ADDRESSING POWER, PERFORMANCE AND END-TO-END QOS IN EMERGING MULTICORES THROUGH SYSTEM-WIDE RESOURCE MANAGEMENT

A Dissertation in
Computer Science and Engineering
by
Akbar Sharifi

© 2013 Akbar Sharifi

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2013
The dissertation of Akbar Sharifi was reviewed and approved* by the following:

Mahmut Kandemir
Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Chita R. Das
Distinguished Professor of Computer Science and Engineering

Padma Raghavan
Distinguished Professor of Computer Science and Engineering

Qian Wang
Professor of Mechanical Engineering

Lee Coraor
Graduate Program Chair of the Computer Science and Engineering Department

*Signatures are on file in the Graduate School.
Abstract

Multicores are now ubiquitous [1, 2, 3, 4], owing to the benefits they bring over single core architectures including improved performance, lower power consumption and reduced design complexity. Several resources ranging from the cores themselves to multiple levels of on-chip caches and off-chip memory bandwidth are typically shared in a multicore processor. Prudent management of these shared resources for achieving predictable performance and optimizing energy efficiency is critical and thus, has received considerable attention in recent times. In my research study, I have been focusing on proposing novel schemes to dynamically manage various available shared resources in emerging multiocres, while targeting three main goals: (1) Maximizing the overall system performance and (2) Meeting end-to-end QoS targets defined by the system administrator (3) Optimizing power and energy consumption. We consider a wide range of available resources including cors, shared caches, off-chip memory bandwidth, on-chip communication resources and power budgets. Further, towards achieving our goals, we employ formal control theory as a powerful tool to provide the high-level performance targets through dynamically managing and partitioning the shared resources.

Providing end-to-end QoS in future multicores is essential for supporting widespread adoption of multicore architectures in virtualized servers and cloud computing systems. An initial step towards such an end-to-end QoS support in multicores is to ensure that at least the major computational and memory resources on-chip are managed efficiently in a coordinated fashion. In this dissertation we propose a platform for end-to-end on-chip resource management in multicore processors. Assuming that each application specifies a performance target/SLA, the main objective is to dynamically provision sufficient on-chip resources to applications for achieving the specified targets. We employ a feedback based system, designed as a Single-Input, Multiple-Output (SIMO) controller with an Auto-Regressive-Moving-Average (ARMA) model, to capture the behaviors of different applica-
Dynamic management of the shared resources in multicore systems with the goal of maximizing the overall system performance is another main part of this dissertation. As we move towards many-core systems, interference in shared cache continues to increase, making shared cache management an important issue in increasing overall system performance. As a part of my research, we propose a dynamic cache management scheme for multiprogrammed, multithreaded applications, with the objective of obtaining maximum performance for both individual applications and the multithreaded workload mix. From an architectural side, in parallel to increasing core counts, network-on-chip (NoC) is becoming one of the critical shared components which determine the overall performance, energy consumption and reliability of emerging multicore systems. In my research, targeting Network-on-Chip (NoC) based multicores, we propose two network prioritization schemes that can cooperatively improve performance by reducing end-to-end memory access latencies. In our another work on NoCs, focusing on a heterogenous NoC in which each router has (potentially) a different processing delay, we propose a process variation-aware source routing scheme to enhance the performance of the communications in the NoC based system. Our scheme assigns a route to each communication of an incoming application, considering the processing latencies of the routers resulting from process variation as well as the communications that already exist in the network to reduce traffic congestion.

Power and energy consumption in multicores is another important area that we target in my research. In one of the sections of this dissertation, targeting NoC based multicores, we propose a two-level power budget distribution mechanism, called PEPON, where the first level distributes the overall power budget of the multicore system among various types of on-chip resources like the cores, caches, and NoC, and the second level determines the allocation of power to individual instances of each type of resource. Both these distributions are oriented towards maximizing workload performance without exceeding the specified power budget.

As the memory system is a large contributor to the energy consumption of a server, there have been prior efforts to reduce the power and energy consumption of the memory system. DVFS schemes have been used to reduce the memory power, but they come with a performance penalty. In my research study, we propose HiPEMM, a high performance DVFS mechanism that intelligently reduces memory power by dynamically scaling individual memory channel frequencies. Our strategy also involves clustering the running applications based on their sensitivity to memory latency, and assigning memory channels to the application clusters.
# Table of Contents

List of Figures ix

List of Tables xv

Acknowledgments xvii

Chapter 1

Introduction 1

1.1 Meeting End-to-End QoS in Multicores . . . . . . . . . . . . . . 1
1.2 Addressing End-to-End Memory Access Latency in Multicores . 2
1.3 Power Budgeting in Multicores . . . . . . . . . . . . . . . . . . . 3
1.4 Saving Energy in Multicore Memories . . . . . . . . . . . . . . . 4
1.5 Shared Cache Capacity Management in Multicores . . . . . . . 6
1.6 Targeting NoC in Emerging Multicores . . . . . . . . . . . . . . 6

Chapter 2

METE: Meeting End-to-End QoS in Multicores Through System-
Wide Resource Management 9

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2 Motivation and Background . . . . . . . . . . . . . . . . . . . . . 11
  2.2.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.2.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3 Implementation Issues . . . . . . . . . . . . . . . . . . . . . . . . 16
2.4 Control Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 17
  2.4.1 High Level View . . . . . . . . . . . . . . . . . . . . . . . . . . 17
  2.4.2 System Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
  2.4.3 Application Controller . . . . . . . . . . . . . . . . . . . . . 20
4.3 Our Proposed Approach ........................................ 69
  4.3.1 High-Level View of PEPON ........................... 69
  4.3.2 Design Details of PEPON ............................... 70
    4.3.2.1 First-Level Power Distribution .................. 71
    4.3.2.2 Second-Level Power Distribution ............... 72
    4.3.2.3 Design of the Power Controllers ............... 74
    4.3.2.4 Implementation Overheads ..................... 77
4.4 Experimental Evaluation .................................. 78
  4.4.1 Power Estimation Models ............................. 78
    4.4.1.1 Schemes for Performance Evaluation .......... 80
  4.4.2 Results ................................................. 81
    4.4.2.1 Performance Results ............................. 81
    4.4.2.2 Dynamics of PEPON .............................. 83
    4.4.2.3 Sensitivity Experiments ....................... 87
  4.4.3 Power Budget Violations ............................ 88
4.5 Related Work .............................................. 88
  4.5.1 Feedback Control Based Power Management ............ 89
  4.5.2 Ad-hoc Power and Thermal Management ................ 90
4.6 Concluding Remarks ...................................... 90

Chapter 5
HiPEMM: A High Performance Energy-Saving Mechanism for Multicore Memories 92
5.1 Introduction ................................................. 92
5.2 Background and Motivation ............................... 93
  5.2.1 Background ............................................. 93
  5.2.2 Motivation ............................................. 95
5.3 HiPEMM: Our Proposed Energy Saving Mechanism ........ 98
  5.3.1 HiPEMM Algorithm ................................... 99
5.4 Experimental Evaluation ................................ 103
5.5 Related Work ............................................... 111
5.6 Conclusions ................................................. 112

Chapter 6
Courteous Cache Sharing: Being Nice to Others in Capacity Management 113
6.1 Introduction and Motivation .............................. 113
6.2 Our proposed scheme ................................... 117
  6.2.1 Prioritization Scheme ............................... 117
  6.2.2 CCS Architecture ................................... 119
6.2.3 Implementation Issues and Overheads .................................. 123
6.2.4 Discussion ........................................................................ 124
6.3 Experiments ........................................................................... 125
6.4 Related Work .......................................................................... 138
6.5 Conclusions .......................................................................... 140

Chapter 7
Process Variation-Aware Routing in NoC Based Multicore .......................... 142
7.1 Introduction ........................................................................... 142
7.2 Background and Motivation ..................................................... 143
  7.2.1 Process Variations ............................................................... 143
  7.2.2 Process Variation in NoCs ..................................................... 144
7.3 P-V Aware Source Routing ...................................................... 144
  7.3.1 Our Source Routing Algorithm ............................................ 145
  7.3.2 Flexibility Metric ............................................................... 147
  7.3.3 Routing ............................................................................ 148
  7.3.4 Handling Deadlocks ........................................................... 149
7.4 Experimental Evaluation .......................................................... 150
  7.4.1 Setup ............................................................................. 150
  7.4.2 Results ........................................................................... 151
    7.4.2.1 Results with Different Mappings ................................... 151
    7.4.2.2 Impact of Communication Volume ............................... 153
    7.4.2.3 Path Reassignment ...................................................... 154
    7.4.2.4 Impact of the Flexibility Metric ................................. 156
7.5 Conclusion ............................................................................. 156

Chapter 8
Conclusions .................................................................................. 157
  8.1 Summary of Contributions ..................................................... 157

Bibliography ................................................................................. 160
List of Figures

2.1 Three types of shared resources provisioned to two co-runner applications. .............................................. 10
2.2 The impacts of resource allocation on the performance of two running applications. ............................... 10
2.3 Behavior of applu in a system consisting of three controllers (one for each resource). Oscillation occurs, since at time t the cache layer controller increases/decreases the allocated cache space to meet the performance target unaware of the changes made by the off-chip bandwidth controller to the memory bandwidth allocation. .............................................. 14
2.4 High level view of a feedback control system with a SIMO controller. .................................................. 15
2.5 High level view of our control architecture for n number of applications. .............................................. 18
2.6 Input, output and state in a state space model. ......................................................................................... 20
2.7 Detailed view of an application controller. ............................................................................................... 20
2.8 Measured IPC values vs. predicted IPC values by our model. The horizontal axis shows different resource assignments considered for our model evaluation. In (c : w : m), “c” is number of allocated cores, “w” is the number of allocated ways from the shared L2 cache and “m” (%) is the off-chip bandwidth allocation. .......................................................... 26
2.9 Average IPC values achieved by different mixes (workloads). ............................................................... 26
2.10 Tracking the IPC targets for eight applications from Mix1-Mix6. The IPC targets are shown in dash lines in the charts. .......................................................... 29
2.11 Cache allocation during execution. ........................................................................................................ 30
2.12 Off-chip bandwidth allocation during execution. ...................................................................................... 30
2.13 Cores allocation during execution. ........................................................................................................ 30
2.14 QoS target sensitivity. .......................................................................................................................... 31
2.15 Transducer design for tracking high-level performance metrics. ......................................................... 31
2.16 Resource Broker evaluation (lack of resources). ..................................................................................... 32
2.17 Resource Broker evaluation (use of excess resources). ......................................................................... 32
2.18 Results for 4 applications and 16-core simulations. .............................................................................. 35
2.19 Sensitivity to execution interval. ........................................... 35
3.1 A 5 × 5 NoC based multicore. .............................................. 40
3.2 Detailed flow of a memory access. In order to look-up the correspond- 
ing L2 bank, a request message is sent from L1 to L2 (path1). If the 
data is a hit in L2 cache, then the response data is sent back to 
the L1 cache (path5). In case of an L2 miss, the request is forwarded to 
the memory controller (path 2). The memory controller schedules this 
request (path3) and the response is sent first to the 
L2 cache (path 4) and then to the L1 cache (path 5). .......... 42
3.3 An instruction window with four outstanding memory requests. .. 43
3.4 The average delays of the off-chip memory accesses issued by one 
of the cores (executing milc in workload-2) ...................... 44
3.5 The latency distribution of the memory accesses issued by one of 
the cores (executing application milc in workload-2). ............ 45
3.6 The idleness of different banks of a memory controller. .......... 46
3.7 A snapshot that shows the states of the three banks controlled by 
one of the memory controllers. ................................. 46
3.8 An example to illustrate how Scheme-1 works. MC1 has received a 
memory request from core-1, and R0 is the corresponding response 
message for that request after it is serviced by MC1. ............ 49
3.9 Latency distributions of the memory accesses (issued by the core 
executing application milc from workload-2). The solid curve shows 
the distribution of the so-far delays at the point right after the 
memory controller, while the dashed curve plots the distribution of 
the round-trip delays of the messages. ........................... 50
3.10 Pipeline stages in baseline and pipeline bypassing. ............... 53
3.11 Speedup values achieved for different workloads executed on a 32-
core system. ......................................................... 57
3.12 (a) The CDFs of the off-chip accesses for the first 8 applications 
in workload-1. The values on the x-axis are the total (end-to-end) 
memory access latencies (in cycles), and the y-axis shows the fraction 
of the total number of off-chip memory accesses. (b) The CDFs 
of the off-chip accesses for the first 8 applications in workload-1, 
when Scheme-1 is employed. (c) The delay distributions of the 
memory accesses for one of the applications (lbm) in workload-1 
before and after Scheme-1 is employed. This distribution change 
results in 8% performance improvement. ..................... 58
3.13 Idleness values of different banks of a memory controller when
Scheme-2 is employed and when no scheme is used. As can be
observed, Scheme-2 reduces idleness in most of the banks. 59
3.14 Average idleness of the memory banks (workload-1) over the exe-
cution. 59
3.15 Speedup values achieved for different workloads executed on a 16-
core system. 60
3.16 Results from our sensitivity experiments (32 cores). 61
3.17 Sensitivity to the number of pipeline stages. 62

4.1 Performance (normalized weighted speedup) of a mix of 16 appli-
cations on a 16-core machine under different distributions of the
same power budget (90W). All bars are normalized to the first bar.
(x,y,z) on the x-axes indicates the specified core, cache and NoC
power budgets (in watts), respectively. 67
4.2 An NoC-based multicore architecture. 68
4.3 High level view of PEPON. 70
4.4 Feedback control loop for regulating the core/NoC power. 76
4.5 Static power versus the number of active (L2) cache ways. 76
4.6 (a-b) Normalized weighted speedup values achieved by different
schemes under certain power budget, and (c-d) Dynamic tracking
of the total power budget. Note that both (c) and (d) plot all
the mixes together. The point here is that, in the low power case,
chip-wide power consumptions of our mixes approach 40W, and
similarly, in the high power case, chip-wide power consumptions
of our mixes approach 90W. 82
4.7 Power budget partitioning and tracking at the first level of PEPON
for Mix-1 (low power mode). 85
4.8 Power budget partitioning and tracking at the first level of PEPON
for Mix-1 (high power mode). 85
4.9 Power budget partitioning at the second level (Mix-1). Note that
each graph plots all 16 core (or cache) components. 85
4.10 Modulation of core frequencies and the number of active cache ways
over execution (Mix-1). Note that, the curves are stacked on top of
each other for better visibility. Note also that, time intervals where
the cumulative number of (cache) ways activated is low correspond
to periods where caches are not allocated much power. 86
4.11 Employing P and PID controllers. 86
4.12 Performance improvements achieved for Mix-1 by varying the de-
fault values of configuration parameters. 87
5.1 High level view and the main components of our target system. . . 94
5.2 Memory energy and the workload performance for different memory
frequencies. The results are normalized to the base case in which
the memory system operates at the maximum frequency (800 Mhz). 96
5.3 The slow downs of individual applications running on our 32-core
system when the memory frequency is set to 720 Mhz. . . . . . . . 96
5.4 Distribution of the off-chip accesses across 4 memory channels in
Figure 5.1 for 6 cores. . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.5 The energy saving and the achieved performance loss reduction by
employing our application clustering. . . . . . . . . . . . . . . . 98
5.6 The slow downs of the individual applications when our clustering
is employed. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
5.7 The high level view of HiPEMM. . . . . . . . . . . . . . . . . . . 99
5.8 The correlation between the average DMPKC and MPKC values
of 32 applications and their performance loss when the memory
frequency is scaled down (each dot represents an application). . . 100
5.9 HiPEMM example. . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
5.10 Memory and system energy consumption when HiPEMM and the
base scheme are employed. . . . . . . . . . . . . . . . . . . . . . . . 107
5.11 (a) Achieved performance and (b) average memory access latency
for each off-chip request (the values are normalized to the case where
all the memory channels operate at the maximum frequency (800
Mhz)). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
5.12 Memory and system EPR values. . . . . . . . . . . . . . . . . . . 108
5.13 Memory controller cluster/frequency assignments over time. . . . 109
5.14 Sensitivity analysis. . . . . . . . . . . . . . . . . . . . . . . . . . . 110
6.1 The measured IPCs for three different applications. For each ap-
plication, bars correspond to different threads. The IPC values
are normalized to the maximum IPC achieved by any thread that
belongs to the same application. . . . . . . . . . . . . . . . . . . . . 114
6.2 The measured IPCs of the threads of three co-runner applications
on an 8 core multicore. D0, D1, D2 and D3 are mgrid’s threads;
U0 and U1 are lu’s threads and G0 and G1 are cg’s threads. Each
bar is normalized to the maximum IPC achieved by any thread of
the same application. . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.3 Different mappings of the threads that belong to three multithreaded
applications running on an eight-core multicore. P0 through P3
denote cores, and D0-3, U0-1 and G0-1 are mgrid’s, lu’s and cg’s
threads, respectively. . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
6.4 High-level view of CCS. The Arbiter component takes the measured IPCs of the cores as input and, at each time interval, determines a “rank value” for each of the cores in the system. Each L2 Manager takes the ranks of the threads running on its cores as input, and adjusts the eviction weights of the running threads.

6.5 Weighted speedup values achieved for Mix-2 using 34 different mappings.

6.6 Weighted speedup values achieved for Mix-5 using 34 different mappings.

6.7 Different multicore configurations tested in our experiments.

6.8 The performance improvement achieved for the applications’ threads of each mix (Experiment-1, 16 cores, 4 L2 caches). For each mix, bars correspond to the six different mappings.

6.9 Dynamic adjustment of the eviction weights of the Mix-2’s threads using Map-2 (each line corresponds to a thread).

6.10 The normalized measured IPCs of LU’s threads during execution.

6.11 The weighted speedup values achieved for L2 count sensitivity analysis (Experiment-2).

6.12 The fair speedup values achieved for L2 count sensitivity analysis (Experiment-2).

6.13 Cache capacity sensitivity analysis (Experiment-3, 1MB/core).

6.14 Cache capacity sensitivity analysis (Experiment-3, 256KB/core).

6.15 The weighted and fair speedup values achieved for core count sensitivity analysis (Experiment-4).

6.16 The weighted speedup values achieved for application count sensitivity analysis (Experiment-5).

6.17 The weighted and fair speedup values achieved for enforcement interval sensitivity analysis (Experiment-6). The control intervals are 100 million cycles.

6.18 The weighted and fair speedup values achieved for enforcement interval sensitivity analysis (Experiment-6). The control intervals are 400 million cycles.

6.19 The weighted speedup values achieved for the multicore with L3 caches. The capacity of the L3 caches is set to be 1MB per core and L3 access latency is five times greater than L2 access latency.

6.20 Achieving QoS targets. The solid lines indicate the specified QoS targets for applications.

6.21 The weighted and fair speedup values achieved with δ sensitivity analysis (Experiment-7) (Mix-5 with six different mappings and four δ values).
List of Tables

2.1 General terms used in control theory and their descriptions in our problem domain................................. 15
2.2 Notation used in this chapter............................................... 17
2.3 Baseline configuration.................................................. 25
2.4 Various mixes (workloads) composed using the SPECOMP applications.............................................. 25
2.5 Model parameters and assessment.................................... 28
3.1 Baseline configuration.................................................. 54
3.2 Workloads used in our 32-core experiments......................... 56
4.1 A sample window (table) of the power budget allocations and corresponding performance numbers (first column)............................................. 71
4.2 \{frequency (GHz), voltage (V)\} levels considered for the cores.......................................................... 75
4.3 \{frequency (GHz), voltage (V)\} levels considered for the NoC........................................................... 75
4.4 Baseline configuration.................................................. 79
4.5 Various mixes of applications considered for our multiprogrammed workloads........................................ 80
4.6 Various experimental schemes with the fixed power budget values.................................................... 81
5.1 Parameters/functions used by HiPEMM.................................. 101
5.2 Baseline configuration.................................................. 104
5.3 Workloads used in our 32-core experiments. The numbers within parentheses show the number of running instances of each application. 106
6.1 An example that involves three applications (X, Y and Z) and 8 cores.................................................... 123
6.2 Baseline configuration.................................................. 125
6.3 Various mixes of applications considered for our multithreaded, multiprogrammed workloads.................... 127
6.4 Our experimental configurations.................................... 127
6.5 Mappings used in 16 cores experiments. \( X_i, Y_i \) and \( Z_i \) are the threads that belong to the applications shown in Table 6.3 in the same order for each mix. ...................................................... 129

6.6 Three mappings used in the experiments with an 8-core system. . . 134

7.1 The routing Decisions. ...................................................... 149

7.2 System configuration parameters. ........................................ 150
 Acknowledgments

First, I would like to express my sincere gratitude to my dissertation advisor, Dr. Mahmut Kandemir for guiding me to do quality research and also for his support, help, suggestions and ideas during my PhD study at Penn State. It has been a great experience and honor for me to work with him over these years. This dissertation would not have been possible without him.

I wish to thank Dr. Chita Das for providing me with valuable advice and also for his help and time in discussing the ideas and reviewing and improving the quality of our research papers.

I thank Dr. Padma Raghavan and Dr. Qian Wang for serving in my dissertation committee.

I would also extend my appreciation for my lab mates and friends, Shekhar Srikantiaiah, Asit Mishra, Emre Kultursay, Hui Zhao and Diana Guttman who collaborated with me in publishing the papers.

Finally, I would like to thank my parents, Esfandiar and Mahin Sharifi and my family. Words cannot express my gratitude for their emotional support, unconditional love and encouragement.
Dedication

To My Parents...
Chapter 1

Introduction

1.1 Meeting End-to-End QoS in Multicores

Multicores are now ubiquitous [1, 2, 3, 4], owing to the benefits they bring over single core architectures including improved performance, lower power consumption and reduced design complexity. Several resources ranging from the cores themselves to multiple levels of on-chip caches and off-chip memory bandwidth are typically shared in a multicore processor. Prudent management of these shared resources for achieving predictable performance and optimizing energy efficiency is critical and thus, has received considerable attention in recent times. Most of this research has focussed around managing either the shared cache [5, 6, 7, 8, 9, 10, 11] or off-chip memory bandwidth [12, 13, 14, 15, 16] in isolation.

In general, almost all these approaches attempt to guarantee a certain level of quality-of-service (QoS) like weighted IPC, harmonic speedup, etc by managing a single shared resource or at most a couple of interacting resources. There are at least three fundamental shortcomings with such approaches: (i) Considering that system performance is heavily influenced by complex interactions among multiple resources [17, 18], attempting to optimize performance/guarantee QoS by managing a single resource is not only less effective, but may also be impossible; (ii) In most existing schemes, there is no feedback among mechanisms, trying to provide QoS by managing different resources, and this leads to an anarchy in resource management; and (iii) Working with low-level resources like cache or memory-bandwidth restricts the system-performance metrics that can be controlled to only
low-level metrics like IPC, which are not easily comprehended by system administrators or application programmers.

On the other hand, providing end-to-end QoS in future multicore is essential for supporting wide-spread adoption of these new architectures in virtualized servers and cloud computing systems. An initial step towards such an end-to-end QoS support in multicore is to ensure that at least the major computational and memory resources on-chip are managed efficiently in a coordinated fashion.

In Chapter 2, we propose METE, a platform for end-to-end on-chip resource management in multicore processors. The proposed resource management scheme attempts to address all the three major constraints of existing techniques by (i) providing a multi-level resource provisioning mechanism for end-to-end QoS, (ii) developing a control theoretic model for accurately tracking the system state, and (iii) by demonstrating the applicability of the model to system level parameters. While a few recent works [19, 20] have studied the advantages of using feedback control theory for resource management in multicore processors, to our knowledge, no prior study has taken a holistic multi-level control theory approach as proposed there.

1.2 Addressing End-to-End Memory Access Latency in Multicores

From an architectural side, in parallel to increasing core counts, network-on-chip (NoC) is becoming one of the critical components which determine the overall performance, energy consumption and reliability of emerging multicore systems [39, 40, 41]. While NoC helps to improve communication scalability of data, it also contributes to memory access latencies. Consider as an example a data read access in an NoC-based multicore. Such an accesses first checks L1 cache, and if it misses there, accesses L2 cache by traversing the NoC (assuming an S-NUCA-based [42] cache space management). If it misses in L2, it takes another trip over the NoC to reach the target memory controller determined by the address of the requested data. Depending on the current status of the queue in the target memory bank, it waits in the queue for some amount of time before reaching the DRAM. After
finally reading the data from the DRAM, the same set of components are visited in the reverse order until a copy of the data is brought into the core. Clearly, each of the components in this round-trip access path (L1, L2, network, bank queue, and memory access) contributes to the end-to-end latency of this data access.

From an application side, interferences across simultaneously executing applications on various shared resources (e.g., NoC, memory controllers) can lead to high variances across the latencies of the off-chip accesses made by an application [43, 44, 45]. Specifically, while one request can be lucky to retrieve its data very quickly, another request of the same application can get delayed in the network and/or the memory controller queue, suffering a much larger round-trip latency. This variance can in turn lead to degradation in overall system performance as requests that experience much higher latencies than the average can block the progress of the application. To achieve high performance, it is crucial to reduce the number of memory accesses that observe very high latencies, but this should be done with care as improving the latency of an access can worsen the latency of another as a result of resource sharing. Therefore, from an overall system performance point of view, instead of each application aggressively trying to minimize the latencies of all its memory accesses, it is better to have each application balance the latencies of its memory accesses.

Motivated by these observations, in Chapter 3, we propose two network prioritization schemes that can cooperatively reduce end-to-end memory access latencies in NoC-based multicores. One of these schemes targets memory response messages (i.e., messages from the memory controller), whereas the other targets memory request messages (i.e., messages from the core side).

### 1.3 Power Budgeting in Multicores

Power budgeting and performance optimization under a given power budget in multicore systems have received considerable attention recently [71, 72, 73, 74]. A majority of these solutions focus on power budgeting at the processor level and the only knob they use to control the power of a multicore processor is the voltage/frequency level of the cores. While modulating the voltage/frequency level of the cores in a multicore processor has the potential to control power consumption
of the processor, it provides no direct control on the efficiency (performance per Watt) of the processor. This is because the performance of an application is not controlled by the core frequency alone (although it has a vital influence), but is also influenced by other on-chip components; specifically, the last-level-cache and the network-on-chip (NoC). In fact, the significance of the power consumption of the non-core components is increasing and projected to be quickly on par with the power consumption of cores themselves, if not greater [75, 76, 77]. Our experimental studies also reinforce these observations made in the literature. For example, our experiments on a 16-core machine with 8MB last-level cache and a 4 × 4 mesh-based NoC show that about 33% of the power is consumed by the non-core components on average (15% and 18% by the last-level cache banks and the NoC, respectively).

A recent network based multicore from Intel [76] reports 31% and 79% non-core power contribution in full power and low power operating modes, respectively. Despite the fact that the power consumptions of caches and NoC are on par with that of the cores, to the best of our knowledge, there has been no prior work on distributing the power budget among all these resources to maximize performance, while maintaining the same overall power budget of the system as a whole. We believe significant performance improvements can be achieved by distributing the overall power intelligently among the resources based on the workload’s sensitivity to core/NoC frequency and cache allocation.

In Chapter 4, we propose a power budget distribution mechanism (called PE-PON) that distributes the chip-wide power budget among various types of critical resources like the cores, caches, and NoC to maximize performance, while respecting the allocated budget in the corresponding resources, thereby respecting the overall power budget.

### 1.4 Saving Energy in Multicore Memories

As datacenters play an increasingly important role in different application domains, more energy efficient servers are desirable. The increasing amount of energy used by datacenters affects not only the ownership costs, but also puts strain on local power grids and has long-term environmental implications. The servers themselves
consume more than 80% of datacenter energy [101] and within servers the memory system has become a major energy consuming component that uses up to 40% of the total power, not including the power used by on-chip memory controllers [102, 103]. However, memory accesses are also a performance bottleneck for many applications, so any strategy that attempts to reduce memory power/energy must be careful not to hurt performance.

Dynamically changing the operating voltage and frequency (DVFS) has been employed as an efficient scheme to reduce the power and energy consumption of the main memory system in modern multicores [104, 105, 106, 107, 102]. Here, similar to DVFS in the processor domain, the main idea is to reduce the memory power by lowering the operating frequency. However, a slower memory system can impose performance overhead and increase the execution times of the running applications due to increased off-chip access latencies. Performance loss of applications can cause two main problems: first, some applications cannot tolerate the loss in performance; and second, although the power consumption of the memory system decreases by lowering the frequency, it may not be beneficial in terms of the total system energy, since the application execution times increase. Deng et al. [102] proposed a dynamic scheme that adjusts the frequency of the memory system with the goal of saving energy, while considering the imposed performance overhead. In their work, at each time epoch, they compute the performance slack (the tolerable performance loss) for each running application. Then, the memory frequency is reduced based on the smallest slack among all the running applications. However, several issues can limit the use of this approach: (1) Computing the slack for each application requires a reference execution time which may not be easily available. (2) The mathematical model proposed in this work to capture the impact of the frequency scaling on the application performance of in-order cores can be quite complex and highly inaccurate for out-of-order processors and multicores with a large number of cores. (3) Once the slacks are computed for running applications, the smallest slack among all applications should be considered in order to find an appropriate memory frequency. This is because changing the frequencies of the memory channels affects all the running applications, which have different sensitivities to the off-chip access latency. This can be very restrictive since any sensitive application has the potential to become a bottleneck in
reducing the memory frequency.

To overcome these limitations, in Chapter 5 we propose HiPEMM, a high-performance DVFS mechanism for reducing memory system power by scaling individual memory channel frequencies to match application demand.

1.5 Shared Cache Capacity Management in Multicores

Most recent multicore processors [1, 3, 4] share caches among the cores. Memory requests made by different cores interfere with each other in different levels of the cache, causing the eviction of each core’s data, which otherwise would have good hit rates. As we move towards many-core systems, interference in shared cache continues to increase, making shared cache management an important issue in increasing overall system performance. Further, to effectively make use of these many-core systems, application programmers and compiler writers alike are trying their best to exploit the increased potential of the higher number of cores by having more multithreaded workloads. Given the limited ability of current compilers to extract a large fraction of the available parallelism, we envision a typical workload on future multicore processors to consist of a dynamic and diverse range of applications that exhibit varying degrees of thread-level parallelism. This brings managing shared caches for multiprogrammed, multithreaded workloads to the limelight.

Our goal in Chapter 6 is to design a configurable shared cache management scheme that provides high system throughput without requiring significant coordination between multiple sockets.

1.6 Targeting NoC in Emerging Multicores

Network-on-Chips (NoCs) [60, 149, 150] offer several important advantages over alternate on-chip communication schemes including providing higher system performance by reducing global wire latencies, higher scalability and lower power consumption. Recently, process variation (PV) resulting from manufacturing im-
perfections [151] has become a pressing issue in on-chip circuits, as technology scales to smaller feature sizes [151]. Various manufacturing related factors may result in process variation such as random dopant fluctuations, dose, focus and overlay variations [151].

Latency and power variations are two high-level impacts of process variation. For instance, in CMPs, process variation can be translated into core-to-core variations [152]. In these systems, processing cores constitute a heterogeneous CMP architecture, instead of a homogenous one (as originally designed), in which each core has its (possibly different) frequency and power characteristics. One strategy adopted to face this phenomena is to compensate for this heterogeneity by employing circuit level mechanisms such as adaptive voltage scaling (AVS) and adaptive body biasing (ABB) [152] to create a homogenous system or slow down the processors to the slowest one. Unfortunately, these methods have negative impacts on leakage power and system performance [152]. An alternate approach is to let each core or hardware module operate at its own frequency. Then, this heterogeneity can be exploited by high level decisions such as application scheduling and mapping in the context of CMPs. [153, 154, 155, 156]. Teodoresco and Torrellas [154] proposed algorithms to schedule applications in a CMP considering the potential variations across the cores. To determine the impacts of variations on the frequency of the processors they employed critical path distribution based models [157].

