the program for higher number of threads and whether to increase the number of threads by T_{step} or $T_{step}/2$, is based upon changes in profiles observed over the past two runs. The profile consists of the four factors: lock contention, thread migration rate, voluntary context switch rate, and the degree of parallelism (processor utilization). The values of each of these measures are characterized as either *low* or *high* based upon set *thresholds* for these parameters. Our algorithm not only examines the current values of above profiles, it also examines how rapidly they are changing. The changes of these values over the past two runs are denoted as ΔLock-contention, ΔMigration-rate, ΔVoluntary-context-switch-rate, and ΔCPU-utilization. The changes are also characterized as *low* and *high* to indicate whether the change is gradual or rapid. At any point in the penultimate run represents the current best solution of our algorithm and the last run is compared with the previous run to see if it should be viewed as an improvement over the penultimate run. If it is considered to be an improvement, then the last run becomes our current best solution. Based upon the strength of improvement, the program is run with T_{step} or $T_{step}/2$ additional threads. The above process continues as long as improvement is observed. Eventually Thread Reinforcer terminates if no improvement or degradation is observed, or the maximum number of threads T_{max} is reached.

Algorithm 1 presents Thread Reinforcer in detail. Thread Reinforcer is initiated

Algorithm 1: FindN() returns the best value for command line parameter.

Thresholds and Profile Data Structure:

-- Profile P: (CPU_UTIL, LOCK, VCX_RATE, MIGR_RATE);

-- (T_{best}, N_{best}) is the current best solution; P_{best} is its profile;

-- (T_{try}, N_{try}) is the next solution tried; P_{try} is its profile;

-- Δ P.field = P_{try}.field - P_{best}.field;

-- *low* returns true/false if P.field or Δ P.field is *low*/not *low*;

-- *high* returns true/false if P.field or Δ P.field is *high*/not *high*;

-- T_{step} is increments in which number of threads is increased;

-- T_{min}/T_{max} is minimum/maximum number of threads allowed;

Subroutines:

-- Convert(T): converts number of threads T into command-line parameter value N;

-- getProfile(N): collect profile P for 100 milliseconds run with parameter N;

Input : Target Multithreaded Benchmark Program

Output: Returns command-line parameter corresponding to the optimal number of threads

$T_{best} = T_{min}$; N_{best} = Convert(T_{best});P_{best} = getProfile(N_{best});

$T_{try} = T_{min} + T_{step}$; N_{try} = Convert(T_{try});P_{try} = getProfile(N_{try});

if *Terminate*(P_{best},P_{try}) == *TRUE* **then**

  return (N_{best});

end

repeat

  $T_{best} = T_{try}$; $N_{best} = N_{try}$; $P_{best} = P_{try}$;

  T_{try} = ComputeNextT(P_{best},P_{try});

  N_{try} = Convert(T_{try});

  P_{try} = getProfile(N_{try});

  terminate = Terminate(P_{best},P_{try});

until *terminate* == *TRUE*;

return (N_{best});

Algorithm 1: FindN() continued

Terminate(P_{best}, P_{try}):

```
// terminate if no more parallelism was found.
```

if *low*($P_{try}.\Delta CPU_UTIL$) **then**

  return (TRUE);

```
// terminate for high lock contention, VCX rate, and migration rate.
```

else if *high*($P_{try}.LOCK$) **then**

  if *high*($P_{try}.\Delta LOCK$) **or** ($T_{try} \leq NumCores$ **and** *high*($P_{try}.\Delta VCX_RATE$)) **or** ($T_{try} >$ $NumCores$ **and** *high*($P_{try}.\Delta$ $MIGR_RATE$)) **then**

    return (TRUE);

  end

```
// terminate if no more threads can be created.
```

else if $T_{try} == T_{max}$ **then**

  $T_{best} = T_{try}$; $N_{best} = N_{try}$;

  return (TRUE);

```
// otherwise do not terminate.
```

else

  return (FLASE);

end

Algorithm 1: FindN() continued

```
ComputeNextT(P_best, P_t ry):

if T_try ≤ NumCores then
    if low(P_try.LOCK) or low(P_try.ΔVCX_RATE) or (high(P_try.LOCK) and
    low(P_try.ΔLOCK)) then
        ΔT = T_step;
    else
        ΔT = (T_step)/2;
    end
else                                                    // T_try > NumCores
    if low(P_try.LOCK) or low(P_try.ΔMIGR_RATE) then
        ΔT = T_step;
    else
        ΔT = (T_step)/2;
    end
end
return (minimum(T_try + ΔT, T_max);
```

by calling $FindN()$ and when it terminates it returns the value of command like parameter n that is closest to the number of threads that are expected to give the best performance. $FindN()$ is iterative – it checks for termination by calling $Terminate()$ and if termination conditions are not met, it calls $ComputeNextT()$ to find out the number of threads that must be used in the next run. Consider the code for $Terminate()$. It first checks if processor utilization has increased from the penultimate run to the last run. If this is not the case then the algorithm terminates otherwise the lock contention is examined for termination. If lock contention is high then termination occurs if one of the following is true: lock contention has increased significantly; number of threads is no more than the number of cores and

voluntary context switch rate has sharply increased; or number of threads is greater than the number of cores and thread migration rate has sharply increased. Finally, the algorithm is not terminated if the above termination condition is not met. However, it is terminated if the upper limit for number of threads is reached. Before iterating another step, the number of additional threads to be created is determined. $ComputeNextT()$ does this task – if the overheads of locking, context switches, or migration rate increase slowly then T_{step} additional threads are created; otherwise $T_{step}/2$ additional threads are created.

Next section presents evaluation of *Thread Reinforcer* against PARSEC and SPEC OMP programs. However, before experimentation, the various thresholds used by Thread Reinforcer need to be selected.

2.2.2 Finding Thresholds

Three of the eight programs: *fluidanimate*, *facesim*, and *blackscholes* to guide the selection of thresholds. These selected programs were run with *small* inputs: for *fluidanimate* and *blackscholes*, the *simlarge* input is used and for *facesim* the *simsmall* input is used. Next, the profiles of the programs are studied and the threshold values for LOCK, MIGR_RATE, VCX_RATE, CPU_UTIL are identified as follows. The threshold values were chosen such that after reaching the threshold value, the value of the profile characteristic became more sensitive to the number of threads and showed a rapid increase. There are two types of threshold values: absolute thresholds and Δ thresholds. The Δ threshold indicates how rapidly the corresponding characteristic is changing. For LOCK and VCX_RATE both thresholds are used by our algorithm. For MIGR_RATE and CPU_UTIL only Δ threshold

is used. It should be noted that the three programs that were chosen to help in selection of thresholds collectively cover all four of the profile characteristics: for *fluidanimate* both LOCK and MIGR_RATE are important; for *facesim* VCX_RATE is important; and for *blackscholes* CPU_UTIL is important.

2.3 Evaluating Thread Reinforcer

First Thread Reinforcer is evaluated against PARSEC programs. Table 2.4 presents the number of threads found by Thread Reinforcer and compares it with the OPT-Threads number that was reported earlier in Table 2.1. The corresponding speedups for these number of threads are also reported. As shown in Table 2.4, for the first four programs (*facesim*, *bodytrack*, *swaptions*, *ferret*) the number of threads found by our algorithm is exactly the same as OPT-Threads. For the next two programs, *fluidanimate* and *streamcluster*, the numbers are close as they differ by $T_{step}/2 (= 4)$ and $T_{step} (= 8)$ respectively. The loss in speedups due to this suboptimal choice of the number of threads in quite small. For the last two programs, *canneal* and *blackscholes*, the number of threads Thread Reinforcer selects is much smaller than OPT-Threads. This is because the speedup of these programs rises very slowly and thus the change in CPU-utilization is quite low. The search overhead varies from 0.5 seconds to 3.2 seconds while the parallel execution times of the programs range from 21.9 seconds to 226 seconds.

Since Thread Reinforcer uses the thresholds of PARSEC programs, Thread Reinforcer is evaluated against programs other than PARSEC. For this, Thread Reinforcer is tested against seven SPEC OMP programs and PBZIP2 program, a total of eight other

programs. As shown in Table 2.5, Thread Reinforcer identifies optimal or near optimal number of threads for most of these programs. Moreover, the search overhead is very low compared to the parallel execution-time of the programs. Tables 2.6 and 2.7 show this.

Table 2.4: Algorithm vs. Optimal (PARSEC programs).

Program	**Number of Threads**		**Speedups**	
	Algorithm	Optimal	Algorithm	Optimal
facesim	16	16	4.9	4.9
bodytrack	26	26	11.4	11.4
swaptions	33	33	21.9	21.9
ferret	63	63	14.1	14.1
fluidanimate	25	21	12.2	12.7
streamcluster	25	17	4.0	4.2
canneal	9	41	2.9	3.6
blackscholes	9	33	3.7	4.9

2.4 Summary

This chapter presented a runtime technique, *Thread Reinforcer*, to dynamically determine the suitable number of threads for a multithreaded application for achieving high performance on a multicore system. *Thread Reinforcer* monitors performance limiting factors *degree of parallelism, lock contention, thread migrations* and *context-switches* at runtime using simple utilities available on modern OS for determining appropriate number of threads that are expected to yield the best speedup. Thread Reinforcer identifies optimal

or near optimal number of threads for most of the PARSEC programs studied and as well as for SPEC OMP and PBZIP2 programs.

Table 2.5: Algorithm vs. Optimal (Other programs).

Program	**Number of Threads**		**Speedups**	
	Algorithm	Optimal	Algorithm	Optimal
ammp	24	24	11.8	11.8
art	32	32	8.8	8.8
fma3d	16	20	5.5	5.7
gafort	64	48	9.7	9.8
mgrid	16	16	5.0	5.0
swim	32	24	3.9	4.0
wupwise	24	24	8.6	8.6
pbzip2	24	28	6.7	6.9

Table 2.6: Search Overhead (seconds) for PARSEC programs.

Program	T_{search}	$T_{parallel}$	Percentage
canneal	0.5	131	0.4%
facesim	1.1	186	0.6%
blackscholes	0.5	85	0.6%
streamcluster	3.2	226	1.4%
fluidanimate	1.5	69	2.2%
ferret	1.3	41.9	3.1%
bodytrack	1.6	43.8	3.7%
swaptions	0.9	21.3	4.2%

Table 2.7: Search Overhead (seconds) for Other programs.

Program	T_{search}	$T_{parallel}$	Percentage
ammp	0.9	267.1	0.3%
art	1.3	62.8	2.1%
fma3d	0.7	23	3.0%
gafort	1.6	238.9	0.7%
mgrid	0.7	32.1	2.2%
swim	1.3	302.4	0.4%
wupwise	1.2	162.5	0.7%
pbzip2	1.1	201.3	0.6%

Chapter 3

Selecting System Policies

The performance of a multithreaded program is sensitive to the OS scheduling and memory allocation policies. This is because the interactions between program memory reference behavior and the OS scheduling and memory allocation policies make the performance of a program highly sensitive to small changes in resource usage characteristics of the program. In particular, significant variations in the performance are observed from one execution of a program to the next, even when the program input remains unchanged and no other programs are being run on the system. Even after a program has been *tuned*, it may exhibit significantly different levels of performance from one execution to next. This is demonstrated by the results of the following experiment.

In this experiment, 15 multithreaded programs, including the TATP database program, SPECjbb2005, as well as programs from PARSEC and SPEC OMP suites, on a 24-core Dell PowerEdge R905 server running OpenSolaris. Each program was executed 10 times while no other programs were being run. Programs were executed with OPT threads

where OPT threads is the minimum number of threads that gives best performance on our 24-core machine. Speedup is relative to the serial version of the programs. Performance of TATP is expressed in transactions per second (throughput) and speedup is relative to the single client throughput. Performance of SPECjbb2005 is expressed in 'SPECjbb2005 bops' (throughput) with OPT warehouses. Speedup of SPECjbb2005 is relative to the single warehouse (single warehouse, i.e., 35 threads) throughput. Table 3.1 provides the minimum and maximum execution times observed, the standard deviation (SD) for execution time, the minimum and maximum speedups achieved, and the percentage difference between Max Speedup and Min Speedup (% Diff). Table 3.1 also shows the type of the program -- Memory-intensive (Mem) or CPU-intensive (CPU).

As shown in Table 3.1, most of the programs exhibit significant performance variation. For example, the standard deviation of the performance of streamcluster is 10.2. Since the width of one standard deviation is about 68% in a normal distribution, standard deviation of 10.2 means that there is 32% chance that the performance will lie beyond + or - 4.8% of the mean.

Minimizing performance variation while simultaneously maximizing performance is clearly beneficial. Elimination of performance variation has another advantage. Many optimization techniques for improving performance or optimizing power consumption [3, 4, 5, 6] on multicore machines rely on performance monitoring data. The presence of high variation in performance degrades the accuracy of the information collected and the benefits of the optimization techniques. Moreover, due to high performance variation multiple runs of target programs must be used for collecting average performance values. However, with

Table 3.1: Performance variation of the programs.

Program	Min Time (seconds)	Max Time (seconds)	SD (σ)	Min Speedup	Max Speedup	% Diff	OPT Threads	Type
swim	265.0	324.0	25.1	3.7	4.5	17.8	48	Mem
wupwise	147.2	190.3	17.4	7.3	9.5	23.2	72	Mem
equake	182.5	217.9	12.8	2.5	2.9	13.8	12	Mem
gafort	221.6	256.4	12.0	9.0	10.5	14.3	24	Mem
streamcluster	204.4	225.0	10.2	3.8	4.2	9.5	13	Mem
facesim	172.6	199.3	7.7	4.5	5.1	11.8	17	Mem
mgrid	28.6	35.7	3.4	4.5	5.6	19.6	24	CPU
canneal	93.3	102.0	3.0	4.6	5.1	9.8	43	Mem
x264	54.0	58.1	1.4	7.3	7.9	7.6	64	CPU
fluidanimate	64.4	68.5	1.4	12.2	13.0	6.2	21	Mem
bodytrack	43.4	45.2	0.7	11.0	11.5	4.3	26	CPU
swaptions	21.1	22.5	0.5	20.0	21.4	6.5	33	CPU
ferret	41.4	42.9	0.5	14.7	15.2	3.3	63	CPU
TATP	35836	45976	3358	5.2	6.6	13.6	43	Mem
SPECjbb2005	105154	121177	4741	4.1	4.7	12.8	42	Mem

low performance variation one may collect the same quality information via fewer runs.

To solve the above problem, this dissertation presents a runtime technique called *Thread Tranquilizer*, which simultaneously reduces performance variation and improves performance by adaptively choosing appropriate memory allocation and process scheduling policies according to the important resource usage characteristics of the programs.

3.1 Performance Variation Study

To find the causes of performance variation, a performance variation study of the above 15 multithreaded programs was conducted. The reasons for performance variation of a program depend upon the kind of resources it uses. The benchmark programs that stress mainly CPU and main memory are used in this study. Therefore, OS policies that affect the usage of CPU and memory hierarchy are analyzed. First, a study of how different memory allocation policies along with thread migrations affect the performance of memory-intensive programs is conducted. Next, a study of the effect of CPU scheduling policies on the performance of CPU-intensive programs is conducted. Since this dissertation is conducted using OpenSolaris, it benefits from the rich set of tools to examine and understand the behavior of programs. The *memory placement optimization* feature and *chip multithreading* optimizations allow OpenSolaris to support hardware with asymmetric memory hierarchies, such as cache coherent NUMA systems and systems with chip-level multithreading and multiprocessing. To capture the distance between different CPUs and memories, a new abstraction called "locality group (lgroup)" has been introduced in OpenSolaris. Lgroups are organized into a hierarchy that represents the latency topology of the machine [7]. In the following discussion cache miss-rate refers to the last-level cache miss-rate.

3.1.1 Thread Migrations and Memory Allocation Policies

The OS scheduler migrates threads from one core to another core to balance the load across the cores. Thread migrations are expensive as they cause a thread to pull its working set into cold caches, often at the expense of other threads [7]. Moreover, on NUMA

machines, the negative impact of thread migrations is even higher because of variation in memory-latency. Therefore, the negative impact of thread migrations on performance of memory-intensive applications is even higher.

To understand the impact of thread migrations on our machine, two single threaded micro benchmarks are created -- one program is CPU-intensive as it executes arithmetic operations in a loop and another program is memory-intensive as it creates several large arrays using malloc and then reads and writes to them. These programs were run for the same amount of time (nearly 16 seconds) for 100 times in two different configurations: the program is bound to only one core in the no-migration configuration; and the control of this program is given to the OS scheduler in the allow-migration configuration so that thread migrations under default policies are possible.

DTrace scripts [8, 2] are used to find the number of migrations experienced by these two programs and also to measure the average time a thread takes to migrate. In Figure 3.1, the Table shows the average and standard deviations of the execution time and the average number of migrations per run for these programs. DTrace scripts are used to find that a thread migration takes on average around 100 μs on our machine. Therefore, the migration cost from the OS side is around 420 μs for the CPU-intensive program, but the program experiences the overhead around 8000 μs. However, the system overhead due to thread migrations with the memory-intensive program is around 330 μs, but the memory-intensive program experiences the overhead of around 459000 μs. This experiment clearly shows that the impact of thread migration on the performance and performance variation of memory-intensive program is much higher in comparison to CPU-intensive program.

Config.	**Program**	**Type**	**Time (millisecs)**	**SD**	**#Mig.**
1	CPU-intensive	No Binding	16379	0.04	4.2
2	CPU-intensive	Binding	16371	0.01	**0.0**
3	Memory-intensive	No Binding	16687	0.67	3.3
4	Memory-intensive	Binding	16228	0.17	**0.0**

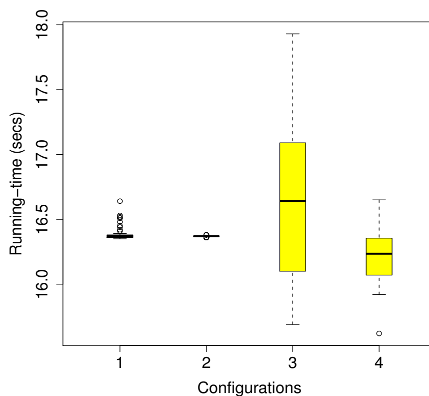


Figure 3.1: The impact of thread migration on CPU-intensive and memory-intensive single threaded micro-benchmarks. Standard deviation (SD) of the running-times and the average number of thread migrations (last column) per run are presented in the table.

Figure 3.1 also shows that the memory-intensive program experiences significantly larger performance variation compared with the CPU-intensive program. To minimize thread migration overhead and preserve locality awareness, Opensolaris tries to migrate the threads among the cores belonging to the same chip. Actually in this experiment, the thread migrations happened among the cores of the same chip. However, when a program is running with number of threads greater than number of cores, for balancing the load across the cores the OS migrates threads from one chip to another chip. This will further increase the migration cost and degrade the speedups.

Next-touch (the default memory allocation policy)

The memory allocation policies significantly affect the impact of thread migrations on performance. The key to delivering performance on a NUMA system is to ensure that physical memory is allocated close to the threads that are expected to access it [7]. The next-touch policy is based on this fact and it is the default policy on OpenSolaris. Thus, memory allocation defaults to the home lgroup of the thread performing the memory allocation. Under next-touch policy a memory-intensive thread can experience high memory latency overhead and high cache miss-rate and most importantly high variance in cache miss-rate when it is started on one core and migrated to another core which is not in its home locality group. This also leads to HyperTransport traffic which degrades performance due to high variation in memory latency of a NUMA system. Moreover, lock-contention, IO, and memory-demanding behavior cause an increase in thread migrations. The thread migrations cause changes in thread priorities which further increases the variation in performance.

Random and round-robin policies

While next-touch is the default memory allocation policy for private memory (heap and stack), the *random* allocation policy is the default policy for shared memory with explicit sharing when the size of shared memory is beyond the default threshold value 8MB [7]. This threshold is set based on the communication characteristics of Message Passing Interface (MPI) programs [7]. Therefore, it is not guaranteed that the random policy will be always applied to the shared memory for multithreaded programs that are based on pthreads and OpenMP. If the shared memory is less than 8MB, then the next-touch policy is the default

also for the shared memory. More importantly, programs with huge private memory (e.g., the heap size for facesim is around 306MB) can dramatically benefit from random/round-robin policies rather than the default next-touch policy. Instead of using the next-touch policy for private memory, random or round-robin (RR) policies are used for both the private and the shared memory for memory-intensive multithreaded programs. Table 3.2 lists these memory allocation policies [7].

Policy	**Description**	**Short name**
LGRP_MEM_POLICY_NEXT	next to allocating thread's home lgroup (next-touch)	NEXT
LGRP_MEM_POLICY_ROUNDROBIN	round robin across all lgroups	RR
LGRP_MEM_POLICY_RANDOM	randomly across all lgroups	RANDOM

Table 3.2: Memory Allocation Policies.

RR policy allocates a page from each leaf lgroup in round robin order. Random memory allocation just picks a random leaf lgroup to allocate memory for each page. Therefore, both RR and Random policies eventually allocate memory across all the leaf lgroups and then the threads of memory intensive workloads get a chance to reuse the data in both private and shared memory. This reduces cache miss rate (i.e. cache misses/instruction) and memory latency penalty. These policies optimize for bandwidth while trying to minimize average latency for the threads accessing it throughout the NUMA system [7]. They spread the memory across as many memory banks as possible, distributing the load across

many memory controllers and bus interfaces, thereby preventing any single component from becoming a performance-limiting hot spot. Moreover, random placement improves the reproducibility of performance measurements by ensuring that relative locality of threads and memory remains roughly constant across multiple runs of an application [7]. Therefore, RR or Random policies minimize cache miss-rate and more importantly *variation in cache miss-rate*.

3.1.2 Dynamic Priorities and Involuntary Context-switches

The main goal of a modern general-purpose OS scheduler is to provide *fairness*. Since, it is not guaranteed that all the threads of a multithreaded program behave similarly at any moment (e.g., due to differences in accessing resources such as CPU, Memory, Locks, Disk), the OS scheduler makes frequent changes to thread priorities to maintain an even distribution of processor resources among the threads. By default, the OS scheduler prioritizes and runs threads on a time-shared basis as implemented by the the default Time Share (TS) Scheduler Class. The adjustments in priorities are made based on the time a thread spends waiting for or consuming processor resources and the thread's time quantum varies according to its priority.

Thread priorities can change as a result of event-driven or time-interval-driven events. Event-driven changes are asynchronous in nature; they include state transitions as a result of a blocking system call, a wakeup from sleep, a preemption, etc. Preemption and expiration of the allotted time-quantum produces involuntary thread context-switches (ICX). Here changing priorities means updating priority of threads based on their CPU

usage and moving them from one priority-class queue to another priority-class queue according to their updated priority. If multiple threads have their priorities updated to the same value, the system implicitly favors the one that is updated first since it winds up being ahead in the run queue. To avoid this unfairness, the traversal of threads in the run queue starts at the list indicated by a marker. When threads in more than one list have their priorities updated, the marker is moved. Thus the order in which threads are placed in the run queue of a core the next time thread priority update function is called is altered and fairness over the long run is preserved [7].

Since all the threads of an application do not behave similarly at any moment (e.g., due to their CPU usage, lock-contention time, sleep time etc.), the positions of the threads in run queues are different from one run to another run. The frequent changes in thread priorities produces *variation in ICX-rate* and thus, variation in performance. Moreover, ICX often includes lock-holder thread preemptions and increases the frequency of lock-holder preemptions as load increases (i.e., thread count grows). More importantly, whenever a lock-holder thread is preempted, the threads that are spinning for that lock will be blocked, which in turn increases VCX-rate, and leads to poor performance under high loads [7].

Fixed Priority Scheduling

The Fixed Priority (FX) scheduling class [7] attempts to solve the above issue of frequent thread ping-ponging with TS class. Threads execute on a CPU until they block on a system call, are preempted by a higher-priority thread that has become runnable, or have

used up their time quantum. The allotted time quantum varies according to the scheduling class and the priority of the thread. OS maintains time quanta for each scheduling class in an object called a dispatch table. Threads in the fixed priority class are scheduled according to the parameters in a simple fixed-priority dispatcher parameter table. The parameter table contains a global priority level and its corresponding time quantum. Once a process is at a priority level it stays at that level at a fixed time quantum. The time quantum value is only a default or starting value for processes at a particular level, as the time quantum of a fixed priority process can be changed by the user with the priocntl(1) command or the priocntl(2) system call. By providing same priority to all the threads of a multithreaded application, FX class dramatically reduces ICX, completely avoids lock-holder thread preemptions, and thus reduces performance variation. Moreover, unlike TS class, only time-driven tick processing [7] is done for FX class. This reduces dispatcher locks, and minimizes OS intervention. Moreover, FX class reduces ICX-rate, more importantly it reduces *variation in ICX-rate*, and thus, reduces performance variation.

