UNIVERSITY OF CALIFORNIA, SAN DIEGO

Software and Hardware Techniques for Attacking the Multicore Interference Problem

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Anshuman Gupta

Committee in charge:

Professor Michael Bedford Taylor, Chair
Professor Chung-Kuan Cheng
Professor Brian Demsky
Professor Bill Lin
Professor Steven Swanson

2013

Copyright

Anshuman Gupta, 2013

All rights reserved.

The Dissertation of Anshuman Gupta is approved and is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

Chair

University of California, San Diego

2013

DEDICATION

I dedicate this work to my family.

To my dear *father*, who has inspired me to always soldier on.

To my caring *mother*, who has always encouraged me to dream higher.

To my beloved *wife*, who has stood by me through all my hardships.

And finally, to my lovely *sisters*, who always believed in me.

Thank you!

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Table of Contents v

List of Figures vii

List of Tables xi

Acknowledgements xii

Vita xiii

Abstract xiv

Introduction 1

Chapter 1 Qtime: Measuring Execution Quality in Software 12
- 1.1 Architectural Interference and Performance Measurement 15
- 1.2 Qtime: A Tool to Measure Online Application Execution Quality 19
- 1.3 Qtop: A Dashboard for Monitoring and Controlling Execution Qualities of Other Applications 26
- 1.4 Results 29
  - 1.4.1 Evaluation Methodology 29
  - 1.4.2 Quality Time Estimation Results 30
- 1.5 Related Work 33
- 1.6 Conclusion 36

Chapter 2 Qplacer: Improving Execution Quality in Software 37
- 2.1 Qplacer: A Quality Time based Affinity Mapping Tool 39
  - 2.1.1 Simulated Annealing 39
- 2.2 Results 41
  - 2.2.1 Evaluation Methodology 41
  - 2.2.2 Affinity Results 42
- 2.3 Related Work 42
- 2.4 Conclusion 44

Chapter 3 TimeCube: Measuring Execution Quality in Hardware 46
- 3.1 TimeCube Overview 49
- 3.2 Shadow Performance Modeling in TimeCube 51
- 3.3 Results 55

3.3.1 Evaluation Methodology 55
3.3.2 TimeCube's Quality Time Estimation is Highly Accurate 61
3.3.3 Area and Energy Distribution in TimeCube 61
3.4 Related Work 63
3.5 Conclusion 65

Chapter 4 Quality Tables: Controlling Execution Quality in Hardware 67
4.1 Dynamic Execution Isolation in TimeCube 71
4.2 Quality Tables in TimeCube 75
4.3 Dynamically Repartitionable Static NUCA (DR-SNUCA) 78
4.3.1 Dynamically Repartitionable Static NUCA Design 81
4.3.2 Flattened Partial LRU Vector 86
4.4 Results 87
4.4.1 Quality Tables Created in TimeCube 87
4.4.2 DR-SNUCA Evaluation 92
4.4.3 Area and Energy Distribution in TimeCube 95
4.5 Related Work 96
4.6 Conclusion 98

Chapter 5 SPOT: Improving Execution Quality in Hardware 100
5.1 Maximizing Mean Quality Time: A Unified Resource Management Objective 103
5.2 SPOT: Finding the Resource Allocation to Maximize the Mean Quality Time 107
5.3 Prefetcher Throttling 109
5.4 TimeCube Execution Model with SPOT 111
5.5 Results 113
5.5.1 Overall Results 113
5.5.2 Prefetcher Throttling 117
5.5.3 Varying Workload Composition 120
5.5.4 System Scaling 124
5.5.5 Load characteristics 125
5.5.6 Varying Workload Diversity 131
5.5.7 Variable Cache Switching Frequency 134
5.5.8 Cache and Bandwidth Sensitivity Study 135
5.5.9 Area and Energy Distribution in TimeCube 137
5.6 Related Work 140
5.7 Conclusion 142

Chapter 6 Conclusion 144

Bibliography 146

LIST OF FIGURES

Figure 1. Execution time of applications under varying co-schedules 2

Figure 2. Worst-case Application Slowdowns on a simulated 32-core processor. 3

Figure 1.1. Choosing hardware events that have low extrapolation error. 18

Figure 1.2. Quality Time estimation accuracy increases with increasing sampling overhead. 19

Figure 1.3. Qtime framework. 20

Figure 1.4. State Machine for application execution when running Qtime tool. 21

Figure 1.5. Shared Memory Region. 22

Figure 1.6. Application Execution. 23

Figure 1.7. Qtop monitors the online quality of applications using the Qtime tool. 28

Figure 1.8. Quality Time can be accurately and efficiently estimated by our technique. 31

Figure 1.9. Quality Time estimation, by benchmark. 32

Figure 1.10. Instantaneous tracking of Quality Time. 33

Figure 2.1. Qplacer can improve throughput by using Quality Time for placement. 43

Figure 3.1. Worst-case Application Slowdowns on a simulated 32-core processor. 47

Figure 3.2. TimeCube Layout. 49

Figure 3.3. Augmenting manycore processors to measure Quality Time in TimeCube 50

Figure 3.4. Benchmarks can be classified based on the sensitivity of their miss rate to L2 cache sizes. 57

Figure 3.5. Area distribution in TimeCube for calculating Quality Time. 61

Figure 4.1. Quality Tables provide Quality Times for a spectrum of resource allocations. 69

Figure 4.2. Static DRAM buffer partitioning in TimeCube. 74

Figure 4.3. Quality Tables for TimeCube. 75

Figure 4.4. Associatively partitioned DNUCA caches are not energy scalable. 79

Figure 4.5. Layout of Dynamically Repartitionable Static NUCA 81

Figure 4.6. Cache access with Indirect Cache Addressing in DR-SNUCA. . . . 82

Figure 4.7. DR-SNUCA Reconfiguration 84

Figure 4.8. Tag-Duplication in DR-SNUCA 85

Figure 4.9. Flattened Partial LRU Vector 86

Figure 4.10. Normalized Quality Tables for astar, hmmer, and namd. 88

Figure 4.11. Normalized Quality Tables for bwaves, lbm, and sjeng. 89

Figure 4.12. Normalized Quality Tables for bzip2, leslie3D, and soplex. 90

Figure 4.13. Normalized Quality Tables for h264ref, mcf, and specrand. 91

Figure 4.14. DR-SNUCA reduces overall execution energy 93

Figure 4.15. DR-SNUCA performance is comparable to the baseline DNUCA. 93

Figure 4.16. Area distribution in TimeCube for calculating Quality Tables and providing Dynamic Execution Isolation. 95

Figure 5.1. Simultaneous resource allocation leads to better resource utilization for better overall performance. 106

Figure 5.2. TimeCube uses Simultaneous Performance Optimization Table, or SPOT, to find the optimal resource allocation. 108

Figure 5.3. Prefetcher throttling in TimeCube 110

Figure 5.4. TimeCube execution model. 112

Figure 5.5. TimeCube's resource allocation leads to higher throughput. 114

Figure 5.6. Performance improves when cache and bandwidth are allocated simultaneously. 116

Figure 5.7. Prefetcher Throttling maximally utilizes the available bandwidth by intelligently switching between full prefetching, no prefetching, as well as in between aggression levels. 118

Figure 5.8. Prefetcher throttling further improves performance gains achieved by simultaneous resource allocation. 119

Figure 5.9. Variation in IaaS earning gains and fairness with changing workload composition. 121

Figure 5.10. IPC contours for varying compositions. 123

Figure 5.11. With TimeCube, performance improvement increases as we increase the system size. 124

Figure 5.12. TimeCube gives increasingly better performance with increasing number of applications per chip. 126

Figure 5.13. Simultaneous Resource Allocation performs better under increasing system load. 127

Figure 5.14. IPC and speedup wrap-contours for changing load. 128

Figure 5.15. IPC and speedup contours for changing load. 130

Figure 5.16. The performance improvements of TimeCube are impervious to changing diversity of applications within types. 131

Figure 5.17. System throughput improvement remains high with changing application diversity. 132

Figure 5.18. Throughput and speedup contours for changing application diversity 133

Figure 5.19. Different mappings between cache size and reconfiguration interval have varying benefits. 134

Figure 5.20. Simultaneous cache and bandwidth allocation provides higher resource utilization for a range of total cache and memory bandwidths. 136

Figure 5.21. System IPC iso-contours showing the cache and bandwidth sensitivity of TimeCube. 138

Figure 5.22. Overall energy distribution in TimeCube. 139

Figure 5.23. Overall area distribution in TimeCube. 139

LIST OF TABLES

Table 1. Modern multicore and manycore processors provide abundant execution resources for concurrent threads [BFPS11]. 1

Table 2. Low per-core resource availability in modern manycore processors leads to increased contention. 2

Table 1.1. Machine configuration used in Qtoolkit evaluation. 29

Table 1.2. Benchmarks used in Qtoolkit evaluation. 30

Table 3.1. Processor Model used for TimeCube evaluation 56

Table 3.2. Energy (pJ) consumption numbers for some operations in TimeCube 58

Table 3.3. Characteristics of benchmarks used in TimeCube evaluation. 60

Table 3.4. TimeCube's Quality Time estimation is highly accurate. 62

Table 3.5. TimeCube can estimate the execution times of applications with a good precision. 63

Table 4.1. FPLV efficiently provides shadow statistics for DR-SNUCA. 94

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Michael Bedford Taylor for his support as the chair of my committee, and funding my research endeavours through many years. I would also like to acknowledge Dr. Jack Sampson for his valuable advices and help in editing the publications that came out of this work.

Chapters 1 and 2, in parts, are a reprint of the material currently being prepared for submission to the Technical Program Committee of the 2014 International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, for their consideration to include the paper in the conference technical program. The dissertation author was the primary investigator and author of this paper.

Chapter 3, 4, and 5, in part, are a reprint of the paper "Timecube: A Manycore Embedded Processor with Interference-agnostic Progress Tracking", Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford; published in the proceedings of the 2013 IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2013. The dissertation author was the primary investigator and author of this paper.

Chapters 4 and 5, in parts, are a reprint of the paper "DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache", Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford; published in the proceedings of the 2013 International Conference of Computer Design, ICCD 2013. The dissertation author was the primary investigator and author of this paper.

Permission to use these contents has been obtained through signed letters from the co-authors.

VITA

2001-2005	Bachelor of Technology, Computer Science and Engineering Indian Institute of Technology, Kanpur
2005–2006	Design Engineer, Advanced Micro Devices
2006–2013	Graduate Research Assistant, Computer Science and Engineering University of California, San Diego
2006–2009	Master of Science, Computer Science and Engineering University of California, San Diego
2006–2013	Doctor of Philosophy, Computer Science (Computer Engineering) University of California, San Diego

FIELDS OF STUDY

Major Field: Engineering (Specialization or Focused Studies)

Studies in Computer Architecture
Professor Michael Bedford Taylor

ABSTRACT OF THE DISSERTATION

Software and Hardware Techniques for Attacking the Multicore Interference Problem

by

Anshuman Gupta

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2013

Professor Michael Bedford Taylor, Chair

Multicore processors are ubiquitous in servers and have started dominating other domains, such as embedded systems. In order to increase utilization, these multicore processors are sharing memory resources among an increasing number of cores. This sharing leads to memory interference, which causes significant non-uniform slowdowns of concurrent applications, such as 12× on 32-core processors and 2.8× on 4-core processors. The distorting effect of interference on execution quality has posed new challenges for these consolidated systems, which continue to use legacy software that relies on application CPU Time to infer progress. Novel solutions are required to

improve our reasoning about myriad properties from fairness, to QoS, to throughput optimality in the increasingly concurrent computing environments, such as smartphones and datacenters, under the influence of interference.

There are three key challenges posed by interference on multicore processors: How do we measure, control, and improve execution quality of concurrent applications? This dissertation proposes three novel solutions to tackle these problems. First, Quality Time, a metric that represents application progress on a standalone processor and can be used to measure execution quality. Second, Quality Tables, a data structure that provides execution qualities for different system configurations and can be used to control execution qualities. And third, SPOT, a data structure to determine resource allocations that improve execution qualities.

I present scalable software and hardware implementations of these techniques that can handle live applications. The software techniques are bundled into Qtoolkit, a publicly available package, while the hardware techniques are demonstrated using TimeCube, a manycore processor. I present a detailed evaluation of these mechanisms, which show that we can measure Quality Time with just 1% error in hardware and about 8% error in software. By using the resource mangement techniques proposed, we can control the QoS provided to application in hardware with just 1% error and improve throughput by 36% in hardware, using SPOT, and 5.76% in software, using Qplacer. The area, energy, and performance overheads for these mechanisms are very low.

Results of the experimental evaluation show that these solutions can be practically implemented in both software as well as hardware to attack the interference problem, and strongly argue for implementing these techniques in consolidated multicore systems of the future.

Introduction

Multicores, which integrate multiple processors, or cores, onto a single die, have become ubiquitous in both the server and embedded spaces. With successive process generations continuing to provide more transistors, in accordance with Moore's Law, we can expect that increasingly large core counts will be integrated into a single chip.

An important attribute of these multicore processors is that they provide an abundance of on-chip resources shared between cores [teg], as shown in Table 1. While this sounds very attractive, in reality the per-core resource availability for these manycore processors is low when compared to the existing multicore processors, as shown in Table 2. This per-core resource scarcity causes an increased resource contention between applications [Jal] [MQ06], leading to large slowdowns, as shown in Figure 1. This figure shows the execution slowdown of a variety of benchmarks when they are scheduled in pairs and with two instances of each application on a 4-core 2.4 GHz Core 2 Duo machine. The slowdowns reach as high as 2.7×, average 1.35×, and have a very high standard

Table 1. An important as well as attractive attribute of recently introduced multicore processors is the abundance of on-chip resources provided at the disposal of concurrently executing applications.

Resources Provided	**Tile Gx 8072**	**Xeon Phi 7120X**
Cores	72	61
Caches	23 MB	30.5 MB
DDR channels	4	16
Memory Bandwidth	100GB/s	352 GB/s

Table 2. While manycore processors boast of a resource abundance, the per-core resource availability is actually lower than existing multicore processors. This leads to resource contention between applications, which in turn leads to unpredictable slowdowns.

Per-Core Resources	Tile Gx 8072	Xeon 4650
Cache / core	327 KB	2.5 MB
Memory BW / core	1.16 B/cyc	4.26 B/cyc

deviation relative to the execution times. The performance effect on one application might differ when running with another application, which makes it challenging to predict the application slowdowns when running on a multicore processor with other applications. This is termed as *Interference*.

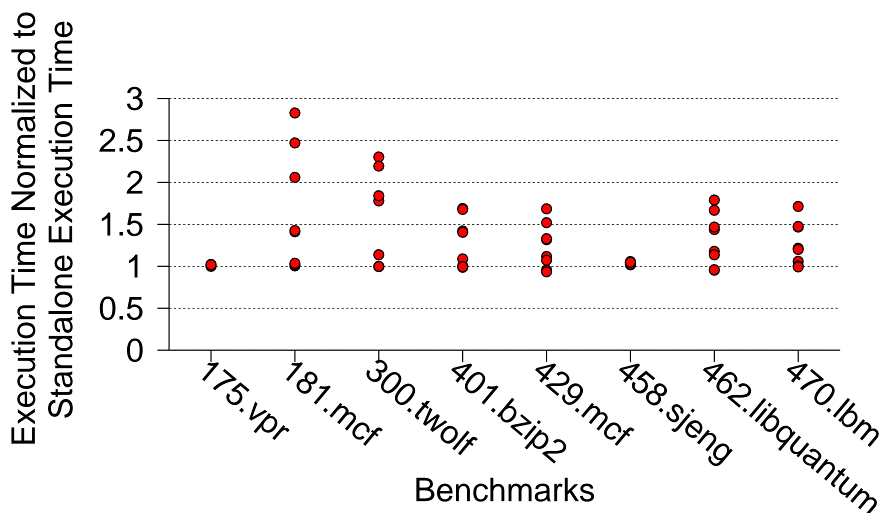


Figure 1. ***Execution time of applications under varying co-schedules*** Despite the existence of both software and hardware mechanisms to improve fairness across concurrent applications, the range of execution times varies widely with different co-schedules due to interference.

The increasing concurrency in multicore processors, compounded with a heightened tendency to share microarchitectural resources, will lead to a continuous increase in the magnitude as well as the unpredictability of these slowdowns, as shown in Figure 2.

This figure shows the worst-case application slowdowns when running a spectrum of 32 benchmark compositions on a 32-core simulated manycore processor with shared last level cache and memory bandwidth. The worst-case slowdown for these manycore runs can be as much as 12× and average 6× over all compositions. The composition details are available in Section 3.3.1. Thus, resource sharing and resulting interference in future multicore processors will lead to increasingly large application slowdowns.

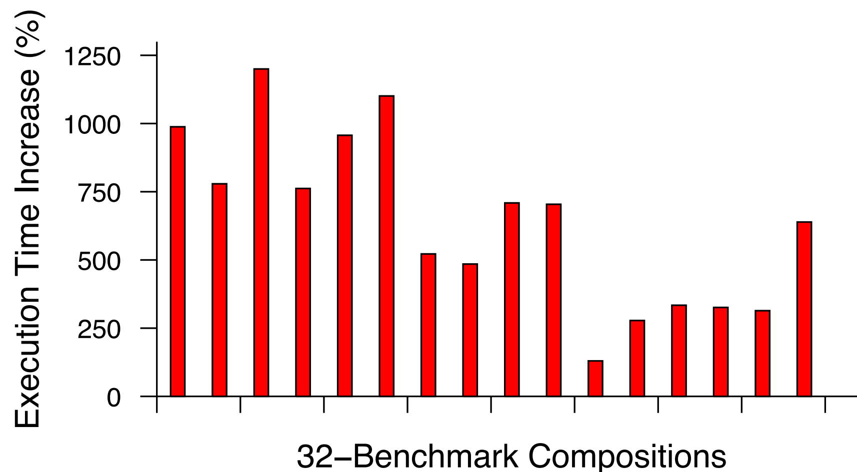


Figure 2. ***Worst-case Application Slowdowns on a simulated 32-core processor.*** On a simulated 32-core processor (see Section 3.3.1 for details), we see that slowdowns are both large, 6× on average, as well as highly variable, from less than 2×, to as much as 12× in the worst case. Thus, CPU-time should not be used as a proxy for progress when making high-order decisions in embedded systems or datacenters using space-multiplexed manycore processors.

Microarchitectural interference manifests itself even at the higher layers of the system architecture and can lead to insignificant inaccuracies. Many multicore systems use legacy software components, such as the operating systems, the batch schedulers, etc., which were originally written for unicore processors. These softwares continue to make several crucial decisions, such as scheduling and resource allocation, maintaining Quality-of-Service (QoS) guarantees, metering, etc. based on the application *CPU-time*,

which was used as representative of the application progress. Interference breaks the representativeness of CPU-times for program execution on multicore processors, and it can lead to significant errors in these crucial decisions.

Challenges Interference is becoming an increasingly important problem for computing domains using multicore processors, and novel techniques and mechanisms are required to reduce its detrimental effects. In this work, I focus on three key challenges posed by interference:

- How do we measure the execution quality of an application running concurrently with other applications while sharing resources?
- How do we provide performance guarantees to applications in the presence of interference?
- How do we allocate resources between applications on a concurrent system for the good of many, but without punishing anyone?

Solutions In this dissertation, I discuss three novel techniques to address these challenges as well as their implementation in software and hardware:

- I introduce a novel metric called *Quality Time* to quantify the execution quality of applications running concurrently on a multicore processor; it is defined as follows:

  > **Quality Time** is the amount of time that it would have taken the thread to attain its current execution progress if that application had had exclusive use of the machine.

  I will demonstrate software as well as hardware mechanisms that estimate Quality Times to measure an application's execution quality while running in live

concurrent systems with high accuracy and low overheads.

- I introduce a novel data structure called *Quality Tables* that can be used to provide performance guarantees to applications on multicore processors.

  > **Quality Tables** provide estimates of application execution qualities for a spectrum of system configurations, such as all possible resource allocations, all possible application placements, etc.

  I will demonstrate mechanisms to generate these Quality Tables in live hardware as well as software and discuss how the system can use them to provide online performance guarantees to applications in the presence of interference.

- I introduce a novel software technique for improving application placement, called *Qplacer*, and a novel hardware-generated data structure called *SPOT*, or *Simultaneous Performance Optimization Table*, to find optimal microarchitectural resource allocations. These techniques use the Quality Tables to improve resource management on multicore processors on-the-fly.

The software implementations of these techniques are packaged into a user-space open-source toolkit called *Qtoolkit*. These techniques were implemented in hardware as well using *TimeCube*, a manycore processor, as a demonstration vehicle. These software and hardware implementations are independent of each other, but they can be used in conjunction as well. This dissertation presents a detailed evaluation of these software and hardware implementations.

An application's Quality Time is equivalent to its standalone execution time on its current processor. Software systems can use Quality Time, as a counterpart for CPU-times on multicore processors, when making decisions based on interference-agnostic application progress. However, calculating Quality Time of applications on a multicore

processor with high accuracy and low overheads is a challenging problem because the hardware mechanisms responsible for sharing resources inside multicore processors are generally black-boxes and do not provide any direct means to measure their effects on application performances. However, most architectures provide hardware performance counters to give insight into the application execution, and these counters can be used to detect interference in the system.

In Chapter 1, I introduce *Qtime*, an architecture-independent user-space tool that uses the hardware performance counters, provided in most modern processors, to measure application Quality Time with low performance overhead and good accuracy. For this purpose, the tool preloads a dynamic library before application execution and continuously tracks application's Quality Time without requiring any code modification or recompilation. Qtime can calculate Quality Time simultaneously and online for multiple applications running on the system. Qtime is included in Qtoolkit, which also contains a user-space dashboard, called Qtop, that can be used to simultaneously visualize the Quality Time of all applications running on the system with Qtime. This dashboard enables an administrator or a user to monitor and analyze the execution quality of a single application as well as the entire system.

While measuring and visualizing application Quality Time is valuable for system software, an even greater opportunity lies with combining the online estimation of Quality Time with system heterogeneity, commonplace in today's multicore systems, to improve the system performance. I describe in Chapter 2 how we can place applications with conflicting resource requirements on cores that do not share the contentious resource and increase the applications' execution qualities. I present *Qplacer*, a user-level tool included in Qtoolkit, that tracks live application qualities for different application placements and uses these tables in an online simulated annealing algorithm to improve application placement on live heterogeneous multicore processors to increase their overall

performance.

I provide experimental evaluation of Qtime and Qplacer in this dissertation, which shows that by using Qtime, in contrast to using CPU-time, we can reduce errors in estimating application progress on a modern commercial quad-core processor from 150.29% to just 23.14% in the worst case, and from 36.91% to 8.18% on average, with an overhead of under 1% and without requiring any code recompilation or modification. I also show that the Qplacer can increase the overall system Quality by 5.76% without requiring any administrator-level or kernel-level privilege. Thus, Qtoolkit can be effectively used in existing multicore systems to measure as well as improve execution quality of applications.

Quality Time estimation in software has two inherent limitations. First, software estimation adversely affects Quality Time accuracy since it uses hardware performance counters to *indirectly* estimate interference on otherwise-opaque multicore processors, and typically only a limited number of hardware events and counters are provided that do not reveal sufficiently detailed information about the microarchitectural execution. Second, the software estimation incurs performance overheads, such as collecting hardware events, running the Qtime code, etc. We can overcome these limitations by doing Quality Time estimation in hardware. Hardware mechanisms will have higher accuracy as they would have the knowledge of processor internals; furthermore, they will have significantly lower energy overheads and no time penalties due to the dedicated logic.

In Chapter 3, I describe an efficient and highly accurate hardware mechanism to estimate live application Quality Time using shadow performance modeling. I demonstrate this mechanism on *TimeCube*, a manycore processor that is augmented with the hardware mechanisms to estimate Quality Times. TimeCube uses an analytical model approximating the hardware execution details and shadow structures for critical shared hardware resources to gather the necessary statistics and accurately determine Quality

Time. I present an experimental evaluation of a 32-core TimeCube instance, which shows that TimeCube can estimate Quality Time and measure an application's execution quality with just about 1% error, at the cost of energy and area overheads of under 0.21% and 6%, respectively.

Estimating Quality Time in hardware allows us to check the execution quality of applications on-the-fly, even in the presence of changing application phases; however, this is insufficient to control the execution quality and provide the performance guarantees required in embedded systems and datacenters. Existing systems use interference-aware application scheduling to control execution qualities, but this approach is coarse-grained and often leads to under-utilization of resources.

In Chapter 4, I demonstrate how TimeCube is augmented to control an application's execution quality at a fine granularity through dynamic execution isolation. TimeCube employs scalable mechanisms for dynamically partitioning critical shared microarchitectural resources to isolate every application's execution and control its quality. It features a novel energy-efficient *Dynamically Repartitionable Static NUCA*, or DR-SNUCA, for shared last-level cache, described in Chapter 4. DR-SNUCA reduces the energy consumption of applications by 16% on average when compared with the existing state-of-the-art DNUCA caches, with less than 1% performance overhead. Therefore, these dynamic partitioning mechanisms are area and energy-efficient and provide dynamic execution isolation in TimeCube with negligible performance overheads.

However, resource isolation is not sufficient to provide performance guarantees because the correspondence between resource availability and application performance is non-trivial. For example, 50% resource allocation does not guarantee 50% application performance. Moreover, slowdown experienced with 50% resource allocation varies widely across applications, and finally, slowdown due to 50% reduction in cache is not the same as slowdown due to 50% reduction in memory bandwidth, as I show in

Chapter 4. TimeCube uses Quality Tables, a collection of Quality Time values for all possible resource allocations, to precisely determine the resources required to attain a certain level of application performance, and then allocate the resources correspondingly to create performance guarantees for live applications. TimeCube periodically generates these Quality Tables in hardware for all applications by using an augmented shadow performance model, described in Chapter 4.

Due to the high resource contention, it is challenging to find a resource distribution in multicore systems that maximizes the resource utilization as well as throughput while maintaining fairness across applications. The ability to control execution quality and resource distribution is insufficient as we need to first determine a *good* resource allocation. In Chapter 5, I present a novel progress-based resource allocation technique that uses Quality Tables inside a dynamic programming algorithm to periodically generate a novel data structure called the *Simultaneous Performance Optimization Table*, or *SPOT*, which gives an optimal resource distribution between active applications that fulfills the system objectives of throughput and fairness with very low overheads. I present a detailed qualitative, as well as quantitative, analysis of this mechanism, and I show that, inside TimeCube, it provides a 36% improvement in throughput compared to existing hardware resource management algorithms.

Dynamic Execution Isolation, in combination with Quality Tables, provides a fine-grained control over execution qualities, but it has the following side-effect as well: Dynamic resource partitioning significantly increases the design space for mechanisms using these shared resources, such as the DRAM prefetchers consuming the memory bandwidth, making it difficult to statically tune these mechanisms at design time. For example, with a variable memory bandwidth availability, it becomes difficult to find a single prefetcher design to perform well for all possible allocations. In Chapter 5, I present a novel *Dynamic Prefetcher Throttling* mechanism that tunes the prefetcher on-

the-fly to send just the right number of prefetches to the main memory, based on the utility of prefetches, for near-optimal bandwidth utilization. I show that prefetcher throttling enables an application to maximize its performance at every possible bandwidth allocation inside TimeCube. Similar dynamic tuning can be done for other such architectural mechanisms to improve the resource utilization and application performance for all possible resource allocations under dynamic execution isolation.

To summarize, interference poses an impediment to systems utilizing multicore processors by making it difficult to measure, control, or improve the execution qualities of concurrent applications. I present three novel solutions in this work to attack these three key interference problems. First, I quantify the distorting effects of interference using Quality Time, a proxy for CPU-time on multicore processors, and use this metric to measure execution quality of applications on concurrent systems. Quality Time can be efficiently measured in software as well as hardware for live applications with high accuracy. Second, I propose generating a novel data structure called Quality Tables, a mapping from all possible system configurations to the resulting application performances, and then using these tables for allocating resources to applications, through dynamic execution isolation, to provide precise performance guarantees and tackle the problem of controlling execution qualities. I show that we can efficiently and accurately create Quality Tables in hardware and control the QoS provided to applications on-the-fly. Third, I propose using these Quality Tables to improve application placements and resource allocations in order to increase the overall system performance without compromising on fairness. I show that we can do this resource management in live systems with significant performance improvements and low overheads. Overall, these software and hardware mechanisms can be used to reduce the detrimental effects of interference, leading to significant gains in system performance as well as transparency. The results make a compelling case to develop and employ these techniques to measure,

control, and improve execution qualities in multicore processors of the future, making them even more attractive to computing domains such as datacenters and embedded systems.

Chapter 1

Qtime: Measuring Execution Quality in Software

Multicore processors are ubiquitous today [Cas] [Bai] and have helped increase the computational density and energy-efficiency of not only personal computers but also datacenters [EC2] [IBM] and embedded systems [arm]. The dynamic sharing of architectural resources on these multicore processors among concurrent applications leads to *interference*, which manifests itself through widely varying and unpredictable slowdowns, as shown in Figure 1, that are dependent on the time-varying interaction of system components and the workload.

Under interference, an application's CPU time, while still indicative of resource occupancy, has become a poor indicator of application progress since total progress now also depends on the net impact of the other concurrently scheduled applications. Unfortunately, many existing codes, ranging from "fair" thread scheduling heuristics, to the user commands ps and top, implicitly employ CPU time as a proxy for application progress, in addition to being the traditional representative of resource consumption. Progress-dependent decisions made by such softwares, such as metering on the Infrastructure-as-a-Service (IaaS) clouds, can lead to substantial inaccuracies in the presence of interference, as shown by Govindan et al. [GLKS11].

Going into the future, with increasing use of multicores and the advent of IaaS clouds, mechanisms that increase transparency about application progress will become increasingly useful, for instance, for improving metering and improving application scheduling or placement to control or reduce interference. I propose a metric, *Quality Time*, which improves on CPU time as a measure of application thread progress. Quality Time is the amount of time that it would have taken the thread to reach its current execution progress if that application had had exclusive use of the machine. Using Quality Time, we can also compute `%Quality`, analogous to the `%CPU` field displayed by `ps` or `top`.

Quality Time, measured for live applications in concurrent systems, is a tremendously beneficial metric as it allows online tracking of an application's progress as well as execution quality. Offline measurements are not able to adequately account for interference, especially in the presence of dynamic application phases, i.e. working sets and code regions, as there are nearly limitless combinations of these phases across applications. Moreover, offline profiling incurs an additional execution and storage overhead and requires knowledge of all applications that can ever execute in the system. Another benefit of in-situ execution quality measurement is that there is no additional effort required in either replicating the execution environment or verifying it, unlike in the case of offline profiling.

In this chapter, I show a simple sampling-based technique that enables low-cost online estimation of each application thread's Quality Time and `%Quality`. The technique is highly portable because it runs entirely in user-space and employs existing hardware mechanisms that are prevalent across most general purpose processors: an event counter that counts instructions issued, or alternatively, a counter that counts L1 data cache accesses. The technique allows measurement accuracy and overhead to be traded off, with typical settings of $< 1\%$ overhead yielding accurate estimations. I also

introduce Qtime and Qtop, which make use of the Quality Time metric. Qtime, analogous to the time utility, allows profiling or metering of a single application by providing online reporting of its Quality Time. Qtop, analogous to the top utility, provides system-wide monitoring and visualization of individual and collective quality of execution over time across applications. These applications run in user-space and require only that an environment variable be set to enable the QLib library to use the Qtime tool; no recompilation is required.

The novel contributions described in this chapter include:

- **The QLib library for accurately estimating Quality Time** I developed a library that coordinates the sampling of hardware counters to estimate Quality Time within 8.18%. I have released the source code for both QLib and the three utilities that rely on it under GNU license.

- **Efficient estimation of Quality Time entirely in user-space** I demonstrate that by merely setting an environment variable to pre-load QLib when an application runs, we can estimate Quality Time for unmodified binaries entirely in user-space.

- **Qtime** I have developed Qtime, a utility that provides real-time, online estimation of a given application's Quality Time for profiling and metering purposes. Compared to using CPU-time to estimate progress, Qtime drops worst case errors from 150.29% to 23.14%, and reduces average case error from 36.91% to 8.18% for an overhead of less than 1%.

- **Qtop** I have developed Qtop, a dashboard utility that provides visibility of execution quality across an entire system.

The remainder of the chapter proceeds as follows. Section 1.1 describes the insights and methods used for inferring Quality Time from hardware counters. Section 1.2

describes the implementation of QLib and the Qtime utility, and Section 1.3 describes the implementation of Qtop. Section 1.4 showcases results for Qtime accuracy. Section 1.5 reviews related work, and Section 1.6 concludes.

1.1 Architectural Interference and Performance Measurement

In modern processors, interference is difficult to predict and manage because of two factors. First, microarchitectural resources, such as cache occupancy and bandwidth, are often shared in a free-for-all fashion and resource allocation decisions occur at very fine temporal granularity. Since main memory is vastly slower than on-chip memory, every memory access from any thread could have potentially detrimental effects on the execution of its cohorts if servicing the memory request forces the eviction of a more useful cache line. Second, applications running on out-of-order superscalars differ greatly in both their demand for resources, and in their sensitivity to not having their demands met. Third, threads frequently transition through execution phases [SPHC02], which results in many possible phase combinations when different threads are run together. Thus, the space of all possible combinations of co-scheduled resource demands is large, frequently changing, and effects depend on interactions of address streams at run-time.

Analyzing CPU Time As a Proxy for Application Thread Progress In the context-switched uniprocessor domain, it was sensible to rely on CPU Time as a metric for thread progress. With interference, we are driven to cast about for potential hardware mechanisms that might be more effective substitutes. Implicitly, CPU Time is computed by the OS using either a timer circuit that asserts a periodic interrupt, or a cycle counter, which counts the number of clock cycles that have elapsed. Relying upon these hardware mechanisms for measuring thread progress works poorly in a multicore environment

because these measures are oblivious to any external effects from other threads that may slow execution.

Alternative Proxies Hardware metrics that are more closely tied to program progress are more promising. Of particular interest are hardware event counters, which count events in a processor's execution, and have become increasingly ubiquitous in modern processors. Such event counters provide a plausible proxy for execution progress because they are highly correlated to real-program progress. For example, in a simple loop accessing an array, the number of L1 data cache accesses, or even simply the number of instructions executed, correlate directly with the number of iterations/progress through the loop. Ideally, we can select among the many event counters that are available and find one or more that are not particularly sensitive to interference, making their measurement more strongly tied to program fundamentals than the current execution environment.

Measuring Event Rates However, there is one key challenge in using counters that are essentially counting program properties – normal, interference-free values may vary between different programs or different inputs, leaving us with no sense of the expected value for that run. We need to estimate the expected rate that those events occur at in interference-free execution, in order to convert from "event" units to "time" units. Because of program phases, event rates are likely to change with time even within a single thread, so we can not rely on static conversion ratios.

In order to estimate an application's Quality Time, we repeatedly measure event counter statistics first over a very short sample period, where other application threads have been temporarily suspended, and then over a much longer execution phase, wherein it is co-scheduled. The short sample period is used to establish the event rate under interference-free execution. Then, we use that rate to convert the total number of events

into a interference-free time value, aka Quality Time.

Choosing a Counter Using this sampling-based approach to sample the interference-free event rate relies upon the hypothesis that the sampled event-rate of a small execution period of the application can be used to extrapolate the behavior of a much larger period of execution. The choice of event counter greatly affects the quality of that extrapolation. Ideally, we would have an event that is 1) high frequency, to avoid aliasing errors due to integer precision counters; 2) oblivious to interference, in the sense that the number of counted events in a fixed sequence of instructions should not vary when other programs are run and 3) low variability in measured event-rates (e.g. events/cycle) across program execution.

Every architecture provides a number of hardware event counters for insight and debugging purposes. To choose among the many such options, we examined the events exposed by the PAPI [TJYD09] library over all the benchmarks from Figure 1, each running interference-free, to select the events with minimal extrapolation error. We used the cycles/event ratio present in a single interval of 1 ms to predict the cycles for the next 99 ms using the event count for those 99 intervals (1% sampling). As shown in Figure 1.1, the extrapolation error varies widely depending on the event selected. L1 data cache access (L1-DCA) and instructions committed (TOT-INS) ended up being among the best metrics, because they were strong on all three requirements for events. First, they have low aliasing error compared to low frequency events such as L2 cache misses (L2-TCM). Second, they are interference-oblivious, unlike for instance, L2 cache misses. Third, and finally, they have relatively low variability (with L1-DCA having the lowest) across programs. For the remainder of the paper, we will focus on L1 data cache access and instructions committed as the two hardware events to track.

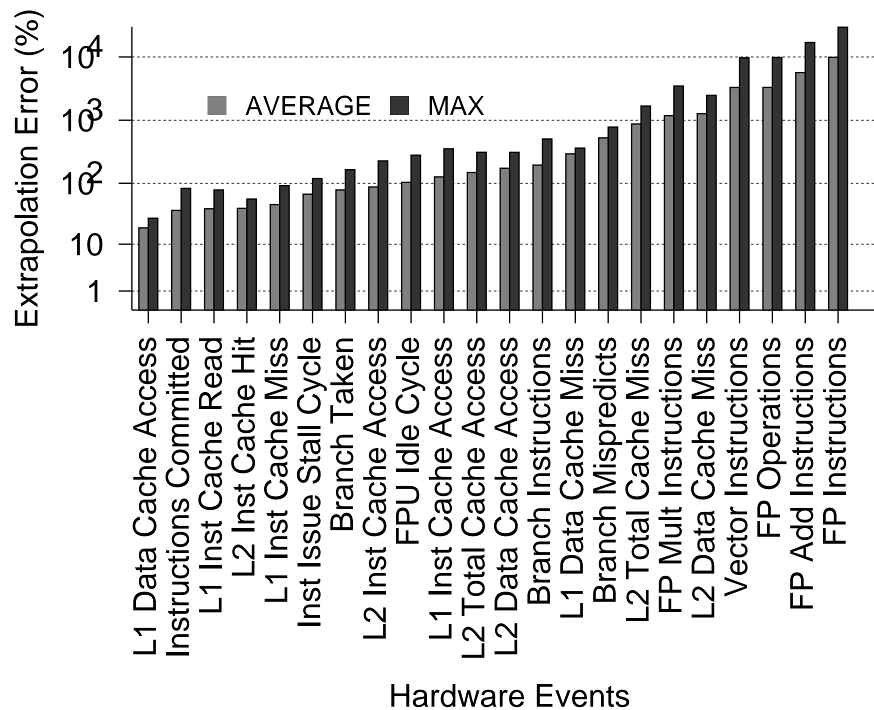


Figure 1.1. ***Choosing hardware events that have low extrapolation error.*** Quality Time estimation samples a small region of execution and acquires an event rate that is then applied to estimate the behavior of a larger region of code. Event counters that generate good event rates have low aliasing errors, are interference-agnostic, and have low variability. The top counters were *Instructions Committed* and *L1 Data Cache Access*, and are used for the remainder of the paper.