In recent years, researchers have also investigated process variations in NoCs from different perspectives. In [151], several architectural solutions have been proposed to mitigate the impacts of process variation in the pipeline stages of the routers in an NoC architecture. Even though their proposed router design can reduce the impact of process variations, it will not able to entirely remove them. Several other prior works have studied the modeling of the impacts of process variation in NoCs [158, 159]. For instance, Li et al [160] investigated how process variations affect leakage power in NoCs. However, very few strategies have been proposed to cope with NoC performance degradation due to process variations during run time. Markovsky et al [161] have showed that employing adaptive routing can increase the expected saturation bandwidth in a heterogenous NoC architecture. Employing adaptive routing has the capability to balance the
network traffic when the network is highly loaded. However, in their work, they have not considered the performance characteristic of each individual router, and consequently, the packets of a communication may not traverse the best route in terms of the time it takes to reach their destination.

In Chapter 7, focusing on a heterogenous NoC in which each router has (potentially) a different processing delay, we propose a process variation-aware source routing scheme to enhance the performance of the communications in the NoC based system.
METE: Meeting End-to-End QoS in Multicores Through System-Wide Resource Management

2.1 Introduction

In this chapter, we propose METE, a platform for end-to-end on-chip resource management in multicore processors. The proposed resource management scheme attempts to address all the three major constraints of existing techniques by (i) providing a multi-level resource provisioning mechanism for end-to-end QoS, (ii) developing a control theoretic model for accurately tracking the system state, and (iii) by demonstrating the applicability of the model to system level parameters. While a few recent works [19, 20] have studied the advantages of using feedback control theory for resource management in multicore processors, to our knowledge, no prior study has taken a holistic multi-level control theory approach as proposed here.

Figure 2.1 shows the high-level view of the three resources managed by a specific implementation of METE (cores, on-chip shared caches and off-chip bandwidth) in an integrated manner among two applications. The main goal behind this management is to ensure that any performance target is tracked by provisioning resources in an end-to-end manner. Assuming that each application specifies a (potentially
different) performance target, the main objective of METE is to dynamically provision sufficient on-chip resources to applications in order to achieve the specified targets, if it is possible to do so. Apart from low-level metrics like IPC (instructions per cycle) that have been used in the past, METE also accepts high-level (e.g., application specific) performance metrics like transactions per second for database applications. More importantly, the proposed system is flexible enough to handle different metrics for different applications at the same time.

METE employs a feedback control based system designed as a Single-Input, Multiple-Output (SIMO) controller with an Auto-Regressive-Moving-Average (ARMA) model to capture the dynamic behaviors of different applications. Control theory

Figure 2.1. Three types of shared resources provisioned to two co-runner applications.

(a) Cache ways variation (Allocated number of cores is 4, Allocated bandwidth is 50%).
(b) Memory bandwidth variation (Allocated number of cores is 4, Allocated cache ways is 16).
(c) Number of cores variation (Allocated cache ways is 16, Allocated bandwidth is 50%).

Figure 2.2. The impacts of resource allocation on the performance of two running applications.
[21, 22] is a powerful tool that offers several unique advantages over alternate schemes:

- Feedback control theory provides a robust strategy to track specified objectives over time. It can achieve this by modulating resource allocations in a coordinated fashion.
- A system equipped with a feedback controller can respond quickly to variations in the dynamic behaviors of running applications.
- A feedback controller can be designed to control multiple high-level metrics of interest at the same time under various constraints.
- Using control theory enables rejecting unexpected disturbances in the controlled system. For instance, in the context of multicores, a sudden change in the demands for a shared resource can be interpreted as an external disturbance in the system.

We evaluate METE on 8 and 16 core systems using a detailed full system simulator and workloads formed from the SPECOMP and SPECJBB multithreaded applications. The collected results indicate that our proposed scheme is able to provision shared resources among co-runner applications dynamically over the course of execution, to provide end-to-end QoS and satisfy specified high-level performance targets.

The remainder of this chapter is structured as follows. The next section discusses background on feedback control theory, and motivates our solution. Section 2.3 gives the mechanisms employed to partition each type of resource we target in this work across applications. Section 2.4 presents our proposed METE platform that enables QoS-aware multi-resource management. Our experimental evaluation is presented in Section 2.5. Section 6.4 discusses the related studies and finally, we conclude the chapter in Section 7.5.

2.2 Motivation and Background

2.2.1 Motivation

The allocated number of cores, amount of on-chip shared cache space and off-chip memory bandwidth are three major parameters that affect the performance of an
application running on a multicore machine. However, the degrees to which each of these parameters influences the performance are not the same, both within an application’s execution (i.e., across its different phases) as well as across different applications. To illustrate this point, we performed a set of experiments that evaluates the impact of different shared resources on the performance of applications. Two applications, applu and swim from the SPECOMP benchmark suite [23], are used in these experiments. Unless otherwise mentioned, each of these applications is assigned 4 cores, 4 MB, 16 way associative shared on-chip cache space, and 6.4 Gb/s bandwidth. Then, each of the resource allocations, one at a time, is changed to study the impact the resource has on each application’s performance. Figures 6.2(a), (b) and (c) plot the performance (measured in IPC) as the amount of resources (cache ways, percentage bandwidth, and cores) allocated to the applications varies. In all experiments, each application has 8 threads. The detailed simulation parameters and system configuration used in this work are given later in Section 2.5.1.

Several observations can be made from these plots. First, the overall impact of varying the amount of each type of resource is significantly different from the others. For example, the number of allocated processing cores has much larger impact on the performance as compared to the amount of off-chip bandwidth allocated to the running applications. Therefore, while it is important to consider all resources in the system to provide end-to-end QoS, it is also crucial to consider the differences in their impacts on application performance. Second, for a given change in a resource, different applications react differently, based on the operating point. For example, for a bandwidth allocation change from 5% to 10%, swim receives a bigger boost in performance than applu. That is, the slope of the performance vs. bandwidth allocation curve for swim is greater than that for applu in the 5-10% operating region. However, the corresponding slopes are similar in the 50-100% operating range. Therefore, it is important to consider the operating point of each resource for each application. Third, it is possible to track the same target performance for the same application by allocating different combinations of resources. For example, applu is able to achieve an IPC of 1.2 by using either

---

Note that, in a more bandwidth constrained system the significance of bandwidth may be reversed.
4 cache ways and 50% of peak bandwidth (Figure 6.2(a)), or 10% of the peak bandwidth and 16 cache ways (Figure 6.2(b)). On the other hand, when some of the resources are constrained, we may not have flexibility in resource allocations to track a specified target performance. For example, once swim is constrained to an allocation of 4 cores and an allocation of 50% bandwidth, it cannot achieve an IPC of 1.7 even with an allocation of 32 cache ways, while the same can be achieved by allocating 4 cores, 16 cache ways and 100% bandwidth. Therefore, it is important to consider the allocations across all resources in unison (end-to-end), such that all performance targets can be met.

A recent work [19] elaborated on the advantages of using formal feedback control and modeling application performance to achieve performance QoS by partitioning only the shared cache (assuming that the other resource allocations do not change). The customized oscillation resistant controller proposed in that work is essentially a Single-Input, Single-Output feedback controller that can track the performance of an application by performing shared cache partitioning. As a preliminary study, we extended this design based on the same principles to design three controllers, one for each type of resource (core, cache and bandwidth). Each of the three controllers takes the same performance QoS target as input (corresponding to an IPC of 1.25) and seeks to satisfy this goal in isolation. The observed performance of one of our applications (applu) with such a system is shown in Figure 2.3. We can see that, the specified QoS target is often violated by this system. One of the major reasons for this behavior is the lack of coordination between the three controllers. Specifically, each of the three controllers decides the resource allocations in their layer without considering the impact that the other may have on the performance. This can often lead to conflicting decisions in different resource allocations [17, 18]. For example, at time $t = 5$, 5 cores, 16 cache ways and 80% of the off-chip memory bandwidth are allocated to the running application. Since the measured performance is higher than the target, the amount of allocated cache space and memory bandwidth are reduced to 7 and 60%, respectively. As a result, the measured IPC decreases to 1.2, which is lower than the performance goal. This simple experiment clearly motivates the need to have a coordinated end-to-end feedback controller that takes the specified QoS target as input, but simultaneously controls all the resources based on modeling application behavior.
Figure 2.3. Behavior of applu in a system consisting of three controllers (one for each resource). Oscillation occurs, since at time $t$ the cache layer controller increases/decreases the allocated cache space to meet the performance target unaware of the changes made by the off-chip bandwidth controller to the memory bandwidth allocation.

### 2.2.2 Background

In this work, we employ Single-Input, Multiple-Output (SIMO) controller. Figure 2.4 illustrates a canonical feedback control loop with a SIMO controller. As an example for SIMO controller, consider a water pool (plant) that has to be maintained at a constant temperature by letting in both hot and cold water flows run into it (and correspondingly let equal volume of water out of the pool). Suppose that the rates of the flows can be controlled by a controller and the controller’s role is to adjust these rates in an automated fashion to keep the pool temperature at the desired value. The desired temperature is called Reference Input in the control theory terminology. Since in this case, the controller takes a desired pool temperature as a single input and controls it using multiple outputs (specifically two: hot and cold water), the controller is a SIMO controller. The controller functions by comparing the reference input to the current water temperature (System Output) and based on the obtained Error value, the rates of the hot and cold flows are modulated. The Transducer converts the System Output (pool temperature) to the same type as the Reference Input if they have different types and cannot be compared. This component aids in implementing a flexible system with different high-level target specifications by converting the system output into a comparable metric to the target specification. We will discuss the implementation of a transducer component in Section 2.5.5.

In METE, a separate SIMO controller is assigned for each application. At the end of each time interval (sampling period, the controller increases/decreases the
Table 2.1. General terms used in control theory and their descriptions in our problem domain.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description in Our Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Input</td>
<td>Desired IPC value for an application</td>
</tr>
<tr>
<td>Control Input</td>
<td>Resource allocation</td>
</tr>
<tr>
<td>System</td>
<td>multicore with running applications</td>
</tr>
<tr>
<td>System Output</td>
<td>Measured IPC value of an application</td>
</tr>
<tr>
<td>Controller</td>
<td>Application Controller</td>
</tr>
<tr>
<td>System Model</td>
<td>the employed ARMA model</td>
</tr>
</tbody>
</table>

control inputs (resource allocation), taking into account the variation observed in the measured IPC value (Observable System Output) over the last time interval. Note that the controller requires knowledge on the reaction of the application to the modulations in the resource allocations. A system model in control theory tries to capture this knowledge. Table 2.1 summarizes the basic terms used in formal control theory and the corresponding descriptions in our problem domain. We want to emphasize that our scheme can work with any performance QoS. In most of our discussion however, we use IPC as our target metric.

If the values of the model parameters do not change over time, we refer to this type of model as static. In adaptive feedback control systems on the other hand, the system model is updated dynamically. In this work, we employ the latter type since behaviors of applications do not remain constant during the course of execution and each usually has multiple execution phases. Specifically, in METE, we employ an Auto-Regressive Moving Average (ARMA) [22] model. The ARMA model can approximate the behavior of a system with multiple inputs/outputs in a linear form and also can be updated dynamically, suitable for adaptive control system designs. We also employ a global controller (manager), called the Resource Broker, to handle cases where (i) resources requested by applications exceed available capacities, and (ii) after satisfying all requests, there are still idle resources. In the following sub-sections, we study our proposed control design for end-to-end QoS management in
multicores.

2.3 Implementation Issues

Figure 2.1 gives the high-level view of how the shared resources are partitioned between two applications. We envision implementing METE in the operating system, with dynamic feedback from hardware based counters about the different resource usages. The cores need to be partitioned in the OS-level since the OS can manipulate the set of cores used to run an application dynamically over the course of execution. The cache and bandwidth partitioning can also be handled by the OS with hardware support from the shared resources. In the following discussion, we explain the mechanisms METE employs to partition each type of resource across applications.

**Core Partitioning:** METE uses the `psrset` utility in Solaris to create a processor set containing one or more cores and run a particular application on it. Cores can be added or removed from that processor set over execution. Consequently, the OS provides a mechanism to adjust the number of cores allocated to each running application dynamically.

**Cache Partitioning:** In METE, we adopted the OS-level mechanism proposed in [24] to partition the shared cache space across co-runner applications or threads. In [24], the hardware part of the proposed scheme guarantees that the quotas specified by the OS are enforced in shared caches. In this scheme, an m-bit tag is associated with each cache block indicating which core that block belongs to. Also, each memory request contains an identifier indicating its cache block access domain.

**Off-Chip Bandwidth Partitioning:** In METE, the available memory bandwidth is partitioned across co-runner applications based on the priority of each sharer. This is similar to the fair queuing systems proposed in [16, 25]. For example, as can be seen in Figure 2.1, the memory controller serves the requests from the two applications in such a way that each of them receives its quota of the off-chip memory bandwidth based on its relative weight specified by the OS. To implement this scheme, a service time (which is inversely related to its weight) is computed for each application request flow. The application with smaller service
time (as compared to the memory controller’s virtual clock) is served at each time.

After explaining these actuators (control knobs), we next discuss the details of our control architecture.

## 2.4 Control Architecture

### 2.4.1 High Level View

Figure 2.5 illustrates the high level view of METE. Each Application Controller shown in Figure 2.5 is implemented in software. Assuming that each running application is assigned a specified IPC target (desired performance goal, \( r_{ei} \)), the application controller of application \( a_i \) determines the amount of resources, \( u_{ik} \), that need to be allocated for that application in order to satisfy the specified performance goal (\( IPC_{ik} \)) at the k-th execution epoch (Table 2.2 gives the notation we use in this chapter).

As mentioned earlier, in our current implementation of METE, three types of resources are targeted and can be partitioned among co-runner applications. These resources include processing cores, shared L2 cache space, and off-chip memory.

<table>
<thead>
<tr>
<th>( A )</th>
<th>set of applications running on the multicore</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_i )</td>
<td>application number i</td>
</tr>
<tr>
<td>( r_{ei} )</td>
<td>the desired IPC of ( a_i )</td>
</tr>
<tr>
<td>( IPC_{i}(k) )</td>
<td>the actual IPC of ( a_i ) at the k-th time interval</td>
</tr>
<tr>
<td>( \hat{IPC}_{i}(k) )</td>
<td>the predicted IPC by the employed model of ( a_i ) at the k-th time interval</td>
</tr>
<tr>
<td>( c_{ik} )</td>
<td>the number of cores requested by the controller associated with ( a_i ) at the k-th time interval</td>
</tr>
<tr>
<td>( w_{ik} )</td>
<td>the number of cache-ways from the shared L2 cache requested by the controller associated with ( a_i ) at the k-th time interval</td>
</tr>
<tr>
<td>( b_{ik} )</td>
<td>the memory access bandwidth requested by the controller associated with ( a_i ) at the k-th time interval</td>
</tr>
<tr>
<td>( u_{i}(k) )</td>
<td>vector of ( \begin{pmatrix} c_{ik} \ w_{ik} \ b_{ik} \end{pmatrix} )</td>
</tr>
<tr>
<td>( \hat{c}_{ik} )</td>
<td>the number of cores allocated to ( a_i ) by the Resource Broker at the k-th time interval</td>
</tr>
<tr>
<td>( \hat{w}_{ik} )</td>
<td>the number of cache-ways from the shared L2 cache allocated to ( a_i ) by the Resource Broker at the k-th time interval</td>
</tr>
<tr>
<td>( \hat{b}_{ik} )</td>
<td>the memory access bandwidth reserved for ( a_i ) by the Resource Broker at the k-th time interval</td>
</tr>
<tr>
<td>( \Delta u_{i}(k) )</td>
<td>( u_{i}(k) - u_{i}(k-1) )</td>
</tr>
<tr>
<td>( C )</td>
<td>the number of cores of the multicore</td>
</tr>
<tr>
<td>( W )</td>
<td>the total number of cache ways (associativity) of the shared L2 cache</td>
</tr>
<tr>
<td>( B )</td>
<td>the maximum available off chip memory access bandwidth</td>
</tr>
</tbody>
</table>

### Table 2.2. Notation used in this chapter.
bandwidth. The amount of resources in each type is limited, and consequently, at each epoch, if the total amount of resources requested by applications is less than or equal to available total, a successful resource allocation can be performed. Otherwise (if the three resource constraints shown below are not satisfied), a higher level module (called Resource Broker in this work) intervenes to modify the amount of requested resources determined by the application controllers $u_{ik}$ and make the final resource partitioning decision $\hat{u}_{ik}$. Our resource constraints can be expressed as follows:

$$\sum_{a \in A} \hat{c}_{ik} \leq C, \quad \sum_{a \in A} \hat{w}_{ik} \leq W, \quad \text{and} \quad \sum_{a \in A} \hat{m}_{ik} \leq M. \quad (2.1)$$

The above constraints ensure that the sum of the allocated cores, cache ways and memory bandwidth cannot exceed the total available resources. Below, we first study our employed model and the feedback controller associated with it, and then present the details of the resource broker module.

2.4.2 System Model

As mentioned earlier, a feedback controller needs to know the impact of variations in control parameters on the measured system outputs. In other words, a system model is a mathematical function $f$ that gives the system output for every fea-
sible control parameters (i.e., \( y(k) = f[u(k)] \)), where \( y \) and \( u \) are the output and input of the system, respectively). One way of obtaining the system model is to determine function \( f \) directly by mathematical analysis of how the system works. Unfortunately, this approach cannot be applied to most actual computing systems due to the complexity of the desired modeling function. Instead, a more practical approach would be estimating the system behavior through analysis of the sample measured data. In this method, the model is expressed as a parametric function of independent variables with a finite number of parameters. In this case, system model identification is the process of determining these parameters. Inputs with different values are fed to the system and the corresponding measured outputs are collected. Algorithms such as least square [22] can be employed in this step to determine the values of the model parameters.

Even though the behavior of most real systems is not linear, linear approximation can be used (and has been successfully used in prior works and actual control based systems) as a method to estimate non-linear characteristics of actual systems. In METE, we employ the ARMA model [26] to capture the effect of the control parameters \((u_{ik})\) on our system output \((IPC_{ik})\). The ARMA model is a linear recursive equation that can be updated dynamically and is very suitable for adaptive control system designs. The equation below gives the mathematical representation of our employed model for each application:

\[
\hat{IPC}_{i}(k+1) = a_{ik} \times IPC_{i}(k) + b_{ik}^{T} \times \Delta \hat{u}_{i}(k),
\]

(2.2)

where \(a_{ik}\) and \(b_{ik}^{T}\) are the parameters of the model that are dynamically determined for each application. Note that \(IPC_{i}(k)\) and \(\Delta \hat{u}_{i}(k)\) correspond to the actual IPC of application \(a_{i}\) and the variations in the resource allocations at the k-th time interval, respectively. In this equation, \(b_{ik}^{T}\) is \(\left( c \quad w \quad m \right)\), where the values of \(c\), \(w\) and \(m\) capture the influence of the variations in the number of allocated cores, the number of cache ways and the reserved memory bandwidth on the application IPC value. As we later show in the experimental results section, the value of \(c\) for different applications is significantly larger than the value of \(w\), meaning that, if one core is added to the set of cores allocated to an application, the application’s IPC will improve much better as compared to the case in which one extra cache way is added to the previously reserved cache ways for that application.
Figure 2.6. Input, output and state in a state space model.

Figure 2.7. Detailed view of an application controller.

The values of our model parameters are determined in this work dynamically using the recursive least square algorithm [22]. In this algorithm, at each time interval, the model is updated based on the new operating point that has been obtained (i.e., by considering the performance impact of the last resource allocation). To evaluate the obtained model, the IPCs predicted by the model can be compared against the actual ones, while the inputs are taking different values. We present the results from our model evaluation in Section 2.5.

2.4.3 Application Controller

The State Space approach is a compact and convenient method to model, analyze and design a wide range of systems [21], especially systems with time-varying characteristics and multiple inputs/outputs. To use this approach, first, the system behavior has to be represented using a State Space Model. The following equations characterize the general form of a state space model (as illustrated in Figure 2.6):

\[
x(k+1) = Ax(k) + Bu(k) \tag{2.3}
\]

\[
y(k) = Cx(k), \tag{2.4}
\]
where \( \mathbf{x} \) is the state vector, \( \mathbf{u} \) and \( \mathbf{y} \) are, respectively, the input and the output vectors of the system, and \( \mathbf{A} \), \( \mathbf{B} \) and \( \mathbf{C} \) are the model parameters. The state vector \( \mathbf{x} \) contains the state variables that reflect the current state of the system. Equation (2.3) is solved to predict the next state of the system based on the current state and input vector \( \mathbf{u} \). The input vector contains the control \( \text{input} \) parameters. Equation (2.4) is called the output equation and is used to determine the system outputs.

It should be observed that our system model in Equation (2.2) can be expressed as a state-space model, described by Equations (2.3) and (2.4), where we have:

\[
\mathbf{A} = (a_{ik}), \quad \mathbf{B} = \mathbf{b}_{ik}^T, \quad \mathbf{C} = (1), \quad \mathbf{y} = IPC_i.
\] (2.5)

The next step in the state space approach is to design an appropriate controller based on the obtained system model. Figure 2.7 illustrates the internal structure of an application controller to achieve this goal. The feedback control is composed of two main components: controller gains (\( \mathbf{K} \)) and pre-compensator (\( \mathbf{N} \)), as shown in Figure 2.7. Our goal is to determine the values of \( \mathbf{K} \) and \( \mathbf{N} \) in such a way that the stability of the applications around the targets is ensured. Note that, \( \mathbf{K} \) and \( \mathbf{N} \) are two vectors with three elements. pole placement and linear quadric regulation [27] are two algorithms that can be employed to determine the values of \( \mathbf{K} \) and \( \mathbf{N} \). In the pole placement strategy, which is the method employed in this work, first, the values of \( \mathbf{K} \) are determined in such a way that the poles of the system ensure the stability of the system. After that, the contents of \( \mathbf{N} \) are obtained using the method outlined in [26].

The final resource demands (\( \mathbf{u}_i(k) \)) of application \( i \) (\( a_i \)) are determined by adding the amount of decreasing/increasing values of the control parameters (\( \Delta \mathbf{u}_i(k) \)) and the application resource allocation at the (k-1)th time interval (\( \hat{\mathbf{u}}_i(k - 1) \)), as given in Equation (2.6):

\[
\mathbf{u}_i(k) = N \times ref_i - k \times IPC_i(k - 1) + \hat{\mathbf{u}}_i(k - 1).
\] (2.6)
2.4.4 Stability Guarantees

Stability is one of the most important properties of a feedback control based system [26]. If a system equipped with a feedback controller is stable, the measured output of the system converges to the desired target over time. For instance, the system with the output shown in Figure 2.3 is not stable, since the controller is not able to adjust the values of the system inputs in such a way that the system output converges to the desired target over time.

Different properties of a control system can be analyzed easily when the system is described in frequency domain (z-domain). Considering Figure 2.6 and assuming $U(z)$ and $Y(z)$ are the z-domain representations of $u(k)$ and $y(k)$ respectively, the value of $u(k)$ at time $k = k_0$ is the coefficient of $z^{-k_0}$ in $U(z)$. In other words, the values of $u(k)$ for different $k$s are encoded as the coefficients of $z$ terms in $U(z)$. Further, the transfer function of the system is defined as $G(z) = \frac{Y(z)}{U(z)}$ and indicates how an input $U(z)$ is transformed into the output $Y(z)$. In our multicore system, the transfer function can be determined by taking z-transform of all terms in Equation 2.2 which describes the behavior of our system.

The Stability Theorem in the formal control theory states that a system represented by a transfer function $G(z)$ is stable if and only if the poles (roots of the dominator polynomial) of $G(z)$ are within the unit circle in the complex coordinate plane [26]. Therefore, in METE, to ensure the stability of the system, we first need to find the poles of each application controller and then determine the parameters of the controller in such a way that the poles are placed within the unit circle. As shown in [26], if a system is described by a state space model as in Equation (2.3), the poles of the system can be obtained solving the following equation in terms of $z$:

$$\text{det}[zI - (A - BK)] = 0,$$

where $A$ and $B$ are the model parameters in Equation (2.3), and $K$ is the coefficient vector in the feedback path (see Figure 2.7). By replacing the parameters in Equation (2.7) with the values given in Equation (2.5), the pole of each application controller’s transfer function would be:

$$z = a_{ik} - c \times k_1 - w \times k_2 - m \times k_3,$$
assuming that, in Equation (2.7), $B$ is $(c \ w \ m)$ and $K$ is a vector with three elements, $k_1$, $k_2$ and $k_3$. Consequently, to ensure stability of our system, the $K$ values are determined such that $|z| < 1$ in Equation (2.8).

### 2.4.5 The Resource Broker

So far, we have discussed how an application controller determines the amount of different types of resources (processing cores, L2 cache ways, and off chip memory bandwidth) an application needs to satisfy its specified performance QoS ($ref_i$). However, since each of our application controllers operates independently, there can be cases where (i) the available resources are not sufficient to meet the requirements of all of the running applications or (ii) the available resources exceed the requirements of the entire workload. In METE, these cases are handled by the resource broker component.

**Lack of Resources:** Note that, resource contention may occur in any of the resource types (i.e, when one or more than one of the constraints in Expression (2.1) given earlier are not satisfied). In this case, the resource broker is responsible for performing a *best effort* allocation by considering the relative resource demands of the co-runner applications. As an example, suppose that $Q_1$ and $Q_2$ are the performance targets of two co-runner applications ($a_1$ and $a_2$), and the available resources are not sufficient to satisfy both of the specified targets. Assume further that the amount of resources required to satisfy $Q_1$ (determined by the application controller) is much larger than the resource demands of the other application to achieve $Q_2$. In this situation, it would not be a fair policy to penalize both applications evenly in an attempt to compensate for the lack of resources, since such policy may degrade the performance of the application with lower resource requirement significantly.

Without loss of generality, let us focus now on core allocation. Assuming that the total demands of applications is greater than the total available number of cores, one can observe that $\delta$ cores have to be spilled, where $\delta = \sum_{a \in A} c_{ik} - C$. The approach that we adopt in this work (i.e. our default policy) is to distribute $\delta$ among the applications in proportion to their demands. The goal here is to distribute the penalty across applications in a fair manner. Consequently, in this
case, the core demands \( (c_{ik}) \) are modified as:

\[
c_{ik} = \text{floor} \left( c_{ik} \left( 1 - \frac{\delta}{\sum_{a_i \in A} c_{ik}} \right) \right).
\]  

(2.9)

A similar strategy can be adopted when the contention occurs in other types of resources as well (shared L2 cache ways and off chip memory bandwidth).

**Excess resources:** If sum of the resources required by co-runner applications is less than the available amount, one has different options. An energy-aware option would be turning off the excess resources (or placing them into a low-power operating mode) if the underlying hardware supports that. A performance-centric option (our default policy) on the other hand would proceed as follows. To extract additional performance from the available resources, we may be able to use the application priorities assigned to applications by system administrators. Assuming that each application in a workload has been assigned a priority/weight, \( (w_i) \) indicating its relative importance, \( \delta \) excess resources can then be distributed across applications based on Expression (2.10) given below:

\[
c_{ik} = \left[ c_{ik} + \frac{\delta \times w_i}{\sum_{a_i \in A} w_i} \right], \quad \text{if } c_{ik} < 1 \rightarrow c_{ik} = 1.
\]  

(2.10)

Note that, these priorities can be determined based on the values of vector \( b_{ik} \) in Equation (2.2). As mentioned earlier, these values capture the influence of the variations in the number of allocated cores, the number of cache ways and the reserved memory bandwidth on the IPC value of application \( i \). Consequently, if a change in the allocation of the contented resource has a larger impact on the performance of application \( i \), this application would receive more (excess) resources than the other applications to maximize performance benefits.

**2.5 Experiments**

In this section, we present a detailed experimental evaluation of METE.

**2.5.1 Benchmarks and Setup**

We used the SPECOMP benchmark suite [23] to evaluate our control scheme. In addition, we also performed experiments with the SPECJBB benchmark [28] in
Processors: 8 cores with private L1 data and instruction caches
Processor Model: 4-way issue superscalar
Private L1 D–Caches: Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency
Private L1 I–Caches: Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency
Shared L2 Cache (the second layer): 64–way set associative, 8MB, 64 bytes block size, 10 cycle access latency
Memory: 4GB, 200 cycle off-chip access latency
Control Enforcement Interval: 20 Million cycles

Table 2.3. Baseline configuration.

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mix 1</td>
<td>applu, apsl</td>
</tr>
<tr>
<td>Mix 2</td>
<td>art, gafort</td>
</tr>
<tr>
<td>Mix 3</td>
<td>gqgel, mgxdl</td>
</tr>
<tr>
<td>Mix 4</td>
<td>swim, wupwise</td>
</tr>
<tr>
<td>Mix 5</td>
<td>mgxdl, applu</td>
</tr>
<tr>
<td>Mix 6</td>
<td>wupwise, gqgel</td>
</tr>
</tbody>
</table>

Table 2.4. Various mixes (workloads) composed using the SPECOMP applications.

In order to evaluate the use of high-level QoS with METE, our implementation and evaluations are carried out using SIMICS [29], which is a full system simulator that allows simulation of multi-processor systems [29]. Table 6.2 gives the baseline configuration used in our experimental evaluation.

Our default configuration contains 8 cores. Each core has a private L1 cache and the on-chip L2 cache is shared by all cores. In most of our experiments, we formed our workloads using applications from the SPECOMP benchmark suite. Each workload is composed of two SPECOMP applications. Table 6.3 shows different mixes (workloads) we consider in most of our experimental evaluation. The SPECOMP benchmark applications are multithreaded programs and the number of running threads can be specified before the execution starts. In our experiments, each of these programs is executed using 8 threads and, therefore, one to eight cores can be allocated to each application by our proposed control centric scheme ($1 \leq \hat{c}(k) \leq 8$), since it is not beneficial from the performance and utilization perspectives to allocate more than eight cores to an application with eight running threads (note that, the number of threads does not change during execution when using METE). In other words, the number of threads of a running application indicates the maximum number of cores that can be allocated to it.

To run a workload that consists of two applications in our 8-core multicore machine, first, two processor sets (one core in each) are created using the `psrset` command.

\(^{2}\)SPECJBB is a multi-threaded Java program emulating a 3-tier system. In this program, each thread is a user that initiates transactions within a warehouse [28].
utility provided by the Solaris OS, and the minimum number of L2 cache ways (one way) and the minimum amount of memory bandwidth (5% of maximum available bandwidth) are allocated to each set. Each application in the workload is executed on one of the processor sets. At the end of each epoch, METE determines the resource allocations for the next time interval, and all three types of resources are partitioned among the co-runner applications based on that. If, for example, the new partitioning of the cores suggests an increase in the number of allocated cores to the first application by two, two cores are added to the processor set the first application belongs to. The default sampling period (epoch) to enforce our control decisions is set to 20 million cycles. In our sensitivity experiments, we study the impact of varying this default value.
Further, there are three other parameters that need to be specified in our evaluation: (i) Reference IPC: As in [19], the reference performance for each application can be specified as a percentage performance degradation with respect to the case when the application is executed independently on the multicore. Note that, this specification can be part of service-level agreement (SLA) between the system administrator and OS. We experimented with 10% to 40% degradations in our simulations for different applications. We also demonstrate the working of METE with a high-level metric (transactions per second) later in the chapter. (ii) Model Parameters: The model parameters are determined initially by using regression analysis (for each application separately); they are updated during execution to track the variations in the execution phases of the co-runner applications. (iii) Enforcement Interval: This is one of the important parameters in control design that affect the transient response and overall stability of the system. In our simulations, its default value is selected to be large enough to capture the effects of phase changes. Although increasing the duration of the intervals can reduce the performance overheads incurred by METE, it may also result in system instability and slow reaction to the environmental variations. In any case, the results presented below include all performance overheads incurred by METE.