3.1.3 Combination of Memory Allocation and Scheduling Policies

To find the appropriate configuration, the 15 programs are tested in the six configurations shown in Table 3.3. Figure 3.2 shows the running-times and cache miss-rates of facesim program (a memory-intensive program) in 10 runs with the six configurations. As shown by the boxplots of Figure 3.2, there is a reduction in the cache miss-rate and also reduction in the variation of cache miss-rate with the combination of Random (or RR) and FX policies. Therefore, as shown in Figure 3.2, the reduction in the variation of cache

Table 3.3: Configurations

No.	Configuration
1	(NEXT + TS)
2	(RANDOM + TS)
3	(RR + TS)
4	(NEXT + FX)
5	(RANDOM + FX)
6	(RR + FX)

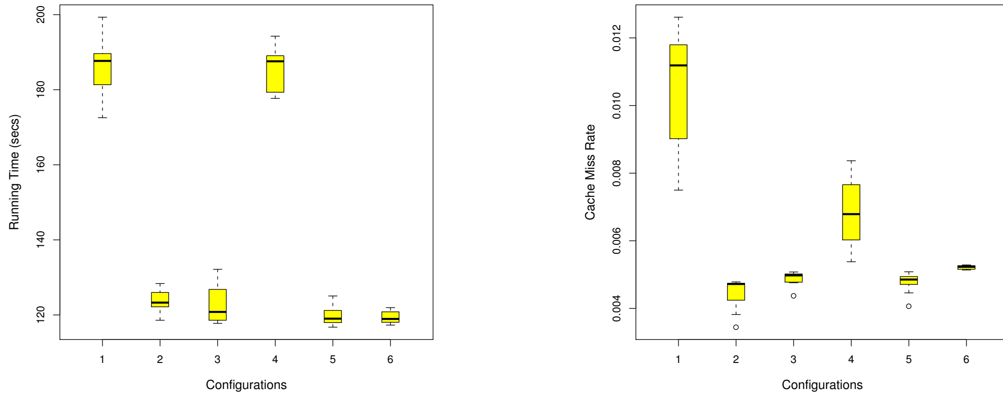


Figure 3.2: Running-times and cache miss-rates of facesim (memory-intensive program) in 10 runs. Table 3.3 lists the configurations.

miss-rate reduced the performance variation. This clearly shows that threads reuse the data from private memory which is spread across the nodes by the RR or Random policy. There is also significant improvement in the performance. Therefore, memory-intensive programs experience low performance variation with improved performance using Random or RR memory allocation policies.

Figure 3.3 shows the running-times and ICX-rates of mgrid for 10 runs. Mgrid is a

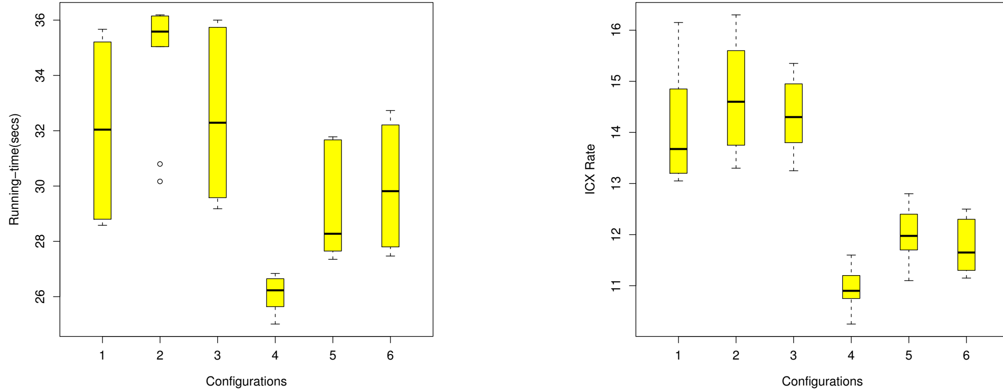


Figure 3.3: Running-times and ICX Rates of mgrid (CPU-intensive program) in 10 runs. Table 3.3 lists the configurations.

CPU-intensive program and scales well in comparison to facesim. As shown by the boxplots of Figure 3.3, with FX scheduling policy the variation in the ICX-rate is reduced and thus there is a reduction in the variation of running-times. Moreover, there is no significant impact of Random and RR policies on the performance of mgrid.

Figures 3.4 and 3.5 show that the combination of Random and FX policies reduces performance variation and improves performance simultaneously for memory-intensive programs. Figure 3.6 shows that FX scheduling policy reduces performance variation and improves performance simultaneously for CPU-intensive programs. There is no significant impact of memory allocation policies on CPU-intensive programs. FX is very effective for programs with high lock-contention. Since swaptions, x264, and ferret are CPU-intensive have low lock-contention, FX slightly improves their performance. However, the performance variation with (FX + Next) is low compared to (TS + Next) for these three programs. Since bodytrack is a CPU-intensive and high lock-contention, the variation with

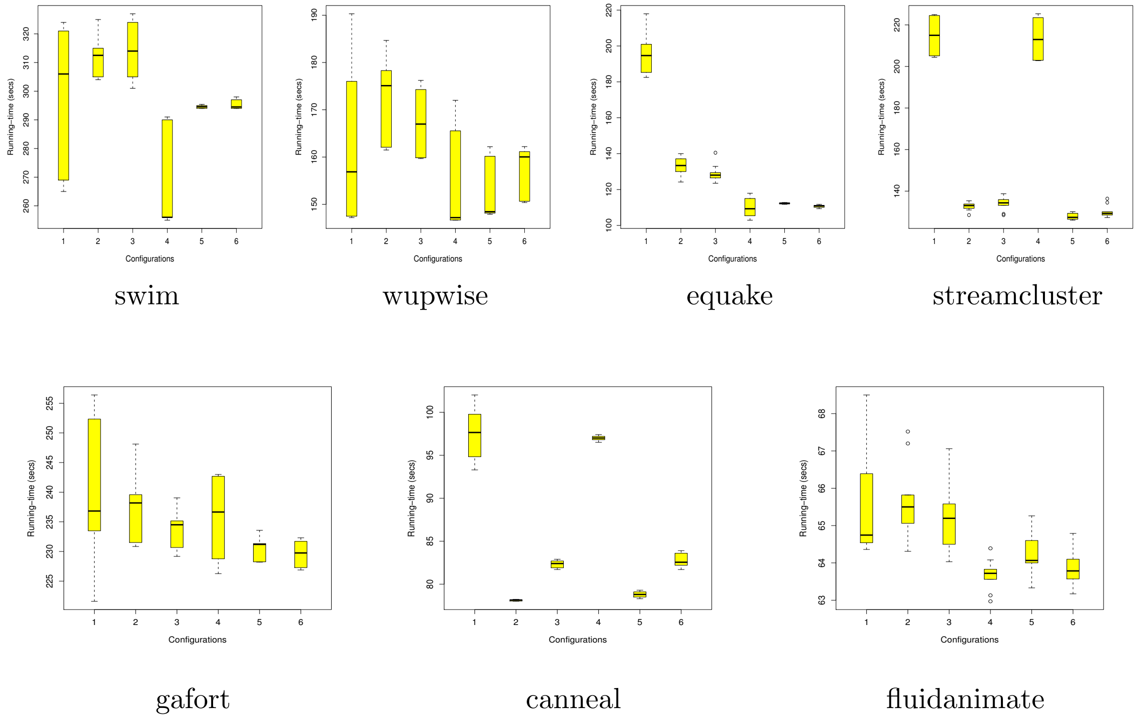


Figure 3.4: Performance variation of memory-intensive programs is reduced with the combination of Random memory allocation and FX scheduling policies.

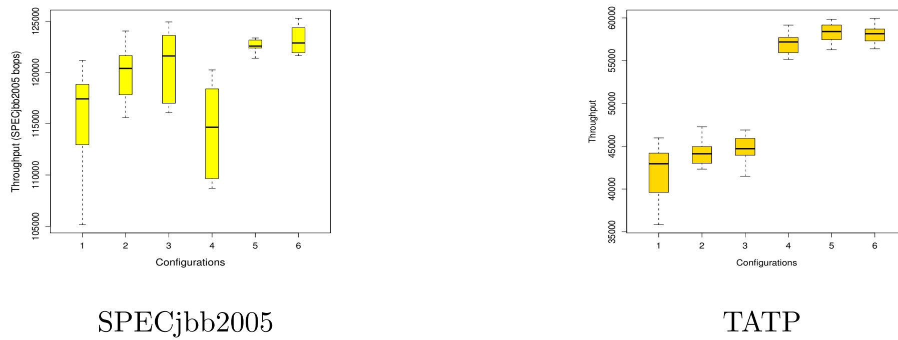


Figure 3.5: Performance variation of SPECjbb2005 and TATP is reduced with the combination of Random and FX policies. Performance (**throughput**) is also improved.

(FX + Next) is significantly lower compared to (TS + Next). Moreover, among the five CPU-intensive programs, mgrid benefits significantly from FX scheduling policy. This is be-

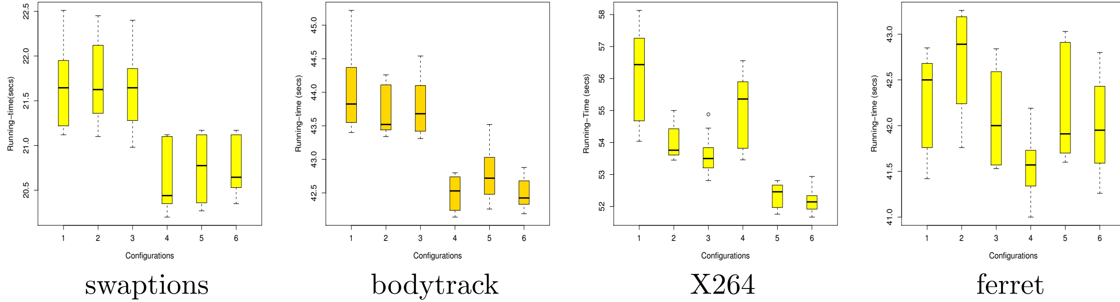


Figure 3.6: Performance variation of CPU-intensive programs is reduced with FX scheduling policy. There is no significant effect of Random or RR policies on CPU-intensive programs.

cause the tuning techniques libmtmalloc and larger page already improved the performance and reduced the performance variation significantly for the other four CPU-intensive programs (swaptions, bodytrack, ferret, and x264). From the above experiments, it is clear that memory-intensive programs get benefit from the combination of Random memory allocation and FX scheduling policies and CPU-intensive programs benefit significantly only from the FX scheduling policy.

We observe that the variation in cache miss-rate causes performance variation in memory-intensive programs and the variation in ICX-rate causes performance variation in CPU-intensive programs. based upon this observation, in the next section, a framework is presented that monitors cache miss-rate and ICX-rate of the target program and dynamically applies proper memory allocation and scheduling policies.

3.2 The Thread Tranquilizer Framework

Thread Tranquilizer monitors the cache miss-rate and thread ICX-rate of a running program and based on their variation, it dynamically applies appropriate memory allocation and scheduling policies. The execution of the target program is begun with the default Next-Touch and TS policies and the program's miss-rate and ICX-rate is monitored once the worker threads have been created. Thread Tranquilizer takes a maximum of five seconds to complete one pass to select appropriate memory allocation and scheduling policies according to the phase changes of the programs. Therefore, programs with very short running times will not benefit from Thread Tranquilizer. Thus, in this dissertation, Thread Tranquilizer is evaluated with the programs where their worker threads run for more than five seconds.

Miss-rate is measured by using cputrack(1) utility and ICX-rate is measured by using mpstat(1) utility. The minimum timeout value with the default mpstat(1) utility is one second. However, this utility is modified to allow time intervals with millisecond resolution to measure the ICX-rate with 100 ms interval. Therefore, using cputrack(1) utility and the modified mpstat utility, 10 samples of miss-rate and ICX-rate are collected with 100 ms interval, then derive a profile data structure from the 10 samples, which contains average miss-rate, average ICX-rate, and standard deviations of miss-rate and ICX-rate.

Figure 3.7 shows the state-transition diagram for one pass of Thread Tranquilizer, i.e. in a time-interval of five seconds. If the average miss-rate is greater than the miss-rate threshold, then the program is treated as a *memory-intensive* and Random memory allocation policy is applied through pmadvise(1) [7] utility with proper advice options. Alternatively, the kernel debugger mdb [7] utility also can be used. FX policy is also

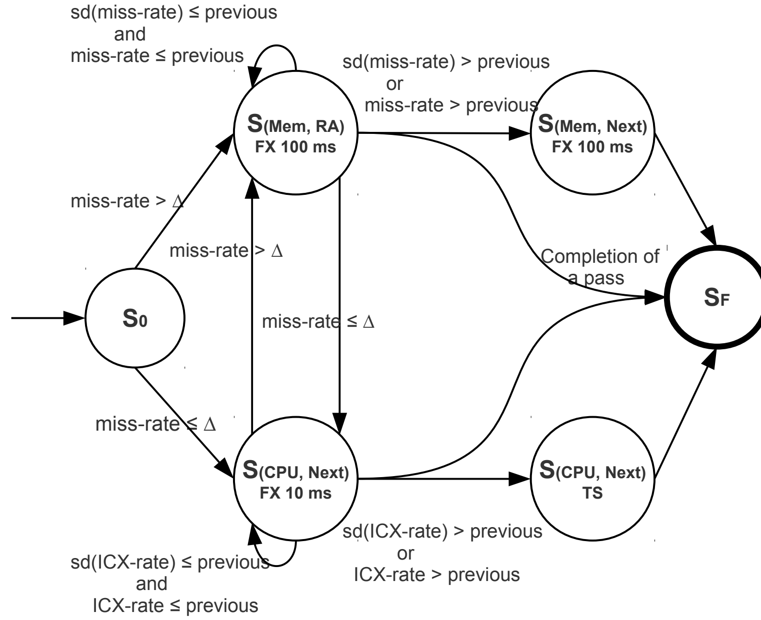


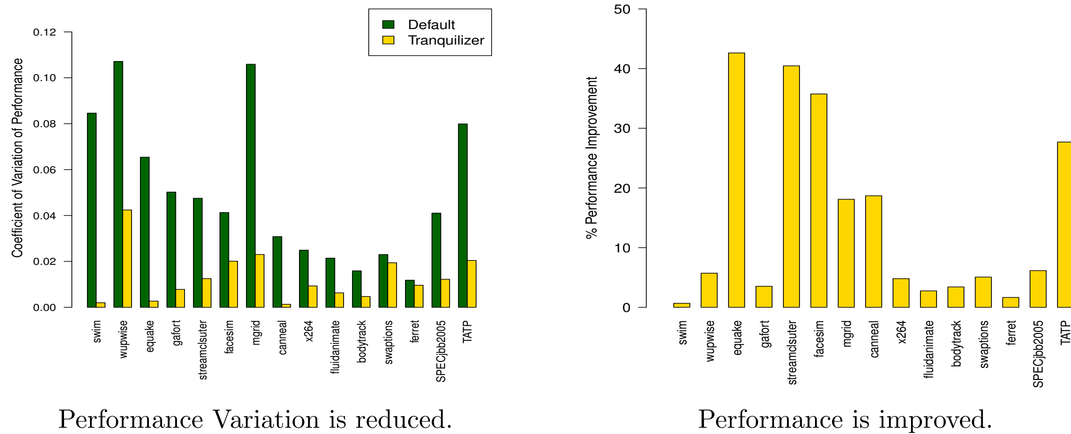
Figure 3.7: State-transition diagram shows one pass of Thread Tranquilizer.

applied with 100 ms time-quantum using priocntl(1) utility. To see the effectiveness of these new policies, a new profile is collected with 10 samples of program's miss-rate and ICX-rate. Since the goal is to reduce performance variation without reducing performance, the average miss-rate is also considered along with the standard deviation of miss-rate. If the average miss-rate is less than the previous average miss-rate and the standard deviation of miss-rate is less than the standard deviation of previous miss-rate, then the program continues running with the new policies. Otherwise, the default Next-Touch memory allocation policy and FX scheduling policy with 100 ms time-quantum are applied. Therefore, Next-Touch vs Random policies are decided for each allocation based on the size of the shared memory requested by the programs.

If the program is *CPU-intensive* (i.e., average miss-rate is less than the Δ miss-rate) then FX scheduling policy with 20 ms time-quantum is applied. To see the effectiveness of

the FX policy, average ICX-Rate and standard deviation of ICX-rate are again collected. If the average ICX-rate is less than the previous average ICX-rate and the standard deviation of ICX-rate is less than the previous standard deviation of ICX-rate, then the target program continues running with the FX policy until the completion of the pass. Otherwise, the default TS scheduling policy is applied. Thread Tranquilizer uses a daemon thread to continuously monitor the target program and to deal with its phase changes. Every five seconds, a timer sends a signal and the daemon thread catches the signal and repeats the above process to effectively deal with the phase changes of the target program.

Thus, Thread Tranquilizer monitors the target program's miss-rate and ICX-rate online, according to these events it dynamically applies appropriate memory-allocation and scheduling policies, and simultaneously reduces performance variation and improves performance.



Performance Variation is reduced. Performance is improved.

Figure 3.8: Performance variation is reduced and performance is improved with Thread Tranquilizer. The bar plot shows another view of the reduction in performance variation (*coefficient of variation*) with Thread Tranquilizer.

3.3 Evaluating Thread Tranquilizer

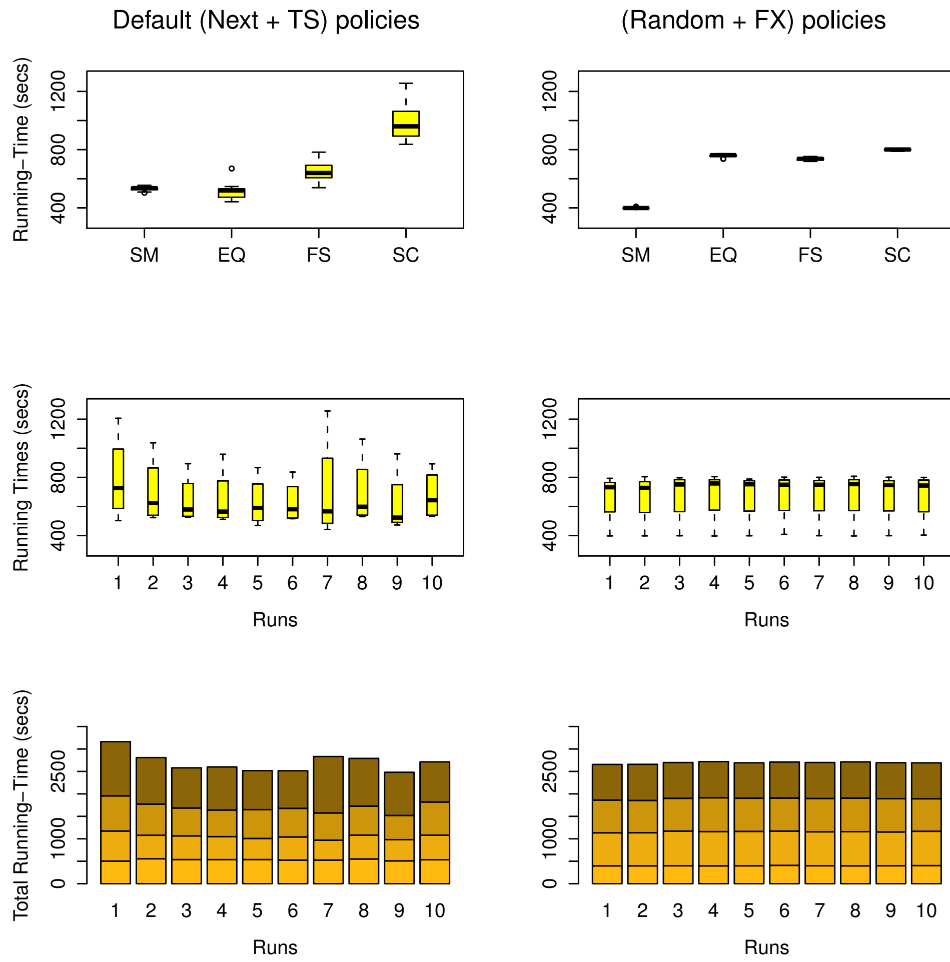
3.3.1 Improved Performance and Reduced Performance Variation

Thread Tranquilizer is evaluated with the 15 programs used for the above performance variation study on our 24-core machine running Solaris. The Memory Placement Optimization feature and Chip Multithreading optimization allow Solaris OS to effectively support hardware with asymmetric memory hierarchies such as NUMA [7]. Specifically Solaris kernel is aware of the latency topology (through lgroups) of the hardware to make optimal decisions on scheduling and resource allocation. Moreover, Solaris provides a rich user interface to modify process scheduling and memory allocation policies. It also provides several effective low-overhead observability tools including DTrace, a dynamic kernel tracing framework [8].

Figure 3.8 and Table 3.4 show that performance variation (coefficient of variation) is reduced and performance is improved simultaneously. As shown in Figure 3.8, the combination of Random and FX policies has significant impact on the performance variation of the programs-- memory-intensive programs benefit from the combination of Random and FX policies and CPU-intensive programs from only the FX scheduling policy. Though there is no significant performance improvement for the swim program, the performance variation is reduced dramatically with Thread Tranquilizer. As shown in Table 3.4, the performance variation is reduced significantly upto 98% (on average 68%) and also performance is improved upto 43% (on average 15%) with the Framework.

Table 3.4: Thread Tranquilizer improves performance and reduces performance variation simultaneously by applying the combination of Random and FX policies. Standard deviation values are used to allow the readers to easily map the boxplots (length of the boxplot) with the standard deviation values.

No.	**Program**	**Default**			**Thread Tranquilizer**		
		Avg. Time	**Speedup**	**SD (σ)**	**Avg. Time**	**Speedup**	**SD (σ)**
1	*swim*	296.6	4.0	**25.1**	294.6	4.1	**0.6**
2	*wupwise*	162.5	8.6	**17.4**	154.2	9.1	**6.5**
3	*equake*	195.7	2.7	**12.8**	112.3	4.8	**0.3**
4	*gafort*	238.9	9.7	**12.0**	230.5	10.1	**1.8**
5	*streamcluster*	214.8	4.0	**10.2**	127.9	6.7	**1.6**
6	*facesim*	186.3	4.8	**7.7**	121.7	7.3	**2.4**
7	*mgrid*	32.1	5.0	**3.4**	26.1	6.1	**0.6**
8	*canneal*	97.4	4.9	**3.0**	79.2	6.0	**0.1**
9	*x264*	56.2	7.6	**1.4**	53.5	8.0	**0.5**
10	*fluidanimate*	65.5	12.8	**1.4**	64.1	13.1	**0.4**
11	*bodytrack*	44.0	11.3	**0.7**	42.5	11.7	**0.2**
12	*swaptions*	21.7	20.8	**0.5**	21.2	21.4	**0.4**
13	*ferret*	42.3	14.9	**0.5**	41.8	15.1	**0.4**
14	*TATP*	42009	6.0	3358	58110	8.4	1186
15	*SPECjbb*	115650	4.5	4741	122762	4.8	1502



!t

Figure 3.9: Thread Tranquilizer is very effective against parallel runs of more than one application.

3.3.2 Improved Fairness and Effectiveness Under High Loads

The combination of Random memory allocation and FX scheduling policies improves fairness in scheduling when there is more than one application running. For showing this, four programs swim (SM), equake (EQ), facesim (FS), and streamcluster (SC) are run for 10 times simultaneously with the configurations of (Next + TS) and (Random + FX) policies. Figure 3.9 shows that the (Random + FX) combination not only reduces the performance variation of individual multithreaded programs, it also reduces the performance variation of the total running-times of the multithreaded programs in concurrent runs. The figures in the first row (of Figure 3.9) show the running-times of individual programs in 10 runs, the figures in the second row show the running-times of the four programs in individual runs, and the figures in the third row show the total running-times of the four programs in each run. That is, in each run, under (Random + FX) configuration, OS allocates resources fairly to all the four programs, and also the total running-time (throughput) is improved. Therefore, the combination of Random and FX policies provide *fairness* relative to the default policies of OpenSolaris. Moreover, as shown in Fig. 3.9, the combination of FX and Random policies is very effective under heavy loads. There are a total of 90 threads from the four multithreaded programs running on 24 cores, i.e., over 375% load. Thus, Thread Tranquilizer is also effective when there is more than one application running on the system.

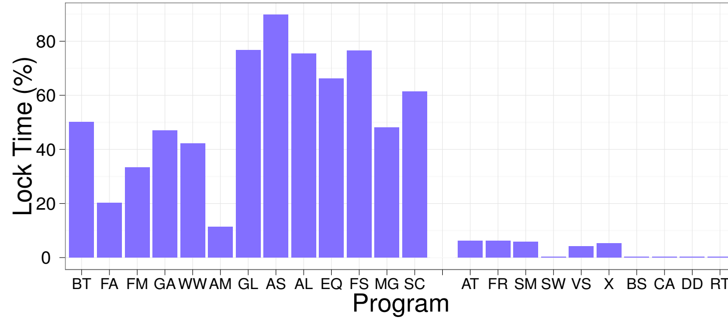
3.4 Summary

This chapter presented a runtime technique, *Thread Tranquilizer*, to simultaneously reduce performance variation and improve performance of a multithreaded program running on a multicore system. *Thread Tranquilizer* continuously monitors last-level cache miss-rate and involuntary context-switch rate of a multithreaded program using simple utilities available on a modern OS. Based on these, it adaptively chooses appropriate memory allocation and process scheduling policies and simultaneously reduces performance variation and improves performance. Thread Tranquilizer yields up to 98% (average 68%) reduction in performance variation and up to 43% (average 15%) improvement in performance over default policies of OpenSolaris.

Chapter 4

Reducing Lock Acquisition Overhead

The performance of a multithreaded program is often impacted greatly by lock contention on a ccNUMA multicore system. Figure 4.1 presents the lock times of programs from PARSEC and SPEC OMP benchmark suites. Here, the lock time is defined as the percentage of elapsed time a program spends on performing operations on user-space locks. As shown in Figure 4.1, the first 13 programs out of a total of 23 programs, exhibit very high lock times when they are run with 64 threads on a 64-core machine (i.e, four 16-core CPUs). This is because of the performance of a multithreaded program is highly sensitive to the distribution of threads across the multiple multicore CPUs on a ccNUMA multicore systems. In particular, when multiple threads compete to acquire a lock, due to the NUMA nature of the architecture, the time spent on *acquiring locks* by threads distributed across different CPUs is greatly increased.



Lock times.

PARSEC: blackscholes (BS); bodytrack (BT); canneal (CA); dedup (DD); fluidanimate (FA); facesim (FS); ferret (FR); raytrace (RT); streamcluster (SC); swaptions (SW); vips (VS); x264 (X); **SPEC OMP:** applu (AL); ammp (AM); art (AT); apsi (AS); equake (EQ); fma3d (FM); gafort (GA); galgel (GL); mgrid (MG); swim (SM); wupwise (WW)

Figure 4.1: Lock times of PARSEC and SPEC OMP programs when 64 threads are run on 64 cores spread across 4 CPUs.