Choosing a sampling interval that is accurate and low overhead For our sampling approach, we must select both sampling rate and per-sample durations in order to balance overheads and accuracy. Ideally, the sampling duration should be long enough to *absorb* micro-variations in the application execution. At the same time, increasing the sampling duration increases the disruption to other threads during sampling, and, given a fixed sampling overhead, may negatively impact the dynamism of our predictions and our coverage of different application phases. We explored the impact of sampling overhead and duration on Quality Time estimation accuracy. Figure 1.2 shows the results of

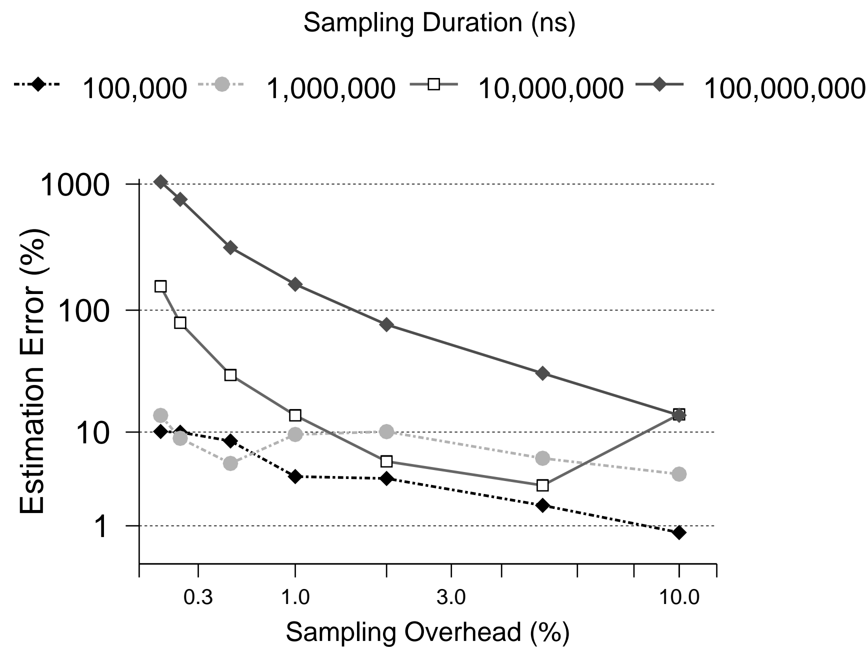


Figure 1.2. ***Quality Time estimation accuracy increases with increasing sampling overhead.*** We can estimate Quality Time with less than 10% inaccuracy using sampling with an overhead of 1%. In this paper, we employ 1 millisecond sampling durations.

our sweep over sampling duration and frequency for the L1DCA event using the same experimental setup as in Figure 1.1. The accuracy increases with higher sampling rates, as expected, but this also leads to a higher overhead. On the other hand, the impact of sampling duration was not as direct, since different applications have different phase durations and behave differently for varying sampling durations. For this paper, we use a 1 ms sampling duration.

1.2 Qtime: A Tool to Measure Online Application Execution Quality

There are two paths toward implementing counter-based approaches for estimating Quality Time. Namely, the approach can be implemented either in the kernel or in

user-space. The former would provide a system-wide view of the effects of interference and allows for the greatest variety of responses to the incoming data. A user-space approach is also desirable because any user can portably reason about the quality of execution of their jobs on any system they may find themselves executing those jobs on. However, since a user-space approach can only control that user's applications, there is the potential for lost accuracy due to interference from other concurrent users.

In the following, we describe the design of Qtime, a user-space tool for estimating an application's Quality Time. Qtime requires measurements of hardware event counters to indicate application progress when running alone as well as concurrently. The tool uses the PAPI library [TJYD09] to provide access to the hardware event counters and read them out periodically. The PAPI library itself relies on the perfCtr module in linux kernel. PAPI virtualizes the event counters, so the tool can continue to get meaningful statistics even in the face of context switches.

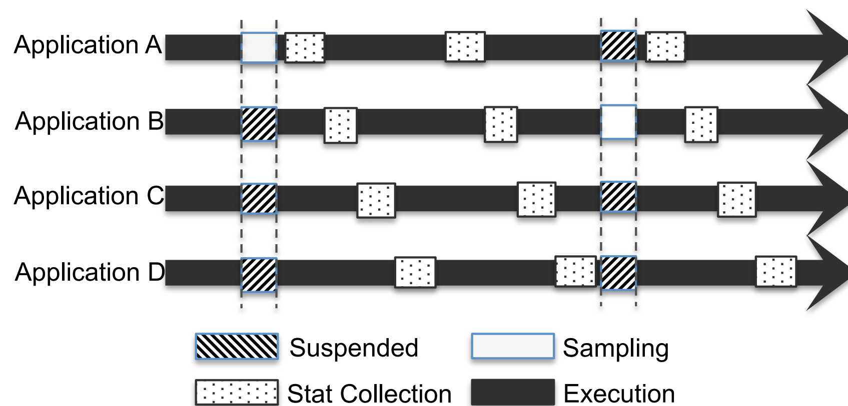


Figure 1.3. ***Qtime framework.*** Qtime uses a sampling-based framework implemented in a dynamically linked library called QLib. Through this library, the applications regularly collect hardware event counters, either in standalone mode or concurrent mode, to estimate the application progress and execution quality online. To collect samples in standalone mode, all other applications are suspended. This synchronization between applications is managed through a shared memory region, which is managed by the QLib.

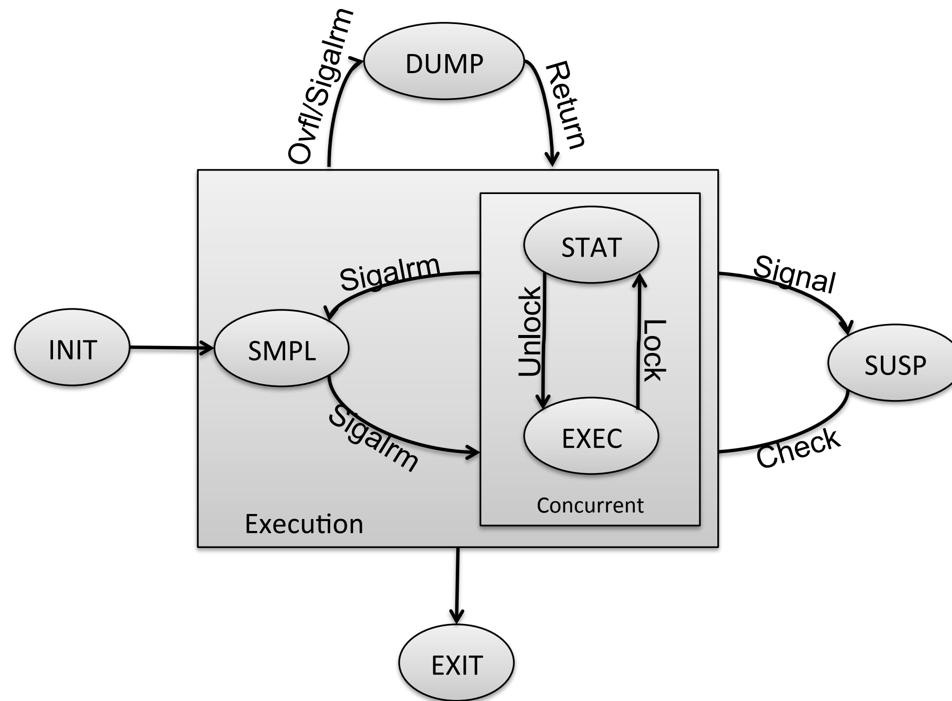


Figure 1.4. ***State Machine for application execution when running Qtime tool.*** An application, when using Qtime tool to estimate its Quality Time, starts in the sampling state (SMPL) where it tries to collect event counters in standalone mode by signaling other applications to suspend (SUSP). Samples are collected (DUMP) in the counter overflow handler. After some predetermined time, the application resumes execution (EXEC) in concurrent mode and periodically collects event samples in the concurrent mode (STAT).

In our user-space implementation, we allow concurrent applications to simultaneously estimate their Quality Time. Each invocation of Qtime attaches to a single application, whose Quality Time will then be measured. Each application must collect its event statistics both in isolated and concurrent execution. These measurements occur throughout execution to provide robustness against application phase-changes. Since we use hardware event counters to track progress, applications periodically read the current hardware event counters for their execution, as shown in Figure 1.3. Since hardware counters are usually shared among all concurrent applications on a processor, there is some orchestration required to get accurate measurements during concurrent

execution. In addition to the sampling phases, wherein other applications are suspended, the applications also time-multiplex the collection of event counts during concurrent (non-sampling) periods. This is clearly seen in Figure 1.3 in the pattern of stat-collection periods between the two indicated sampling periods. Overall stat counts for the entire concurrent execution period are extrapolated linearly.

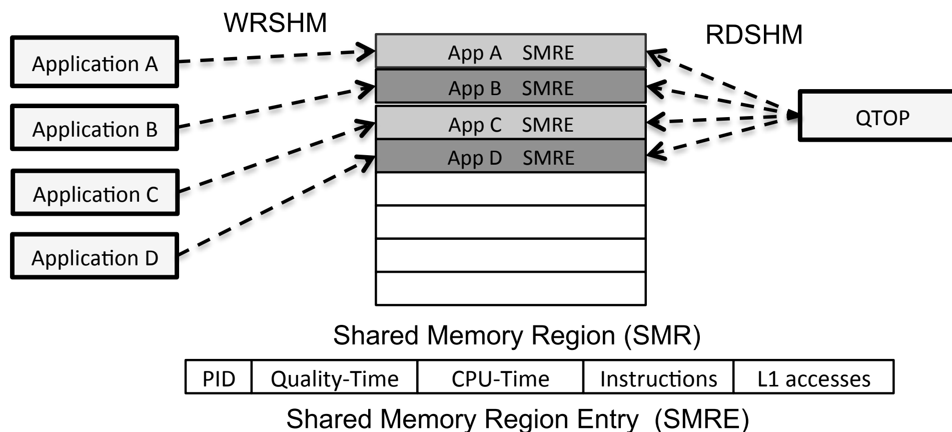


Figure 1.5. ***Shared Memory Region.*** Applications write their Quality Time, CPU-times as well as execution statistics into a Shared Memory Region (SMR) where they can be read (RDSHM) by other applications for synchronization as well as management purposes. Each application is allocated a Shared Memory Region Entry (SMRE), which contains application PID, Quality Time, CPU-time, and other hardware statistics. The shared memory region also contain the variables indicating currently sampling application as well as the application currently collecting execution statistics.

As described above, there is a need for synchronization between applications for using the counters. This synchronization can be provided by a centralized controller; however, in order to reduce unnecessary context switches for the controller execution, calculate application Quality Time using its own resources, and provide scalability through decentralization, we let the application execute the sampling and Quality Time estimation code. We create a library, called QLib, which contains this code.

The QLib library has to be linked to the application binary. To eschew recompilation, we take advantage of *LD_PRELOAD* to intercept *__libc_start_main* and link

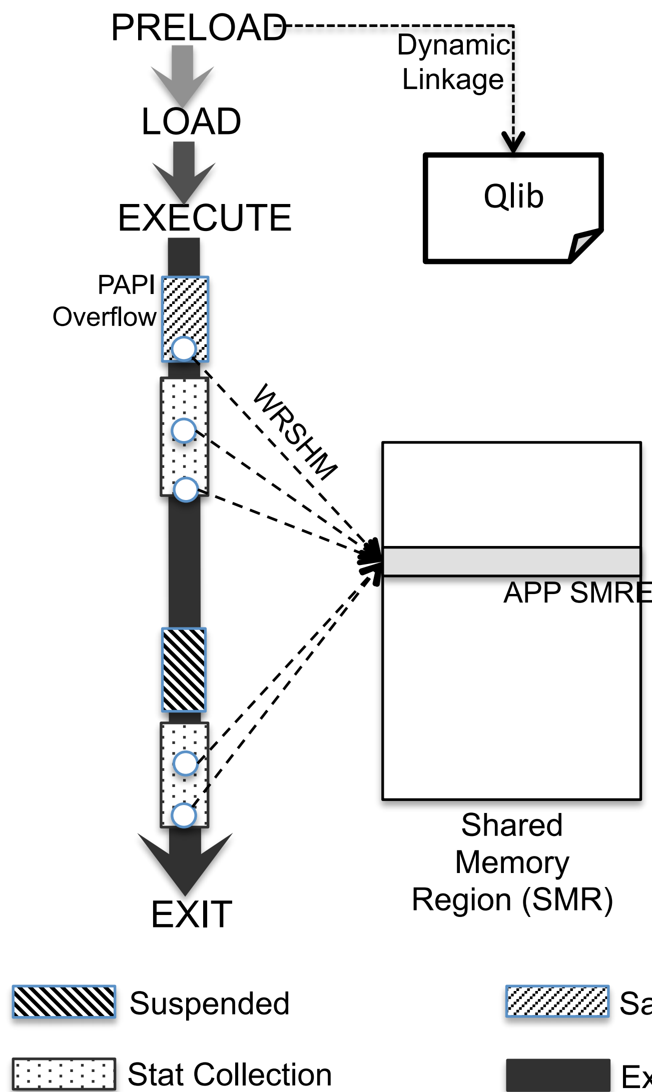


Figure 1.6. ***Application Execution.*** Qtime preloads QLib at the launch of an application, which sets up the application sampling framework. During execution the statistics are written to the SMR when PAPI is triggered by an event counter overflow. Applications suspend other applications to collect samples for standalone execution by sending a signal to other applications. On receiving the signal the application remains suspended, and periodically checks the SMR to see if the application has finished sampling, after which it resumes execution. In concurrent mode, applications can collect samples after obtaining a stat collection lock present in the SMR.

in *QLib*, a library to handle the communications among the applications, PAPI, and a shared memory region holding Quality Time statistics, as shown in Figure 1.6. QLib

connects with PAPI and configures the hardware event counters. It also sets up an overflow controller that tells PAPI to read out the hardware event counters every time the *PAPI_TOT_CYCLES* counter exceeds a given threshold. During these overflows, the statistics will be written (WRSHM) into the shared memory region shown in Figure 1.5. QLib writes the collected event counts during sampling phases to a different location in shared memory than the non-sampled statistics. These isolated application statistics are collected for use as representative samples. QLib also sets up the signal handlers for beginning the sampling phases, and an exit handler such that when the application exits, it frees up associated entries in the shared memory region. It also sets up a SIGALRM handler.

During both the sampling and execution phases, the applications periodically send event statistics, CPU-time, and the currently calculated Quality Time to a *Shared Memory Region*, or SMR, for communicating this information, as shown in Figure 1.5. SMR is divided into Shared Memory Region Entries, or SMREs, where each SMRE can store an application's PID, Quality Time, CPU-time, instructions committed and L1 cache accesses. At the beginning of execution, every application allocates a SMRE. The application periodically updates the SMREs with its execution statistics. The statistics reported are cumulative, so they are overwritten and there is no need of a read-modify-write. When an application exits, its SMRE is freed and can be allocated to other applications.

Figure 1.4 describes the state machine in QLib that controls the sampling and statistic collection behaviors for each thread. We use SIGALRM to periodically schedule when the application will be ready to get samples in standalone mode or concurrent mode, and when it should stop collecting samples. Once an application decides to sample an application, it sends a *real-time signal* to all other applications. On receiving the signal, all the other applications enter a suspended state and periodically read a shared memory

region to check if the sampling state is over. Meanwhile the sampling application initiates the hardware event counters and collects the counters on periodic counter overflows. After a predetermined sampling duration, the application stops collecting samples, writes a value to the shared memory region that indicates that the sampling is over, and resumes execution. When the suspended applications next read the shared memory region, they resume their execution as well.

The applications also collect samples during the concurrent execution. In the SIGALRM handler, if it needs to start collecting samples, it tries to obtain a lock placed in the shared memory region to ensure that no other application is currently using the hardware counters. On obtaining the lock, the application initializes the hardware counters, records event counts, and calculates the Quality Time. The Quality Time is also updated in the shared memory region every time the PAPI_TOT_CYCLES counter overflows.

Before the application begins standalone sampling, it dumps its current statistics in the Shared Memory Region, changes the mode to sampling mode and resets the hardware event counters. Then it dumps the statistics in the shared memory region on overflows until it finishes standalone sampling. At this point, it again dumps the statistics, resets the hardware counters, changes to the EXEC state, and resumes execution. During the EXEC state, the application is not collecting any samples and, since it temporarily suspends the event gathering, it doesn't register any event overflows. As a result, during the EXEC state the Quality Time estimation is done inside the SIGALRM handler. Qtime records the ratio of application's Quality Time progress versus the CPU-time progress during the STAT state. Qtime then uses this ratio during the EXEC state to update the application Quality Time based on its CPU-time progress.

Since the hardware counters are read on cycle overflow, while the state transitions happen inside the SIGALRM handler, which is triggered on time, the state transitions

and statistics collection are not synchronized. Thus we can get dirty data across these transitions. To avoid that, every time the application transitions from a statistics collecting state, it reads the counters, dumps the data into the shared memory region, and resets the hardware counters before starting the next state.

Taken as a whole, Qtime provides an efficient and accurate tool that allows an application to see its own Quality Time. This is already highly useful for purposes such as profiling, but in the next section, we will show how collecting Quality Times for all currently running applications is useful for managing overall system quality.

1.3 Qtop: A Dashboard for Monitoring and Controlling Execution Qualities of Other Applications

In addition to Qtime, we also implemented Qtop, a dashboard which continuously tracks application qualities, and provides monitoring and controlling facility for the overall system quality. Applications run with Qtime dump application statistics including PID, Quality Time, and CPU-time in the shared memory region to communicate with other applications. Qtop periodically reads this shared memory region for the execution statistics and maintains a history of the application qualities over time.

The Qtop dashboard presents a compact visualization of the system execution quality at present as well as over the past. Qtop creates a live display of the quality of applications executing using the Qtime tool, as shown in Figure 1.7. It displays not just an application's Quality Time and CPU-time, but also takes a ratio of these two time metrics to calculate the application's execution quality over recent execution, and displays it in ascii at a resolution of 10% execution quality. It also summarizes the application quality over the entire execution as well as customizable periods, such as last 1 second or last 5 seconds, and displays them in the application summary. Finally, it also shows the cores on which the application was executing for each update.

Qtop can be readily used for detecting a lack of overall quality in the system as well as the offending workload. Similarly, it could be used to detect whether the system is underutilized and can be further consolidated. Qtop can monitor the entire system with very low overhead (¡ 1% core overhead) comparable to common linux monitoring tools such as *top*.

```
-------------------------------------------------------------------------------
Application  Quality(full)  Quality(1s)  Quality(5s)  Quality-Time  CPU-Time
-------------------------------------------------------------------------------
 22581          69.8 %        62.5 %        64.0 %       117.10s    167.64s
 22585          51.1 %        47.9 %        48.3 %        86.09s    168.45s
 22589          69.6 %        61.4 %        69.1 %       116.97s    168.14s
 22593          50.4 %        56.8 %        53.0 %        85.02s    168.63s
-------------------------------------------------------------------------------

Quality of Application 0 over time (sec) -->
-------------------------------------------------------------------------------
100% |
     |: .   .. . :   .. . ..: .. .. :.: . .  .    .  . :  . .  :... . . .. .
 50%-|:.:.::::::::.::::.::::::::.:::::::.:::: :.:::::::.::.:.:.::::::::::::::.
     |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
  0% |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Core |00000000000000000000000000000000000000000000000000000000000000000000000
-------------------------------------------------------------------------------

Quality of Application 1 over time (sec) -->
-------------------------------------------------------------------------------
100% |
     |       .                 .                         :
 50%-|    ...:::      .. .     .::       ...    ..:.....:::: .   .....  ...::
     |:..::::::::..::::::::::::::.   .::::::::::::::::::::::::.:::::::::::::
  0% |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Core |33311113213231222223113123321222112132311132333332121213122111211312132
-------------------------------------------------------------------------------

Quality of Application 2 over time (sec) -->
-------------------------------------------------------------------------------
100% |                            .         ..
     |:.:.:. . .:::.. . .  .    .  ::: :    :::: . .  .    .  ..     .
 50%-|:::::: :.:::::::: ::.::::::::::.::.:.::::.:.:..:.::.:.:::::.:::: :.:  :
     |:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::.::::::
  0% |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Core |22322222323323233332222233232333223222222223223223322322323223222223223
-------------------------------------------------------------------------------

Quality of Application 3 over time (sec) -->
-------------------------------------------------------------------------------
100% |
     |                           .                          .
 50%-|       .::    . ..... .....:       .... ..    ..    .:::..    .    .....:
     |  .::::::::::.::::::::::::::..   .::::::::::::::::::::::...::::::::::::
  0% |::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Core |11133131131113131111331311113111131311133311111111311131311133131311311
-------------------------------------------------------------------------------
```

Figure 1.7. ***Qtop monitors the online quality of applications using the Qtime tool.*** Qtop periodically reads the shared memory region for information about the applications running Qtime, and displays their overall as well as current execution quality, for example during the last 1 second and 5 seconds. It shows the history of each application's quality in a live curve where each dot represents a 10% quality during that second. Underneath, the core on which the application was running at that point is shown.

1.4 Results

1.4.1 Evaluation Methodology

We now describe the evaluation of our user space tools: Qtime, which approximates application Quality Times, and Qplacer, which uses simulated annealing to improve application placements.

We implemented a user-space tool, Qtime, to suspend and sample applications, and calculate Quality Time using statistics collected from applications. The other user-space tool Qplacer is a tool that monitors application qualities and improve application placement using simulated annealing. We run our experiments on the setup described in Table 1.1. We use the PAPI library [TJYD09] version 5.0.1.0 to collect application execution statistics. We use sampling periods of 1 millisecond and sampling periods of 1% compared to concurrent execution periods. We use the PAPI overflow threshold as 1 million cycles.

We use benchmarks from SPEC2000 [Hen00] and SPEC2006 for our evaluation. We describe the benchmark characteristics in Table 1.2. For each workload, we run all the benchmarks in a loop until all the applications have finished at least once and measure the Quality Time for all the applications simultaneously.

Evaluating our Quality Time scheme and comparing alternatives requires us to replay each scheme in the face of potentially variable machine behavior. Thus, we

Table 1.1. We run our user-space tools included in Qtoolkit on a modern multicore processor to determine Quality Time estimation accuracy in software.

OS	CentOS release 5.8 (Final), Linux 2.6.39.4
Processor	Quad-core Intel Xeon X3220, 2.40GHz, 2 x 4MB shared L2 cache
Memory	1066MHz FSB, 6GB DDR3

Table 1.2. We use 13 benchmarks from SPEC2000 and SPEC2006 benchmark suites. These suites are a good representative of typical applications that are run on multicore applications.

Application	Suite	Description
164.gzip	SPEC2000	File compression
175.vpr	SPEC2000	Place and route CAD tool
181.mcf	SPEC2000	Vehicle scheduling algorithm
183.equake	SPEC2000	Seismic wave propagation
188.ammp	SPEC2000	Computational chemistry
197.parser	SPEC2000	Word processing
256.bzip2	SPEC2000	File compression
300.twolf	SPEC2000	Computer aided design
401.bzip2	SPEC2006	File compression
429.mcf	SPEC2006	Vehicle scheduling algorithm
458.sjeng	SPEC2006	Pattern recognition
462.libquantum	SPEC2006	Quantum computing
470.lbm	SPEC2006	Computation fluid dynamics

run both our baseline isolated executions and each benchmark tuples multiple times to better cover the scope of real program behaviors. Overall we collected >750 data-points to evaluate the accuracy of Qtime. For comparing the accuracy of instantaneous estimates, we use the simplifying assumption that, for these benchmarks, an equivalent number of committed instructions implies an equivalent amount of application progress. This simplifying assumption is borne out by the minimal variance in total committed instructions across runs for SPEC benchmarks.

1.4.2 Quality Time Estimation Results

We ran Qtime with a sampling period of 1% compared to the concurrent execution period and a sampling duration of 1 millisecond using the L1-DCA and TOT-INS events. We observe that while the existing method of using CPU time to track application progress has an error of 36.91% on average, our initial simple sampling-based technique is able

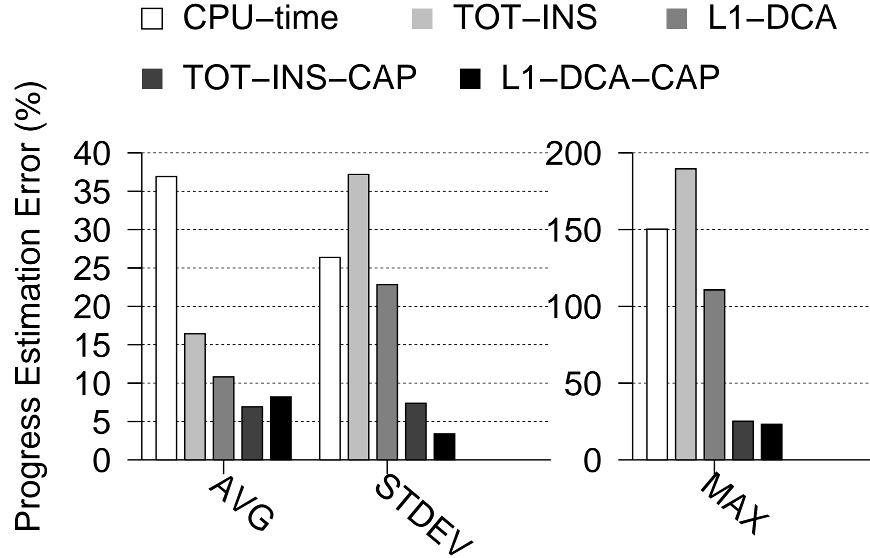


Figure 1.8. ***Quality Time can be accurately and efficiently estimated by our technique.*** Sampling-based Quality Time estimation has an accuracy of 10.81% on average and 1.74% geometric mean, when using L1-DCA. However, the maximum error is 110.70% with L1-DCA and 189.61% with TOT-INS. We are able to reduce the max error to 23.14% (and 8.18% on average) when using L1-DCA by capping the Quality Time each interval.

to reduce this error for almost all the workloads, as shown in Figure 1.8, to 10.81% on average when using L1 data cache access and 16.43% when using instructions committed, as shown in Figure 1.8. However, for some workloads, the error goes up significantly when using our sampling technique. This is due to the amplification effects of sampling: If our "representative" sample is actually in a different phase than the execution we use it to predict, then our back-calculation of time can be erroneous. In particular, if we sample during a low event-frequency phase, and predict for a high event-frequency phase, our estimation of time elapsed can be implausibly high. As a result, the maximum error is still very high, 110.70% and 189.61% for L1 data cache access and instructions committed respectively, as shown in Figure 1.8.

Fortunately, we can refine our technique by applying *capping* to our Quality Time

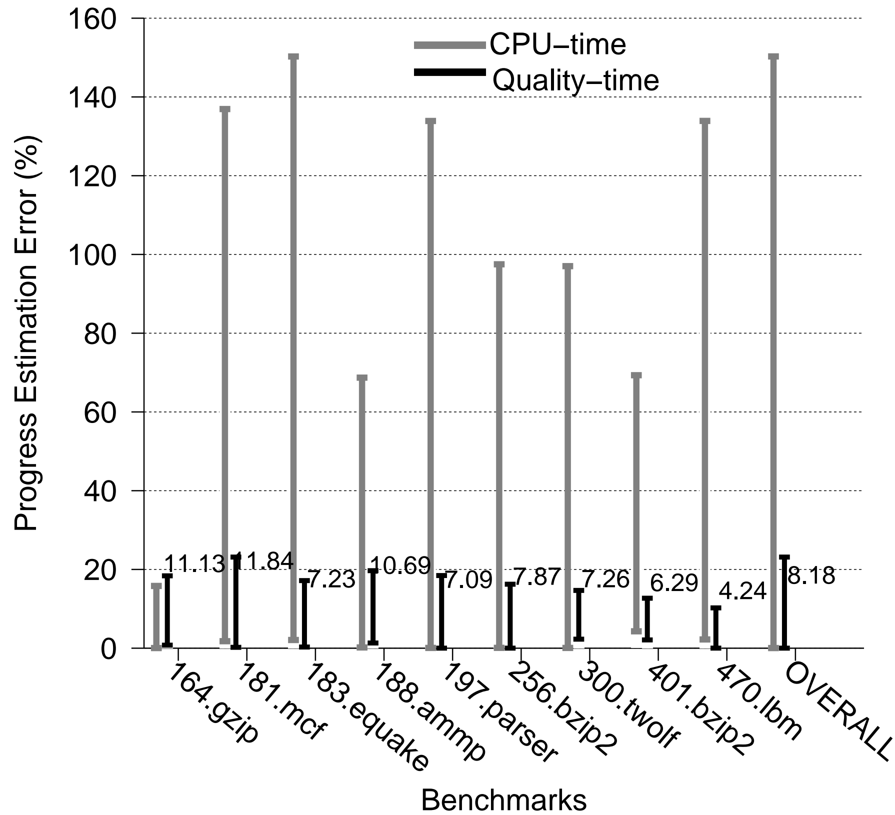


Figure 1.9. ***Quality Time estimation, by benchmark.*** We show the ranges of error for both CPU-Time and L1-DCA Quality Time estimation. Our technique improves the accuracy and reduces the variability in approximating application progress in the presence of interference.

estimation to improve our results. We know that for every interval, the Quality Time cannot be less than zero. Also, ignoring the rare case of speedups when sharing resources, the Quality Time can be assumed to be less than or equal to the CPU time. We apply these two limits every intervals to bound our Quality Time estimation. As shown in Figure 1.8, capping leads to better Quality Time estimation, especially when using L1 data cache accesses. When using L1 data cache access, capping reduces the maximum error to 23.14% and average error to 8.18% with a standard deviation of 3.39%, as shown in Figure 1.8. Figure 1.9 shows the final results by benchmark.

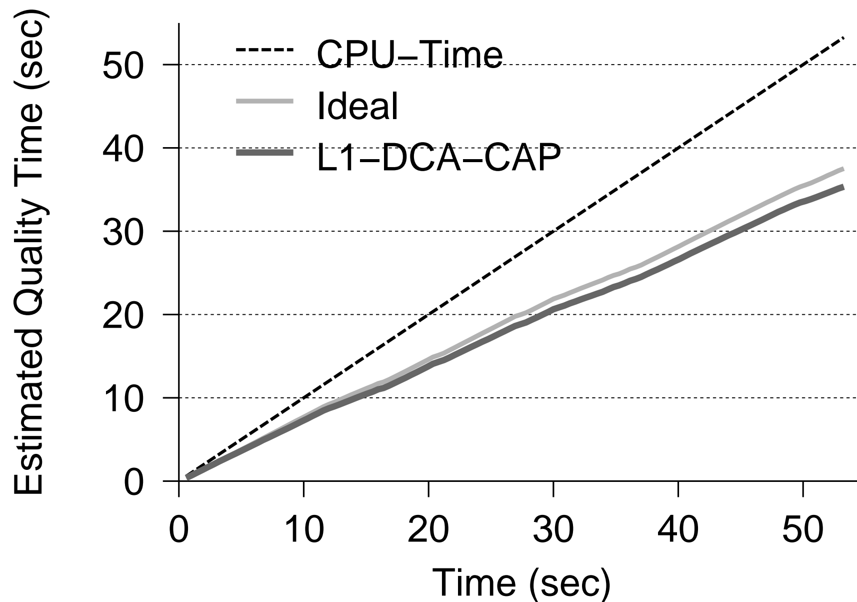


Figure 1.10. ***Instantaneous tracking of Quality Time.*** We plot the calculated and ideal Quality Time for 401.bzip2 as a function of time. We show that our technique can provide accurate instantaneous estimations of Quality Time as well as accurate holistic estimations, with the estimated Quality Time closely tracking the ideal.

Figure 1.10 shows an example of how our technique can provide consistently accurate estimations over the course of execution, as well as accurate summary statistics. Figure 1.10 shows that, for the entire course of the execution of 401.bzip2, the estimation of progress tracks very closely with the ideal. While whole-execution accuracy is sufficient for use cases such as IaaS metering, fine-grained accuracy is necessary for using Quality Time for scheduling, resource allocation, or other dynamic decisions.

1.5 Related Work

Architectural Interference. Typical commercial multicore processors share resources among concurrent threads, and resource sharing leads to interference between applications, as described by Tang et al. [TMV^{+}11]. Govindan et al. [GLKS11] show

that even with the use of hypervisors the unpredictability in slowdowns is very high. Stillwell et al. [SSVC09a] also examined the performance impact of resource sharing in servers at the system level. For these resource-sharing processors, it is important to precisely estimate the performance of applications in order to improve resource accounting and utilization, as shown by Armbrust et al. [AFG+10]. For resource-sharing commercial systems, it is important to accurately estimate the progress of applications and exercise control over it in order to maintain performance guarantees and improve resource utilization, as also pointed out by Buttazzo [But06], since even state-of-the-art resource management schemes, such as the ones proposed by Gohner et al. [GWG+] and Elmroth et al. [EMHF09], do not account for application slowdowns due to sharing of processor resources.

Quality Time provides a performance abstraction for interference-free execution. Such abstractions can be useful for high-order decisions such as resource management and progress tracking, as suggested by Zhang et al. [ZDFS07], in multicore systems.

Performance Analysis. Several tools and techniques have been proposed previously that do performance analysis of application executions, such as VTune [Rei05] and Cilkview [HLL10]. These techniques are mostly offline, such as by London et al. [LDM+01]; moreover, these techniques analyze application performance in isolation. Zagha et al. [ZLTI96] propose an offline performance analysis using hardware counters for specifically MIPS R10K, whereas our technique can be used on any platform. Compared to context-sensitive technique proposed by Ammons et al. [ABL97], our technique provides less information, which is sufficient for our purpose; as a result, the tool is efficient enough to be used online.

Kambadur et al. [KMHK12] propose using remote machines to analyze profile data obtained from live datacenter applications using Google Wide Profiler [RTM+10].

This technique has a very low overhead; however, since they collect profiles on live applications, but process them on remote machines, the round-times are too large to make phase-sensitive analysis/scheduling. Moreover, they do not have standalone performance estimates, which is very useful in providing bounded QoS to applications in certain settings, such as IaaS or embedded-systems.

Performance Estimation. Performance estimation has been studied in existing literature for different objectives. For example, Eyerman et al. [EHE11] used a mechanistic performance modeling to create CPI-stacks, which can be used to determine performance bottleneck in systems. These models however require some knowledge of the microarchitecture as well as some offline regression. Lee et al. [LB06] use offline regression to estimate performance and power consumption of applications. However, our performance estimation technique is online and requires no knowledge of the underlying microarchitecture.

Research has been done to even create application progress estimates in hardware. It is faster to accumulate execution statistics in hardware and most importantly, the performance estimation inside architectures can produce very high accuracy due to their intimate knowledge of the architectural details. TimeCube [GST13] tracks application progress using an analytical performance estimation model similar to the one proposed by Solihin et al. [SLT99]. These models are able to model minute architectural details such as tracking prefetches, measure memory bandwidth constraints, and cache intricacies such as dirty lines, such as the mechanism proposed by Kaseridis et al. [KSCJ10]. They even model off-chip architectural resources that affect application performance, such as the details of DRAM DDR protocol and bank buffer behaviors. These mechanisms can even obviate the need for standalone execution by using shadow structures, for example shadow cache techniques that have been proposed for associative caches, such as by Zhou

et al. [ZPS^{+}04], which are based on the LRU-stacking property [Hil87]. The shadow structures measure hardware events such as cache misses for arbitrary resource amounts, such as cache sizes, which is required for the hardware shadow performance modeling. These hardware mechanisms are efficient and more accurate, but they are not useful for existing processors which are already facing architectural interference problems, for which the solution has to be implemented in software.

1.6 Conclusion

While multicore processors are commonplace in desktops as well as servers, it is only recently that their use in commercial cloud computing and embedded computing has highlighted the problem of microarchitectural interference present in these processors. This interference, which can lead to a wide variation in execution times, can be efficiently measured for all active applications to track their live progress, or Quality Time, using Qtime and monitor all of them simultaneously using Qtop, which can be used to get system insights as well as make other higher-order decisions, such as metering and resource management.

Acknowledgment

Parts of these chapters are reprinted from the following papers:

- Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford. "Quality Time: A Simple Online Technique for Quantifying Multicore Execution Efficiency". To be submitted at the International Symposium on Performance Analysis of Systems and Software, ISPASS 2014.

Permission to use these contents has been obtained through signed letters from the co-authors. Dissertation author was the primary investigator and author on this paper.

Chapter 2

Qplacer: Improving Execution Quality in Software

Multicore processors are used by systems to promote consolidation, which in turn provides an increased compute density and reduced operating costs. This consolidation in turn leads to interference, which can prove detrimental to the system in terms of accuracy and throughput. However, the consolidation itself also provides an opportunity to reduce this interference. The consolidated systems typically provide some heterogeneity within the multitude of applications being executed, as well as the processing elements and their associated physical resources.