2.5.2 Model Validation

As discussed earlier, the model that we employ (as given in Equation (2.2)) captures the influence of the variations in control parameters ($\hat{u}_i(k)$) on the measured IPC values ($IPC_i(k+1)$). We determine the values of the model parameters using the least square algorithm for each application and these values ($A$ and $B$) are shown in Table 2.5. To evaluate the accuracy of the obtained model, we compare the IPC values predicted by the model ($\hat{IPC}_i(k+1)$) and the actual (measured) IPCs ($IPC_i(k+1)$), as the resource allocation varies. Figure 2.8 plots the predicted and actual IPC values for different applications as the layout of the available resource partitioning changes. It can be observed from these plots that the employed model tracks the actual system output with small errors.

The coefficient of determination ($R^2$) and the Mean Absolute Percentage Error (MAPE) are two widely-used metrics to assess the accuracy of a model. MAPE
Table 2.5. Model parameters and assessment.

<table>
<thead>
<tr>
<th>Applications</th>
<th>A</th>
<th>B</th>
<th>MAPE</th>
<th>$R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>1.02</td>
<td>(0.31, 0.015, 0.077)</td>
<td>0.08</td>
<td>0.91</td>
</tr>
<tr>
<td>apsi</td>
<td>1.02</td>
<td>(0.28, 0.013, 0.067)</td>
<td>0.09</td>
<td>0.89</td>
</tr>
<tr>
<td>art</td>
<td>0.98</td>
<td>(0.33, 0.011, 0.083)</td>
<td>0.06</td>
<td>0.96</td>
</tr>
<tr>
<td>gafort</td>
<td>1.01</td>
<td>(0.32, 0.018, 0.094)</td>
<td>0.11</td>
<td>0.86</td>
</tr>
<tr>
<td>galgel</td>
<td>1.06</td>
<td>(0.34, 0.019, 0.074)</td>
<td>0.1</td>
<td>0.93</td>
</tr>
<tr>
<td>mgrid</td>
<td>1.02</td>
<td>(0.29, 0.014, 0.072)</td>
<td>0.05</td>
<td>0.98</td>
</tr>
<tr>
<td>swmn</td>
<td>1.01</td>
<td>(0.33, 0.017, 0.083)</td>
<td>0.06</td>
<td>0.96</td>
</tr>
<tr>
<td>wupwise</td>
<td>1.04</td>
<td>(0.32, 0.016, 0.073)</td>
<td>0.12</td>
<td>0.9</td>
</tr>
</tbody>
</table>

indicates the average error of the model, and an $R^2$ value which is close to 1 indicates high accuracy. These metrics are calculated as follows:

$$R^2 = 1 - \frac{\sum(IPC_i(k) - \hat{IPC}_i(k))^2}{\sum(IPC_i(k) - IPC_i(\text{avg}))^2}, (2.11)$$

$$MAPE = \frac{1}{K} \sum_{k=1}^{K} \frac{|IPC_i(k) - \hat{IPC}_i(k)|}{IPC_i(k)}. (2.12)$$

Table 2.5 lists the obtained values of the above metrics and also the values of the model parameters for the SPECOMP applications. We can see from the values presented in Table 2.5 that our employed ARMA model is a good approximation of applications’ behaviors.

2.5.3 Dynamics of METE

In this set of experimental results, we first show how different static partitioning schemes fail to satisfy the QoS targets of different applications when they execute on a multicore together. We then demonstrate that the same targets (QoS values) can be achieved by employing METE, which is able to dynamically track the specified performance targets (desired IPC targets) by partitioning the available resources (cores, L2 cache and off-chip bandwidth) dynamically during the course of execution. In addition, the performance of the running applications can be further enhanced beyond the specified targets by allocating the excess resources, as will be shown later in Section 2.5.6. We also study a case in which the QoS targets cannot be satisfied by partitioning the available resources due to the lack of resources.

Figures 2.9(a)-(c) plot the IPCs achieved by the applications in the mixes given
in Table 6.3, when the multicore resources are statically partitioned between the co-runner applications in each mix (for each mix, the two bars denote the two applications, given in Table 6.3). In Figure 2.9(a), each application receives an equal share of each type of resource, whereas in Figures 2.9(b) and (c), 75% of each type of resource is allocated to one of the applications in each mix; the remaining 25% is given to the other application. The IPC targets set for the applications in these mixes are shown as (horizontal) solid lines in Figure 2.9. Our main observation from Figures 2.9(a), (b) and (c) is that these static resource partitioning schemes fail to satisfy some of the specified QoS values. In comparison, METE takes the specified IPC targets as the reference inputs and decides how the resources we target need to be partitioned in each time interval. Figure 2.9(d) plots the achieved IPC values when METE is employed. In obtaining these results with METE, when not all the resources are used, we did not distribute them (the use of excess resources will be later discussed). As one can observe from these results, the target QoS values are satisfied successfully in this case for all workloads tested.

![Figure 2.9](image-url)

**Figure 2.9.** Tracking the IPC targets for eight applications from Mix1-Mix6. The IPC targets are shown in dash lines in the charts.

Figure 2.10 illustrates the dynamics of how METE tracks the IPC targets (shown in Figure 2.9) of different applications during execution. This tracking is achieved by varying the amount of resources allocated to each application dynamically. We observe from Figure 2.10 that the average maximum overshoot and
settling time\textsuperscript{3} of different applications are 0.12 and 5 time epochs, respectively. Note that the maximum overshoot and settling time are two important metrics for the evaluation of control based systems. Based on these results, we can conclude that METE can track specified QoS targets reasonably well. It is important to mention that all types of resources allocated to the applications are dynamically modulated to achieve the results shown in Figure 2.10. The dynamic variations of these resource allocations are shown in Figures 2.11, 2.12 and 2.13 for caches, bandwidth and cores, respectively. Further, METE is also able to compensate for the changes in the applications’ behaviors over time. For instance, in Figure 2.10(f) the measured IPC of mgrid unexpectedly increases at time 11 due to the variation in the application’s behavior (i.e., a phase change). METE compensates for this

\textsuperscript{3}The maximum overshoot is the largest difference between the system output and its steady state divided by the steady state value [26]. The settling time is the time in which the system output get close enough to the targets (for instance, 5% of their values)
variation and achieves the target QoS by changing the allocations of the available resources. Note that, in these tracking experiments, the goal is to meet minimum resource requirement of each running application to achieve the specified targets. Consequently, there exist excess resources that have not been allocated, as can be seen in Figures 2.11, 2.12 and 2.13.

As mentioned earlier, typically, the behaviors of the applications are not the same in different epochs and, as a result, we can observe that the controllers try to compensate for this by varying the resource allocations. Note also that the number of allocated cores has a significant impact on the measured IPC. Therefore, as can be seen from Figure 2.13, partitioning of the cores does not vary much once the IPC values get close enough to the specified targets.
2.5.4 QoS Sensitivity

The results presented so far were for specific QoS values highlighted in Figure 2.9. It is also important to study how METE behaves under different QoS values. Figure 2.14 plots the achieved IPC values by employing METE when different performance targets are specified for applu in mix1. In this experiment, we fixed the target IPC value of the other application (apsi) at 0.5. Our main observation is that, METE is successful in satisfying the QoS values, even when the specified value is high. Only when the QoS target (IPC) is 1.8, METE fails to satisfy.

2.5.5 Using High-Level Performance Targets

So far we have assumed that the performance target accommodated by METE are in terms of IPCs. However, depending on the application domain, higher level performance metrics (like the number of transactions per second) may be specified to METE in a flexible manner. The only change necessary to incorporate any high-level metric is the inclusion of a transducer component that transforms an observable low-level metric like IPC to high-level metric like the number of transactions per second. Note that, such a conversion is necessary because the high-level metrics cannot be measured directly from hardware. However, it can be periodically input to the controllers by the OS or the application. The transducer itself can be designed to model the relationship between the high-level metric and the observable low-level metric. To demonstrate this, we use SPECJBB and use transactions per second as our high level QoS metric. First, in Figure 2.15(a), we show how our model captures the relationship between the specified high-level QoS and the IPC. We run this benchmark with applu on our multicore with the baseline configuration (given in Table 6.2). Figure 2.15(b) shows how METE can track the
high-level performance target. Our observation from this plot is that METE can successfully track a high level metric such as transactions per second. Further, Figure 2.15(c) shows that performance goals of both of the co-runner applications are satisfied in this case.

2.5.6 Evaluation of the Resource Broker

If the cumulative resource demands of applications are greater than what the available system resources can provide (i.e., Expression (2.1) is not satisfied), the resource broker is triggered and it arbitrates between the resource demands from the over-demanding or greedy applications. As a case study, we evaluated the efficacy of the resource broker when QoS target set by the application mixes is 1.8 IPC which is impossible to satisfy with the available resources. Figure 2.16 shows the result of this study. We evaluated two policies for the resource broker: (1) Policy 1 penalizes each application equally i.e. if the core demands from each application is 4 but there are only 6 cores available, then this policy penalizes each application by a core (2) Policy 2 (our default policy) penalizes each application in proportion to their demands, i.e., it penalizes each application in proportion to its demand. We observe that, using Policy 1, none of the applications meets their specified target and penalizing each application equally can lead to undesirable behaviors since each resource has different implication for each application (see Figure 6.2). The overall fair speedup\(^4\) achieved using Policy 2 is 4.9% better than Policy 1, implying that penalizing each application proportionately is better than a simplistic approach of penalizing them equally.

To further evaluate the efficacy of the resource broker, we evaluate a second case in which there exist resources in excess of what the applications specify. In Section 2.5.3, we evaluated a case where all performance targets where achieved but still few cache ways and some off-chip bandwidth were left un-allocated. Using our resource broker, such a scenario can be leveraged to provide further application level improvement. Figure 2.17(a) shows a case where the un-allocated system re-

\(^4\)The Fair Speedup (FS) metric is defined as the harmonic mean of per application speedup with respect to the baseline equal resource share case [19] (i.e., \(FS = N/\sum_{i=1}^{N}(IPC_{ai}(base)/IPC_{ai})\), where \(N\) is the number of applications). FS can be used to assess the overall improvement in IPC values across different schemes.
sources are partitioned equally among the applications, and Figure 2.17(b) shows the case where they are allocated based on policy mentioned in Section 2.4.5 (our default policy: resources are allocated based on their potential performance improvements with this additional resource allocation). In both cases we find that the individual application performance improves but using our default policy, we observe that the FS metric is 4.8% better when compared to the scheme of redistributing the resources that oblivious of application behavior.

2.5.7 Sensitivity Study

Our goal in this section is to study sensitivity of our scheme to the values of some of the important parameters. The duration of the sampling periods (time intervals) is one of the important design parameters of our control scheme. To study the sensitivity of METE to this parameter, we performed experiments with the time intervals of 5-million and 20-million cycles. Figure 5.14 shows how well the measured IPC values for different applications track the specified IPC targets (as in Figure 2.10) when the time intervals are 5-million and 20-million cycles long. The results show that METE is not very sensitive to the execution intervals.

Smaller execution intervals can result in faster responses to the dynamic variations in application behaviors, and therefore, the magnitude of fluctuations may get reduced. In comparison, reducing the duration of time intervals, will increase the control scheme computation and enforcement overheads. Further, the intervals may not be long enough to see the impact of varying resource allocations in the next epoch that leads to inaccurate control decisions.

We next investigate the behavior of our scheme when the workload size and core count are modified. Figure 2.18(a) plots both the IPC targets and the achieved IPC values when the workloads consists of 4 applications running together on 8 cores. We also tested METE on a multicore with 16 cores. As shown in Figure 2.18(b), the IPC targets are not achieved on the 16-core multicore when no partitioning scheme is used. By employing METE, these targets are satisfied through integrated partitioning of the available resources. As can be observed from these results, the specified targets are still achieved when we increase the number of running applications and cores and METE is scalable for more applications.
2.6 Related Work

There has been extensive research on management and partitioning of the cache and off-chip memory bandwidth in multicores with the goal of improving the performance of hosted applications [5, 8, 9, 10, 11, 13, 14, 15]. Researchers have also explored various strategies to provide QoS in multicores [30, 31, 32, 33]. However, most prior studies have focused on the management of a single resource and have not considered multi-resource partitioning in multicores.

A multiple resource partitioning scheme called Symbiotic Resource Partitioning has been proposed in [18]. In this scheme, each of the shared cache space and off-chip memory bandwidth is partitioned dynamically based on the feedback from the partitioning of the other resource. The proposed scheme improves global
performance metrics and does not consider individual QoS requirements that each application may have. Bitirgan et al. [17] have proposed a framework to manage multiple shared resources on a multicore dynamically to achieve higher level performance objectives by using an artificial neural network based global resource manager that searches the design space of resource allocations by repeatedly querying the performance models followed by selecting the best candidate. Searching the allocation space is, in general, expensive and requires an efficient search mechanism.

Additionally, researchers have applied formal control theory in various domains of computer systems [34, 26] such as software services and performance [35, 36] and power management [20, 37]. A formal control theory approach is proposed in [38] to allocate shared virtualized resources to host application in order to meet QoS requirements.

In most of the prior schemes, the primary goal is to maximize the overall performance of a hosted workload. However, there may be applications that receive excess resources, whereas others might lack resources to achieve the specified targets. In contrast to this approach by prior works, in our work, the primary objective is to achieve the performance target of each individual application. Additionally, to our knowledge, ours is the first work that attempts to provide end-to-end QoS in multicore machines.

2.7 Concluding Remarks

In this chapter, we propose a control theory centric scheme, called METE, to partition multiple shared resources in a multicore machine among concurrently executing applications over the course of execution. In the current implementation of METE, we consider three types of shared resources: processing cores, shared cache space, and off-chip memory bandwidth. Assuming that each running application has a performance target to be satisfied, our main goal is to provide applications with sufficient resources to achieve the specified targets. For this, we propose a global resource broker for system wide resource management and a SIMO controller with an ARMA model for capturing the per-application demands. Our experimental results with application workloads indicate that METE is able to
partition the multi-level shared resources among co-running applications in most cases, such that the specified QoS targets such as IPC and throughput are satisfied. In summary, the results make a strong case for using an integrated resource management scheme like METE for multicore architectures.
Addressing End-to-End Memory Access Latency in NoC-Based Multicores

3.1 Introduction

In this chapter, we propose two network prioritization schemes that can cooperatively reduce end-to-end memory access latencies in NoC-based multicores. One of these schemes targets memory response messages (i.e., messages from the memory controller), whereas the other targets memory request messages (i.e., messages from the core side). Our first scheme prioritizes memory response messages such that, in a given period of time, messages of an application that experience higher latencies than the average message latency of that application are expedited. This helps us reduce the number of off-chip requests with high latencies and achieve a more uniform memory latency pattern. While one might want to apply a similar optimization for memory request messages as well, it is hard to identify whether a data access will experience high latency at the time of the request generation as it has not even entered the network yet. Instead, our second scheme approaches problem from another angle, where it expedites request messages to improve memory bank utilization. More specifically, observing that at any given time frame some banks are heavily loaded whereas others are idle, our second scheme prioritizes (at
network routers) the request messages that are destined for idle banks over the others. This helps to improve bank utilization and prevent long queues in front of the memory banks. These two network prioritization-based optimizations, when applied together, result in uniform memory access latencies with a low average value.

We implemented our schemes in a simulation environment and tested their effectiveness using a diverse set of multiprogrammed workloads. Our experiments with a 4 × 8 mesh network-based multicore show that our first scheme is very effective in reducing the number of off-chip requests with very high latencies. As a result, we are able to achieve 11%, 6%, and 10% performance improvement on memory intensive, memory non-intensive, and mixed multiprogrammed workloads, respectively. When both our schemes are used together, these improvements jump to 15%, 10% and 13%, in the same order. Our experimental evaluation also reveals that the proposed schemes consistently generate good results under different values of the main simulation parameters.

The rest of this chapter is structured as follows. We provide an overview of our target multicore architecture and the main motivations behind our optimizations in Section 5.2. In Section 6.2, we give the details of our proposed schemes and discuss our network prioritization implementation. Section 3.4 presents our experimental framework and discusses our results. We contrast our approach with related work in Section 6.4, and conclude the chapter in Section 3.6.

3.2 Background and Motivation

3.2.1 Target Multicore System

Figure 3.1 shows the high level view of an NoC-based multicore system that we target in this chapter. As can be observed, this multicore system has a number of nodes arranged in a 2D grid (an example present day product using a similar organization is TILEPro36 by Tilera [46]). In this architecture, each node contains a processing element and a hierarchy of caches; only two levels of caches are shown in Figure 3.1. The shared last level cache (L2) has a banked organization where the L2 space in each tile represents a separate bank. The mapping of data to L2 cache
banks is very important as far as performance is concerned, and in one mapping scheme (SNUCA [42]) which is also used in this chapter, each cache block-sized unit of memory is statically mapped to one of the cache banks based on its address [47]. This address-based mapping results in an interleaving of cache blocks across cache banks. The main memory is connected to this multicore via memory controllers (MCs) on the corners (or sides), each controlling a single memory channel with possibly many memory modules (DIMMs) connected. Each channel consists of one or more ranks, and each rank is a group of memory banks. Different memory banks can be accessed at the same time, but they share common address and data buses. More details on the internal structure of the main memory can be found in [48, 49, 14]. Note that the target memory bank for each memory request is fixed and depends on the OS address mapping. If more than one memory request are queued in a memory controller trying to access the same bank, the memory scheduler decides which one should be served first [50, 51].

3.2.2 Memory Accesses in an NoC-based Multicore

Once a core issues a memory request, first the local L1 is checked for the presence of the requested data and if this fails, the request is forwarded to the corresponding L2 bank via the on-chip network. Finally, if the data is not found in the L2 bank, an off-chip memory access is made, again using the on-chip network. Figure 3.2 depicts this flow in our target architecture. One can make several critical observations from this figure. First, the L2 access latency is a function of the
distance between the requesting core (L1) and the target L2. It is to be noted that, longer this distance gets, higher the chances for network contention. Second, the total latency to serve an L2 miss (i.e., the overall memory latency) includes memory controller queuing and memory service latencies (i.e., the memory access latency) as well as the latency of four trips on the network: L1 to L2, L2 to memory controller (MC), MC to L2, and L2 to L1. As a result, network latency can play a significant role in overall memory access latency (in fact, our experiments show that, even in a 4 × 8 multicore, cumulative network latency can be comparable to memory access latency). Third, the time spent in an MC depends on other requests that target the same MC as well as the (memory request) scheduling policy adopted by the architecture. Fourth, at any given time, there will be many memory requests traveling on the network. These requests contend for the shared network resources, resulting in additional delays. For example, a message from a core destined to an L2 may contend for the same link/router with another message going from an L2 to an MC. Lastly, the time it takes to service a memory request (after it gets its turn from the memory scheduler) depends on whether it results in a row buffer hit or not.

3.2.3 Out-of-Order Execution

In our target system, each processing element (core) is a high performance out-of-order processor. It can issue multiple instructions by looking at its instruction window. As a result, if these instructions need to access the main memory, the memory requests can be issued at the same time with the hope of improving the overall performance by not stalling the processor for a single memory request. This is referred to as Memory Level Parallelism (MLP) [52, 53]. Note that although memory requests may return out-of-order, the instructions are committed in-order and therefore, a higher memory access latency for an instruction may affect the performance of an application significantly [54]. As an example, suppose that an application issues four load instructions (load-A, load-B, load-C and load-D, as shown in Figure 3.3) that need to access memory. As discussed, these instructions may be ready to commit at different times due to the network and memory latencies and also whether they are hits or misses in the on-chip caches. Note that in the
Figure 3.2. Detailed flow of a memory access. In order to look-up the corresponding L2 bank, a request message is sent from L1 to L2 (path1). If the data is a hit in L2 cache, then the response data is sent back to the L1 cache (path5). In case of an L2 miss, the request is forwarded to the memory controller (path2). The memory controller schedules this request (path3) and the response is sent first to the L2 cache (path4) and then to the L1 cache (path 5).

instruction window shown in Figure 3.3, the load instruction (e.g., load-A) that needs to wait for a longer time to access the data (as compared to the other instructions) becomes a bottleneck for the application. The chances that a memory request will be a bottleneck depend on its delay and also other outstanding memory requests from the same application. For example, if an application is not memory intensive (an application with a low MPKI [misses per kilo instructions]), the likelihood that one of its off-chip memory requests would be a bottleneck is high. This is why previously-proposed memory scheduling schemes [55, 44] and network prioritization [56], higher priorities are given to the requests coming from this type of applications. However, these prior works do not consider the network and memory access latencies together. For instance, in the application-aware network prioritization scheme proposed in [56], the time it takes to access the memory is assumed to be constant, whereas in reality memory requests (when they reach the memory controller in an NoC based multicore) may face different latencies due to queuing.
3.2.4 Motivations

Our proposed approach consists of two network prioritization based schemes (Scheme-1 and Scheme-2). These schemes are proposed based on two main motivations presented in this section. Specifically, Motivation-1 and Motivation-2 below motivate for Scheme-1 and Scheme-2, respectively.

3.2.4.1 Motivation 1: Some Memory Accesses Experience Much Higher Delays than Others

As illustrated in Figure 3.2, a round-trip memory access has five paths/stages. The total delay of each access is equal to the sum of the delays on these paths. To have a better understanding of how much time each memory request spends on different stages/paths on average, we simulated a $4 \times 8$ NoC-based multicore with 4 memory controllers attached to the corners, and each core executing an application from the spec2k6 benchmark suite [57] (see workload-2 in Table 5.3; more details on the experimental configuration and workloads are given in Section 6.3). Figure 3.4 plots the average round-trip delays of the off-chip memory accesses issued by one of the cores (executing application *milc*), broken into individual components. The values on the x-axis are the “delay ranges” and each bar gives the “average latency” of the memory accesses which experienced a delay value within the corresponding range on the x-axis. For example, the first bar from the left gives the average round-trip delay of the memory accesses with delay falling between 150 cycles and 200 cycles. Each bar is partitioned into five parts, which shows the breakdown of the average delays. These partitions from bottom to top represent: (1) the average network delay between L1 and L2 banks (path-1 in Figure 3.2), (2) the average
Figure 3.4. The average delays of the off-chip memory accesses issued by one of the cores (executing milc in workload-2).

delay from the L2 bank to the memory controller (path-2 in Figure 3.2), (3) the average delay added by the memory (queuing delay + memory access latency), (4) the average network delay from the memory controller to the L2 bank (path-4 in Figure 3.2), and (5) the average network delay form the L2 bank to the source core. As can be seen from this figure, due to the large network and memory queuing delays, some messages experience significantly larger delays as compared to others.

However, the numbers of accesses in different delay ranges are not the same. Figure 3.5 plots the distribution of the memory accesses across different delay regions. The values on the x-axis correspond to different latencies observed during execution, and the y-axis shows the percentage of the total memory requests for different latencies. The area under the curve between $x = d_1$ and $x = d_2$ gives the fraction of the memory accesses that have a total latency between $d_1$ and $d_2$. As can be observed from this plot, the latencies of most of the accesses are around the average, but there are few accesses with very large delays (600 cycles or more). Unfortunately, these high latency accesses can be very problematic and degrade overall application performance significantly.

Our first scheme identifies these (high latency) accesses (after the DRAM access is completed) and attempts to reduce their latencies with the goal of improving the overall system performance. Specifically, our proposed scheme tries to expedite these slow messages on their return path by giving them a higher priority in the on-chip routers. In other words, we try to make the latencies of the different memory
Figure 3.5. The latency distribution of the memory accesses issued by one of the cores (executing application milc in workload-2).

accesses issued by the same application as uniform as possible.

3.2.4.2 Motivation 2: Bank Loads are Non-Uniform

Once a memory request passes through the on-chip network and reaches the memory controller, it may face a queue in front of the target memory bank. We observed during our experiments that, at any given time, different banks can have very different queue lengths (i.e., the number of pending requests). For instance, some banks may be idle for a period of time, whereas other banks may be serving their requests. Figure 3.6 plots the average idleness of different banks of one of our memory controllers (again in our $4 \times 8$ multicore). To compute the average idleness, the queue of each bank is monitored at fixed intervals. For example, an average idleness of 0.8 in this plot implies that 80% of the times, the bank is monitored, there has been no request in its queue.

To explain how this observation can be exploited, consider Figure 3.7 which shows the states of three banks controlled by one of the memory controllers at time $t_1$. As can be seen, Banks 0 and 1 have requests to be serviced, whereas Bank-2 is idle. Assume further that there are three memory requests (from L2 misses) in the on-chip network destined for these banks as shown in Figure 3.7 (R-0, R-1 and R-2 which are destined for Bank 0, 1 and 2, respectively). In this example, the memory utilization could be improved if R-2 could reach its memory controller faster so it can be serviced by Bank-2 immediately. Our proposed scheme exploits
Figure 3.6. The idleness of different banks of a memory controller.

![Figure 3.6.](image)

Figure 3.7. A snapshot that shows the states of the three banks controlled by one of the memory controllers.

![Figure 3.7.](image)

this load variation across different banks by using the local history at each node to accelerate the memory requests on the network destined for the idle banks.

3.3 Our Approach

In this chapter, we propose two network prioritization schemes that consider the states of the memory bank queues and the overall memory access delays in NoC-based multicores. In this section, we first explain how our schemes work and then discuss the implementation details of the specific network prioritization method we employ. In our proposed approach, Scheme-1 focuses on the response messages coming back from the memory and Scheme-2 considers request messages destined for the off-chip memory. We assume a one-to-one mapping between applications
3.3.1 Scheme-1: Reducing Variations across Memory Accesses

This scheme is based on the motivation discussed in Section 3.2.4.1. Recall from Figure 3.5 that some memory accesses issued by an application can take much longer times to complete compared to other accesses from the same application (we call them late accesses). As an example, in Figure 3.5, about 10% of the accesses have delays larger than 600 cycles (while the average latency is about 350 cycles). As mentioned earlier, these late accesses can severely degrade the performance of applications. Suppose for example that an application executes \( a = b + c \). In this instruction, if one of the operands becomes ready only after a long time, this will delay the completion of this instruction as well as all instructions that depend on it. The main goal behind our Scheme-1 is to mitigate the impact of the late accesses by helping them reach their destinations faster.

As shown in Figure 3.4, three different factors can cause memory accesses to be late in the system: (1) the network latency from the source cache bank to the corresponding memory controller, (2) the memory access latency (which includes both queuing delay and the time it takes to access the DRAM), and (3) the network latency from the memory controller to the source cache bank. Therefore, in order to reduce the delay experienced by the late accesses, one can potentially target these three delays. The first and third delays can be reduced by giving higher priorities to the request and the response messages in the network. However, for the first part (network latency after an L1 miss, from the L1 bank to the memory controller), it is not known that a memory request is going to be a late memory access, since the total delay depends mainly on the memory response time (factor 2) and the network delay of the response messages (factor 3), as illustrated in Figure 3.4. To reduce the response time of the memory (factor 2), late memory requests could be scheduled faster at the bank queues. However, prioritizing these requests would harm other late requests in the same bank. In other words, if one request in a bank queue suffers long queuing delay, it is most likely that this bank is a “common bottleneck” for all co-running applications, and consequently
prioritizing the requests from one application over others would not help in a global sense.

Motivated by these observations, Scheme-1 attempts to reduce the delays coming from the third factor, since right after the memory controller the delays accumulated so far can give a good estimate of whether a memory access is going to be “late” or not. In this scheme, each memory message (packet) has a field that maintains the age (“so-far delay”) of the corresponding access (based on the delay at each router/stage, this field is updated over the round-trip of the memory access). Note that the variations in network latencies (the first factor) can come from two contributions: (1) the distances in the network from L1 to L2 and from L2 to MC, and (2) the network traffic. Both these sources are captured by our so-far delay field, and consequently handled by our scheme.

To illustrate how Scheme-1 works, let us explain it through a simple example. In Figure 3.8, suppose that MC1 has received a memory request from core-1, and R0 is the corresponding response message for that request after it is serviced by MC1. Once R0 is ready, the age field (the so-far delay) in R0 gets updated based on the delay that has been imposed by the memory. After that, the updated so-far delay is compared with a threshold value \( Th \). If it is larger than \( Th \), this memory access is considered to be late and, on the return path, R0 will have a higher priority to reach its destinations (the paths from MC1 to L2 and L2 to L1 illustrated in Figure 3.8). Note that in the memory controllers, each core/application is associated with a threshold value. These threshold values are determined and sent to the MCs by the cores periodically (every 1ms) over the execution and they are also prioritized over other requests (each MC has storage to keep the threshold values). Since this happens every 1ms, the overhead on the other messages is small. In our default configuration, each core sets its threshold value to \( 1.2 \times Delay_{avg} \), where \( Delay_{avg} \) is the average delay of the off-chip memory accesses that belong to that application/core. Once a response message for an off-chip memory access comes back, the round-trip latency of that off-chip memory access is read from the message (the age field) and \( Delay_{avg} \) is updated accordingly. Consequently, during the course of execution, the threshold value used for each application changes dynamically.

Note also that we compute the priority of a response message when the message
Figure 3.8. An example to illustrate how Scheme-1 works. MC1 has received a memory request from core-1, and R0 is the corresponding response message for that request after it is serviced by MC1.

is about to be injected into the network by the memory controller. However, at that time instant, the memory controller only knows the so-far delay of the request, and therefore, any delay threshold that will be used in calculating the priority must be defined based on the average delay up to and including the memory access latency ($\text{Delay}_{\text{so-far-avg}}$). For this reason, we set our threshold to $1.2 \times \text{Delay}_{\text{avg}}$ (which is almost equivalent to $1.7 \times \text{Delay}_{\text{so-far-avg}}$; both averages are marked in Figure 3.9 using vertical solid lines). Figure 3.9 plots two delay distributions of the memory accesses (issued by the core executing application milc from workload-2). Specifically, the dashed curve shows the distribution of the round-trip delays of the messages, while the solid curve plots the distribution of the so-far delays at the point right after the memory controller (i.e., when the data is read from the memory, and is about to be injected into NoC).

Implementation Details. As discussed earlier, Scheme-1 needs each memory access message to have a field (called “age” field) capturing the so-far delay of the message in the NoC. We assume that this field is 12-bit long and its value is in cycles. We believe a 12-bit field is sufficient in our work, since it is very rare
that the round-trip of an off-chip message takes more than 4096 − 1 cycles in the system. We assume that the header flit has room for the so-far-delays (12-bit) in a 128-bit header. At each router/memory-controller, once the message is ready to be sent out, the age-field value is updated as follows:

\[
age = age + \frac{(local\_time - message\_entry\_time) \times FREQ\_MULT}{local\_frequency},
\]

where \(local\_time\) is the local router/MC clock cycles, \(message\_entry\_time\) is the time when the message enters the routers/MCs, \(FREQ\_MULT\) is a constant value (used to work in integer domain, instead of fractions of cycles) and \(local\_frequency\) is the local operating frequency. As one can observe, in each hop, first, the local delay of each incoming message is computed \((local\_time - message\_entry\_time)\) and then this value is added to the age-field of the message. Note that dividing the local delay by the \(local\_frequency\) allows our scheme to work when routers/MCs operate at different frequencies. Note also that, the entry time of each message is not transferred over the network, and each router keeps track of its own local delay and computes the so-far delay \(locally\). Instead of updating the age field at each hop, one may choose to employ a global clock to compute the so-far-delays. In this method, each message is labeled with the starting time and, at the destination, the starting time is subtracted from the current time to obtain the so-far-delay. However, in this mechanism, all the nodes need to have clocks and these clocks
should be synchronized.

3.3.2 Scheme-2: Balancing Memory Bank Loads

Based on Motivation-2 discussed in Section 3.2.4.2, the main goal behind this scheme is to increase the utilization of the memory banks in the system. To do this, the requests destined for the idle banks are given higher priorities in the network to reach their target memory controllers faster. However, the number of requests queued in each memory bank varies over the execution, and more importantly, a core in our 2D mesh does not know about the states of the memory banks. In other words, no global information is available to the nodes to help them decide whether different memory requests they issue are destined for idle banks or not. To address this problem, in our proposed scheme, each node exploits “local information” to estimate the pressure imposed by the requests on the memory banks. Specifically, each router keeps and updates a table (called “Bank History Table”) that records the number of off-chip memory requests sent from this router to each bank in the last $T$ cycles. An off-chip request is given a higher priority in the on-chip network if no message has been sent to the same bank according to the value in the bank history table. In Section 3.4, we present results that show how bank idleness is reduced when Scheme-2 is employed.