To address the above problem, this dissertation presents a runtime technique, *Thread Shuffling*. *Thread Shuffling* continuously monitors lock times of individual threads of a multithreaded program for adapting the location of threads across the CPUs of a ccNUMA machine to reduce the time spent by the threads in *acquiring locks*.

4.1 Performance Degradation due to Locks and ccNUMA

Although CPU schedulers of modern operating systems, such as Solaris and Linux, are effective in scheduling multiple single threaded programs, they fail to effectively schedule threads of a multithreaded program on a ccNUMA system. This is because they do not

distinguish between threads representing single threaded applications from threads of a single multithreaded application. As a result, the decisions made by the schedulers are oblivious to the lock contention among the threads of a multithreaded program. Decisions of scheduling threads across different CPUs that ignore lock contention can lead to significant performance degradation.

Lock contention results when multiple threads compete to acquire the same lock. This situation arises often in programs where threads must frequently synchronize with each other. For example, barrier synchronization is commonly used to synchronize threads in parallel applications that exploit fork join parallelism. The implementation of the barrier requires all threads to acquire a lock before they can increment a counter to indicate their arrival at the barrier. This section first describes the cause of performance degradation during barrier synchronization on a ccNUMA system. Next the idea of *thread shuffling* for ameliorating the impact of NUMA nature of the system on cost of barrier synchronization is introduced.

Let us consider the performance behavior of barrier synchronization time on the 64 core ccNUMA machine. The description of the machine and the costs of barrier synchronization are presented in Figure 4.2. Barrier synchronization time is measured for a *pthreads* based implementation that uses a *mutex* variable and a *condition variable*. The threads indicate their arrival by executing a critical section, guarded by mutex, which increments a counter. The threads that arrive at the barrier early, wait on the condition variable. The last arriving thread wakes up the waiting threads by performing a broadcast on the conditional variable. The above program is run for varying number of threads (16,

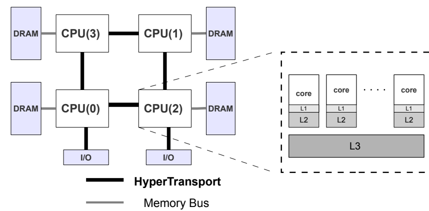
Supermicro 64-core server:

4 × 16-Core 64-bit AMD Opteron^{TM} 6272 Processors (2.1 GHz);

L1/L2: 48 KB / 1000 KB; Private to a core;

L3: 16 MB shared by 16 cores; RAM: 64 GB;

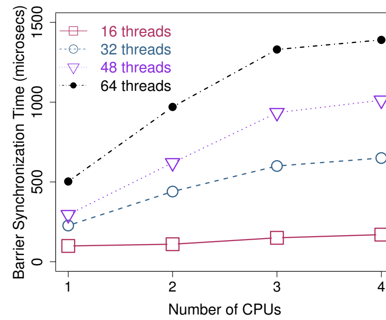
Operating System: Oracle Solaris 11^{TM}



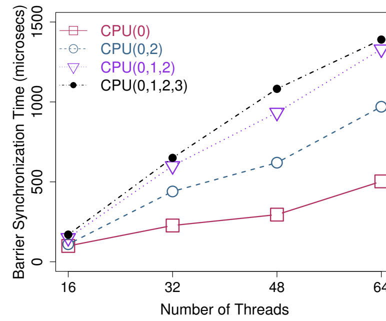
Relative CPU Communication Latencies

Same CPU	Neighboring CPUs	Furthest CPUs
1.0	1.6	2.2

(a)



(b)



(c)

Figure 4.2: (a) ccNUMA machine; (b) Barrier execution times for varying number of CPUs; and (c) Barrier execution times with varying number of threads.

32, 48, and 64) – fewer than 16 threads are not used because each CPU on our system has 16 cores and thus it is interesting to run an application on multiple CPUs when there are sufficient threads. The threads were run on the following four configurations: CPU(0) - where all threads are executed on a single CPU, i.e. CPU(0); CPU(0,2) - where the threads are distributed equally among two neighboring CPUs, i.e. CPU(0) and CPU(2); CPU(0,1,2) - where the threads are distributed equally across 3 CPUs; and CPU(0,1,2,3) - where the threads are distributed equally among all four CPUs. The above distribution of threads across the CPUs is enforced through creation of appropriate *processor sets* (pools of cores) and assigning subsets of threads to them. The Solaris OS guarantees that threads are never migrated across different processor sets. From the two plots in Figure 4.2 the following observations can be made:

- (CPU scaling effect.) From the first plot in Figure 4.2(b) it can be seen that, for a given number of threads, the barrier synchronization time *increases with the number of CPUs* used. That is, the barrier synchronization times are ordered as follows: CPU(0) $<$ CPU(0,2) $<$ CPU(0,1,2) $<$ CPU(0,1,2,3). This is direct consequence of the NUMA nature of the machine.

- (Thread scaling effect.) From the second plot in Figure 4.2(c) it can be seen that, when using a given number of CPUs, as expected barrier synchronization time increases with the number of threads. However, due to the NUMA nature of the machine, barrier synchronization time increases *more rapidly* for configurations that involve use of larger number of CPUs.

The Cause. Let us consider how the time spent on barrier synchronization is impacted by the NUMA nature of the machine. This time is impacted by the accesses to shared memory locations by all the synchronizing threads. In particular, let us consider the location that represents the `lock` variable. A barrier synchronization involves series of *successful* `(Acquire; Release)` operation pairs that are performed by all the participating threads such that each thread performs one successful operation pair. If $T(Acq_i; Rel_i)$ denotes the time for the ith successful operation pair, and there are a total of n threads synchronizing at the barrier, then the total time is given by $\sum_{i=1}^{n} T(Acq_i; Rel_i)$. On a ccNUMA machine like the one shown in Figure 4.2, the time for each successful `(Acquire; Release)` can vary significantly. Let us consider two consecutive operations $(Acq_i; Rel_i)$ and $(Acq_{i+1}; Rel_{i+1})$ such that they are performed by two threads located at CPUs CPU_i and CPU_{i+1} respectively. The time for the second operation pair, i.e. $T(Acq_{i+1}; Rel_{i+1})$ can vary as follows:

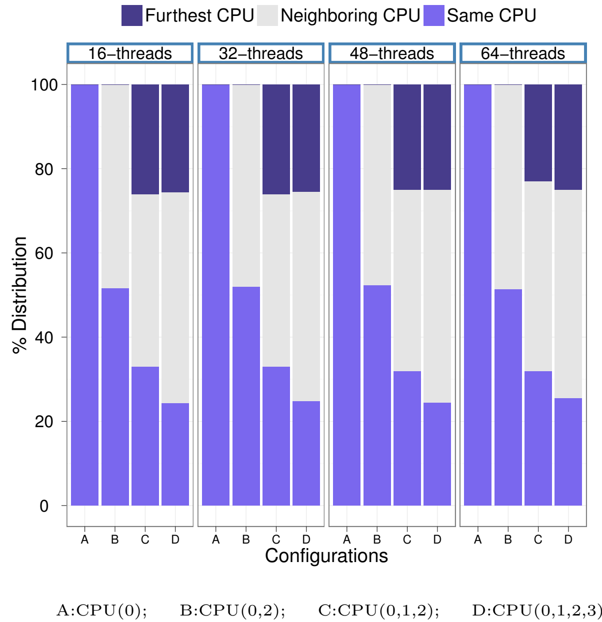
- $T(Acq_{i+1}; Rel_{i+1})$ is the *minimum* (T_{min}) when the threads that perform operations $(Acq_i; Rel_i)$ and $(Acq_{i+1}; Rel_{i+1})$ are located on the same CPU, i.e. $CPU_i = CPU_{i+1}$;

- $T(Acq_{i+1}; Rel_{i+1})$ is *intermediate* (T_{int}) when the threads that perform operations $(Acq_i; Rel_i)$ and $(Acq_{i+1}; Rel_{i+1})$ are located on neighboring CPUs, e.g., $CPU_i =$CPU(0) and $CPU_{i+1} =$CPU(2); and

- $T(Acq_{i+1}; Rel_{i+1})$ is the *maximum* (T_{max}) when the threads that perform operations $(Acq_i; Rel_i)$ and $(Acq_{i+1}; Rel_{i+1})$ are located on CPUs that are furthest from each other, e.g. $CPU_i =$CPU(0) and $CPU_{i+1} =$CPU(1).

The reason for the above variation is due to the NUMA nature of the ccNUMA machine. This machine uses the MOESI cache coherency protocol [9]. In this protocol a cache line can be in one of five states: Modified, Owned, Exclusive, Shared, and Invalid. Therefore, following the execution of $(Acq_i; Rel_i)$, the cache line containing the lock is in *Modified* state. If the CPU_{i+1} is different from CPU_i, then the cache line must be transferred from CPU_i to CPU_{i+1} when $(Acq_{i+1}; Rel_{i+1})$ is executed. The time for this transfer varies with the relative communication latencies between CPUs as shown in Figure 4.2(a) (i.e., 1.0 vs 1.6 vs 2.2).

How often successful lock acquire events entail an overhead of T_{min}, T_{int}, and T_{max} are collected to confirm that the cause of the impact of NUMA on barrier synchronization time. For a given number of threads, in each of the four configurations used, the total number of *successful* $(Acquire; Release)$ operations performed is the same. However, how often these operations require T_{min}, T_{int}, and T_{max} overhead varies across the four configurations. Figure 4.3 provides this distribution across the three types of events: *Same CPU* (T_{min}); *Neighboring CPU* (T_{int}); and *Furthest CPU* (T_{max}). As shown in Figure 4.3, irrespective of the number of threads involved, in configuration CPU(0) as expected 100% of the time the cost is T_{min}, in configuration CPU(0,2) around 50% of the time the cost is T_{min}, in configuration CPU(0,1,2) around 35% of the time the cost is T_{min}, and in configuration CPU(0,1,2,3) around 25% of the time the cost is T_{min}. This explains why barrier synchronization time is the least for CPU(0) configuration and the most for configuration CPU(0,1,2,3). For configurations CPU(0,1,2) and CPU(0,1,2,3) around 25% of the time the cost is T_{max}.



A:CPU(0); B:CPU(0,2); C:CPU(0,1,2); D:CPU(0,1,2,3)

Figure 4.3: (Barrier) The distribution of lock transfers for successful $(Acquire; Release)$ operations.

Given the above observations, in the next section, a technique is proposed for *reducing the frequency of lock transfers among CPUs.*

4.2 Thread Shuffling

In this section a technique called *thread shuffling* is proposed. It reduces the frequency of lock transfers among CPUs by shuffling threads between CPUs so that threads seeking locks are more likely to find the locks on the same CPU. There are two expected consequences of thread shuffling. First, the performance of an application is expected to improve. Second, the performance of the application will become less sensitive to the CPU configuration being used by the application. In this description the basic ideas and ideal functioning of thread shuffling algorithm are presented. Practical implementation of this

algorithm will be presented in the next section.

The thread shuffling technique observes the behavior of threads with respect to acquiring locks and uses this information to identify *contending threads* and migrate them closer to each other. That is, the number of CPUs across which the contending threads are distributed is reduced and thus the frequency of lock transfers between CPUs caused by the identified contending threads is reduced. This technique is aimed at countering the *CPU scaling effect*, i.e. spreading of *contending threads across more CPUs degrades performance*. Its scope extends across multiple thread synchronization episodes – by observing behavior during past synchronization episodes, the behavior during future synchronization episodes is optimized. Thread shuffling migrates threads by swapping a pair of threads between two CPUs so that the number of threads on the CPUs remains the same. This is so the *load balance* across the CPUs is maintained.

Thread shuffling is based upon the premise that threads in an application can be divided into different groups such that shuffling to bring threads from the same group close to each other will reduce lock transfers. This situation may arise if each group represents subset of threads contending for a distinct lock. A more common situation is where all threads are contending for the same lock; however, due to different loads, one group of threads contends for the lock earlier than the other group. Thus, each of these groups benefits from thread shuffling.

Ideal functioning of thread shuffling is illustrated in Figure 4.4. In this example there are two groups of 8 threads – the slow (S) group and the fast (F) group such that the members of the fast group tend to arrive at the barrier before the members of the slow

	CPU(0)	CPU(2)	CPU(1)	CPU(3)
1	$T_0^S\ T_1^S\ \underline{T_0^F}\ T_1^F$	$T_4^S\ T_5^S\ T_4^F\ T_5^F$	$\underline{T_2^S}\ T_3^S\ T_2^F\ T_3^F$	$T_6^S\ T_7^S\ T_6^F\ T_7^F$
2	$T_0^S\ T_1^S\ T_2^S\ \underline{T_1^F}$	$T_4^S\ T_5^S\ T_4^F\ T_5^F$	$T_0^F\ \underline{T_3^S}\ T_2^F\ T_3^F$	$T_6^S\ T_7^S\ T_6^F\ T_7^F$
3	$T_0^S\ T_1^S\ T_2^S\ T_3^S$	$T_4^S\ T_5^S\ \underline{T_4^F}\ T_5^F$	$T_0^F\ T_1^F\ T_2^F\ T_3^F$	$\underline{T_6^S}\ T_7^S\ T_6^F\ T_7^F$
4	$T_0^S\ T_1^S\ T_2^S\ T_3^S$	$T_4^S\ T_5^S\ T_6^S\ \underline{T_5^F}$	$T_0^F\ T_1^F\ T_2^F\ T_3^F$	$T_4^F\ \underline{T_7^S}\ T_6^F\ T_7^F$
5	$T_0^S\ T_1^S\ T_2^S\ T_3^S$	$T_4^S\ T_5^S\ T_6^S\ T_7^S$	$T_0^F\ T_1^F\ T_2^F\ T_3^F$	$T_4^F\ T_5^F\ T_6^F\ T_7^F$

Figure 4.4: Illustration of Thread Shuffling.

group. Further, the threads from each group are initially distributed equally across the four CPUs. Over a period of time threads are shuffled as shown bringing threads in the S group to CPU(0) and CPU(2) while bringing the threads from F group to CPU(1) and CPU(3). Please note that in the figure at each step the pair of threads that are about to be swapped are underlined. Following the entire sequence of thread migrations, when threads in any one of the groups contend for the lock, lock transfers only take place between neighboring CPUs. In other words, the most expensive lock transfers between furthest CPUs are avoided.

Several *policy* decisions must be made in developing a practical implementation of thread shuffling. First, it must be decided how identification of thread groups will be carried out so that candidates for shuffling can be selected. Second, the number of threads to be shuffled in each step must be decided – in our illustration a single pair of threads are shuffled in each step but shuffling rate can be accelerated to reach the desired impact

of thread shuffling by shuffling multiple pairs or threads between multiple pairs of CPUs. Finally, shuffling interval must be selected, i.e. duration for which the behavior of threads is observed before performing thread shuffling. All these issues will be addressed in Section 3 where detailed implementation of thread shuffling is presented and evaluated.

Migrating threads between CPUs is preferable to moving locks between CPUs. This is because when threads are contending for locks, they are not doing useful work and hence the thread migration cost is not expected to impact the execution time. On the other hand, lock transfers occur when locks are successfully acquired and thus lock transfer times are on the critical path of execution. Thus, preventing lock transfers between CPUs shortens the critical path and reduces execution time.

Next maximum potential benefit of shuffling in context of barrier synchronization is analyzed. For this purpose, perfect implementations of thread shuffling is assumed. Then it is observed how it reduces the worst case time spent on *lock transfers* during barrier synchronization of N threads that are distributed equally on the 4 CPUs by the operating system thread scheduler to maintain load balance across the CPUs.

– *Without Thread Shuffling.* The worst case situation arises if every consecutive pairs of lock acquires are performed by threads on different CPUs, i.e. a total of $N-1$ lock transfers are performed. Of these, at least one transfer must occur between neighboring CPUs and the rest can occur between furthest CPUs in the worst case. Hence, the worst case total lock transfer time is given by: $(N-2) \times T_{max} + T_{int}$.

– *With Thread Shuffling.* For purpose of this analysis let us assume that the threads are divided into 2 groups – a fast group of $\frac{N}{2}$ threads and a slow group of $\frac{N}{2}$ threads. Let

us assume that initially threads in each group are distributed equally among the 4 CPUs; however, perfect thread shuffling, over a period of time, migrates threads in each group to a different pair of neighboring CPUs. Following shuffling, lock acquire operations by threads within each group requires maximum of $(N/2-1)$ lock transfers and an additional transfer is performed during the transition from the fast to the slow group. Moreover, since all transfers are between neighboring CPUs, the worst case total lock transfer time is reduced to: $(N-1) \times T_{int}$.

Thus, it is observed that thread shuffling has significant potential of reducing lock transfers between the CPUs. Next thread shuffling algorithm and its evaluation are presented. The overview of thread shuffling is provided in Algorithm 2. Thread shuffling is implemented in form of a daemon thread which executes throughout an application's lifetime repeatedly performing the following three steps: monitor threads; form thread groups; and perform thread shuffling. The first step monitors the behavior of threads in terms of the times they spend on lock operations. If this time exceeds a preset threshold (used 5% of execution time as the threshold), thread shuffling is triggered by executing the next two steps. The second step forms groups of similarly behaving threads using the lock times collected during the monitoring step. Finally the third step performs thread shuffling to ensure that threads belonging the same thread group are all moved to the same CPU. Next each of these steps are described in greater detail.

Algorithm 2: Thread Shuffling Daemon.

Input: N: Number of threads; and C: Number of CPUs.

repeat
  I. Monitor Threads – sample lock times of N threads.
  if *lock times exceed threshold* **then**
    II. Form Thread Groups – sort threads according to lock times and divide them into C groups.
    III. Perform Shuffling – shuffle threads to establish newly computed thread groups.
  end
until *application terminates*;

4.2.1 Monitoring Threads

The design of the monitoring component involves two main decisions. First what thread behavior characteristic to monitor must be decided. Since lock contention is considered, the fraction of execution time that each thread spends in the code that performs lock operations is monitored – this is referred to as the *lock time. Threads that experience similar lock times will be placed in the same group* as they are likely to represent threads that contend with each other for locks. The daemon thread maintains per thread data structure that holds the lock time values collected for the thread as well as the id of the CPU on which the thread is running.

Second a monitoring duration (i.e., time interval after which thread groups are formed and thread shuffling is carried out) must be chosen and the mechanism used to carry out monitoring must be selected. The monitoring mechanism used is as follows. The

lock times are sampled at regular intervals over the monitoring duration. The monitoring duration and sampling frequency are chosen to strike a balance between the overhead of thread shuffling and the responsiveness of the thread shuffling algorithm to the changing behavior of application threads. Through experimentation the overhead of our approach is studied and finally selected monitoring duration of 2 seconds (i.e., thread shuffling is performed after 2 seconds of monitoring) and during this duration a total of 10 samples of lock times are taken (i.e., for every 200 milliseconds interval the lock times are collected and per thread data structures are updated). The prstat(1) utility available on Solaris [7, 2, 8] is used to collect lock times for each thread. However, the default prstat(1) utility uses one second as the minimum time interval. This utility was modified to allow enable shorter time intervals and then collected lock times for 200 ms time intervals by using the modified utility.

4.2.2 Forming Thread Groups

Every two seconds the daemon thread examines the profile data collected for the threads, and if the lock times exceed the minimum preset threshold, it constructs thread groups such that one group per CPU is formed. The goal is that the lock time behavior of threads within each group to have low variation, i.e. the threads should be similar. Therefore to form the groups the threads are sorted according to their lock times and then divided into as many groups of consecutive threads as the CPUs being used to run the application. For example, if all CPUs are being used, the first group is assigned to CPU(0), the second to CPU(2), the third to CPU(3), and the fourth to CPU(1). In other words

the thread groups that are the most similar are assigned to neighboring CPUs and thread groups that are most dissimilar are assigned to furthest CPUs. Since the size of each thread group is the same, load balance across the CPUs is maintained.

4.2.3 Performing Thread Shuffling

This step simply affects the thread groups computed for each of the CPUs in the preceding step. The psrset(1) utility is used for binding a group of threads to a set of cores (called a processor-set in Solaris terminology). Thus for every 2 seconds, thread shuffling daemon simultaneously migrates as many threads as needed to realize the new thread groups computed in the previous step. In this step only a subset of threads will be migrated as many threads may already bound to set of cores on the CPUs where they should be. It is also possible that in some programs, over many monitoring durations, the behavior of threads does not change significantly. If this is the case, the thread groups formed will not change, and hence no threads are migrated. Thus, effectively the shuffling step is skipped altogether and monitoring is resumed. In other words, when thread migrations are not expected to yield any benefit, they will not be performed.

4.3 Evaluating Thread Shuffling

Thread shuffling is applied to the programs from the PARSEC and SPEC OMP suites to evaluate its impact on program performance. For comparison, as a baseline, the performance data collected when the programs are executed under the default Solaris 11 thread scheduler is used.

Program	Sequential TTT (secs)	Thread Shuffling			No Shuffling		
		TTT (secs)	Lock Time (secs)	Speedup	TTT (secs)	Lock Time (secs)	Speedup
BT	1848	137	58.9	13.5	154	77.3	12
FA	682	40.3	4.96	16.9	44.4	8.1	15.4
FM	3154	149	33.9	21.2	166	55.1	19
GA	6264	228	89.7	27.5	240	112.5	26.1
WW	1779	114	43.6	15.6	128	53.4	13.9
AM	4070	334	30.1	12.2	342	38.2	11.9
GL	689	230	165.6	3.0	246	188.7	2.8
AS	254	29.8	24.9	8.5	33.1	30.4	7.7
AL	108	16.4	10.8	6.6	19	13.5	5.7
EQ	646	78	43.1	8.3	91	59.9	7.1
FS	1746	192	126.5	9.1	221	168.5	7.9
MG	206	24.2	9.31	8.5	28.6	13.5	7.2
SC	1187	202	114.0	5.9	212	130.2	5.6

Figure 4.5: Performance of Thread Shuffling.

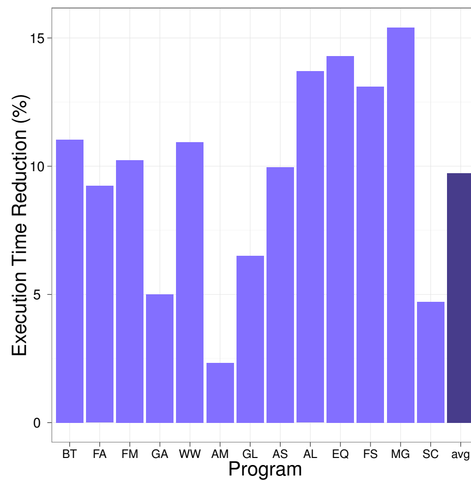
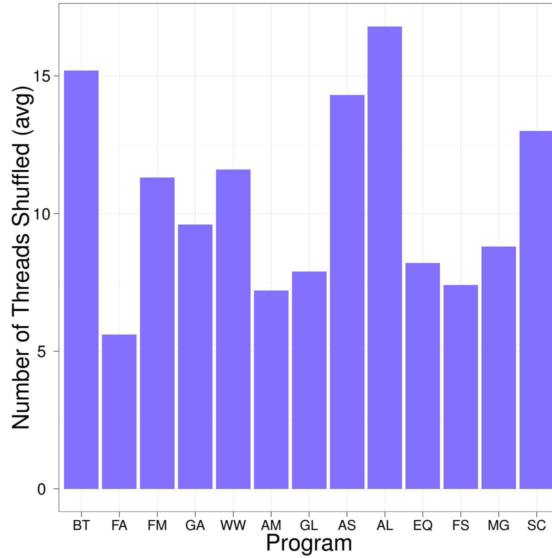


Figure 4.6: Reductions in execution time.

4.3.1 Performance Benefits

The results of executing the programs are summarized in Figure 4.5. To collect this performance data each program is run for 10 times and the data presented is the average over the 10 runs. For each program the *Total Turnaround Times* (TTTs) are presented for: the sequential execution, parallel execution with thread shuffling, and parallel execution without shuffling. The lock times and speedups for both parallel executions are also presented. As shown in Figure 4.5, thread shuffling improves performance: the lock times are reduced, and therefore the TTTs; thus leading to better speedups. The percentage reductions in program execution time achieved by thread shuffling are shown in Figure 4.6. As shown in Figure 4.6, substantial reductions in execution time are observed – for seven programs the reductions in execution time exceed 10%. The average reduction in execution time is 9.7%. For the `mgrid` (MG) program the reduction is the highest – around 16%.

While the above detailed performance data is for the 13 programs with high lock times, thread shuffling is also applied to the remaining 10 programs from PARSEC and SPEC OMP which have low lock times (less than 5% of the execution time). For some of these programs the lock time is low because the serial part of the execution, executed by the main thread, accounts for 90% to 95% of the execution time – the programs are BS, DD, CA, and RT. The purpose of this experiment was to determine if thread shuffling can hurt performance when it is applied to programs where it is not needed. The change in execution time was insignificant, i.e. less than 1% is observed. This is not surprising as, when lock times are small, the cost of thread shuffling is simply the cost of monitoring as no thread migrations are triggered.



Program	$\eta = \Delta TTT/\Delta LT$
BT	0.92 = 17/18.4
FA	1.31 = 4.1/3.14
FM	0.80 = 17/21.2
GA	0.53 = 12/22.8
WW	1.43 = 14/9.8
AM	0.99 = 8/8.1
GL	0.69 = 16/23.1
AS	0.60 = 3.3/5.5
AL	0.96 = 2.6/2.7
EQ	0.77 = 13/16.8
FS	0.69 = 29/42
MG	1.05 = 4.4/4.19
SC	0.62 = 10/16.2

Figure 4.7: The cost and efficiency of thread shuffling.

4.3.2 Cost and Efficiency

Now let us study the cost of thread shuffling. The cost of monitoring thread lock times is very low – around 1% of the CPU utilization can be attributed to the monitoring task. The cost of migrating threads during shuffling is also small. The bar graph in Figure 4.7 shows the average number of threads shuffled in a single thread shuffle operation for each of the programs. This number ranges from a minimum of 5.6 threads for `fluidanimate` (FA) to a maximum of 16.8 threads for `applu` (AL). Across all the programs, the average is 10.5 threads being migrated during each thread shuffling operation. Since the total number of threads is 64, this represents around 17% of the threads. The system call for changing the binding of a single thread from cores in one CPU to cores in another CPU is around 29

microseconds. Therefore every 2 seconds thread shuffling spends around 305 microseconds (29×10.5) on changing the binding of migrated threads. Thus, this represents 0.00015% of the execution time.