The variations in applications transcend to the differences in their execution behaviors and, more specifically, their resource requirements and sensitivities. This implies that while certain combinations of applications contend for the same resources, leading to larger interference, an alternative application combination can be symbiotic, leading to lesser resource contention and, therefore, reduced interference.

At the same time, there is physical heterogeneity in multicore processors, both across as well as within. The variations between processors across generations or different commercial offerings is a common source of heterogeneity multi-processor systems. There can be heterogeneity within a single server with multiple sockets, as a core

shares the off-chip bandwidth with another core on the same socket, but not with a core on a different socket. Even within multicore processors, we are witnessing increasingly complex resource distribution; for example, a core might share on-chip cache only with a subset of the remaining cores on the socket. This physical heterogeneity can be exploited to reduce interference in multiple ways, such as co-scheduling cache-sensitive applications on non-cache sharing cores.

In this chapter, I introduce Qplacer, a user-space tool that recognizes the application and physical heterogeneity on the system and exploits it by intelligently placing the applications on cores in order to reduce the interference on the system. Qplacer uses Quality Time information to discover preferable affinities among co-scheduled applications and can re-map applications to different cores to improve execution quality and throughput.

The novel contributions described in this chapter include:

- **Qplacer** I have developed Qplacer, a user-space affinity mapping tool that actively monitors inter-application interference using Quality Time metrics, and efficiently moves application threads among cores to reduce the interference.

- **Improvement of application placement in user-space** I present a quantitative analysis of Qplacer on a multicore system with $>$100 data-points to show that Qplacer achieves 5.76% higher average throughput over a random scheduling and provides a maximum throughput improvement of 47.42%.

The remainder of the chapter proceeds as follows. Section 2.1 describes the implementation details of Qplacer. Section 2.2 presents the evaluation results for Qplacer showing throughput improvements. Section 2.3 reviews related work, and Section 2.4 concludes.

2.1 Qplacer: A Quality Time based Affinity Mapping Tool

The quality of application execution can be significantly affected by architectural resource contention, and different schedules of applications to cores will result in different levels of contention. Since Qtime enables online monitoring of application qualities, it provides an opportunity to alter system configurations on-the-go and quickly react to changing application phases. This level of dynamic reactivity is very difficult when monitoring or profiling is done offline or on remote machines. We create a user-space tool, Qplacer, that attempts to improve system throughput by suggesting application placements that will improve execution quality. Qplacer is a user-space tool and does not require root access to run.

While Qplacer does not interfere with OS application scheduling in terms of which applications are currently scheduled, it attempts to increase system quality by discovering better application placements. For this purpose Qplacer uses simulated annealing, which ensures that even if the system converges to a locally optimal placement, it continues to probabilistically try other configurations and provide robustness against dynamically changing application phases and interactions.

2.1.1 Simulated Annealing

Application qualities provide a direct indication of possible resource contention in the system. However, it doesn't give an insight about which resource is under contention; as a result, even if we see a drop in application qualities, we cannot determine alternate placements for the applications to cores that do not share the contentious resource. While it is possible to use the program counters to determine the resource under pressure, this reduces the portability of the placer by making it architecture-dependent. Instead, we rely

on the empirical knowledge of interference between applications for different placements.

There exists a fixed number of possible application placements on a processor. Some of these placements can be equivalent, i.e. all pairs of applications share the same set of resources in both placements. We identify these unique configurations for the processor, empirically determine the quality of applications for each of these unique configurations, and switch to the configuration with the highest overall application quality. However, the application phases can change over time; as a result, we need to regularly update our empirical knowledge of application qualities, and continually switch to the best possible configuration. We employ simulated annealing to accomplish this, since it allows us to get out of local maxima for the total application quality.

Qplacer uses the following simulated annealing model:

- *States, S*: Represented by unique configurations in the system. Two configurations are not unique if all applications are homogeneous co-locations in the two configurations. For example, on our evaluation machine, there are three possible unique configurations, i.e. ”ab,cd”, ”ac,bd”, and ”ad,cb”.

- *Energy, E(S)*: Each configuration has an associated energy. Qplacer records the *Weighted-Quality* of each application in each configuration, and uses the sum of these Weighted-Qualities of all applications as the energy of the configuration. Every interval, the Weighted-Quality of all applications are calculated by taking the sum of their existing Weighted-Quality, weighted down by a damping constant, and the current Quality of these applications, weighted down by one minus the damping constant. The initial Weighted-Quality of each application is a 100%.

- *Temperature, T*: The system temperature determines its entropy. Qplacer is less willing to change soon after a configuration switch. So it uses the natural logarithm of the time (in milliseconds) since last switch as the system temperature.

- *Switch Probabilities, P*: Finally, Qplacer determines the probability to switch from state S to S' as:

$$P(S,S',T) = e^{\beta \times (E(S') - E(S)) + T} \tag{2.1}$$

β is a convergence constant. The probabilities are normalized so that the sum of all switch probabilities is 1.0.

Every interval, Qplacer estimates the switching probabilities, and then generates a random number to determine the next state of the system. The Qtop monitoring tool displays the application swaps made. While there are overheads associated with swapping configurations, our results indicate that the swaps occur infrequently enough, and the programs converge to beneficial schedules rapidly enough that affinity control can provide benefits in the common case. This is particularly promising compared to static profiling approaches, as it means that Qplacer or similar approaches will provide benefits even for previously unseen programs. Thus, Qplacer will be particularly useful for expanding IaaS and cloud computing domains wherein arbitrary user computations may be offloaded to consolidated servers for execution.

2.2 Results

2.2.1 Evaluation Methodology

We now describe the evaluation methodology for our experiments. We use a framework similar to the one used in Section 1.4.1. We run the experiments on the system described in Table 1.1. Qplacer is a user-space tool, which allows a user to improve the placement, and therefore the performance, of its applications even in the absence of any system support. We use 0.25 as the damping constant for simulated annealing, and 2.00 as the convergence constant. We collected $>$100 data-points with Qplacer with different application workloads to evaluate the performance improvement with Qplacer. For total

system throughput we compare the sum of execution times for all the applications' first runs.

2.2.2 Affinity Results

Since Quality Time can accurately estimate the impact of interference on an application, making Quality Time information available to a scheduler can be useful in dynamically discovering which application pairings result in the least conflicting schedules.

We evaluated our affinity scheduler's ability to produce good co-schedules by scheduling two copies each of two applications on a multi-chip module with two processors, each with two cores. We compare our tool's dynamic scheduling against the average throughput over the 3 possible distinct static scheduling configurations. Figure 2.1 shows that, for most sets of applications, choosing to dynamically reschedule based on Quality Time indications of interference is beneficial for overall throughput. In cases where there was little potential benefit between the best and average schedules, the technique sometimes decreased throughput due to the errors in estimation of the Quality Time leading to a misguided choice of application placement. The migration and sampling overheads were small and did not significantly impact the overall throughput. Improving our Quality Time estimation and heuristic to avoid these low benefit cases will be a future effort.

2.3 Related Work

Resource Management. Several hardware techniques have been proposed to manage resources inside processor itself, such as profiling based allocation schemes proposed by Liu et al. [Chu04] and Suh et al. [SDR02a]. Bitirgen et al. [BIM08] proposed simultaneous cache and bandwidth allocation using machine learning. These management

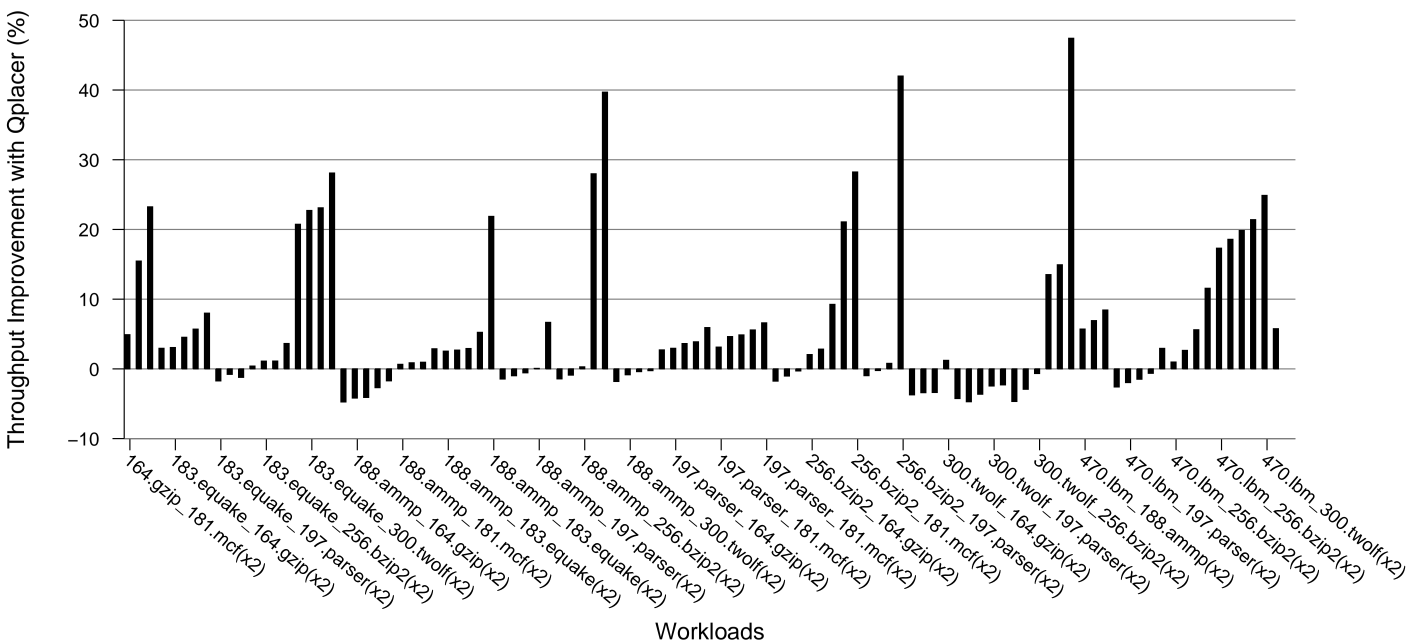


Figure 2.1. ***Qplacer can improve throughput by using Quality Time for placement.*** Qplacer can use the online Quality Time estimates for the applications, which changes over time due to application phases, and dynamically determines application placement using simulated annealing. In our evaluation this leads to an average throughput improvement of 5.76% over 101 runs and a maximum throughput improvement of 47.42%. The improvements should be even larger for systems with higher heterogeneity.

schemes are tuned for varying purposes; for example, Hsu et al. [HRIM06] tune their cache allocation algorithm to maximize different metrics such as fairness and throughput; and Guo et al. [GSZI07] allocate cache partitions based on QoS provided by choosing between strict, elastic, and opportunistic schemes. However, these policies are better managed at the software level, because of the possible changes in the system requirements.

Symbiotic Job Scheduling. In software, co-scheduling can reduce pressure on the resources and increase performance when sharing scarce resources between multiple applications. Amongst previous works, Cazorla et al. [CKS+05], Jiang et al. [JSCT08], El-Moursy et al. [EMGAD06] and Snavely et al. [ST00] discuss mechanisms for application scheduling. Federova et al. [FSSN05] examined OS-level scheduling to optimize CMT (multi-thread CMPs) performance. However, due to increasing heterogeneity within the processors as well as across different processors, application placement is becoming increasingly important. Qtop uses simulated annealing to affect application placement while leaving the scheduling decisions to the kernel.

2.4 Conclusion

Application and physical heterogeneity can be exploited in systems to reduce interference on multicore processors by smart application placement, i.e. applications contending on a certain resource should be placed on cores that do not share that resource, while allowing symbiotic applications on cores sharing resources. Qplacer uses Quality Time to efficiently improve application placement, which can lead to better resource utilization as well as throughput without requiring application modification.

Acknowledgment

Parts of these chapters are reprinted from the following papers:

- Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford. "Quality Time: A Simple Online Technique for Quantifying Multicore Execution Efficiency". To be submitted at the International Symposium on Performance Analysis of Systems and Software, ISPASS 2014.

Permission to use these contents has been obtained through signed letters from the co-authors. Dissertation author was the primary investigator and author on this paper.

Chapter 3

TimeCube: Measuring Execution Quality in Hardware

Multicore processors are common place in datacenters and have already become ubiquitous in some embedded domains, such as smart phones. In these, as in other domains utilizing multiprocessors, there is a trend toward greater concurrency that will soon move us from an era of multicore designs into an era of manycore designs. Manycore processors are attractive for datacenters as well as embedded applications because they optimize energy per operation [BFPS11] for high compute workloads as demonstrated by recent manycore offerings, such as Tile Gx100 [Sch10] and Intel's SCC [HDH+10].

The growth in core-count comes with an increase in concurrency and a lower per-core resource availability, as shown in Table 2. This leads to increased interference on manycore processors, resulting in large unpredictable slowdowns, as shown in Figure 3.1. I proposed using Quality Time in space-multiplexing multicore processors, as a substitute for CPU-time on unicore processors, to provide the notion of interference-free progress and make high-level decisions, such as tracking performance as well as scheduling or distribution of resources. We can continue to use Quality Time for making these decisions in manycore systems as well.

In Chapter 1, I presented a software tool, called Qtime, to measure Quality Time

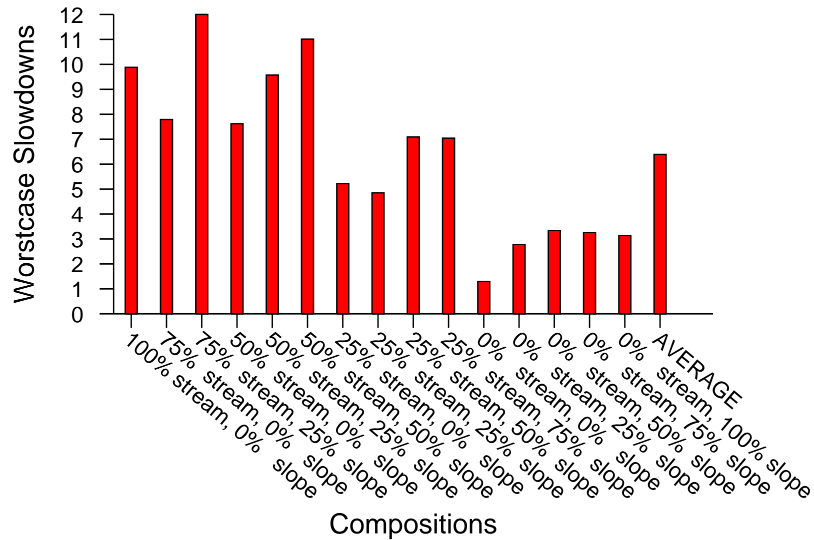


Figure 3.1. ***Worst-case Application Slowdowns on a simulated 32-core processor.*** On a simulated 32-core processor (see Section 3.3.1 for details), we see that slowdowns are both large, $6\times$ on average, as well as highly variable, from less than $2\times$ to as much as $12\times$ in the worst case. Thus, CPU-time should not be used as a proxy for progress when making high-order decisions in embedded systems or datacenters using space-multiplexed manycore processors.

of live applications in user-space without any code modification or recompilation. While Qtime is able to track Quality Time with low error (8.18% on average) and low overheads ($< 1\%$ per application), the lack of knowledge of hardware internals limits the accuracy that can be achieved in software. Moreover, it requires sampling of applications in standalone mode, which leads to relatively large execution overheads. In this chapter, I propose a hardware mechanism to measure live application Quality Time with no performance overheads, and the mechanism's knowledge of the hardware internals leads to a significantly higher accuracy as well.

I describe *TimeCube*, a manycore processor that is augmented by hardware to efficiently enable the simultaneous and online estimation of Quality Times for all applications with a high degree of accuracy using shadow performance modeling. TimeCube

is modeled along the lines of typical commercial manycore processors, such as Tile GX and Intel Xeon Phi, where many application threads execute concurrently on independent cores while sharing architectural resources, such as last-level cache, memory bandwidth, DRAM channels, etc. Their exists a hierarchy of caches, and though all these cache levels are kept coherent for an application, coherence of data shared across multiple applications are kept coherent through software.

I also present detailed experimental evaluation of a 32-core instance of TimeCube, which shows that the shadow performance modeling provided in TimeCube enables online estimation of application Quality Time with an average error of less than 1% on a 32-core concurrent system with very low area and energy overheads, even when the slowdowns witnessed due to interference were 6× on average and as much as 12× in the worst case. The accuracy of Quality Time estimation makes it highly reliable for use in high-order decisions, as I show in subsequent chapters.

In summary, this chapter describes the following novel contributions:

1. **Quality Time** I develop *Quality Time* as a counterpart of application CPU-time for interference-free progress in computing domains using space-multiplexed manycore processors. These systems can use Quality Time to accurately track online application progress in hardware and make high-order decisions.

2. **TimeCube** I describe TimeCube, a manycore processor, which provides online Quality Time estimation with very low energy and area overheads and no performance penalties.

3. **Efficient and accurate estimation of live Quality Time in hardware** I present a detailed manycore evaluation of the shadow performance modeling used by TimeCube, and show that it can track Quality Time with less than 1% error, even in the presence of 6× average slowdown.

The remainder of the chapter proceeds as follows. Section 3.1 presents an overview of TimeCube design. Section 3.2 explains the shadow performance modeling used by TimeCube to calculate live Quality Time. Section 3.3 provides a detailed manycore evaluation of TimeCube. Section 3.4 discusses related work and Section 3.5 concludes.

3.1 TimeCube Overview

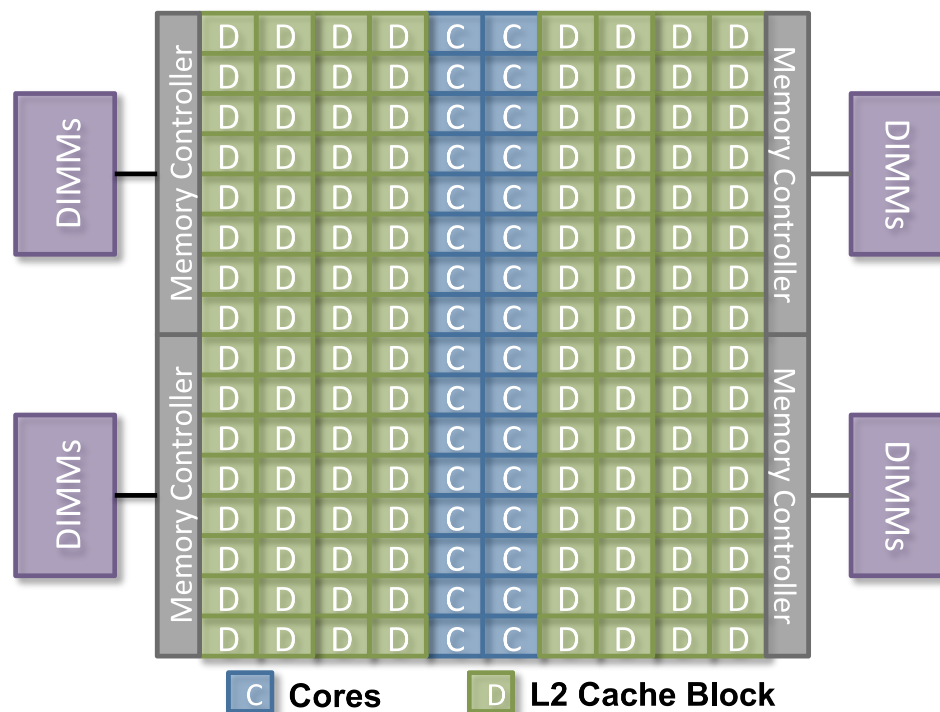


Figure 3.2. ***TimeCube Layout.*** TimeCube is a scalable architecture (b) with spatially distributed cores (C) and L2 cache blocks (D) connected over a network-on-chip, or NOC. TimeCube allows many applications to execute concurrently while sharing these architectural resources.

TimeCube is a manycore processor with abundant architectural resources, such as multiple levels of caches for instruction as well as data, many DRAM prefetchers, multiple DRAM channels, main memory bandwidth, IO bandwidth etc. This is a scalable

architecture, and in order to keep the access latencies of these resources low, while maintaining their energy-efficiency, they are spatially distributed across the processor, as shown in Figure 3.2, over tiles which are accessed using dynamic on-chip mesh networks. Multiple applications can execute simultaneously on TimeCube; even more than the number of cores, since it supports temporal-multiplexing. Every application executes on an independent core with private L1 data and instruction caches and a DRAM prefetcher These applications share the last-level caches, memory bandwidth, and DRAM banks, similar to existing commercial manycore processors [SB08].

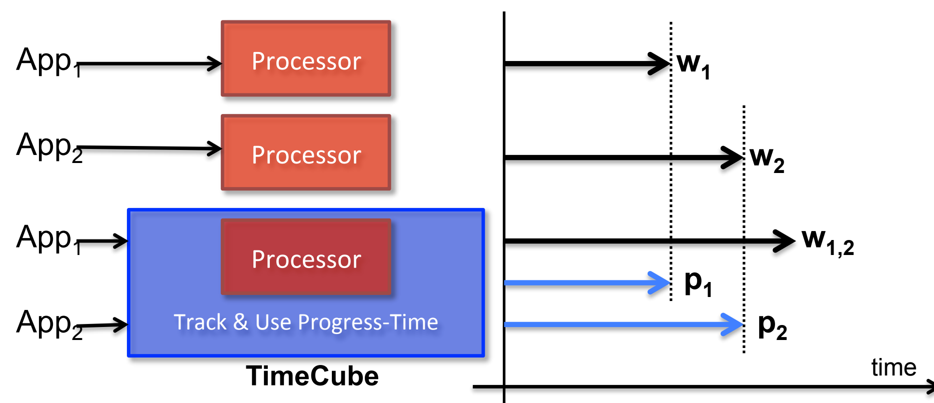


Figure 3.3. ***Augmenting manycore processors to measure Quality Time in TimeCube*** TimeCube uses shadow performance modeling to estimate the quality times (v_1 and v_2) for live applications (App_1 and App_2) running simultaneously for CPU-time $w_{1,2}$. Application quality times are equal to their standalone CPU-times (w_1 and w_2).

The interference resulting from this resource sharing breaks the correspondence between CPU-time and actual performance on space-multiplexed manycore systems, which can lead to erroneous estimations regarding progress of execution, even in the presence of state-of-the-art virtualization techniques [GLKS11]. In existing multicore systems, many high-order decisions, such as resource allocation and scheduling, are done in accordance to application progress. Therefore, we propose that these decisions should not be made based on application CPU-times, but their *Quality Times*: the amount of

time required for an application to complete the same amount of work it has done so far, were it to have been allocated *all* CPU resources. TimeCube is augmented with hardware mechanisms to efficiently and accurately track application progress. It uses shadow performance modeling to estimate the Quality Time simultaneously for all concurrent applications running in the system, as shown in Figure 3.3.

While offline techniques [Chu04][SDR02a] have been proposed to measure the architecture-specific interference between applications, online progress tracking allows us to handle previously unseen applications, handle live input data for known applications, capture phase-specific interference, and finally provide better online control over application progress rates. TimeCube tracks the online application Quality Time with no performance penalties as the shadow performance modeling is done using dedicated hardware on each core in parallel with application execution. In order to strike a balance between shadow performance modeling overhead, which is proportional to the Quality Time update frequency, and its fine-grained accuracy, which improves with update frequency due to changing application phases, an application's Quality Time is updated after regular time-intervals of 1ms.

3.2 Shadow Performance Modeling in TimeCube

Shadow Performance Modeling allows an application's standalone performance to be estimated with a high degree of accuracy using an extrapolation of its actual execution. TimeCube takes into account micro-architectural resource usage to accurately estimate quality time for a single manycore chip. For taking into account system-level resources, we would need to isolate and estimate performance based on all system components including network and I/O.

TimeCube calculates quality time using an analytical performance model that uses execution statistics collected through shadow hardware structures placed on every

core. TimeCube updates the Quality Time, which is stored in the cores, in parallel with application execution, after regular execution intervals. I now describe TimeCube's analytical model to calculate quality time for application i for an execution interval j, if it were allocated all the system cache and memory bandwidth.

$$\begin{aligned} ExecTime_j[i] &= const_j + (L2Hit_j[i] \times L2HitLatency_j[i]) \\ &+ (PrefHit_j[i] \times PrefHitLatency_j[i]) \\ &+ (PageHit_j[i] \times PageHitLatency_j[i]) \\ &+ (PageMiss_j[i] \times PageMissLatency_j[i]) \\ &+ (PageCnfl_j[i] \times PageCnflLatency_j[i]) \end{aligned} \tag{3.1}$$

TimeCube's analytical model estimates the hypothetical execution time for the work done by an application in the last interval, but for an arbitrary cache and bandwidth allocation, by estimating the delays caused by the in-order [1] L1 private cache misses in the shared L2 cache, prefetcher and the DRAM (Equation 3.1).

We assume that the in-core execution time remains unaffected by changing cache and bandwidth allocations. We represent this in-core execution time by $const_j$ for interval j in our analytical model. We collect this value for the current execution by counting the cycles for which the application executed while discarding the cycles spent waiting on L1 private cache misses. We then use this value as the in-core execution time for calculating application quality times for the next interval for all possible cache and bandwidth allocations. The time spent inside I/O calls is included within the cycles spent inside the core ($const_j$), not waiting for the memory system. We assume this time to be

[1] TimeCube has an in-order memory system, like RAW [MBT04], where the core is stalled during its memory miss. Thus, there is no miss concurrency for a single application.

independent of the cache and memory bandwidth allocation.

To find the time spent in L2 caches we use a shadow cache structure, described below, to estimate the L2 cache hits if the application was allocated all the cache in the system. We measure the average L2 hit latency for the current cache and bandwidth allocation, and use it for our estimation for the next interval. Similarly, we use a shadow prefetching structure, described below, to estimate the number of prefetch hits, while using the average prefetch hit latency from the current execution. When a request misses both L2 and the prefetcher, it is served by the main memory. We measure the DRAM page hit, miss and conflict latencies for the current execution, and use them along with the page hit, miss and conflict rates calculated using a shadow DRAM structure, described below, to calculate the remaining components of our analytical model.

With this model we calculate quality time for all applications if each of them was exclusively given all the cache and memory bandwidth in the system for next interval. According to this model we need to estimate certain shadow L2 cache, prefetcher, and DRAM statistics to estimate the execution times for full resource allocations. We use the following shadow hardware structures to collect these shadow statistics:

- *Shadow-Tags* [MQ06] provide an efficient hardware mechanism to estimate the cache miss rates for any arbitrary cache size. In order to reduce the shadow cache overheads we use set-sampling.

- *Shadow Prefetchers* run a dummy prefetching algorithm by tracking miss streams and launching *fake* prefetches, i.e. while the prefetch request is created, no actual data request is sent to the memory system, and maintain shadow statistics such as prefetches issued, prefetch hit rate, and prefetch hit latency.

- *Shadow Banking* tracks the current state of the DRAM row buffers by modeling DDR behavior for DRAM requests and maintains shadow statistics such as page

hits, misses, and conflicts. Our experimental results suggest that shadow banking may not be absolutely essential for this system; using a fixed memory latency imparts on average an error of only 2% in estimating application quality times.

We use one shadow-tags structure, one shadow prefetcher and one shadow banking structure per core.

$$ReqBW_j[i] = \frac{L2Misses_j[i] + PrefRqs_j[i] - PrefHits_j[i]}{ExecTime_j[i]} \tag{3.2}$$

$$Performance_j[i] = \begin{cases} \frac{Instructions_j[i]}{ExecTime_j[i]}, if\ ReqBW_j[i] \leq B_{total} \\ \frac{Instructions_j[i] \times B_{total}}{ExecTime_j[i] \times ReqBW_j[i]}, otherwise \end{cases} \tag{3.3}$$

The hardware also needs to estimate bandwidth stalls to estimate application performance. We use cache misses and prefetch statistics to calculate the required bandwidth (Equation 3.2). If the allocated bandwidth exceeds required bandwidth then we assume no bandwidth stalls. Otherwise, the bandwidth stalls are accounted for by reducing performance by the ratio of required and allocated bandwidths (Equation 3.3). This is based on the assumption that the memory requests are uniformly randomly distributed over program execution.

$$Quality\ Time_j^{inc}[i] = \frac{Real\ Performance_j[i]}{Performance_j[i]} \times IntervalTime \tag{3.4}$$

$$Quality\ Time_i = \sum_{interval\ \ j} Quality\ Time_j^{inc}[i] \tag{3.5}$$

We calculate the application's quality time for this interval for full cache and bandwidth allocation by multiplying the interval-time with the ratio of the performance for current execution and the one with all the cache and memory bandwidth allocated (c_{total},b_{total}), as shown in Equation 3.4. TimeCube sums up an application's quality times for all past intervals to get its total quality time (Equation 3.5).

TimeCube, in-line with existing commercial manycores like Tile64, uses wimpy cores and in-order memory systems to provide energy-efficiency with high throughput. However, performance for out-of-order cores can be modeled as well, as shown by Moreto et al. [MCRV]. Moreover, even though TimeCube is designed for multiprogramming rather than parallel programming, it is reasonable to believe that the techniques outlined here would support consolidated multi-threaded applications as well if given their associated performance models.

Quality Times can also be used on multi-chip multicore systems, and potential processor heterogeneity can be managed in a way similar to the existing heterogeneous systems, which calibrate processor performances over a workload. TimeCube can likewise re-normalize the quality time estimates over heterogeneous processors.

3.3 Results

In this section, I will describe an experimental evaluation of the shadow performance modeling done by TimeCube. I will present a manycore evaluation methodology followed by the model's accuracy results.

3.3.1 Evaluation Methodology

In this section, I describe the processor model used for evaluation and the benchmarking methodology. The evaluation prototype is modeled along the lines of commercial manycore processors (e.g. Tile64 [SB08]). Each core is superscalar, i.e. it can simultaneously execute multiple instructions, but the memory system requests are sent in-order. A reconfiguration interval of 25 million cycles is used. The evaluation is based on PTLsim [You07] and a memory-system emulator to simulate execution of multiple applications on a single many-core chip while sharing last level cache and off-chip memory. The emulator internally uses DRAMsim2 [WGT^{+}] for modeling

Table 3.1. The processor model used to evaluate TimeCube is along the lines of typical commercial manycore processors, such as Tile64 [SB08].

Cores	32, x86-64 ISA, 3GHz, superscalar, in-order memory
L1 cache	32KB inclusive, 4 way associative, 8 word line, 1 bank, 3 cycle hit, pipelined, 1 read/write port
L2 cache	128 cache-arrays, 1 bank per cache-array, 128KB per bank, 8 word line, 4-way associative, pipelined, 1 read/write port
Network	64-wide, mesh, dynamic router, 1-cycle hop
Prefetcher	stream prefetcher, 128 streams, 32 buffers
Memory	4 controllers, bit-interleaved, 4 DIMMs/channel, 4 Ranks/DIMM, 8 Banks/Rank, 64MB/Bank, 16 Banks and 1GB DDR3 per core, 96Gb/s memory bandwidth

details of the DRAM memory system. Detailed specifications of the evaluation model are presented in Table 3.1. I analytically model the area and power consumption using area and energy numbers obtained from RAW [MBT04] and McPAT [LAS$^+$09] scaled to 45nm, as specified in Table 3.2. In order to reduce simulation run times, I extract application representative phases using SimPoint [SPHC02] and then concurrently run SimPoint combinations.

Benchmarks and their Classification In order to simulate a typical manycore processor workload, I run combinations drawn from 26 benchmarks that span SPEC2K, SPEC2K6, and an I/O intensive benchmark suite I developed internally to model data-intensive workloads, as shown in Table 3.3. This selection provides a rich spectrum of cache and memory characteristics, as well as instruction level heterogeneity as shown by uops/inst, and includes applications such as web crawlers, photo filters, face detection, computer aided design tools, scientific computations, data compression, parsing, image recognition, and security algorithms.

The manycore evaluation space, where I run all possible benchmark combinations,

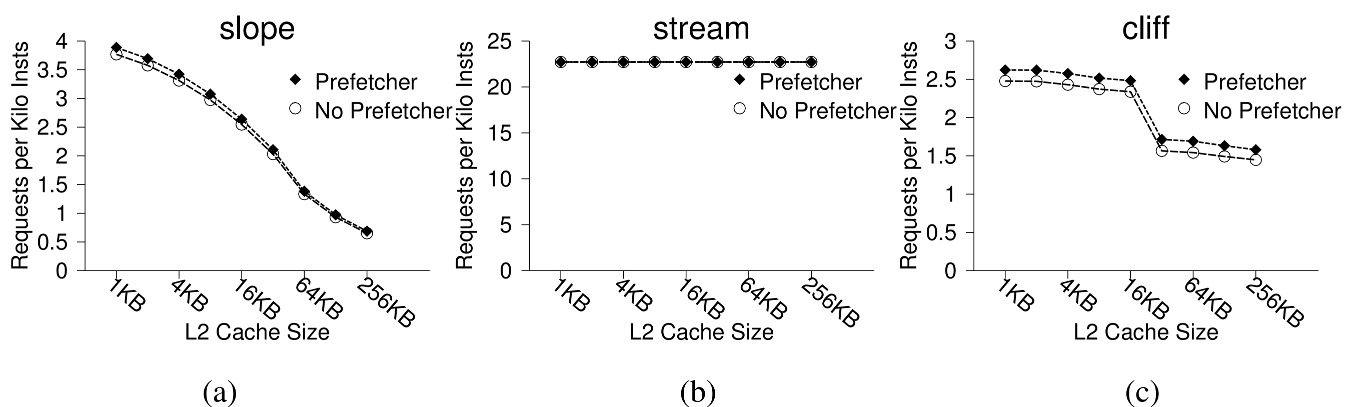


Figure 3.4. ***Benchmarks can be classified based on the sensitivity of their miss rate to L2 cache sizes.*** For some applications like bzip2 (a) cache size has a steady impact on miss rate, while for others like apsi (b) it has no effect, and some applications like mgrid (c) have a cliff-like profile.

Table 3.2. I derive energy and area numbers for TimeCube evaluation using RAW [MBT04] and McPAT [LAS+09] scaled to 45nm.

Operation	**Energy**
Instruction Execution	57.2
L1 Tag Match	22.5
L1 Data Read	36.0
L1 Data Write	38.2
L2 Tag Match	42.2
Quality Time Calc	53.4
Shadow-Tag Shift	21.1
L2 Data Write	70.9
Memory Read	5230.1
Memory Write	5120.0
L2 Data Read	65.7
Network Send	6.2
Network Rcv	6.4
Network Hop	4.3

is very large. Moreover, it provides no intuition about the benchmarks that I have not included in our evaluation. In order to limit the evaluation space as well as incorporate a structure into our evaluation, I classify our benchmarks according to a three-type taxonomy, and then examine runs that include different ratios of the three types. The taxonomy is as follows: An application which sees no drop in miss rate with increasing cache size is a *stream* application, an application which sees a sudden drop in miss rate with cache size is a *cliff* application, and an application whose miss rate drops

gradually with increasing cache size is a *slope* application, as described in Figure 3.4. I can then run representatives of these classes to estimate behavior of similar applications to refine our manycore evaluation space. For our experiments, I run workloads with incrementally changing composition of benchmarks classes. For each composition, I run all possible combinations of benchmarks within every benchmark class, and report the arithmetic mean of their results. In the applications examined, cache sensitivity was a strong classifier that predicted other characteristics, such as stream applications having good prefetching behavior and high bandwidth requirements. For a workload with high variance within cache sensitivity categories, additional classification axes would be beneficial.

Table 3.3. I use benchmarks from SPEC2000, SPEC2006 and some IO applications, which provide a diverse mix of memory characteristics such as miss rates in L1, hit rate in L2, and cache miss profiles for TimeCube evaluation.

Benchmark	uops /Inst	32KB MPKI	L2Hit 128KB	L2Hit 16MB	Type
IO/webCrwlr	1.67	8.01	12.27%	20.52%	slope
IO/fotoBlur	1.69	11.34	10.79%	17.22%	slope
CFP2000/wupwise	1.68	15.37	0.07%	1.11%	cliff
CFP2000/swim	1.68	28.86	0.00%	56.47%	cliff
CFP2000/mgrid	1.68	2.52	0.00%	35.96%	cliff
CFP2000/applu	1.68	2.56	4.38%	7.87%	strm
CINT2000/vpr	1.65	11.82	8.99%	87.82%	slope
CFP2000/art	1.68	45.02	0.00%	0.00%	strm
CFP2000/equake	1.67	11.41	7.68%	11.89%	strm
CINT2000/astar	1.71	1.47	27.36%	40.57%	slope
CINT2000/bwaves	1.73	0.17	0.39%	2.01%	cliff
CFP2000/h264ref	1.67	1.54	18.57%	59.90%	slope
CINT2000/hmmer	1.68	2.60	2.70%	84.69%	cliff
IO/faceDetect	1.71	0.27	60.81%	60.96%	strm
IO/diskBckup	1.70	9.33	12.87%	19.21%	slope
CFP2000/ammp	1.68	9.14	3.21%	97.25%	slope
CFP2000/lucas	1.73	5.38	0.00%	0.04%	strm
CFP2000/fma3d	1.73	3.44	4.05%	22.53%	cliff
CINT2000/parser	1.65	8.52	11.32%	97.72%	slope
CINT2000/bzip2	1.70	2.21	3.08%	78.48%	slope
CINT2000/twolf	1.65	19.03	3.55%	88.51%	slope
CFP2000/apsi	1.68	22.64	0.00%	00.00%	strm
CFP2000/namd	1.71	2.49	62.77%	87.94%	slope
CINT2000/sjeng	1.70	1.08	53.55%	74.23%	slope
CFP2000/soplex	1.71	2.56	10.19%	56.23%	slope
CINT2000/specrnd	1.65	0.06	4.17%	4.31%	strm

3.3.2 TimeCube's Quality Time Estimation is Highly Accurate

In this section we evaluate the shadow performance modeling mechanisms for estimating Quality Time in TimeCube. For validation of the performance estimation model, we measure an application's estimated standalone performance in concurrent mode, or Quality Time, and compare it to its actual performance in the standalone mode. I ran experiments using the methodology explained previously, and the results show that TimeCube is able to estimate an application's standalone performance and its estimated slowdown with just about 1% average error, as shown in Table 3.4. Moreover, TimeCube was also able to track Quality Time for all the benchmarks with very high accuracy, as shown in Table 3.5. Thus, we can reliably use TimeCube's shadow performance modeling to estimate Quality Time for progress measurement and resource management in concurrent multicore systems.