Note that due to Scheme-2 and the variation in the network latencies, different requests that are destined for the same memory bank may not reach the bank queue in order (based on their ages). This may cause more delays for the late accesses. Message ordering can be handled by the memory scheduler, since the scheduler knows the timing information for all arriving messages.

3.3.3 Network Prioritization Implementation

Further, that would be very costly to use stronger routers and more network resources. Our schemes can be used along with other architectural solutions to provide additional benefits. In this section, we give the implementation details of our prioritization method. In our NoC-based multicore, we assume a typical architecture with 5-stage pipeline for the on-chip routers [58]. We further assume flit-buffering and VC (virtual channel) credit-based flow control at every router.
In this architecture, each network message is split into several flits with fixed-lengths and the flow of the flits follows the wormhole switching protocol [60]. For further details, we refer the reader to [58].

Once a header flit enters a router, in the first stage of the pipeline, which is called buffer write (BW), the flit is written into the allocated input buffer. In the next stage (routing computation (RC)), the routing algorithm is invoked and the right output port is determined for the packet based on the position of the current router and the destination address extracted from the header flit. The next stage, called virtual channel allocation (VA), is where a free output virtual channel is reserved for the packet. After this stage, the flit needs to get the permission to use the crossbar switch (switch allocation (SA)). In the last stage, the flit traverses the switch (ST) to be sent over the physical link (LT). Note that only the header flit passes through these stages, and the body and tail flits skip the RC and VA stages as they simply follow the header flit.

In our approach, the messages to be expedited have higher priorities in VC and SW arbitrations (for virtual channel and switch allocations). The VC arbitration is done when an output VC is preferred by more than one input VC and, one of them should be granted. The SA arbitration on the other hand has two phases. In the first phase, only one of the ready VCs is selected per input port and, in the second phase, one VC is selected for each output port. The reason for these two arbitrations is that, at each cycle, only one flit can be sent from an input and each output is able to receive a single flit.

Although prioritization in the arbitration stages expedites the messages and helps them reach their destinations faster, there is a limit in this message acceleration, since it takes at least 5 cycles for the data flits to pass through a router in the network. To tackle this limitation, we also employ a mechanism called “pipeline bypassing” [58], which gives the late messages the opportunity to go through fewer number of pipeline stages in the routers. In this mechanism, once the header flit of a message (with high priority) enters a router, in the same cycle, after it is written to the input buffer, one free output VC is allocated to it (VC allocation) and the switch is reserved for this flit (SW allocation) (in other words, the BW, RC, VA and SA stages are combined and performed in the first stage which is called the “setup stage”). Figure 3.10 illustrates the pipeline stages for the baseline router.
as well as the one with pipeline bypassing). If in this cycle, there is a conflict in selecting the free output VC and the switch allocation between this flit and a flit (with normal priority) in another VC, the flit with the higher priority is prioritized over the others. Body flits also use this “bypassing” only when the input buffer is empty once they enter the router (Note that the input buffers are always idle when the header flits enter).

To avoid starvation in the system, we also use “age fields” in the messages (which are also used in Scheme-1). In our prioritization scheme, flit A is prioritized over flit B in the cases discussed above, if (1) flit A has a high priority but flit B has the normal one, and (2) the age of message B is not more than \( T \) cycles greater than that of message A (flits A and B belong to messages A and B). Note that the routers also consider the local delays in addition to the age fields in the messages. “Batching” \([43]\) is another method that can be used to avoid starvation. In this method, time is divided into intervals of \( T \) cycles and there is a field in each message that indicates the batching interval in which the message was injected to the network. The packets are prioritized if they belong to the same batch. However, this method requires a synchronization among the cores since they need to access a common global time.

### 3.4 Experimental Evaluation

In this section, we first explain our simulation framework and the different workloads that we used in our evaluations and then, present and discuss our experimental results.
3.4.1 Setup, Metrics, and Workloads

Setup: We use GEMS [61] as our simulation framework to evaluate our proposed schemes. This framework allows us to simulate a NoC-based multicore system. GEMS consists of two main components called Opal and Ruby, and employs Simics [29] as the base simulator. Opal implements an accurate model for out-of-order cores, and Ruby implements the network modules (the routers and the links) and the memory controllers. The instructions are executed by Opal and, if they require a memory access, a request message is injected to Ruby. Opal receives the response message once the requested data comes back from an L2 bank or one of the memory controllers (simulated by Ruby). Table 6.2 gives our baseline configuration. As shown in this table, our baseline system has 32 cores connected by a $4 \times 8$ mesh-based NoC, and also 4 memory controllers are connected to the four corners of the network. In our evaluation, we employ cache line interleaving where the consecutive lines of an OS page are mapped to different memory controllers. Note that this strategy helps to avoid creating hot spots in the memory controllers. In the results presented below, the overheads incurred by our proposed schemes are included.

| Processors | 32 out-of-order cores with private L1 data and instruction caches, instruction window size: 128, LSQ size: 64 |
| NoC Architecture | $4 \times 8$ |
| Private L1 D&I-Caches | Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency |
| Number of L2 Cache Banks | 32 (distributed over the network) |
| L2 Cache | 64 bytes block size, 10 cycle access latency |
| L2 Cache Bank Size | 512KB |
| Number of Banks Per Memory Controller | 16 |
| Memory Configuration | DDR-800, Memory Bus Multiplier: 5, Bank Busy Time: 22 cycles, Bank Delay: 2 cycles, Read-Write Delay: 3 cycles, Memory CTL latency: 20 cycles, Refresh Period: 3120 |
| Cache Coherency Protocol | MOESI_CMP_Directory |
| NoC parameters | 5-stage router, flit size: 128 bits, buffer size = 5 Flits, Number of virtual channels per port = 4, Routing algorithm: X-Y |

Table 3.1. Baseline configuration.

Evaluation Metric: We use normalized weighted speedup metric to quantify the benefits of our proposed scheme. Weighted speedup is defined as: $WS = \sum \frac{IPC_i^{(shared)}}{IPC_i^{(alone)}}$, where $IPC_i^{(shared)}$ is the IPC of application $i$ when it is executed with the other applications in the workload and $IPC_i^{(alone)}$ is the IPC of the same application when it is executed alone without having any contention with the other applications. In this section, we present the weighted speedup values
that are normalized to that achieved by the default case where no prioritization is employed.

**Workloads:** We formed our workloads using the applications from the spec2k6 benchmark suite [57]. Table 5.3 gives the 18 workloads that we used in our experiments on our 32-core system (the numbers in the parenthesis represent the number of copies for each application in the workload). The workloads listed in Table 5.3 cover all applications and are categorized into three different groups based on the memory intensity of the applications: (1) Workloads 1 through 6 (mixed workloads): in these workloads, half of the applications are memory intensive (applications with high MPKI) and the remaining ones are memory non-intensive. (2) Workloads 7 through 12: all applications in this category are memory intensive. (3) Workloads 13 through 18: none of the applications in this group is memory intensive. In other words, to form a workload, we first categorize the applications into three groups based on the memory intensity (since our scheme is proposed for memory accesses) and then, for each workload, we randomly pick 32 applications from the specific category (note that this strategy may choose some applications multiple times, but all the applications in a workload belong to the same memory-intensity region). MPKI values representing the memory intensities of the applications in the spec2k6 suite can be found in [55].

### 3.4.2 Results

We now present the experimental results collected using various configurations. In executing a workload, the simulation was fast-forwarded for 1 billion cycles, and then we collected our statistics in the next 100-million cycle run. As stated earlier, we employed a one-to-one mapping between applications and cores, that is, each core executes one application and the application-to-core mapping does not change during the execution.

**Results for a 32-core system with the baseline configuration:** As mentioned earlier, in this configuration, 32 cores are connected by a $4 \times 8$ 2D mesh-based NoC and 4 memory controllers are placed in four corners of the mesh (other parameters are as given in Table 6.2).

Figure 3.11 plots the normalized weighted speedup values achieved for our work-
| Workload-1 | mcf(3), lbm(2), xalancbmk(1), milc(2), libquantum(1), leslie3d(5), GemsFDTD(1), solexp(1), omnetpp(2), perlbench(1), astar(1), wrf(1), tonto(1), sjeng(1), namd(1), hammer(1), h264ref(1), game(1), calculix(1), bzip2(3), bwaves(1) |
| Workload-2 | mcf(4), lbm(2), xalancbmk(2), milc(3), libquantum(2), GemsFDTD(1), solexp(2), perlbench(2), astar(3), wrf(3), povray(1), namd(3), hammer(1), h264ref(1), gcc(1), dealII(1) |
| Workload-3 | mcf(4), lbm(1), milc(2), libquantum(5), leslie3d(2), sphinx3(1), GemsFDTD(1), omnetpp(1), astar(2), zeusmp(2), wrf(2), tonto(1), sjeng(1), h264ref(1), gobmk(1), gcc(1), game(1), dealII(1), calculix(1), bwaves(1) |
| Workload-4 | mcf(1), lbm(2), xalancbmk(3), milc(2), leslie3d(1), sphinx3(3), GemsFDTD(1), solexp(3), omnetpp(1), astar(2), zeusmp(1), wrf(1), tonto(1), sjeng(1), h264ref(2), gcc(1), game(1), dealII(1), calculix(1), bwaves(1) |
| Workload-5 | mcf(4), lbm(2), xalancbmk(4), milc(1), leslie3d(1), sphinx3(1), solexp(4), astar(2), zeusmp(2), wrf(1), sjeng(1), povray(2), namd(1), hammer(1), h264ref(2), gromacs(1), gcc(1), game(1), dealII(2), bzip2(3) |
| Workload-6 | mcf(2), xalancbmk(2), milc(1), libquantum(1), leslie3d(2), sphinx3(3), GemsFDTD(3), solexp(2), omnetpp(1), perlbench(2), wrf(1), tonto(2), hammer(1), gromacs(1), gobmk(1), gcc(1), game(1), dealII(2), bzip2(3) |

| Workload-7 | mcf(1), lbm(5), xalancbmk(5), milc(1), libquantum(5), leslie3d(4), sphinx3(3), GemsFDTD(6), solexp(2) |
| Workload-8 | mcf(3), lbm(2), xalancbmk(4), milc(3), libquantum(8), leslie3d(3), sphinx3(4), GemsFDTD(5) |
| Workload-9 | mcf(4), lbm(5), xalancbmk(4), milc(3), libquantum(4), leslie3d(2), sphinx3(6), GemsFDTD(2), solexp(2) |
| Workload-10 | mcf(4), lbm(3), xalancbmk(3), milc(2), libquantum(4), leslie3d(3), sphinx3(4), GemsFDTD(8), solexp(1) |
| Workload-11 | mcf(3), lbm(6), xalancbmk(2), milc(5), libquantum(1), leslie3d(2), sphinx3(4), GemsFDTD(4), solexp(5) |
| Workload-12 | mcf(2), lbm(3), xalancbmk(3), milc(6), libquantum(5), leslie3d(4), sphinx3(4), GemsFDTD(5) |

| Workload-13 | perlbench(1), astar(3), zeusmp(2), wrf(2), sjeng(3), povray(2), hammer(1), gromacs(2), gcc(1), game(2), dealII(2), calculix(5), bzip2(2), bwaves(2) |
| Workload-14 | omnetpp(3), perlbench(1), zeusmp(2), tonto(1), sjeng(1), povray(2), namd(2), hammer(4), h264ref(3), gromacs(2), gobmk(3), game(3), bzip2(1), bwaves(4) |
| Workload-15 | omnetpp(2), perlbench(1), astar(1), zeusmp(3), sjeng(1), povray(1), namd(1), hammer(2), h264ref(1), gromacs(2), gobmk(2), gcc(2), game(1), dealII(4), calculix(2), bzip2(2), bwaves(2) |
| Workload-16 | omnetpp(3), perlbench(3), astar(2), zeusmp(1), wrf(2), sjeng(3), povray(3), namd(1), hammer(2), h264ref(1), gobmk(1), gcc(4), game(3), dealII(2), bzip2(1), bwaves(1) |
| Workload-17 | omnetpp(2), perlbench(2), astar(1), zeusmp(2), wrf(1), tonto(2), sjeng(1), povray(2), namd(1), hammer(4), h264ref(1), gobmk(2), gcc(2), game(1), dealII(3), calculix(2), bzip2(3) |
| Workload-18 | omnetpp(2), perlbench(4), zeusmp(2), wrf(2), tonto(2), sjeng(2), namd(1), hammer(2), h264ref(1), gromacs(2), gobmk(2), gcc(4), game(3), calculix(2), bzip2(1), bwaves(1) |

Table 3.2. Workloads used in our 32-core experiments.

loads, which are categorized into three groups as shown in Table 5.3. As discussed in Section 6.2, our approach to reducing end-to-end latency employs two complementary schemes. In Scheme-1, if the so-far delays of the response messages are larger than a threshold value (the default value of this threshold is \( 1.2 \times \text{Delay}_{avg} \)) after the memory controller stage, they are given a higher priority in the network on the return paths to reach their destinations faster, whereas Scheme-2 acceler-
ates the request messages that are destined for the idle memory banks. As stated earlier, in this scheme, the decision of whether a request message is destined for a idle bank or not is made based on the local information. When an L2 miss occurs, if over the last $T$ cycles, the number of requests sent to the same bank is less than a threshold ($th$), the request is given high priority (the default values for $T$ and $th$ are 200 cycles and 1, respectively).

![Graphs showing speedup values for different workloads](image)

(a) Mixed workloads.  
(b) Memory intensive workloads.  
(c) Memory non-intensive workloads.

**Figure 3.11.** Speedup values achieved for different workloads executed on a 32-core system.

In Figure 3.11, two bars are presented for each workload. The first bar shows the performance improvement achieved when Scheme-1 is employed in the system and the second one is for the case where Scheme-1 and 2 are employed together. Note that the weighted speedup values presented in Figure 3.11 are normalized to the base case in which the applications are running together but no prioritization scheme is employed.

As can be observed from Figure 3.11, our approach (Scheme-1 + Scheme-2) improves performance by up to 13%, 15% and 10% for the mixed, memory intensive and memory non-intensive workloads, respectively. Further, in general, higher performance improvements are achieved when the applications are more memory intensive. This is because when the co-running applications are more memory intensive, the traffic on the network is higher and, as a result, the impact of our network prioritization based approach on accelerating the late messages becomes more pronounced. However, our proposed scheme does not improve the performance for all the workloads tested. For instance, the speedup values achieved by Scheme-1 for workloads 2 and 9 are slightly less than 1. The reason for this behavior is that giving higher priorities to some of the messages in the network hurts
the other messages and, this can offset the benefits of our scheme in some cases.

The impact of employing Scheme-1 and Scheme-2 can be observed in the distribution of the end-to-end latencies and the average idleness of the memory banks, respectively. Figure 3.12(a) plots 8 “cumulative distribution functions” (CDFs) of the off-chip memory accesses generated by the first 8 applications in workload-1. In this figure, the values on the x-axis are the total latencies (in cycles) and the y-axis shows the fraction of the total number of off-chip memory accesses. Each curve corresponds to one of the applications and the $F(x)$ value gives the fraction of the total accesses with delays less than $x$. Figure 3.12(b) plots the new CDFs of the off-chip accesses for the same 8 applications when Scheme-1 is employed. It can be observed from Figure 3.12(a) that 90% of the messages have an average total latency about 700 cycles (shown with dashed lines), whereas Scheme-1 reduces it to about 600 cycles (see Figure 3.12(b)).

![Figure 3.12](image)

**Figure 3.12.** (a) The CDFs of the off-chip accesses for the first 8 applications in workload-1. The values on the x-axis are the total (end-to-end) memory access latencies (in cycles), and the y-axis shows the fraction of the total number of off-chip memory accesses. (b) The CDFs of the off-chip accesses for the first 8 applications in workload-1, when Scheme-1 is employed. (c) The delay distributions of the memory accesses for one of the applications (lbm) in workload-1 before and after Scheme-1 is employed. This distribution change results in 8% performance improvement.

Figure 3.12(c) plots the “probability density function” (PDF) of the memory accesses issued by one of the applications (lbm) in workload-1 before and after Scheme-1 is employed (the area under the curve shows the fraction of total number of accesses). Employing Scheme-1 reduces the number of messages with large delays and move them from Region-1 to Region-2. This results in about 8% performance improvement for this application. As one can observe from this figure, not all the messages are transferred to Region-2 and there are still messages in
Figure 3.13. Idleness values of different banks of a memory controller when Scheme-2 is employed and when no scheme is used. As can be observed, Scheme-2 reduces idleness in most of the banks.

Figure 3.14. Average idleness of the memory banks (workload-1) over the execution.

Region-1. This is because Scheme-1 reduces a portion of the total memory access latency and, for instance, if a message spends a long time in the memory queue (e.g., 700 cycles), Scheme-1 will not be able to move this message to Region-2.

To illustrate the impact of Scheme-2, we plot in Figure 3.13 the idleness values for different banks of a memory controller when Scheme-2 is employed and when no scheme is used (workload-1). As can be seen, Scheme-2 reduces the idleness of our banks (resulting in an overall system performance improvement of more than 5%). Figure 3.14 plots dynamic reduction in bank idleness over the course of execution for one of our workloads (workload-1).

Results for a 16-core system: We also ran our experiments in a smaller system
with 16 cores connected by a $4 \times 4$ mesh-based NoC. In this configuration, two memory controllers with the same parameters as in Table 6.2 are attached to two opposite corners of the mesh.

Figure 3.15 plots the speedup values achieved by our proposed scheme when our workloads are executed on this smaller system. We picked the first half of the applications in each workload shown in Table 5.3 for this experiment (for the mixed workload, the first half of the memory intensive and memory non-intensive applications are selected). Averaged over all the workloads, the speedups achieved in these experiments are about 8%, 10% and 5% for the mixed, memory intensive and memory non-intensive workloads, respectively.

A comparison between Figures 3.11 and 3.15 reveals that the speedup values for the 32-core system are generally higher than those for the 16-core system. This is because as the network size increases, the network delay contributes more to the round-trip latencies of the off-chip accesses and, as a result, the impact of the network prioritization is amplified.

**Impact of the threshold value in Scheme-1:** As discussed earlier, to determine whether a response message is late or not, after a request is serviced by the corresponding memory controller, the so-far delay of the request is compared against a threshold and if it is larger than this threshold value, the corresponding memory access is considered to be “late”. Recall also that the default value for this threshold is $1.2 \times \text{Avg}_{\text{delay}}$. Note that $\text{Avg}_{\text{delay}}$ is the average round-trip latency of the off-chip requests that belong to the same application (issued by the same core) and is computed dynamically by the source core. To study the impact of the threshold values on the achieved speedups, we performed experiments for two
other threshold values: \(1.1 \times \text{Avg}_{\text{delay}}\) and \(1.4 \times \text{Avg}_{\text{delay}}\). Figure 3.16(a) plots the speedup values achieved for the three cases when workloads 1 through 6 are used. As can be observed, when the threshold is set to a larger value, the speedup values reduce since fewer messages are considered to be late in this case, and prioritized in the network. However, reducing the threshold value may not help performance since when the number of late messages in the network increases, prioritizing too many messages in the network hurts other messages, leading eventually to network congestion and performance degradation.

**Impact of the T value (history length):** In Scheme-2, the decision of whether a request message is destined for a idle bank or not is made based on the local information. When an L2 miss occurs, if over the last \(T\) cycles, the number of requests sent to the same bank is less than a threshold \((th)\), the request is given high priority (the default values for \(T\) and \(th\) are 200 cycles and 1, respectively). Figure 3.16(b) plots the speedup values for \(T = 100, 200,\) and 400. As can be observed, for \(T = 400\) the speedup values are not as high as the case where \(T=200\) since the number of late requests decreases. However, reducing the \(T\) length \((T = 100)\) does not necessarily increase the speedups either (see workload-2 and workload-4) due to the imprecision in finding the idle banks and the slow down imposed on the other messages in the network.

![Graph](image)

(a) Impact of the threshold value in Scheme-1.
(b) Impact of the history length in Scheme-2.
(c) Impact of the number of memory controllers.

**Figure 3.16.** Results from our sensitivity experiments (32 cores).

**Impact of the number of memory controllers:** Finally, Figure 3.16(c) plots the performance improvements for our mixed workloads when there are two and four memory controllers in the system. One can observe that, the performance improvement is slightly increased when there are fewer number of memory controllers in the system. The reason for this can be explained as follows. When there are
fewer number of memory controllers in the system, due to the pressure increase on the bank queues, there will be more critical and late accesses in the system that can be enhanced by Scheme-1, and this results in higher performance. Note also that, although the benefits form Scheme-2 may reduce with fewer number of memory controllers (since there will be less idle banks), due to the improvement achieved by Scheme-1, the overall improvement (Scheme-1 + Scheme-2) is slightly better (see Figure 3.16(c)). However, the performance improvement is reduced for w-2 and w-3 due to the lower improvements achieved by Scheme-2.

**Sensitivity to the structure of the routers:** Although many schemes have been proposed by prior research to reduce message latency and contention in the on-chip networks, they do not completely eliminate the network contention and we are far from achieving the ideal performance. Therefore, our proposed schemes can be employed with the presence of the other proposed strategies to speed up the critical off-chip messages by prioritizing them in the network. One of the techniques to reduce the network latency is to employ routers with the fewer number of pipeline stages. As given in Table 6.2, in our default configuration, the routers are implemented as five-stage pipelines. However, the number of stages can be reduced to two (as discussed in Section 3.3.3, but here, all the flits have the opportunity to traverse each router in two cycles if there is no contention). Figure 3.17 compares the performance improvement achieved by our proposed schemes (for w-1 to w-6) when the routers have two and five pipeline stages. As can be observed from this plot, the performance is still improved up to 10% for the 2-stage pipeline. However, the improvement is 25-40% lower compared to the 5-stage pipeline case. This is because the percentage of the network latency that can be reduced (for the selected
messages) by the network prioritization decreases for the 2-stage pipeline case (note that in this case no router bypassing is employed and the critical messages are prioritized in the message arbitration done in the routers). However, other techniques such as express channels proposed in [58] can be employed to increase the latency savings.

3.5 Related Work

Main memory accesses have been identified as a critical bottleneck in both multi-programmed and multithreaded workloads [43, 44, 45]. Prior studies [62, 63, 64] have shown that packet latency variance can be reduced by using age-based prioritization at the network routers. These schemes added a new field to each packet to track its current network latency and assigned a higher priority in the router to packets that suffered higher network latencies during switch allocation. Lee et al. [65] proposed the use of a different metric with age-based prioritization, where the hop count of a packet is used to assign its network priority. However, livelocks can occur when such a static metric is used for prioritization, and hence, this approach required the implementation of complex probabilistic priority arbiters to prevent livelocks. In contrast to these age-based schemes, in our work, we compare the latency of a request coming from a core only against the average latency of the requests from the same core. Therefore, priorities for each core are calculated independently, which enables per-core optimization of the memory access latency variance.

Two recent approaches to network prioritization-based latency optimization are proposed by Das et al. [56, 43]. In [56], packets are prioritized based on the types of the co-running applications and in [43], the slack of a network packet is defined as the number of cycles the packet can be delayed without affecting the execution time of the application. Although these works also employ network prioritization, they do not have a detailed memory model (they assume fixed delay). In contrast, we have a detailed memory model (with queuing and row-buffer modeling) and use accurate so-far delay information dynamically updated during execution. Further, in [56, 43], latency/slack calculation is done at the core side based on indirect parameters such as hit/miss status of the packet and number of hops. However, it
becomes extremely difficult to accurately predict latency/slack at the cores when multiple requests from different cores compete for the same MCs. Lastly, these prior works do not consider memory bank idleness. Consequently, our schemes are different from these prior works.

As mentioned earlier, in our work we use an S-NUCA [42] based cache space management. In [66], a novel scheme called Reactive NUCA is proposed for data placement in the distributed caches. In the proposed method, data is placed in the local L2 slices for the private data patterns seen in multi-programmed workloads. This scheme reduces the network latency of the memory accesses by eliminating paths #1 (L1 to L2) and #5 (L2 to L1) in Figure 3.2. However, it does not completely remove the network latency due to paths #2 (L2 to MC1) and #4 (MC1 to L2) in Figure 3.2, and our scheme will still provide performance benefits (at a lower level).

Another important component of off-chip memory access latency variance is the variations in memory queuing latency. In [14, 15], the authors targeted at reducing the variance of memory access latencies at the memory controller. These approaches coordinate multiple on-chip memory controllers to equalize the performance penalty suffered by each application due to interferences coming from other applications. The effectiveness of these schemes on a large multicore with high network latencies is unclear. On a large network, communicating information across memory controllers itself can easily have a very high latency, thus increasing the reaction time to dynamic changes in execution behavior.

Clearly, directly improving off-chip memory access latencies can also improve overall system performance. Memory queuing latencies can be improved by finding a better scheduling of memory requests in the memory bank queues [55, 44], designing lower latency routers [67, 68, 69], or employing better flow-control [70, 58]. We would like to note that the network prioritization schemes we propose in this work can be used with any memory scheduling algorithm, router microarchitecture, or flow control scheme. For instance, while our Scheme-1 is orthogonal to the memory scheduling scheme employed (as it optimizes network latency component after the memory), the increase in the average number of entries in memory bank queues due to our Scheme-2 can enable these complex memory schedulers to find even better schedules as they can see a larger window of memory requests. We
postpone the investigation of such interactions to a future study.

### 3.6 Conclusion

We proposed two network prioritization schemes that reduce the *end-to-end memory access latency* in multicores. Our first scheme addresses the latency variance in memory accesses that belong to the same application, and expedites memory response messages that experience high latencies by prioritizing them. This reduces the number of off-chip requests with high latencies and achieves a more uniform memory latency pattern. Our second scheme improves memory performance by prioritizing memory request messages that are destined for idle banks over other requests. This optimization increases bank level parallelism and improves memory utilization. Through an extensive evaluation of our schemes on a 4 × 8 mesh-based multicore, we show that our first scheme is very effective in reducing the number of off-chip requests that experience very high latencies and achieves 11%, 6%, and 10% performance improvement on memory intensive, memory non-intensive, and mixed multiprogrammed workloads, respectively. When both our schemes used together, these improvements jump to 15%, 10% and 13%, in the same order.
4.1 Introduction

A recent network based multicore from Intel [76] reports 31% and 79% non-core power contribution in full power and low power operating modes, respectively. Despite the fact that the power consumptions of caches and NoC are on par with that of the cores, to the best of our knowledge, there has been no prior work on distributing the power budget among all these resources to maximize performance, while maintaining the same overall power budget of the system as a whole. We believe significant performance improvements can be achieved by distributing the overall power intelligently among the resources based on the workload’s sensitivity to core/NoC frequency and cache allocation.

To illustrate the key insight leveraged by this chapter, that different distributions of the overall system power can lead to different system performances (despite keeping the overall power budget the same), we performed a simple experiment. We allocated different percentages of a fixed chip-wide power cap (90W) to different resources in the system. The core frequency is tuned so that the overall power allocation to the cores is respected. Similarly, the number of active cache ways
in a set-associative last-level cache (LLC) are tuned such that the overall cache power remains within the allocated budget, and the NoC frequency is set such that the NoC power budget is also respected. The performance of a multiprogrammed workload mix of 16 applications on a 16-core $4 \times 4$ system as measured by the normalized weighted speedup under various distributions of the (same) overall power budget is shown in Figure 4.1. From this figure, we can see that different power budget distributions (x-axis) result in significantly different system performance, under the same power budget. Note that, the results presented in Figure 4.1 are only with a static distribution of the overall power among different resources, which does not react to changes in the workload behavior. Further, any changes in the power allocation to a specific resource requires tracking the actual (now-limited) resource allocation to applications over time. Therefore, a dynamic distribution of the power budget can potentially achieve much higher overall performance and a mechanism to track the allocated power among applications can also ensure that the power budget is always respected despite changing workload behavior.\footnote{Note that, while (for a given workload/configuration), one may find a static partitioning that comes close to the dynamic one, one would need an exhaustive search to achieve this (which is not possible in practice). Further, in other cases, one may not even find a comparable static partitioning (since static partitionings do not consider dynamic modulations in workload behavior).}

To this end, in this chapter, we propose a power budget distribution mechanism (called PEPON) that distributes the chip-wide power budget among various types of critical resources like the cores, caches, and NoC to maximize performance, while respecting the allocated budget in the corresponding resources, thereby respecting the overall power budget. Specifically, the main contributions of this work can be summarized as follows:
We propose a novel two-level power distribution scheme that distributes a specified power budget among different resources by carefully managing the allocated power to attain best possible performance for a given workload. This scheme employs different models at different power distribution levels and uses feedback controllers to enforce the core and NoC power allocations.

Extensive evaluation of the proposed technique using a full system simulation and detailed power/performance models shows that carefully distributing the power budget among all the on-chip resources not only ensures that the power cap on the processor is respected, but also optimizes the system performance. Our experimental analysis indicates that overall workload performance can be improved by as much as 29%, as compared to the case where no power distribution is implemented. Further, with respect to a recently-proposed power budgeting scheme that considers redistribution of only core power [72], our scheme achieves, on average, about 13% performance improvement –under the same chip-wide power budget.

4.2 Target Multicore System

Figure 4.2 shows an NoC-based multicore architecture. It is a shared-memory based two-dimensional mesh structure, where each node contains a processor core, private L1 data or instruction caches, a shared L2 cache bank, and a router through which it gets connected to the neighboring nodes. Although all nodes share the same logical on-chip L2 cache space, data access latencies depend on the distance between the requesting core and the cache bank that holds the requested data. The set of links used during an access to a data element in some other node’s L2 is determined by the routing policy, which is XY-routing [78] in this work.
One of the critical issues in designing such multicore architectures is power consumption, which mainly comes from three sources: cores, on-chip cache components (shared L2 banks), and the NoC. To reduce the power consumption of cores, prior dynamic voltage/frequency scaling (DVFS) based methods [79, 76, 80] proposed in the literature can be employed. The idea of DVFS is to exploit the variance in processor utilization by lowering the voltage/frequency when the processor is lightly loaded and increasing the voltage/frequency when the processor is highly utilized. Like processor cores, there is also a wide variance in cache line utilization in on-chip memories as well as link utilization in NoC-based multicore architectures, depending on applications’ communication/data sharing patterns. In fact, one can obtain up to $10 \times$ power savings, by accurately shutting down unused cache ways or lowering frequency and voltage of links when link usage is decreased [81]. We exploit DVFS for varying the power allocations of cores and the NoC.

### 4.3 Our Proposed Approach

#### 4.3.1 High-Level View of PEPON

Targeting the multicore architecture shown in Figure 4.2, our goal in this work is to maximize workload performance under a chip-wide power budget constraint. More specifically, given a multicore-wide power budget, we want to distribute/redistribute it across different hardware components such that (i) the given power budget is not exceeded and (ii) performance is maximized. Consequently, we are not interested in solutions that maximize workload performance but violate the specified power budget. Further, among all solutions that satisfy the power budget constraint, we are interested in finding the one that maximizes performance.

Figure 4.3(a) illustrates the high level operation of our proposed two-level power budget distribution strategy (PEPON). In the first level of this strategy, the overall power budget is partitioned among cores, NoC and L2 caches. In the second level, the power budget assigned to cores is further partitioned among individual cores and, similarly, the power budget assigned to L2 caches is further partitioned across individual L2 caches. While it is also possible to divide the NoC into voltage islands [82] and distribute the power allocated to the NoC across these islands, we did not
pursue that direction in this work because such a partitioning can have a non-deterministic impact on different applications. Instead, in this work, we treated the NoC as a single monolithic entity, all components of which have the same voltage/frequency level at a given time (i.e., when voltage/frequency scaling is applied, it is applied to the entire NoC).