Since the cost of monitoring and changing binding of threads is very small, the impact on performance can be due to other factors related caused by migrations (e.g., impact of data locality). Therefore next the *efficiency* (η) of thread shuffling is analyzed, i.e. the degree to which the reductions in lock times are passed on as reductions in TTTs. The efficiency η is defined as follows: $\eta = \frac{\Delta TTT}{\Delta LT}$, where ΔTTT and ΔLT are the reductions in total turnaround time and lock times achieved by thread shuffling respectively. Therefore if the value of is η is 1.0, it means that all the reductions achieved in lock time fully translate into reductions in TTT. The values of η are presented in a table in Figure 4.7. The value of η across the programs ranges from a minimum of 0.53 to a maximum of 1.43. The average value of η across all programs is 0.87, i.e. thread shuffling has 87% efficiency. Therefore it is clear that thread shuffling is effective and the cost of thread migrations performed is well justified by the net performance benefits.

4.3.3 Time Varying Behavior

Finally the time varying behavior of lock times and thread shuffling are studied as observed throughout the executions of the programs. In Figure 4.8 the cumulative lock times of each program over its entire run are shown for both with and without thread shuffling. As shown in Figure 4.8, the lines representing the cumulative lock times consistently move further apart as execution proceeds. In other words, during no period of time thread

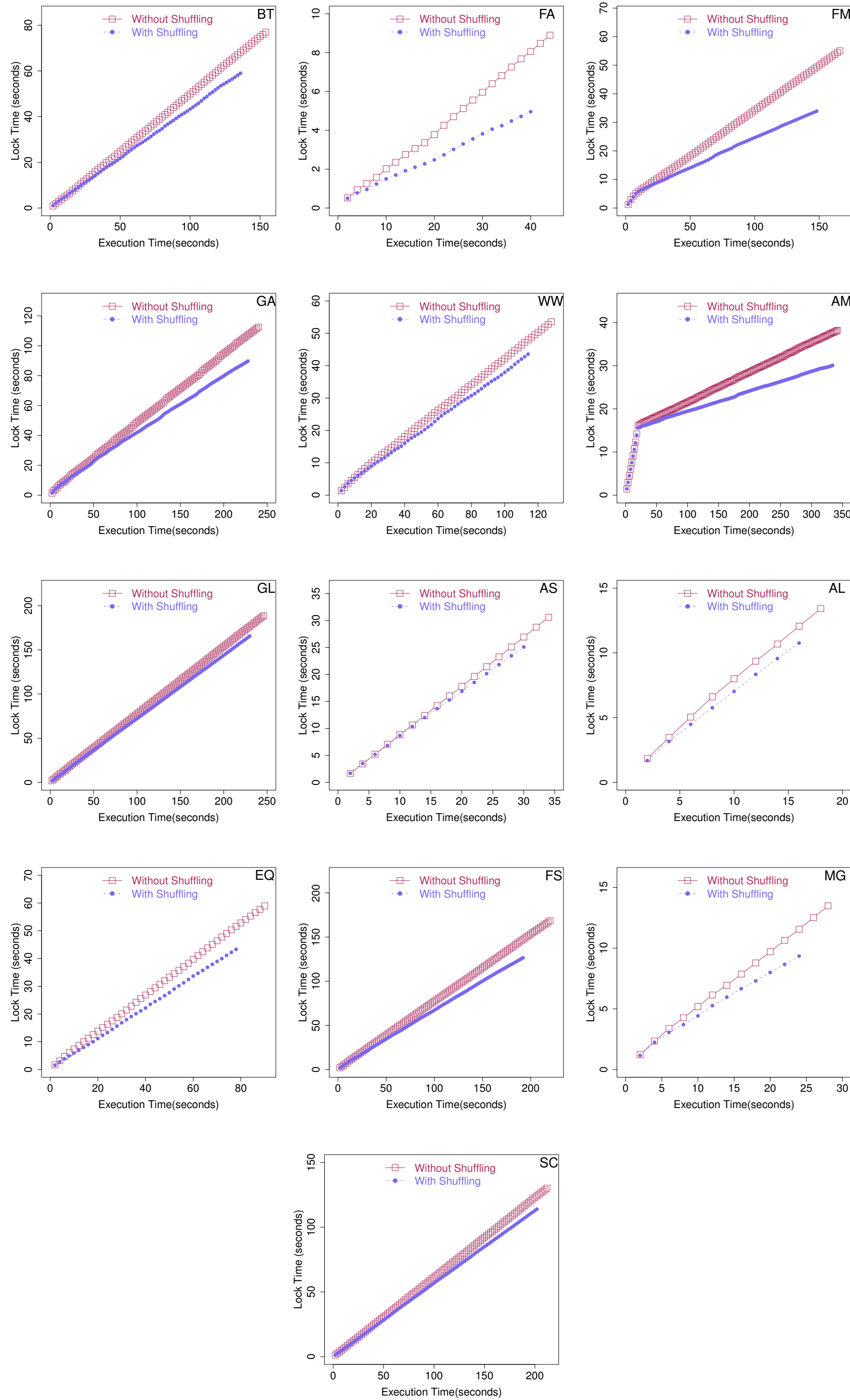


Figure 4.8: Time varying behavior of cumulative lock times without thread shuffling and with thread shuffling.

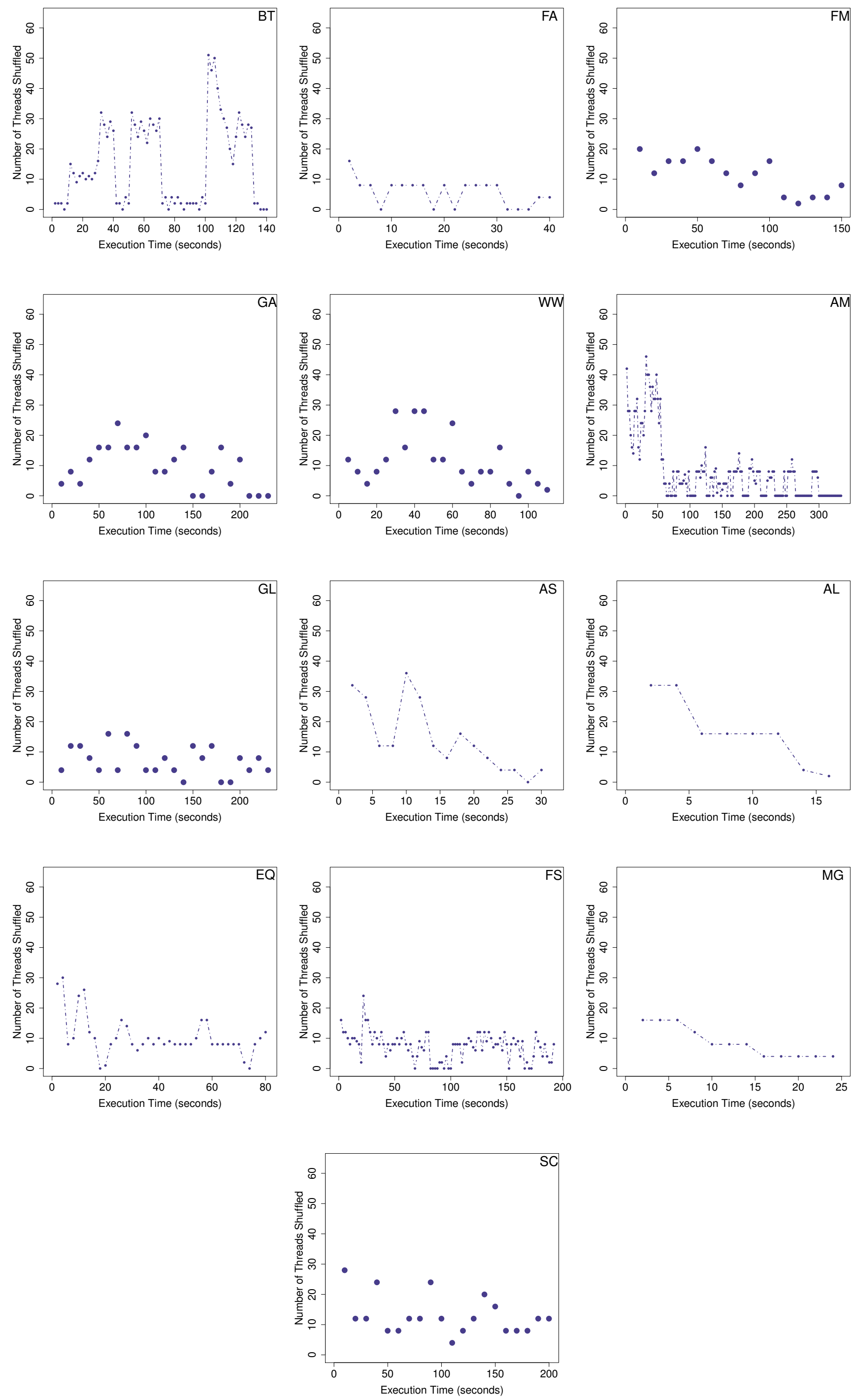


Figure 4.9: Time varying behavior of degree of thread shuffling.

shuffling is harmful and therefore the savings in lock times consistently continue to accumulate. In Figure 4.9 the number of threads that are shuffled every 2 seconds is plotted over the entire execution of the programs. As shown in Figure 4.9, for some benchmarks (BT, FA, AS, AL, EQ, MG) the execution can be divided into *small number of intervals* such that during each interval the number of threads shuffled *varies within a narrow range*. For a few benchmarks (AM, FS) the number of threads shuffled *varies rapidly within a narrow range* after the initial execution period. For the remaining programs (FM, GA, WW, GL, SC) the number of threads shuffled *varies rapidly across a wide range* (for clarity only the points in the graph are shown). Finally, *often no threads are migrated* is observed for some programs – (GA, AM, GL).

4.3.4 Multiple Applications

So far performance of a single application being run on the system is considered. In practice multiple multithreaded applications may be run simultaneously on the system. Next thread shuffling is tested against pairs of programs, each using all 4 CPUs, to see if thread shuffling improves performance of both applications. A pair of compute intensive applications (MG and AS); a pair of memory intensive applications (FS and EQ); and a combination of compute (FA) and memory (EQ) intensive applications were used. The results of running these application pairs with Solaris 11 and thread shuffling are presented in Table 4.1 (s next to the times indicates seconds). As shown in Table 4.1, in all three cases, both applications benefit from thread shuffling experiencing reductions in execution time of: 10.7% and 8.2%; 7.4% and 4.9%; and 6.4% and 7.2%. Therefore the above results

Table 4.1: Thread shuffling multiple applications.

	TTT	Lock Time	TTT	Lock Time
	MG		AS	
No Shuffling	25.1s	13.7s	27.1s	23.4s
Thread Shuffling	22.4s	11.2s	24.9s	20.4s
Δ	2.7s	2.5s	2.2s	3.0s
Reduction	10.7%	18.3%	8.2%	12.8%
η	1.08		0.73	
	FS		EQ	
No Shuffling	228.3s	131.3s	105.4s	30.9s
Thread Shuffling	211.4s	106.5s	100.2s	21.6s
Δ	16.9s	24.8s	5.2s	9.3s
Reduction	7.4%	18.9%	4.9%	30.1%
η	0.68		0.56	
	EQ		FA	
No Shuffling	98.1s	32.6s	71.3s	18.2s
Thread Shuffling	91.8s	23.5s	66.2s	11.3s
Δ	6.3s	9.1s	5.1s	6.9s
Reduction	6.4%	27.9%	7.2%	37.9%
η	0.69		0.74	

showed that thread shuffling is robust because it is beneficial for the multiple applications that are running on each CPU simultaneously.

4.4 Summary

This chapter presented a runtime technique, *Thread Shuffling*. *Thread Shuffling* continuously monitors lock times of individual threads of a multithreaded program for

adapting the location of threads across the CPUs of a ccNUMA machine to reduce the time spent by the threads in *acquiring locks* and improve performance of multithreaded programs. An average reduction in execution time is 9.7% was observed for programs belonging to PARSEC and SPEC OMP suites.

Chapter 5

Reducing Critical Section Delays

The performance of a multithreaded program is sensitive to the implementations of contention management policies and scheduling policies. For example, negative interaction between the time share (TS) thread scheduling policy and the spin-then-block lock-contention management policy dramatically increases lock holder thread preemptions under high loads. Therefore, under high load conditions, frequent preemption of lock holder threads can slow down the progress of lock holder threads and hence they spend more time in critical sections. Thus, degradation in performance of programs results.

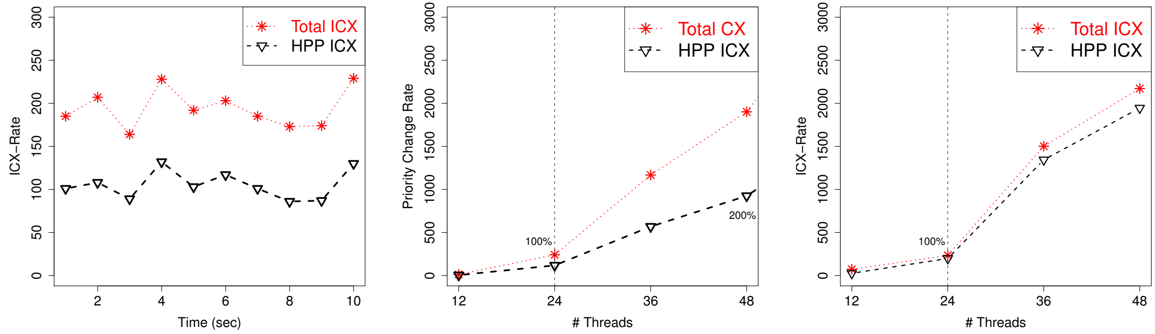
To address the above problem, this dissertation presents, *Faithful Scheduling* policy. *Faithful Scheduling* minimizes lock holder thread preemptions via adaptive time-quanta allocations and achieves high performance for multithreaded programs.

The next section explains how the interaction between contention management policy and OS scheduling policy hurts the performance of multithreaded programs running on multicore systems.

5.1 Interaction between OS Scheduling and Contention Management

Time Share (TS) is the default scheduling policy in a modern OS such as OpenSolaris. With TS scheduling policy, priorities of threads change very frequently for balancing load and providing fairness in scheduling. Priority adjustments are made based on the times threads spends waiting for processor resources, consuming processor resources, etc [7]. Therefore, at a given point in time, some of the threads belonging to an application get higher priority while the others get lower priority. This leads to preemptions of the low-priority threads of an application by the high-priority threads of the same application, i.e. 'Backstabbing' (BS). BS often includes lock-holder thread preemptions which increases the ICX rate. Further ICX is divided into two types: time-quantum expiration context-switches (TQE ICX) due to expiration of time-quantum; and preemption context-switches happen when a higher priority thread preempts a lower priority thread (HPP ICX).

As shown in Fig. 5.1(a), applu program (from SPEC OMP) experiences a high degree of HPP ICX (56% of total ICX) when it is run with 24 threads on 24 cores (100% load). Using DTrace [8] scripts, it is observed that almost all of these HPP ICX are caused by applu threads i.e., applu experiences around 55% BS at 100% load. Here BS is specifically defined as % of HPP ICX caused by the same application threads. This is because HPP ICX is also caused by high priority system processes running along with the application threads. However, one can expect that BS is the major portion of HPP ICX (i.e., HPP ICX $\sim$ BS) when load crosses 100%. As shown in Figure 5.1(b), priority change-rate



(a) HPP ICX occupies a major portion of total ICX. (b) HPP ICX leads to changes in thread priorities. (c) Drastic increase in HPP ICX as load crosses 100%.

Figure 5.1: Frequent changes in thread priority drastically increases thread context-switches.

increases as load increases and also a major portion of priority changes are due to HPP ICX. Another important point to note is that ICX (HPP ICX and TQE ICX) is responsible for a major portion of VCX. Figure 5.1(c) shows a drastic increase in HPP ICX as load crosses 100%. Therefore, one can expect that the frequency of lock-holder thread preemptions will increase once load crosses 100%. Thus, frequent ping-ponging [7] of thread priorities increases HPP ICX, specifically BS, which in turn increases CX-Rate (ICX-Rate + VCX-Rate), and ultimately vicious cycle is created between context-switches and priority changes.

As shown in Fig. 5.2, the TS policy changes priorities of threads based on their usage of system resources. Frequent ping-ponging of thread priorities leads to HPP ICX, i.e., forcing of threads off the CPU, which often include lock-holder threads. When a lock-holder thread is preempted then all the threads that are waiting for that lock will be blocked, i.e., VCXs are generated. Then threads will join the lock's sleep queue and their priorities will be changed based on their waiting time in the sleep queue. This process repeats continuously, increasing CX-Rate and priority change-rate, and thus leads to poor performance.

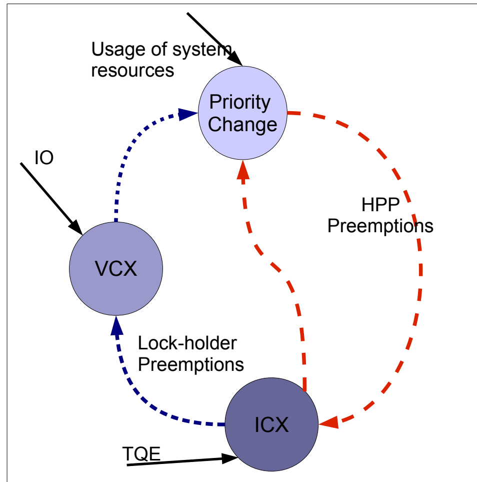
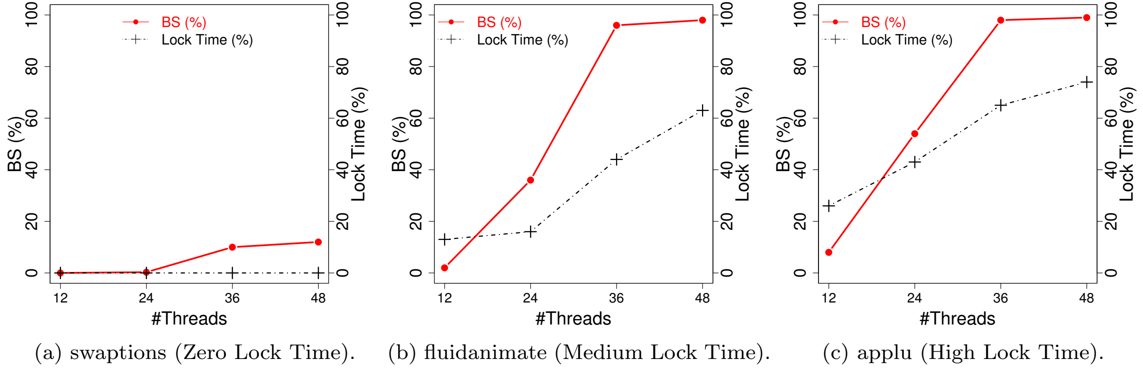


Figure 5.2: The interactions between the TS policy and the spin-then-block policy create vicious cycles between priority changes and context-switches.

Backstabbing (BS) vs Lock Time. High lock contention applications suffer more from BS than contention-free applications. This is because threads of high contention application seriously compete for lock acquisitions leading to high CX-Rate. Contention-free applications scale well and typically they experience CX-Rate far lower than high contention applications. To get a clear idea about this, three different benchmark programs (nearly contention-free, medium lock contention, and high lock contention) were run and variation in their BS was observed with varying thread count (i.e., load). Fig. 5.3 shows the results.

As shown in Fig. 5.3(a), swaptions is a nearly contention-free program and it does not significantly suffer from BS. BS is almost nil when the load is below 100% and small under high loads. This is because when the load crosses 100%, there are more chances of lock-holder thread preemptions and also HPP ICX is higher. However, this becomes a prominent problem for the high lock contention programs. As shown in Fig. 5.3 (b)



(a) swaptions (Zero Lock Time). (b) fluidanimate (Medium Lock Time). (c) applu (High Lock Time).

Figure 5.3: BS vs Lock Time (24 threads is 100% load).

and (c), programs fluidanimate and applu experience high % of BS. As applu is a high contention program, it suffers from high % of BS even under low loads. These observations demonstrate two things: (1) BS rapidly increases under high loads, specifically when the load crosses 100%, and (2) high contention programs experience significant BS even when the load is below 100%. Therefore, if BS is completely avoided then CX-Rate is minimized and performance is improved. In order to avoid BS completely, the vicious cycle between priority changes and context-switches must be broken.

5.2 FaithFul Scheduling (FF)

The previous section highlights the fact that the interactions between contention management and OS scheduling create vicious cycle between priority changes and context-switches, which leads to poor performance. Therefore, to break the vicious cycle and achieve high performance, a scheduling policy called Faithful Scheduling Policy (FF) with the following key characteristics is proposed:

1. Same priority is assigned to all the threads of a given application.
2. Time-quantum is allocated based on the resource usage of the entire application, specifically based on lock-contention and cache miss-ratio of the application.

By providing same priority to all the threads of an application, FF policy completely avoids BS, dramatically reduces CX-Rate, and leads to high performance. Since priorities of all the threads of an application are same, FF allocates equal time-quantum to all of them for reducing unwanted TQE ICX. Moreover, this makes all the threads of an application fair to each other. However, finding the right time-quantum for an application is tricky. For this, via extensive experimentation with a wide variety of benchmarks, a metric called "scaling-factor" was derived and a scaling-factor table is developed to guide time quantum allocation.

5.2.1 Scaling-factor Table

Finding the right time-quantum is very important to provide fair allocation of CPU cycles to all the threads of a multithreaded application. Threads of a CPU-intensive and low contention application heavily compete for CPU resources. Therefore, it is appropriate to provide small time quantum for both CPU-intensive and low contention application threads. In this way no thread will have to wait for a long time for a CPU. In contrast, it is appropriate to provide large time-quantum for both high-contention and memory-intensive application threads. In case of high-contention applications, large time-quantum allows lock-holder thread to complete its work quickly, release the lock, and allow other threads to make progress. Moreover large time-quantum for contention bound application threads

reduces unwanted TQE ICX and also reduces the lock acquisition overhead since a wakeup and a context-switch are required before the blocking thread can become the owner of the lock it requires [7]. Based on the above observations, the metric scaling-factor is defined in Eq. (1).

$$\boxed{\text{Scaling-factor} = 1 \text{ - } \max\text{ (Miss-ratio, Lock-time (\%))}} \tag{5.1}$$

In Eq. (1), the Miss-ratio is last-level cache miss ratio (misses/accesses) and Lock-time is the percentage of time application threads spend waiting for user locks, condition-variables, etc. Whether an application is memory-intensive or not is identified using the Miss-ratio.

Based upon the application's cache miss ratio and lock-contention, scaling-factor of the application is between one and zero. For scalable applications such as CPU-intensive and low-contention applications, scaling-factor is high and close to one, and for non-scalable applications such as high memory-intensive or high lock-contention applications, scaling-factor is close to zero. One important point here is that the scaling-factor value is for the entire application not per thread. Based on the scaling-factor value, FF policy allocates corresponding time-quantum to all the threads of the application.

By conducting experiments with a wide variety of multithreaded programs and different time-quanta, the scaling-factor table shown in Table 5.1 was developed. The time-quantum goes down as the scaling-factor goes up – this table is inspired by the priority dispatcher tables [7] of modern OS. More specifically, to derive the table, first the applications were categorized as memory intensive, CPU intensive, high contention, or low

Table 5.1: The Scaling-factor Table. The range of the scaling-factor is 0.10.

scaling-factor	**TQ(ms)**
(0.01 -- 0.10)	250
(0.11 -- 0.20)	200
(0.21 -- 0.30)	150
(0.31 -- 0.40)	120
(0.41 -- 0.50)	100
(0.51 -- 0.60)	80
(0.61 -- 0.70)	50
(0.71 -- 0.80)	30
(0.81 -- 0.90)	20
(0.91 -- 1.00)	10

contention applications. Then a few of the applications were selected from each category -- a total of 8 out of 22 applications, and they were run with varying time-quantum ranging from 10 ms to 400 ms to populate the table. The 8 applications used to populate the scaling factor table are: streamcluster, swim, swaptions, ferret, apsi, applu, art, and bodytrack. The scaling factor table obtained was then used in our experiments for all 22 applications.

5.2.2 Dealing with Phase Changes

Some applications have multiple execution phases which exhibit different usages of system resources. Therefore, the applications must be monitored continuously and appropriate time-quantum must be applied according to the resource usage during the current phase. However, among the 22 benchmark programs studied above, only a couple of programs ammp and SPECjbb2005 show two significantly different phases during their exe-

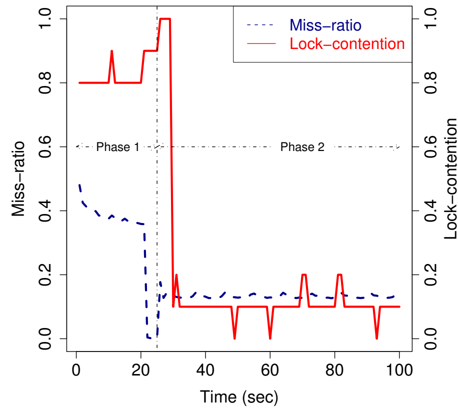


Figure 5.4: Phase changes of ammp. Here ammp is run with 24 threads. Lock-contention value 1 means application experiences lock-contention for 100% of the total elapsed time.

cution. For example, consider ammp, a SPEC OMP program. As shown in Fig. 5.4, ammp has two significantly different phases. While ammp experiences high miss-ratio and high lock-contention in Phase-1 (i.e., for the first 25 seconds), it experiences low miss-ratio and low lock-contention in Phase-2. Therefore according to the scaling-factor table, FF policy allocates large time-quantum for the first 25 seconds and small time-quantum for the rest of its execution.