3.3.3 Area and Energy Distribution in TimeCube

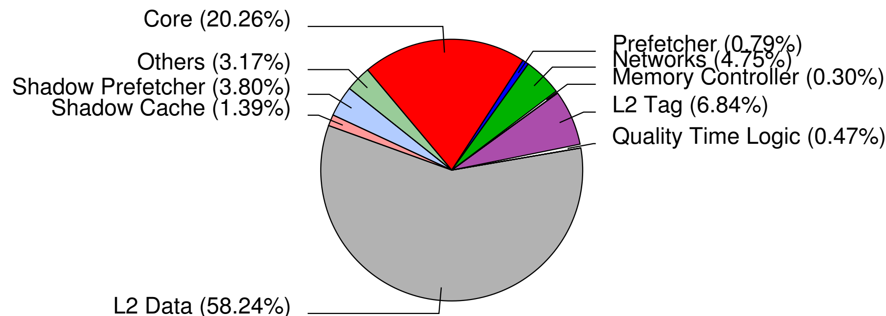


Figure 3.5. ***Area distribution in TimeCube for calculating Quality Time.*** The area consumed by Shadow Tags, Shadow Prefetchers, and Quality Tables is small (1.39%, 3.80%, and 0.47% respectively).

I now analyze the area and energy distribution for Quality Time estimation in TimeCube. The microarchitectural mechanisms required to estimate Quality Time

Table 3.4. TimeCube estimates Quality Time with only 1% average error over a spectrum of benchmark compositions, so they can be reliably used for progress measurement and resource management.

strm (%)	**slope** (%)	**cliff** (%)	**error** (%)
100	0	0	0.39
75	0	25	0.41
75	25	0	0.27
50	0	50	1.26
50	25	25	0.51
50	50	0	0.01
25	0	75	1.72
25	25	50	0.49
25	50	25	0.19
25	75	0	0.16
0	0	100	7.04
0	25	75	1.76
0	50	50	0.61
0	75	25	0.40
0	100	0	0.35
AVERAGE			1.01

consume less than 6% chip area. Shadow-Tags consumes 1.39%, Shadow Prefetchers 3.80% and Quality Tables 0.47%, as shown in Figure 3.5. For an example 32 application mix, my experiments revealed that the energy consumed for supporting Quality Time is low, just 0.01%, while shadow structures consume only about 0.20% energy. Therefore, the mechanisms for measuring Quality Time in TimeCube are energy and area efficient.

Table 3.5. TimeCube uses performance estimation mechanisms to postulate application slowdown when ran concurrently with other applications. Our experiments reveal that our estimates are correct within roughly 1% of the actual slowdowns, on average for all the benchmarks over a spectrum of compositions.

Benchmark	**error (%)**
164.gzip	0.05
168.wupwise	3.68
171.swim	6.18
172.mgrid	0.08
173.applu	2.62
175.vpr	0.03
179.art	0.07
183.equake	0.07
187.facerec	1.13
188.ammp	0.35
189.lucas	2.90
191.fma3d	0.37
197.parser	0.43
256.bzip2	0.05
300.twolf	0.03
AVERAGE	1.20

3.4 Related Work

Interference in manycore systems. The emergence of manycore computing in the server space, punctuated by the arrival of Tilera's Tile Gx100 [Sch10] and Intel's 48-core SCC [HDH+10], offers higher density and energy-efficiency. However, these benefits are only realizable if interference is more carefully controlled, as shown by Tang et al. [TMV+11], as these manycore processors heavily rely on shared resources. We are also witnessing an emergence of multicores in other ecosystems such as embedded

computing on the smartphones. These systems are under an even bigger pressure to share resources due to limited area budgets, and this can lead to difficulties in existing resource management techniques for multiprogrammed embedded systems, such as the ones proposed by Lipari et al. [LB00], Bernat et al. [BB02], and Beccari et al. [BCZ05], reducing the effectiveness of techniques such as resource reservation [AB04] and proportional resource sharing [SAWJ+96] for real-time systems.

Govindan et al. [GLKS11] show that even with the use of software mechanisms, such as hypervisors, the unpredictability in slowdowns when sharing architectural resources is very high. Stillwell et al. [SSVC09b] also examined the performance impact of resource sharing in servers at the system level, reducing the effectiveness of techniques such as resource reservation [AB04] and proportional resource sharing [SAWJ+96] for real-time systems. For resource-sharing embedded systems, it is important to accurately estimate the progress of applications and exercise control over it in order to maintain performance guarantees and improve resource utilization, as also pointed out by Buttazzo [But06]. Our novel Quality Time abstraction quantifies the utility of resource partitions for applications, and allows a quantification of application progress on manycore systems that can be used to both measure and control application execution.

Shadow Performance Modeling. TimeCube creates quality times using an analytical performance estimation model similar to the one proposed by Solihin et al. [SLT99]. Performance estimation of an application for an artificial machine configuration is non-trivial. Some previous works have proposed algorithms and mechanisms to do application performance estimation. Emma et al. [Emm97], Solihin et al. [SLT99] and Luo et al. [LLW+98] give a performance estimate for an application performance. I further enhance the model by tracking prefetches, adding memory bandwidth constraints by tracking dirty lines, similar to the mechanism proposed by Kaseridis et al. [KSCJ10], and

modeling the details of DRAM DDR protocol and bank buffer behaviors. For TimeCube's model we need cache miss estimates for full cache allocation. Shadow cache techniques have been proposed for associative caches, such as by Chandra et al. [CGKS05], Iyer et al. [Iye03], Gecsei et al. [GST70], Hill et al. [Hil87], Suh et al. [SDR02b], Zhou et al. [ZPS$^+$04] and Suh et al. [SDR01]. These techniques are based on the LRU-stacking property [Hil87]. In order to reduce the shadow cache overheads we use set-sampling, as suggested by Qureshi et al. [MQ06].

3.5 Conclusion

Manycore processors have to tackle the challenge of interference due to space-multiplexing, which can cause large and unpredictable slowdowns if left unmanaged. Overcoming this hurdle can improve their usability for systems using multicore processors, which need to accurately measure application progress and maintain guarantees about quality of execution. Quality Time can be used to quantify application progress irrespective of resource heterogeneity. TimeCube, a manycore processor, uses shadow performance modeling to accurately estimate quality time with just 1% error for live applications with very low overhead. Overall, the results argue for adding the requisite micro-architectural structures to estimate Quality Time in manycore chips to allow accurate measurement of execution quality of concurrent applications.

Acknowledgment

Parts of these chapters are reprinted from the following papers:

- Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford. "DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache", International Conference on Computer Design, ICCD 2013.

- Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford. "TimeCube: A Manycore Embedded Processor with Interference-agnostic Progress Tracking", International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2013.

Permission to use these contents has been obtained through signed letters from the co-authors. Dissertation author was the primary investigator and author on these papers.

Chapter 4

Quality Tables: Controlling Execution Quality in Hardware

Manycore processors share microarchitectural resources, such as the last-level caches, memory bandwidth, etc., between applications to increase the resource utilization. However, the performance interference arising from contention for these shared resources poses several challenges: Resource interference erodes performance guarantees, impedes measurement and management of applications' memory demands, complicates resource allocation policies, and distorts metering for consumers in cloud-services.

While I propose Quality Time as a means to measure interference-agnostic execution quality of applications using either software or hardware mechanisms, these techniques are insufficient for a fine-grained control over execution qualities. Previous research has pressed upon the challenges in controlling execution qualities, leading to proposals for partitioning the shared resources. Static partitioning cannot fully utilize the resources because application working sets and memory access patterns are time-varying. As a result, many proposals explore dynamic partitioning of resources, such as cache and memory, for both space as well as bandwidth. These mechanisms together provide *Dynamic Execution Isolation*.

Dynamic Execution Isolation ensures that an applications execution, and

hence its Quality Time, is not *unpredictably* affected by other concurrently running applications.

Dynamic Execution Isolation can allow us to dynamically control an application's execution quality by precisely controlling its resource consumption. TimeCube includes hardware mechanisms that control interference over the critical microarchitectural resources, i.e., the last-level cache, memory bandwidth, and memory banks, by dynamically partitioning them. On the other hand, conventional runtime software mechanisms, such as virtual machine monitors and hypervisors, can handle interference between I/O threads contending for network bandwidth, or other such system-level resources, by applying thread priorities in the scheduler and/or using backoff algorithms to reduce contention.

The existing mechanisms for dynamic cache partitioning do not scale well in energy-efficiency when applied to manycore processors. I propose a novel dynamic cache partitioning scheme, called *DR-SNUCA*, which provides an energy-efficient way to reduce resource interference over shared caches on manycore chips. Our results show that using DR-SNUCA reduces energy consumption by an average of 16.27%, compared to associatively partitioned caches, such as DNUCA.

Even though dynamic execution isolation allows TimeCube to have a fine-grained control over the resources allocated to applications, it is insufficient to control an application's execution quality because the relationship between resource allocation and application performance is non-trivial. For example, 50% resource allocation does not guarantee 50% application performance. Moreover, slowdown experienced with 50% resource allocations varies widely across applications, and finally, slowdown due to 50% reduction in cache is not the same as slowdown due to 50% reduction in memory bandwidth, as I will show in Section 4.4.1. The correspondence between resources and performance varies not only across applications, but also across phases within an

application's execution. To find this correspondence, we need to find the performance that an application will get for its current phase for all possible resource allocations. I term a collection of these performance estimates as the *Quality Tables*, or qTables, as shown in Figure 4.1.

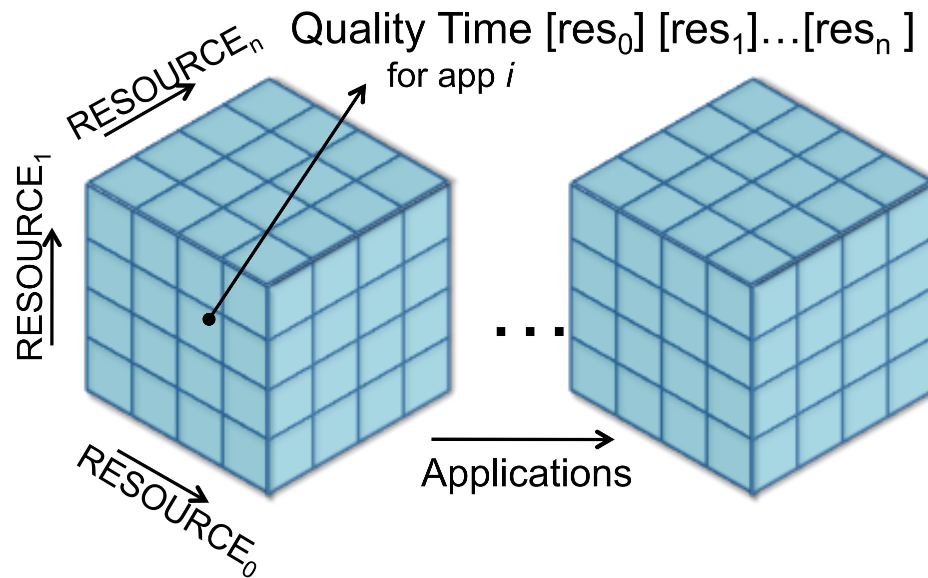


Figure 4.1. ***Quality Tables provide Quality Times for a spectrum of resource allocations.*** Quality Tables, or qTables, provide Quality Times for a spectrum of possible allocations of shared resources, say res_0, $res_1 \cdots res_n$, for all applications. These qTables are then used to make high-order decisions in TimeCube.

> **Quality Tables**, or qTables, are a collection of Quality Time values for a spectrum of shared resource allocations for the applications.

Quality Tables provide resource abstraction at the right granularity, i.e., concise enough to be calculated by the hardware during program execution and rich enough to be used in high-order decisions. Now even though there are numerous shared resources on a chip, we can isolate a critical subset of resources and provide the Quality Time for a range of possible allocations of these critical shared resources. Shared resources in

manycore processors, such as bandwidth, cores, and caches are countably discrete. Thus, we calculate the Quality Time only at discrete points to create qTables. The mechs are periodically (every 1ms) updated to account for the changing application phases.

Thus, TimeCube can use the Quality Tables to determine precisely how much resources are required for an application to attain a certain level of performance, and then it can dynamically allocate resources, partitioned under the dynamic execution isolation scheme, to the application and provide guarantees about its execution quality. A higher accuracy of Quality Time estimation allows TimeCube to provide stronger guarantees.

In summary, this chapter details the following novel contributions:

- **Dynamic Execution Isolation** I propose Dynamic Execution Isolation in manycore processors to enable a fine-grain control over microarchitectural resource allocation. TimeCube achieves this by dynamically partitioning critical shared architectural resources.

- **Quality Tables** I propose Quality Tables, or qTables, a collection of Quality Time estimates for a spectrum of shared resource allocations. An enhanced shadow performance modeling model is presented that creates the Quality Tables in hardware for live applications with low error and low overheads.

- **Dynamically Repartitionable Static NUCA** I introduce Dynamically Repartitionable Static NUCA, or DR-SNUCA, an energy-efficient dynamically partitionable shared cache for manycore processors.

- **Flattened Partial LRU Vector** I introduce Flattened Partial LRU Vector, a shadow-tag structure for DR-SNUCA that allows efficient Quality Time estimation when using DR-SNUCA in TimeCube.

- **Energy-scalable dynamic execution isolation** I present a detailed manycore eval-

uation that shows DR-SNUCA reduces energy-consumption in TimeCube by 16.27% while performing within 0.5% when compared to existing DNUCA cache designs.

The remainder of the chapter proceeds as follows. Section 4.1 describes Dynamic Execution Isolation for manycore processors. Section 4.2 explains the enhanced shadow performance modeling used by TimeCube to create Quality Tables. Section 4.3 introduces design details of the novel DR-SNUCA cache. Section 4.4 presents a detailed manycore evaluation of TimeCube's dynamic execution isolation and DR-SNUCA. Section 4.5 discusses related work and Section 4.6 concludes.

4.1 Dynamic Execution Isolation in TimeCube

TimeCube provides dynamic execution isolation by partitioning critical shared resources and dynamically allocating portions of resources to the competing applications after regular intervals, as shown in Figure 3.2. This partitioning of shared micro-architectural resources eliminates resource interference, and an application's execution is not affected by other concurrently running applications. The allocation is done dynamically to avoid under-utilization of resources, since different applications have different utility for on-chip resources, which can also vary over time. An application's working set size can change, average bandwidth utility can change etc. Thus, one time partitioning of resources is not sufficient. TimeCube checks the application behavior from time to time and can readjust the partitioning. In this work, I use a periodic interval based partitioning triggered using a periodic interrupt. It might lead to some unnecessary checks but this keeps it invisible to software, and therefore allows us to use legacy code.

There are many shared resources in manycore architectures, but here I focus on three resources critical to compute workloads: last-level cache, off-chip memory band-

width, and DRAM space. Contention over memory controllers in an in-order memory system is low - the situation for our NoC is similar. In a system with high contention in either resource, TimeCube could be extended with fair queuing arbiters [NLS07], or virtual channels, respectively, to provide dynamic isolation over these resources.

Along with the core, each application also gets a dynamically allocated portion of the shared last-level (L2) cache [1], a portion of the shared memory bandwidth and some statically allocated DRAM banks (determined in software). The partitioning and reconfiguration of resources is kept invisible to software, which allows us to use legacy code. The programs execute continuously and uninterrupted even while the resource partitions are being reconfigured.

I now present the dynamic cache partitioning mechanisms used for each of the shared microarchitectural resource in more detail.

Dynamic Cache Partitioning TimeCube partitions the shared last-level cache between applications to provide dynamic execution isolation. The last level cache itself is a NUCA cache, spatially distributed across the chip into a grid of cache-arrays. These cache arrays are clustered into data tiles, as shown in Figure 3.2, which are accessed in a scalable manner using a dynamic on-chip mesh network. Each cache-arrays is dynamically allocated to an application, and any application can be allocated a multitude of these cache-arrays, though only in powers of two in order to simplify the cache management. Each application's cache allocation is stored in its core, which is used by the application to determine where to look for its cache lines. There are multiple mechanisms possible to dynamically partition these cache-arrays, but TimeCube uses a novel *Dynamically Repartitionable Static NUCA*, or DR-SNUCA, cache design, which is explained in detail in Section 4.3. However, TimeCube can use any other dynamic cache

[1] TimeCube is a non-cache coherent architecture like Intel SCC [HDH+10]; inter-process coherence is handled by the OS through separate memory allocation.

partitioning scheme to provide dynamic execution isolation over the shared last-level cache.

Nesbit et al. [NLS07] described that it is not sufficient to merely partition the cache space to remove interference, since contention over a limited cache access bandwidth can also lead to interference. However, since TimeCube allocates every cache-array to only one application, there is no contention over the cache port. There is contention over the on-chip network that is used for accessing the cache-arrays, but experiments show that due to an abundance of on-chip network bandwidth, network interference is insignificant. In case of higher bandwidth contention, we can use virtual channels to provide dynamic execution isolation over the on-chip networks.

Dynamic Memory Bandwidth Partitioning TimeCube dynamically partitions the memory bandwidth between applications to reduce interference. Even if an application is given its allocated bandwidth, if the memory scheduling is not done fairly, the applications might have unpredictable slowdowns. TimeCube uses a fair queueing arbiter [NALS06], which does fair scheduling across applications while staying within their bandwidth quotas. The performance of individual applications can be further improved by using state-of-the-art memory traffic scheduling techniques, which may reorder application memory requests based on prefetcher accuracies [LMNP08], or the status of DRAM row buffers [RDK+00] etc. In order to limit the possible bandwidth allocations, TimeCube *bins* the bandwidth i.e. it allocates bandwidth only in multiples of a fixed percentage of total bandwidth (1%).

Static DRAM Partitioning DRAMs are typically composed of a number of banks that are fronted with a row buffer to reduce access latency on repeated accesses to a line. In order to reduce interference on DRAM banks, TimeCube splits the memory banks statically among the applications along with the corresponding row buffers; however, the number of DRAM banks allocated to an application is not fixed and depends on

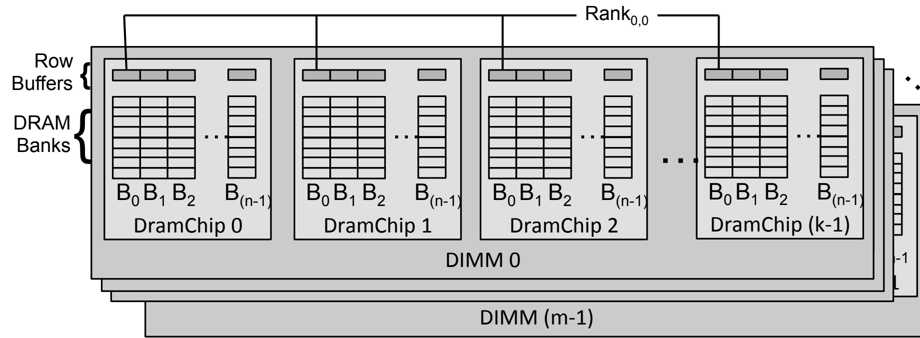


Figure 4.2. ***Static DRAM buffer partitioning in TimeCube.*** DRAM memory on every channel consists of multiple DIMMs. These DIMMs contain multiple ranks created by combining DRAM banks over multiple DRAM chips. TimeCube statically splits these banks among applications, along with their row buffers, to reduce DRAM space interference.

the amount of memory allocated to the application by the operating system. Thus, an application cannot alter the contents of another application's row buffers to unpredictably affect its memory access time. This bank partitioning is maintained at the memory controllers [2] , and the memory page allocator (OS) allocates pages to applications only on the memory banks assigned to them, as described by Liu et al. [LCX+12]. While this is a simpler approach compared to interference-prone performance-preserving techniques like ballooning [Wal], experiments done showed that bank partitioning does not significantly reduce performance for typical manycore architectures, since the memory-sensitive workloads are bottlenecked at the DRAM pin interface and not the DRAM row buffers.

TimeCube thus dynamically partitions the critical shared architectural resources to achieve dynamic execution isolation, which allows it to remove resource interference and exercise a fine-grained control over resources allocated to each application. Now, I describe how to calculate Quality Tables inside TimeCube on-the-fly, which will allow TimeCube to determine how much resource should be allocated to an application to

[2] TimeCube could leave DRAM management to software given support for dynamic execution isolation and shadow performance modeling.

guarantee a certain level of execution quality.

4.2 Quality Tables in TimeCube

TimeCube collects the Quality Time for a spectrum of shared resource allocations for all applications, as shown in Figure 4.3. These collections of Quality Times are called the *Quality Tables*, or *qTables*. We believe that qTables provide resource abstraction at the right granularity, i.e. concise enough to be calculated by the microarchitecture during program execution and rich enough to be used in high-order decisions, such as resource management in TimeCube. TimeCube updates the qTables, which are stored in the cores, in parallel with application execution, after regular execution intervals. TimeCube calculates Quality Time using an analytical performance model, which is the same as the one proposed in Section 3.2 except that it is extended to calculate Quality Time for all possible resource allocations simultaneously. It uses shadow statistics collected through enhanced shadow hardware structures, which track shadow statistics for all possible resource partitions simultaneously, placed on every core.

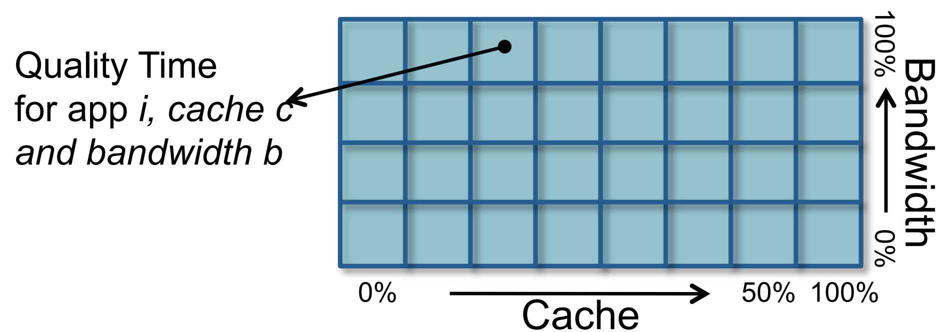


Figure 4.3. ***Quality Tables for TimeCube.*** TimeCube calculates Quality Times for all possible allocations of last-level cache and the memory bandwidth for each application. The bandwidth is *binned* and the cache-arrays are allocated in powers of two.

I now describe TimeCube's analytical model to calculate Quality Time for applica-

tion i for an execution interval j, if it were allocated c cache-arrays and b bandwidth-bins.

$$\begin{aligned} ExecTime_j[i,c] &= const_j + (L2Hit_j[i,c] \times L2HitLatency_j[i,c]) \\ &\quad +(PrefHit_j[i,c] \times PrefHitLatency_j[i,c]) \\ &\quad +(PageHit_j[i,c] \times PageHitLatency_j[i,c]) \\ &\quad +(PageMiss_j[i,c] \times PageMissLatency_j[i,c]) \\ &\quad +(PageCnfl_j[i,c] \times PageCnflLatency_j[i,c]) \end{aligned} \tag{4.1}$$

TimeCube's analytical model estimates the hypothetical execution time for the work done by an application in the last interval, but for an arbitrary cache and bandwidth allocation, by estimating the delays caused by the in-order L1 private cache misses in the shared L2 cache, prefetcher and the DRAM (Equation 4.1).

To find the time spent in L2 caches we use an enhanced shadow cache structure, described below, to estimate the L2 cache hits for the cache size c. We use a shadow prefetching structure, described previously, to estimate the number of prefetch hits and prefetches issued and measure the DRAM page hit, miss and conflict rates calculated using a shadow banking structure. We reuse the average L2 hit latency, prefetch hit latency, and the DRAM page hit, miss and conflict latencies for the current cache and bandwidth allocation.

$$ReqBW_j[i,c] = \frac{L2Misses_j[i,c] + PrefRqs_j[i,c] - PrefHits_j[i,c]}{ExecTime_j[i,c]} \tag{4.2}$$

$$Performance_j[i,c,b] = \begin{cases} \frac{Instructions_j[i]}{ExecTime_j[i,c]}, if\ ReqBW_j[i,c] \leq b \\ \frac{Instructions_j[i] \times b}{ExecTime_j[c] \times ReqBW_j[i,c]}, otherwise \end{cases} \quad (4.3)$$

The hardware now estimates bandwidth stalls to estimate application performance for all possible cache and bandwidth allocations. We use cache misses and prefetch statistics to calculate the required bandwidth (Equation 4.2), and then the resulting performance (Equation 4.3).

$$qTables_j[i,c,b] = \frac{Performance_j[i, c_{total}, b_{total}]}{Performance_j[i,c,b]} \times IntervalTime \quad (4.4)$$

$$Quality\ Time_i = \sum_{interval\ j} qTables_j[i, c_{alloc}, b_{alloc}] \quad (4.5)$$

Every cell in qTables stores the Quality Time for the corresponding cache and bandwidth allocation by multiplying the interval-time with the ratio of the performance for this allocation and the one with all the cache and memory bandwidth allocated (c_{total},b_{total}), as shown in Equation 4.4. TimeCube sums up an application's Quality Times for all past intervals, for the actual cache and bandwidth allocations (c_{alloc},b_{alloc}), to get its total Quality Time (Equation 4.5).

With this model we calculate Quality Time for all possible cache and bandwidth allocations for all applications for next interval. Now we need to estimate certain shadow L2 cache, prefetcher, and DRAM statistics to estimate the execution times for all possible resource allocations. We use the following enhanced shadow hardware structures to collect these shadow statistics:

- *Shadow-Tags* are enhanced to calculate cache miss rate for all possible power of two cache allocations. In order to reduce the shadow cache overheads we use LRU-stacking [MQ06].
- *Shadow Prefetchers* are the same as before, but now we need one instance per

cache allocation.

- *Shadow Banking* is the same as before, but now we need one instance per cache allocation.

We use one shadow-tags structure per core, and one shadow prefetcher and shadow banking structure per cache configuration per core. 40B are required to store an application's qTables. Our experimental results show that these mechanisms do not have significant area (2.46%) and energy (0.24%) overheads.

4.3 Dynamically Repartitionable Static NUCA (DR-SNUCA)

Sharing the last level of cache allows higher cache utilization than statically partitioned caches. TimeCube uses dynamic cache partitioning to provide dynamic execution isolation and control the resulting interference. To have a scalable last-level shared cache, TimeCube uses Non-Uniform Cache Access (NUCA) architecture [KBK02] that spatially divide the cache into multiple cache-arrays, accessed through a network-on-chip, or NoC, to provide energy-efficiency. For dynamic execution isolation, TimeCube requires a dynamically partitionable NUCA architecture, which continues to provide scalability in terms of the access energy.

In a NUCA cache, access energy depends heavily on the number of cache-arrays accessed for each request. Static Non-Uniform Cache Access (SNUCA) architectures use a fixed indexing function and accesses only one cache-array for every cache request, which keeps the energy consumption low, as shown in Figure 4.4. By configuring these index functions for each core, these SNUCA architectures can divide the cache into independent partitions specific to a given workload, thus providing resource guarantees; however, due to fixed hashing, the number of SNUCA cache sets cannot be dynamically

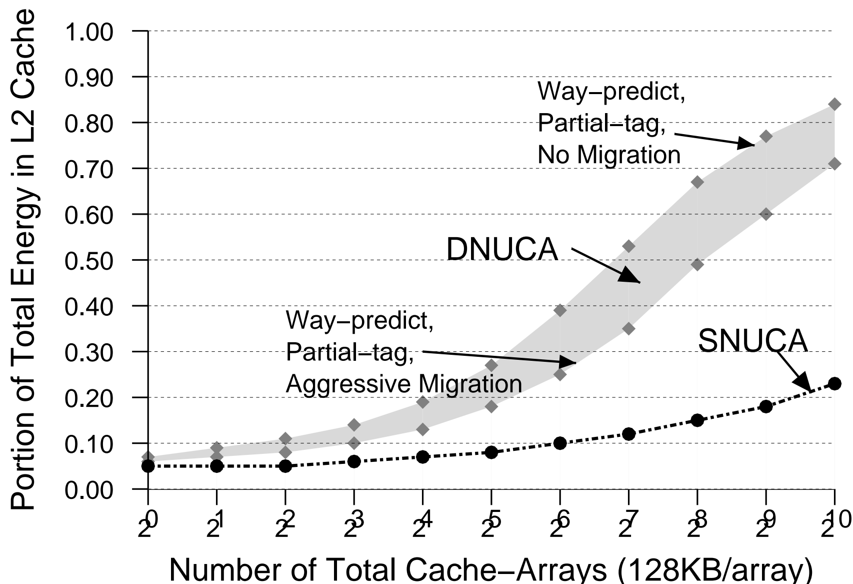


Figure 4.4. ***Associatively partitioned DNUCA caches are not energy scalable.*** The portion of energy spent in L2 caches increases with cache size when using DNUCA, even with XOR-based way-prediction [PAV+01], partial-tag match [KJLH89] and cache migration [HKS+05]. For SNUCA, the portion of energy spent in caches remains low, but SNUCA is not dynamically repartitionable.

changed. As a result, if we partition a shared cache into SNUCA components to keep the associativity fixed, cache allocations cannot be dynamically changed, which leads to reduced cache utilization due to changing application working sets.

The cache designs proposed by Nesbit et al. [NLS07], Hsu et al. [HRIM06], and Guo et al. [GSZI07], use associative partitioning to create a dynamically partitioned (DNUCA) cache. However, in DNUCA a cache line can be placed on any of the allocated cache-arrays, so it may need to check multiple cache-arrays for every cache access. Mechanisms such as XOR-based way-prediction [PAV+01], partial-tag match [KJLH89], and cache-block migration [HKS+05] reduce the number of cache-arrays checked per

cache access. However, Figure 4.4 shows that, even with these mechanisms[3], cache accesses in an associatively partitioned DNUCA consume a progressively larger portion of processor energy as cache size increases. This makes DNUCA based last-level caches energy-inefficient as aggregate cache size grows.

> *Insight*: While working sets change, necessitating dynamism in partitioning, they do not change rapidly. Thus, the frequency with which we can repartition does not need to be high and we should optimize performance and energy for the time *between* allocations.

I introduce *Dynamically Repartitionable Static NUCA*, or DR-SNUCA, a dynamically repartitionable shared cache with static-mapping during steady-state. DR-SNUCA provides energy efficiency and high cache utilization as well as fixed resource guarantees, and it does not interrupt execution during reconfiguration. This chapter presents a complete set of results for DR-SNUCA on a manycore processor, including architectural mechanisms that make the decisions as to how to reconfigure the caches at runtime without interrupting execution.

DR-SNUCA uses set partitioning i.e. growing or shrinking cache allocations by changing the number of sets allocated to an application while keeping associativity constant. DR-SNUCA uses indirect cache addressing to reduce reconfiguration overheads introduced during changes to cache allocations, to enable online reconfiguration. I also introduce *Tag-Duplication* to avoid execution stalls during the cache reconfiguration and keep DR-SNUCA's performance comparable to DNUCA.

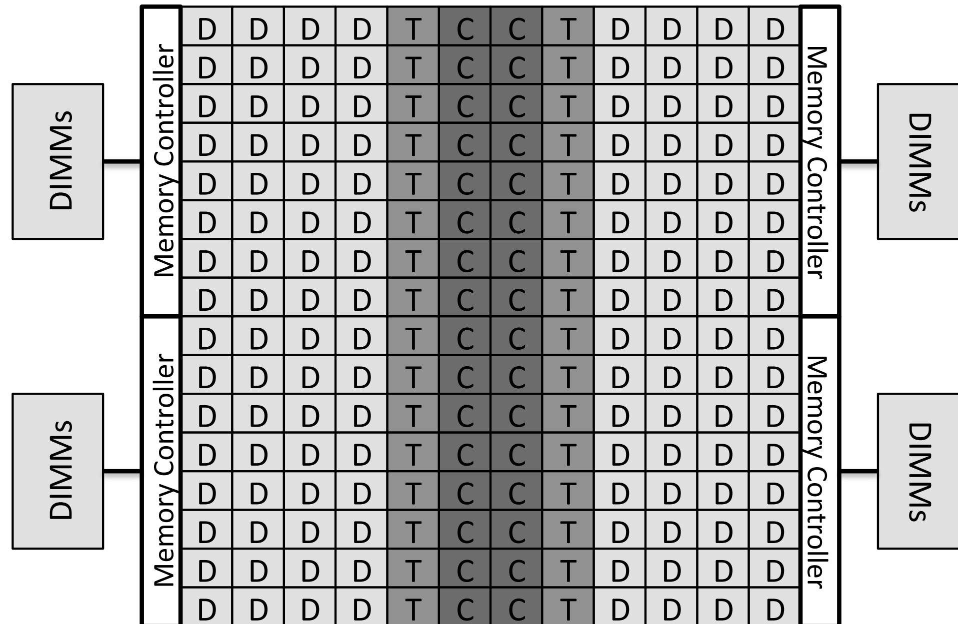


Figure 4.5. ***Layout of Dynamically Repartitionable Static NUCA*** Dynamically Repartitionable Static NUCA, or DR-SNUCA, spatially distributes the last-level cache shared by the cores (C) into cache-arrays(T,D) connected using an on-chip network, or OCN. It also physically separates the tag and data for each cache-array, consolidates them into tag-arrays (T) and data-arrays (D) respectively, and uses indirect cache addressing for cache accesses to reduce the online reconfiguration costs. The cache misses are sent to DRAMs through memory controllers.

4.3.1 Dynamically Repartitionable Static NUCA Design

DR-SNUCA provides a last-level shared cache for manycore processors that can be dynamically partitioned and is also energy-scalable. DR-SNUCA dynamically partitions the shared cache between applications and each application's cache portion operates as an SNUCA, except during reconfiguration periods. DR-SNUCA has physically separated cache-arrays connected through a point-to-point pipelined on-chip memory network, as shown in Figure 4.5, which are dynamically allocated to applications. When multiple cache-arrays are allocated to an application, they are merged by increasing the number of cache sets allocated to the application while keeping the number of associative

[3]For every cache size, I chose the partial tag size that minimized the overall energy consumption.

ways constant. DR-SNUCA allocates cache-arrays to applications in powers of two.

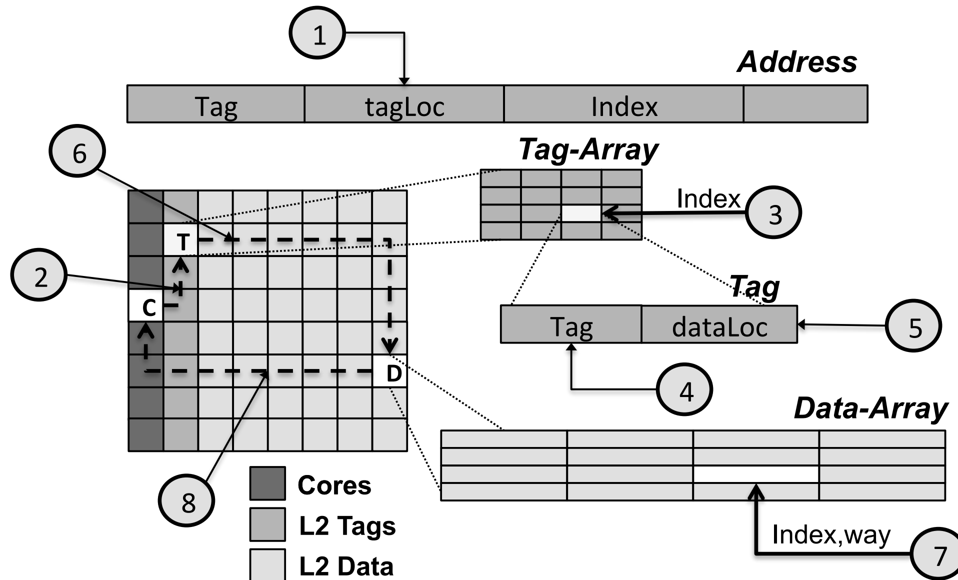


Figure 4.6. ***Cache access with Indirect Cache Addressing in DR-SNUCA.*** DR-SNUCA uses indirect cache addressing by separating the tags and data, and maintains a one-to-one correspondence between all tag and data locations by storing the corresponding data location in each tag using *dataLoc* bits. On a cache request, DR-SNUCA finds the tag array using *tagLoc* bits (1) in the address, sends a request to that array (2), and then uses index bits to find the set within that cache-array (3). If some tag matches (4), it uses the dataLoc in that tag to find the location of the data cache-array (5). A request is sent to the data-array (6), and the cache line is fetched (7) and sent back to the core (8).

In DR-SNUCA, when the cache allocation changes for an application, its number of cache sets are altered; as a result, its cache hashing changes as well. Naively moving all cache lines to their corresponding new locations would consume excessive energy. Instead, DR-SNUCA uses *indirect cache addressing* [AP93], shown in Figure 4.6, to greatly reduce reconfiguration costs. Indirect cache addressing separates the tags and data for each cache-array, maintaining them on physically separate locations in tag-arrays (T) and data-arrays (D), as shown in Figure 4.5, and storing the location of data in each tag using *dataLoc* bits. All these arrays are connected through a dynamic OCN. With this indirect addressing, cache reconfiguration moves only the tag and not the data, which

significantly reduces the reconfiguration costs, since cache tags are significantly smaller than lines.