At each level of this power distribution strategy, a different approach is employed. Specifically, for the first-level distribution, we adopt a regression-based performance model that guides us to decide the most performance-efficient distribution of power. For the second-level distribution, on the other hand, we use different strategies for caches and cores. The power budget assigned to cores is distributed using a control theoretic approach. On the other hand, for power budget distribution across different L2 banks, we employ a utility-based model. The technical details of these models as well as the details of our two-level power distribution strategy and our power control knobs are described next.

### 4.3.2 Design Details of PEPON

For the first level power distribution, the system power budget ($P_{\text{system}}$) is partitioned among different types of components, namely, cores ($P_{\text{cores}}$), cache banks ($P_{\text{caches}}$), and NoC ($P_{\text{noc}}$). That is, we have: $P_{\text{system}} = P_{\text{cores}} + P_{\text{caches}} + P_{\text{noc}}$. At the second level, $P_{\text{cores}}$ is distributed among the cores and $P_{\text{caches}}$ is distributed among the cache banks. Both the first and the second-level distributions are performed at every $x$ million cycles. Further, each component of interest (each core, cache bank and NoC) has a controller associated with it, which tracks the allocated budget.

As illustrated in Figure 4.3(b), the longer epochs ($x$ million cycles) are divided into several sub-epochs, and the controller associated with each component is invoked
Table 4.1. A sample window (table) of the power budget allocations and corresponding performance numbers (first column).

every $k$ million cycles (sub-epochs) to track its power budget, which is determined at every $x$ million cycles. While we performed experiments with different values of $x$ and $k$ parameters, the default values used in most of our experiments are 100 and 20, respectively.

4.3.2.1 First-Level Power Distribution

At this level, we employ a regression-based estimator:

$$\text{Perf} = \epsilon + a_1 \times P_{\text{cores}} + a_2 \times P_{\text{caches}} + a_3 \times P_{\text{noc}}.$$  \hspace{2cm} (4.1)

In this expression, $\text{Perf}$ is the overall workload performance, $a_i$ ($1 \leq i \leq 3$) represent model coefficients, and $\epsilon$ captures the model error value. This model gives the overall performance ($\text{Perf}$) expected to be achieved over an epoch when the total power budget is partitioned into $P_{\text{cores}}$, $P_{\text{caches}}$ and $P_{\text{noc}}$ values in that epoch.

In our model, we define the overall workload performance ($\text{Perf}$) over each time interval as $\sum_i \frac{\text{IPS}_i^k}{\text{Avg} \text{IPS}_i}$, where $\text{IPS}_i^k$ is the instruction per second (IPS) of the $i$th application over the $k$th time interval and $\text{Avg} \text{IPS}_i$ is the average measured IPS of that application when it is executed without any power constraint. $\text{Avg} \text{IPS}_i$ can be obtained for each application by running that application on a core with the highest possible frequency and without imposing any power budget constraint.

At each time interval, first, the coefficients are updated based on the measured performance and allocated power budgets in the previous epoch and then our scheme increases the power budget of the component with the largest coefficient and reduces the power budget of the component with the smallest coefficient. As an example, if $a_1 > a_2 > a_3$, then $P_{\text{cores}}$ and $P_{\text{noc}}$ are increased by $\delta$ and reduced by $\delta - \epsilon_1$, respectively ($\delta$ is an increment/decrement unit and $\epsilon_1$ is a small number and helps the regression to be computable). To update the model, we keep a window (table) with four entries that store the power budget allocations and corresponding measured performance numbers for the four recent epochs. Each entry in this
table is in the form of \((\text{Perf}, \text{P}_{\text{cores}}, \text{P}_{\text{caches}}, \text{P}_{\text{noc}})\); an example table is given in Table 4.1. At each budget distribution epoch (the longer of the intervals shown in Figure 4.3(b)), first, the oldest entry in the table is replaced with the new entry \((\text{Perf}^k, \text{P}_{\text{cores}}^k, \text{P}_{\text{caches}}^k, \text{P}_{\text{noc}}^k)\), where \text{Perf}^k is the measured overall system performance over the last epoch and \text{P}_{\text{cores}}^k, \text{P}_{\text{caches}}^k and \text{P}_{\text{noc}}^k are the recent power budgets assigned to the cores, caches and NoC, respectively. The data stored in Table 4.1 and the model coefficients can be stacked together and written as follows:

\[
PF = P \times A + \epsilon, \quad \text{where}
\[
PF = \begin{pmatrix}
\text{Perf}^k \\
\text{Perf}^{k-1} \\
\text{Perf}^{k-2} \\
\text{Perf}^{k-3}
\end{pmatrix}, \quad
P = \begin{pmatrix}
\text{P}_{\text{cores}}^k \\
\text{P}_{\text{caches}}^k \\
\text{P}_{\text{noc}}^k \\
\text{P}_{\text{cores}}^{k-1} \\
\text{P}_{\text{caches}}^{k-1} \\
\text{P}_{\text{noc}}^{k-1} \\
\text{P}_{\text{cores}}^{k-2} \\
\text{P}_{\text{caches}}^{k-2} \\
\text{P}_{\text{noc}}^{k-2} \\
\text{P}_{\text{cores}}^{k-3} \\
\text{P}_{\text{caches}}^{k-3} \\
\text{P}_{\text{noc}}^{k-3}
\end{pmatrix}, \quad
A = \begin{pmatrix}
a_1 \\
a_2 \\
a_3
\end{pmatrix}, \quad
\epsilon = \begin{pmatrix}
\epsilon_1 \\
\epsilon_2 \\
\epsilon_3
\end{pmatrix}.
\]

After updating the regression history, we use the least square method to estimate \(a_1\), \(a_2\) and \(a_3\) values. In the least square method, regression coefficients are computed as: \(A = (P^T \times P)^{-1} P^T \times PF\). PEPON then increases the power budget with the largest coefficient and reduces the power budget with the smallest coefficient, since these values reflect the impact of varying the power budget of the corresponding components on the overall performance improvement.

### 4.3.2.2 Second-Level Power Distribution

At the second level, we employ different power distribution strategies for cores and caches. To distribute \(P_{\text{cores}}\) across the different cores in the multicore system, we use the approach proposed in [72]. Specifically, we allocate a power budget of \(P_i\) to core \(i\) based on its \(\text{IPS}_{\text{watt}}\) (performance per watt) over the last epoch (\(\text{IPS}_{\text{watt}} = \frac{\text{Instruction per second}}{\text{power consumption}}\)). The allocated budgets are proportional to this parameter. In mathematical terms, if \(\text{IPS}_i\) and \(\text{Pow}_i\) are the measured IPS and the estimated power consumption of the application running on core \(i\) over the last epoch, and
$Avg_{IPS_i}$ is the average IPS of that application when it is running without any power constraint, then the power budget allocated to core $i$ would be computed as:

$$Power\_Budget_{core}(i) = \frac{IPS_i}{Avg_{IPS_i}} \times P_{cores}.$$ 

Note that, in this equation, $\frac{IPS_i}{Avg_{IPS_i}}$ is used instead of $IPS_i$ since some applications intrinsically has low IPS numbers even when they are executed without any power budget constraint. We can obtain $Avg_{IPS}$ for each application by running that application on a core with the frequency set to the highest possible value and without imposing any power budget constraint. Note that, this profiling is performed once for each application (not for workloads). Previous work such as [83], also uses the average IPS value as a baseline. In Section 7.4.2, we present experimental results to show that partitioning power only at the core level (as in the case of [72]) may not be sufficient to maximize performance. Once the power budget of each core is determined, the controller associated with each core attempts to track the budget by adjusting frequency/voltage of the core. As stated before, we use DVFS as our knob to control power consumption of the cores. More details are given later in this section.

To distribute $P_{caches}$ among different cache banks, we employ a utility-based strategy. More specifically, the power share of each bank is proportional to its utility over the last epoch. Our goal is to allocate more power budget to the cache banks which are likely to bring more benefits towards the overall system performance. Different strategies can be used to compute utility of cache banks. As an example, the utility of a bank can be computed as the total number of accesses to that bank over the last epoch. However, this would not be a good metric for power budget partitioning since those accesses may have a small cache footprint and it is not probably beneficial to allocate more power budget to the bank with higher utility (note that, more budget means larger size for the banks). The approach adopted in this work is based on cache footprints. In this approach, we consider the number of unique cache lines that are hits in a cache bank over the last epoch as the utility of that cache bank. To implement this scheme, a small bit vector is associated with each cache bank. This bit vector is used as a hash table. Once a miss occurs in a cache bank, a bit in its bit vector is set to ‘1’. This
bit is indexed using the tag of the new data as the key (the key is hashed using an efficient hardware function). At the same time, the corresponding bit of the evicted data is set to ‘0’. Consequently, within a period of time, the ‘1’s in the bit vector of a cache bank indicate the number of unique active cache lines that have been accessed. Since a cache line may not be accessed over a long time after the first access, we reset the bit vectors at the beginning of each time interval. In PEPON, the power budget share of each cache bank is proportional to its utility (i.e., the number of ‘1’s in the bit vector) over the last epoch.

Note that, in our system, each core runs a single-application, whereas NoC and cache-banks are shared. Therefore, a pure application-centric budget-partitioning is not possible for an NoC or caches. This is why we take a component-centric partitioning strategy instead.

4.3.2.3 Design of the Power Controllers

In this subsection, we explain how the power budgets assigned by PEPON are tracked by the controllers associated with different hardware components.

Core and NoC Controllers. As stated before, we use the frequency/voltage of the cores and the NoC as our knob to control power consumption of these modules (we employ DVFS). Tables 4.2 and 4.3 give the frequency/voltage pairs we consider in our DVFS scheme. A simple $P$ (proportional) feedback controller is configured and associated with each core as well as the NoC to modulate the frequency levels, with the goal of tracking the allocated power budgets. Figure 4.4 illustrates the high level view of how our controllers operate. We employ $P$ controllers in PEPON due to two main reasons: (i) they are relatively easy to design and implement with low overhead since the control output is the control error multiplied by a constant value, and (ii) they are fast to take the system output to the desired value (we need this property in PEPON, since the controllers have only limited number of time intervals to reach the desired power budgets). In this section, based on our system model, we show that employing only the $P$ component of a PID controller provides a fast and stable controller with zero steady error$^3$.

$^2$The $P$ controller [26] is a type of controller in which the output of the controller is proportional to the error value, which is the difference between the target value and the current system output.

$^3$Although most our experiments are performed using a $P$ controller, we later report results with a PID controller as well.
At each epoch, the controller compares the current power budget and the measured power over the last epoch and, based on the result of that comparison, adjusts (increases or decreases) the frequency level to minimize the measured error value (which is the difference between the power budget and the actual power). Each controller in PEPON implements the following rule to modulate the frequency level:

\[
f(T + 1) = f(T) + K \times (P_{\text{budget}} - P_{\text{current}}),
\]

where \(f(T)\) is the frequency at \(T^{th}\) time interval, \(K\) is a constant value, and \(P_{\text{budget}}\) and \(P_{\text{current}}\) are the specified power budget and the current power, respectively. The controller first computes \(f(T + 1)\) and then sets the current frequency level to the level which is the closest one to \(f(T+1)\). The controller ensures that its output be one of the available frequency levels in Tables 4.2 and 4.3. As can be observed from this control rule, if the measured power is greater than the reference power, the controller reduces the frequency level to reduce power consumption over the next epoch.

The next step to configure the controller is to determine the value of \(K\). This value is computed based on the system power behavior (core/NoC) when the frequency level is varied. As shown in [73], the impact of varying the frequency on power consumption can be estimated linearly as follows:

\[
P_2 = P_1 + a \times \Delta F,
\]

where \(P_1\) is the power being consumed at the old frequency and \(P_2\) is the power consumption after the frequency is increased by \(\Delta F\). The value of \(a\) can be estimated offline by performing experiments with different applications and frequencies and averaging over the obtained values for \(a\). By combining Equations (4.2) and (4.3), we observe that, if we set the \(K\) value to \(\frac{1}{a}\) in the controller, the frequency is changed towards reaching the reference (target) power budget (the value of \(K\) is

| \(0.25, 7\) | \(0.41, 0.8\) | \(0.60, 0.9\) | \(0.65, 0.94\) | \(0.76, 1\) |
| \(0.84, 1.05\) | \(0.95, 1.1\) | \(1.0, 1.14\) | \(1.23, 1.28\) | \(1.33, 1.34\) |

Table 4.2. \{frequency (GHz), voltage (V)\} levels considered for the cores.

| \(0.06, 0.55\) | \(0.55, 0.7\) | \(0.88, 0.8\) | \(1.24, 0.91\) | \(1.34, 0.94\) |
| \(1.54, 1\) | \(1.7, 1.05\) | \(2.0, 1.14\) | \(2.23, 1.21\) | \(2.46, 1.28\) |

Table 4.3. \{frequency (GHz), voltage (V)\} levels considered for the NoC.
Figure 4.4. Feedback control loop for regulating the core/NoC power.

Figure 4.5. Static power versus the number of active (L2) cache ways.

60M in our scheme).

Control System Analysis. One of the main methods to characterize a control system is to analyze it in the frequency-domain [26]. In this method, the system is represented by a transfer function that shows the correlation between its input and output in frequency-domain (z-transform is used for discrete time systems). Our system, which is modeled by Equation (4.3), can be represented as $Y(z) = X(z) / z - 1$, where $Y(z)$ is the power consumption of the system and $X(z)$ is the variation in the operating frequency in $z$-domain. Since we employ a $P$ controller, our closed loop transfer function can be obtained as $H(z) = Y(z) / R(z) = \frac{K \times a}{z - 1 + K \times a}$, where $R(z)$ is the power budget target.

In formal control theory, a control system is stable and tracks the target over the steady state if the poles of the closed loop transfer function is placed inside the unit circle in $z$-plane. The poles of the transfer function are obtained by solving the characteristic equation (setting the nominator of the transfer function to zero). Therefore, as can be observed, our frequency control system has a pole at $z = 1 - K \times a$. Further, as mentioned before, the value of the $K$ parameter is set to $\frac{1}{a}$, which ensures the stability of the system.

Steady state error is another important metric for a control system. This error is defined as the difference between the input reference and the actual system output when the controller reaches its steady state. It can be shown that the steady state
error equals \( \lim_{z \to 0} \frac{1}{1 + H(z)} \) for step reference inputs, where \( H(z) \) is the closed loop transfer function. In our control system, the reference inputs are step signals since they vary between two different frequency levels instantly. If we replace \( H(z) \) with our system transfer function, one can observe that the steady error would be zero based on the mathematical analysis. Note that, although our analysis shows an accurate control system, there may still be some inaccuracies in tracking the reference targets as will be discussed in our experimental evaluations. These imprecisions can occur because we employ a linear estimation for modeling the system (see Equation(4.3)) and it does not ideally capture the dynamic behavior of co-runner applications.

**Cache Bank Controller.** As technology scales down, static (leakage) power of caches becomes an important part of the overall power consumption. The static power highly depends on the size of the caches. As an example, Figure 4.5 plots the leakage power of a 512KB, 16-way set associative cache manufactured with the 45nm technology, when the number of “active” cache ways varies. This figure clearly indicates that there is a *linear correlation* between the number of active cache ways and the static power consumption. Motivated by this observation, in PEPON, power is controlled in cache banks by turning ‘ON’ or ‘OFF’ the cache ways. Specifically, at each epoch, the controller associated with each cache bank compares the current power and the reference power budget and, using the data in Figure 4.5, decides to turn on or off a number of cache ways, to minimize the difference between the actual and reference power.

### 4.3.2.4 Implementation Overheads

PEPON is implemented in software at the OS level with support from hardware. One of the overheads imposed by PEPON is the process of computing the individual power budgets. This includes simple mathematical and matrix operations to partition power at the first and second levels. These overheads are minimal, since the scheme is implemented in software and executed only in every 100-million cycles. In comparison, the controllers are invoked with a higher frequency to track the power budgets. It should be noted, however, that our controllers perform a simple operation (subtraction and multiplication) since they are implemented as \( P \) controllers. Therefore, the additional overheads caused by our controllers are also
not very high. The power and performance numbers presented in the next section include all the overheads incurred by PEPON. In addition to these software-level overheads, PEPON assumes that the underlying system has support for DVFS, which is a standard in almost all processors today. We make the same assumption for the DVFS overheads as in [72] (20us DVFS actuation overhead). If the voltage/frequency values are enforced every 20-million cycles (our default value) and if the system operates at 500MHz, the resulting DVFS overhead (20us) would be about 0.05%. In addition, PEPON also assumes that the processor has support for software level control for reconfiguring (turning on/off) cache ways in the L2 cache and tuning the operating frequency of the NoC.

4.4 Experimental Evaluation

In this section, we experimentally evaluate PEPON, and compare it against alternate power budgeting strategies quantitatively. We use SIMICS [29], which is a full-system simulator for multicores, as our simulation framework. For our experiments, SIMICS is enhanced with detailed latency models as well as power models from CACTI [84], Orion [85] and Wattch [86], to estimate the power consumption of different components in our target multicore architecture.

4.4.1 Power Estimation Models

**Power model for cores.** We use the same model as in [73] to estimate power consumption of the cores. The power consumed by a core depends almost linearly on the utility of the core. In other words, once an application starts its execution, the average power consumption of the core can be estimated based on this equation: $$P_i = k_0 \times U_i + k_1,$$ where $P_i$ is the power consumption of core $i$, $U_i$ is its utilization, and $k_0$ and $k_1$ are constant values. Note that the values of $k_0$ and $k_1$ may be different for different applications. In this work, we estimated values of these constants by running each application independently and using Wattch [86] for monitoring power consumption over execution.

**Power model for caches.** We use numbers obtained from CACTI [84] to estimate power consumption of the caches. Power of each cache bank can be estimated as:
\[
\text{Power} = \text{Leakage Power} + (\text{read power} \times \# \text{ of reads}) + \\
(\text{write power} \times \# \text{ of writes}),
\]

where the leakage power linearly depends on the cache size. Since, in our scheme, we use the number of active cache ways as the knob to control the power consumption of a cache bank, the leakage power can be estimated considering the active cache size at each time interval over the execution.

**Power model for the NoC.** The average power consumption of the NoC, over a time interval \(T\), is estimated as follows: 

\[
P_{\text{noc}} = P_{\text{dynamic}} + P_{\text{static}} = \frac{NH_{\text{noc}} \times E_{\text{noc}}}{T} + P_{\text{static}},
\]

where \(NH_{\text{noc}}\) is the total number of hops that all the packets in the network traverse over \(T\) and \(E_{\text{noc}}\) is the energy per hop. In this expression, “energy per hop” is the amount of energy consumed when a packet passes through a router. We used the energy values obtained from Orion [85] to estimate the total power consumption of the NoC.

<table>
<thead>
<tr>
<th>Processors</th>
<th>16 cores with private L1 data and instruction caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoC Architecture</td>
<td>4 × 4</td>
</tr>
<tr>
<td>Processor Model</td>
<td>4-way issue superscalar</td>
</tr>
<tr>
<td>Private L1 Data and Instr Caches</td>
<td>Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency</td>
</tr>
<tr>
<td>Number of L2 Cache Banks</td>
<td>16 (distributed over the network)</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>64 bytes block size, 10 cycle access latency</td>
</tr>
<tr>
<td>L2 Cache Bank Size</td>
<td>512KB per core</td>
</tr>
<tr>
<td>Memory</td>
<td>4GB, 200 cycle off-chip access latency</td>
</tr>
<tr>
<td>Power Budget Control Enforcement Interval</td>
<td>100 Million cycles</td>
</tr>
<tr>
<td>Tracking Control Enforcement Interval</td>
<td>20 Million cycles</td>
</tr>
</tbody>
</table>

Table 4.4. Baseline configuration.

Table 6.2 gives the baseline configuration we use in our experimental evaluation. In our baseline configuration, the power budget allocated to each component by PEPON is computed and enforced every 100-million cycles. Further, the controller assigned to each component has the opportunity to track its power target over five intervals with 20-million cycles duration. We form six application mixes (workloads) using the SPEC CPU 2006 benchmark suite [57] for our experimental evaluation. Each of our application mixes consists of 16 applications. We consider one-to-one mapping, that is, in each experiment, 16 applications of a mix are mapped and executed on 16 different cores in our system (one application per core). These six workloads are given in Table 6.3. In forming these workloads, we
tried to have diversity in terms of memory access behavior. For each application, we skip the first 1-billion cycles for warm-up, and then the simulation continues with the next 10-billion cycles (100 power budget enforcement intervals).

<table>
<thead>
<tr>
<th>Workload</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mix-1</td>
<td>mcf, soplex, libquantum, GemsFDTD, lbm, leslie3d, mcf, sphinx3, zalancbnmk, cactusADM, mcf, soplex, libquantum, GemsFDTD, lbm, leslie3d</td>
</tr>
<tr>
<td>Mix-2</td>
<td>gobmk, sjeng, gcc, wrf, dealII, namd, perlbench, calculix, tonto, povray, povray, tonto, calculix, perlbench, namd, dealII</td>
</tr>
<tr>
<td>Mix-3</td>
<td>mcf, soplex, libquantum, GemsFDTD, lbm, leslie3d, mcf, sphinx3, zalancbnmk, cactusADM, omnetpp, astar, hminer, kbp2, h264ref, gromacs</td>
</tr>
<tr>
<td>Mix-4</td>
<td>gobmk, sjeng, gcc, wrf, dealII, namd, perlbench, calculix, tonto, povray, omnetpp, astar, hminer, kbp2, h264ref, gromacs</td>
</tr>
<tr>
<td>Mix-5</td>
<td>mcf, soplex, libquantum, GemsFDTD, lbm, leslie3d, mcf, sphinx3, zalancbnmk, cactusADM, gobmk, sjeng, gcc, wrf, dealIII, namd</td>
</tr>
<tr>
<td>Mix-6</td>
<td>gobmk, sjeng, gcc, wrf, dealII, namd, perlbench, calculix, tonto, povray, cactusADM, zalancbnmk, sphinx3, mcf, leslie3d, mcf,</td>
</tr>
</tbody>
</table>

Table 4.5. Various mixes of applications considered for our multiprogrammed workloads.

4.4.1.1 Schemes for Performance Evaluation

Table 4.6 lists the five different schemes we consider in our experimental evaluation to compare with PEPON. In this table, “dynamic” indicates that the corresponding power budget is varied over the execution. As can be observed from Table 4.6, all the entries are shown as “dynamic” for PEPON, indicating that the power budgets are varied over the execution in both the first and second levels of the budget partitioning. However, in other schemes (Schemes 1 through 4), the power budgets are fixed at the first level of partitioning during runtime (i.e., $P_{\text{cores}}$, $P_{\text{caches}}$ and $P_{\text{noc}}$ are fixed as given in Table 4.6). This table also gives the actual power budget values (for the high and low power modes) assigned when using the different partitioning schemes. However, in Scheme 1, the total power budget of the cores and the cache banks are still partitioned among the individual components dynamically. All the power budgets of different components are fixed in Scheme 2, and the total power budget of the cores and the caches are evenly distributed among them. In Schemes 3 and 4, on the other hand, only one of the power budget distribution mechanisms is enabled at the second level (for cores in Scheme 3 and for caches in Scheme 4). Note that, Scheme 3 mimics a recently-proposed approach to power budget partitioning in multicore processors [72]. In addition to these schemes, we also implemented another scheme in which at the first level, the total power budget is distributed dynamically.
and at the second one, the power budgets of the cores and cache banks – from the first level – are partitioned statistically (evenly) among the individual components (cores and caches). We observed that PEPON improves the performance over this last scheme by about 8% on average. The detailed results for this scheme is omitted due to space concerns.

### 4.4.2 Results

#### Scheme

<table>
<thead>
<tr>
<th>Scheme</th>
<th>(\sum \text{CorePowerBudget}(i))</th>
<th>(\text{CorePowerBudget}(i))</th>
<th>(\sum \text{BankPowerBudget}(i))</th>
<th>(\text{BankPowerBudget}(i))</th>
<th>(\text{NoCPowerBudget})</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEPON</td>
<td>dynamic</td>
<td>dynamic</td>
<td>dynamic</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
<tr>
<td>Scheme1</td>
<td>static (29 watt – 68 watt)</td>
<td>dynamic</td>
<td>static (5 watt – 10 watt)</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
<tr>
<td>Scheme2</td>
<td>static (29 watt – 68 watt)</td>
<td>static (29/16 watt – 68/16 watt)</td>
<td>static (5 watt – 10 watt)</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
<tr>
<td>Scheme3</td>
<td>static (29 watt – 68 watt)</td>
<td>dynamic</td>
<td>static (5 watt – 10 watt)</td>
<td>static (5 watt – 10 watt)</td>
<td>dynamic</td>
</tr>
<tr>
<td>Scheme4</td>
<td>static (29 watt – 68 watt)</td>
<td>static (29/16 watt – 68/16 watt)</td>
<td>static (5 watt – 10 watt)</td>
<td>static (5 watt – 10 watt)</td>
<td>dynamic</td>
</tr>
</tbody>
</table>

**Table 4.6.** Various experimental schemes with the fixed power budget values.

#### 4.4.2.1 Performance Results

We performed our experiments under two power modes: (i) Low power mode. In this case, the total power budget of the chip is set to be 40 watts, and (ii) High power mode. The total power budget is set to be 90 watts in this case. These power budgets are partitioned in two levels by PEPON over the course of execution. For the other schemes (Schemes 1-4) in Table 4.6, the total power budgets at the first level of the power budget partitioning which are fixed during the runtime, to be about 80% and 40% of the maximum power value in each case as given in Table 4.6. Figure 4.6(a-b) plots the normalized weighted speedup values achieved for different mixes when employing the schemes listed in Table 4.6. The weighted speedup for \(n\) applications is computed as follows:

\[
\text{Weighted Speedup} = \sum_{i=0}^{n-1} \frac{IPS^i_{\text{scheme}}}{IPS^i_{\text{alone}}},
\]

where \(IPS^i_{\text{scheme}}\) is the achieved instruction per second (IPS) by the employed scheme for application \(i\) and \(IPS^i_{\text{alone}}\) is the measured IPS achieved when the
application is executed alone without any power constraints. The weighted speedup values in Figure 4.6 are normalized to Scheme 2 (assumed to be the base scheme) in Table 4.6, where power budgets are not modulated over the execution (i.e., all power distributions are fixed throughout the execution). Note that, we consider IPS instead of instruction per cycle (IPC) to compute the speedup values since we use the operating frequency as a knob to control the power consumption of the cores and the IPC value does not change by varying the core frequency.

Figures 4.6(a) and (b) show that PEPON achieves significant performance (workload speedup) improvements in both the low and high power modes (more than 25% in the low power case and more than 20% in the high power case). Further, the total power consumptions track the power budgets quite closely as shown in Figures 4.6(c) and 4.6(d). These plots reveal that partitioning the total power budget at the first level by PEPON plays a significant role in achieving better performance under a certain power budget. As an example, as can be seen from Figure 4.6(a), although performance is improved by partitioning the power budget at the second level among the cores and cache banks in Mix-2 (Schemes 1, 3, and 4), it is less than the improvement achieved by PEPON since the budgets are fixed at the first level in all those schemes. We also note that there is a significant difference between PEPON and Scheme 3 (which mimics a recently proposed scheme [72]), further emphasizing power budgeting at both the levels.

Figures 4.6(a) and (b) also show the average speedup values achieved by different schemes under certain power budget, and (c-d) Dynamic tracking of the total power budget. Note that both (c) and (d) plot all the mixes together. The point here is that, in the low power case, chip-wide power consumptions of our mixes approach 40W, and similarly, in the high power case, chip-wide power consumptions of our mixes approach 90W.

Figures 4.6(a) and (b) also show the average speedup values achieved by different schemes across all the workloads. One can observe that, PEPON improves the overall workload performance significantly as compared to the other schemes.
tested in both high and low power modes (about 21% in the low power mode and 17% in the high power mode). Also, as can be seen, the performance improvement achieved by Scheme 1 is higher than the other schemes (Schemes 2, 3 and 4). This is because this scheme takes advantage of partitioning the power budgets among both the cores and cache banks at the second level, whereas in all other schemes tested (except ours), power budgets (among the cores or caches or both) are fixed at the second level.

4.4.2.2 Dynamics of PEPON

We next focus on how the total power budget is dynamically partitioned by PEPON over the course of execution to achieve the performance enhancements shown in Figure 4.6. Here, we present the results for one of our workloads (Mix1); the remaining workloads exhibit similar behavior.

Figures 4.7(a)-(c) plot the power budgets assigned to cores, caches and NoC by PEPON during runtime for Mix-1, when the total power budget is set to 40 watts. These figures also plot the measured power consumption for different components. As can be observed, PEPON first distributes the total budget at the first level and then, the budget targets are successfully tracked at each group. The accumulated power consumptions of the cores, caches and NoC are plotted in Figure 4.7(d) (this chart shows the total power consumption and not the total power budget). It is important to note that the power budget violation experienced when using PEPON is very low (less than 2%); the power budget violations will be further discussed later. There are two other important points in Figures 4.7(a)-(c) that should be noted: (i) the power consumption of the cores and NoC (Figures 4.7(a) and (c)) oscillate around the target power budgets and the budgets are slightly violated over the execution, whereas the power consumption of the caches is always below the budget, and (ii) the power budget of the caches is tracked faster. As an example, at time ‘0’, as shown in Figures 4.7(a) and (c), the power budgets of the cores and NoC are not the same as their power consumptions, and the power budgets are tracked after some intervals. The main reason for these two differences is that the type of the controllers employed for the caches are different from the ones associated with the cores and NoC. As stated before, in the caches, power consumption is dominated by the static power and the static power exact
model is known (the correlation between the number of active ways and power consumption). Therefore, the controllers associated with the caches are able to select the correct number of active cache ways based on the model in a short time, such that the power budget is not violated. On the other hand, we employ $P$ controllers for the cores and NoC in which the exact model is not known and the frequencies are determined based on the error values, leading to small oscillations.

Figure 4.8 plots the power budget distribution at the first level for Mix-1 (similar to Figure 4.7) but when our total budget is 90 watts (high power mode). As one can observe, the power budget is successfully partitioned and tracked when we are operating at high power mode, too. We observed the same dynamic behavior at the first level of power budget partitioning from our other mixes as well. Note that, the NoC power in these experiments is controlled directly by modulating the NoC frequency/voltage level. However, the power budgets of the cores and caches are not directly controlled and are distributed further among the individual components. Therefore, tracking the total power budgets of the cores and caches at the first level (as shown in this set of figures) implies that each individual component (cores and caches) has been successful in tracking its own budget.

Figures 4.9 illustrates how the power budgets determined at the first level are partitioned among the individual components (cores and caches) at the second level over the execution for Mix-1. As can be observed from each of these plots, the total power budget is partitioned into 16 pieces, and the size of each piece indicates the amount of power budget allocated to each component. Different components receive different shares from the power budget since the total budgets are distributed based on the utility of the caches or the performance power ratio of the cores over the recent time interval (the schemes that we employ at the second level). The controller configured for each hardware component tracks the target power budget by adjusting the voltage-frequency and the number of active cache ways of the cores and caches, respectively. To better understand these power distribution curves, we plot in Figure 4.10 the selected frequencies for the cores and the number of cache ways for Mix-1. We see that the values of these knobs are also different for different components at a specific time. As an example, at time 10, the frequencies assigned to some of the cores are low as compared to the others. This is because, the power budgets are not the same for different cores. Note also
Controller Selection. As discussed before, we employ simple and fast $P$ controllers to modulate the frequency of the cores and the NoC to track the specified power budgets. Instead of a $P$ controller, one may suggest to employ a more complicated $PID$ controller for frequency modulation. One of the main advantages of
employing PID controllers is that they can drive steady-state errors to zero. As discussed before, in our system, this error is already zero when employing P controllers. Figure 4.11 plots how the power budgets (which is assigned to one of the cores in Mix-1 experiment) are tracked by P and PID controllers over the course of execution. As can be observed from this figure, the P controller is able to track the targets as well as the PID controller. However, the PID controller slightly reduces the number of oscillations at some time intervals (e.g., at 30th time interval). On the flip side, the P controller has the capability to respond more quickly to the target variation (e.g., at 70th time interval). In addition, as compared to the PID controller, the P controller incurs less overhead.
Figure 4.12. Performance improvements achieved for Mix-1 by varying the default values of configuration parameters.