5.2.3 Dealing with Pipeline Parallelism

It is fine to allocate equal time-quantum to all the threads of an application when they exploit data level parallelism. This is because, the threads of such applications more or less do similar work. However, it may not be appropriate to allocate equal time-quantum to all the threads of an application that use pipelined parallelism because resource usage of the threads from different pipeline stages may differ greatly.

However, our experiments with different pipeline parallel applications reveal that

allocating equal time-quantum to all the threads also works well for pipeline parallel applications. This is because although the scaling-factor is calculated based on the resource usage of the entire application, it is dominated by the threads of the stage that dominates the execution. For example, consider the pipeline of the ferret parallel application from the PARSEC suite. *ferret* is a search engine which finds a set of images similar to a query image by analyzing their contents. The program is divided into six pipeline stages -- the results of processing in one stage are passed on to the next stage. The stages are: Load, Segment, Extract, Vector, Rank, and Out. The speedup of ferret increases linearly starting from 6 threads to all the way up to 63 threads even though only 24 cores are available. The reason for the observed behavior is as follows. The *Rank* stage performs most of the work and thus the speedup of the application is determined by the *Rank* stage. Moreover the other stages perform relatively little work and thus their threads together use only a fraction of the computation power of the available cores. Thus, as long as cores are not sufficiently utilized, more speedup can be obtained by creating additional threads for the *Rank* stage. Therefore, Rank stage threads of ferret dominate the overall behavior of the application and thus the resource usage of whole program. Thus, allocating time-quantum based on the scaling-factor works well also for pipeline parallel programs such as ferret.

5.2.4 Implementation of FF Policy

There are two important components of the implementation of FF policy framework: providing same priority to all the application threads; and allocating appropriate time-quantum based on the resource usage of the application. OpenSolaris provides a

scheduling class called Fixed Priority scheduling [7]; with the combination of this class and priocntl(1) [2] utility, all the threads of an application can be assigned the same priority. However, there is no way to find appropriate time-quantum for an application in Open-Solaris with the fixed priority scheduling class. Moreover, this class does not provide any capability for updating time-quantum [7]. Thus, there is no way to deal with the phase changes of an application. Therefore, in addition to developing a scaling-factor table, continuously monitoring of an application is performed to allocate appropriate time-quantum according to its phase changes.

Let us consider the FF policy implementation in detail. As shown in Algorithm 3, the implementation uses a daemon thread. First the target program is started with the default TS policy and the monitoring of the program's last-level cache miss-ratio and lock-contention, after the creation of target program's worker threads, is started. The utility cputrack(1) is used to monitor miss-ratio and prstat(1) utility is used for lock-contention with one second interval. A timer is used, which fires a timer signal for every one second and the framework catches the signal and collects miss-ratio and lock-contention of the target program with one second interval, calculates a scaling-factor, and based on this it allocates appropriate time-quantum to the application threads using the scaling-factor table. More specifically, the framework measures the scaling-factor of the target application every second, and checks whether to change the time-quantum or not by comparing the current scaling-factor with the previous one. Although an interval with milliseconds resolution can be used, one second interval was used because our experiments showed that one second interval is enough to deal with the phase changes of the programs studied in this work.

Algorithm 3: FF Policy Framework

Profile Data Structure and Variables;

Profile P: (missRatio, lock time).

```
// range of the scaling-factor
```

range = 0.10;

Subroutines:

getProfile(): return Profile P;

getScalingFactor(missRatio, lockContention): return [1 - max(missRatio, lockContention)] ;
getTimeQuantum(scalingFactor): return corresponding TQ from the Scaling-Factor Table;

Input : Target Multithreaded Benchmark Program

Output: Apply FF policy.

Start the target program with TS policy;

while *program hasn't create its worker threads* **do**

  Sleep(); `// checks like a deamon process`

end

Wait for one more second to allow the application threads for their initialization period;

oldP = getProfile();

oldScalingFactor = getScalingFactor(oldP.missRatio, oldP.lockContention);

oldTQ = getTimeQuantum(oldScalingFactor);

Allocate oldTQ and same priority using *priocntl*(1) utility;

```
// continuous monitoring
```

repeat

  newP = getProfile();

  newScalingFactor = getScalingFactor(newP.missRatio, newP.lockContention);

  if *(newScalingFactor > (oldScalingFactor + range))* **or** *(newScalingFactor < (oldScalingFactor - range))* **then**

    newTQ = getTimeQuantum(newScalingFactor);

    oldScalingFactor = newScalingFactor;

    Allocate newTQ using priocntl utility;

  end

until *completion of the target program*;

Although, the minimum timeout value with the default implementation of prstat(1) utility provides one second time-interval, this utility was modified to allow time intervals with millisecond resolution to monitor lock-contention. Therefore, it is possible to use an interval of less than one second for an application that experiences rapid phase changes.

In summary, the above framework continuously monitors the target multithreaded program and allocates same priority using priocntl(1) utility and assigns appropriate time-quantum based on the scaling-factor table. Moreover, the overhead of this framework is negligible (0.02% of CPU utilization) and it requires no changes to the application source code or to the OS kernel.

5.3 Evaluating FF policy

5.3.1 Benchmarks

FF policy is evaluated with a wide variety of benchmarks -- 22 benchmark programs in all. A micro-benchmark [10] is also included to study how FF policy works under varying levels of contention. This benchmark consists of M threads running on N cores that repeatedly acquire and release a single global lock. The critical section consists of a single call to gethrtime(), which takes around 300 ns to execute on our machine. Between lock acquires, threads busy-wait a fixed period of time before the first measurement and stop after the last one. Threads increment a local counter with each lock release, and the benchmark harness computes throughput by comparing two successive reads of each thread's counter while threads continue to run.

The other 21 complete programs are as follows: eight programs (*streamcluster, facesim, canneal, x264, fluidanimate, swaptions, ferret, and bodytrack*) from PARSEC [1], 11 programs (*swim, wupwise, equake, gafort, art, apsi, ammp, applu, fma3d, galgel, and mgrid*) from SPEC OMP [11], *SPECjbb2005* [11], and *TATP* [12] database transaction program. The implementations of PARSEC programs are based upon *pthreads* and they were run on *native inputs*. SPEC OMP programs were run on medium input data sets. SPECjbb2005 with single JVM is used in all our experiments. TATP (a.k.a NDBB and TM-1) uses a 10000 subscriber dataset of size 20MB with a solidDB [13] engine. TATP is not IO-intensive and disk performance does not affect it significantly [10]. In this work, each experiment was run 10 times and present average results from the ten runs.

5.3.2 Against varying contention levels

Since FF policy completely avoids BS and specifically lock-holder thread preemptions, it is very effective against varying lock-contention levels. Fig. 5.5 demonstrates this. A microbenchmark is used where threads contend for a single global lock, with a fixed delay between requests [10]. High contention occurs for short requests on the left of the x-axis and drops off moving toward the right. Here, three cases are considered, where the machine is 95% loaded (i.e., 23 threads), 150% loaded (i.e., 36 threads) and 200% loaded (i.e., 48 threads) [10]. As shown in Fig. 5.5, contention decreases along the x-axis, and throughput is improved in all three cases. As shown in Fig. 5.5, when contention is high and the system is overloaded, program experiences high BS, and leads to poor performance. For lightly loaded systems, FF performs slightly better than TS because program experiences low BS. However, overall, FF outperforms TS significantly at all contention levels.

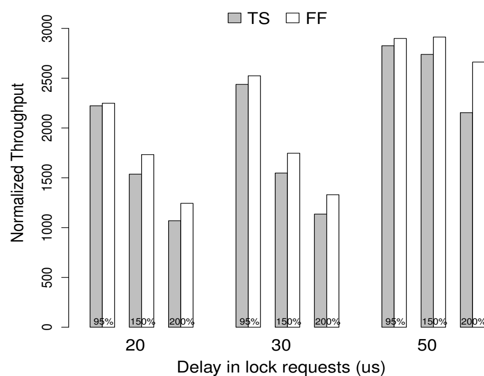
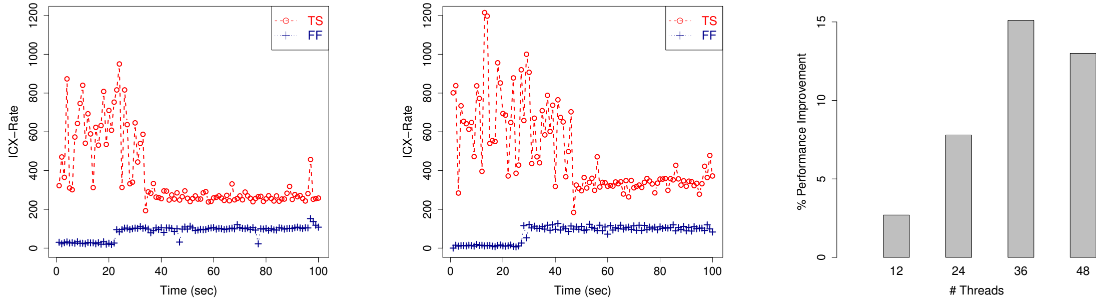


Figure 5.5: FF policy is very effective against varying contention levels.

5.3.3 Against phase changes

As explained in Section 5.2.4, the FF framework continuously monitors the target multithreaded program and allocates appropriate time-quantum to effectively deal with its phase changes. For example, consider the *ammp* program which exhibits two significantly different execution phases described in Section 5.2.2. Using the scaling-factor table, the FF policy allocates appropriate time-quantum according to the resource usage of its phases. As *ammp* suffers from high lock contention for around 84% of elapsed time in the first phase, scaling-factor is 0.16 for the first phase. Here lock-contention is higher than miss-ratio value. Likewise, scaling-factor is 0.88 for the second phase of the *ammp* program as it suffers from low lock contention for around 12%. Therefore, using continuous monitoring, the FF policy allocates time-quantum 200 ms for the first phase, 20 ms for the second phase, and thus effectively deals with the phase changes of the *ammp* program.

As shown in Fig. 5.6, FF policy is very efficient against the phase changes of *ammp* program. It dramatically reduces ICX-Rate and leads to high performance. As shown in



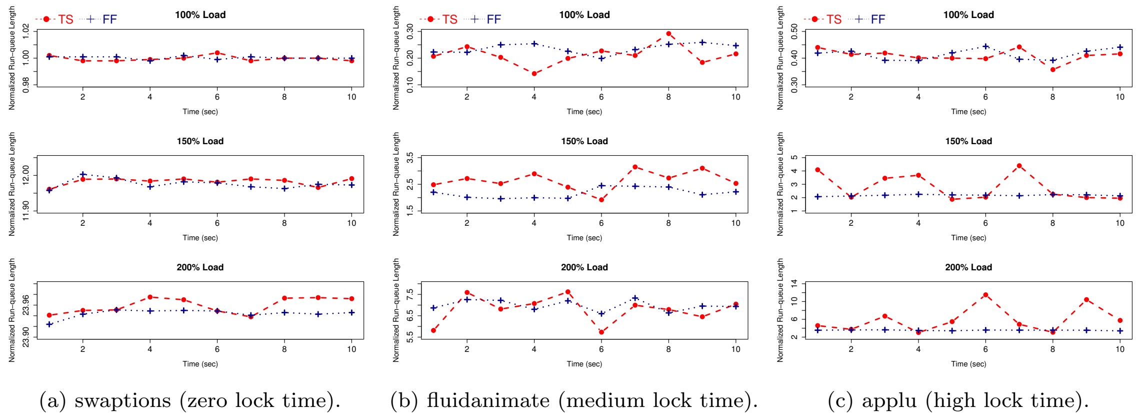
(a) Dramatic reduction in ICX-Rate at 100% load. (b) Dramatic reduction in ICX-Rate at 150% load. (c) Performance improvement over TS policy.

Figure 5.6: FF policy effectively deals with phases of ammp program and improves its performance.

Fig. 5.6(c), *ammp* achieves up to 15% performance improvement with FF policy. Since, small time-quantum is used for the phases that have high scaling-factor, one can expect a little increase in TQE ICX. Thus, as shown n Fig. 5.6, there is a rise in the ICX-Rate in the second phase with FF policy. However, FF policy produces less TQE ICX compared to TS policy at both 100% and 150% loads.

5.3.4 Against dynamic load changes

Since FF policy completely eliminates BS, consequently reducing CX-Rate, it brings stability in load management. Fig. 5.7 demonstrates this. The y-axis of the figure represents normalized run-queue length of the system, i.e. total number of runnable threads on the dispatcher queues of the system [2, 7]. The x-axis shows the time in seconds. Fig. 5.7 shows the normalized run-queue lengths of swaptions, fluidanimate, applu programs at 100%, 150%, and 200% loads. As shown in Fig. 5.7(a), there are no significant



(a) swaptions (zero lock time). (b) fluidanimate (medium lock time). (c) applu (high lock time).

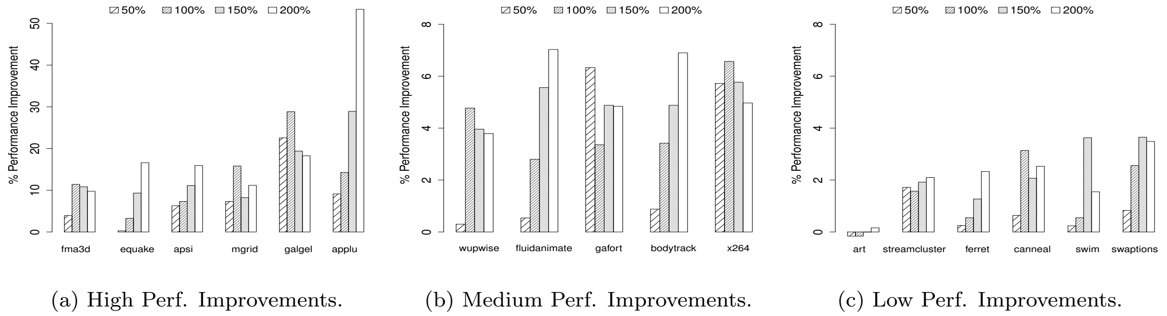
Figure 5.7: FF policy avoids spikes in the load.

load changes with both TS and FF policies in case of very low contention swaptions program even at 200% load. However, there are significant spikes in the load for high contention programs -- fluidanimate and applu -- with TS policy, but there are no spikes in the load with FF policy. Therefore, by completely eliminating BS and consequently reducing CX-Rate, FF policy avoids spikes in the load and leads to high performance. Moreover, threads experience higher CPU latencies with TS policy under high loads compared with FF policy, i.e., threads wait for longer times in the dispatch queues with TS policy, which slows down the progress of the application.

Thus, FF policy is agnostic to dynamic load changes and improves performance predictability of multithreaded programs running on multicore machines. In contrast to this, the load-controller [10] is sensitive to spikes in the load.

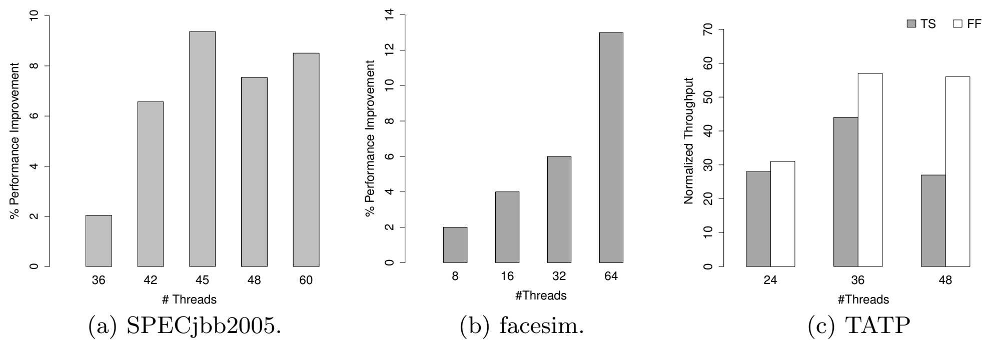
5.3.5 Performance Improvements

As shown in Fig. 5.8 and 5.9, FF policy improves performance for a wide variety of programs at 50%, 100%, 150%, and 200% loads over TS policy. As high contention programs



(a) High Perf. Improvements. (b) Medium Perf. Improvements. (c) Low Perf. Improvements.

Figure 5.8: FF policy improves performance of a wide variety of programs.



(a) SPECjbb2005. (b) facesim. (c) TATP

Figure 5.9: Performance improvement of SPECjbb2005, facesim, and TATP with FF policy.

suffer heavily from BS, they achieve tremendous performance improvement with FF policy. Fig. 5.8(a), Fig. 5.9, and Fig. 5.6(c) all show this. There are moderate improvements for the medium contention programs shown in Fig. 5.8(b) and small improvements for the low contention programs shown in Fig. 5.8(c). Although FF policy considers whole application for allocating time-quantum, as shown in Fig. 5.8(b) and Fig. 5.8(c), FF policy improves performance of *pipeline parallel* programs bodytrack, x264, and ferret.

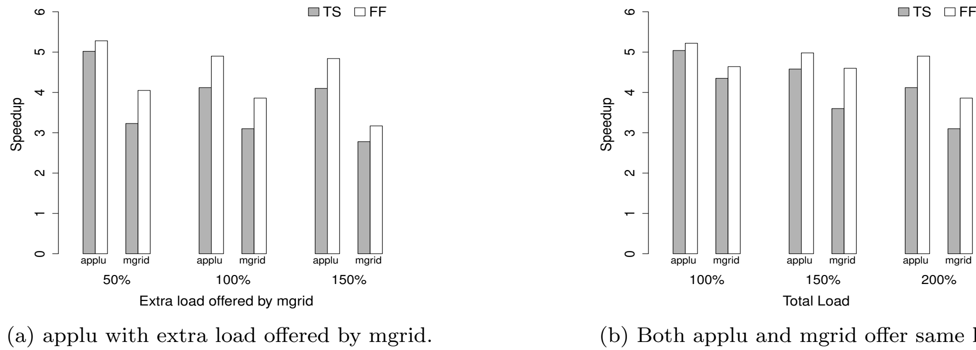
More specifically, at 100% load, FF policy achieves more than 10% performance improvement for five programs with a maximum of 35% improvement, 4%-10% for six programs, less than 4% for nine programs, and there is no improvement for one program over TS policy. At 200% load, FF policy achieves more than 10% performance improvement

for eight programs with a maximum of 107% improvement, 4%-10% for six programs, less than 4% for seven programs over TS policy. Moreover, FF policy also achieves performance improvements for several programs under light loads, specifically at 50% load.

Since our execution environment is different from [10], it is not possible to directly compare the performance improvement data of TATP using our FF policy against the performance improvement with the load-controller [10]. However, as shown in Fig. 5.9 (c), FF policy improves performance of TATP like the load-controller does and also the performance degradation is steady as load increases. Moreover, in contrast to the load-controller, there is no need to modify the application source code for ensuring visible spin locks and also FF policy is agnostic to dynamic load changes.

5.3.6 Multiple Applications

TS policy has been widely used in modern operating systems. It does not consider the whole application but rather assigns priority and time-quantum on a per thread basis. That is why FF policy significantly outperforms TS policy when single multithreaded application is running on the system. However, TS policy is quite effective when there are multiple multithreaded applications running on a multicore system. Therefore, FF policy is tested with parallel runs of more than one application on a multicore system. For this evaluation, two experiments were conduced. In the first experiment applu with 24 threads was run along with extra load offered by mgrid – run 24 threads of applu along with 12 threads, 24 threads, and 36 threads of mgrid. In the second experiment both applu and mgrid were run with equal number of threads – (12, 12), (18, 18), and (24, 24) threads. As



(a) applu with extra load offered by mgrid. (b) Both applu and mgrid offer same load.

Figure 5.10: FF policy is very effective against parallel runs of more than one application.

shown in Fig. 5.10 (a) and Fig. 5.10 (b), FF policy greatly outperforms TS policy. Thus, FF policy is also effective when more than one application is running on the system.

5.4 Summary

This chapter presented a scheduling policy, *FaithFul Scheduling* (FF), which continuously monitors last-level cache miss-ratio and lock time of a multithreaded program using simple utilities available on modern OS. Using these, it adaptively allocates time-quantum and significantly reduces lock holder thread preemptions. FF policy significantly improves performance of multitreaded programs running on a multicore systems under high loads. The experimental results show that at 100% load, FF policy achieves more than 10% performance improvement for five programs with a maximum of 35% improvement, 4%-10% for six programs, less than 4% for nine programs over TS policy. At 200% load, FF policy achieves more than 10% performance improvement for eight programs with a maximum of 107% improvement, 4%-10% for six programs, less than 4% for seven programs over TS policy.

Chapter 6

Coscheduling Multiple Multithreaded Applications

Since the performance of multithreaded applications often does not scale to fully utilize the available cores in a multicore system, simultaneously running multiple multithreaded applications becomes inevitable to fully utilize such machines. However, coscheduling multithreaded programs effectively on such machines is a challenging problem because of their complex architecture [14, 15]. For effective coscheduling of multithreaded programs, the OS must understand the resource-usage characteristics of multithreaded programs and then adaptively allocate cores as well as select appropriate memory allocation and scheduling policies.

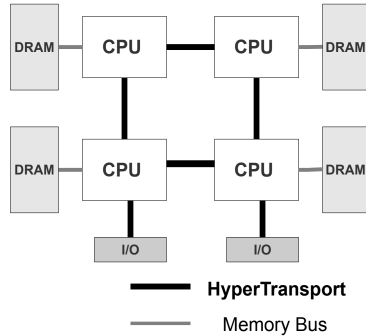
To address this problem, this dissertation presents a runtime technique called ADAPT. It uses supervised learning techniques for predicting the effects of interference between programs on their performance and adaptively schedules together programs that

interfere with each other's performance as little as possible. It achieves high throughput, high system utilization, and fairness when running multiple multithreaded applications.

6.1 Cache Miss-Ratio vs Lock-contention vs Latency

Identifying important program resource-usage characteristics is crucial in developing an effective coscheduling technique for multithreaded programs running on multicore systems. For this, experiments involving coscheduling four multithreaded benchmarks from the PARSEC and SPEC OMP suites, facesim (FS), bodytrack (BT), equake (EQ), and applu (AP) on a 64-core machine running Solaris 11 were conducted. In this work, multithreaded programs are run with OPT Threads, where OPT Threads of a multithreaded program is the minimum number of threads that gives maximum performance on our 64-core machine. As shown in Figure 6.1(a), the machine has four 16-core CPUs (four sockets), i.e., a total of 64-cores. To capture the distance between different CPUs and memories, a new abstraction called "locality group" (lgroup) has been introduced in Solaris. Lgroups are organized into a hierarchy or topology that represents the latency topology of the machine [7].

As shown in Figure 6.1(b), there are two different cores configurations that can be used for coscheduling multiple multithreaded programs on a multicore system: *all-cores* configuration; and *processor-set* configuration. In all-cores configuration each program is run using all the cores while in processor-set configuration each program is run on a separate processor-set to minimize interference between the programs. A processor-set is a pool of cores such that if a multithreaded program is assigned to a processor-set, OS migrates the threads of the program only across the cores belonging to the pool for balancing load. Next



(a) Our 64-core machine.

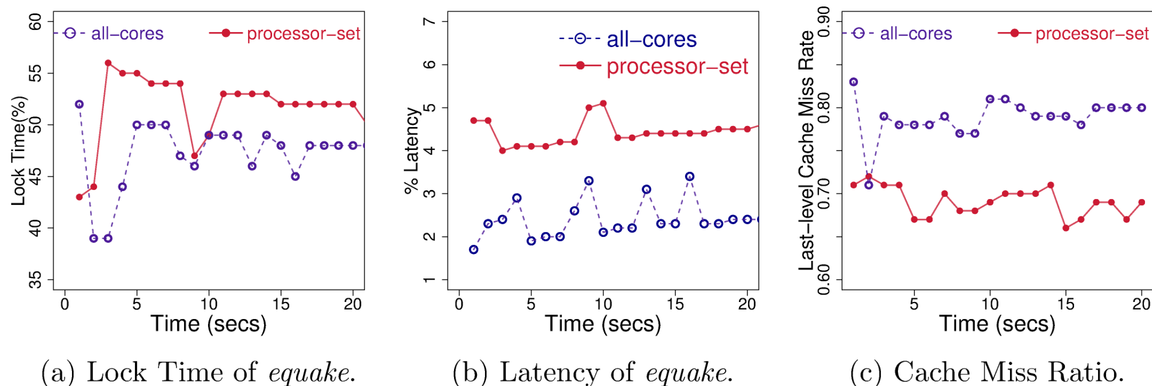
No.	Configuration	#Cores to A	#Cores to B
1	All-cores	64	64
2	Processor-set	32	32
3	Processor-set	24	40
4	Processor-set	40	24
5	Processor-set	16	48
6	Processor-set	48	16

(b) Cores-configurations.

Figure 6.1: The machine has four 16-core CPUs and are interconnected with HyperTransport. Table shows the number of cores allocated to two programs A and B in different cores-configurations.

how the impact of cores configuration on lock times (time spent on lock operations and in critical sections), latency (time ready threads spend waiting for a core to become available), and last level cache miss ratios collectively determines which configuration is most suitable is illustrated.