For accessing a cache line under indirect cache addressing, the location of the tag-array holding the cache line tag is determined based on the *tagLoc* bits in the address, but the data can be placed on any of the data-arrays, as shown in Figure 4.6. An application's tagLoc width equals log_2 of the number of cache-arrays allocated. The cache set for an address in both tag-arrays and data-arrays is determined based on the index bits. The associative way for both tag and data-arrays is also the same for every cache line present in the cache, as shown in Figure 4.6. Thus, we only need to store the location of the data-array in dataLoc bits. To find an address location, we use *tagLoc* bits from the address, located just above the index bits, to determine the location of the cache-array that can hold the data for the application. We then use the index bits to determine the cache set within the cache-array, as shown in Figure 4.6. DR-SNUCA maintains a *one-to-one correspondence* between all tag and data locations in the cache at all times. This correspondence is useful in avoiding dead dataLoc references, or unreachable data locations, as well as improving the indirect cache access and reconfiguration performance, and can change only during cache reconfiguration.

Reconfiguration. Application cache allocations can dynamically change during application execution on interval boundaries. During this online reconfiguration we may have to shift some tags and evict some data while maintaining the one-to-one correspondence between all tag and data locations. During reallocation, we find the new tag locations for cache lines based on their tagLoc bits, and if it is not the same as their current locations, due to a possible change in the hashing scheme, we move the tags to their new blocks at the same index and way, as shown in Figure 4.7. The dataLoc bits in the existing tags at these new locations are also copied back to the old locations to maintain the tag and data location correspondence.

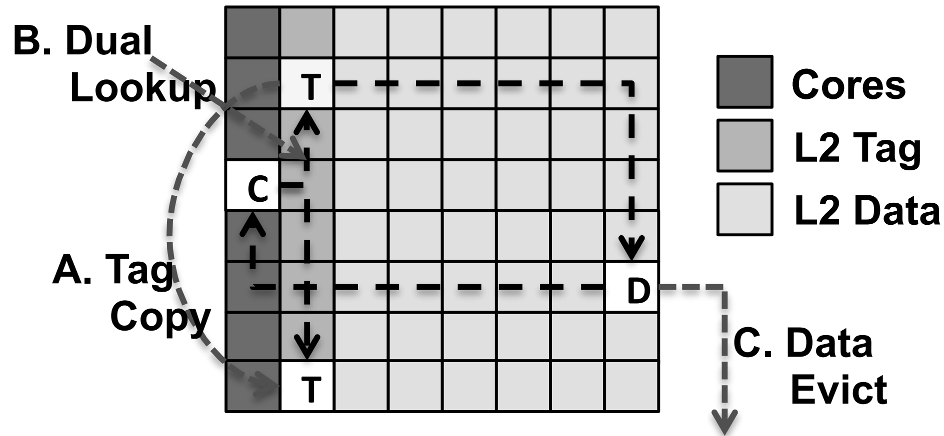


Figure 4.7. ***DR-SNUCA Reconfiguration*** If an application's cache allocation changes, its tags are copied (A) to their new locations in the duplicate tag-arrays for the next interval, while maintaining the tag to data location correspondence. Cache accesses are sent to both the new and old tag-arrays (B) during this reconfiguration period. If the new cache allocation is smaller, less recently used lines are evicted, and written back if dirty (C).

If the cache allocation increases, no change is required in the data blocks. However, if the cache allocation shrinks, we must select which cache lines to preserve and which to evict. To support this selection, in addition to the associative LRU within a set, we also maintain an LRU vector for every equivalent location (same index, way) across all cache-arrays currently owned by an application. On allocation reductions we evict the LRU entries for each equivalent location, as shown in Figure 4.7. We have to pro-actively evict these lines because if we fail to writeback all the dirty lines, the cache will become incoherent. We writeback only the dirty lines to save bandwidth. For the cache lines that are to be evicted, we still maintain the dataLoc bits in them in order to preserve the correspondence between tag and data locations.

Tag-Duplication. During cache reconfiguration, it is difficult to handle memory requests for a cache line whose tag is in transit. There are three basic approaches to handling this scenario. First, we could have a protocol to track the tag during its

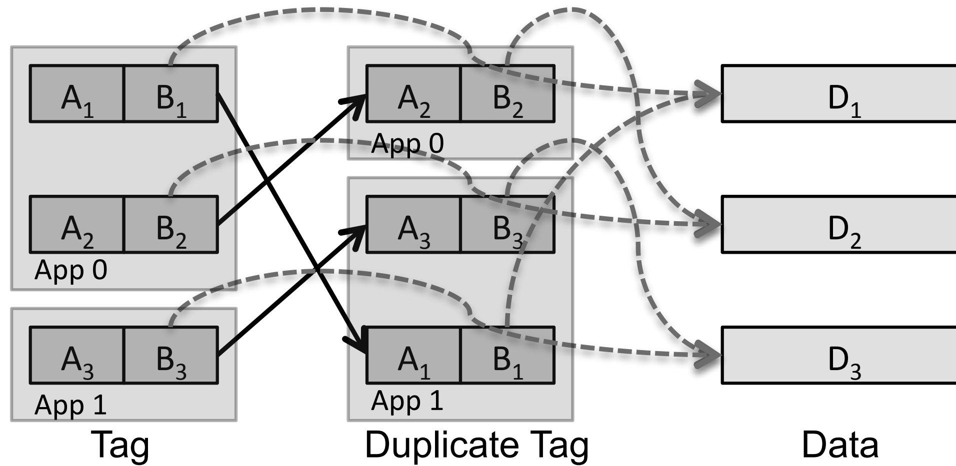


Figure 4.8. ***Tag-Duplication in DR-SNUCA*** During cache reallocation, tags are copied to their new locations in the duplicate tag-arrays for the next interval, while maintaining the tag to data location correspondence. All cache accesses are sent to both the new and old tag-arrays during this reconfiguration period.

transit and allow intermediate structures to respond, but this is complicated and can cost additional time and energy. Second, we could stall the application execution until the reconfiguration finishes, but this will reduce application performance. Third, *Tag-Duplication*, which maintains two tag block arrays and copies the tags from the arrays allocated for the current interval into the arrays allocated for the next interval, as shown in Figure 4.8. While the reconfiguration is going on, all tag lookups for an application are sent to the tag blocks allocated to the application for both the current and previous intervals, which guarantees that the tag will be matched if present in the cache. DR-SNUCA uses tag-duplication to prevent application stalling and handle memory requests during cache reconfiguration. Our experiments show that the reconfiguration period is relatively small compared to the interval length, which keeps the costs of dual-lookup during reconfiguration low.

4.3.2 Flattened Partial LRU Vector

In order to create the qTables, TimeCube needs to find the cache miss rates for all possible cache size allocations using an area and power-efficient mechanism. Shadow-Tags [MQ06] have been proposed for this purpose for partitioned caches with *total-LRU-order*. However, DR-SNUCA does not have total LRU-order, but a *partial-LRU-order*, which we describe next.

For a four-way DR-SNUCA, four addresses in set i, say A_1, A_2, A_3, A_4, have a total-LRU-order: $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4$. Similarly, four addresses in set j, say B_1, B_2, B_3, B_4, have another total-LRU-order: $B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B_4$. Merging these sets gives another total-LRU-order, say: $A_1 \rightarrow A_2 \rightarrow B_1 \rightarrow A_3$. These three total-LRU-orders together create a partial-LRU-order. DR-SNUCA has one partial-LRU-order per cache-index to include the sets with that index across all cache-arrays.

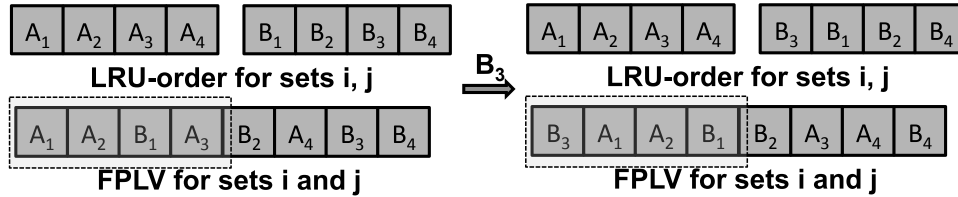


Figure 4.9. ***Flattened Partial LRU Vector*** The FPLV stores the partial-LRU-orders for DR-SNUCA using topological sorting; ordering is maintained using pointer-chains. For a w way cache, the first $w \times c$ locations contain the addresses present in cache configuration $log_2(c)$.

The *Flattened Partial LRU Vector*, or FPLV, stores a partial-LRU-order using topological sorting, and records the ordering using pointers, as shown in Figure 4.9. FPLV maintains another crucial property: For a w way cache, the first $w \times c$ locations contain the addresses present in cache-configuration $log_2(c)$. TimeCube maintains one FPLV per core.

FPLV efficiently determines cache statistics for all possible cache allocations[4] in DR-SNUCA. For every cache access by an application, its corresponding FPLV does tag-matching in only one cache set, the one where the cache line would have gone if the application was allocated a single cache-array. This is sufficient to check if it is a cache hit or a miss. If the address is indeed present, its location in FPLV will determine the cache-configurations for which the access would have been a hit. For updating the LRU ordering, in the worst case FPLV has to modify the position of tags within one cache set for every possible cache-configuration. For example, on seeing address B_3, which maps to set j, we only need to match B_1, B_2, B_3, B_4, and the LRU update would shift only the addresses B_3, A_1, A_2, B_1, A_3, and A_4, as shown in Figure 4.9. Thus, FPLV provides an area and power-efficient mechanism for finding cache miss rates for all cache sizes, since FPLV is accessed for only the requests going to the sampled sets.

4.4 Results

I now present the results for TimeCube's mechanisms to control applications' execution qualities. This includes the examples of the Quality Tables generated in TimeCube, the evaluation of Dynamically Repartitionable Static NUCA, and the overheads of implementing dynamic execution isolation in TimeCube. I use the evaluation methodology described in Section 3.3.1.

4.4.1 Quality Tables Created in TimeCube

In this section I present examples of Quality Tables generated by TimeCube using the enhanced shadow performance modeling with the enhanced shadow structures. As can be seen from Figures 4.10, 4.11, 4.12, and 4.13, these Quality Tables are highly variable across applications. I have normalized these Quality Tables so that they give the

[4] We use set-sampling [MQ06] to significantly reduce FPLV's area and energy requirements with a small loss in accuracy.

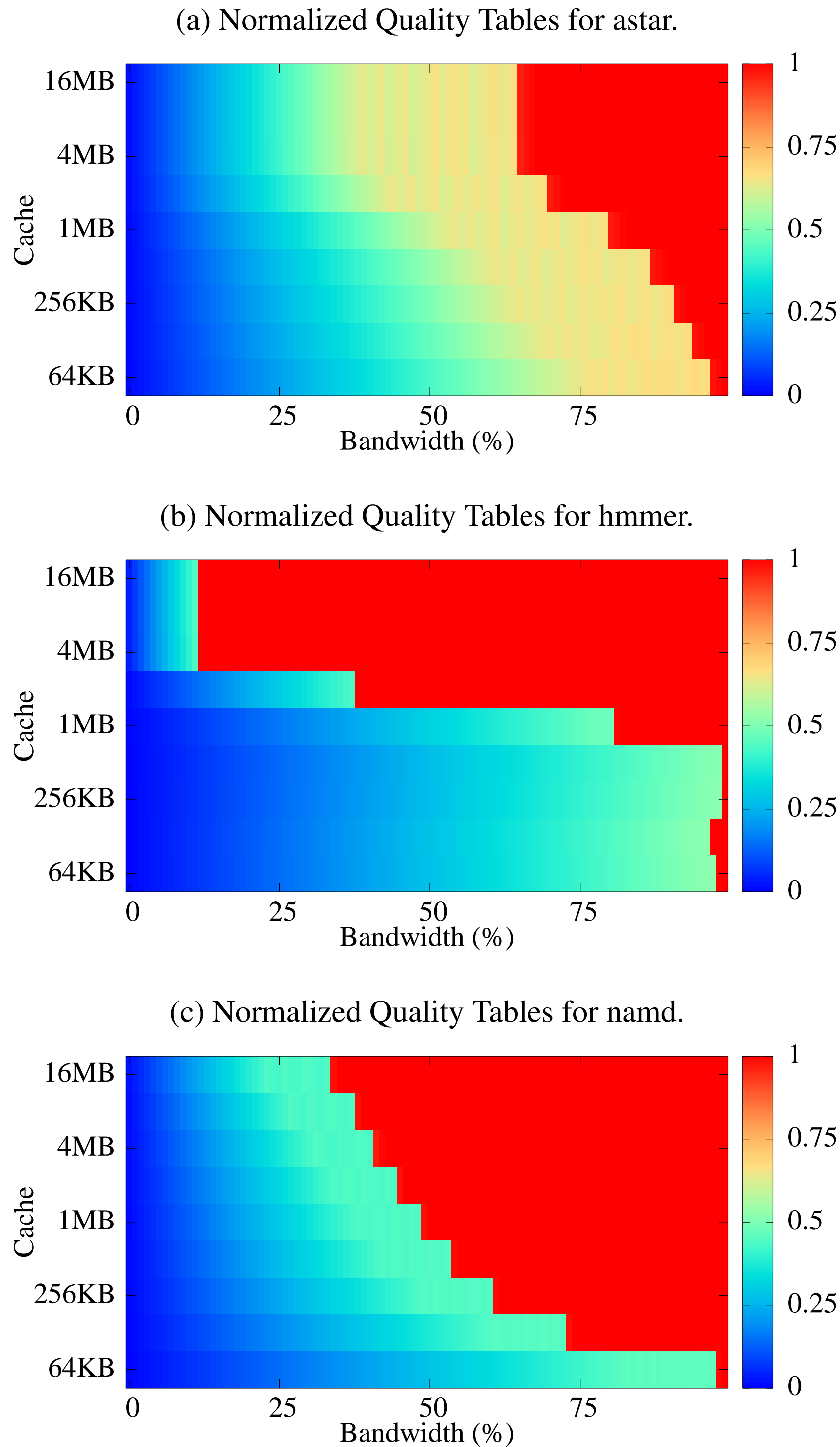


Figure 4.10. ***Normalized Quality Tables for astar, hmmer, and namd.*** Normalized Quality Tables provide the performance estimate for a spectrum of resource allocations, relative to standalone performance. Red depicts full performance, while blue represents zero performance.

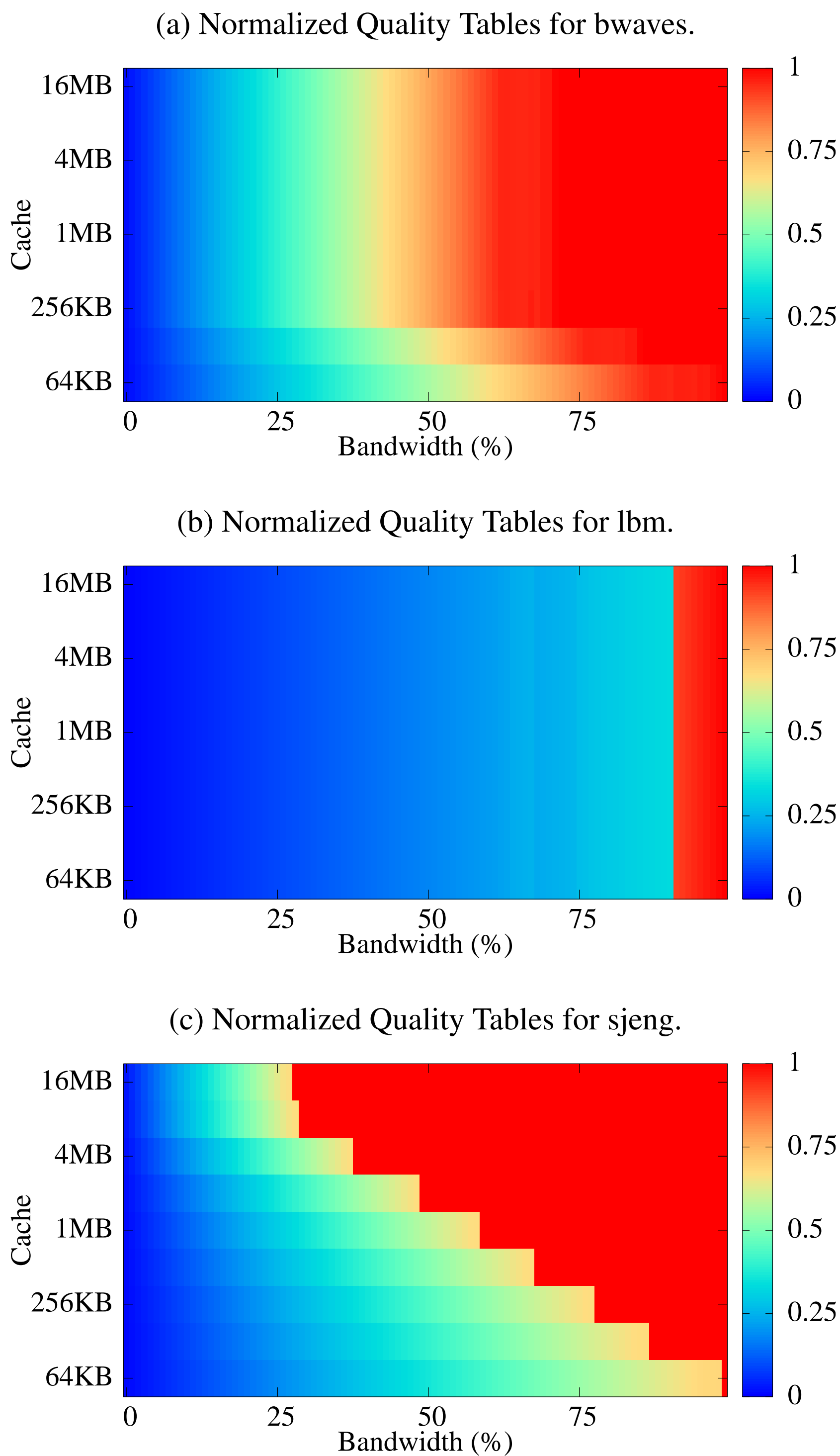


Figure 4.11. ***Normalized Quality Tables for bwaves, lbm, and sjeng.*** Normalized Quality Tables provide the performance estimate for a spectrum of resource allocations, relative to standalone performance. Red depicts full performance, while blue represents zero performance.

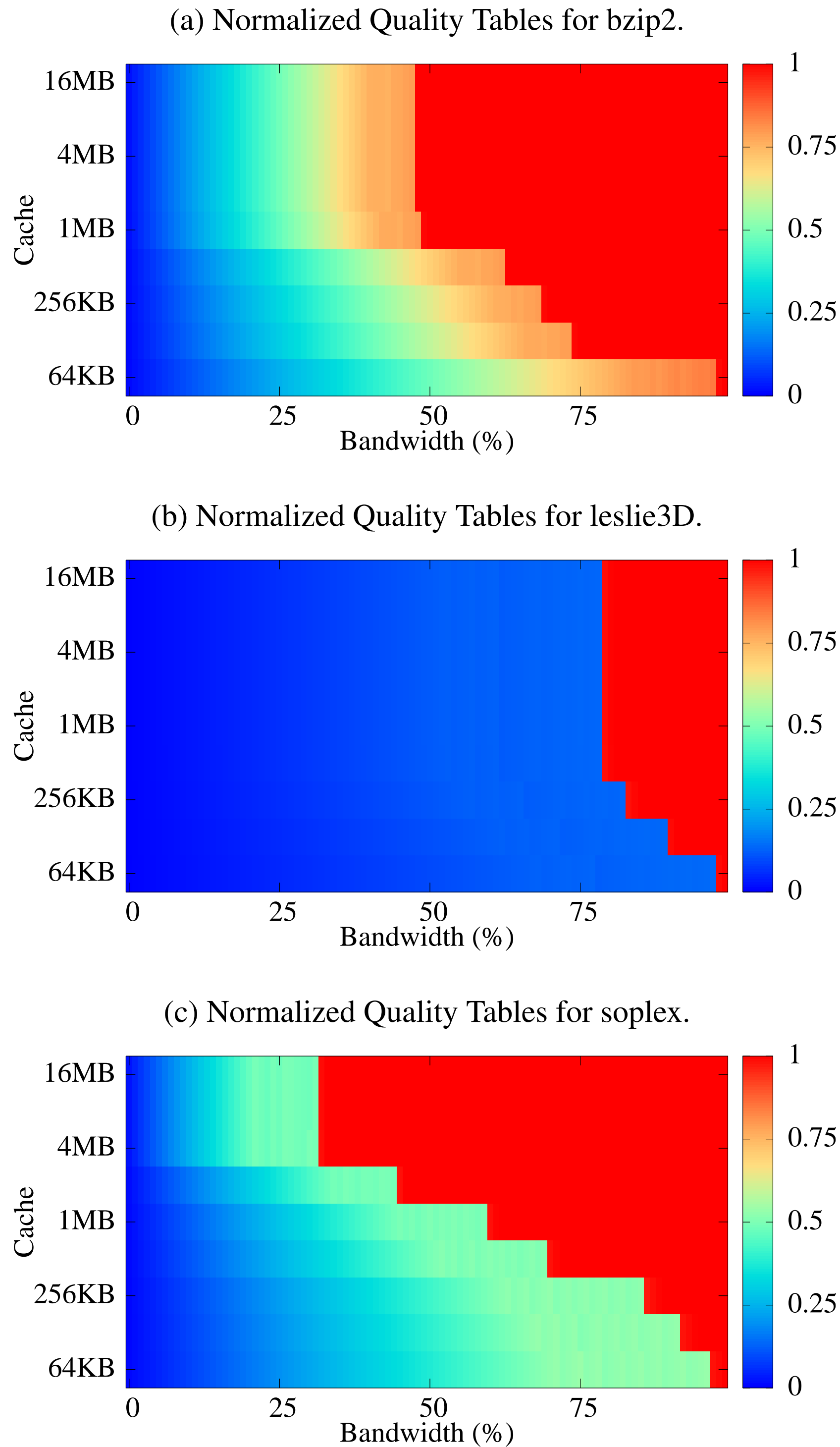


Figure 4.12. ***Normalized Quality Tables for bzip2, leslie3D, and soplex.*** Normalized Quality Tables provide the performance estimate for a spectrum of resource allocations, relative to standalone performance. Red depicts full performance, while blue represents zero performance.

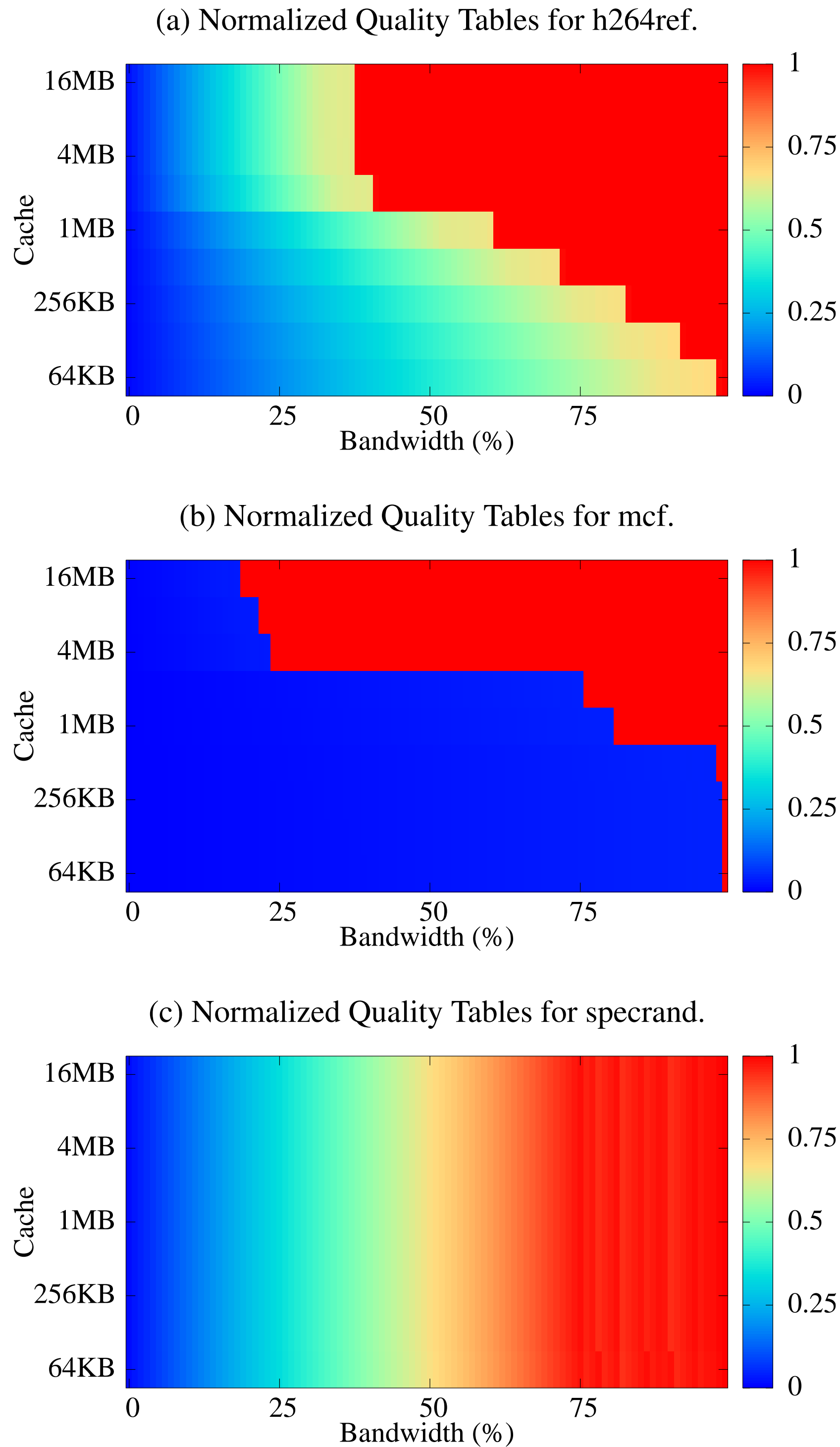


Figure 4.13. ***Normalized Quality Tables for h264ref, mcf, and specrand.*** Normalized Quality Tables provide the performance estimate for a spectrum of resource allocations, relative to standalone performance. Red depicts full performance, while blue represents zero performance.

estimated performance of applications for all possible resource allocations relative to the standalone performance with all the CPU resources. Red depicts full performance, while blue represents zero performance. As evident from these tables, an application's performance doesn't necessarily drop linearly with decreasing resource allocation. In fact, the correspondence between performance and resource allocation varies widely across applications.

Another observation is that the drop in performance with reducing resource allocation depends on the type of resource. For example, lbm has almost no performance drop with decreasing cache allocation, but sees a significant drop in performance with reducing memory bandwidth, as shown in Figure 4.11b. This is typical of stream applications. Another interesting observation is the behavior of cliff applications such as mcf, as visible in Figure 4.13b. These applications see a sudden drop in performance with reduction in cache allocation, when their working-set no longer fits inside the cache.

TimeCube is able to create these Quality Tables with very low errors (about 1%), as shown in Chapter 3, and with very low energy and area overheads. These tables are created in parallel with application execution and impart no performance penalty on applications. Thus, TimeCube can reliably use these Quality Tables to determine the exact amount of resources required for attaining different application performance levels, and to provide guarantees about execution qualities for live applications.

4.4.2 DR-SNUCA Evaluation

DR-SNUCA is Energy-Scalable. Based on my experiments with 32 cores using the DNUCA associative caches, we observe that a significant portion of energy is consumed in L2 (20.01% on average). In contrast, with DR-SNUCA, average L2 energy consumption is only 2.39%. Figure 4.14 shows that the greater energy-scalability of DR-SNUCA results in an average overall energy reduction of 16.27% compared to

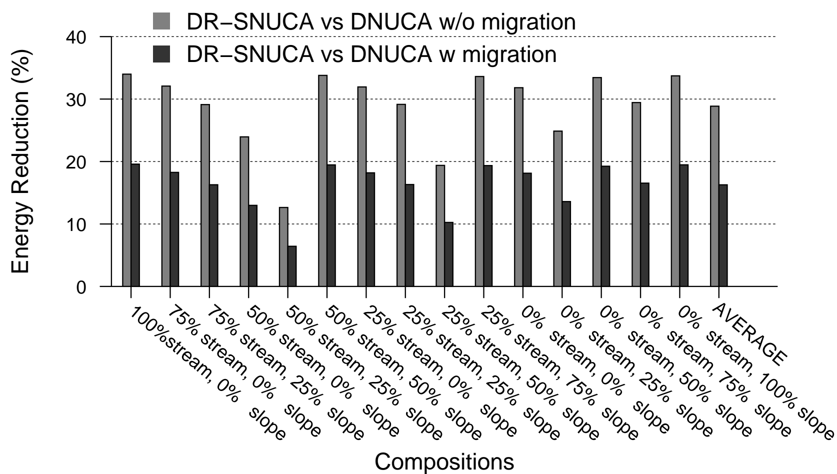


Figure 4.14. ***DR-SNUCA reduces overall execution energy*** DR-SNUCA reduces overall execution energy by 16.27% on an average when compared to DNUCA even with aggressive migration.

DNUCA.

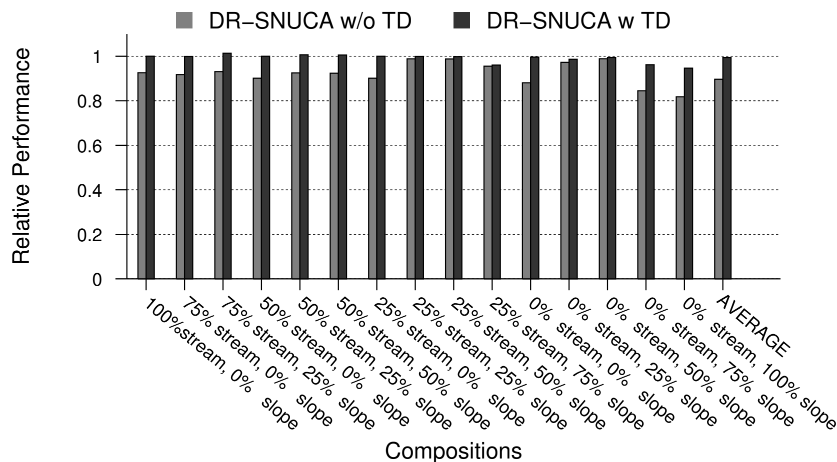


Figure 4.15. ***DR-SNUCA performance is comparable to the baseline DNUCA.*** Without tag-duplication (TD), DR-SNUCA performs 10.3% worse in comparison to the baseline due to reconfiguration stalls. However, after adding tag-duplication, DR-SNUCA is able to perform within 0.5% of the baseline.

The execution times for DNUCA and DR-SNUCA are comparable, as shown in Figure 4.15, due to similar cache hit rates. When running a mix of 32 applications on our prototype architecture, the results show that without tag-duplication TimeCube can lose 10.3% of performance. However, with tag-duplication, reconfiguration for DR-SNUCA can be done in parallel with execution with no timing overhead, and DR-SNUCA is able to perform within 0.5% of the baseline DNUCA, which in turn performs 14.7% better than SNUCA [KBK02], at the expense of just 6.42% of the overall chip area, as seen in Figure 4.16. This small penalty is due to indirect cache addressing in DR-SNUCA.

Flattened Partial LRU Vectors are Efficient. Table 4.1 shows an analytical comparison of the shadow tag schemes for associative caches and FPLV, where W = ways, C = cache configs, T_a = tag area, P_m = tag match energy, and P_l = tag LRU update energy. We can see that area consumed for the two schemes is similar; however, energy consumption is much less for FPLV for tag matching as well as LRU updates. Moreover, FPLV consumed just 1.23% area on our prototype TimeCube when using set-sampling [MQ06], as seen in Figure 4.16. This makes FPLV efficient enough to be employed for shadow-caching in manycore architectures, such as TimeCube, using DR-SNUCA.

Table 4.1. FPLV has the same area as associative shadow tags but much lower energy consumption; therefore, they provide efficient shadow-caching for dissociative caches.

Metric	**Associative**	**FPLV**
Area (per set)	$2^C W T_a$	$2^C W T_a$
Match Energy	$2^C W P_m$	$W P_m$
Update Energy	$2^C W P_l$	$W C P_l$

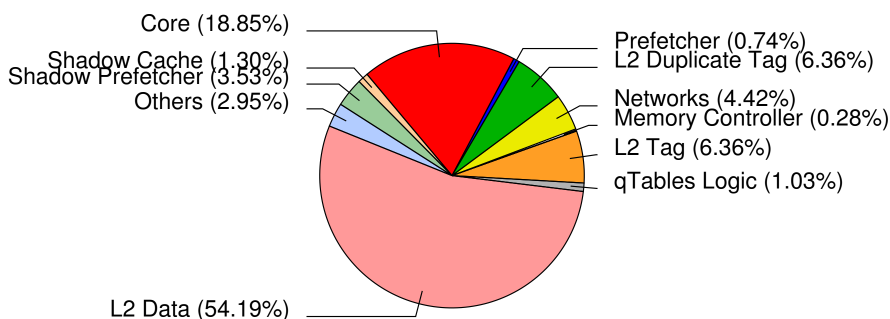


Figure 4.16. ***Area distribution in TimeCube for calculating Quality Tables and providing Dynamic Execution Isolation.*** The area consumed by Shadow Tags, Shadow Prefetchers, and Quality Tables is small (1.30%, 3.53%, and 1.03% respectively). For a 32 core TimeCube, the mechanisms added for Dynamic Execution Isolation consume less than 7% of total chip area, the largest portion being the duplicate tags (6.36%).

4.4.3 Area and Energy Distribution in TimeCube

I now analyze the area and energy distribution for TimeCube when providing Dynamic Execution Isolation and calculating Quality Tables. The microarchitectural mechanisms required to generate Quality Tables consume less than 7% chip area. Dynamic Execution Isolation mechanisms consume less than 7% chip area as well. Quality Tables consume only about 1.03% area, and L2 tag duplication for DR-SNUCA consumes 6.36% area, as shown in Figure 4.16. For an example 32 application mix, the energy consumed by Quality Tables creation is low, 0.01%, because TimeCube is able to reuse Quality Time calculations across the Quality Tables cells. Shadow structures consume about 0.26% energy, the largest portion being the Shadow Tags. Overall, the mechanisms for generating Quality Tables and supporting Dynamic Execution Isolation in TimeCube are energy and area efficient.

4.5 Related Work

Dynamic Execution Isolation. Performance isolation has been proposed as a means to reduce resource interference. Verghese et al. [VGR98] proposed mechanisms for performance isolation for resources such as I/O bandwidth and storage, while Banga et al. [BDM99] suggested resource containers to isolate and account for system-level resource usage. However, since typical manycore architectures rely on shared processor resources, this performance isolation (and not just resource isolation [NLS07]) should be extended to the micro-architectural levels to account for application slowdowns due to sharing of processor resources, since even state-of-the-art resource management schemes, such as the ones proposed by Gohner et al. [GWG$^+$] and Elmroth et al. [EMHF09], do not account for application slowdowns due to sharing of processor resources.

Resource Partitioning. TimeCube dynamically partition the critical shared architectural resources to provide resource isolation. Dynamic resource partitioning has been proposed previously, such as fair-caching [SK04]. Nesbit et al. [NLS07] improved upon previous techniques by partitioning cache access bandwidth as well. Rafique et al. [RLT07] proposed bandwidth partitioning to provide fair bandwidth distribution between applications by adaptively changing the quota of an application based on the observed DRAM latency. Liu et al. [LJS10] proposed partitioning bandwidth between applications with the aim of increasing weighted system speedup. TimeCube reuses fair-queueing arbiters [NLS07], but allocates bandwidth using Quality Time based resource management.

Dynamically Repartitionable Static NUCA. TimeCube spatially distributes the shared cache (NUCA), as proposed by Kim et al. [HKS$^+$05], to reduce access energy and time. However, the existing NUCA techniques proposed, such as s-NUCA and

d-NUCA, do not satisfy our requirements, since s-NUCA is not dynamically re-sizable and d-NUCA is not energy-efficient for large cache sizes, even when we use optimization techniques, such as way-prediction [PAV+01], partial-tag matching [KJLH89] and data migration [HKS+05]. Thus, I extend s-NUCA with cache-indirection [AP93] to create DR-SNUCA, which is both dynamically reconfigurable as well as energy-efficient for large cache sizes. The cache partition sizes can be determined in hardware or left to the OS, as proposed by Rafique et al. [RLT06]. Cache partitioning techniques at different spatial granularities have been proposed, such as the page-level scheme by Cho et al. [CJ06] and the memory-address-map based scheme proposed by Lin et al. [LLD+08]; however, DR-SNUCA partitions the cache at a finer spatial granularity without incurring heavy reconfiguration and access costs.

Existing techniques partition the caches by dividing associative ways among the applications which is not power-efficient. TimeCube's dissociative cache partitioning handles the cache traffic and reconfiguration efficiently for many applications. Kim et al. [HKS+05] discuss using a hashing scheme similar to ours to determine the cache block that can hold an address, and bound the cache associativity; however, their hashing scheme is static and cannot handle dynamically changing cache allocations. They also talk about the performance penalties of highly associative caches and propose using partial tags. However, partial tag matching consumes energy proportional to the cache size and is not as scalable as DR-SNUCA.

Memory Scheduling. TimeCube's distributed memory controllers can utilize any memory scheduling scheme which fairly distributes the available bandwidth between applications. Several fair memory scheduling techniques have been proposed previously, such as stall-time fairness [MM07], batch-scheduling [MM08b], distributed-order scheduling [MM08a], self-optimizing controllers [IMMC08], or prioritizing threads

receiving least service [KHMHB10]. However, TimeCube uses the fair queue arbiter [NLS07], since it does fair-scheduling while staying within allocated bandwidth limits for each application.