4.4.2.3 Sensitivity Experiments

We also measured sensitivity of Mix-1 (under the low power mode) to different parameters. As stated before, at each epoch, the power budgets are increased and decreased by \( \delta \) and \( \delta - \epsilon_1 \), respectively at the first level of power budget partitioning tree. Recall from Table 6.2 that the default value for \( \delta \) is 1 in our experimental evaluations. Figure 4.12 plots the speedups achieved for Mix-1, when the \( \delta \) value is set to 2 (the results with a value of 1 are reproduced for ease of comparison). We see that increasing the \( \delta \) value has a negative impact on the achieved speedup by PEPON. This is because, the power budgets may oscillate around the optimal values and not reach the optimal ones by having a larger \( \delta \).

Our experiments with different cache sizes show that our speedups are not very sensitive to the cache bank sizes, since in this case, due to the low budget allocated to the cache banks, even when more cache space is available, only a portion of them is used during runtime. One of the parameters that affects the performance improvements achieved by PEPON is the frequency at which the control decisions are made and enforced (our epoch sizes). Our experiments show that the speedup achieved for small epoch sizes is generally higher. This is because, in this case PEPON is invoked more frequently, and the behaviors of the running applications are monitored with a higher frequency; as a result, we may end up achieving better performance improvements. However, this also means more overhead.
4.4.3 Power Budget Violations

As can be observed from Figures 4.7(d), 4.8(d), the overall power budget could not always be tracked with 100% accuracy and is violated over the course of execution (although this violation is in general very low). Also, in some cases, the budget is not fully utilized. This is because the power consumption of the individual components at the bottom of the partitioning tree (see Figure 4.3(a)), is not always exactly the same as their power budgets. There are two main reasons for this: (i) the knobs used by the controllers can only take values from a set of discrete levels and cannot be varied continuously. As an example, the frequency/voltage of a core can only be set to the one of the discrete values selected from Table 4.2. Consequently, the frequency chosen by the controller may not be the exact one required to reach the power budget; and (ii) when the target power budget of a controller is modulated, it takes time for the controller to track the new target and, during that time, the output power may fluctuate. Even if the budget is fixed, the controller needs to compensate for the potential change in the behavior of the running application. One of the methods using which these over- and under-utilizations of the available power budget can be handled is through dynamically changing the reference budgets. For example, if we under-utilize the available budget in the previous interval by \( x\% \), we could increase the reference budget value by \( x\% \). Investigating this and similar compensation schemes is in our future research agenda.

4.5 Related Work

Control theory as a tool for power management has recently gained popularity but most prior control centric works do not account for chip-wide power management that takes into account cores, on-chip caches, and the underlying NoC. We have already quantitatively compared our work with [72] and show that our proposal fares better since it is a more holistic approach that considers the on-chip cores, caches and the NoC. In this section, we discuss the most relevant prior works to our proposal.
4.5.1 Feedback Control Based Power Management

[73] proposes a control theoretic framework for power management in multi-clock domain architectures, where each domain is controlled by a separate clock. In this work, a two-tier power management architecture is proposed, where the first tier allocates a target budget to each local island and the second tier controls the power. The solution becomes simpler in this case since it is implicitly assumed that all cores (and their caches) in a domain would be running similar applications, i.e., all cores in a particular domain would either host CPU intensive or memory intensive applications. This allows for DVFS of an entire domain using the per domain clock. Compared to this architecture, our solution targets a realistic CMP with cores, shared caches and an NoC, where domain level clock management is prohibitive. To tackle this, our proposal has a two level power control: The first level allocates power budget among the cores, shared caches and the NoC, and the second level redistributes the core power among different cores, and the cache power among different caches. Other related works advocating control theory for power and thermal management include [87, 88, 83, 74, 89]. In [87], a PI-based core thermal controller is proposed and an outer control loop is used to decide process migrations on thermal emergencies. The proposal in [88] extended [89] and proposes a distributed version of the scheme, applying the basic mechanics of the formal approach to CMPs. In [83], a hierarchical control architecture is proposed that controls the total power consumption of a large-scale data center to stay within a constraint imposed by its power distribution capacity. This work, however, only models cores in a processor when it comes to power management. In [74], Wang et al. propose a multi-input multi-output solution to CMP temperature management. One major difference between our proposal and the work in [74] is that, while our approach is simple and flexible (in fact flexibility policy was one of the prime design time consideration behind our proposed approach), [74] use sophisticated controllers whose complexity is proportional to the number of cores in the chip. Consequently, it is hard to apply the same to really large CMPs. Further, when compared to the work in [72], our proposal targets all major components in a CMP (cores, caches and NoC), whereas [72] only targets power control in cores.
4.5.2 Ad-hoc Power and Thermal Management

There are many prior proposals focusing on the design of power and thermal management schemes [90, 91, 92, 93, 94, 95, 96]. These works highlight the importance of adapting to workload behavior under power and thermal constraints. However, most of these works adopt a combination of independent local schemes in the architecture or scheduling decisions in the operating system. Meng et al. [94] proposed an adaptive, multi-optimization strategy for multi-core power management and addressed the problem of meeting a global chip-wide power budget through runtime adaptation of configurable processor cores. Isci et al. [92] also used a global power manager, similar to ours, that provides a mechanism to sense the per-core power and performance state of the chip at periodic intervals; it then sets the operating power level or mode of each core to enforce adherence to known chip-level power budgets. However, the local monitoring used to enforce these power budgets were based on open loop control and does not provide the same robustness as the formal feedback control in our work. Furthermore, the constraints on the achievable performance were less stringent as they perform per-core DVFS as opposed to core, cache and NoC as we do in our work. Prior proposals like [97, 98, 99] have looked at energy saving mechanisms for cache alone by dynamically reconfiguring the cache hierarchy or partitioning a shared cache among multiple applications. Works such as [100] have investigated memory cell sizing for cache energy efficiency. Our work also uses cache reconfiguration for energy management, but we do so as part of managing the energy consumption of an entire CMP rather than the on-chip caches alone.

4.6 Concluding Remarks

The main contribution of this work is a two-level power distribution strategy oriented towards maximizing system performance under a specified power budget. The results collected from our experiments with 6 workloads formed using 16 applications selected from a set of 30 applications clearly demonstrate the importance of power distribution at both the levels. Specifically, our experimental results indicate that, when using PEPON, overall performance can be improved by as much as 29% over the case when no power distribution is adopted. Further, our results
also show that distributing power only at a core level (as done in a recent study [72]) may not be sufficient for maximizing performance.
Chapter 5

HiPEMM: A High Performance Energy-Saving Mechanism for Multicore Memories

5.1 Introduction

In this chapter, we introduce HiPEMM, a high performance DVFS mechanism for reducing memory system power by scaling individual memory channel frequencies to match application demand. Our strategy involves partitioning applications by their sensitivity to memory latency. Less sensitive applications can tolerate a slower memory channel without loss of performance; however, the most sensitive applications should still be given the maximum memory channel frequency or they will suffer from reduced performance with little energy savings. We introduce a metric called Discrete Misses per Kilo Cycle (DMPKC) to capture the sensitivity of running applications. Our initial experiments show that even statically partitioning the memory controllers can reduce power by assigning lower frequency memory channels to the applications that can tolerate it, while keeping some memory channels at a higher frequency for the sensitive applications.

The contributions of this chapter are as follows: (1) We introduce DMPKC as a metric to capture the sensitivities of the running applications to frequency modulation. (2) We propose HiPEMM as a high performance energy-saving DVFS
mechanism for the memory system that dynamically partitions the memory channels across the concurrently running applications and assigns frequency levels to the memory channels. HiPEMM dynamically adjusts this partitioning to achieve the optimal assignment of applications to channel frequencies at each time slice. Starting from the lowest frequency, it gathers cores (applications) with similar DMPKC values until the channels at that frequency can be fully utilized, and then assigns groups with higher DMPKC values to increasing channel frequencies until all cores have been assigned. In our experiments, HiPEMM was able to save 25% of memory energy, and 10% of system energy with only a 3% loss of performance.

We provide an overview of our target multicore architecture and the motivation behind our proposed scheme in Section 5.2. In Section 5.3, we give the details of our proposed scheme. Section 5.4 presents our experimental framework and discusses our results. We contrast our approach with related work in Section 6.4, and conclude the chapter in Section 3.6.

5.2 Background and Motivation

5.2.1 Background

Figure 5.1 illustrates the high level view and the main components of the multicore system that we target in this chapter. In our default system, 32 cores execute co-runner applications and 4 on-chip memory controllers manage the flow of the off-chip accesses. The detailed system parameters are given in Table 6.2.

**DRAM organization.** In Figure 5.1, the main memory system consists of 4 channels and each of them is controlled by a memory controller (MC). One or more DRAM boards, called dual inline memory modules (DIMMs), are connected to each channel. DRAM chips are placed on the DIMMs and each chip participates in providing a part of the requested data. A *rank* is the subset of the DRAM chips that are activated for a single off-chip access. Data is stored in 2D memory arrays (organized as rows and columns) called *banks*. There are several banks per DRAM chip. Once data from a bank is requested, first the corresponding row is loaded into a buffer called the *row buffer*, and then the requested portion of the row data is selected based on the column address. More details on the memory system and
DRAM organization can be found elsewhere [48, 49, 14].

**Address mapping.** In Figure 5.1, when a miss occurs in the cache system (L1 and L2 in our target system), an off-chip memory request is sent to one of the MCs via the interconnect module. The destination memory controller is determined based on the mapping of physical addresses to channels and MCs. Two commonly-used policies for the channel mapping are *cache line interleaving* and *page interleaving*. In the cache line interleaving policy, two consecutive cache lines are mapped to two consecutive channels, whereas in page interleaving two consecutive OS pages are mapped to consecutive channels. We use the page interleaving policy in this chapter as our default policy. It should be noted that, under both mapping policies, the off-chip memory requests generated by a running application (executed on a core) are distributed across the channels and each memory controller receives requests from all running applications (in this chapter we assume one-to-one mapping between applications and on-chip cores).

**DVFS in main memory.** DVFS can be employed to reduce the power and energy consumption of the main memory system in modern multicores [104, 105, 106, 107, 102]. For instance, the JEDEC standard [108] provides such a mechanism in emerging memory systems. This mechanism can be used in three main components of a memory subsystem: memory controllers (MCs), off-chip buses, and DIMM modules. In our study, considering real systems, we assume that hardware lets OS change the frequency of DIMMs, buses and the voltage/frequency of the MCs.
Further, to avoid extra synchronization hardware, we assume that DIMMs and the off-chip bus operate at the same frequency and the frequency of the MC is set to double the bus frequency. The voltage of the MC is adjusted based on the assigned frequency. The same assumptions have also made in prior studies [102]. Therefore, from now on, we refer to one frequency value for each channel as the channel frequency.

Lowering the frequency of a channel reduces the background power (leakage and refresh) and dynamic power (activation, read/write and termination) of the memory system. On the other hand, this decision may incur a significant performance overhead. Specifically, reducing the frequency increases the DRAM access time, bus data transfer time and the memory request waiting time in MC bank queues. In short, the off-chip memory access delay increases. This added delay affects the execution time of the running applications. Therefore, although reducing the frequency saves power and energy in the memory system, the resulting performance loss should be taken into account.

5.2.2 Motivation

In this section, we present and discuss the experiments we ran to motivate our proposed scheme. For these experiments, we use workload-1 in Table 5.3. This workload is a mixture of the memory intensive and non-intensive applications. We ran these applications on our default 32-core target system with 4 memory channels. We give the details of our experimental setup, performance metrics, and the model we use for calculating the memory energy consumption in Section 5.4.

First, we ran the workload seven times, each time with the frequency of the memory system set to a different value. In this experiment, we assume that all four memory channels have the same frequency. Figure 6.2 plots the memory energy consumption and the workload performance under different memory frequencies. Note that the values are normalized to the base case in which the memory system operates at the maximum frequency (800 Mhz). We refer to the average slow down of the workload applications (in terms of IPC) as the “workload performance”.

As shown in this figure, the memory energy decreases with lowering the memory frequency. However, this energy reduction comes with the expense of performance penalty. For instance, although the memory energy is reduced about 50% when
Figure 5.2. Memory energy and the workload performance for different memory frequencies. The results are normalized to the base case in which the memory system operates at the maximum frequency (800 Mhz).

Figure 5.3. The slow downs of individual applications running on our 32-core system when the memory frequency is set to 720 Mhz. the frequency is set to 320 Mhz, there is a 30% performance loss implying that the execution times of the workload applications have increased about 30%.

As mentioned above, the performance values presented in Figure 6.2 are the “average values” across all the applications in the workload. However, different applications may have different sensitivities to modulations in the memory frequency. Figure 5.3 plots the individual performances of the running applications when the memory frequency is set to 720 Mhz. We see that reducing the memory frequency affects the performance of applications to varying degrees. For instance, this reduction slows down the application running on core 13 significantly, while other cores such as core 29 remain unaffected.

In the previous experiment, we assume that all the channels operate at the same frequency. However, any of the cores/applications may face performance degradation even when only the frequency of one of the channels it uses is reduced, even all other channels operate at the maximum frequency. This is because, as mentioned before, the memory accesses of different cores are typically distributed across all
memory channels according to the address mapping policy and, consequently, reducing the frequency of one of the channels may make the memory requests coming to that channel become a bottleneck for the performance of the running applications. As an example, Figure 5.4 plots the distribution of the memory requests of six of the cores across the four memory channels in the previous experiment. As can be observed, the memory accesses issued by the six cores are almost evenly distributed across the 4 MCs (each MC receives around 25% of the total off-chip accesses issued by the same core).

Based on our observations made so far, next we perform our final motivational experiment. In this experiment, we first select a subset of applications that are not sensitive to the memory frequency change (performance loss < 2% in Figure 5.3). We find 10 applications in this category, and then map all the pages of these applications to MC0. We also map the pages that belong to the other applications to MC1, MC2 and MC3. By doing this, the off-chip memory requests are clustered into two groups: MC0 receives the requests from the non-sensitive applications and MC1, MC2 and MC3 receive the rest of the requests. In the next step, the frequencies of the MCs are assigned. In Figure 5.3 the frequencies of all the 4 channels are set to 720 Mhz. However, in this experiment, the frequency of MC0 is set to 480 Mhz and the frequencies of MC1, MC2 and MC3 are set to the maximum value (800 Mhz). The goal here is to reduce the memory energy consumption by reducing the channel frequency and at the same time attempt to avoid the performance loss of sensitive applications by mapping them to the MCs that operate at the maximum frequency available. Figure 5.5 compares the workload energy and performance in two cases: 1. When no clustering is used and the frequencies of all the channels are reduced (Figure 6.2; 720 Mhz). 2. When the clustering based scheme is employed. It is clear from this figure that we achieve better performance and save more energy (about 7%) by using the proposed
5.3 HiPEMM: Our Proposed Energy Saving Mechanism

In this section, we discuss the details of the different components of HiPEMM. As shown in Section 5.2.2, the memory system frequency scaling can be used with clustering the off-chip requests and memory channels to cut both energy consumption and performance penalty. That motivational example illustrates the clustering mechanism (note that the values are normalized to the case in which all the memory channels operate at the maximum frequency). In the clustering case, the power consumption is reduced by lowering the memory frequency, while the performance loss experienced by the sensitive applications is reduced. As can be seen in Figure 5.6, the clustering scheme improves the performance of those applications that lose significant performance when no clustering scheme is employed (see Figure 5.3).
main idea behind our proposed mechanism. However, there are several details of the HiPEMM implementation that we discuss in this section.

Unlike our motivational example that used static partitioning of memory controllers and fixed frequencies for memories, HiPEMM is a dynamic scheme that regulates the memory channel frequencies and the application clusters over the course of execution with the goal of saving more energy while maintaining high performance. Figure 5.7 illustrates the high level view of HiPEMM. Our approach takes the system performance counters as input and gives the application clustering decisions and the corresponding channel frequencies as output. The OS runs the HiPEMM algorithm over the execution at fixed time epochs (intervals) and enforces the output decisions. The hardware provides the system performance counters which are used to identify the sensitivities of the running applications to memory frequency.

5.3.1 HiPEMM Algorithm

The HiPEMM algorithm first clusters the running applications into different groups based on the application sensitivities and then, assigns memory channels to the application clusters. At the same time, the algorithm also determines the values of the frequencies assigned to the memory channels.

Recall that Figure 5.3 shows sensitivities of applications to memory frequency scaling. We generated this graph by running our workload twice (once with the maximum memory frequency as a baseline and once with the reduced memory frequency) and comparing the results. However, this comparison cannot be carried out at runtime. Since HiPEMM is a dynamic mechanism, the application sensitivities need to be estimated based on a parameter that can be measured over the execution. In HiPEMM, we use a metric called Discrete Misses Per Kilo Cycles (DMPKC) to predict the sensitivity of each application. DMPKC of an application is defined as the number of discrete groups of off-chip memory requests (per thou-
sand cycles) that belong to the same application. A subset of memory requests is considered a “group” if the difference between the memory arrival times of any two of the requests in that group is less than the threshold (DMPKC-th). The DMPKC value is incremented by 1 for each off-chip memory request group. This is because our goal is to consider off-chip requests issued by the same application in parallel to estimate the application sensitivities to the off-chip delay increase coming from the memory frequency reduction. In our experiments, we observed that, if two applications have the same number of misses per thousand cycles, the application with higher off-chip access parallelism is less sensitive to the memory frequency reduction since the added off-chip access delays have less impact on the application performance if the memory requests of the application are issued in parallel.

Figure 5.8(a) plots the correlation between the average DMPKC values of the 32 applications in our motivational example (in Section 5.2.2) and their performance loss when the memory frequency is scaled down (each dot in this plot represents an application). As can be seen, applications with higher DMPKC values lose more performance in this case. Figure 5.8(b) plots the correlation as in Figure 5.8(a) but the x-axis is Misses Per Kilo Cycles (MPKC) instead of DMPKC. As one can observe from this figure, the correlation is much less strong without considering the off-chip access parallelism.

At each time interval, the OS runs the HiPEMM algorithm using the DMPKC values of the running applications. Algorithm 1 gives an algorithmic representation of HiPEMM. A brief description of the parameters used in this Algorithm
Table 5.1. Parameters/functions used by HiPEMM.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>number of cores</td>
</tr>
<tr>
<td>$Ch$</td>
<td>number of memory channels</td>
</tr>
<tr>
<td>DMPKC_TH</td>
<td>a constant threshold value (default: 0.6)</td>
</tr>
<tr>
<td>FREQ_BASE</td>
<td>minimum channel frequency (320 Mhz)</td>
</tr>
<tr>
<td>CHANNEL_FREQ</td>
<td>frequency of each memory channel</td>
</tr>
<tr>
<td>CORE_MAP</td>
<td>$\text{CORE_MAP}[i][j] = 1$ shows that the OS pages of core $i$ can be mapped to channel $j$</td>
</tr>
<tr>
<td>core_done</td>
<td>$\text{core_done}[i] = 0$ shows that HiPEMM has not taken of core $i$ so far</td>
</tr>
<tr>
<td>inc(dmpkc_level)</td>
<td>increments $\text{dmpkc_level}$ by 1 level (default: 0.1)</td>
</tr>
<tr>
<td>inc(freq_level)</td>
<td>increments $\text{freq_level}$ by 1 level</td>
</tr>
</tbody>
</table>

is given in Table 5.1. Algorithm 1 takes the DMPKC values as input and gives the frequency of the memory channels ($\text{CHANNEL\_FREQ}$) and the channel mappings ($\text{CORE\_MAP}$) as output. As an example, if the algorithm maps core $i$ to two memory channels (channel-0 and channel-1, i.e., $\text{CORE\_MAP}[i][0] = 1$ and $\text{CORE\_MAP}[i][1] = 1$), this means that the OS pages accessed by core $i$, from this point on, will be mapped to channel-0 and channel-1. The main loop of the algorithm (Lines 5-26) terminates once all the channels in the memory system are assigned to the cores. The algorithm starts working with the minimum available memory frequency ($\text{freq\_level}$) and a minimum DMPKC threshold ($\text{dmpkc\_level}$). First, it counts the number of the cores that have DMPKC values less than $\text{dmpkc\_level}$ (Lines 7-11). Then, $\text{get\_num\_channels}()$ decides how many channels can be allocated to that number of cores (we will discuss the policy we employ for that later in this section). Based on the number decided by $\text{get\_num\_channels}()$, the cores selected at Lines 7 to 11 are mapped to the available channels with the $\text{freq\_level}$ frequency. Finally, $\text{dmpkc\_level}$ and $\text{freq\_level}$ values are incremented for the next iteration of the main loop. In summary, each $\text{dmpkc\_level}$ has a corresponding $\text{freq\_level}$, and HiPEMM assigns one or more channels (with $\text{freq\_level}$) to the cores with that $\text{dmpkc\_level}$ (if they are sufficient in number to be allocated channels). We employ the policy used in the channel partitioning scheme proposed in [109] to determine the number of memory channels at line 12. This policy is based on the number of cores in the subset, the total number of cores, and the total number of channels. If $C$ is the total number of cores and $Ch$ is the total number of channels, $\left\lfloor \frac{n}{C} \times Ch \right\rfloor$ gives the number of channels that will be allocated to the $n$ selected cores.

We now give an example to better explain how HiPEMM works in practice. Suppose that we have an 8-core machine with 2 memory channels and, at the $k^{th}$ time interval, the DMPKC values of the running applications are assumed to be
Algorithm 1 HiPEMM Algorithm

Input: $DMPKC[1..C]$  
Output: $CHANNEL\_FREQ[1..Ch]$, $CORE\_MAP[1..C][1..Ch]$  

1: $dmpkc\_level = DMPKC\_TH$  
2: $freq\_level = FREQ\_BASE$  
3: $current\_channel = 0$  
4: $core\_done[1..C] = 0$  
5: while $current\_channel < TOTAL\_CHANNELS$ do  
6: \hspace{1em} $counter = 0$  
7: \hspace{2em} for $i = 0 \text{ to } C$ do  
8: \hspace{3em} if $core\_done[i] == 0$ and $DMPKC[i] < dmpkc\_level$ then  
9: \hspace{4em} $counter++$  
10: \hspace{3em} end if  
11: \hspace{2em} end for  
12: $ch\_num = get\_num\_of\_channels(counter, C, Ch, freq\_level)$  
13: for $j = 0 \text{ to } ch\_num$ do  
14: \hspace{1em} for $i = 0 \text{ to } C$ do  
15: \hspace{2em} if $core\_done[i] == 0$ or $core\_done[i] == freq\_level$ then  
16: \hspace{3em} if $DMPKC[i] < dmpkc\_level$ then  
17: \hspace{4em} $CORE\_MAP[i][current\_channel] = 1$  
18: \hspace{4em} $CHANNEL\_FREQ[current\_channel] = freq\_level$  
19: \hspace{4em} $core\_done[i] = freq\_level$  
20: \hspace{3em} end if  
21: \hspace{2em} end if  
22: \hspace{1em} end for  
23: \hspace{1em} end for  
24: inc($dmpkc\_level$)  
25: inc($freq\_level$)  
26: end while

as shown in Figure 5.9(a). We further assume that our base $DMPKC$ is 0.6 and the available memory frequencies are 320, 400, 480, 560, 640, 720 and 800 Mhz. Therefore, $dmpkc\_level$ and $freq\_level$ values start at 0.6 and 320 Mhz, respectively. At the first iteration, the algorithm finds two cores that have $DMPKC$ values less than 0.6 (cores 1 and 2). Based on our employed policy, two cores are not sufficient to be allocated a channel in this case. Then, $dmpkc\_level$ and $freq\_level$ are increased by two more levels to 0.8 and 480 Mhz, respectively, at which point four cores (cores 1 to 4) have $DMPKC$ values less than 0.8 and channel-0 is assigned to them. We apply the same strategy to the rest of the cores and channel-1 is assigned to them with a frequency of 800 Mhz. Figure 5.9(b) shows the final mapping in our example.

When a running application tries to access a page in the memory, there are three possible scenarios: (1) The page is not already allocated and a page fault occurs. In this case, the page will be allocated in one of the preferred channels determined by HiPEMM. (2) The page is already allocated and resides in one of the preferred channels suggested by HiPEMM. No action is taken here, and the page is accessed by the application. (3) The page is already allocated in the memory but
is not located in one of the preferred channels (due to the changing of the channel mapping over the execution). In this case, there are two options. One option is to migrate the page to one of the preferred channels which leads to TLB and cache block invalidation, and incurs performance overhead [109]. The other option is not performing the migration and the page is accessed from its current channel. We chose the second option since, based on our experiments, migration does not bring significant additional performance benefits (since an application’s behavior does not vary much during execution).

5.4 Experimental Evaluation

In this section, we first explain our experimental setup and the metrics we use to evaluate our proposed scheme. Next, we present our collected results that demonstrate the effectiveness of HiPEMM in reducing the memory and system energy while minimizing the performance impact.

Setup: We use GEMS [61] as our simulation framework. GEMS is a full-system simulator that allows simulation of multicore systems with different configurations. It also contains two components: Opal and Ruby. Opal is the module that implements a detailed model for out-of-order processing cores that run the target workloads. Once a core issues a memory request, that request is sent to the Ruby
Processors
32 out-of-order cores with private L1 data and instruction caches:
instruction window size: 128, LSQ size: 64.

Private L1 D&I-Caches
Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency.

L2 Cache
64 bytes block size, 10 cycle access latency.

L2 Cache Bank Size
512KB per core.

Number of Banks Per Memory Controller
16.

Memory Configuration
DDR-800, Memory Bus Multiplier: 5, Bank Busy Time: 22 cycles,
Rank Delay: 2 cycles, Read-Write Delay: 3 cycles, Memory
CTL latency: 20 cycles, Refresh Period: 3120.

Epoch length
5M cycles

Simulation length
100M cycles

| Table 5.2. Baseline configuration. |

component. Ruby models the entire memory system including the L1 and L2 caches, interconnects, and off-chip memory. For calculating the memory system energy, we employ the power model provided by Micron for DDR3 memory technology [110]. The Micron power model takes the activity and frequency as input and estimates the power consumption of the memory system. Energy numbers are calculated based on the obtained power numbers and the execution times of the running applications.

Table 6.2 gives our baseline configuration. Our baseline system has 32 cores that run 32 applications and 4 DDR3 memories (channels) connected to 4 memory controllers. More details on the timing and implementation of the memory system can be found in [111]. We factor the performance impact of the memory frequency regulation as the variation in the access and bus data transfer time.

Metrics: We compare HiPEMM against a “base scheme” in which no clustering is employed and the entire memory system frequency (i.e., all the MCs operate at the same frequency) is varied to obtain the best performance (similar to the scheme proposed in [102]). This comparison is done in terms of the memory energy, full-system energy, performance, and Energy-Performance-Ratio (EPR). Note that, all the energy, performance, and EPR values presented in this section are normalized to the case where all the memory channels operate at the maximum frequency (800 Mhz). To estimate the full system-energy saving we make the same assumption as in [102, 103] that, on average, 40% of the system energy is consumed in the memory system. We compute the performance as:

$$\text{Perf}(\text{scheme}) = \frac{\sum_{i=1}^{N} \frac{\text{IPC}_{\text{scheme}}(i)}{\text{IPC}_{\text{base}}(i)}}{N},$$

where $\text{IPC}_{\text{base}}(i)$ is the Instructions-Per-Cycle of application $i$ under assumption
that all the channels operate at the maximum frequency, and $IPC_{\text{scheme}(i)}$ is the IPC of that application when our scheme is employed. Energy-Performance-Ratio (EPR) is a metric that captures the ratio between the amount of energy that is saved and the performance loss of the employed mechanism, and is defined as $EPR = \frac{\text{Energy}}{\text{Performance}}$. As an example, suppose that we have two schemes, Scheme-1 and Scheme-2, and both save 20% energy but Scheme-1 incurs 10% performance overhead while Scheme-2 incurs 20%. The EPR values of Scheme-1 and Scheme-2 would be 0.89 and 1. A smaller EPR value indicates a better performance-energy efficiency since it implies that we can save more energy with less performance loss. This metric in a sense represents a “tradeoff” between energy and performance; in general a scheme with an EPR greater than 1 does not represent a good tradeoff and the energy savings of that scheme are not worth the cost in performance.

**Workloads:** We formed our workloads using the applications from the spec2k6 benchmark suite [57]. Table 5.3 gives the 30 workloads that we used in our experiments on our 32-core system (the numbers in parentheses represent the number of copies for each application in the workload). The workloads listed in Table 5.3 cover all applications and are categorized into three different groups based on the “memory intensity” of the applications: (1) Workloads 1 through 10 (mixed workloads): in these workloads, half of the applications are memory intensive (applications with high MPKI) and the remaining ones are memory non-intensive. (2) Workloads 11 through 20: all applications in this category are memory intensive. (3) Workloads 21 through 30: none of the applications in this group is memory intensive.

MPKI values representing the memory intensities of the applications in the spec2k6 suite can be found in [55].

**Main Results:** Figure 5.10(a) plots the memory energy consumption for the workloads given in Table 5.3 for the base scheme and HiPEMM. Note that the energy numbers are normalized to the case in which all memory channels operate at the maximum frequency. The reflections of these memory energy savings on the system total energy are shown in Figure 5.10(b). As can be observed from Figures 5.10(a) and 5.10(b), the base scheme and HiPEMM save, on average, 18% and 25% memory energy (7% and 10% system energy), respectively, for our mixed workloads (workloads 1 to 10) by reducing the memory frequencies. That is, HiPEMM saves
Table 5.3. Workloads used in our 32-core experiments. The numbers within parentheses show the number of running instances of each application.

7% more memory energy than the base scheme. Recall that our base scheme is similar to a recently proposed one [102] (the entire memory system frequency is varied). As shown in Figure 5.11(a), HiPEMM saves this energy while incurring, on average, only about 3% performance loss in the case of the mixed workloads; in comparison, the average performance loss incurred by the base scheme is around 11%. This is because HiPEMM reduces the performance overhead of the memory frequency reduction by clustering the running applications and partitioning the memory channels over the execution. HiPEMM’s clusters can take advantage of the varied memory demands of applications in the mixed workload, while the base scheme must subject all applications to the same channel frequency. Figure 5.11(b) plots the average memory access latency for each request when our base scheme and HiPEMM are employed for each workload. As can be seen, HiPEMM imposes less overhead on memory accesses as compared to the base scheme. These changes in off-chip delay cause the performance variations shown in Figure 5.11(a).

In the case of memory-intensive workloads (workloads 11 to 20), as can be seen
in Figures 5.10(a) and 5.10(b), the energy consumption values are not reduced by employing the base scheme. This is because, there is a high load and pressure on the memory system; even a slight decrease in the speed of the memory system (by reducing the frequency) can lead to a significant increase in the memory queue latencies and ultimately to performance loss. Therefore, although reducing the memory frequency reduces the power consumption, the resulting performance loss increases the total energy consumption for the base scheme. Again, HiPEMM saves 8% and 3%, memory and system energy respectively, by avoiding the performance loss caused by the frequency scaling. However, even HiPEMM could not save energy for some of the workloads, such as workloads 15 and 17, since no non-sensitive applications could be found in any execution epoch.