When two memory-intensive and high lock contention multithreaded programs *facesim* and *equake* are run simultaneously in the above two configurations, all-cores configuration gives better overall performance. This is even though, due to their memory-intensive nature, these programs suffer from higher last-level cache miss ratios under all-cores configuration. This is because *facesim* and *equake* are also high lock-contention programs. As shown in Figure 6.2, *equake* experiences high lock times and latency in processor-set con-



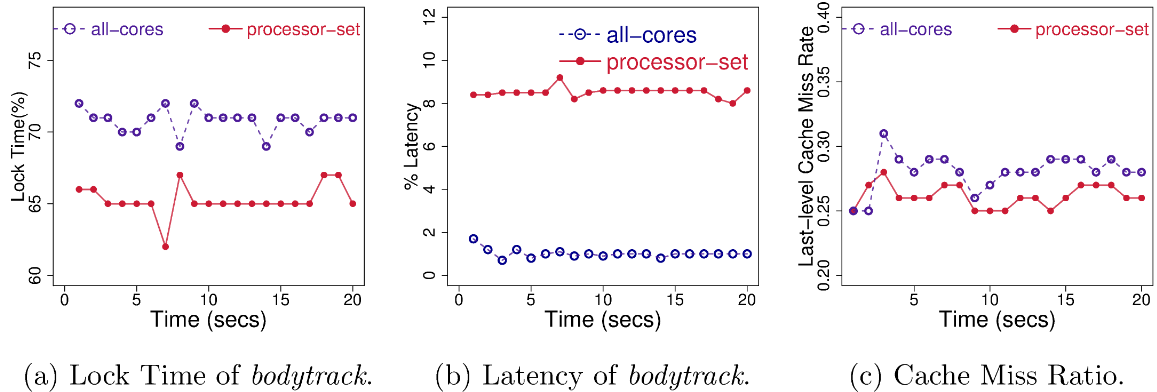
(a) Lock Time of *equake*. (b) Latency of *equake*. (c) Cache Miss Ratio.

Figure 6.2: Lock time (% of execution time), Thread Latency (% of execution time), and Last-level Cache Miss Ratio of *equake* when executed with *facesim*.

figuration compared to all-cores configuration. Likewise, although not shown here, *facesim* also experiences high lock times and latency in processor-set configuration compared to all-cores configuration. Thus, the trade-off between lock contention, latency, and last-level cache miss ratio results in the all-cores configuration delivering better performance.

When two CPU intensive and high lock-contention multithreaded programs *bodytrack* and *applu* were run, processor-set configuration provides high performance compared to all-cores configuration. As shown in Figure 6.3, thread latency of *bodytrack* is low in all-cores configuration compared to processor-set configuration. Likewise, although not shown here, the thread latency of *applu* is also low in all-cores configuration. However, as shown in Figure 6.3, lock time and last level cache miss ratio for *bodytrack* are higher in all-cores configuration. Likewise, the lock times of *applu* is also high in all-cores configuration. Therefore, the trade-off between lock times, last level cache miss ratio, and thread latency results in the processor-set configuration delivering better performance.

Therefore, the above experiments demonstrate that architectural factors such as



(a) Lock Time of *bodytrack*. (b) Latency of *bodytrack*. (c) Cache Miss Ratio.

Figure 6.3: Lock Time (% of execution time), Thread latency (% of execution time), and Last-level Cache Miss Ratio of *bodytrack* when executed with *applu*.

last-level cache miss-rate (MPA) alone are not enough for effective coscheduling of multithreaded programs on a multicore system. The OS must consider application characteristics of lock-contention and thread latency along with MPA. Based on these observations, in the next section, a framework called ADAPT is presented. It continuously monitors appropriate resource-usage characteristics of the target multithreaded programs on line, and based on these, it effectively coschedules multithreaded programs.

6.2 The ADAPT Framework

The ADAPT framework has two major components: Cores Allocator; and Policy Allocator. The Cores Allocator is responsible for selecting appropriate cores-configuration, and the Policy Allocator is responsible for applying appropriate memory allocation and scheduling policies adaptively according to the resource-usage characteristics of the programs. The following sections provide detailed description of these components.

6.2.1 The Cores Allocator

To capture a variety of application resource-usage characteristics for effective coscheduling of multithreaded programs, the Cores Allocator uses statistical models constructed using supervised learning, where a set of sample input-output values is first observed and then a statistical model is trained to predict similar output values when similar input values are observed [16]. The Cores Allocator uses two statistical models: one for approximating performance loss of a program when it runs together with another program and another for approximating performance of a program when the configuration is changed from processor-set to all-cores configuration, and vice-versa. Let us call the first model as **PAAP** (**P**erformance **A**pproximation of a program when it is running with **A**nother **P**rogram) and the second model as **PACC** (**P**erformance **A**pproximation of a program when it is running with different **C**ores **C**onfiguration). Using PAAP model, Cores Allocator predicts average performance of the programs in all-cores configuration, and using PACC model, it predicts average performances of the programs in the five different processor-set configurations listed in Figure 4.2(b). Then it chooses the configuration that gives the best average performance.

Developing Statistical Models.

To cover the resource-usage characteristics of a wide spectrum of programs, first the programs are categorized as memory intensive, CPU intensive, high lock-contention, or low lock-contention programs. Then 12 programs are selected (out of 26), a few of programs from each category, and they are used to develop the models. The following 12 programs were chosen: bodytrack, facesim, ferret, fluidanimate, streamcluster from PARSEC; applu,

art, swim, equake from SPEC OMP; SPEC JBB2005; kmeans and pca from Phoenix. The resource-usage characteristics of the programs are used as inputs to the statistical models as explained next.

The PAAP Model.

Data Collection. Twelve predictors (or inputs) shown in Table 6.1 were chosen for developing PAAP model. The goal is to predict the performance of a program A when it is running with another program B. These 12 predictors (6 from each program) represent the resource-usage characteristics of both programs A and B. As shown in Table 6.1, r_X represents a resource-usage characteristic value 'r' of program 'X' in its solo run with OPT Threads. From the combinations of the above mentioned 12 programs, 144 data points are collected, where each data point is a 13-tuple containing 12 predictors and the observed usr_ab as the target parameter shown in Table 6.1. Here each of the 12 programs contributes 12 data points including a combination with itself. Solaris 11 utilities prstat(1) and cpustat(1) with 100 ms time-interval are used to collect 100 samples, and the average of these samples are used as the final values of the predictors. The cpustat(1) utility is used to collect *mpa* and the prstat(1) utility is for the remaining predictors. Here, the assumption used is that the percentage of elapsed time a program spends in user mode represents its progress or performance in the coscheduling run.

Finding Important Predictors. To balance the prediction accuracy and cost of the approximation, forward and backward input selection techniques with "Akaike information criterion" (AIC)are used for finding important predictors among the 12 initial predictors.

Table 6.1: Initial predictors and the target *usr_ab* of the PAAP model.

Predictor	**Description**
mpa_x	average last-level cache miss ratio of x.
usr_x	the percentage of elapsed time x spends in user mode.
sys_x	the percentage of elapsed time x spends in processing system calls, system traps, etc.
lat_x	latency of x.
lock_x	lock-contention of x.
ct_x	cores to threads ratio of x, i.e., (#cores / #threads of x).
usr_ab	the percentage of elapsed time A spends in user mode when it is running with B.

Table 6.2: VIF values of PAAP predictors.

mpa_a	lock_a	lat_b	ct_b	sys_a
1.6	2.1	2.1	1.6	1.3

The AIC is a measure of the relative goodness of fit of a statistical model [16]. Five most important predictors: *lock_a, lat_b, ct_b, mpa_a,* and *sys_a* from the above 12 initial predictors are derived by using R stepAIC() [17] method. The predictors were also tested against multicollinearity problem for developing robust models. Multicollinearity is a statistical phenomenon in which two or more predictor variables in a multiple regression model are highly correlated. In this situation the coefficient estimates may change erratically in response to small changes in the model or the data. R "Variance Inflation Factor" (VIF)

method is used to observe the correlation strength among the predictors. If VIF > 5, then the variables are highly correlated [18]. As shown in Table 6.2, the variables are not highly correlated and therefore there is no multicollinearity problem.

Model Selection. Next, three popular models based on supervised learning techniques are developed using the five important predictors. The models are: a) Linear Regression (LR); b) Decision Tree (DT); and c) K-Nearest Neighbour (KNN) [16]. R statistical methods lm() [17], rpart() [17], and kknn() [17] are used for developing these models. Here the decision tree model is pruned using R prune() [17] method to avoid over-fitting problem. As shown in the LR model (Equation 6.1), lock-contention, latency, cache miss-rate, system overhead of program A are affecting negatively on its performance when it is running with program B, i.e., if there is an increase in any of these four predictors, then *usr_ab* decreases. If cores-to-threads ratio of program B is increased (i.e., # threads of B is decreased) then the performance of program A will be increased, and vice-versa.

The three models: LR, DT, and KNN are evaluated using a 12-fold cross-validation (CV) test [16]. Table 6.3 shows the adjusted R^2 values of these models on full training data and prediction accuracies in the 12-fold CV test. In a 12-fold CV test, the data (144 points) is split into 12 equal-sized parts, the function approximator is trained on all the data except for one part and a prediction is made for that part. For testing the models thoroughly, the models are trained on the data of 11 different programs (132 data points) and tested against

$$usr_ab = (65.2)+(-0.6*lock_a)+(-0.8*lat_b)+(-9.6*mpa_a)+(-10.2*sys_a)+(7.8*ct_b) \quad (6.1)$$

the data of *12th program* (12 data points test-set). The testing data is completely different from the training data. The metric "prediction accuracy" is defined as: (100 - sMAPE), where sMAPE is symmetric mean absolute percentage error defined in Equation 6.2 [19]. DT model has the highest prediction accuracy among the three models and therefore the DT model was chosen as PAAP model.

Table 6.3: Models

Model	**Adjusted R^2**	**Prediction Accuracy**
LR	0.90	88.5
DT	0.94	90.4
KNN	0.88	87.1

$$sMAPE = \frac{1}{N}\sum_{i=1}^{N} \frac{|A_i - F_i|}{(A_i + F_i)/2} \times 100 \tag{6.2}$$

where A_i is the actual value and F_i is the forecast value.

The PAAC Model.

Data Collection. Six predictors were chosen for developing PACC model. The goal is to predict the performance of a program A when it is running with different cores-configuration. The six predictors are: *usr_A, sys_A, lat_A, lock_A, ct_A*, and *rct_A*. As described in the development of PAAP model, the first five predictors represent the resource-usage characteristics values of program A in its solo run, and the remaining predictor rct_A is a cores-configuration with reduced cores (less than 64) to threads ratio of program A. The

goal is to predict the performance (usr_acc, i.e., % user-mode time) of program A when it is running with different cores-configuration. Each of the above 12 programs was run with 64, 56, 48, 40, 32, 24, and 16 cores and collected 6 points from each run with their OPT threads. From the solo runs of the above 12 programs, 72 data points were collected, where each data point is a 7-tuple containing six predictors and the observed usr_acc.

Finding Important Predictors and Model Selection. As in the development of PAAP model, two most important predictors were derived for PACC model among the six predictors, and the two predictors are: $lock_a$ and rct_a. The LR model developed with these two predictors is shown in Equation 6.3. As shown in Table 6.4, the VIF values of these predictors are also less than 5. Therefore, there is no multicollinearity problem. As in deriving the PAAP model, LR, DT, and KNN models are developed using the two important predictors: $lock_a$ and rct_a. As shown in Table 6.5, LR model (Equation 6.3) has the best prediction accuracy in a 12-fold CV test. Therefore, the LR model was chosen as PACC model.

Thus, using PAAP and PACC models, Cores Allocator allocates appropriate cores-configurations according to the resource-usage characteristics of the programs. Moreover, the overhead of these models is not considerable as they use a very few predictors that capture all all of the important information. In the next section, the design of Cores Allocator is described.

$$usr_acc = (18.6) + (-0.3 * lock_a) + (32.5 * rct_a) \tag{6.3}$$

Table 6.4: VIF values of the PACC model predictors.

Predictor	**lock_a**	**rct_a**
VIF	1.2	1.2

Table 6.5: Models

Model	**Adjusted R^2**	**Prediction Accuracy**
LR	0.88	89.2
DT	0.86	87.6
KNN	0.86	85.2

The Design of Cores Allocator.

Cores Allocator considers a realistic scenario, where programs randomly enter and leave the system. Let us consider a base case, for coscheduling two programs: a program P_1 is already running with its corresponding OPT threads and another program P_2 is just entered into the system. If P_2 is CPU-intensive and low lock-contention program, then irrespective of the current cores-configuration and the programs already running on system, Cores Allocator allocates all-cores configuration to P_2. Otherwise, it predicts performances of P_1 and P_2 using PAAP and PACC models, and allocates the cores-configuration that gives better average TTT. Likewise, it coschedules N programs by allocating appropriate cores-configuration using PAAP and PACC models. Let us consider another scenario, where programs P_1, P_2, ..., P_N are already running and any arbitrary program P_i is complete its execution and leaves the system. If P_i is CPU-intensive and contention-free program, it

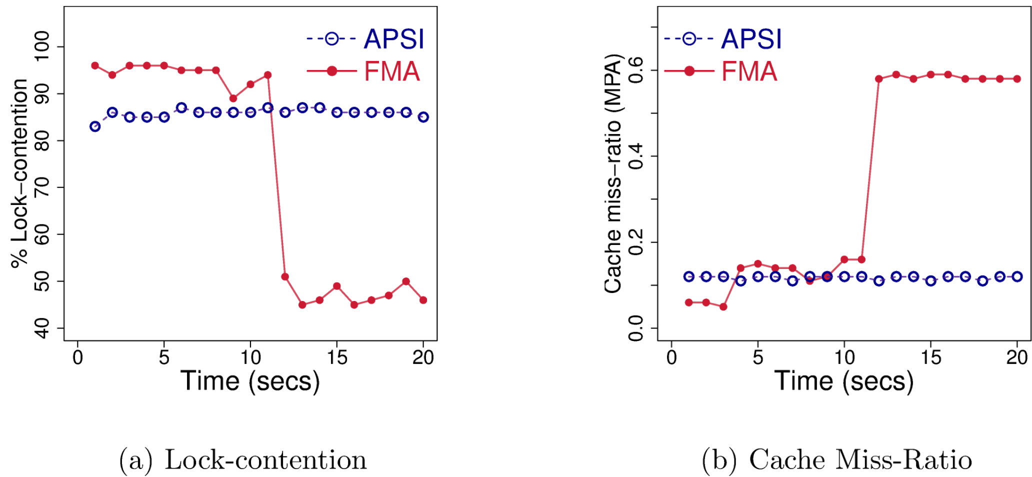
keeps the currents configuration for the remaining programs. Otherwise, it shares the cores released by P_i equally to the remaining programs.

Table 6.6: The actual and predicted *usr_FA* and *usr_SM* values with PAAP and PACC models are shown here.

Program	**All-cores**		**Processor-set**	
	Actual	Predicted	Actual	Predicted
FA	52.3	56.4	41.2	44.5
SM	49.4	45.2	46.8	41.6

Let us consider an example that shows how Cores Allocator selects an appropriate cores configuration for programs fluidanimate (FA) and swim (SM). Both FA and SM are memory-intensive and low lock-contention programs, and their corresponding OPT threads are 49 and 32. Using PAAP and PACC models, first Cores Allocator predicts the performance of FA and SM in all-cores and in the best processor-set configuration (40 cores to FA, 24 cores to SM). Table 6.6 shows that both FA and SM programs have high %USR (the percentage of elapsed time a program spends in user-mode) in the all-cores configuration. Therefore, Cores Allocator selects the all-cores configuration for the coscheduling of FA and SM and the all-cores configuration improves TTT of FA and SM by 14% compared to the processor-set configuration.

Dealing with Phase Changes. Since FA and SM do not show *significant* phase changes on our machine, there is no switching back and forth between different cores configurations.



(a) Lock-contention

(b) Cache Miss-Ratio

Figure 6.4: While APSI has steady behavior, FMA shows a significant phase change.

The initial predicted processor-set configuration gives best performance, and therefore Cores Allocator keeps all-cores configuration for the entire duration of the FA and SM coscheduled run. However, some programs show significant phase changes. Therefore, for adaptively allocating appropriate cores-configuration according to the phase changes of the programs, the program is continuously monitored.

Let us consider a coschedule run of two high lock-contention programs apsi (APSI) and fma3d (FMA) with their OPT threads 16 and 56 respectively. As shown in Figure 6.4, FMA has one significant phase change, while APSI shows steady behavior. FMA experiences very high lock-contention in its first 11 seconds of its life-time, while APSI experiences very high lock-contention steadily throughout of its life-time. Therefore, by continuous monitoring, using PAAP and PACC models, the Cores Allocator applies (16, 48) processor-set configuration during the first 11 seconds and then all-cores configuration for the remaining

time. This results in improved performance of 8% relative to the default OS scheduler (all-cores configuration).

Overhead of Cores Allocator. Since resource-usage information of whole *application* is monitored instead of individual *threads*, the overhead of Cores Allocator is negligible and it scales well. For n programs, (n)C(n-2) combinations are evaluated by PAAP. e.g., for 4 applications (A, B, C, D), 6 combinations (AB, AC, AD, BC, BD, CD) are evaluated. Cores Allocator takes a maximum of 2 milliseconds on our machine for selecting the best cores-configuration for coscheduling four applications.

Thus, using PAAP and PACC models, Cores Allocator adaptively allocates appropriate cores-configuration according to the resource-usage characteristics of the programs and effectively deals with the phase changes of the programs. In the next section, how the Policy Allocator adaptively selects appropriate memory allocation and processor scheduling policies based on the resource-usage characteristics of the programs is described.

6.2.2 The Policy Allocator

Contemporary operating systems such as Solaris and Linux do not distinguish between threads from multiple single threaded programs and multiple threads corresponding to a single multithreaded program. Though, the default OS scheduling and memory allocation policies work well for multiple single threaded programs, this not the case for multithreaded programs. This is because many multithreaded programs involve communication between threads, leading to contention for shared objects and resources. Since OS doesn't consider application level characteristics in scheduling and memory allocation decisions,

the default OS scheduling and memory allocation policies are not appropriate for achieving scalable performance for multithreaded programs. Most of the existing contention management techniques are primarily designed for single threaded programs and they only deal with allocating cores among the threads. To address this, ADAPT uses another component, the Policy Allocator, which is responsible for dynamically selecting appropriate memory-allocation and process scheduling policies based on programs resource-usage characteristics (application level scheduling instead of thread level).

Memory Allocation vs OS Load-balancing

As shown in Figure 4.2, the HyperTransport (HT) is used as the CPU interconnect and the path to the I/O controllers. Using HT, CPUs can access each other's memory, and any data transferred with the I/O cards travel via the HT. Effective utilization of HT on a NUMA machine is very important for achieving scalable performance for multithreaded programs, specifically for memory-intensive multithreaded programs as OS scheduler migrates threads across the CPUs for load balancing.

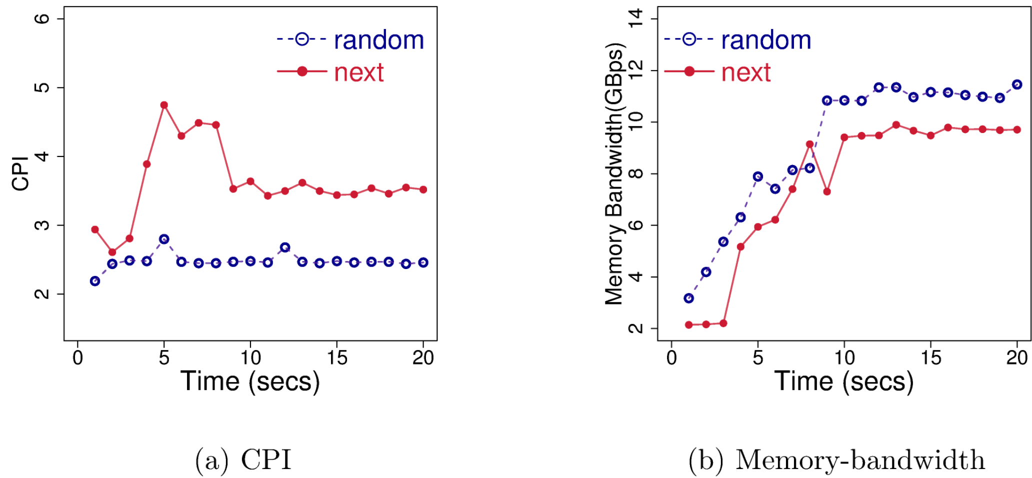
In Solaris 11, *next policy* (which allocates memory next to thread) is the default memory allocation policy for private memory (heap, stack) and *random policy* is the default memory allocation policy for shared memory when the size of shared memory is beyond the threshold value of 8MB. This threshold is set based on the communication characteristics of Message Passing Interface (MPI) programs [7]. Therefore, it is not guaranteed that the *random policy* will be always applied to the shared memory for multithreaded programs that are based on pthreads. If the shared memory is less than 8MB, then the *next* is also the

memory allocation policy for the shared memory. Moreover, with the default *next* policy, a memory-intensive thread can experience high memory latency overhead and consequently high cache miss-rate when it is started on one core and migrated to another core which is not in its home lgroup. More importantly, this creates HT as a performance-limiting hot spot. Therefore, the interaction between inappropriate memory allocation policy and OS load balancing degrades memory bandwidth and limits scalable performance for memory-intensive multithreaded programs.

Unlike *next policy*, *random policy* picks a random leaf lgroup to allocate memory for each page and it eventually allocate memory across all the leaf lgroups and then the threads of memory intensive programs get a chance to reuse the data in both private and shared memory. This reduces memory latency penalty and cache miss-rate. Moreover, it spreads the memory across as many memory banks as possible, distributing the load across many memory controllers and bus interfaces, thereby preventing any single component from becoming a performance-limiting hot spot[7]. This is demonstrated by running a very memory-intensive program streamcluster (SC) with both *next* and *random* policies in all-cores configuration.

In this experiment, SC is run with its corresponding OPT Threads (17) in all-cores configuration. Cycles per instruction (CPI) indicates whether HT and Memory buses are performance limiting spots or not. As shown in Figure 6.5, CPI of SC with *next-touch* is higher compared to *random policy*, and total memory bandwidth (GB/sec) is improved 17% with *random policy.* Therefore, random policy relieves pressure on HT and improves overall performance of memory-intensive programs as well as system utilization. Thus,

multithreaded programs with huge private memory benefit greatly from the *random policy*. Moreover, random policy not only improves performance, it also reduces performance variation of multithreaded programs.



(a) CPI

(b) Memory-bandwidth

Figure 6.5: CPI is high with next policy. Random policy improves memory-bandwidth.

Memory-allocation vs Access-latency of Locks

The performance of SC is dramatically improved by 56% with *random policy* compared to *next policy*. This improvement is not only because of the improved memory-bandwidth, there is also a reduction in lock-contention because of the allocation of private memory (heap and stack) randomly across lgroups. Allocating private memory across lgroups using *random policy* allows threads to quickly access lock-data structures in the shared cache, and thus minimizes memory traffic and delay loop time for acquiring locks. As shown in Figure 6.6, applying *random policy* for private memory dramatically reduces lock-contention of SC by 19% and improves performance.

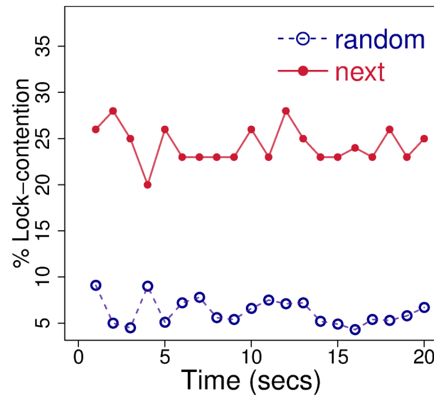


Figure 6.6: Random policy reduces lock-contention.

Scheduling Policy vs Lock-contention

The default Time Share (TS) scheduling policy is not appropriate for high lock-contention multithreaded programs under high loads as the interaction between TS policy and the state-of-the-art spin-then-lock contention management policy dramatically increases thread context-switch rate, and leads to drastic degradation in the performance [10, 20]. Both Load Controller [10] and FF policy were considered over TS policy for dealing with lock-contention of the programs in coscheduling running on multicore machines. However, since application is need to be modified for applying Load Controller and also its overhead increases linearly with the number of threads, the Policy Allocator selectively uses FF policy. By assigning same priority to all the threads of a multithreaded program, FF policy breaks the vicious cycle between thread priority changes and context-switches, dramatically reduces context-switch rate (CX-Rate), and improves performance. It is very effective under high loads. FF policy allocates time-quantum based on the resource-usage of the target multithreaded program for achieving fair allocation of CPU cycles among the threads. For

example, a high lock-contention program apsi is run with its OPT threads (16), FF policy reduces its CX-Rate (shown in Figure 6.7) and improves its performance by 9%.

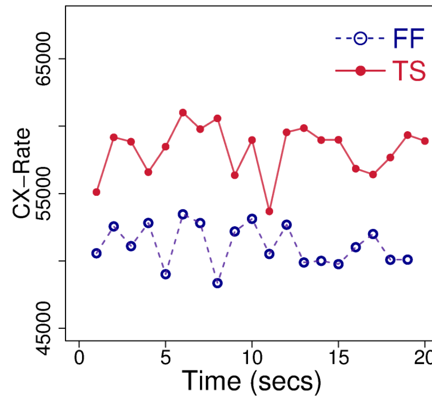


Figure 6.7: FF policy reduces context-switch rate of APSI.

Design of Policy Allocator

Policy Allocator continuously monitors cores-configuration selected by the Cores Allocator and the resource usage characteristics, MPA and Lock-contention, of the target multithreaded programs for selecting appropriate memory-allocation and scheduling policies. If the program is CPU-intensive and low lock-contention and it is in all-cores configuration, then the Policy Allocator applies *next* policy and TS policy. Otherwise, it applies *random* policy (or *random_pset* policy for processor-set configuration) and FF policy with appropriate time-quantum. Since, there will be interference between programs in all-cores configuration, Policy Allocator always applies one of the policies (TS or FF, but not both at a time) in *all-cores* configuration. This is because, running the programs with different scheduling policies (TS and FF) in all-cores configuration dramatically degrades overall performance (for some programs) is observed. However, both policies (TS and FF) are

applied at a time in processor-set configuration selectively according to the resource-usage characteristics of applications.