4.6 Conclusion

IaaS computing systems as well as embedded systems need to tackle the challenge of interference due to space-multiplexing, which can cause unevenness in slowdowns to the order of 12$\times$ in a 32-core processor and prevents any kind of execution quality guarantees, in order to fully capitalize on the benefits of manycore computing. Dynamic Execution Isolation can reduce this resource interference. TimeCube generates Quality Tables in hardware to determine the exact resource requirement for attaining different performance levels, and then uses dynamic execution isolation, achieved through dynamic resource partitioning, to enforce resource allocations to guarantee execution guarantees for live applications. TimeCube uses DR-SNUCA, an energy-scalable dynamically partitioned cache, to reduce the energy consumption for a 32-core system by 16.27% compared to DNUCA caches, while performing within 0.5%.

The area and energy overheads of these dynamic partitioning mechanisms are low; therefore, dynamic execution isolation can be used to provide interference-free resource sharing for manycore processors, while Quality Tables can be used for enforcing execution guarantees.

Acknowledgment

Parts of these chapters are reprinted from the following papers:

- Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford. "DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache", International Conference on

Computer Design, ICCD 2013.

- Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford. “TimeCube: A Manycore Embedded Processor with Interference-agnostic Progress Tracking”, International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2013.

Permission to use these contents has been obtained through signed letters from the co-authors. Dissertation author was the primary investigator and author on these papers.

Chapter 5

SPOT: Improving Execution Quality in Hardware

Manycore processors, such as TimeCube, pack a large amount of resources on a single die to provide high compute-density. Even if they have large quantities of shared resources, such as cache and memory bandwidth, the per core cache size and per core bandwidth is low; moreover, these resources are usually insufficient to satisfy memory requirements of all the applications. As a result, highly consolidated manycore systems, such as TimeCube, are especially sensitive to the memory resource allocation making resource management extremely crucial.

Existing processors manage these resources locally in the hardware, enabling a finer-grained control while hiding the hardware complexity from the software stack. However, the local control leads to interference. As mentioned in Chapter 4, we can use dynamic execution isolation in manycore processors to reduce resource interference and, in conjunction with Quality Tables, control execution qualities. TimeCube dynamically partitions the shared resources to provide this isolation. However, for resource management, TimeCube still needs to dynamically distribute these partitioned resources between applications based on their requirements as well as system objectives, and it is a challenge to find a *good* resource allocation for manycore processors with high resource

utilization, even in the presence of dynamic execution isolation.

In this chapter, I introduce a novel manycore resource management scheme, which can lead to increased resource utilization, resulting in higher system performance. The scheme is based on a novel hardware-generated data structure, called *Simultaneous Performance Optimization Table*, or *SPOT*.

> **Simultaneous Performance Optimization Table**, or **SPOT**, is a three-dimensional structure with the three axes being applications, cache, and bandwidth; each cell (i, j, k) of SPOT provides the maximum possible mean Quality Time for i applications using j cache-arrays and k bandwidth bins, and it also provides the corresponding resource allocation.

I use TimeCube as a demonstration vehicle for this scheme. TimeCube employs a novel dynamic programming algorithm, based on the insight that we can reuse the result of the sub problem - best allocation of a subset of cache and bandwidth between a subset of applications, to periodically generate SPOT using the application Quality Tables. TimeCube determines the resource allocation to satisfy multiple system objectives, i.e., high throughput and fairness, and addresses these conflicting objectives by maximizing the *Mean Quality Time*. TimeCube then optimally allocates portions of the partitioned last-level cache and memory bandwidth between active applications using SPOT's last cell. This allocation is done periodically to accommodate changing application phases.

Dynamic Execution Isolation leads to another problem on manycore processors: How to tune mechanisms that use the shared resources, such as the DRAM prefetchers using memory bandwidth, for all possible allocations? TimeCube dynamically tunes the application DRAM prefetchers based on their bandwidth utilization in order to maximize their bandwidth utilization at all possible allocations.

TimeCube allows programs to execute all the time without ever stalling them to create Quality Tables, generate SPOT, or repartition resources. In every interval, it collects the execution and shadow statistics using special hardware. At the end of the interval, it uses those statistics to create Quality Tables and generate SPOT. This is done in parallel with the program's execution. Once the optimal allocation is calculated, the cache and bandwidth partitions are reconfigured. This also is done in parallel with program execution. The remaining interval proceeds with the new partitioning. Meanwhile the statistics are being accumulated for this interval to determine partitioning for the next interval.

I ran experiments to evaluate TimeCube's resource allocation scheme using the experimental setup explained in Section 3.3.1. TimeCube's SPOT-based resource allocation increases throughput by 36% on average. Moreover, this algorithm provides increasingly better utilization for larger systems, as well as systems running an increasingly large number of applications. The low overheads of the mechanism as well as good results make a compelling case for adding the requisite micro-architectural structures to do SPOT-based optimal resource allocation and dynamic microarchitectural tuning in future manycore processors.

In summary, this chapter describes the following novel contributions:

1. **Progress-based dynamic resource management** I propose a novel simultaneous resource management scheme based on maximizing the Mean Quality Time for manycore processors that provide dynamic execution isolation.

2. **Simultaneous Performance Optimization Table** I describe a novel dynamic programming algorithm that uses Quality Tables to generate a data structure called *Simultaneous Performance Optimization Table*, or SPOT, and determine an optimal resource allocation to maximize the Mean Quality Time.

3. **Dynamic prefetcher throttling** I propose a novel prefetcher throttling mechanism that tunes the prefetching intensity based on the dynamically allocated bandwidth.

4. **Scalable resource allocation for higher throughput in manycore processors** I provide a detailed evaluation of the novel resource management mechanisms using TimeCube. The mechanisms increase throughput by 36% on average when compared to existing allocation schemes, and improve system scalability with increasing number of cores as well as system load, or applications.

The remainder of the chapter proceeds as follows. Section 5.1 explains my insights into resource management for concurrent systems. Section 5.2 describes the implementation details of SPOT. Section 5.3 explains the design of the dynamic prefetcher throttling mechanism. Section 5.4 presents the execution model for TimeCube when using Quality Tables and SPOT for resource allocation. Section 5.5 presents a detailed manycore evaluation of the mechanisms proposed in this chapter. Section 5.6 discusses the related work and Section 5.7 concludes.

5.1 Maximizing Mean Quality Time: A Unified Resource Management Objective

At the microarchitectural level, even though manycore processors have large quantities of shared resources such as cache and memory bandwidth, the per core cache size and per core bandwidth is low. Thus, system performance is especially sensitive to memory resource allocation. TimeCube has to allocate these resources between the many applications running concurrently on the processor. A free-for-all resource management, i.e. letting local independent mechanisms manage sharing of individual resources, can lead to interference. This makes it difficult to control application performances as well as improve global resource utilization.

I described Dynamic Execution Isolation in Chapter 4, which keeps resource interference under check and, in combination with Quality Tables, lets TimeCube control applications' execution qualities. TimeCube can use dynamic execution isolation to allocate equal resources to all applications, but this can lead to poor resource utilization and system performance, since different applications have different resource utility, as shown in Section 4.4.1. Thus, in order to maximize their utilization, resources should be distributed between application's based on their utility, which also leads to performance improvement. Quality Tables provide application performance estimates for all possible resource allocations with very high accuracy. TimeCube can use Quality Tables to determine the utility of different resources to applications, which in turn can be used to determine a *good* resource allocation between application that can be enforced through dynamic execution isolation.

Now, in a system with multiple concurrent applications contending for shared resources, such as TimeCube, an application's progress depends on the amount of resources allocated to it. In a fair system, the progress should be similar between the applications, which means that even if there is a shortage of resources, they are distributed such that the applications that provide lower performance are also given a fair share. However, to attain a high overall system performance, more resources should be given to the applications which provide higher performance. Thus, these two system objectives require conflicting resource distribution strategies.

TimeCube attempts to address the two conflicting goals simultaneously by creating a single metric to summarize the performance of multiple applications that will reflect both throughput as well as fairness. It takes the mean of application Quality Times accumulated over their entire executions up to that point. For every application, TimeCube accumulates this running Quality Time by adding the incremental Quality Time in every interval, provided by the last cell of its Quality Tables. Maximizing this metric

allows us to maximize the overall system performance, while also maintaining some balance between slowdowns across applications. For the mean, we can use arithmetic, geometric or harmonic means. These means provide varying degrees of fairness between applications with the lowest fairness provided by arithmetic mean and the highest by harmonic mean. For this work, we maximize the geometric mean of application speedups. For every interval j, TimeCube finds a cache ($\hat{c}$) and bandwidth ($\hat{b}$) distribution between applications that maximizes the Mean Quality Time, to find a balance between throughput and fairness.

$$Mean\ Quality\ Time_{j,\hat{c},\hat{b}} = \prod_i (Quality\ Time_i + qTables_j[i,c,b]) \tag{5.1}$$

After several time intervals, Mean Quality Time can be approximated to -

$$Mean\ Quality\ Time_{j,\hat{c},\hat{b}} = \sum_i \frac{qTables_j[i,c,b]}{Quality\ Time_i} \tag{5.2}$$

This additive formulation enables TimeCube to do time-multiplexing, i.e. handle more applications than the number of cores in the processor. Interestingly enough, this formulation can be generalized to -

$$Mean\ Quality\ Time_{j,\hat{c},\hat{b}} = \sum_i \frac{qTables_j[i,c,b]}{Quality\ Time_i^{\alpha}} \tag{5.3}$$

At $\alpha = 1$, this formula approximates the geometric mean, but for $\alpha = 0$, it is equal to the arithmetic mean of application Quality Times. Moreover, for $\alpha = 2$, this approximates the harmonic mean of the application Quality Times. Thus, an increasing α value leads to an increasing fairness of the metric, at the cost of performance. Thus, a system can tune α in accordance with the policies to choose the right balance between fairness and throughput. For this work, I choose $\alpha = 1$.

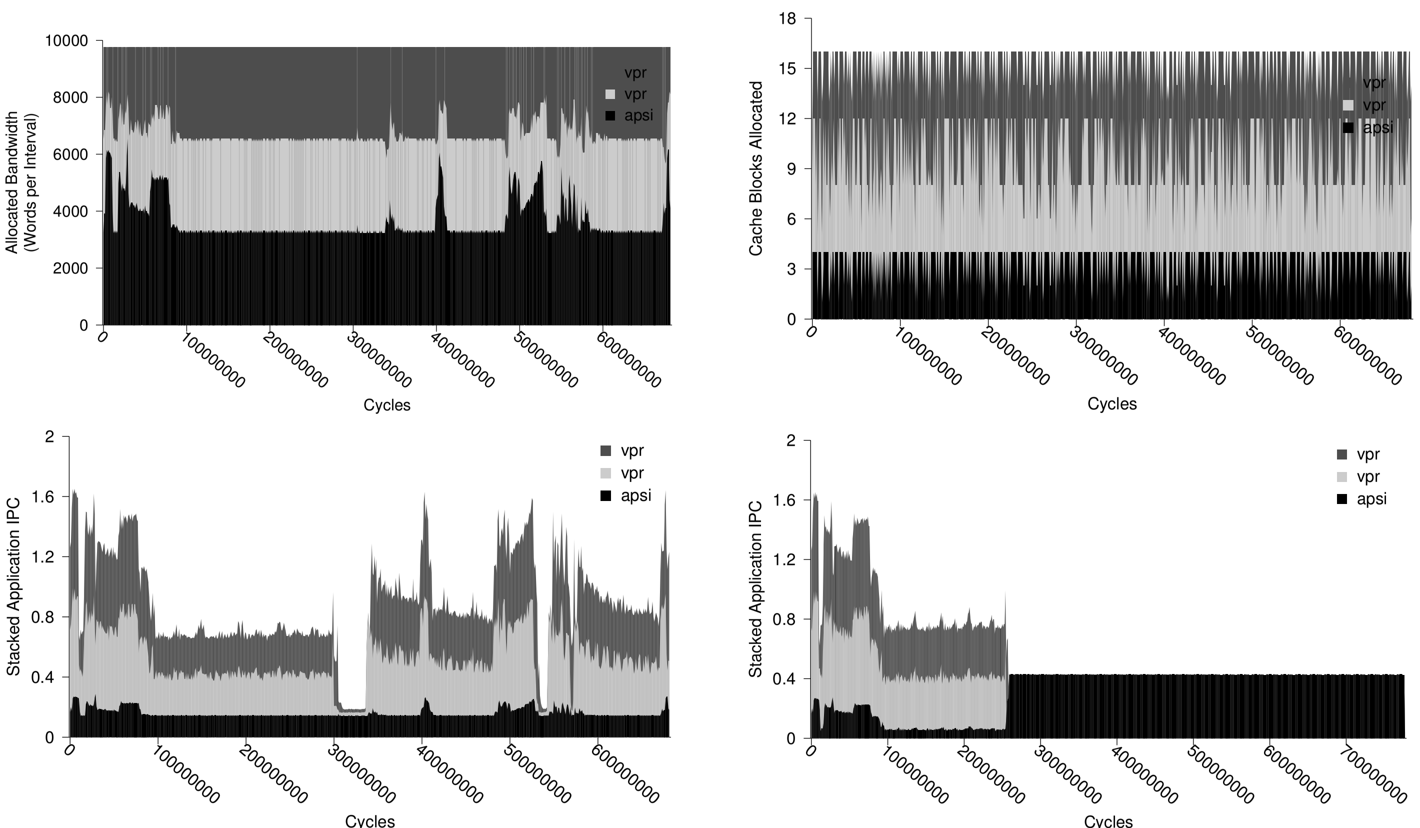


Figure 5.1. ***Simultaneous resource allocation leads to better resource utilization for better overall performance.*** I ran two instances of vpr and an apsi application on a 4 core chip. The required bandwidth of apsi exceeds the total available bandwidth. However, our resource allocation algorithm prevents apsi from hogging all the bandwidth (a), and some cache blocks (b). Moreover, we can see that using geometric mean for finding the best allocation ensures fairness among all applications (c), which cannot be guaranteed if we use arithmetic mean (c).

To examine the effectiveness of Mean Quality Time based resource allocation, I ran an experiment with three applications (vpr, vpr, apsi) running on a 4 core TimeCube instance. Memory bandwidth requirement of apsi exceeds the total available bandwidth in the system. However, apsi also has the lowest utility for bandwidth, i.e. the number of instructions blocked on a single miss is lesser for apsi compared to vpr. Thus, to increase the bandwidth utilization, TimeCube should allocate more bandwidth to vpr than the ratio of the miss requests generated by the applications. As we can see, Mean Quality Time based allocation divides the bandwidth amongst all applications and doesn't let apsi hog all the bandwidth, as shown in Figure 5.1(a). It also provides a small portion of cache to apsi, as shown in Figure 5.1(b), since vpr has a higher utility for last-level cache compared to apsi. In order to find the best allocation, I use the geometric mean for the applications' performances, shown in Figure 5.1(b), as this provides better fairness in our allocation algorithm. On the other hand, arithmetic mean does not even guarantee forward progress for all applications (non-zero IPC), as shown in Figure 5.1(c).

5.2 SPOT: Finding the Resource Allocation to Maximize the Mean Quality Time

TimeCube needs an efficient algorithm to maximize the Mean Quality Time for the system every time slice. TimeCube can use the application Quality Tables to determine a shared resource allocation between the applications based on their characteristics and find the desired allocation. TimeCube can find this allocation even while satisfying SLAs or real-time constraints, such as guarantees of forward progress, minimum execution rate, or maximum slowdown, as discussed in Chapter 4.

Using the Quality Tables, TimeCube can calculate the metric for all possible resource distributions and choose the best allocation; however, this brute force method is inefficient. TimeCube employs a dynamic programming based algorithm to calculate the

cache and bandwidth allocation that maximizes Mean Quality Time. This algorithm is based on the following insight: We can reuse the result of a sub problem, i.e. a subset of cache and bandwidth partitioned between a subset of applications to maximize their Mean Quality Time.

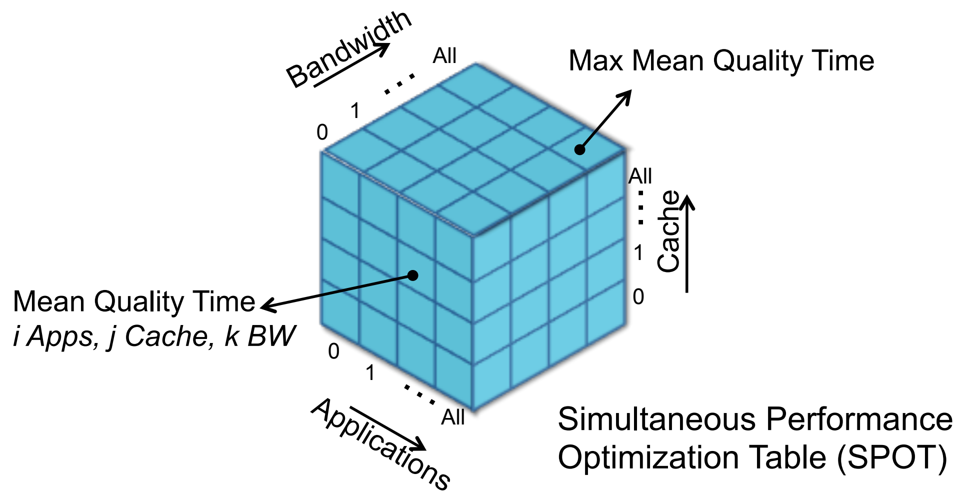


Figure 5.2. ***TimeCube uses Simultaneous Performance Optimization Table, or SPOT, to find the optimal resource allocation.*** TimeCube uses Simultaneous Performance Optimization Table, or SPOT, a three-dimensional hardware generated data structure to find the optimal resource allocation that maximizes the Mean Quality Time.

TimeCube creates a three-dimensional cube, *Simultaneous Performance Optimization Table*, or *SPOT*, and use the algorithm, shown in Equation 5.4, to derive the optimal allocation in the last cell of SPOT, i.e. $SPOTc[N,\$_t,B_t]$. This calculation is done in parallel with execution.

$$
\begin{aligned}
SPOTv[i,\$,B] &= Max\ possible\ mean\ progress - time \\
&\quad for\ i\ apps,\ \$\ cache,\ and\ B\ bandwidth \\
SPOTc[i,\$,B] &= Cache\ and\ bandwidth\ distribution \\
&\quad for\ SPOTv[i,\$,B]
\end{aligned}
$$

$$\begin{aligned}
SPOTv[i,\$,B] &= max_{\$',B'}\{SPOTv[i-1,\$-\$',B-B'] \\
&\quad +qTables[i,\$',B']\} \\
SPOTc[i,\$,B] &= SPOTc[i-1,\$-\$',B-B'] \\
&\quad .append([\$',B']_{max}) \\
BestPartition &= SPOTc[N,\$_t,B_t] \qquad (5.4)
\end{aligned}$$

TimeCube creates the Quality Tables for all applications, and then runs the dynamic programming algorithm to find the resource allocations that yields the highest possible Mean Quality Time. It uses this allocation for the next interval, and thus reconfigures the shared resource partitions accordingly. Details of execution flow are present in Section 5.4. In my experiments, I give equal shares of cache and bandwidth to all applications at the start of a run. In a real-world system, applications can be started off with a predetermined fixed starting cache and bandwidth allocation.

For a 32-core TimeCube instance, the hardware allocation mechanism occupies 2.19% of the chip area and 0.23% of the total execution energy. This formulation can handle I/O threads as well, since if a thread is blocked on I/O, the application's Quality Tables will show a low quality time for resources, which can then be allocated to other threads. This also elegantly handles more applications than cores because of its additive (rather than multiplicative) formulation. The Quality Tables for the suspended applications are stored within their context.

5.3 Prefetcher Throttling

When applications are dynamically allocated the memory bandwidth, the prefetchers need to be dynamically tuned to maximally utilize the available bandwidth. For example, for a certain cache and bandwidth partition, an application might face a short-

age of bandwidth and the bandwidth loss due to incorrect prefetches might overshadow the latency savings because of correct prefetches. I propose a new mechanism which dynamically adjusts the prefetching aggressiveness to maximize the utilization of the dynamically changing available memory bandwidth and can work in conjunction with any existing prefetcher accuracy improvement mechanism. Prefetch throttler reduces

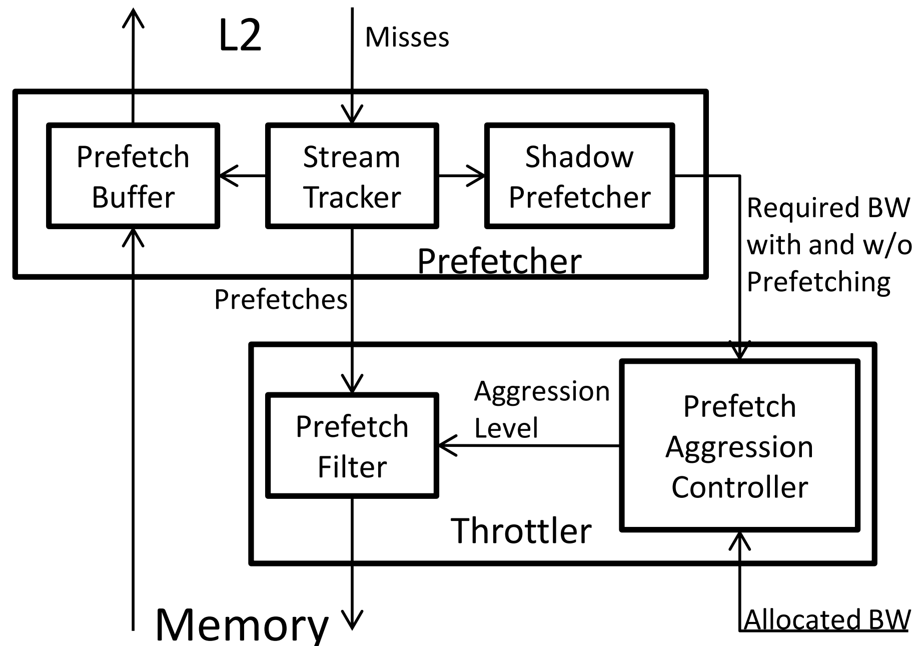


Figure 5.3. ***Prefetcher throttling in TimeCube*** The prefetcher throttling mechanism changes the prefetcher aggression based on the bandwidth requirements and availability. Prefetch aggression controller (PAC) finds the aggression level to best utilize the available bandwidth, and the prefetch filter drops the corresponding ratio of prefetches.

the number of prefetches without affecting the internals of the prefetcher by dropping a fixed ratio of prefetches issued by the prefetcher. This fixed ratio is called the *prefetch aggression level* and it is determined by the *prefetch aggression controller*, or PAC, based on the utility of prefetches to an application, as shown in Figure 5.3.

PAC takes in the allocated bandwidth for the application and the required bandwidth for the application with and without prefetching, as calculated by the *shadow*

prefetchers. It uses them to determine the required bandwidth at different aggression levels, for which it assumes that on dropping a fixed ratio of prefetches, a proportional number of good and bad prefetches are dropped. Thus prefetcher accuracy remains almost same leading to a proportional drop in prefetch hits. This provides the rough estimate for the number of memory requests at an aggression level. To determine bandwidth for that aggression level, PAC determines the overall latency by assuming that the average prefetch hit latency savings remains the same at different aggression levels, and using it to calculate the total latency savings by using the estimated number of prefetch hits. Once PAC determines the required bandwidth for different aggression levels, it finds the prefetch aggression level for which the required bandwidth is just higher than the available bandwidth, and the prefetcher accordingly drops a fixed ratio of prefetches during the next execution interval.

5.4 TimeCube Execution Model with SPOT

TimeCube allocates cache and bandwidth simultaneously between applications based on their behaviors. However, an application's behavior also changes over time. An application's working set size can change, average bandwidth utility can change etc. Thus, one time allocation of resources is not sufficient. TimeCube checks the application behavior from time to time and readjust the partitioning. In this work, I use a periodic interval based reallocation, as shown in Figure 5.4, triggered using a periodic interrupt. It might lead to some unnecessary checks, but this keeps it invisible to software, and therefore allows TimeCube to use legacy code.

TimeCube allows programs to execute all the time, without ever stalling them to collect statistics, calculate Quality Tables, generate SPOT, or reconfigure resources. In every interval, TimeCube collects the statistics using special hardware. At the end of the interval, it uses those statistics to create Quality Tables using shadow performance

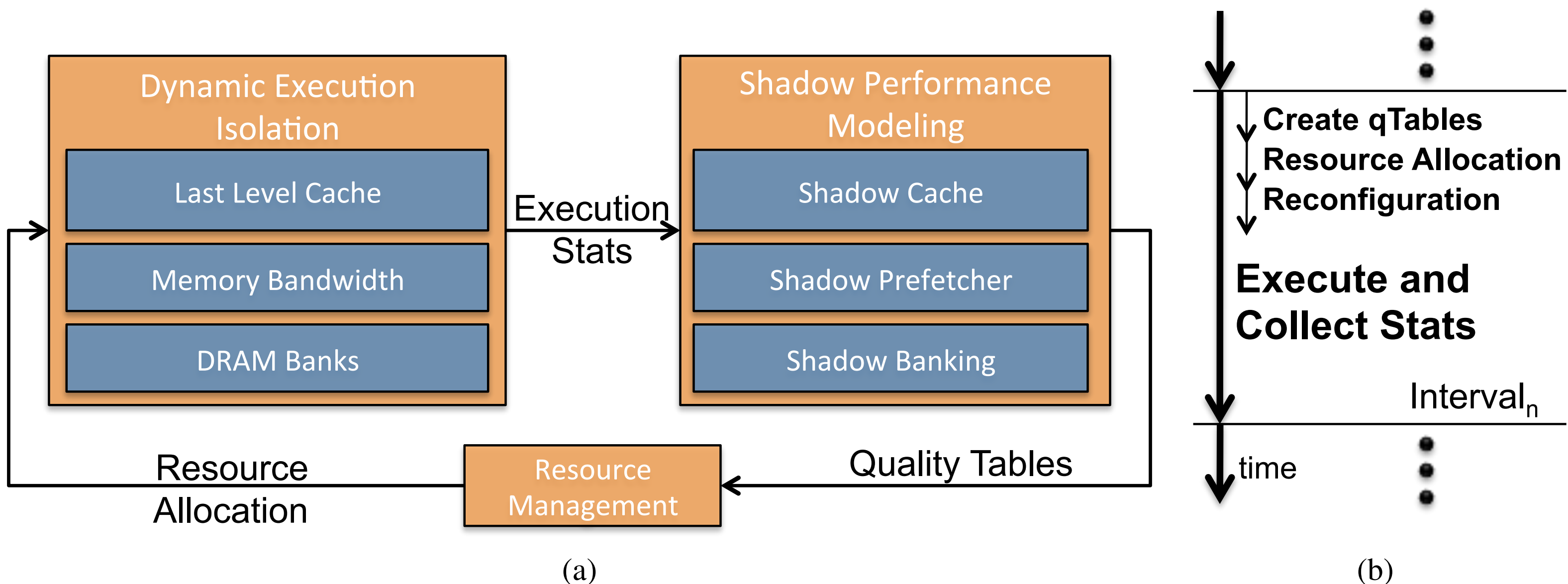


Figure 5.4. ***TimeCube execution model.*** TimeCube partitions the critical shared resources to provide dynamic execution isolation, uses Shadow Performance Modeling to cerate Quality Tables for live applications, and then uses these tables inside SPOT to find the optimal resource allocation maximizing the Mean Quality Time (a). This optimal allocation is then used to repartition the shared resources for the next interval. Every interval TimeCube collects statistics, creates Quality Tables, generates SPOT, finds the optimal allocation, and reconfigures shared resources simultaneously for all applications, all in parallel with execution (b).

modeling, and characterize the application behavior. These Quality Tables are then used to generate SPOT, and calculate an optimal allocation of cache and bandwidth between applications by maximizing the Mean Quality Time. This is all done in parallel with the program's execution. Once the new allocation is calculated, the cache and bandwidth partitions are reconfigured. This also is done in parallel with the program execution. The remaining interval proceeds with the new resource allocations. Meanwhile, the statistics are being accumulated for this interval to determine the optimal resource allocation for the next interval.

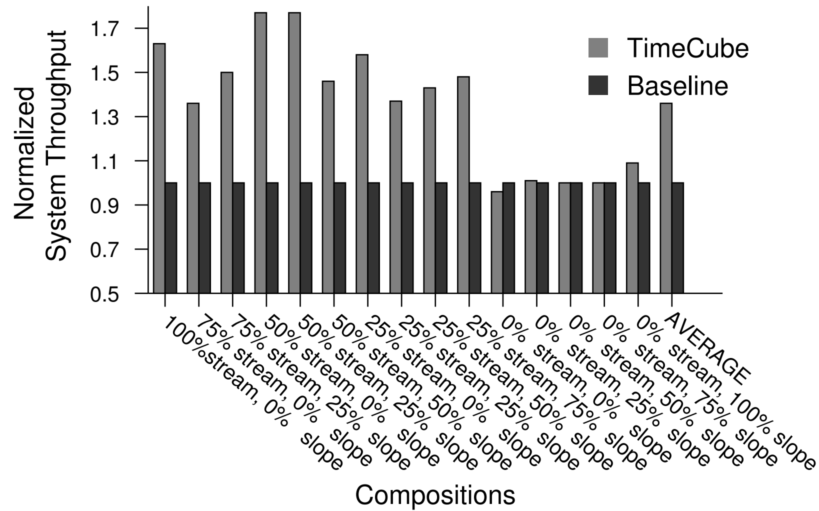
5.5 Results

In this section we describe the results of our experimental evaluation of Time-Cube's simultaneous resource allocation using SPOT, and the dynamic prefetcher throttling, along with several other scalability and sensitivity studies. We use the evaluation methodology detailed in Section 3.3.1 unless explicitly specified.

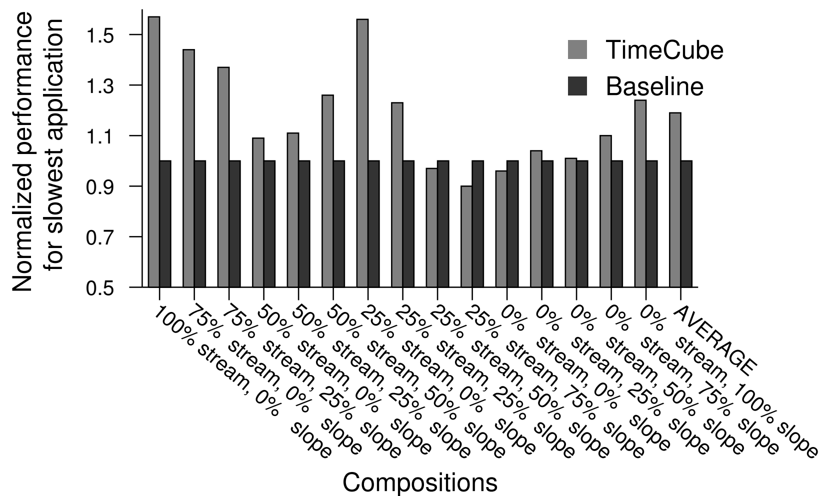
5.5.1 Overall Results

Throughput is a first order concern for concurrent manycore systems. We quantitatively analyze the throughput obtained with quality time based resource allocation and compare our scheme against a *baseline* in which we first partition the caches to minimize the cache misses [MQ06], and then we partition the bandwidth to provide equal slowdown between applications [NALS06]. Our baseline provides a higher performance compared to existing commercial manycore system resource allocations, which provide fair bandwidth sharing between applications, but do not minimize cache miss rates by through demand-based dynamic cache allocation.

We observed that throughput improves by 36% on average for our scheme, compared to the baseline, as shown in Figure 5.5(a). These gains are made possible due to



(a)



(b)

Figure 5.5. ***TimeCube's resource allocation leads to higher throughput.*** TimeCube's Quality Time-based resource allocation leads to higher throughput (a) for the system (36% on average), and higher performance (b) for the applications (19% on average) due to better resource utilization.

simultaneously allocating different resources with a shared objective, leading to increased resource utilization, as opposed to existing architectures that end up allocating different resources (cache and bandwidth) with possibly conflicting objectives (throughput and fairness respectively), due to the lack of system-wide online performance metrics, such as the quality time. This higher resource utilization also leads to an improvement in application performances by 19% on average, as shown in Figure 5.5(b). qTables provide the required information that helps us allocate these resources simultaneously and increase utilization.

Comparison Between Various Allocation Algorithms

We run different combinations of benchmarks with different mechanisms to see the quantitative gains. Based on the classification provided earlier, we choose representatives from the different categories to find the performance for different category combinations. We run both four core (Figure 5.6a) and eight (Figure 5.6b) core experiments. We compare the performance of our system against a baseline which has the same number of cores, cache, bandwidth and prefetchers. Though the cache is equally partitioned amongst the cores, the prefetcher is always ON and the memory controller sends the requests to the off-chip pins in the ratio of the requests received.

For the first mechanism, we partition the cache using our performance estimation but the bandwidth is not partitioned. We see on an average the performance drops by 2.66% and 0.01% for 4 and 8 cores respectively. In cases involving slope applications, we see a decrease in performance as reconfiguration costs exceed miss-minimizing benefits of cache allocation. When we add bandwidth partitioning to this, we see a significant increase in performance with 4 core average going up by 52.78% and 8 core performance going up by 70.97%. We see a further increase in performance to 65.17% and 81.09% for 4 and 8 cores on adding prefetcher throttling to the system. Finally, on adding variable

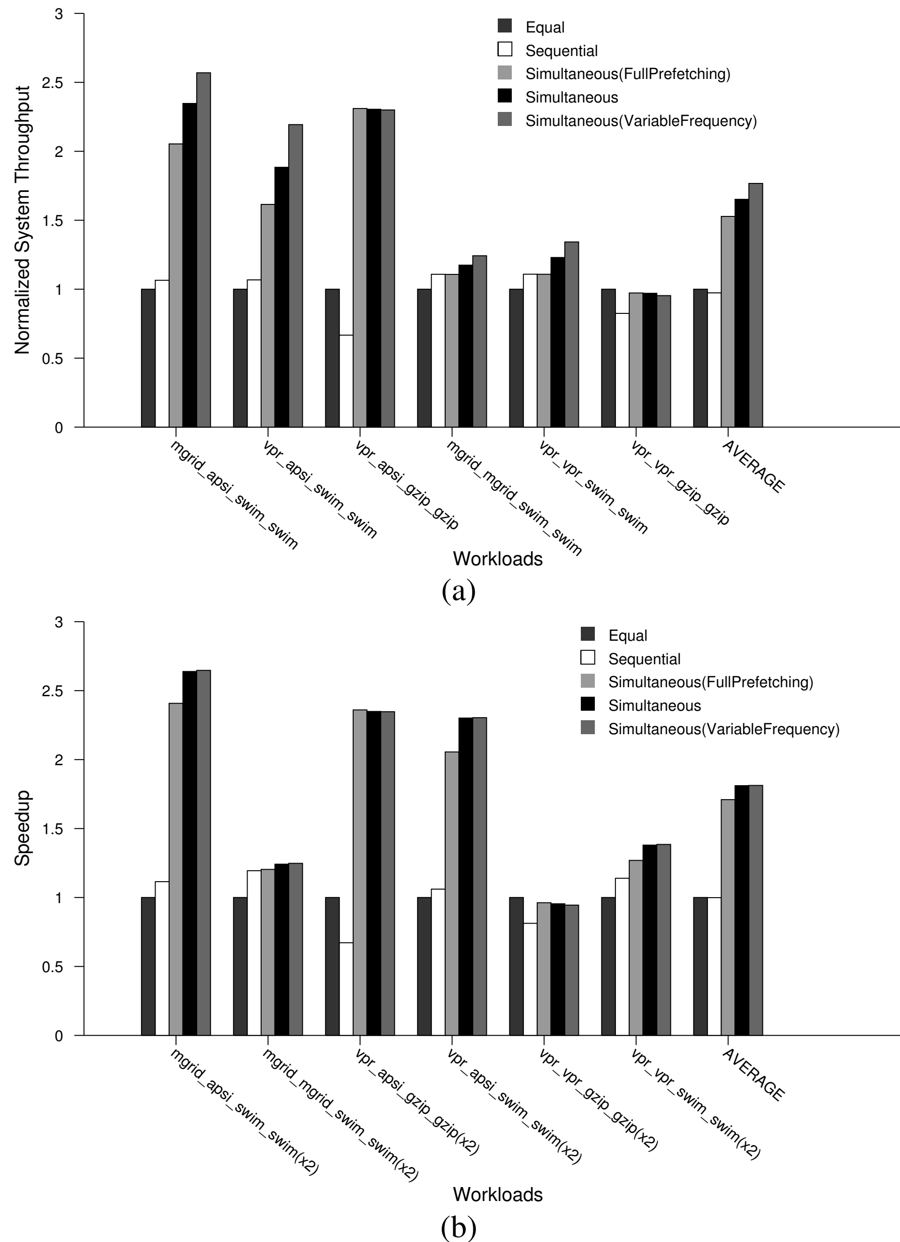


Figure 5.6. ***Performance improves when cache and bandwidth are allocated simultaneously.*** The performance drops by 2.66% and 0.01% on average for 4 and 8 cores respectively with miss-minimizing cache allocation compared to the *baseline*, i.e. equal cache and bandwidth distribution. However, simultaneous cache and bandwidth allocation increases performance by 52.78% and 70.97% for 4 and 8 cores respectively. The performance further improves to 65.17% and 81.09% respectively with prefetcher throttling and to 76.69% and 81.25% with variable cache partitioning.

cache switching frequency mechanism, which we explain later, we see a further increase of 76.69% and 81.25% on 4 and 8 core average performances respectively.