![Graphs showing Memory and System Energy Consumption](image)

(a) Memory Energy  
(b) System Energy

**Figure 5.10.** Memory and system energy consumption when HiPEMM and the base scheme are employed.

In general, memory non-intensive applications are less sensitive to the memory frequency regulations. Therefore, more energy can be saved by scaling the memory system frequency. As can be observed from Figures 5.10(a) and 5.10(b), HiPEMM on average saves 43% and 17% memory and system energy with 3% performance loss. However, in most of these (non-intensive) workloads, the same amount of energy and performance is also saved by employing the base scheme. This is because the sensitivities of all the applications in such workloads are almost the same and HiPEMM does not bring much additional benefit by clustering the applications.

Figures 5.12(a) and 5.12(b) plot the memory and system EPRs, which show both the energy savings and performance loss (shown in Figures 5.10(a), 5.10(b) and 5.11(a)) together in a single metric. As can be observed from these plots, HiPEMM achieves 11%, 13% and 6% system level EPR improvements for the mixed, intensive and non-intensive workloads, respectively, as compared to the base scheme. Note that, although the EPR is improved by 13% for intensive
workloads as compared to the base scheme, the actual value is close to 1, which
means we lose about 1% performance to save 1% system energy. It should also be
observed that non-intensive workloads have the best EPR savings, indicating that
they present the best opportunity to reduce the memory frequency to save power
without losing much performance.

As discussed before, HiPEMM is a “dynamic scheme” and varies the application
clusters, channel partitioning and frequencies over the course of execution. To
illustrate its dynamic behavior, Figure 5.13(a) plots the frequencies assigned by
HiPEMM to our 4 memory channels during runtime when workload-1 is running,
and Figure 5.13(b) shows the same result for workload-2. Figures 5.13(c) and
5.13(d) plot two snapshots of how the channels are partitioned across the running
applications at the 5th and 15th time epochs for workload-1. For instance, the
frequency shown for core 6 at the 5th time epoch (Figure 5.13(c)) is 800 Mhz and,
as shown in this figure, only MC-2 and MC-3 operate at that frequency, therefore,
core 6 is mapped to MC-2 and MC-3 at the 5th time interval. At the 15th epoch
(Figure 5.13(d)), core 6 is still assigned to 800 Mhz but only MC-3 is operating at
this frequency, so core 6 is in the cluster mapped to MC-3 and a different cluster
is mapped to MC-2.

(a) Memory controller frequency over time (workload-1). (b) Memory controller frequency over time (workload-2). (c) Epoch 5: memory frequency assignments (workload-1). (d) Epoch 15: memory frequency assignments (workload-1).}

**Figure 5.13.** Memory controller cluster/frequency assignments over time.

**Sensitivity Results.** In this section, we present our sensitivity analysis to show the impact of changing the system parameters on the improvements achieved by HiPEMM. For the following experiments, we picked the first 4 workloads from each category (mixed, intensive, and non-intensive) given in Table 5.3. Therefore, for each experiment, we tested 12 workloads from the three categories.

As mentioned before, our default target system has 32 cores with 4 memory controllers that manage the off-chip data accesses to/from 4 memory channels. In our first sensitivity experiment, we ran our workloads on systems with different numbers of memory controllers to investigate how the number of MCs affects the behavior of HiPEMM. Figure 5.14(a) plots the memory level EPR values achieved by HiPEMM for three cases: systems with 2, 4, and 8 MCs. Note that, the EPR values are normalized to the base scheme in each case. This experiment shows that, in general, HiPEMM achieves better (lower) EPR values compared to the base scheme as the number of MCs increases. This is because HiPEMM has the opportunity to cluster the concurrently running applications and partition the memory channels at a finer granularity (i.e., it has more flexibility). The worst case for HiPEMM in Figure 5.14(a) is when there are 2 MCs and we run memory-intensive workloads. In this case, the base scheme performs even better than HiPEMM (indicated by values larger than 1 since the values are normalized with respect to the base scheme in each case).

*DM PKC.TH* in the HiPEMM algorithm (Algorithm 1) is the main parameter that affects the HiPEMM aggressiveness in reducing the memory frequency. This parameter is the initial threshold used to identify the sensitivities of the applica-
Figure 5.14. Sensitivity analysis.

...tions. To better illustrate the impact of $DMPKC_{TH}$ on the aggressiveness of HiPEMM, consider our example in Figure 5.9(a). The $DMPKC_{TH}$ value in this example is assumed to be 0.6. Therefore, $dmpk_{level}$ and $freq_{level}$ are increased two more levels and reach 0.8 and 480 Mhz, respectively, where four cores (cores 1 to 4) have $DMPKC$ values less than 0.8 and channel-0 is assigned to them. However, if $DMPKC_{TH}$ is set to be 0.8 instead of 0.6 in this example, at the first iteration in the main loop in Algorithm 1, channel 0 will be assigned to the first four cores with the frequency of 320 Mhz. Increasing the $DMPKC_{TH}$ value would result in a lower frequency assignment for channel 0. Generally speaking, $DMPKC_{TH}$ can be used as a knob to control the aggressiveness of HiPEMM such that more power is saved by increasing the value of this parameter (since the levels of the assigned frequencies are reduced) with the cost of losing more performance. Figure 5.14(b) plots the memory EPR values achieved by HiPEMM under three $DMPKC_{TH}$ values: 0.4, 0.6 (default) and 0.8. As can be observed from this graph, the impact of changing the $DMPKC_{TH}$ values is not the same across different workloads. For instance, workload-4 benefits from increasing the $DMPKC_{TH}$ value (the memory EPR decreases), whereas this increase is harmful for workload-21 since the performance overhead is a dominant factor in this workload.

In another sensitivity experiment, we chose half of the applications in each of our sensitivity analysis workloads to run on a 16-core multicore with 4 MCs. Figure 5.14(c) plots the memory EPR values achieved by HiPEMM for 32-core and 16-core systems. As can be observed from this figure, in most of the workloads, more memory EPR is saved with the 16-core system, since, as the memory load decreases, HiPEMM has the opportunity to cluster the running applications and reduce the channel frequencies with more flexibility.
5.5 Related Work

There have been different approaches to managing and reducing the power and energy consumption of main memory systems. Meisner et al. [112] proposed an energy-saving scheme in which the load on a server determines if the server operates in active or idle mode. This is also the main idea behind a group of prior works on memory energy management. In these works, the memory system has different high and low power modes and the transitions between these modes are performed based on the traffic and activity of the memory system with the goal of reducing energy consumption [113, 114, 115, 116], limiting peak power [117, 116], or avoiding high memory temperatures [118, 119]. [120] and [121] proposed schemes to mitigate the latency overhead incurred by the transitions between different power modes. However, as the number of cores and concurrently running applications increases in multicore, finding the long enough memory idle periods becomes harder. Further, the performance overhead of changing the memory power mode needs to be considered [122] and, as mentioned before, this overhead may differ across different applications based on their sensitivities to the latencies seen by the off-chip memory requests. Yan et al. [123] proposed a scheme where sensitive applications are executed on a group of processors fed by an on-chip regulator. On-chip regulators can do faster power mode transitions than off-chip ones. In the mechanism proposed in [124], the criticality of the off-chip data is specified by the software and the memory refresh rate is reduced for non-critical data to save energy with a low accuracy overhead.

DVFS has been employed as a scheme to reduce the power and energy consumption of main memory systems in modern multicore [104, 105, 106, 107, 102]. In the schemes presented in [104, 106, 125], DVFS is employed for the cores and memory system together to achieve the specified power budget [104] and minimize the full system energy consumption [106, 125]. The memory DVFS schemes employed in these prior studies affect all co-runner applications and the running applications are not distinguished. In comparison, HiPEEMM considers the sensitivity of each application individually and attempts to improve the performance and the efficiency of the energy saving achieved by the memory DVFS.

Other prior works such as [126, 127] proposed new DRAM organizations/architectures.
and data mapping schemes to improve the energy efficiency of the main memory. In the scheme proposed in [127], a conventional DRAM rank is divided into multiple mini-ranks by adding a bridge chip to reduce the number of DRAM chips activated for a memory access. Udipi et al. [126] proposed a novel scheme where the layout of the DRAM arrays are reorganized such that an entire cache-line is fetched from a single sub-array. Note that HiPEMM does not change the DRAM architecture and can be employed in parallel with these schemes.

Page allocation in the main memory has been employed as a mechanism to improve the opportunity of saving memory power and energy [128, 129]. Huang et al. [128] also proposed a mechanism to reshape the memory traffic by migrating the OS pages in order to increase the length and number of memory idle periods. However, page migration incurs performance overhead in multicores and, in their work they do not propose a method to allocate new accessed OS pages.

5.6 Conclusions

Saving energy in the memory system of multicore processors has the potential of significantly reducing the energy use of servers. One method of reducing power is to lower the operating frequency of memory components at the cost of access latency. In this chapter, we propose an agile DVFS technique for dynamically modulating the frequency of memory components based on the sensitivity of running applications. The proposed HiPEMM technique utilizes the DMPKC parameter for monitoring the memory sensitivity of applications for dynamically partitioning the applications to different groups, and modulating the frequency of each group for optimizing energy conservation at the cost of minimal performance penalty. Our experiments with a large number of workloads demonstrate the effectiveness of this energy-saving mechanism for mixed-, high- and low-intensity workloads on various multicore systems. HiPEMM provides around 25% saving in the memory system energy and 10% saving in the total system energy, with only a 3% loss in workload performance.
Chapter 6

Courteous Cache Sharing: Being Nice to Others in Capacity Management

6.1 Introduction and Motivation

Cache sharing can bring benefits, if the threads are scheduled well (by employing schemes such as the ones presented in [130, 131, 132]) for a certain class of applications with extensive data sharing. However, for the other applications with a lesser degree of data sharing, interferences due to shared caches can be a more pressing problem than the advantages that cache sharing brings.

We go over an experiment to illustrate how sharing in the last level cache space can lead to performance variations across the threads that belong to the same multithreaded application. This experiment also demonstrates how the overall system performance can be improved by exploiting these variations. First, consider an eight-core system with each core having a private L1 cache assuming that no L2 cache space is shared among the cores (there is no L2 cache in the system). We ran three multithreaded applications, mgrid (from the SPECOMP benchmark suite [23]), lu and cg (from the NAS benchmark suite [135]) using four, two and two threads, respectively. Each thread is pinned to a core (our simulation configuration is described later in Section 6.3) and all threads are running simultaneously.
Figure 6.1 plots the normalized IPC\textsuperscript{1} values of different threads of the running applications. As can be observed, variations among the IPCs of the threads that belong to the same application are very low (below 2\%). In other words, in a fixed interval of time, the number of instructions executed by the different threads of the same application are similar. This implies that the threads that belong to the same application perform similar amounts of “work” during the course of execution.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure1.png}
\caption{The measured IPCs for three different applications. For each application, bars correspond to different threads. The IPC values are normalized to the maximum IPC achieved by any thread that belongs to the same application.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2.png}
\caption{The measured IPCs of the threads of three co-runner applications on an 8 core multicore. D0, D1, D2 and D3 are mgrid’s threads; U0 and U1 are lu’s threads and G0 and G1 are cg’s threads. Each bar is normalized to the maximum IPC achieved by any thread of the same application.}
\end{figure}

However, if threads from the same application as well as co-runner applications can share the same cache space, performance variations across threads of a given application are possible. To illustrate this, let us assume that our multicore has eight cores in two sockets, each core having a private L1 cache and each socket having a last level L2 cache shared by the four cores in that socket, as illustrated in Figure 6.3(a). We ran the applications used in the previous experiment (mgrid, lu and cg) with the same number of threads on this multicore. In this case, a good mapping strategy is to map the threads of mgrid to P0-P3 and the threads of lu

\textsuperscript{1}We use IPC as our measurement metric in estimating and comparing the execution speed of the threads that belong to the same multithreaded application.
and cg to P4-P7 (see Figure 6.3(a)). This mapping gives the threads that belong to mgrid the opportunity to exploit inter-thread data sharing in the L2 cache space, and also isolates this application from the other two. However, this mapping may not always be possible when different applications have different entry/exit times. For example, if lu and cg arrive earlier than mgrid for execution and their threads are mapped to P2-P3 and P6-P7 respectively (Figure 6.3(b)); there are two options for mgrid. The first one is to migrate the threads of applications lu or cg to the cores sharing a last level cache and map the threads of mgrid to the other cores sharing a last level cache (as in Figure 6.3(a)). However, thread migration can impose significant performance overheads, since the migrated threads suffer from loosing their cache locality. The other option would be mapping the threads of the incoming application mgrid to the available free cores, as shown in Figure 6.3(c).

As stated above, since various applications with different numbers of threads may execute and terminate at arbitrary times during the course of execution, threads might be mapped to the cores of a multicore in different ways.

Figure 6.3. Different mappings of the threads that belong to three multithreaded applications running on an eight-core multicore. P0 through P3 denote cores, and D0-3, U0-1 and G0-1 are mgrid’s, lu’s and cg’s threads, respectively.

Figure 6.2 plots the measured IPCs of the threads that belong to our three co-runner applications. The thread mapping and the multicore cache configuration are assumed to be similar to Figure 6.3(c) (i.e, D0, D1, U0 and U1 sharing a common L2 cache and D2, D3, G0 and G1 sharing another L2 cache). As can be observed from Figure 6.2, the paces of the four threads of mgrid (D0-D3) are not all the same. This is because, in this case, U0 and U1 are more memory intensive than G0 and G1, and consequently, D2 and D3 can take better advantage of the shared L2 space. Since the performance and the execution speed of a data-parallel multithreaded application is mainly dictated by its slowest thread, the fact that D2 and D3 are very fast is not particularly beneficial to this application as far as its overall
performance is concerned. To summarize, while different threads of a data-parallel application exhibit similar behavior when executing in isolation, they can exhibit different behaviors in an environment where they share a common cache space with threads of different applications; that is, different threads of a multithreaded application can experience different magnitudes of destructive interferences from their co-runners.

There are at least two ways in which this situation can be improved: (i) The performance of \( cg \) can be enhanced by giving lower priorities to D2 and D3 (as long as either D2 or D3 is not the slowest thread in the application) and correspondingly higher priorities to G0 and G1 in using the shared L2 cache. (ii) The performance of \( mgrid \) can be enhanced by assigning lower priorities to U0 and U1 (as long as the contribution of resultant improved performance of \( mgrid \) supersedes the degradation in \( lu \)'s performance) and correspondingly higher priorities to D0 and D1 in using the shared L2 cache. In general however, in a large multicore with many concurrently-running multithreaded applications, finding the right solution may not be at all easy.

Our goal in this chapter is to design a configurable shared cache management scheme that provides high system throughput without requiring significant coordination between multiple sockets. To this end, targeting multiprogrammed, multithreaded workloads, we develop a fundamentally new approach to shared cache management, called CCS (Courteous Cache Sharing), with the following key contributions:

- CCS attempts to balance the execution speeds of different threads that belong to the same data-parallel multithreaded application in a multiprogrammed workload. This generally results in de-prioritizing the faster threads of an application, which in turn generates a slack that can be used to expedite threads of other applications sharing the same cache.

- CCS uses the newly created slack to improve overall system performance by improving the performance of applications that can benefit from the additional slack.

- We propose a simple mechanism to manage the priority of each running thread in using the shared L2 cache space.
We evaluate our cache management scheme on a variety of multiprogrammed, multithreaded workloads on a range of multicore systems with 8-16 cores and 2-8 L2 caches with different thread-to-core mappings for each workload.

Averaged over six workloads, our shared cache management scheme improves the performance of the combination of applications by 18%. These improvements across applications in each mix are also fair, as indicated by average fair speedup improvements of 10% across the threads of each application (averaged over all the workloads).

The rest of this chapter is organized as follows. The next section presents our proposed scheme (CCS) and its implementation issues and overheads. Our experimental evaluation for different configurations is presented in Section 6.3. Section 6.4 discusses the related studies and finally, we conclude the chapter in Section 7.5.

6.2 Our proposed scheme

We first describe a simple, yet novel prioritization scheme we employ to control the performance of each running thread by means of assigning different priorities to different cores (application threads\textsuperscript{2}) in evicting cache lines in the shared L2 cache space. After that, we describe the details of our CCS approach.

6.2.1 Prioritization Scheme

Cache partitioning has been proposed as a mechanism to prioritize applications by allocating different capacities of the shared cache space to co-runner applications [99, 136, 24, 19]. For instance, in the utility-based partitioning scheme [99], different numbers of cache ways in a set are allocated to the cores sharing a set-associative cache. There are at least three significant issues in adopting a cache partitioning scheme like [99], [136] or [19] for multiprogrammed workloads of multithreaded applications:

- The prior cache partitioning schemes do not consider the variations in execution speeds of different threads of an application, that may lead to the overall

\textsuperscript{2}We consider different one-to-one mappings of threads to cores.
performance degradation. As an example, consider that a cache partitioner is employed to manage the L2 caches in Figure 6.3(c). Consider further that $D_2$ and $D_3$ have better cache utility than $G_0$ and $G_1$ and, as a result, the policy of the cache partitioner would be allocating more cache space to $D_2$ and $D_3$ at runtime in a bid to improve the weighted speedup of the threads. However, this strategy is not beneficial to the overall system performance, since the cache partitioner does not take into account the execution speeds of $D_0$ and $D_1$, which are the bottleneck threads as far as $D$ (mgrid) is concerned. Consequently, the performance of $G_0$ and $G_1$ would degrade without improving the performance of $D$.

- To remedy the thread-obliviousness of existing cache partitioning schemes, one would need detailed performance models for each thread of each application. Each of these performance models further needs to maintain a predicted performance curve for different numbers of allocated cache ways. The area overheads of a hardware implementation of such a scheme would be prohibitively expensive in practice. That is, straightforward extensions of existing cache management schemes to multithreaded, multiprogrammed workloads can be very costly.

- In general, the reaction time of cache partitioning is very slow. Any change in cache partitioning is effective once in a long interval/epoch and multiple such epochs would be necessary to notice any changes in application performance as a result of the change in partitioning.

To alleviate these problems, in CCS, we employ a prioritization mechanism which is able to manage the shared cache space with finer granularity and less overheads. Our prioritization scheme requires a simple change in the LRU eviction priority. In a normal LRU algorithm, if a cache miss occurs, the cache line which is accessed least recently is evicted from the set indexed by the memory access request. In our prioritization scheme, on the other hand, each core is assigned a positive integer number as its priority (weight). First, based on the weights assigned to different cores, a core is selected as the victim from among the ones having cache lines in the corresponding set, and then, the LRU cache line is selected
from the lines that belong to this victim core. Algorithm 2 gives the high level view of our victim core selection strategy.

As an example, consider four cores sharing an L2 set-associative cache with core weight values of $w_1, w_2, w_3$ and $w_4$. Assuming further that all four cores have some cache lines in the corresponding set (determined by line 5 in Algorithm 2), once a cache miss happens, a probability is assigned to each core based on the weight values (line 11). This represents the probability with which each core may be selected as the victim core. For instance, core 1 is chosen as the victim with $\frac{w_1}{\sum w_i}$ probability. Finally, the victim core is selected based on these probability values (lines 13 through 22). If a core is assigned a larger weight value, there is a higher chance that this core will be selected as the victim and, as a result, less cache space will be allocated to it at runtime. In CCS, we use these priority levels to classify application threads and manage the shared cache space.

**Algorithm 2 Victim Core Selection**

<table>
<thead>
<tr>
<th>Input: The index of the set in the cache: set_num</th>
<th>Output: Victim Core: core</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: cache_lines = cacheLines(set_num)</td>
<td>1: core_lines = cacheLines(set_num)</td>
</tr>
<tr>
<td>2: cores = {}</td>
<td>2: cores = {}</td>
</tr>
<tr>
<td>3: sum = 0</td>
<td>3: sum = 0</td>
</tr>
<tr>
<td>4: for each cache_line in cache_lines do</td>
<td>4: for each cache_line in cache_lines do</td>
</tr>
<tr>
<td>5: cores.add(cache_line.core)</td>
<td>5: cores.add(cache_line.core)</td>
</tr>
<tr>
<td>6: end for</td>
<td>6: end for</td>
</tr>
<tr>
<td>7: for each core in cores do</td>
<td>7: for each core in cores do</td>
</tr>
<tr>
<td>8: sum = sum + core.weight</td>
<td>8: sum = sum + core.weight</td>
</tr>
<tr>
<td>9: end for</td>
<td>9: end for</td>
</tr>
<tr>
<td>10: for each core in cores do</td>
<td>10: for each core in cores do</td>
</tr>
<tr>
<td>11: probability[core.num] = (\frac{core.weight}{\sum w_i})</td>
<td>11: probability[core.num] = (\frac{core.weight}{\sum w_i})</td>
</tr>
<tr>
<td>12: end for</td>
<td>12: end for</td>
</tr>
<tr>
<td>13: random = generateRandom(0,1)</td>
<td>13: random = generateRandom(0,1)</td>
</tr>
<tr>
<td>14: partial_sum = 0</td>
<td>14: partial_sum = 0</td>
</tr>
<tr>
<td>15: for each core in cores do</td>
<td>15: for each core in cores do</td>
</tr>
<tr>
<td>16: if partial_sum ≥ random then</td>
<td>16: if partial_sum ≥ random then</td>
</tr>
<tr>
<td>17: return core</td>
<td>17: return core</td>
</tr>
<tr>
<td>18: else</td>
<td>18: else</td>
</tr>
<tr>
<td>20: end if</td>
<td>20: end if</td>
</tr>
<tr>
<td>21: end for</td>
<td>21: end for</td>
</tr>
</tbody>
</table>

### 6.2.2 CCS Architecture

This section explains the main contribution of this work, a novel cache space management scheme targeting multiprogrammed, multithreaded workloads that consist of data-parallel applications. The key insight in CCS is that, the overall perfor-
mance of a data-parallel multithreaded application is determined by its slowest thread, and therefore, we can afford to de-prioritize the faster threads of one application, while prioritizing the slower threads of another application sharing the same cache. In this section, we discuss CCS for a sample architecture with eight cores, in which a set of four cores in a socket share a common L2 cache space (see Figure 6.4). Note that CCS works similarly for the architectures with different number of cores and different on-chip cache topologies. In Section 6.3, we present experimental results for different types of multicore architectures.

Figure 6.4 illustrates the high-level view of CCS. During the course of execution, CCS monitors the measured IPCs (our performance metric) of the threads that belong to each running application and, based on that, adjusts the weight value of each core (eviction weights of the cores). As can be observed from Figure 6.4, the Arbiter component takes the measured IPCs of the cores as input and determines a rank value for each of the cores. These rank values (which will be used to compute the weights) are obtained as follows:

$$\text{rank}_i = \frac{\text{IPC}_i - \text{MIN}}{\text{MIN}},$$  

(6.1)

where \(\text{rank}_i\) and \(\text{IPC}_i\) are, respectively, the rank value and the measured IPC of core \(i\). Assuming \(A\) is the application to which the thread running on core \(i\) belongs, \(\text{MIN}\) is the minimum IPC of the \(A\)’s threads. In other words, rank values capture how fast each thread is, as compared to the slowest thread of the same multithreaded application. If the rank value for core \(i\) is zero, it indicates that the thread mapped to this core has the smallest IPC across all threads that belong to the same application. Algorithm 3 shows how the rank of each core is computed based on the mappings of the running threads. In this algorithm, \(\text{mapping}[i]\) indicates the application that the thread mapped to core \(i\) belongs to.

In CCS, other than the Arbiter module, an L2 Manager is associated with each shared L2 cache in the system. An L2 Manager takes the ranks of the threads running on the connected cores as input, and adjusts the eviction weights of the running threads accordingly. For instance, in Figure 6.4, the ranks of the threads mapped to P0-P3 and produced by the Arbiter constitute the inputs to the L2-1 Manager. Based on these ranks, at the \(k\)th time interval, the L2-1 Manager increases the eviction weight of the thread (threads) with the highest rank value.
**Algorithm 3** Operation of the Arbiter Component

**Input:** The measured IPCs $IPC[1 \cdots n]$

**Input:** Thread mapping $mapping[\cdot]$ 

**Output:** Ranks of the cores $R[1 \cdots n]$

1: for $i = 1$ to $n$ do
2: $app = mapping[i]$
3: $min = \infty$
4: for $j = 1$ to $n$ do
5: if ($IPC[i] < min$) and ($mapping[j] == mapping[i]$) then
6: $min = IPC[i]$
7: end if
8: end for
9: $R[i] = \frac{IPC[i] - min}{min}$
10: end for

and reduces the eviction weight of the thread (threads) with the lowest rank value by the amount of $\delta$ as shown below:

$$w_i(k) = w_i(k - 1) + \begin{cases} 
\delta, & \text{if } i \in \text{MAX} \\
-\delta, & \text{if } i \in \text{MIN}, 
\end{cases}$$

(6.2)

where $w_i(k)$ is the eviction weight of core $i$ at the $k$th time interval, and $MIN$ and $MAX$ are the sets of minimum and maximum rank values, respectively, received by the L2 Managers. That is, eviction weights at the $k$th interval are computed using the rank values as well as the eviction weights used at the $(k - 1)$th interval.
Algorithm 4 shows how the eviction weights are adjusted by the L2 Managers. Note that, if an eviction weight reaches 1, since we assume that eviction weights are positive, instead of reducing this weight, the weights for the other cores connected to the same L2 cache are increased (as can be observed in lines 21 through 25).

Algorithm 4 Operation of an L2 Manager

Input: Ranks of the cores \( R[m \cdot \ldots \cdot m'] \)
Output: Eviction weights of the cores at time \( k \) \( W_k[1 \cdot \ldots \cdot n] \)

1: for \( i = m \) to \( m' \) do
2: \( \text{rank} = R[i] \)
3: \( \text{increaseFlag} = 1 \)
4: \( \text{decreaseFlag} = 1 \)
5: for \( j = m \) to \( m' \) do
6: if \( R[j] < \text{rank} \) then
7: \( \text{decreaseFlag} = 0 \)
8: end if
9: if \( R[j] > \text{rank} \) then
10: \( \text{increaseFlag} = 0 \)
11: end if
12: \( W_k[i] = W_{k-1}[i] \)
13: if \( (\text{increaseFlag} == 1) \) and \( (\text{decreaseFlag} == 0) \) then
14: \( W_k[i] = W_{k-1}[i] + \delta \)
15: end if
16: if \( (\text{increaseFlag} == 0) \) and \( (\text{decreaseFlag} == 1) \) then
17: \( W_k[i] = W_{k-1}[i] - \delta \)
18: end if
19: if \( (W_k[i] < 1) \) then
20: \( \text{for} \ c = m \text{ to } m' \text{ do} \)
21: \( \text{if} \ c \neq i \text{ then} \)
22: \( W_k[c] = W_k[c] + \frac{1-W_k[i]}{m'-m} \)
23: \( \text{end if} \)
24: \( \text{end for} \)
25: end if
26: end if
27: end for

CCS first classifies the co-runner threads into slow cores and fast ones (responsibility of the Arbiter) and then, reduces the eviction weights of the bottleneck threads (or the threads that will be likely bottleneck) to enhance their performance (responsibility of the L2 Managers). At the same time, it increases the weights of the fast threads (cores), so that they are no faster than the new slowest thread. To illustrate this, let us assume that three applications \( X \) (with four threads), \( Y \) (with two threads) and \( Z \) (with two threads) are executing on an eight-core architecture shown in Figure 6.3(c) (see the first two columns of Table 6.1 for thread-to-core mappings). Suppose that the measured IPCs at the \( k \)th time interval are as shown in the third column of Table 6.1. Firstly, the thread ranks (in the order of their individual speeds) are determined by the Arbiter. As can be seen in Table 6.1, the slowest thread of application \( X \) is \( X_1 \) and, conse-
quently, the ranks of application $X$’s threads are obtained based on the $X_1$’s IPC (see the forth column of Table 6.1). After the rank is determined by the Arbiter, the L2 Managers select the candidate threads to increase (decrease) their eviction weights. In our example, the L2-1 Manager increases the weight of P0 (the core that $X_0$ is running on) and reduces the weight of P1 (which has the least rank) by $\delta$ ($\delta = 2$ in this example). Similarly, the L2-2 Manager increases the weight of P6 and reduces the eviction weights of P5 and P7, with the goal of accelerating the slow threads and slowing down the fast ones dynamically to improve the overall system performance.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Cores</th>
<th>Normalized</th>
<th>IPCs</th>
<th>Ranks</th>
<th>Weights Before</th>
<th>Weights After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_0$ P0</td>
<td>0.8</td>
<td>0.66</td>
<td>35</td>
<td>37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_1$ P1</td>
<td>0.48</td>
<td>0</td>
<td>15</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Y_0$ P2</td>
<td>1.0</td>
<td>0.5</td>
<td>30</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Y_1$ P3</td>
<td>0.61</td>
<td>0.33</td>
<td>25</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_2$ P4</td>
<td>1.0</td>
<td>0.1</td>
<td>30</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_3$ P5</td>
<td>0.9</td>
<td>0</td>
<td>25</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Z_0$ P6</td>
<td>1.0</td>
<td>0.14</td>
<td>35</td>
<td>37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Z_1$ P7</td>
<td>0.87</td>
<td>0</td>
<td>20</td>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1. An example that involves three applications ($X$, $Y$ and $Z$) and 8 cores.

### 6.2.3 Implementation Issues and Overheads

CCS is implemented by the OS with hardware support. At each interval, the OS runs our algorithm based on the threads’ mappings and the IPC values read from the hardware counters. After that, the eviction weights of the cores are determined (as explained above) and sent to the modified LRU algorithm implemented in hardware. If a cache miss occurs, the modified LRU algorithm first determines the victim core, and evicts the least recently used L2 cache block from the blocks that belong to that victim core. For an $m$-core machine, a $\lceil \log_2(m) \rceil$-bit tag is associated with each cache block indicating the core it belongs to.

The OS runs CCS implemented in software (both Arbiter and L2 Managers) to compute the weight of each core based on the current IPC values. The CCS algorithm for a multicore with 16 cores takes about 6K cycles to run, which, on a 2.1GHz machine, represents a negligible overhead. Note that the value of the weight of each core is not increased beyond the saturation points. However, the weights of the cores attached to the same L2 cache can be scaled down by the minimum weight value to avoid saturation. It is also to be noted that this overhead is
not incurred once in every replacement, but only once in every enforcement interval (e.g., every 200 million cycles in our experiment), when the eviction priorities of cores are recomputed.

In CCS, a pseudo-LRU tree is maintained for each core instead of a global one. Firstly, the victim core is selected based on the cores’ weights, and then the LRU cache block is determined based on the victim core’s pseudo-LRU tree. Therefore, choosing the victim core (as given in Algorithm 2) is the only overhead imposed by CCS in the LRU algorithm.

6.2.4 Discussion

One can broadly divide multithreaded applications into these classes:

**Data parallelism.** In our work, we focus on OPENMP based, data-parallel applications where the threads that belong to the same application perform very similar tasks on different sets of data. In these types of applications, the thread with the lowest IPC will be the bottleneck and other threads will have to wait for this thread to join at synchronization barriers. CCS balances the execution speeds of the co-runner threads based on the measured IPC values.

**Pipeline parallelism.** In applications with this parallelization model, different threads execute different functions and work as a pipeline. In this pipeline architecture, thread pools are dedicated to each parallelized stage [137]. CCS can be employed in these applications through balancing the threads dedicated to the pipeline stages. Further, for the threads that belong to different stages, a synergy of our scheme can be established with the load balancing scheme in place, so that our scheme can be informed about the expected load on each thread. Once the relative amount of work that each thread has to do is known, our scheme can determine the existing slack and work towards the same objective.

**Master-slave model.** In this category of parallel multithreaded applications, typically one of the threads works as the master thread and controls the data distribution among the other co-runner threads. Consequently, the measured performance (IPC) of the master thread differs from the others and needs to be excluded from the CCS algorithm.

This chapter focuses on evaluating our approach in data parallel applications.
To use our approach in pipeline and master-save parallelism, some enhancements are needed. Specifically, in the pipeline parallelism based applications the threads that belong to the same pipeline stage need to be detected. For the master-slave model, master threads should be identified by the OS and not included in CCS.