6.2.3 Implementation of ADAPT

Our implementation of ADAPT uses a daemon thread for continuously monitoring programs, maintaining programs resource-usage characteristics, assigning cores-configuration (using Cores Allocator), and selecting memory allocation and scheduling policies (using Policy Allocator). It maintains one resource-usage vector (RSV) per program. RSV of each program contains the following resource-usage characteristics of the program: *usr, lock, lat, sys, ct, mpa*, *CPU utilization* per processor-set if programs are coscheduled in processor-set configuration, and *cores-configuration* selected by the Cores Allocator. Based on the cores-configuration, cores-to-threads (*ct_a*) ratio is correctly interpreted as either *ct_a* for the PAAP model or *rct_a* for the PACC model (see Section 6.2.1).

For monitoring programs and collecting resource-usage data, and assigning different cores-configurations, memory allocation and scheduling policies, ADAPT uses the following Solaris 11 utilities: *prstat, mpstat, priocntl, pmadvise, mdb*, and *cputrack*. *prstat* is used to collect *usr, lock, lat,* and *sys* characteristics, while cputrack is used to collect *mpa.* mpstat collects system-wide resource-usage characteristics such as overall system utilization and CPU utilization per processor-set. *mdb, pmadvise* are used to apply memory allocation policies and *priocntl* is for scheduling policies. While *mdb* is used to apply memory allocation policies system-wide for all programs, *pmadvise* is used for applying memory-allocation policy per program.

Moreover, it is difficult to respond to rapid phase changes of the multithreaded programs, because The minimum timeout value with the default implementation of prstat(1) and mpstat(1) utilities is one second time-interval. Therefore, these utilities were enhanced to allow time intervals with millisecond resolution to capture phase changes of multithreaded programs. Furthermore, the default cpustat utility does not support the use of performance monitoring events to collect system-wide resource-usage characteristics such as last-level cache miss-rate if there are more than one processor-set present in the system. Therefore, *cpustat* utility was also enhanced to collect system-wide characteristics on a multicore system with arbitrary number of processor-sets.

Selecting Appropriate Monitoring Time-interval. Using the above enhanced utilities, ADAPT collects resource-usage characteristics of the target programs with millisecond resolution to respond to rapid phase changes present in the programs. However, using appropriate time-interval for collecting resource-usage data is very important as it has significant impact on the overhead of ADAPT. Though very small time-interval allows to collect very fine-grain details of the resource-usage data, it increases system overhead. Therefore selecting appropriate time-interval is very important. For this, as shown in Figure 6.8, ADAPT is evaluated with different time-intervals for monitoring four multithreaded programs simultaneously running on our machine. As Figure 6.8 shows, When ADAPT is used with 50 ms and 100 ms time-intervals, the system overhead is considerably high. Using very small monitoring time-interval increases the rate of interprocessor interrupts and cross-calls and consequently leads to high system time [7]. With 200 ms and bigger time-intervals

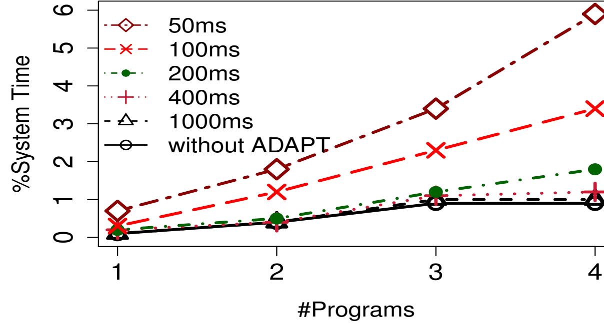


Figure 6.8: Size of time-interval vs System overhead.

the overhead of ADAPT is negligible (< 1.5% of system time) and also its impact on the performance of the programs is not considerable. Therefore, ADAPT uses 200 ms time-interval for collecting RSVs of the multithreaded programs. ADAPT collects 10 samples of the resource-usage data of the target programs with 200 ms time-interval and updates RSVs of programs with the average of these 10 samples every two seconds. Therefore, every two seconds, based on the phase changes, it applies appropriate cores-configuration, memory allocation, and scheduling policies.

However, more frequent changes in *cores-configuration* diminish the benefits of ADAPT is observed. ADAPT keeps track of the last three RSVs of each program and decides whether to change cores-configuration if one of the following conditions is satisfied:

1. If programs are running in processor-set configuration, and average CPU utilization of any processor-set is less than that of any other processor-set by a threshold of α.

2. For any program P_i, if its usr_Pi decreases at a rate greater than of a threshold β in the last two intervals.

3. $\frac{\sum_{i=1}^{N}(usr_P_i^P)}{N} > (\frac{\sum_{i=1}^{N}(usr_P_i^C)}{N} + \gamma)$ in the last two-intervals, where γ is the threshold value.

where usr_P_i^C is the actual %USR of P_i in current cores-configuration, while usr_P_i^P is the predicted %USR of program P_i using either PAAP or PACC model based on the current cores-configuration. From extensive experimentation with the programs used in this work, the threshold values α, β, and γ as 6%, 4%, and 8% were derived. By employing these thresholds, the influence of switching back and forth between cores configurations on the performance of the programs can be reduced. In the current implementation of ADAPT, the assumption used is that solo run RSVs of the target programs are available. Alternatively, the applications can be run for a few milliseconds and collect their RSVs. Using signals (SIGSTOP/SIGSTART) other applications can be paused while collecting RSVs.

6.3 Evaluating ADAPT

Our experimental setup consists of a 64-core machine running Solaris 11. ADAPT is evaluated with a wide variety of benchmarks. A total of 26 programs are used -- TATP [12] database transaction application; SPECjbb2005 [11]; eight programs streamcluster (SC), facesim (FS), canneal (CA), x264 (X264), fluidanimate (FA), swaptions (SW), ferret (FR), bodytrack (BT) from PARSEC [1]; 11 programs swim (SM), equake (EQ), wupwise (WW), gafort (GA), art (ART), apsi (AS), ammp (AM), applu (AP), fma3d (FMA), galgel, (GL), mgrid (MG) from SPEC OMP [11]; and five programs kmeans (KM), pca (PCA), matrix-multiply (MM), word-count (WC), string-match (STRM) from Phoenix [21].

6.3.1 Performance and System Utilization Improvements

Inspired by [22], two metrics are used to evaluate ADAPT: 1) user-oriented metric: average total turn-around time (TTT); 2) system-oriented performance metric: average system utilization, where system utilization = 100 − (%CPU idle time).

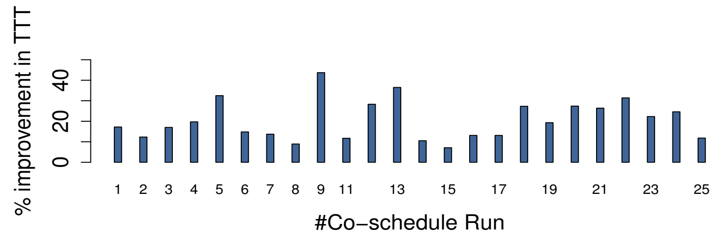
ADAPT is evaluated by coscheduling either two, three, or four programs as shown in Figure 6.9(a). As shown in Figure 6.9(b), TTT improvement with ADAPT is on average 21% (average lies between 16.1% and 25.2% with 99% confidence interval) and up to 44% relative to the default Solaris 11 scheduler.

As shown in Figure 6.9(b), while ADAPT achieves high TTT improvements for the coscheduled runs of memory-intensive and high lock-contention programs (e.g. FS), it achieves moderate TTT improvements for the coschedule runs of CPU-intensive and low lock-contention programs (e.g. SW). ADAPT achieves high throughput improvements for the coscheduled run of TATP database transaction application and JBB. ADAPT improves throughput of TATP and JBB by 23.7% and 18.4% compared to the default Solaris scheduler. For CPU-intensive and low lock-contention programs, Policy Allocator contributes more to the improvements in TTT than the Cores Allocator, because Cores Allocator allocates all-cores configuration like the default OS scheduler for these programs. Figure 6.9(c) shows that ADAPT achieves high system utilization, compared to the default Solaris scheduler.

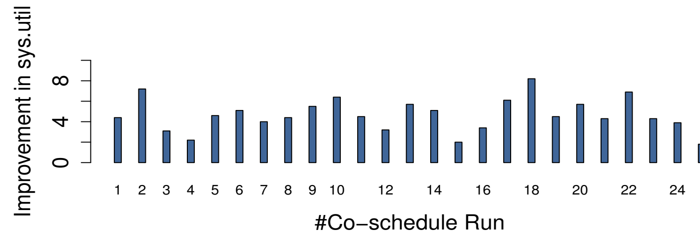
Moreover, since the existing coscheduling algorithms [4, 5, 6] are based on cache usage, they are very effective for a mix of workloads where half of the threads are memory-intensive and other half are CPU-intensive. However, they may not work well on a mix of

#	Programs	OPT Threads	#	Programs	OPT Threads
1	FS;EQ	(32,32)	9	AM;SC	(56,17)
2	BT;AP	(50,24)	10	TATP;JBB	(54,69)
3	AS;FMA	(16,56)	11	SM;EQ	(32,32)
4	SW;MG	(73,16)	12	PCA;STM	(48,16)
5	FA;SC	(49,17)	13	MG;PCA	(16,48)
6	GL;BT	(16,50)	14	KM;ART	(24,40)
7	FA;SM	(49,32)	15	MM;SW	(64,73)
8	ART;x264	(40,68)	16	WC;MG	(48,16)
17	KM;CA;X264	(16,33,68)	20	FS;AP;STM	(32,24,16)
18	AS;BT;FR	(16,50,83)	21	AM;WW;PCA	(56,24,48)
19	GA;SM;FA	(64,32,24)			
22	FS;SC;EQ;WW	(32,17,32,24)	24	PCA;AM;AS;ART	(48,48,34,40)
23	AS;AP;BT;FMA	(16,24,50,56)	25	GA;FMA;x264;FA	(64,56,68,49)

(a) Coschedule run numbers and corresponding programs.



(b) % Improvement in TTT.



(c) Improvement in System Utilization.

Figure 6.9: ADAPT improves TTT and system utilization compared to the default Solaris scheduler. Here, improvement in system utilization = (utilization with ADAPT - utilization with Solaris).

workloads where all the threads are either CPU-intensive or Memory-intensive. Therefore, ADAPT was also evaluated against a mix of four Memory-intensive multithreaded programs (FS:SC:EQ:WW) and as well as against a mix of four CPU-intensive multithreaded programs (AS:AP:BT:FMA). Like the above experiments, this experiment was repeated for 10 times. Figures 6.10 and 6.11 show the total running-times (or TTT) of the four programs in each run. As shown in these figures, ADAPT significantly improves performance of both the memory intensive and the CPU intensive programs.

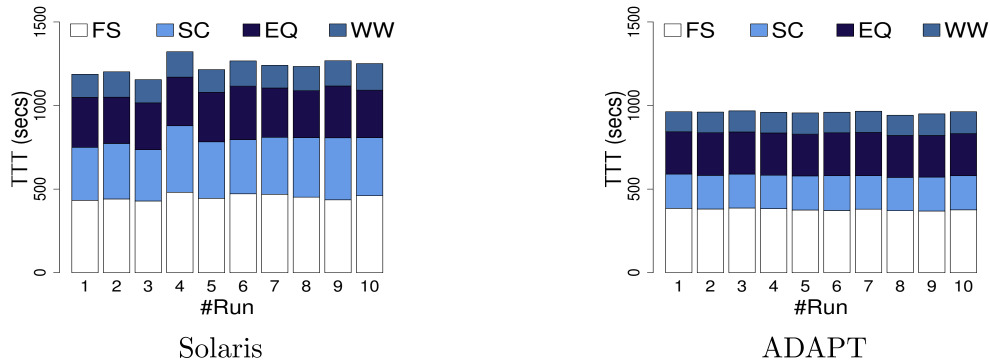


Figure 6.10: ADAPT improves performance of all the four memory-intensive programs.

6.3.2 Impact on Performance Variation

As Section 6.2 describes, for memory-intensive and high lock-contention programs, ADAPT simultaneously improves memory bandwidth and reduces lock-contention. It relieves pressure on HT module, and consequently reduces paging activity and improves performance. In the second coscheduled run of four programs, ADAPT was evaluated for a mix of four CPU-intensive and high lock-contention programs. By assigning appropriate cores-configuration and the FF scheduling policy with appropriate time-quanta, ADAPT dramatically reduces context-switch rate improves overall TTT of the programs. Furthermore, as shown in Figures 6.10 and 6.11, ADAPT not only improves performance of programs and

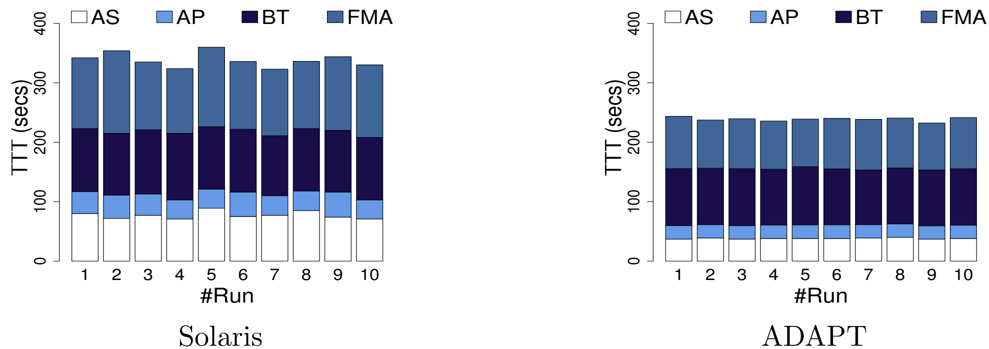


Figure 6.11: ADAPT improves performance of all the four CPU-intensive programs.

also simultaneously reduces variation in their performance.

6.4 Summary

This chapter presented ADAPT, a framework for effective coscheduling multithreaded programs on multicore systems. ADAPT is based on supervised learning techniques for learning the effects of the interference between multithreaded programs on their performance. It uses simple modern OS performance monitoring utilities for continuously monitoring the resource-usage characteristics of the target programs, adaptively allocating resources such as cores, and appropriate memory allocation and scheduling policies. ADAPT achieves up to 44% improvement in turnaround time and also improves throughput of TATP and JBB by 23.7% and 18.4% relative to the default Solaris 11 Scheduler.

Chapter 7

Related Work

In this chapter the work performed on improving the performance of multithreaded programs on multicore systems is summarized. The prior work is organized into four parts. The first two sections study the techniques used to configure application and system settings. The next two sections summarize variety of solutions developed for minimizing the impact of high lock contention on performance and effective scheduling of threads to maximize performance.

7.1 Balancing Parallelism and Resource Usage

While higher degrees of parallelism can be exploited by creating greater number of threads and distributing the workload among these threads, the number of threads that provide the best speedup depends upon many additional factors. In particular, the performance of a multithreaded program running on a multicore system is also sensitive to the program's resource-usage characteristics which in turn varies with the number of threads.

7.1.1 One Thread Per Core Model

It is often assumed that to maximize performance the number of threads should equal the number of cores [1, 23, 24, 25, 26]. On a machine with few cores, one-thread-per-core model which binds one thread with each core may slightly improve performance; but this is not true for machines with larger number cores, specifically NUMA multicore systems. The problem with binding is that bounded thread may not be able to run promptly and there is no other thread to take its place. Therefore, dynamically finding a suitable number of threads for a multithreaded application to optimize the use of system resources in a multicore environment is an important problem.

7.1.2 Dynamically Determining Number of Threads

In [27], Lee et al. show how to adjust number of threads in an application dynamically to optimize system efficiency. They develop a runtime system called "Thread Tailor" which uses dynamic compilation to combine threads based on the communication patterns between them in order to minimize synchronization overhead and contention of shared resources (e.g., caches). They achieve performance improvements for three PARSEC programs on quad-core and 8-core systems. However, they used a baseline of number of threads equals the number of cores (4 or 8) for performance comparisons and they did not present the optimal number of threads resulting from their technique.

To improve performance and optimize power consumption for OpenMP based multithreaded workloads, Suleman et al.[28], proposed a framework that dynamically controls number of threads using runtime information such as memory bandwidth and synchroniza-

tion. They show that there is no benefit of using larger number of threads than the number of cores. Similarly, Nieplocha et al. [29] demonstrate that some applications saturate shared resources as few as 8 threads on an 8-core Sun Niagara processor. Thus once again the above works considered small number of cores and used one-thread-per-core binding model. On a machine with few cores, one-thread-per-core model may slightly improve performance; but this is not true for machines with larger number cores.

Jung et al. [30], presented performance estimation models and techniques for generating adaptive code for quad-core SMT multiprocessor architectures. The adaptive execution techniques determine an optimal number of threads using dynamic feedback and runtime decision runs. Similarly, Kunal et al. [31] proposed an adaptive scheduling algorithm based on the feedback of parallelism in the application. Many other works that dynamically control number of threads are aimed at studying power performance trade-offs [32, 33, 34, 24, 35].

Unlike the above works, this dissertation demonstrated that on a system with large number of cores, many applications can benefit from using more threads than the number of cores. Furthermore this dissertation presented a simple technique, *Thread Reinforcer* [36], for dynamically determining appropriate number of threads without recompiling the application or using complex compilation techniques or modifying OS policies on a 24-core machine.

7.2 Configuring System Policies

The performance of a multithreaded program is sensitive to the OS scheduling and memory allocation policies. This is because the interactions between program memory reference behavior and the OS scheduling and memory allocation policies make the performance of a program highly sensitive to small changes in resource usage characteristics of the program. In particular, significant variations in the performance are observed from one execution of a program to the next, even when the program input remains unchanged and no other programs are being run on the system.

7.2.1 Study on Performance Variation

Many researchers have studied performance variability of parallel applications on large-scale parallel computers. Mraz et al., examined the effect of variance in message passing communications introduced by the AIX Operating System in parallel machines built of commodity work stations [37]. They mainly considered the variance introduced by the interrupts of the AIX Operating System and concluded that globally synchronizing the system clocks gives the best results overall as it generally caused the daemons to run in a coscheduled fashion and did not degrade system stability. Petrini et al. [38] showed that for large-scale parallel computers, OS noise such as periodically monitoring I/O, could cause serious performance problems. The techniques they proposed to eliminate system noise is to turn off unnecessary system daemons, moving heavyweight daemons to one node instead of spreading them across multiple nodes. In [39], Gu et al., demonstrated that a significant source of noise in benchmark measurement in a Java Virtual Machine is due to code layout.

Like [40], Skinner et al. [41] showed that variability in performance is inherently tied to contention for resources between applications and operating system. [42] also studies OS noise on the performance of MPI based NAS applications running on a quad-core machine, and it provides design and implementation of a scheduler to optimize performance by dynamically binding threads to cores and thus minimizing thread migrations. This dissertation identified the reasons for performance variation of multithreaded programs running on NUMA multicore systems. The reason is that the interactions between program memory reference behavior and the OS scheduling and memory allocation policies make the performance of a program highly sensitive to small changes in resource usage characteristics of the program.

7.2.2 Reducing Performance Variation

Alameldeen et al. [43] provided a methodology to compensate for variability, that combines pseudo-random perturbations, multiple simulations and standard statistical techniques; and [44] used page coloring and bin hopping page allocation algorithms to minimize performance variation. Touati et al. [45] proposed a statistical methodology called 'Speedup-Test' for analyzing the distribution of the observed execution times of benchmark programs and improving the reproducibility of the experimental results. While [45] focuses on statistical methodology for enhancing performance analysis methods to improve the reproducibility of the experimental results for SPEC CPU2006 and SPEC OMP 2001, Thread Tranquilizer [46] reduces performance variation of multithreaded workloads by applying appropriate scheduling and memory allocation policies.

7.2.3 NUMA Optimization Techniques

Verghese et al. [47] developed page migration and replication policies for ccNUMA systems and showed that migration and replication polices improved memory locality and reduced the overall stall time. [48] presents several policies for cluster-based NUMA multiprocessors that are combinations of a processor scheduling scheme and a page placement scheme and investigates the interaction between them through simulations. McCurdy et al. [49] introduced Memphis, a data-centric tool set that uses Instruction Based Sampling to help pinpoint problematic memory accesses to locate NUMA problems in some NAS parallel benchmarks.

This dissertation presented a technique, Thread Tranquilizer [46], to simultaneously reduce performance variation and improve performance of a multithreaded program running on a multicore system through proper choice of memory management and scheduling policies. Users can easily apply these techniques on the fly and adjust OS environment for multithreaded applications to achieve better performance predictability.

7.3 Lock Contention

7.3.1 Synchronization Mechanisms

The performance of a multithreaded program is often impacted greatly by lock contention on a ccNUMA multicore system. Therefore the key to achieving high performance for multithreaded applications running on multicore systems is to use appropriate synchronization primitives along with efficient lock contention management policies. Con-

tention management policies are either based on spinning, or blocking, or a combination of both. Spinning resolves contention by busy waiting, therefore waiting threads respond to lock handoffs very quickly. However, spinning threads can wastes CPU resources and prevent the lock-holder thread from running and releasing the lock [7, 10]. This dramatically degrades performance and becomes a prominent problem in systems under high load conditions. In contrast, the blocking scheme reschedules waiting threads and allows other threads to use the system resources. However, blocking scheme increases context-switches, overloads OS scheduler, and thus leads to poor performance [10, 7].

Lock Algorithms. The problems with spinning have prompted many approaches, such as queue-based spinlocks [50, 51] and ticket spinlocks [52], to alleviate these problems. Both Queue-based spinlocks and ticket spinlocks provide an efficient way of orderly lock-handoffs because waiting threads form a FIFO queue and each lock handoff targets a specific thread. However, they also suffer from lock-holder thread preemptions at high load and create lock convoys [53, 10]. Time-published locks [54] eliminate the main problem with queue-based locks by only handing the lock to running threads. However, these also allow lock holders to be vulnerable to preemption [10]. By limiting the number of waiting threads which can respond simultaneously, backoff-based techniques [55, 56] provide another solution to the "thundering herd" problem [10], where all waiting threads race for the lock at each release and cause both contention and memory traffic. However, finding optimal backoff length for the general case is a challenging problem. Hybrid spin-then-block technique [7, 57] use spinning to reduce context switching imposed by a blocking primitive. However, these

also face challenges to provide optimal balance between spinning and blocking as load increases [58]. In [59] and [60] authors propose NUMA-ware locking mechanisms.

Barrier Algorithms. A software barrier synchronizes a number of cooperating threads that repeatedly perform some work and then wait until all threads are ready to move to the next computing phase. It is a well-known fact that the choice of a barrier algorithm is critical to the performance of any library that supports the fork-join programming model since each join action leads to executions of the barrier synchronization among all threads [61].

There are a few popular barrier algorithms used over the years, such as the centralized barrier, the tournament barrier [50] and so on. The centralized barrier works well for a small number of threads but does not scale well for a large number of threads because all threads contend for the same set of variables. The combining tree and the tournament barrier reduce the above contention and work best for a large SMP system but not particularly well for a small SMP system [61]. Recently, the queue-based barrier algorithm [62] has gained popularity because it reduces the contention, performs well for small and large SMP systems and is easy to implement. In [61], authors studied the impact of NUMA on the performance of different barrier synchronization algorithms.

The above are efficient locking and barrier synchronization mechanisms to reduce lock contention. Unlike these, this dissertation presented techniques to reduce the negative effect of high lock contention.

7.3.2 Reducing Lock Acquisition Overhead.

The performance of a multithreaded program is highly sensitive to the distribution of program threads across the multiple multicore CPUs on a ccNUMA multicore systems. In particular, when multiple threads compete to acquire a lock, due to the NUMA nature of the architecture, the time spent on *acquiring locks* by threads distributed across different CPUs is greatly increased. To address this problem, three specific approaches proposed in prior work include: *thread migrations*[63] and *thread clustering* [64, 65, 66].

Thread Migration Techniques. The authors of [63] make the observation that in NUMA systems it may be beneficial to employ thread migration to reduce the execution time cost due to acquiring of locks. They propose a thread shuffling technique that is implemented by modifying the OS thread scheduler. While this technique performs well for a few microbenchmarks [63], for majority of SPLASH2 programs the technique frequently yielded large performance degradation.

Thread Clustering Techniques A number of thread clustering techniques have been developed to improve program performance [64, 66]. Of these, the technique in [64] is aimed at reducing the overhead caused by lock contention. This is achieved by clustering threads that contend for the same lock and then schedule them on the same processor. The number of threads in a cluster can be large, in fact all threads in an application will be in the same cluster if they are synchronizing on a barrier. Thus, when the entire cluster is scheduled on the same processor, load is no longer balanced, and parallelism is sacrificed. In contrast, this dissertation presented *thread shuffling* technique to maintain load balance and thus do

not sacrifice parallelism. Finally, while the approach presented in [64] is effective for server workloads, *thread shuffling* is more relevant for highly parallel applications such as those in PARSEC and SPEC OMP programs.

7.3.3 Reducing Critical Section Delays

Another cause of performance degradation is *lock-holder* thread preemptions as they slow down the progress of thread holding the lock. This problem become serious under high load conditions. Johnson et al.[10] proposed a load control mechanism that decouples load management from lock contention management for reducing lock holder preemptions. This approach uses blocking to control the number of runnable threads and then spinning in response to contention. In contrast, this dissertation presented *FaithFul Scheduling* [20] to reduce the problems caused by the unwanted interactions between OS scheduling policy and lock contention management policy.

7.4 Thread Scheduling

7.4.1 Operating System Scheduling

One of the core functions of any modern multitasking operating system is the management and scheduling of runnable threads on the available processors. The kernel's primary goal is to maintain fairness: allowing all threads to get processor cycles while ensuring that critical work, such as interrupt handling, gets done as needed. This is the function of the kernel dispatcher--selecting threads and dispatching them to available system processors. The dispatcher code attempts to keep the length of the run queues closely

balanced so that no one CPU has an inordinate number of threads on its queue relative to the other CPUs. For this, it needs to migrate threads across CPUs.

Though the above works well for multiple single threaded programs, this is not the case for multithreaded programs. This is because multithreaded programs involve communication and data sharing among threads. In the multicore era, an application relies on increased concurrency to maximize its performance, which often requires the application to divide its work into small tasks. To efficiently distribute and execute these tasks on multicores, fine-grained task manipulation and scheduling must be adopted [67]. To address this, work stealing technique is proposed for managing and scheduling concurrent tasks of multithreaded programs on multiprocessor systems [68].