5.5.2 Prefetcher Throttling

We ran an experiment with varying workload mixes, as described before, to test the usefulness of prefetcher throttling over varying amounts of allocated bandwidth (Figure 5.7). We use *nine* aggression levels (0-8) in TimeCube. Our experiments show that at lower bandwidths, it is beneficial to turn off prefetching as bandwidth is precious and should not be wasted on bad prefetches while at higher bandwidths, we could afford to spend some bandwidth on inaccurate prefetches in lieu of the latency savings of prefetch hits. Prefetcher throttling mechanism figures out the *right point* at which it changes the prefetcher aggression level. This leads to a performance better than both with and without prefetching in the regime, where the available bandwidth lies in between the bandwidths required with prefetcher ON and OFF. Hence, prefetcher throttling provides a near optimal performance at all bandwidths by approximately tracking the Pareto curve for different throttling levels. We can get a smoother Pareto optimal curve and better performance by using more throttling levels, since in this work we limited ourselves to 9 levels.

Better Throughput with Prefetcher Throttling We ran experiments to see if the qualitative gains of prefetcher throttling shown above translate into quantitative gains. We run experiments with varying benchmark compositions with and without prefetcher throttling. When not throttled, all prefetches are sent to the memory. As we can see in Figure 5.8(a), while simultaneous resource allocation increases throughput by 38.02% on average compared to sequential allocation. However, with prefetcher throttling this increase jumps to 50.11%. Similarly, while the geometric mean of performance gains increase by 30.92% without prefetcher throttling, it increases to 45.85% with prefetcher

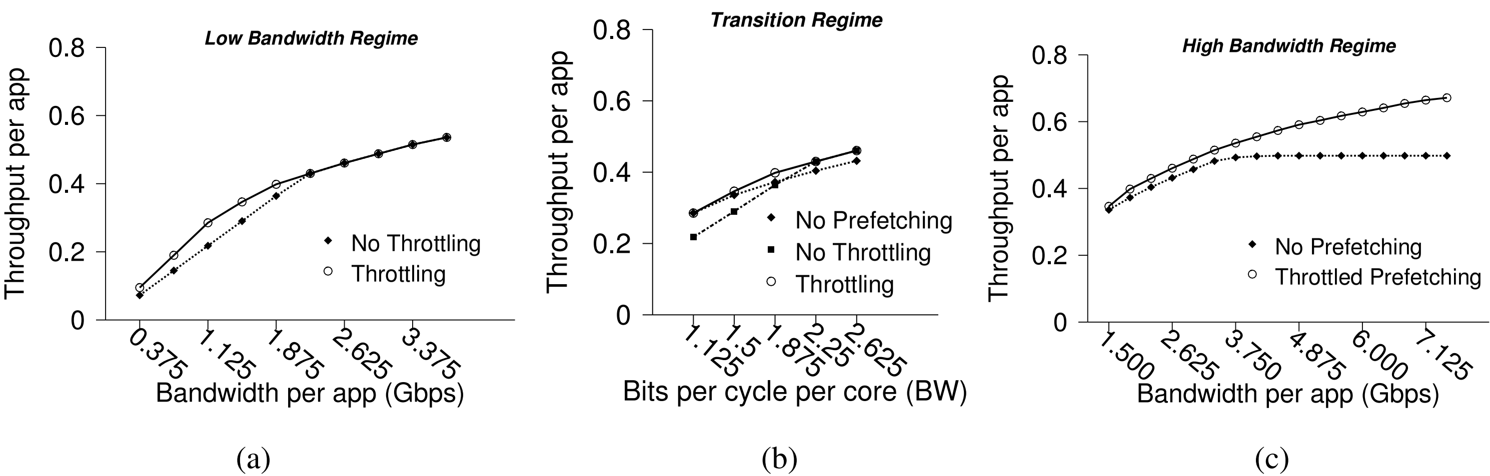
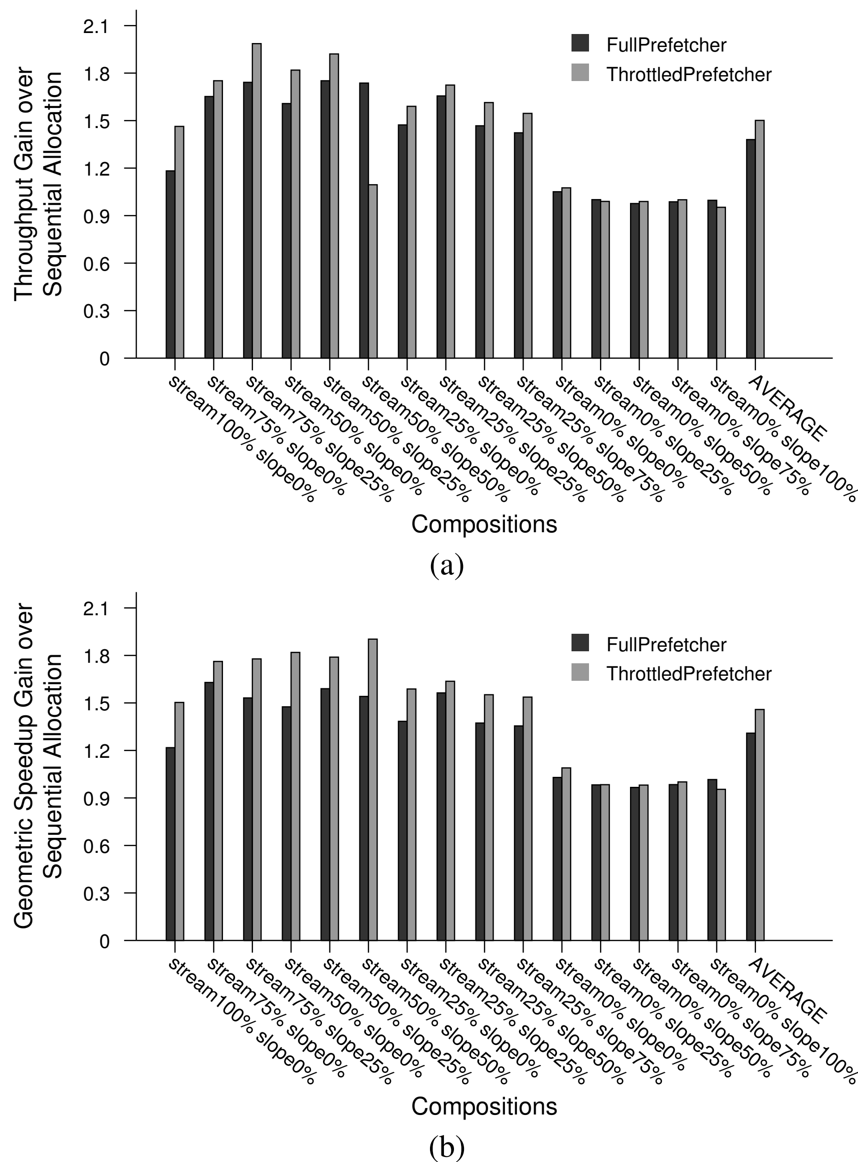


Figure 5.7. ***Prefetcher Throttling maximally utilizes the available bandwidth by intelligently switching between full prefetching, no prefetching, as well as in between aggression levels.*** When provided with sufficient bandwidth, prefetch throttler sends all requests to memory(c), however for lower bandwidth regimes, prefetch throttler switched off prefetching completely to avoid wasting bandwidth on incorrect prefetches (a). Prefetcher throttling mechanism changes the aggression levels at the *right point* between these two regimes (b). Moreover, in between those two regimes only a portion of the prefetches are sent, which leads to a better performance than both no prefetching and full prefetching.



(a)

(b)

Figure 5.8. ***Prefetcher throttling further improves performance gains achieved by simultaneous resource allocation.*** When partitioning cache and bandwidth simultaneously, we get a throughput improvement of 38.02% on average compared to sequential resource management, but when using prefetcher throttling the performs improvement jumps to 50.11% (a). The performance gains are similar when using other metrics, i.e. geometric mean of speedups increase by 30.92% without prefetcher throttling and 45.85% with prefetcher throttling.

throttling, as shown in Figure 5.8(b). Thus, dynamic microarchitectural tuning, such as prefetcher throttling, can further increase resource utilization and performance when using dynamically partitioned architectures, such as TimeCube.

5.5.3 Varying Workload Composition

In this section we try to understand the results for our system, mainly the reasons for the gains in earnings of IaaS providers and fairness for IaaS customers. We run different application mixes on our 32-core chip, but this time we vary the application type compositions by 12.5%, resulting in 45 workload mixes. Different compositions of workload can lead to different overall resource requirements, leading to variations in the improvements in system earning rate and fairness over these mixes. The IaaS application scheduler should take these variations into account to co-schedule symbiotic applications to maximize system earnings and fairness.

In order to evaluate the impact on performance with varying workload composition, we conducted experiments in which we ran 32 applications on a 32 core TimeCube. We find the application execution times for sequential as well as simultaneous allocation. In order to analyze the effects of workload composition we draw two iso-contours, one connecting compositions with equal earning gains and the other connecting compositions with equal slowest application slowdowns, as shown in Figure 5.9.

The experiment results show that the highest system throughput is obtained when all cliff applications are ran for both sequential and simultaneous resource allocation, because both algorithms dynamically allocate the last-level cache partitions to fit the working set sizes of a subset of the cliff applications, resulting in low bandwidth requirement and high IPC for these cliff applications, and the low bandwidth pressure leads to similar performance between the two algorithms. As a result the relative speedup for simultaneous allocation is lowest at this configuration. On the other hand, the performance

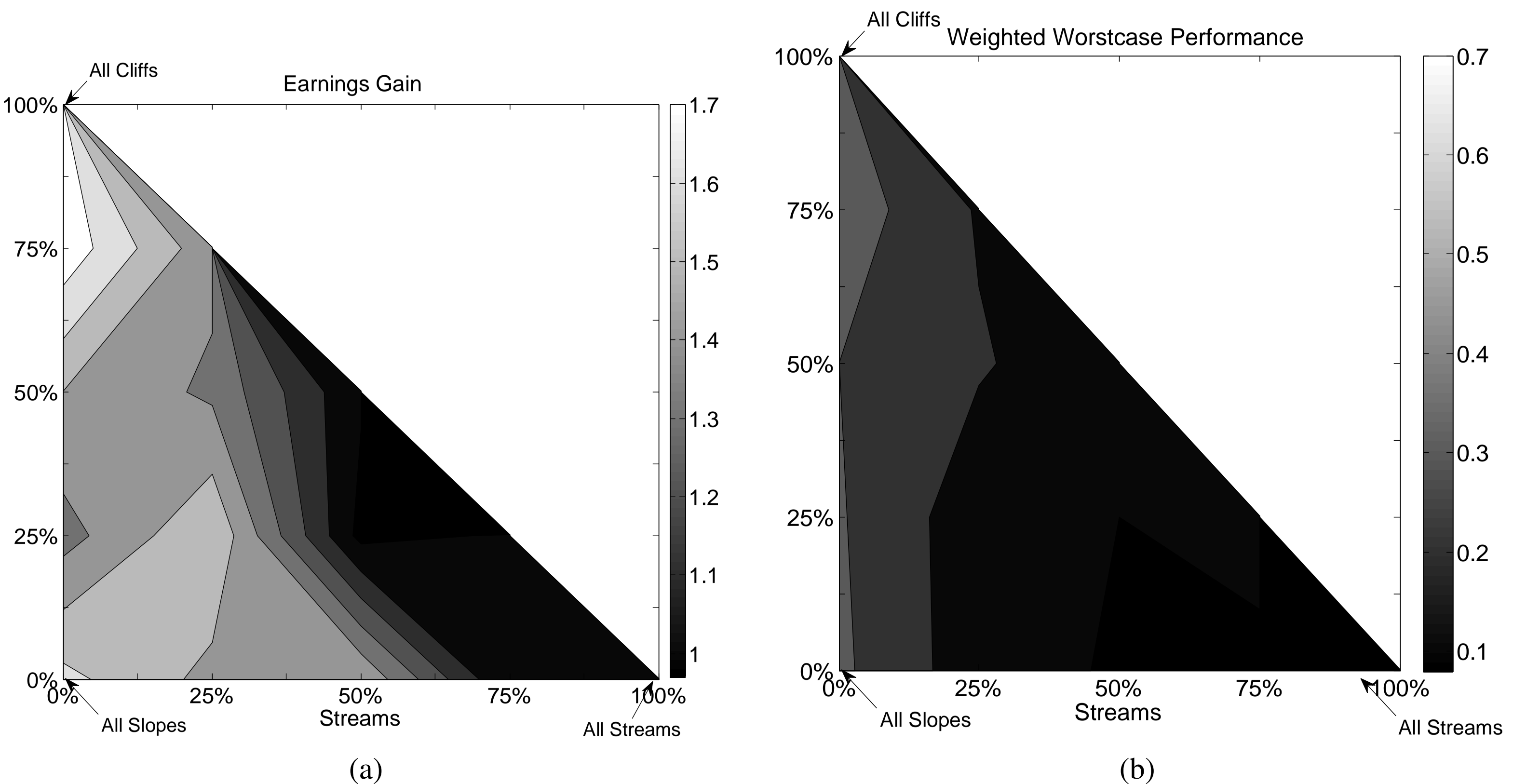


Figure 5.9. ***Variation in IaaS earning gains and fairness with changing workload composition.*** The overall system earning gains reduces (a) and the fairness of the system, as measured in terms of maximum slowdown, decreases (b) as cliffs are replaced by streams. This can be used by IaaS scheduler to decide which applications to co-locate on a chip.

is very low when running only streams. This is because streams create a high bandwidth pressure, while providing a lower IPC. Simultaneous cache and bandwidth partitioning can better handle bandwidth pressures, and provide better performance ($1.2\times$).

In case when only slopes are run, both the algorithms dynamically adjust the cache partitions for the applications. However, they still can't fit the entire working set and there is some bandwidth pressure in the system. Therefore, simultaneous allocation performs better than sequential allocation for this case. The performance decreases as cliffs are gradually replaced by slopes, due to lower IPC provided by slopes compared to cliffs. However, the decline in performance is faster for sequential allocation, due to the gradually increasing bandwidth pressure.

In another case, when cliffs are replaced by streams, the performance drops as cliffs provide higher IPC than streams. But the stream applications also increase the bandwidth pressure. Cliffs are highly sensitive to the memory latencies as a lot of operations depend on every cache miss. So, as the bandwidth pressure increases the miss latencies increase and thus the cliffs start performing poorly. However, this latency increase is much greater for cache only partitioning and thus the cache and bandwidth partitioning performs much better ($2.1\times$) in the case when there is an equal number of cliffs and streams.

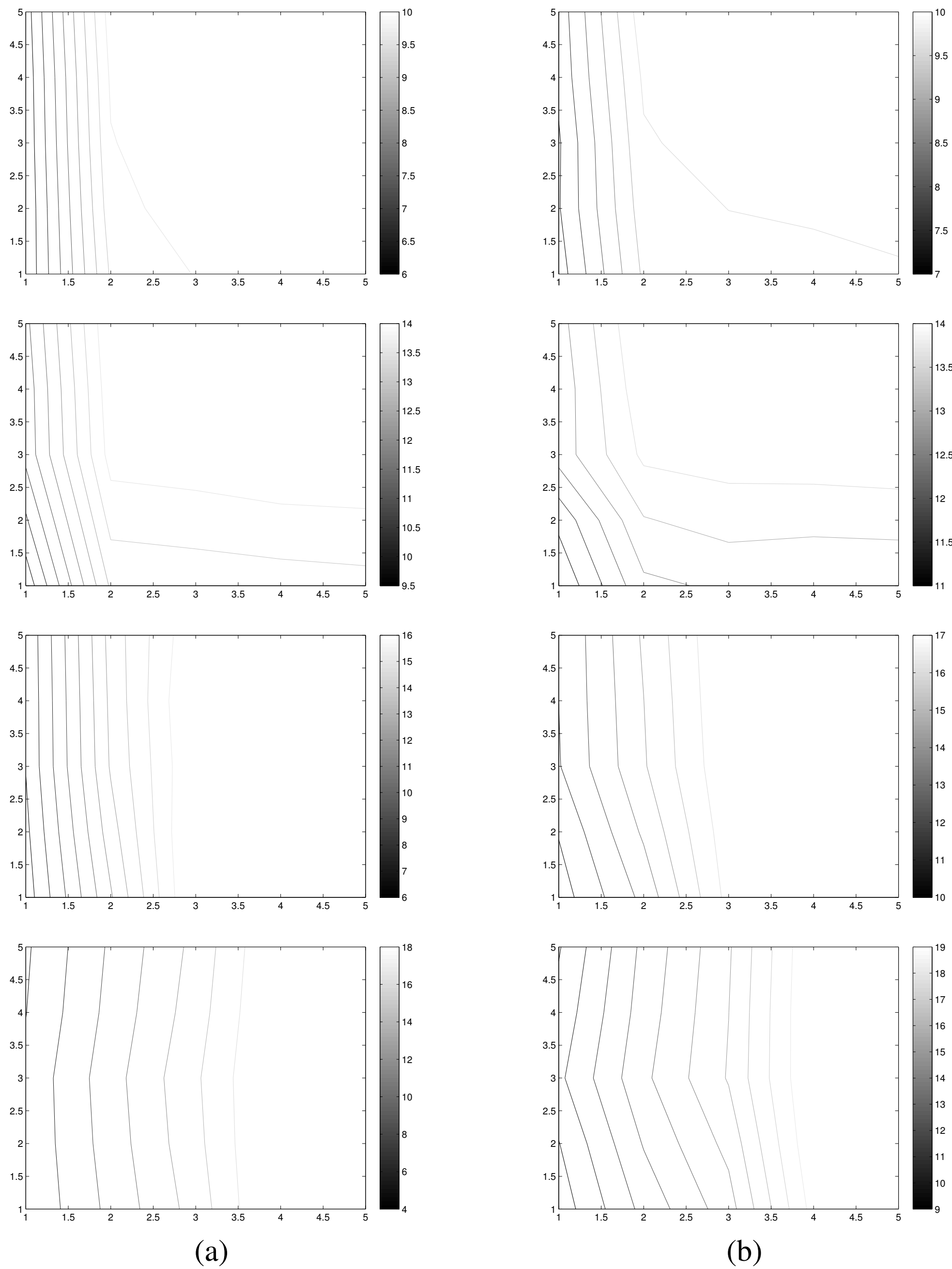


Figure 5.10. ***IPC contours for varying compositions.*** IPC contours for simultaneous and sequential allocation with different compositions.

5.5.4 System Scaling

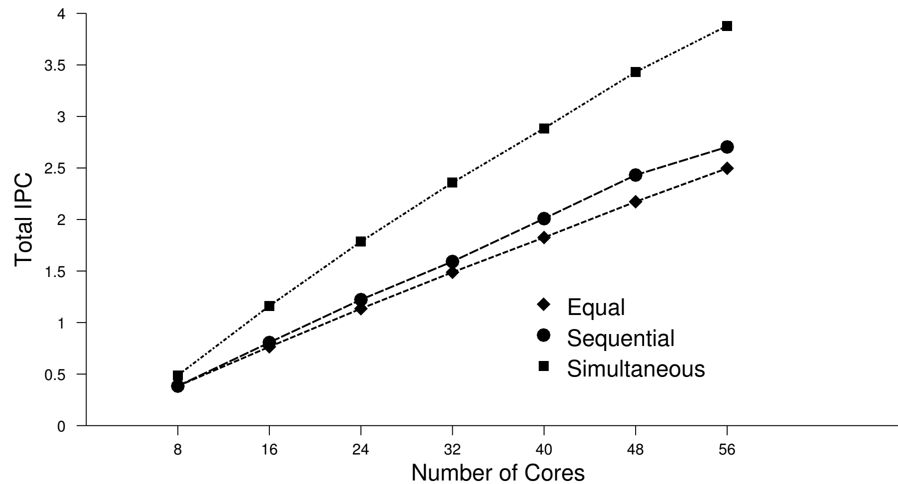


Figure 5.11. ***With TimeCube, performance improvement increases as we increase the system size.*** As we increase the number of cores in the system and accordingly the cache, bandwidth and applications, we see that the baseline system gives a linear increase in performance. With smart cache partitioning, we see a further increase in the performance and even more for simultaneous partitioning. However, we see an increase in the rate of performance improvement with smart partitioning. This is because with more resources, the partitioning algorithm has a higher number of possible configurations to choose from and performs even better.

The mechanism for cache, bandwidth and prefetcher reconfiguration in TimeCube is highly scalable. In fact, as the system size grows, there is an increase in the available configurations, which a provides a greater degree of freedom to the mechanism which translates to even better resource utilization and hence better system performance. We ran an experiment in which we gradually increase the system size from 8 core system to 56 core system in steps of eight. At each step, we also add the same application-set with which we started to the mix of applications running simultaneously.

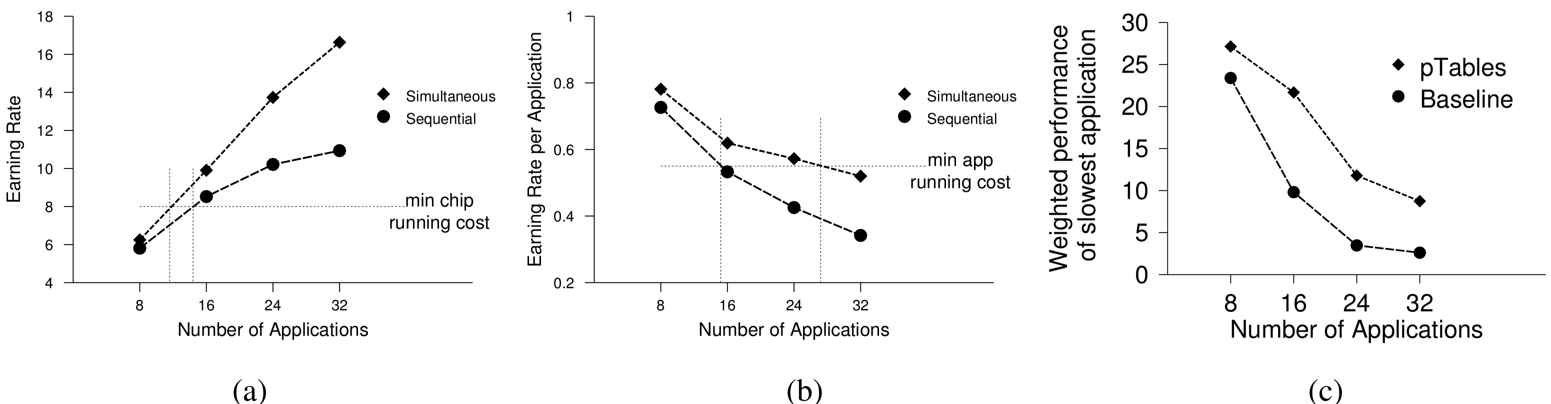
For baseline, we see a linear improvement in system performance, since the resources given to each application is the same, as shown in Figure 5.11. However, we

see a super-linear performance increase when using sequential allocation. But when we allocate resources simultaneously, we see an even greater performance improvement. Moreover, the rate of performance increase also increases gradually. Thus, we get a higher performance gain with bigger system size. This trend continues on adding prefetcher throttling and variable cache switching frequency.

5.5.5 Load characteristics

Elasticity, or the ability to quickly change the deployment size of a customer workload, is highly desirable in IaaS clouds as it allows the customers to quickly scale their services up and down, and this leads to rapid variations in the system load. To understand the effect of changing load on TimeCube performance, we run an experiment with 8, 16, 24 and 32 applications running concurrently on a chip, using two applications per type for diversity.

Our evaluation shows that earning rate of system increases faster for simultaneous partitioning than sequential partitioning, as the number of applications are increased, as shown in Figure 5.12(a) In an IaaS system, there is a minimum chip running cost due to factors such as cooling, rack space and storage, TimeCube provides more flexibility for the IaaS cloud scheduler, as a smaller number of applications are required to meet this minimum cost. With more applications, the earnings per application drop as shown in Figure 5.12(b) due to increased pressure on resources leading to higher application slowdowns. However, this drop is smaller when using simultaneous resource partitioning. Thus, TimeCube can run more applications per chip, while still earning more than the minimum running cost for an application, due to factors such as memory space. The performance of the application with worst slowdown gets worse with increasing number of applications, as shown in Figure 5.12(c). However, simultaneous resource partitioning provides better worst case performance. Thus, TimeCube can run a higher number of



(a) (b) (c)

Figure 5.12. ***TimeCube gives increasingly better performance with increasing number of applications per chip.*** As the number of applications increase on a chip, the total earnings increase (a) even as per application performance reduces (b). TimeCube provides an increasingly better performance compared to baseline. There is a minimum running cost for a chip, which might put a lower bound on applications on a chip. This bound is better for simultaneous resource management (a), which provides a greater scheduling flexibility in the cloud. Moreover, there is a minimum cost of running an application on the cloud along with an expected QoS. This provides an upper-bound on the number of applications that can be run on a chip. Again, simultaneous resource management provides a better upper bound (b). This will reduce the overall required cloud size. Finally, TimeCube gives better fairness due to lower slowdowns in the worst performing application (c).

applications, while still providing the quality of service guaranteed by IaaS vendor in the SLAs.

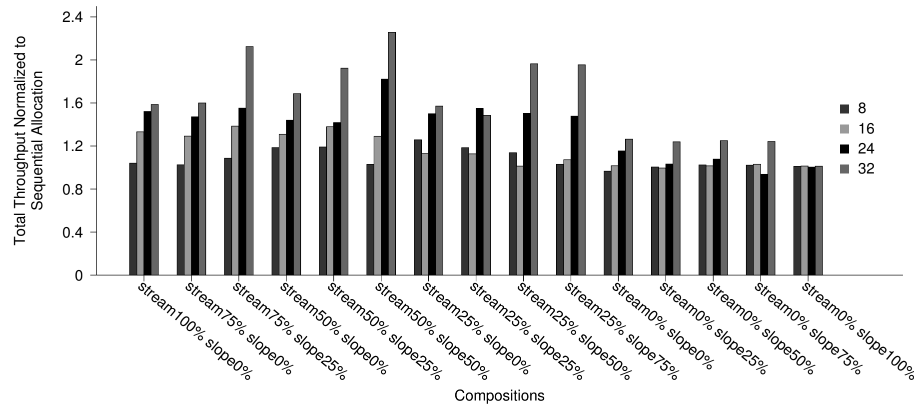


Figure 5.13. ***Simultaneous Resource Allocation performs better under increasing system load.*** For various application compositions, we see that simultaneous resource allocation performs better than sequential allocation and the gains increase as the system load increases, since the resource management becomes increasingly crucial.

We ran experiments to see if different application compositions have different performance impacts for changing system loads. We start with 15 possible compositions and ran 8, 16, 24, and 32 applications by repeatedly adding the same set of applications to the workload. As shown in Figure 5.13, we observed that the performance benefits of simultaneous allocation increases with the increase in system load for almost all the compositions.

In order to understand the impact of system load on varying system compositions, we draw iso-contours for system throughput with sequential as well as simultaneous allocation, as well as the performance improvements, as shown in Figure 5.14. We observe that while the throughput increases with increasing system load, the performance gains with simultaneous allocation also improve with increasing system load, except when using slopes in the system. In particular, compositions with mostly streams and cliffs have a higher performance gains due to higher resource pressure, even at lower

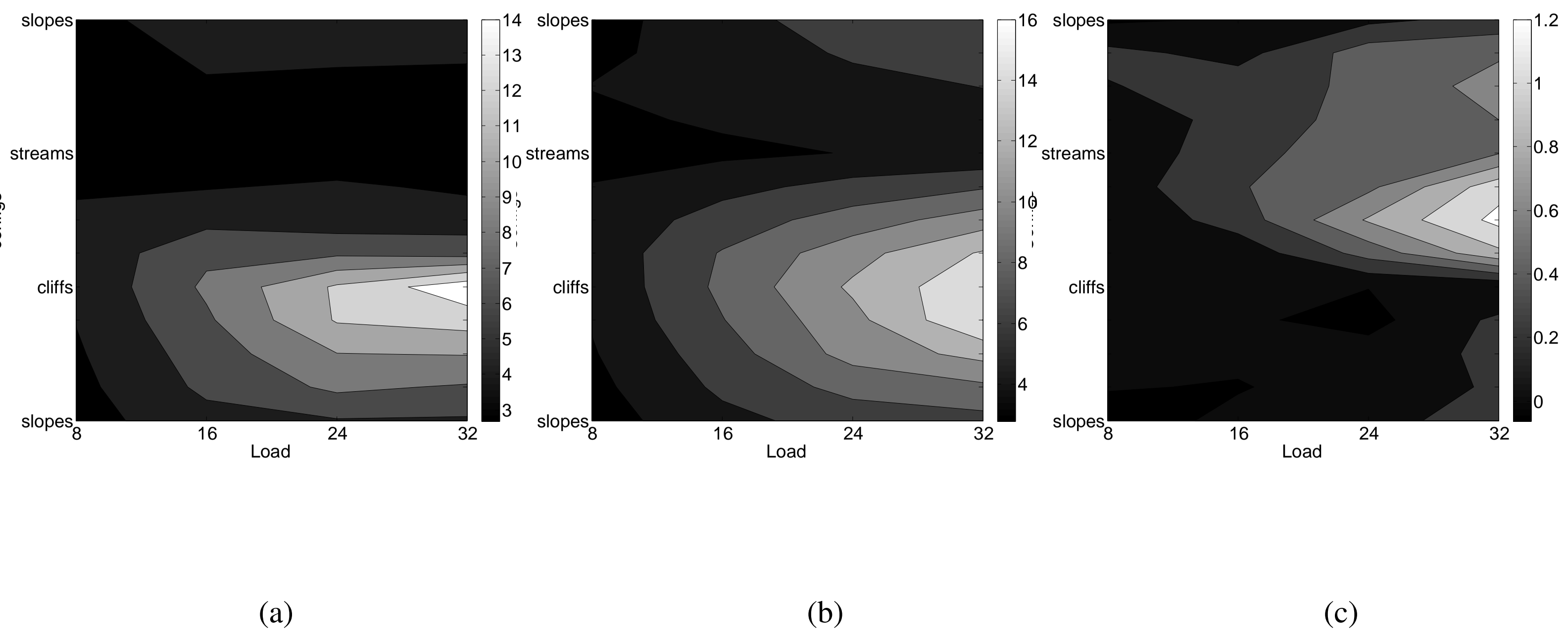


Figure 5.14. ***IPC and speedup wrap-contours for changing load.*** Wrap-contours depicting IPC changes for sequential and simultaneous resource allocations, along with speedups, and the changes in these metrics with increasing application load on the processors.

system loads. We present more detailed iso-contours in Figure 5.15.

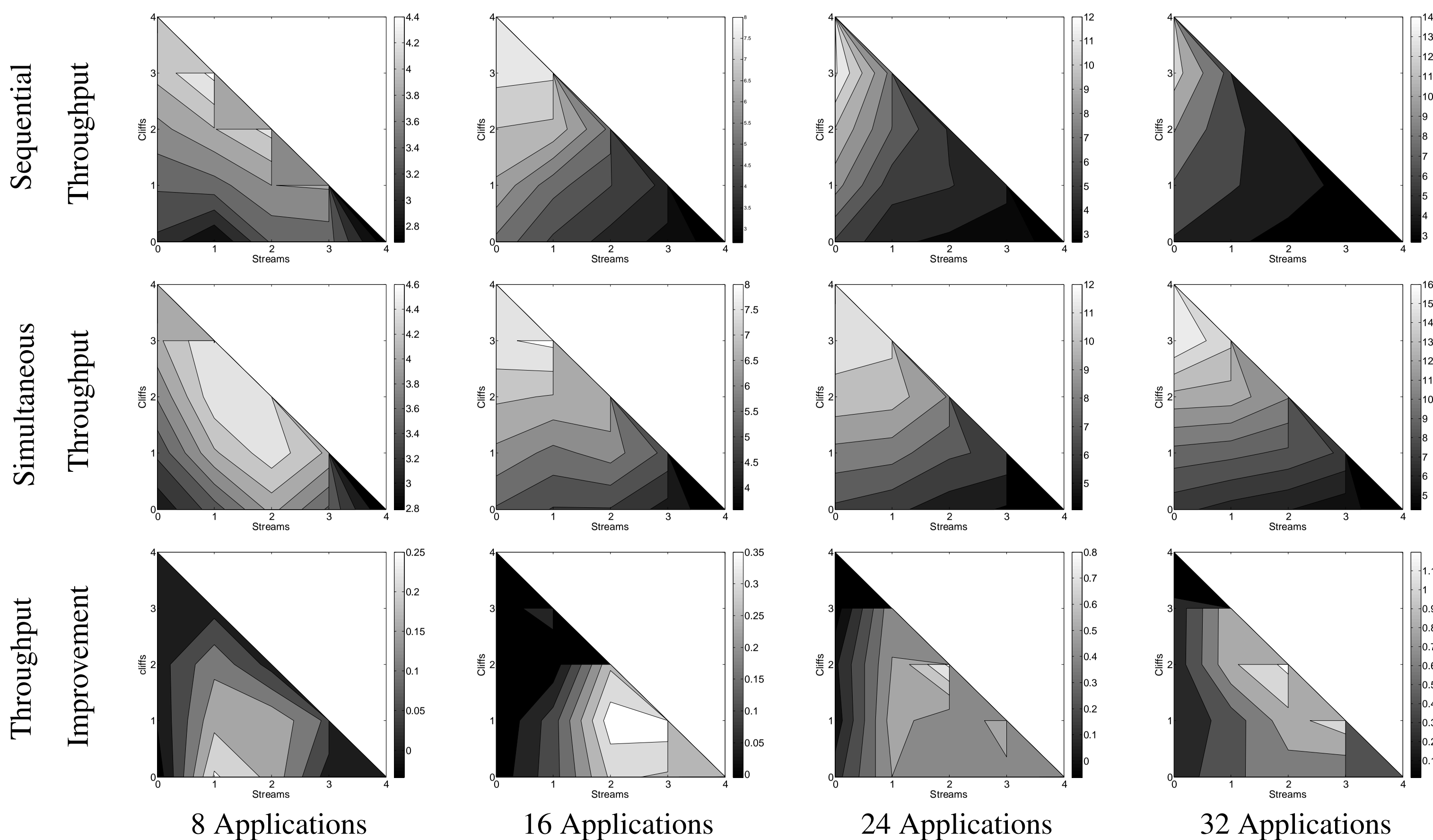


Figure 5.15. ***IPC and speedup contours for changing load.*** Contours for ipc for sequential and simultaneous allocation, as well as the resulting speedups for changing application load.

5.5.6 Varying Workload Diversity

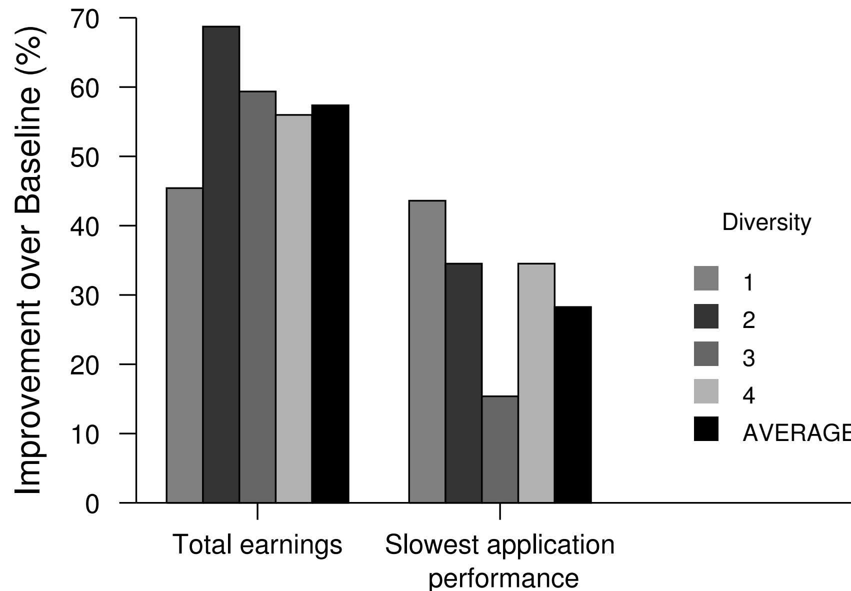


Figure 5.16. ***The performance improvements of TimeCube are impervious to changing diversity of applications within types.*** On changing the numbers of diverse applications representing an application type, we see that the earning rate gains are not diminished. The fairness, weighted performance of application with maximum slowdown is also better for TimeCube.

Varying Workload Diversity Resource partitioning distributes the cache and bandwidth between applications based on their varying requirements. In our experimental evaluation we use application classification into streams, slopes, and cliffs, and even if two applications belong to the same class, they might have slightly different resource requirement, which can have performance impact on a resource limited system like manycore processors. Therefore, an important variable in our experiments is the number of unique applications within a class, which we call **diversity** of the workload. To study the impact of varying diversity on TimeCube, we run an experiment with varying diversity (1 to 4) of applications within each type. Our experiment results show that the

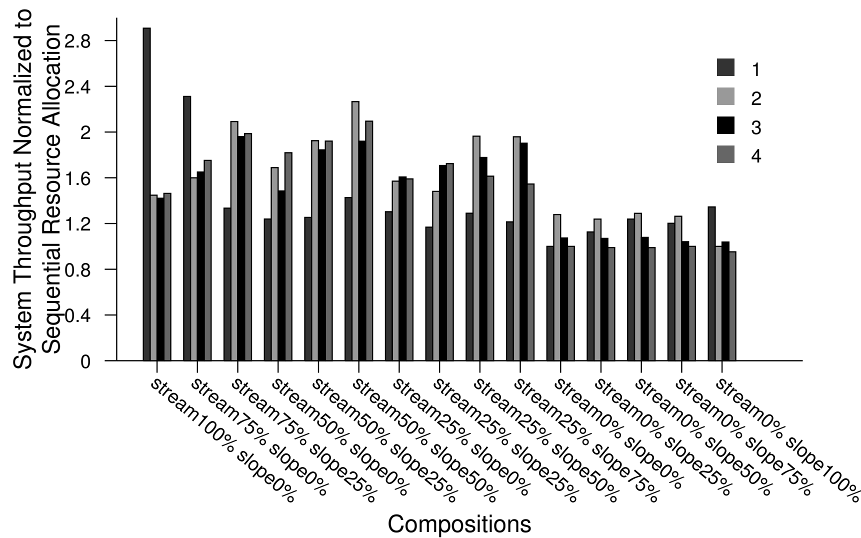


Figure 5.17. ***System throughput improvement remains high with changing application diversity.*** We ran experiments with different application compositions and changing application diversities (1-4). We found that simultaneous resource allocation consistently gives higher throughput compared to sequential resource allocation.

improvements in IaaS earning rates remains unaffected by the diversity in the workload mix (Figure 5.16). We also measured the system fairness, for varying diversity using the performance of application with worst case slowdown and TimeCube gives a better performance compared to baseline, and thus the customers can expect faster turn-around times.

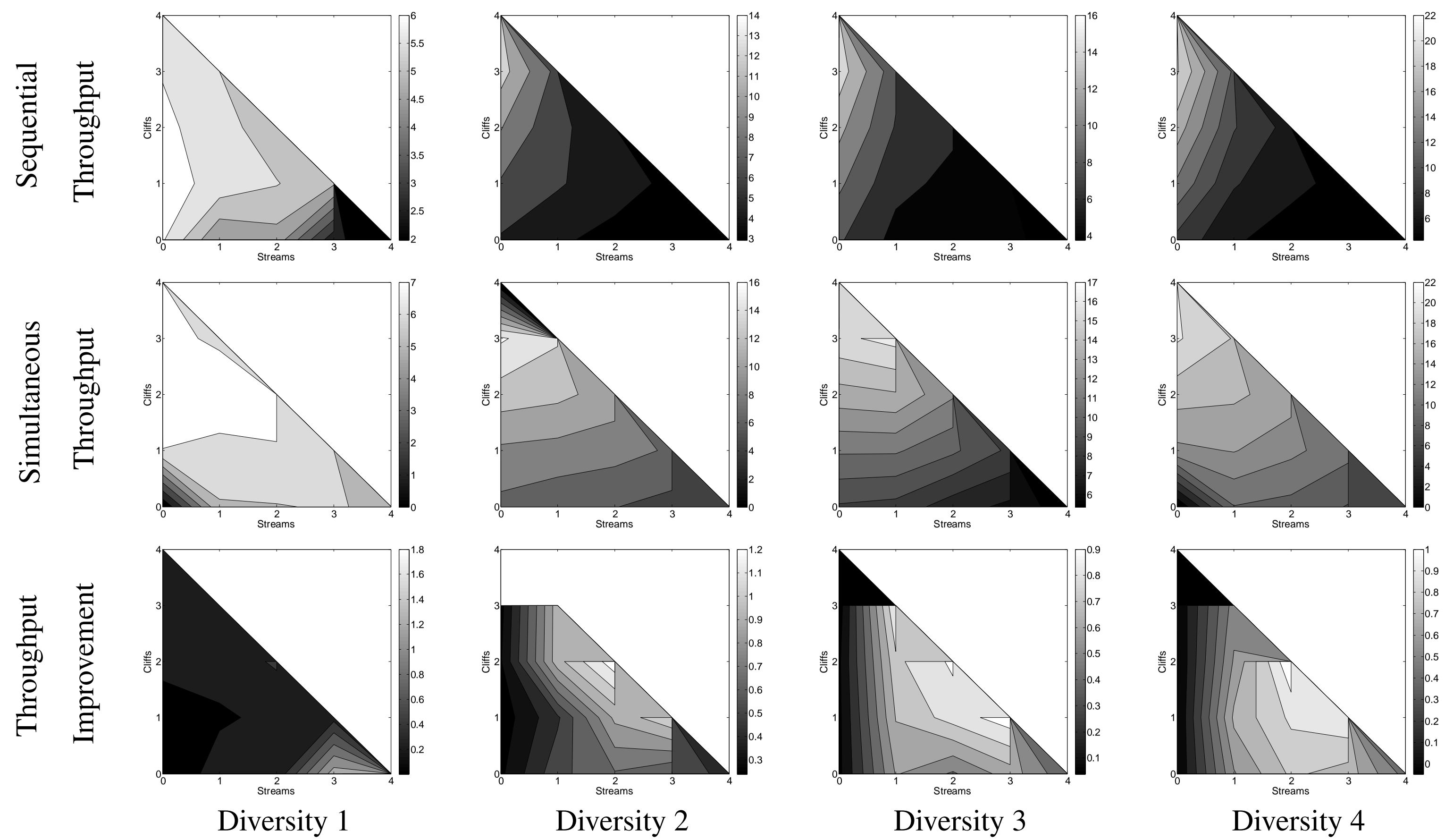


Figure 5.18. ***Throughput and speedup contours for changing application diversity*** . Contours for absolute system throughput when using sequential as well as simultaneous allocation, and the throughput gains for simultaneous allocation while changing the diversity in applications used per type show that TimeCube's resource allocation provides higher throughput when using application with varying diversities.

5.5.7 Variable Cache Switching Frequency

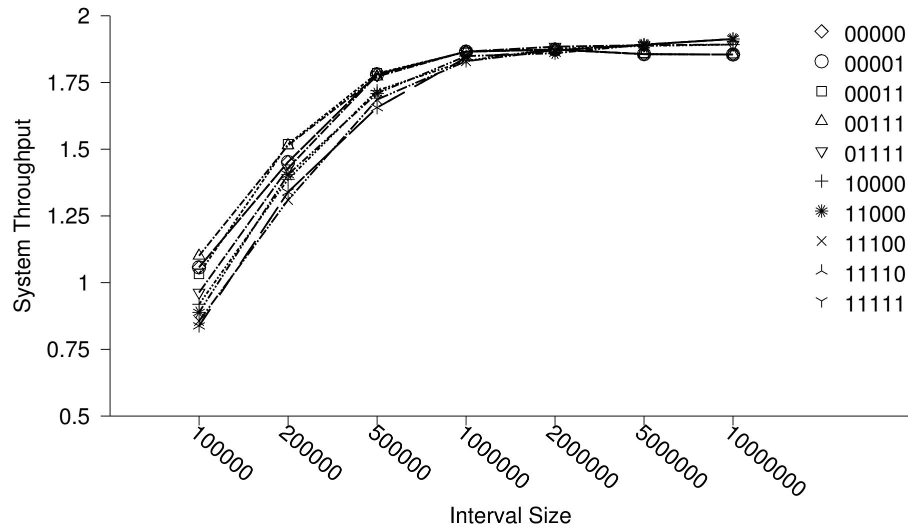


Figure 5.19. ***Different mappings between cache size and reconfiguration interval have varying benefits.*** On changing cache configuration, the reconfiguration penalty is higher for bigger cache sizes. To counteract that, we should reconfigure the larger cache sizes at a slower frequency. Thus, we change the cache reconfiguration interval based on the cache size. However, their can be different monotonic mappings between cache sizes and interval lengths. As we increase the interval size we see an improvement in performance for all mappings. This is because the switching costs exceed the fine grained tracking benefits. However, for smaller intervals we see that the variable cache switching frequency gives better performance due to less switching.

In order to amortize the cost of changing cache size, we reconfigure the larger cache sizes after longer intervals. In order to simplify the system, we vary the interval sizes by powers of two. However, this variation can be done in different steps, i.e. we can change the interval length at every cache size change or we can skip some cache size changes and keep the interval length unchanged between those two cache sizes. We encode this using a binary string in which 1 means the interval length changes and 0 means it remains unchanged. Thus, 11111 means we change the interval length with every cache size change, whereas 00000 means the interval length remains constant

always. We fix the interval length for the smallest cache size, i.e. 1 cache block and then follow the Frequency Change Encoding (FCE), to determine the interval length for larger cache sizes.

We ran an experiment by running 4 applications (apsi, swim, vpr, vpr) on a 4 core system (Fig 5.19). We vary both the base interval length as well as the FCE. The performance is lower at lower base interval lengths, since the cache switching happens at higher frequency and their costs outweigh the benefits of finer grained partitioning. We notice that for smaller base interval lengths, variable cache switching frequency is beneficial, as it reduces the switching frequency for larger cache sizes. As we increase the base interval length, we see that the performance improves since the switching penalty reduces. However, for larger base interval lengths, around 1 million cycles, the performance remains the same, since the switching happens rarely. As a result, the constant switching frequency performs as well as variable switching frequency. Thus, for the remaining results, we choose 1 million cycles as base interval length.

5.5.8 Cache and Bandwidth Sensitivity Study

An increasing cache size leads to a better application performance. Similarly, an increasing amount of total shared cache in the system should lead to an improvement in overall system performance. We ran an experiment with four applications (apsi, swim, mgrid, mgrid) running simultaneously on a 4 core system (Fig 5.20a). However, we vary the total amount of shared cache in the system.

For the baseline system (equal resource allocation), we see a very small increase in system performance with increasing cache size. With sequential allocation, we see a small improvement but as the total cache area increases, the cache size switching penalties increase and eventually the performance drops. When allocating cache and bandwidth simultaneously, we get a much better performance compared to baseline and

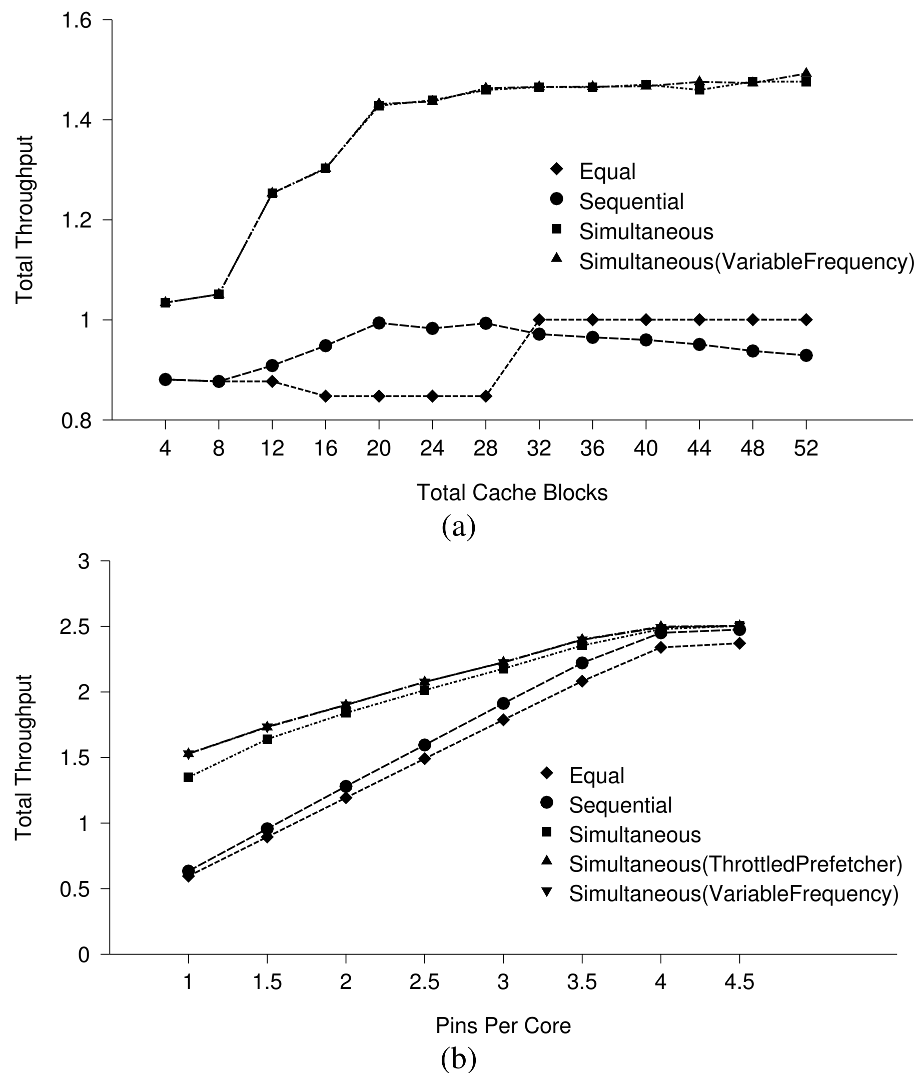


Figure 5.20. ***Simultaneous cache and bandwidth allocation provides higher resource utilization for a range of total cache and memory bandwidths.*** With a large enough cache size, the equal cache allocation performs as well as Sequential allocation (a). However, the simultaneous cache and bandwidth allocation provides a much better performance. On reducing the available cache size, we see a graceful degradation in performance with simultaneous allocation, unlike equal allocation, which shows a steep drop in performance. Even at smaller cache size, simultaneous allocation performs better. As we look at different bandwidth regimes (b), we see that at low bandwidths, simultaneous allocation performs significantly better. As we increase the bandwidth, the performance gradually increases for all algorithms. Eventually, at high enough bandwidth regimes, we see that simultaneous allocation and prefetcher throttling provide no gains on top of sequential allocation. Though sequential allocation has some performance benefit over equal allocation due to the presence of some cliff applications.

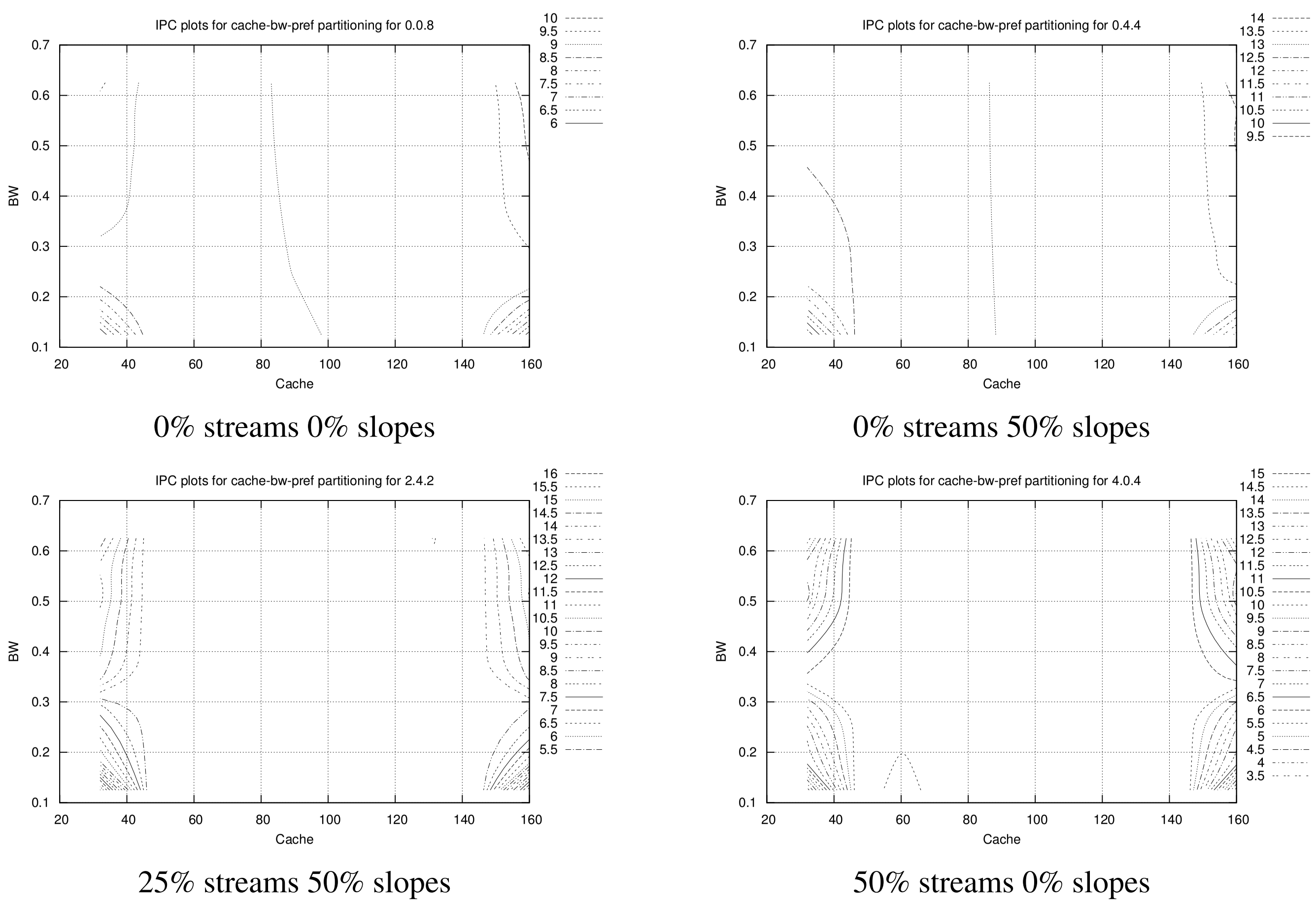
sequential allocation. Moreover, as the cache size increases, this benefit increases as well and at no point do we see a drop in performance due to cache switching penalties. On adding prefetcher throttling to the system, we see an even better performance by the system.

In a bandwidth limited system, increasing bandwidth will lead to a better performance. However, it is important to utilize the bandwidth most judiciously when it is scarce. We run the same experiment as we did for cache sensitivity, but this time we vary the overall system bandwidth (Fig 5.20b). At lower bandwidths, simultaneous cache and bandwidth allocation performs much better than baseline and the sequential allocation. As we increase the bandwidth, we see a gradual and steady increase in performance for all the algorithms. At high enough bandwidths, the system is no longer bandwidth limited and we see no performance improvement with bandwidth increase. But even in those regimes, the techniques using demand-based resource allocation perform better than the baseline by 5.4%.

We did a design space exploration to determine the impact of total last-level cache and memory bandwidth in a 32-core TimeCube instance. We ran different application compositions with varying total last-level cache and memory bandwidth, and drew iso-contours to connect the configurations with the same total IPC, or throughput, as shown in Figure 5.21. These iso-contours can be highly useful for architects trying to determine the amount of cache and DRAM pins required in a manycore chip when targeting a specific workload, such as in embedded systems or datacenters.

5.5.9 Area and Energy Distribution in TimeCube

I now analyze the energy and area distribution for TimeCube. For an example 32 application mix, we observe that the portion of total energy consumed in L2 access is low (0.50%), as shown in Figure 5.22. Most of the energy is consumed in core execution



0% streams 0% slopes

0% streams 50% slopes

25% streams 50% slopes

50% streams 0% slopes

Figure 5.21. ***System IPC iso-contours showing the cache and bandwidth sensitivity of TimeCube.*** On a 32 core TimeCube, we change the total cache and memory bandwidth available in the system, and create iso-contours connecting the hardware configurations with same IPC. Hardware manufactures can utilize these iso-contours to determine the cache and bandwidth requirements for SPEC-like workloads.

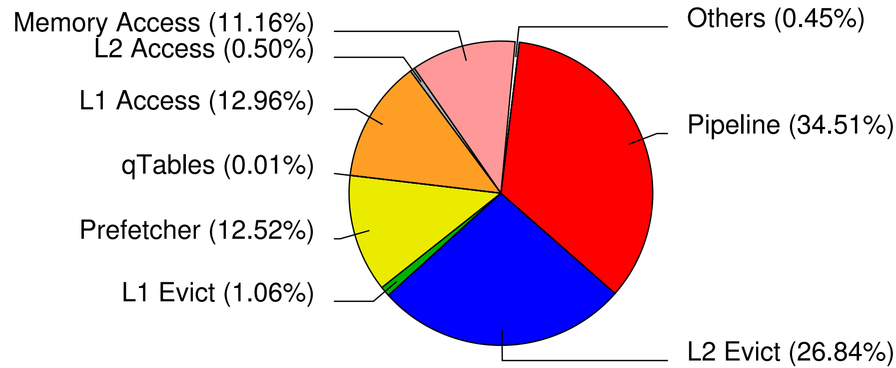


Figure 5.22. ***Overall energy distribution in TimeCube.*** Energy consumed by Quality Tables (0.01%) is very small. The energy consumed by shadow structures is small (0.23%), and the reconfiguration (0.06%) costs are also low.

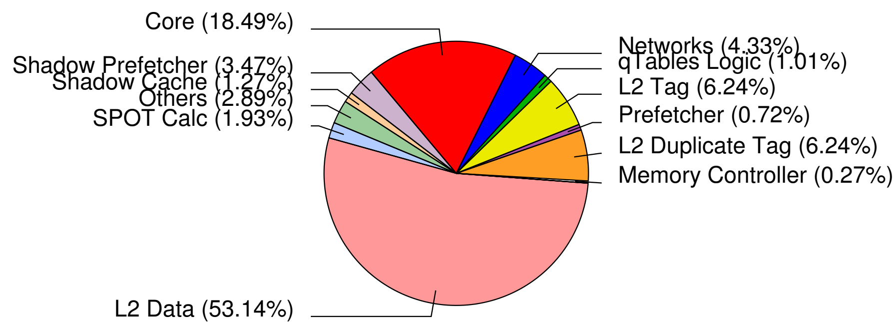


Figure 5.23. ***Overall area distribution in TimeCube.*** The area consumed by shadow-tags and Quality Tables is small (1.27% and 1.01%, respectively). Area consumed by resource allocation is 1.93%. For a 32 core TimeCube, the mechanisms added for Dynamic Execution Isolation consume less than 7% of total chip area. Overall the TimeCube augmentations consume less than 14% of the chip area.

(47.47% including L1 access) and main memory operations (45.36% for access and writeback). Energy consumed for supporting Quality Time is low, i.e. 0.01%, while the energy consumed in using Quality Time to allocate resources was 0.23%. The microarchitectural mechanisms required to estimate Quality Time consume less than 6% chip area. Shadow-Tags consumes 1.27%, Shadow Prefetcher consumes 3.47%, and

Quality Tables 1.11%, as shown in Figure 5.23. Area consumed by hardware resource allocation (1.93%) is small as well. Overall, the mechanisms for measuring and using Quality Time in TimeCube are energy and area efficient.

5.6 Related Work

Simultaneous Shared Resource Management. TimeCube allocates multiple resources simultaneously, and the algorithm can be used with existing resource partitioning mechanisms. In another simultaneous allocation technique proposed by Bitirgen et al. [BIM08], the cache and bandwidth are simultaneously allocated using machine learning; however; it requires a training phase and provides no transparency regarding slowdowns, which makes decisions such as metering the customers in a IaaS computing environment difficult or erroneous. Srikantaiah et al. [SK10] also propose simultaneous resource partitioning; however, they assume a simple exponentially decaying miss rate with increasing cache size, which is an oversimplification as seen in Figure 3.4. TimeCube allocation scheme is online as opposed to offline profiling based allocation schemes proposed by Liu et al. [Chu04] and Suh et al. [SDR02a]. Federova et al. [FSSN05] examined OS-level management to optimize CMT (multi-thread CMPs) performance; however, TimeCube is able to provide a finer grained control over application execution rates.

Sequential Resource Allocation. Previous work has also proposed individual resource partitioning such as cache partitioning based on marginal utilities by Qureshi et al. [MQ06]. Guo et al. [GSZI07] propose cache providing quality of service by controlling cache partitions though strict, elastic, and opportunistic allocation. Hsu et al. [HRIM06] evaluates various cache partitioning policies such as providing fairness and maximizing throughput. Independent bandwidth partitioning has been previously

proposed by Rafique et al. [RLT07] which aims to provide fair bandwidth distribution between application by adaptively changing the quota of an application based on the observed DRAM latency. Liu et al. [LJS10] propose an algorithm to partition bandwidth between applications with the aim of increasing weighted speedup of system. These mechanisms can be used in serial resource distribution, and as I demonstrated, they perform poorly compared to simultaneous partitioning in a bandwidth limited system.

Liu et al. [LSK04], Iyer et al. [IZG+07], Moreto et al. [MCRV08], Stone et al. [STW92a] and Chiou et al. [CJDR00] examined allocation of cache ways in a way-based partitioned cache. Stone et al. [STW92b] studied models for optimal allocation of cache across multiple streams. Suh et al. [SRD04] proposes a way-based cache partitioning scheme, which gives cache based on marginal gains. Zhou et al. [Pin04] examine page allocation based on miss ratio curves. Chandra et al [CGKS] examined inter-thread cache contention to prevent thrashing. Kim et al [SK04] examine multiple online metrics for dynamic cache allocation. Iyer et al. [Iye03] employed classification methods and a variety of allocation mechanisms to assign thread priorities. Suh et al. [SDR02a] proposed an offline-profiling based cache partitioning scheme.

Prefetcher Throttling. Prefetchers consume memory bandwidth and reduce memory latency. A number of previous projects discuss ways to control prefetching for throughput and performance reasons, such as Ebrahimi et al. [EMP09] [EMLP09], Srinath et al. [SMKP07], and Lee et al. [LMNP08]. TimeCube's proposed dynamic prefetcher throttling mechanism can be used in conjunction with mechanisms that improve prefetcher accuracy or timeliness such as Ebrahimi et al. [ELMP11] and Lee et al. [LMNP08]. The motivation for prefetcher throttling is to reduce bandwidth pressure which cannot be significantly reduced by existing mechanisms, such as changing prefetching distance as in FDP [SMKP07], which also can be used in conjunction with the prefetcher throttling

mechanism. Ebrahimi et al. [ELMP10] examines how fairness can be enhanced by throttling programs as opposed to trying to find more optimal allocations of resources between threads.

Symbiotic Job Scheduling. When sharing scarce resources between multiple applications, co-scheduling can reduce pressure on the resources and increase performance. Amongst previous works, Cazorla et al. [CKS^{+}05], Jiang et al. [JSCT08], El-Moursy et al. [EMGAD06] and Snavely et al. [ST00] discuss mechanisms for application scheduling.

5.7 Conclusion

Manycore processors provide an increasing number of compute elements per processor, as well as increasing number of resources to enable faster computation on these elements, such as larger on-chip caches, and memory bandwidth etc. TimeCube proposes simultaneous allocation of these resources to application based on the progress made by applications, which provides significantly better performance compared to architectures that allocate resources either equally or disjointly. Thus, progress-based resource allocation can efficiently manage a large quantity of microarchitectural resources while satisfying system objectives such as high throughput, fairness, or Quality of Service guarantees.

Acknowledgment

Parts of these chapters are reprinted from the following papers:

- Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford. "DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache", International Conference on Computer Design, ICCD 2013.

- Gupta, Anshuman; Sampson, Jack; Taylor, Michael Bedford. “TimeCube: A Manycore Embedded Processor with Interference-agnostic Progress Tracking”, International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2013.

Permission to use these contents has been obtained through signed letters from the co-authors. Dissertation author was the primary investigator and author on these papers.

Chapter 6

Conclusion

Multicore architectures are the microprocessor industry standard today and are expected to remain so in the near future with the number of cores continuing to increase. They are commonplace in most computing domains, such as the datacenters and embedded systems, as they provide high energy-efficiency and compute-density. These multicore processors share their resources to increase utilization, which causes interference, i.e. massive (as much as $12\times$ for a 32-core processor) unpredictable application slowdowns during concurrent executions. This interference can lead to three key problems for highly concurrent systems using these multicore processors:

- How to measure execution quality of an application?
- How to provide guarantees about execution quality?
- How to determine resource allocation for the good of many, but without punishing anyone?

These problems are indeed hindering the widespread adoption of the newly introduced manycore processors in these domains, in spite of their benefits.

Through this dissertation, I have presented three novel solutions to these challenges:

- Quality Time, amount of time the application would have taken with *all* CPU resources, can be used to measure execution quality.
- Dynamic Execution Isolation can be used in combination with Quality Tables, data structures providing application Quality Times for all possible resource allocations, to provide guarantees about execution qualities.
- Simultaneous Performance Optimization Table, or SPOT, can be used to determine an optimal resource allocation to maximize the Mean Quality Time, which improves the system performance while maintaining fairness.

I demonstrate in this work that using these techniques we can not only measure the affects of interference, i.e. reduction in execution quality, but we can also precisely control as well as reduce this interference. I presented a user-space software package, called Qtoolkit, as well as a manycore processor, called TimeCube, which implement these ideas in run-time for live systems using novel scalable mechanisms with very low overheads. I presented a detailed evaluation of these implementations, and show that they provide high accuracy in measurements, a fine-grained and precise quality control, and large throughput improvements, in order to demonstrate the practicality of these solutions.

To summarize, I show that using these mechanisms leads to higher system transparency, QoS control, resource utilization, and system performance. My evaluation of Qtoolkit and TimeCube shows that these solutions can be implemented in software as well as hardware with low overheads and high benefits, making a strong case in favor of including these solutions in consolidated multicore systems of today as well as the future.

Bibliography

[AB04] Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Syst.*, July 2004.

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, 1997.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53, April 2010.

[AP93] Anant Agarwal and Stephen D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *ISCA*, 1993.

[arm] The arm cortex-a9 processors. http://goo.gl/7pclOC.

[Bai] Anderson Bailey. Barcelona's innovative architecture is driven by a new shared cache. http://goo.gl/R2t2uy.

[BB02] Guillem Bernat and Alan Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Syst.*, January 2002.

[BCZ05] G. Beccari, S. Caselli, and F. Zanichelli. A technique for adaptive scheduling of soft real-time tasks. *Real-Time Syst.*, July 2005.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *OSDI*, 1999.

[BFPS11] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *International Green Computing Conference and Workshops*, 2011.

[BIM08] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[But06] Giorgio Buttazzo. Research trends in real-time computing for embedded systems. *SIGBED Rev.*, 2006.

[Cas] Jeff Casazza. First the tick, now the tock: Intel microarchitecture (nehalem). http://goo.gl/1PYYWx.

[CGKS] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: International Symposium on High-Performance Computer Architecture*.

[CGKS05] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, 2005.

[Chu04] Chun Liu, Anand Sivasubramaniam, Mahmut Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *HPCA*, 2004.

[CJ06] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO*, 2006.

[CJDR00] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Design Automation Conference*, 2000.

[CKS+05] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramirez, and Mateo Valero. Architectural support for real-time task scheduling in smt processors. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 166–176, New York, NY, USA, 2005. ACM.

[EC2] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[EHE11] S. Eyerman, K. Hoste, and L. Eeckhout. Mechanistic-empirical processor performance modeling for constructing cpi stacks on real hardware. In *ISPASS*, 2011.

[ELMP10] Eiman Eibrahimi, Chang Joo Lee, Onur Mutlu, and Yale Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *Architecture Support for Programming Languages and Operating Systems*, 2010.

[ELMP11] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Prefetch-Aware Shared-Resource Management for Multi-Core Systems. In *ISCA*, 2011.

[EMGAD06] Ali El-Moursy, Rajeev Garg, David H. Albonesi, and Sandhya Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 141–141, Washington, DC, USA, 2006. IEEE Computer Society.

[EMHF09] Erik Elmroth, Fermin Galan Marquez, Daniel Henriksson, and David Perales Ferrera. Accounting and billing for federated cloud infrastructures. In *International Conference on Grid and Cooperative Computing*, 2009.

[EMLP09] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 316–326, New York, NY, USA, 2009. ACM.

[Emm97] P. G. Emma. Understanding some simple processor-performance limits. *IBM J. Res. Dev.*, 41:215–232, May 1997.

[EMP09] E. Ebrahimi, O. Mutlu, and Y.N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 7 –17, 2009.

[FSSN05] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings 2005 USENIX Technical Conference*, 2005.

[GLKS11] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *SOCC*, 2011.

[GST70] J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage

hierarchies. *IBM Syst. J.*, 9:78–117, June 1970.

[GST13] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Timecube: A manycore embedded processor with interference-agnostic progress tracking. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2013.

[GSZI07] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, 2007.

[GWG+] M. Gohner, M. Waldburger, F. Gubler, G.D. Rodosek, and B. Stiller. An accounting model for dynamic virtual organizations. In *International Symposium on Cluster Computing and the Grid, 2007*.

[HDH+10] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Shekhar Borkar, Vivek De, Rob Can Der Wijngaart, and Timothy Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45 nm CMOS. In *ISSCC*, 2010.

[Hen00] J.L. Henning. Spec cpu2000: measuring cpu performance in the new millennium. *Computer*, 2000.

[Hil87] Mark Donald Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, 1987.

[HKS+05] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *ICS*, 2005.

[HLL10] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The cilkview scalability analyzer. In *SPAA*, 2010.

[HRIM06] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *PACT*, 2006.

[IBM] IBM SmartClouds. http://www.ibm.com/cloud-computing/us/en/.

[IMMC08] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[Iye03] R Iyer. On modeling and analyzing cache hierarchies using CASPER. In *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 2003.

[IZG+07] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 25–36, New York, NY, USA, 2007. ACM.

[Jal] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation.

[JSCT08] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM.

[KBK02] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS*, 2002.

[KHMHB10] Yoongu Kim, Dongsu Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.

[KJLH89] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. 1989.

[KMHK12] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. Measuring interference between live datacenter applications. In *SC*, 2012.

[KSCJ10] Dimitris Kaseridis, Jeffrey Stuecheli, Jian Chen, and Lizy Kurian John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. In *HPCA*, 2010.

[LAS+09] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[LB00] Giuseppe Lipari and Sanjoy K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Real Time Technology and Applications Symposium*, 2000.

[LB06] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *SIGARCH Comput. Archit. News*, 2006.

[LCX+12] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, 2012.

[LDM+01] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis using hardware counters. In *International Conference on Parallel and Distributed Computing Systems*, 2001.

[LJS10] Fang Liu, Xiaowei Jiang, and Yan Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *HPCA*, 2010.

[LLD+08] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, 2008.

[LLW+98] Yong Luo, Olaf M. Lubeck, Harvey Wasserman, Federico Bassetti, and Kirk W. Cameron. Development and validation of a hierarchical memory model incorporating cpu- and memory-operation overlap model. In *International Workshop on Software and Performance*, WOSP, 1998.

[LMNP08] Chang Joo Lee, O. Mutlu, V. Narasiman, and Y.N. Patt. Prefetch-aware dram controllers. In *MICRO*, 2008.

[LSK04] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA ’04, pages 176–, Washington, DC, USA, 2004. IEEE Computer Society.

[MBT04] Jason Miller David Wentzlaff Ian Bratt Ben Greenwald Henry Hoffmann Paul Johnson Jason Kim James Psota Arvind Saraf Nathan Shnidman Volker Strumpen Matt Frank Saman Amarasinghe Anant Agarwal Michael

Bedford Taylor, Walter Lee. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA*, 2004.

[MCRV] M. Moreto, F.J. Cazorla, A. Ramirez, and M. Valero. Online prediction of applications cache utility. In *Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2007*.

[MCRV08] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. Mlp-aware dynamic cache partitioning. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 337–352, Berlin, Heidelberg, 2008. Springer-Verlag.

[MM07] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.

[MM08a] Thomas Moscibroda and Onur Mutlu. Distributed order scheduling and its application to multi-core dram controllers. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 365–374, New York, NY, USA, 2008. ACM.

[MM08b] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. *ISCA*, 2008.

[MQ06] Yale Patt Moinuddin Qureshi. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO*, 2006.

[NALS06] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *MICRO*, 2006.

[NLS07] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. In *ISCA*, 2007.

[PAV+01] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *MICRO*, 2001.

[Pin04] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Architecture Support For Programming*

Languages and Operating Systems, 2004.

[RDK+00] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *ISCA*, 2000.

[Rei05] J. Reinders. Vtune performance analyzer essentials. In *Intel Press*, 2005.

[RLT06] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT*, 2006.

[RLT07] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Effective management of dram bandwidth in multicore processors. In *PACT*, 2007.

[RTM+10] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 2010.

[SAWJ+96] I. Stoica, H. Abdel-Wahab, K. Jeffay, S.K. Baruah, J.E. Gehrke, and C.G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Real-Time Systems Symposium*, 1996.

[SB08] J. Amann R. Conlin K. Joyce V. Leung J. MacKay M. Reif S. Bell, B. Edwards. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. In *ISSCC*, 2008.

[Sch10] Richard Schooler. The processor: Many-core for embedded and cloud computing. In *Workshop on High Performance Embedded Computing*, 2010.

[SDR01] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, 2001.

[SDR02a] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *HPCA*, 2002.

[SDR02b] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, 2002.

[SK04] Y. Solihin S. Kim, D. Chandra. Fair caching in a chip multiprocessor

architecture. In *PACT*, 2004.

[SK10] Shekhar Srikantaiah and Mahmut T. Kandemir. Srp: Symbiotic resource partitioning of the memory hierarchy in cmps. In *HiPEAC*, 2010.

[SLT99] Yan Solihin, Vinh Lam, and Josep Torrellas. Scal-tool: pinpointing and quantifying scalability bottlenecks in dsm multiprocessors. In *SC*, 1999.

[SMKP07] S. Srinath, O. Mutlu, Hyesoon Kim, and Y.N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[SRD04] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, April 2004.

[SSVC09a] Mark Stillwell, David Schanzenbach, Frederic Vivien, and Henri Casanova. Resource allocation using virtual clusters. In *International Symposium on Cluster Computing and the Grid*, 2009.

[SSVC09b] Mark Stillwell, David Schanzenbach, Frederic Vivien, and Henri Casanova. Resource allocation using virtual clusters. In *International Symposium on Cluster Computing and the Grid*, 2009.

[ST00] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 234–244, New York, NY, USA, 2000. ACM.

[STW92a] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41:1054–1068, September 1992.

[STW92b] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41(9):1054–1068, 1992.

[teg] Nvidia tegra 4 family cpu architecture: 4-plus-1 quad core. http://goo.gl/Ta30K2.

[TJYD09] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. 2009.

[TMV+11] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, 2011.

[VGR98] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *ASPLOS*, 1998.

[Wal] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*

[WGT+] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. Dramsim: A memory-system simulator. In *SIGARCH Computer Architecture News, September 2005*.

[You07] Matt T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS*, 2007.

[ZDFS07] Xiao Zhang, Sandhya Dwarkadas, Girts Folkmanis, and Kai Shen. Processor hardware counter statistics as a first-class system resource. In *Workshop on Hot Topics in Operating Systems*, 2007.

[ZLTI96] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the mips r10000 performance counters. In *SC*, 1996.

[ZPS+04] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, 2004.