7.4.2 Work Stealing

Work stealing is used in user mode where the unit of work is typically a "task". In work stealing, the application spawns multiple worker threads and distributes the tasks dynamically among its workers with a user-level task scheduler. Workers execute tasks from their local task queue. Any newly spawned tasks are also added to the local queue. When a worker runs out of tasks, it steals a task from another workers queue and executes it. That is, threads are assumed to cooperate. In contract, in the classic OS scheduling the threads do not cooperate. For example, classic pthreads assumes preemptive scheduling with the kernel handling scheduling. In the case of classic pthreads, "scheduling" can be taken to mean the policies that determine when (and for how long) ready threads should be dispatched to the CPUs in the system.

However, thread scheduling in multicore systems is still a challenging problem because cores on a single chip usually share parts of the memory hierarchy (e.g., last-level caches, prefetchers, and memory controllers) making threads running on different cores interfere with each other while competing for these resources. To address this issue, the following coscheduling techniques have been proposed in prior work.

7.4.3 Thread Coscheduling

Cache contention aware scheduling: single threaded programs. Previous work observed that on modern Intel and AMD systems the degree of contention for shared resources can be explained by the relative memory-intensity of threads that share resources [4, 6, 5]. In that work, threads were classified as memory-intensive or CPU-intensive. Memory-intensity was approximated by a thread's relative last-level cache misses per instruction (MPI): memory-intensive threads have a higher MPI than the CPU-intensive threads. That work found that in order to significantly reduce contention for shared resources the scheduler must run memory-intensive threads (those with high MPI) on separate processor-sets.

Snavely et al. [69] proposed a scheduling algorithm for reducing contention for various architectural resources on a simultaneous multithreading processor. Their technique samples some of the possible thread assignments and then builds a predictive model according to the observation. Likewise, FACT [6] trains a statistical model to predict contention for memory resources between threads based on performance monitoring events recorded by the OS for a list of possible events). The trained model is then used to dynamically schedule together threads that interfere with each other's performance as little as possible.

Several other works [70, 71, 72, 3, 73, 74, 75, 76] showed various scheduling techniques using cache usage characteristics of applications that dynamically estimate the usage of system resources and then optimize performance or power or both.

However, these techniques are primarily designed for either coscheduling multiple single threaded programs or for coscheduling threads of a single multithreaded program.

Cache contention aware scheduling: multiple multithreaded programs. Bhauaria et al. [23] proposed a symbiotic scheduler which is based on memory-hierarchy contention factors such as last-level cache miss-rate for coscheduling multithreaded programs on a machine with a small number of cores (eight core machine). The goal of their coscheduling technique is to balance power and performance. However, as this dissertation demonstrated, cache-usage is not enough for coscheduling multithreaded programs on machines with large number cores, and therefore other characteristics such as lock-contention and latency also should be considered.

Balanced Work Stealing. Though the classic work stealing scheduler works well for a single multithreaded program on multicore systems, it is not effective when multiple applications time-share a single multicore system [77]. Ding et al., [77] showed that the state-of-the-art work stealing schedulers suffer from both system throughput and fairness problems when running multiple multithreaded programs on multicore systems. An underlying cause is that the operating system has little knowledge on the current roles of the threads, such as whether a thread is (i) working on an unfinished task (a busy worker), (ii) attempting to steal tasks when available tasks are plentiful (a useful thief), or (iii) attempt-

ing to steal tasks when available tasks are scarce (a wasteful thief). As a result, wasteful thieves can consume resources that should have been used by busy workers or useful thieves. Existing work-stealing schedulers try to mitigate this problem by having wasteful thieves yield their cores spontaneously. However, such yielding often leads to significant unfairness, as a frequently yielding application tends to lose cores to other concurrent applications. Moreover, system throughput suffers as well because the yielded core may fail to go to a busy worker or may be switched back to the wasteful thief prematurely.

To address this problem, Balanced Work Stealing technique (BWS) is proposed [77]. It improves both system throughput and fairness using a new approach that minimizes the number of wasteful thieves by putting such thieves into sleep and then waking them up only when they are likely to be useful thieves. Therefore, useful thieves become busy workers as soon as they successfully steal a task. Moreover, in BWS a wasteful thief can yield its core directly to a busy worker for the same application, so as to retain the core for that application and put it to better use.

In contrast, this dissertation presented a cache contention-aware coscheduling technique, *ADAPT*, for multiple multithreaded programs based on pthreads. The main problem solved by ADAPT is the selection of cores configuration to be employed to benefit the performance of all coscheduled multiple multithreaded applications.

Thread Clustering In [66] authors examine thread placement algorithms to group threads that share memory regions together onto the same processor so as to maximize cache sharing and reuse. It is assumed that the shared-region information is known a priori, i.e. this

information is not ascertained dynamically. In [65], Tam et al. propose a thread clustering technique to detect shared memory regions dynamically and evaluate it on a quad core system. However, these works did not address coscheduling of multithreaded programs.

7.4.4 Other Scheduling Techniques

Several researchers [78, 79, 80, 81, 82, 83] provided NUMA-related optimization techniques for efficient co-location of computation and related memory on the same node. However, they are not addressed resource contention management in multicore machines. Likewise, [84] developed adaptive scheduling techniques for parallel applications based on MPI on large discrete computers, but their scheduling techniques do not address the contention of shared resources when coscheduling multiple programs concurrently. Corbalan et al. [85] use techniques to allocate processors adaptively based on program efficiency. However, as the above works, this also does not consider resource contention among the programs. McGregor et al. [86] developed coscheduling techniques using architectural factors such as cache resource usage for coscheduling NAS parallel benchmarks on a quad core machine. However, each of the workload used in this work is either a single threaded or a multithreaded with only two threads. Like the above existing contention-management techniques, this technique also will not work for coscheduling multithreaded programs on large multicore machines. Gupta et al. [87] explored the impact of the scheduling strategies on the caching behavior of the applications. Likewise, Chandra et al. [88], evaluated different scheduling and page migration policies on a CC-NUMA multiprocessor system.

Chapter 8

Conclusions

8.1 Contributions

This dissertation makes contributions in the area of runtime techniques for maximizing performance of multithreaded programs on multicore systems. It develops lightweight runtime techniques to monitor important resource-usage characteristics of multithreaded programs, understand the interactions between OS policies and execution behavior of the programs, and then adaptively assign appropriate number of threads, appropriate number of cores, OS scheduling policies, and memory allocation policies.

The contributions of this thesis are divided into three parts: 1) It presents runtime monitoring techniques to select the configuration under which a program performance is expected to be high. The configuration includes factors such as, number of threads, the scheduling policy, and the memory allocation policy; 2) As a program executes under the selected configuration, it presents techniques to minimize the harmful impact of

high lock contention on program performance; 3) It presents runtime techniques for effectively coscheduling multiple multithreaded programs being simultaneously run of multicore systems.

8.1.1 Selecting Configuration for Delivering Performance

The performance of a multithreaded program running on a multicore system is sensitive to the number of threads used to run a multithreaded program (i.e., *threads configuration*), as it impacts the application resource-usage characteristics. Therefore, for getting best performance of a multithreaded program, it should be run with a suitable number of threads. However, an uninformed user may select too few or too many threads for execution and thus achieve suboptimal performance. An attractive technique for solving this is to dynamically determining a suitable number of threads for a multithreaded program. This dissertation presents such runtime technique called Thread Reinforcer [36], which monitors important application characteristics at runtime to guide the search for determining appropriate number of threads that are expected to yield the best speedup.

Moreover, the performance of a multithreaded program is sensitive to the OS scheduling and memory allocation policies. This is because the interactions between program memory reference behavior and the OS scheduling and memory allocation policies significantly impacts application performance. These interactions make the performance of a program highly sensitive to small changes in resource usage characteristics of the program. In particular, significant variations in the performance are observed from one execution of a program to the next, even when the program input remains unchanged and no other

applications are being run on the system. To address this, this dissertation presents a runtime technique called Thread Tranquilizer [46], which simultaneously reduces performance variation and improves performance by adaptively choosing appropriate memory allocation and process scheduling policies according to the important resource usage characteristics of the programs.

8.1.2 Dealing with Performance Impact of Lock Contention

On a ccNUMA system, the performance of a multithreaded application is often impacted greatly by lock contention. This dissertation considers two reasons for high lock contention: 1) On a ccNUMA system the performance of a multithreaded application is highly sensitive to the distribution of application threads across the multiple multicore CPUs. In particular, when multiple threads compete to acquire a lock, due to the NUMA nature of the architecture, the time spent on *acquiring locks* by threads distributed across different CPUs is greatly increased; 2) Under high load conditions, frequent preemption of lock holder threads can slow down the progress of lock holder threads and increase lock times. In particular, negative interaction between the time share (TS) thread scheduling policy and the spin-then-block lock-contention management policy dramatically increases lock holder thread preemptions under high loads.

To address the above problems, this dissertation presents two techniques *Thread Shuffling* and *Faithful Scheduling* [20]. *Thread Shuffling* minimizes the times threads spend on acquiring locks through inter-CPU thread migrations and *Faithful Scheduling* [20] minimizes lock holder thread preemptions via adaptive time-quanta allocations.

8.1.3 Coscheduling Multiple Multithreaded Programs

Since the performance of multithreaded applications often does not scale to fully utilize the available cores in a multicore system, simultaneously running multiple multithreaded applications becomes inevitable to fully utilize such machines. However, coscheduling multithreaded programs effectively on such machines is a challenging problem because of their complex architecture. For effective coscheduling of multithreaded programs, the OS must understand the resource-usage characteristics of multithreaded programs and then adaptively allocate cores as well as select appropriate memory allocation and scheduling policies.

To address this problem, this dissertation presents a runtime technique called *ADAPT*. It uses supervised learning techniques for predicting the effects of interference between programs on their performance and adaptively schedules together programs that interfere with each other's performance as little as possible. It achieves high throughput, high system utilization, and fairness when running multiple multithreaded applications.

8.2 Future Directions

8.2.1 Enhancing Scalability of Resource Usage Monitoring

Performance monitoring is crucial for understanding and isolating performance problems. This dissertation showed the significance of lightweight resource monitoring for enabling a variety of techniques that all help achieve high performance for multithreaded programs running on multicore systems. However, collecting monitoring data on a machine

with large number of cores (e.g., 1000 cores) is a challenging problem as the performance monitoring process it self could take significant amount of time to complete. Moreover, the performance monitoring process could significantly impact the performance of programs running on such systems. Therefore it is important that future work develop simple scalable monitoring techniques for machines with large number of cores.

8.2.2 Using Monitoring for Runtime Power Management

This dissertation focuses on improving performance of multithreaded programs on multicore systems. However, in other domains, specifically for datacenters, optimizing power consumption is a very important problem. To address this problem, future work should develop runtime techniques based on simple resource usage monitoring for optimizing power and improving performance.

8.2.3 Monitoring for Fault Isolation and High Availability

Self-healing functionality for users and administrators of a modern operating system can provide fine-grained fault isolation and component restart capabilities. To do so, monitoring techniques that include intelligent, automated, and proactive diagnoses of errors are required. The diagnosis system can be used to trigger targeted automated responses or help guide human intervention that eliminates an observed problem or at least prevent it from getting worse. Therefore, future work should develop light resource usage monitoring techniques for improving fault tolerance managers of self-healing systems for achieving high system availability.

8.2.4 Monitoring and Coscheduling for Virtualized Systems

Server virtualization technologies help organizations create administrative and resource boundaries between applications. This approach provides improved application performance and security, and also can be a vehicle for rapid application provisioning by delivering pre-installed, pre-configured virtual machine images of enterprise software. Improving system utilization and optimizing power are major challenges in this area. In future work, the coscheduling techniques developed in this dissertation can be adapted for effective scheduling of virtual machines to improve system utilization and optimize power consumption.

Bibliography

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li, The parsec benchmark suite: characterization and architectural implications, in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008, ACM.

[2] R. McDougall, J. Mauro, and B. Gregg, *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*, Prentice Hall, 2006.

[3] A. Merkel, J. Stoess, and F. Bellosa, Resource-conscious scheduling for energy efficiency on multicore processors, in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 153–166, New York, NY, USA, 2010, ACM.

[4] S. Zhuravlev, S. Blagodurov, and A. Fedorova, Addressing shared resource contention in multicore processors via scheduling, in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 129–142, New York, NY, USA, 2010, ACM.

[5] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, A case for numa-aware contention management on multicore systems, in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011, USENIX Association.

[6] K. K. Pusukuri, D. Vengerov, A. Fedorova, and V. Kalogeraki, Fact: a framework for adaptive contention-aware thread migrations, in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 35:1–35:10, New York, NY, USA, 2011, ACM.

[7] R. McDougall and J. Mauro, *Solaris Internals, second edition*, Prentice Hall, USA, 2006.

[8] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, Dynamic instrumentation of production systems, in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004, USENIX Association.

[9] P. Sweazey and A. J. Smith, A class of compatible cache consistency protocols and their support by the ieee futurebus, in *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 414–423, Los Alamitos, CA, USA, 1986, IEEE Computer Society Press.

[10] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry, Decoupling contention management from scheduling, in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 117–128, New York, NY, USA, 2010, ACM.

[11] SPECOMP, 2001, http://www.spec.org/omp.

[12] TATP., IBM telecom application transaction processing benchmark description, 2003, http://tatpbench-mark.sourceforge.net.

[13] solidDB, IBM soliddb 6.5 (build 2010-10-04), https://www-304.ibm.com/support/docview.wss?uid=swg24028071.

[14] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek, Reinventing scheduling for multicore systems, in *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 21–21, Berkeley, CA, USA, 2009, USENIX Association.

[15] S. Peter et al., Design principles for end-to-end multicore schedulers, in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 10–10, Berkeley, CA, USA, 2010, USENIX Association.

[16] T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd ed.,*, Springer Series in Statistics, USA, 2009.

[17] R, lm(), stepaic(), prune(), vif(), rpart(), kknn()., http://www.statmethods.net/.

[18] M. Marasinghe, The American Statistician **61**, 95 (2007).

[19] smape, Symmetric mean absolute percentage error., http://monashforecasting.com/index.php?title=SMAPE.

[20] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, No more backstabbing... a faithful scheduling policy for multithreaded programs, in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 12–21, Washington, DC, USA, 2011, IEEE Computer Society.

[21] R. M. Yoo, A. Romano, and C. Kozyrakis, Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system, in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 198–207, Washington, DC, USA, 2009, IEEE Computer Society.

[22] S. Eyerman and L. Eeckhout, IEEE Micro **28**, 42 (2008).

[23] M. Bhadauria, V. M. Weaver, and S. A. McKee, Understanding parsec performance on contemporary cmps, in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 98–107, Washington, DC, USA, 2009, IEEE Computer Society.

[24] M. Bhadauria and S. A. McKee, An approach to resource-aware co-scheduling for cmps, in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA, 2010, ACM.

[25] N. Barrow-Williams, C. Fensch, and S. Moore, A communication characterisation of splash-2 and parsec, in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 86–97, Washington, DC, USA, 2009, IEEE Computer Society.

[26] E. Z. Zhang, Y. Jiang, and X. Shen, Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs?, in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 203–212, New York, NY, USA, 2010, ACM.

[27] J. Lee, H. Wu, M. Ravichandran, and N. Clark, Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications, in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 270–279, New York, NY, USA, 2010, ACM.

[28] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, SIGARCH Comput. Archit. News **36**, 277 (2008).

[29] J. Nieplocha et al., Evaluating the potential of multithreaded platforms for irregular scientific computations, in *Proceedings of the 4th international conference on Computing frontiers*, CF '07, pages 47–58, New York, NY, USA, 2007, ACM.

[30] C. Jung, D. Lim, J. Lee, and S. Han, Adaptive execution techniques for smt multiprocessor architectures, in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 236–246, New York, NY, USA, 2005, ACM.

[31] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson, Adaptive scheduling with parallelism feedback, in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 100–109, New York, NY, USA, 2006, ACM.

[32] Y. Ding, M. K, P. Raghavan, and M. J. Irwin, A helper thread based edp reduction scheme for adapting application execution in cmps, 2008.

[33] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, Online power-performance adaptation of multithreaded programs using hardware event-based prediction, in *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 157–166, New York, NY, USA, 2006, ACM.

[34] J. Li and J. F. Martínez, Dynamic power-performance adaptation of parallel computation on chip multiprocessors, in *International Symposium on High-Performance Computer Architecture (HPCA)*, Austin, TX, 2006.

[35] K. Singh et al., Comparing scalability prediction strategies on an smp of cmps, in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 143–155, Berlin, Heidelberg, 2010, Springer-Verlag.

[36] K. Pusukuri, R. Gupta, and L. Bhuyan, Thread reinforcer: Dynamically determining number of threads via os level monitoring, in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 116 –125, Austin, Texas, USA, 2011, IEEE Computer Society.

[37] Mraz and Ronald, Reducing the variance of point to point transfers in the ibm 9076 parallel computer, in *Proceedings of the 1994 conference on Supercomputing*, Supercomputing '94, pages 620–629, Los Alamitos, CA, USA, 1994, IEEE Computer Society Press.

[38] F. Petrini, D. J. Kerbyson, and S. Pakin, The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q, in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 53–65, New York, NY, USA, 2003, ACM.

[39] D. Gu, C. Verbrugge, and E. Gagnon, Code layout as a source of noise in jvm performance, in *In Component And Middleware Performance workshop, OOPSLA*, 2004.

[40] W. T. C. Kramer and C. Ryan, Performance variability of highly parallel architectures, in *Proceedings of the 2003 international conference on Computational science: PartIII*, ICCS'03, pages 560–569, Berlin, Heidelberg, 2003, Springer-Verlag.

[41] D. Skinner and W. Kramer, IEEE Workload Characterization Symposium **0**, 137 (2005).

[42] R. Gioiosa, S. A. McKee, and M. Valero, Cluster Computing, IEEE International Conference on **0**, 78 (2010).

[43] A. R. Alameldeen and D. A. Wood, Variability in architectural simulations of multi-threaded workloads, in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 7–22, Washington, DC, USA, 2003, IEEE Computer Society.

[44] M. Hocko and T. Kalibera, Reducing performance non-determinism via cache-aware page allocation strategies, in *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 223–234, New York, NY, USA, 2010, ACM.

[45] S.-A.-A. Touati and J. S. B. Worms, The speed test, Technical report, 2010, http://hal.archives-ouvertes.fr/inria-00443839.

[46] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, ACM Trans. Archit. Code Optim. **8**, 46:1 (2012).

[47] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, Operating system support for improving data locality on cc-numa compute servers, in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 279–289, New York, NY, USA, 1996, ACM.

[48] T. Koita, T. Katayama, K. Saisho, and A. Fukuda, J. Supercomput. **16**, 217 (2000).

[49] C. McCurdy and J. S. Vetter, Memphis: Finding and fixing numa-related performance problems on multi-core platforms, in *International Symposium on Performance Analysis of Systems and Software*, pages 87–96, 2010.

[50] J. M. Mellor-Crummey and M. L. Scott, ACM Trans. Comput. Syst. **9**, 21 (1991).

[51] P. S. Magnusson, A. Landin, and E. Hagersten, Queue locks on cache coherent multiprocessors, in *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171, Washington, DC, USA, 1994, IEEE Computer Society.

[52] D. P. Reed and R. K. Kanodia, Commun. ACM **22**, 115 (1979).

[53] M. Blasgen, J. Gray, M. Mitoma, and T. Price, SIGOPS Oper. Syst. Rev. **13**, 20 (1979).

[54] B. He, W. N. Scherer, and M. L. Scott, Preemption adaptivity in time-published queue-based spin locks, in *Proceedings of the 12th international conference on High Performance Computing*, HiPC'05, pages 7–18, Berlin, Heidelberg, 2005, Springer-Verlag.

[55] A. Agarwal and M. Cherian, Adaptive backoff synchronization techniques, in *Proceedings of the 16th annual international symposium on Computer architecture*, ISCA '89, pages 396–406, New York, NY, USA, 1989, ACM.

[56] A. Gupta, A. Tucker, and S. Urushibara, The impact of operating system scheduling policies and synchronization methods of performance of parallel applications, in *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '91, pages 120–132, New York, NY, USA, 1991, ACM.

[57] H. Bahmann and K. Froitzheim, SIGOPS Oper. Syst. Rev. **42**, 18 (2008).

[58] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein, J. Parallel Distrib. Comput. **21**, 246 (1994).

[59] D. Dice, V. J. Marathe, and N. Shavit, Flat-combining numa locks, in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011, ACM.

[60] D. Dice, V. J. Marathe, and N. Shavit, Lock cohorting: a general technique for designing numa locks, in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012, ACM.

[61] J. Chen and W. W. Iii, Multi-threading performance on commodity multi-core processors, in *In Proceedings of 9th International Conference on High Performance Computing in Asia Pacific Region (HPCAsia*, 2007.

[62] L. Cheng and J. B. Carter, Fast barriers for scalable ccnuma systems, in *Proceedings of the 2005 International Conference on Parallel Processing*, ICPP '05, pages 241–250, Washington, DC, USA, 2005, IEEE Computer Society.

[63] S. Sridharan, B. Keck, R. Murphy, S. Ch, and P. Kogge, Thread migration to improve synchronization performance, in *In Workshop on Operating System Interference in High Performance Applications*, 2006.

[64] F. Xian, W. Srisa-an, and H. Jiang, Contention-aware scheduler: unlocking execution parallelism in multithreaded java programs, in *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 163–180, New York, NY, USA, 2008, ACM.

[65] D. Tam, R. Azimi, and M. Stumm, Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors, in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 47–58, New York, NY, USA, 2007, ACM.

[66] R. Thekkath and S. J. Eggers, Impact of sharing-based thread placement on multithreaded architectures, in *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 176–186, Los Alamitos, CA, USA, 1994, IEEE Computer Society Press.

[67] B. Saha et al., SIGOPS Oper. Syst. Rev. **41**, 73 (2007).

[68] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, Thread scheduling for multiprogrammed multiprocessors, in *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998, ACM.

[69] A. Snavely and D. M. Tullsen, SIGARCH Comput. Archit. News **28**, 234 (2000).

[70] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, IEEE Micro **28**, 54 (2008).

[71] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations, in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 248–259, New York, NY, USA, 2011, ACM.

[72] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, SIGARCH Comput. Archit. News **39**, 283 (2011).

[73] S. Chen et al., Scheduling threads for constructive cache sharing on cmps, in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 105–115, New York, NY, USA, 2007, ACM.

[74] S. Cho and L. Jin, Managing distributed, shared l2 caches through os-level page allocation, in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 455–468, Washington, DC, USA, 2006, IEEE Computer Society.

[75] X. Zhang, S. Dwarkadas, and K. Shen, Towards practical page coloring-based multicore cache management, in *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 89–102, New York, NY, USA, 2009, ACM.

[76] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, Proc. VLDB Endow. **2**, 373 (2009).

[77] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang, Bws: balanced work stealing for time-sharing multicores, in *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 365–378, New York, NY, USA, 2012, ACM.

[78] T. Brecht, On the importance of parallel application placement in numa multiprocessors, in *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, pages 1–1, Berkeley, CA, USA, 1993, USENIX Association.

[79] R. P. LaRowe, Jr., C. S. Ellis, and M. A. Holliday, IEEE Trans. Parallel Distrib. Syst. **3**, 686 (1992).

[80] J. Corbalan, X. Martorell, and J. Labarta, Evaluation of the memory page migration influence in the system performance: the case of the sgi o2000, in *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 121–129, New York, NY, USA, 2003, ACM.

[81] VMware, Vmware esx server 2 numa support. white paper., Technical report, 2005, http://www.vmware.com/pdf/esx2_NUMA.pdf.

[82] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system, in *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 87–100, Berkeley, CA, USA, 1999, USENIX Association.

[83] 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.

[84] C. Severance and R. J. Enbody, Comparing gang scheduling with dynamic space sharing on symmetric multiprocessors using automatic self-allocating threads (asat), in *Proceedings of the 11th International Symposium on Parallel Processing*, IPPS '97, pages 288–, Washington, DC, USA, 1997, IEEE Computer Society.

[85] J. Corbalán, X. Martorell, and J. Labarta, Performance-driven processor allocation, in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 5–5, Berkeley, CA, USA, 2000, USENIX Association.

[86] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos, Scheduling algorithms for effective thread pairing on hybrid multiprocessors, in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, pages 28.1–, Washington, DC, USA, 2005, IEEE Computer Society.

[87] A. Gupta, A. Tucker, and S. Urushibara, SIGMETRICS Perform. Eval. Rev. **19**, 120 (1991).

[88] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, Scheduling and page migration for multiprocessor compute servers, in *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 12–24, New York, NY, USA, 1994, ACM.

[89] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, Self-optimizing memory controllers: A reinforcement learning approach, in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 39–50, Washington, DC, USA, 2008, IEEE Computer Society.

[90] C. Bienia, S. Kumar, J. P. Singh, and K. Li, The parsec benchmark suite: characterization and architectural implications, in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008, ACM.

[91] P. Padala et al., Adaptive control of virtualized resources in utility computing environments, in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 289–302, New York, NY, USA, 2007, ACM.

[92] P. Padala et al., Automated control of multiple virtualized resources, in *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009, ACM.

[93] D. Narayanan and M. Satyanarayanan, Predictive resource management for wearable computing, in *Proceedings of the 1st international conference on Mobile systems, applications and services*, MobiSys '03, pages 113–128, New York, NY, USA, 2003, ACM.

[94] P. Barham et al., Constellation: automated discovery of service and host dependencies in networked systems., Technical report, 2008, TechReport (MSR-TR-2008-67), Microsoft Research.

[95] G. Pekhimenko and A. D. Brown, Machine learning algorithms for choosing compiler heuristics., Technical report, 2008, MSc. Thesis (University of Toronto, CS Department.

[96] SPECjbb, 2005, http://www.spec.org/jbb2005.