Also, in execution scenarios where multiple threads are scheduled on the same core (single core), our approach can seamlessly handle this because of the scheduling interval of processes being much longer than the granularity of replacement prioritization in our approach. Further, in order to extend our scheme to simultaneous-multithreading (SMT) processors that share last level caches, we only need to keep track of each hardware thread context as an independent core and treat each the same way as a different core in a multicore.

6.3 Experiments

In this section, we present an experimental evaluation of our proposed approach under different workloads and mappings.

Setup and workloads. We use SIMICS [29] as our simulation framework to evaluate CCS. SIMICS [29] is a full-system simulator that allows simulation of multiprocessor architectures. Table 6.2 gives the baseline configuration we use for our experimental evaluation.

<table>
<thead>
<tr>
<th>Processors</th>
<th>(16 or 8) processors with private L1 data and instruction caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Model</td>
<td>4-way issue superscalar</td>
</tr>
<tr>
<td>Private L1 D-Caches</td>
<td>Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency</td>
</tr>
<tr>
<td>Private L1 I-Caches</td>
<td>Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency</td>
</tr>
<tr>
<td>Shared L2 Cache</td>
<td>64 bytes block size, 10 cycle access latency</td>
</tr>
<tr>
<td>Memory</td>
<td>4GB, 200 cycle off-chip access latency</td>
</tr>
<tr>
<td>Control Enforcement Interval $\delta$</td>
<td>200 Million cycles</td>
</tr>
</tbody>
</table>

Table 6.2. Baseline configuration.

We performed our experiments targeting multicore with 8 and 16 cores and different workloads (application mixes). Table 6.3 gives the application mixes we use in our experiments. We formed our mixes using the applications from the NAS [135], SPLASH [138] and SPECOMP [23] benchmark suites. The benchmarks used are the ones that are sensitive to the variations in the L2 cache space. We formed different three-application workloads from these benchmarks, and evaluated them. In this section, we present results with six representative workloads. Note that,
these applications may be executed with different numbers of threads.

After an application is initiated, based on the one-to-one mapping strategy
specified, we bind its threads to available cores. To do this in Solaris 10, we use
the \texttt{pbind} command.

\textbf{Evaluation metrics.} We use \textit{weighted speedup} as our metric to evaluate the
behavior of CCS. Weighted speedup indicates the average performance speedup
achieved across the applications of a workload. Assuming equal weights, the
weighted speedup is computed for an \textit{N}-application mix as follows:

\begin{equation}
WS = \frac{\sum_{i} IPC_{\text{scheme}}(A^i)/IPC_{\text{base}}(A^i)}{N},
\end{equation}

\begin{equation}
IPC(A^i) = \min \{ IPC(\text{thread}_i) | \text{thread}_i \in A^i \},
\end{equation}

where $IPC_{\text{scheme}}(A^i)$ is the IPC value of application $A^i$ when CCS is employed. In
our experiments, to measure the performance improvement achieved by CCS, our base case is when no scheme is used to manage shared caches ($IPC_{\text{base}}(A^i)$ is the IPC value of $A^i$ when no scheme is employed). Note that, these values ($IPC_{\text{scheme}}(A^i)$ and $IPC_{\text{base}}(A^i)$) are determined by the slowest thread that belongs to $A^i$.

\textit{Average fair speedup} is the other metric we employ to quantify the behavior of
CCS, and can be computed as follows:

\begin{equation}
Average, FS = \frac{FS_1 + \cdots + FS_N}{N},
\end{equation}

\begin{equation}
FS_{i} = \frac{N_j}{\sum_{k=1}^{j} IPC_{\text{base}}(\text{thread}_k)/IPC_{\text{scheme}}(\text{thread}_k)},
\end{equation}

where $FS_{i}$ is the fair speedup achieved across the $N_j$ threads that belong to
application $A^i$. \textit{Average fair speedup} is the average of the speedsups obtained for the
$N$ applications in each workload. Fair speedup is also a measure of \textit{fairness} [19].

\textbf{Experimental configurations.} We performed experiments with different con-
figurations listed in Table 6.4 and shown in Figure 6.7. As can be seen, these
configurations differ from one another in terms of the number of cores, number
of caches, number of application threads, or the value of a specific experimental
parameter. Our baseline configuration is the one shown for Experiment-1.
<table>
<thead>
<tr>
<th>Workload</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mix-1</td>
<td>mgrid, bt, ocean</td>
</tr>
<tr>
<td>Mix-2</td>
<td>swim, ft, lu</td>
</tr>
<tr>
<td>Mix-3</td>
<td>swim, mg, sp</td>
</tr>
<tr>
<td>Mix-4</td>
<td>bt, sp, ocean</td>
</tr>
<tr>
<td>Mix-5</td>
<td>mgrid, lu, cg</td>
</tr>
<tr>
<td>Mix-6</td>
<td>mg, ft, cg</td>
</tr>
</tbody>
</table>

Table 6.3. Various mixes of applications considered for our multithreaded, multiprogrammed workloads.

<table>
<thead>
<tr>
<th>Experiments</th>
<th># of cores</th>
<th># of L2s</th>
<th># of cores/L2</th>
<th>L2 size/core</th>
<th># of Apps</th>
<th># of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiment-1</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>512KB</td>
<td>3</td>
<td>&lt;6,6,4&gt;</td>
</tr>
<tr>
<td>Experiment-2</td>
<td>16</td>
<td>2,8</td>
<td>8,2</td>
<td>512KB</td>
<td>3</td>
<td>&lt;6,6,4&gt;</td>
</tr>
<tr>
<td>Experiment-3</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>1MB,256KB</td>
<td>3</td>
<td>&lt;6,6,4&gt;</td>
</tr>
<tr>
<td>Experiment-4</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>512KB</td>
<td>3</td>
<td>&lt;4,2,2&gt;</td>
</tr>
<tr>
<td>Experiment-5</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>512KB</td>
<td>2,4</td>
<td>&lt;8,8&gt;,&lt;4,4,4,4&gt;</td>
</tr>
<tr>
<td>Experiment-6</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>512KB</td>
<td>3</td>
<td>&lt;6,6,4&gt;</td>
</tr>
<tr>
<td>Experiment-7</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>512KB</td>
<td>3</td>
<td>&lt;6,6,4&gt;</td>
</tr>
</tbody>
</table>

Table 6.4. Our experimental configurations.

**Experiment-1.** In this experiment, we evaluate a case in which the workloads given in Table 6.3 are executed on a sixteen-core system. The shared cache configuration for this experiment is illustrated in Figure 6.7(a), where each L2 cache space in a socket is shared by four cores. The applications in each mix execute using 6, 6 and 4 threads in the order given by Table 6.3 (for instance, mgrid, bt and ocean in Mix-1 are executed using 6, 6 and 4 threads, respectively).

In this case (a multicore with 16 cores and 4 L2 caches in 4 sockets), assuming all 16 threads are different, \(\frac{16!}{4!\times4!\times4!\times4!}\) different thread-to-core mappings are possible. On the other hand, if we assume that the threads are distinguished by the application which they belong to, the number of different mapping combinations becomes much smaller. For instance, to compute this number in our default configuration (in which three applications A, B and C, are running using 6, 6 and 4 threads), each L2 cache, L2i, can be associated with a triple \(<a_i, b_i, c_i>\). This triple indicates that \(a_i\), \(b_i\) and \(c_i\) number of threads from applications A, B and C are mapped to the cores connected to this \(L2_i\) such that \(a_i + b_i + c_i = 4\).

Consequently, there exists \(\frac{(4+(3-1))!}{4!(3-1)!}\) = 15 different possible combinations for each triple. Note that, each thread mapping is a selection of 4 triples from these 15 combinations such that following constraints are satisfied:

\[
\sum_{i=1}^{4} a_i = 6; \quad \sum_{i=1}^{4} b_i = 6; \quad \sum_{i=1}^{4} c_i = 4.
\]  

(6.7)

Therefore, the number of different mappings for our default configuration is
34, considering the above constraints. Current generation operating systems do not always make the most intelligent choices with thread allocation. Therefore, it is important to evaluate the robustness of the scheme with multiple thread mappings. We do not assume that all these mappings are equally likely, but are possible nevertheless.

Figures 6.5 and 6.6 plot the weighted speedups achieved by CCS for Mix-2 and Mix-5, respectively, using all these 34 different thread mappings. We note that, CCS improves the overall system performance by as much as 30%. While the average improvement is about 12%, the achieved performance improvement varies significantly across different thread mappings. This is because, as stated earlier, CCS attempts to assign higher priorities to the bottleneck (slowest) threads of different applications. Therefore, if the number of the bottleneck threads sharing the same L2 cache increases during execution, CCS may not be able to improve the overall system performance significantly. In other words, the bottleneck threads of different applications compete with each other to make use of the L2 cache space and increasing the priority of one thread may hurt the performance of another thread that belongs to a different application.

Our remaining experiments are performed using six representative mappings that exhibit different cache sharing patterns. These mappings are shown in Ta-
Table 6.5, where \( X_i, Y_i \) and \( Z_i \) represent the threads that belong to the applications shown in Table 6.3 in the same order for each mix. As an example, assuming that Map-1 is used to map Mix-1 applications, four threads of \textit{mgrid} are bound to cores 1 to 4 sharing a common L2 cache and the other two threads are bound to the cores 5 and 6 sharing an L2 cache with two threads of \textit{bt} (\( B_0 \) and \( B_1 \)).

![Multicore configurations](image)

**Figure 6.7.** Different multicore configurations tested in our experiments.

<table>
<thead>
<tr>
<th>Mappings</th>
<th>SharingPattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map-1</td>
<td>( &lt; (X_0, X_1, X_2, X_3), (X_4, X_5, X_6, Y_1), (Y_2, X_3, Y_4, Z_2), (Z_0, Z_1, Z_2, Z_3) &gt; )</td>
</tr>
<tr>
<td>Map-2</td>
<td>( &lt; (X_0, X_1, Y_1, Z_1), (X_2, X_3, Y_2, Y_3), (X_4, X_5, Y_4, Z_5), (Y_6, X_7, Z_1, Z_2) &gt; )</td>
</tr>
<tr>
<td>Map-3</td>
<td>( &lt; (X_0, Y_1, X_2, X_3), (Y_1, Z_2, X_4, Y_5), (X_6, Y_7, Y_8, Z_6), (Z_7, Z_8, Z_9, Z_10) &gt; )</td>
</tr>
<tr>
<td>Map-4</td>
<td>( &lt; (X_0, X_1, X_4, Z_2), (X_3, X_4, X_5, Z_6), (Y_1, Y_2, Y_3, Z_4), (Y_5, Y_6, Z_7, Z_8) &gt; )</td>
</tr>
<tr>
<td>Map-5</td>
<td>( &lt; (X_0, X_1, X_2, Z_3), (X_2, X_3, Z_4, Y_5), (X_6, X_7, Z_8, Z_9), (Z_10, Z_11, Z_12, Z_13) &gt; )</td>
</tr>
<tr>
<td>Map-6</td>
<td>( &lt; (X_0, Y_0, X_1, Z_2), (Y_1, X_2, Z_3, Y_4), (X_5, Z_6, Y_7, Y_8), (Z_9, Y_10, Z_11, X_12) &gt; )</td>
</tr>
</tbody>
</table>

Table 6.5. Mappings used in 16 cores experiments. \( X_i, Y_i \) and \( Z_i \) are the threads that belong to the applications shown in Table 6.3 in the same order for each mix.

Note that, the representative mappings given in Table 6.5 cover different cache sharing patterns, from a mapping that offers maximum sharing across the threads from the same application (Map-1) to a mapping with the maximum L2 cache contention among the co-runner threads (Map-6). The sharing pattern for each mapping in our default configuration (16 cores and 4 L2 caches in 4 sockets) is shown in Table 6.5.\(^3\)

Figure 6.8(a) plots the weighted speedups achieved in our workloads under these six different mappings. We run the workloads for 10 billion cycles and, over this period, CCS is invoked 50 times (every 200 million cycles). In this figure, the six bars presented for each mix correspond to the weighted speedups achieved across the applications of that mix for six different mappings. Note that the achieved performance improvements depend on the chosen mapping and also the running workload. As can be observed, CCS improves the performance.

\(^3\text{<a,b,c,d>}\) shown in the third column indicates that the four L2 caches in our default configuration are shared among the threads from \( a, b, c \) and \( d \) applications.
of Mix-2, Mix-3, Mix-5 and Mix-6 workloads significantly through speeding up their bottleneck threads. Among the representative mappings, there is a minimum contention across the threads of different applications in Map-1, since the threads that belong to the same applications share common L2 slices in this mapping. Consequently, as can be observed from Figure 6.8(a) the speedups achieved when using Map-1 are not as high as those obtained with other mappings. Note that, as mentioned before, while Map-1 would be the most beneficial choice, it cannot always be done by the OS.

As one can observe in Figure 6.8(b), CCS also improves the performance of the individual threads of an application across different mixes in most of the cases. This is because, although CCS slows down some of the fast threads, it compensates for the performance loss through enhancing the bottleneck threads in the long run. It is further observed that the average fair speedups achieved with mixes such as Mix-4 and Mix-3 is less than the weighted speedups reported in Figure 6.8(a). As stated earlier, in computing the weighted speedup metric, the IPC of each application is determined by the slowest thread. CCS attempts to improve the performance of the bottleneck threads, which may in turn lead to performance degradation for the faster ones. Consequently, the average fair speedups may be less than the achieved weighted speedups in such cases.

There are two main factors affecting the magnitude of the savings achieved by CCS. The first one is the sensitivity of the running applications to the assigned eviction weights. For instance, performances of the applications in Mix-1 and Mix-4 are not sensitive to the cache allocation as much as the performance of the other applications and, as a result, the performance cannot be improved significantly

**Figure 6.8.** The performance improvement achieved for the applications’ threads of each mix (Experiment-1, 16 cores, 4 L2 caches). For each mix, bars correspond to the six different mappings.
for these mixes, as shown in Figures 6.8(a) and 6.8(b). The second factor that shapes the performance improvement is thread mapping. If the number of bottleneck threads sharing the same L2 cache increases during execution (e.g., when the slowest threads of different applications happen to share the same L2), CCS may not be able to improve the overall system performance.

Figure 6.9 plots the dynamic adjustment of the eviction weights of different threads of the applications in Mix-2 when Map-2 is used. As one can observe from these graphs, the eviction weights (priorities) assigned to threads of a given application are modulated dynamically during execution to adapt to the changes in the dynamic behavior of application threads. Note also that, assigning the same priority values to the threads that belong to the same application, but executing in different sockets does not necessarily guarantee that the threads will run with the same speed.

Figure 6.10 plots the measured IPCs of the 4 threads of `lu` (in Mix-2), during execution. At each time interval, CCS adjusts the eviction weights of these threads (LU0 through LU3) based on the measured IPC values, as shown in Figure 6.9. For instance, as can be observed from Figure 6.9(c), in the 25th time epoch (marked using an oval), CCS reduces and increases the eviction weights of LU0 and LU1, respectively, since LU0 and LU1 have the lowest and highest performance among LU’s threads at this time (see the oval in Figure 6.10).

Figure 6.9. Dynamic adjustment of the eviction weights of the Mix-2’s threads using Map-2 (each line corresponds to a thread).

**Experiment-2.** In this experiment, we vary the number of cores attached to each L2 cache in a 16-core system (while keeping the per core cache capacity the same as in Experiment-1). Figure 6.7(b) illustrates a multicore with 16 cores in which every two cores sharing an L2 cache space in a socket (1MB, 16-way set associative). To
employ CCS, eight L2 Managers are associated with the L2 caches (see Figure 6.4).

Figures 6.11(a) and 6.12(a) plot the weighted speedup and average fair speedup values achieved for our application mixes using the mappings given in Table 6.5 (as before, for a given mix, each bar corresponds to a different mapping). In this experiment, CCS has more flexibility, since the probability with which the bottleneck threads exercise the same L2 cache is reduced. Consequently, the speedups achieved in this experiment are generally higher than those reported in Experiment-1.

Figures 6.11(b) and 6.12(b) give, respectively, the weighted speedup and average fair speedup values, when running our mixes on a multicore with 16 cores and two shared L2 caches (as shown in Figure 6.7(c)). Note that, as the number of cores sharing an L2 cache increases, the performance improvement achieved by CCS reduces due to the increased conflicts across the threads of different applications. In other words, over the course of execution, the likelihood that the bottleneck threads from different applications share a common cache increases.
Figure 6.12. The fair speedup values achieved for L2 count sensitivity analysis (Experiment-2).

Figure 6.13. Cache capacity sensitivity analysis (Experiment-3, 1MB/core).

which, in turn, affects the effectiveness of CCS.

**Experiment-3.** As noted earlier, in our baseline configuration, the size of the L2 caches is 512KB per core (i.e, if four cores are attached to an L2 cache, its size is set to be 2MB). In this experiment, we vary the L2 cache capacity per core to 1MB and 256KB. Figures 6.13 and 6.14 give the weighted speedup and fair speedup values for these cases. As can be observed, CCS improves the overall performance significantly in several cases. Note that the results reported in these figures are comparable as CCS is able to enhance the performance of the workloads executed on the multicores with different cache capacities. When we run our workloads on a multicore with large shared caches, if the cache access contention among the co-runner threads is low, we may achieve better performance improvement by reducing the cache capacity (as in the cases of Mix-3 when using Map-2 and Map-4), since the running applications take better advantage of our shared cache management mechanism.

**Experiment-4.** We also evaluated CCS in a multicore with smaller number of cores (8 cores) sharing two L2 caches in two sockets (see Figure 6.7(d)). Figures 6.15(a) and 6.15(b) plot the weighted speedups and average fair speedups
achieved for our mixes under three different mappings. These mappings are shown in Table 6.6. We see that, in Mix-1, the performance improvement achieved with Map-3 is about 10%, whereas we cannot achieve any improvement when Map-1 or Map-2 is used. This is because $X_0$ ($mgrid_0$) is the bottleneck thread of $mgrid$ and is not compatible with $X_2$ ($mgrid_2$) and $Y_1$ ($bt_1$) as far as sharing L2 capacity is concerned under Map-3; our proposed scheme has the opportunity to speedup $X_0$ ($mgrid_0$) and subsequently improve overall performance. The overall system performance is improved significantly in Mix-2 when Map-2 is used since, in this case, $Z_1$ ($lu_1$) is the bottleneck thread of $lu$, and when it shares the L2 cache with $X_2$ or $X_3$ ($swim_2$ or $swim_3$), its performance can be improved without causing any performance loss for $swim$ (note that $X_0$ and $X_1$ ($swim_0$ and $swim_1$) are the bottleneck threads for $swim$).

| Map-1  | $<(X_0,X_1,X_2,X_3),(Y_0,Y_1,Z_0,Z_1)>$ |
| Map-2  | $<(X_0,X_1,Y_0,Y_1),(X_2,X_3,Z_0,Z_1)>$ |
| Map-3  | $<(X_0,X_1,Y_0,Z_0),(X_2,X_3,Y_1,Z_1)>$ |

Table 6.6. Three mappings used in the experiments with an 8-core system.

**Experiment-5.** We next varied the number of co-runner applications. Fig-
Figures 6.16(a) and 6.16(b) show the achieved weighted speedups when two (swim and mg, each with 8 threads) and four applications (swim, mg, sp and mgrid, each with four threads) run on a machine with 16 cores and two L2 caches using six different mappings. As can be observed, the overall performance is improved by about 7% on average. One can observe that by adding or removing one application in Mix-3, a lower performance improvement is achieved (compare the speedups presented in these figures with those reported in Figure 6.8(a) for Mix-3). This is because, the magnitude of the performance improvement, as mentioned earlier, depends on the characteristics and the mapping of the bottleneck threads of the co-runner applications and not the number of them.

**Experiment-6.** One of the parameters that affects the performance improvements achieved by CCS is the frequency at which the control decisions are made and enforced. In the previous experiments, this frequency, called the *enforcement interval*, is set to be 200 million cycles. If CCS is invoked more frequently, the behaviors of the running threads are monitored at higher frequency and consequently it may result in achieving better performance improvement. However, increasing
Figure 6.18. The weighted and fair speedup values achieved for enforcement interval sensitivity analysis (Experiment-6). The control intervals are 400 million cycles.

the enforcement frequency means more overhead. Figures 6.17 and 6.18 plot the weighted and fair speedup values achieved by reducing and increasing the length of the enforcement intervals (100 million and 400 million cycles), respectively, while other parameters are similar to Experiment-1. As can be observed, the speedups achieved in Figure 6.17 (the experiment with smaller intervals) are generally higher than those reported in Figure 6.18.

Experiment-7. As explained before, the $\delta$ value (used in Equation 6.2) is the granularity at which the weights associated with the cores are varied in CCS. The default value of this parameter used so far in our experimental evaluation was 2 (as stated in Table 6.2), which corresponds to 10% of the core weights’ initial values. To study the impact of this value on the CCS performance, we performed an experiment with four different $\delta$ values (1, 2, 3 and 4). Figures 6.21(a) and 6.21(b) plot the weighted and fair speedup values achieved for Mix-5 with our six mappings and using different $\delta$ values, while other experimental parameters are the same as Experiment-1. As one can observe, in some cases (e.g., Map-5 and 6), further performance improvements are achieved when the $\delta$ value is increased from 2 to 3. This is because, the core weights can converge to the optimal values faster in these cases through assigning a larger value to $\delta$. However, increasing the $\delta$ value does not always result in better performance improvement (e.g, Map-5, $\delta = 3$ and 4), since the core weights may oscillate around the optimal values and not reach the optimal ones by having a larger $\delta$.

We next report results for our experiments when (i) an L3 cache is used as the last level cache, and (ii) applications have QoS constraints.

Results with L3 Caches. So far, we applied CCS to the last level L2 caches
in different sockets of a multicore machine. Now, we run our workloads on an extended baseline architecture with three levels of caches and eight cores in each socket sharing an L3 cache. In this case, CCS can be employed only in one of the cache levels (L2 or L3) or in both of them. Figure 6.19(a) gives the weighted speedups when CCS is employed in only L3 caches (not in L2 caches). As can be observed from these results, the performance improvement achieved in this case is significantly lower as compared to the case in which CCS is employed in L2 caches, since the number of memory requests received by the L3 caches is lower than the L2 caches. As a result, the co-runner applications are less sensitive to the L3 cache management. Figure 6.19(b) plots the weighted speedup values achieved when CCS is employed in both cache levels (L2 and L3). The performance improvements we achieve in this case are slightly higher than those obtained in Experiment-1, mostly due to the additional improvements coming from the application of CCS to L3.

**Incorporating QoS Constraints.** In a multicore system, co-runner applications may have different *quality-of-service* (QoS) requirements. Our proposed CCS can be used in such scenarios as well. Specifically, to accommodate different QoS requirements of applications, we can tune the rank assignment process in a QoS-aware fashion. In this QoS-aware rank assignment, performance targets of different co-runner applications are compared against the current performance, and the difference between them is used during the rank assignment phase to satisfy all specified QoS targets. In QoS-aware CCS, we assume that each application $A_j$ may have a QoS target $t_j$, which represents the maximum percentage performance degradation allowed with respect to an isolated execution (i.e., when the application is run alone). We can set a reference IPC value for each application based on $t_j$. Algorithm 5 gives the modified version of the Arbiter algorithm that supports QoS guarantees. As can be observed in line 3, the rank of the threads whose current performances are lower than the specified goals are set to $-1$, and as a result, the L2 Managers would select these threads for eviction weight reduction.

We performed an experiment to evaluate this QoS-aware version of CCS. Figure 6.20 plots the normalized IPC values of the threads of different applications in Mix-2 when they are executed on our baseline multicore configuration (Map-5 is used for application mapping). The performance targets are highlighted using
Algorithm 5 QoS-Aware Arbiter

Input: The measured IPCs $IPC[1 \cdots n]$  
Input: Thread mapping $mapping[ ]$  
Input: QoS values $QoS[ ]$  
Output: Ranks of the cores $R[1 \cdots n]$

1: for $i = 1$ to $n$ do  
2: if $(IPC[i] < QoS[mapping[i]])$ then  
3: $R[i] = -1$  
4: else  
5: $app = mapping[i]$  
6: $min = \infty$  
7: for $j = 1$ to $n$ do  
8: if $(IPC[i] < min)$ and $(mapping[j]==mapping[i])$ then  
9: $min = IPC[i]$  
10: end if  
11: end for  
12: $R[i] = \frac{IPC[i]-min}{min}$  
13: end if  
14: end for

solid lines in this figure. As can be seen, all the performance targets are achieved by the QoS-aware CCS, whereas application B’s target is not reached for two of the threads when the QoS support is not activated.

Figure 6.19. The weighted speedup values achieved for the multicore with L3 caches. The capacity of the L3 caches is set to be 1MB per core and L3 access latency is five times greater than L2 access latency.

6.4 Related Work

Contention in the shared resources, such as last level cache in multicores, has become a pressing issue, as the number of cores and co-runner applications increase in the system. Various mechanisms have been proposed to mitigate the impact of this contention on the overall system performance. The schemes proposed in [132, 139] attempt to schedule different tasks efficiently at the OS level to address the potential contention problem.
Figure 6.20. Achieving QoS targets. The solid lines indicate the specified QoS targets for applications.

Figure 6.21. The weighted and fair speedup values achieved with δ sensitivity analysis (Experiment-7) (Mix-5 with six different mappings and four δ values).

Most previously proposed shared cache management approaches in multicores improve performance characteristics of multiple single-threaded workloads. Relatively fewer works in the literature manage shared cache to improve system performance for multithreaded applications. To the best of our knowledge, there is no work that effectively manages shared caches for multiprogrammed, multithreaded workloads to maximize system performance.

Shared cache partitioning [10] has been frequently employed as a mechanism to overcome interference in shared caches among multiple single-threaded applications. Qureshi and Patt [99] proposed a utility-based cache partitioning scheme where the share received by an application was proportional to the utility rather than its demand. Chang and Sohi [6] studied cooperative cache partitioning (CCP), wherein they use multiple time-sharing partitions to resolve cache contention. Rafique et al [24] proposed an OS based scheme that implements a variety of policies including those for fair cache sharing and achieving differentiated instruction throughput. Several other schemes including [31, 133, 33, 140, 19, 8, 7, 141] studied the use of shared cache management for optimizing other metrics like quality of service and fairness to multiple single-threaded applications. Cache partitioning requires extensive hardware support while the prioritization
scheme used in our approach is simple to be implemented. Also, the prioritization scheme is applicable at a much finer granularity than the cache partitioning schemes proposed in the past. Our prioritization scheme can also be viewed as an alternate replacement policy. The key difference between alternate replacement policies [142, 143, 144, 145, 146] proposed in the literature (for managing shared caches) and our prioritization scheme is that our prioritization scheme is able to differentiate between threads of the same application (ranking threads).

Modifying insertion policies in the shared cache has also been explored as a mechanism to induce service differentiation. Qureshi et al [147] also proposed dynamic insertion policies (DIP) that is a flexible replacement policy that dynamically chooses between multiple insertion policies to incur fewer misses. Further, Jaleel et al [147] extended the work on DIP and proposed thread-aware dynamic insertion policies (dip) that takes into account the memory requirements of each application based on the feedback from set-dueling monitors. Xie and Loh proposed promotion/insertion pseudo-partitioning (PIPP) [148] to manage shared caches. PIPP combines dynamic insertion and promotion policies to provide the benefits of cache partitioning, adaptive insertion, and capacity stealing all with a single mechanism.

Muralidhara et al [134] proposed an intra-application cache partitioning strategy that partitions the last-level shared cache among multiple threads of single multithreaded applications. This is the only prior work to our knowledge that specifically deals with cache partitioning for multithreaded applications. However, it is not directly applicable to multiple multithreaded applications as it does not distinguish between threads of different applications. Due to inherent differences in threads of different applications, it would not be possible to obtain fairness among them.

6.5 Conclusions

We introduced a novel way of managing shared caches for obtaining the best performance characteristics from multiprogrammed multithreaded applications. Our proposed scheme (CCS) prioritizes the slowest thread in each multithreaded application and improves performance of each multithreaded application. We also
ensure that overall system performance is improved by managing the eviction priorities of different application threads in the shared cache.

We evaluate CCS on a variety of multiprogrammed multithreaded workloads on a range of systems with 8-16 cores and 2-8 L2 caches with different thread-to-core mappings for each workload. Averaged over six workloads, our shared cache management scheme improves the performance of the combination of applications by 18%. These improvements across applications in each mix are also fair as indicated by average fair speedup improvements of 10% across the threads of each application (averaged over all the workloads).
7.1 Introduction

In recent years, researchers have also investigated process variations in NoCs from different perspectives. In [151], several architectural solutions have been proposed to mitigate the impacts of process variation in the pipeline stages of the routers in an NoC architecture. Even though their proposed router design can reduce the impact of process variations, it will not able to entirely remove them. Several other prior works have studied the modeling of the impacts of process variation in NoCs [158, 159]. For instance, Li et al [160] investigated how process variations affect leakage power in NoCs. However, very few strategies have been proposed to cope with NoC performance degradation due to process variations during run time. Markovsky et al [161] have showed that employing adaptive routing can increase the expected saturation bandwidth in a heterogenous NoC architecture. Employing adaptive routing has the capability to balance the network traffic when the network is highly loaded. However, in their work, they have not considered the performance characteristic of each individual router, and consequently, the packets of a communication may not traverse the best route in terms of the time it takes to reach their destination.

In this chapter, focusing on a heterogenous NoC in which each router has (potentially) a different processing delay, we propose a process variation-aware source
routing scheme to enhance the performance of the communications in the NoC based system. Our scheme assigns a route to each communication of an incoming application, considering the processing latencies of the routers resulting from process variation as well as the communications that already exist in the network to reduce traffic congestion. The packets that belong to each of the communications traverse the assigned path to arrive at their destinations. We developed a simulation framework and evaluated our scheme using applications from the E3S benchmark suite [162]. Our experimental evaluation shows that the proposed scheme can reduce average packet latencies up to 28% as compared to adaptive and dimension-order routings.

7.2 Background and Motivation

7.2.1 Process Variations

The performance and behavior of real systems may differ from the ones targeted in design phase, due to the parameter variations in manufacturing process or during run time [163].

Process variations are categorized into two main types: systematic and random. In systematic variations, the parameters of adjacent on-chip components are strongly correlated, whereas the impacts of random variations caused by different factors such as lithography, etching and duping effects, are randomly distributed across the chip.

Various strategies have been proposed in the literature at different levels of design phase to address the influences of process variations [164, 165]. In the design for manufacturing (DFM) phases (i.e., pre-manufacturing), margins are added to the layout design rules to compensate for the inaccuracy of the manufacturing processes [161]. However, these approaches typically result in larger chip sizes and indeterministic performance impacts. As a result, many recent efforts have focused on post-manufacturing schemes.

One of the main steps in post-manufacturing methods is to model and characterize the potential effects of process variations to enable the effectiveness evaluation of the proposed schemes. In most of the previous works, process variation is modeled as the variation in the channel length and threshold voltage of the
transistors ($L_{eff}$ and $V_{th}$) across a chip. The values of these parameters ($L_{eff}$ and $V_{th}$) directly affect the speed and the power consumed by transistors. For instance, in a pipeline system, the latency of the critical path may change and leads to a variation in the working frequency.

### 7.2.2 Process Variation in NoCs

Similar to other pipeline based architectures, the working frequency and latency of an NoC router is determined by the slowest stage and its critical path delay. Process variation may vary the critical path length and consequently, affect the latency of the router. As stated in [161], although the impact of the variations on the performance of different components of an NoC router is very complex, they can be modeled as the router latency or throughput.

As stated above, from a high level perspective, process variations may convert an otherwise homogenous NoC architecture into a heterogenous one, which consists of routers with different latencies and throughput. In [161], to model an NoC architecture with process variation impacts, it is assumed that the values for the throughput of the routers across the NoC exhibits a normal distribution. In this chapter, we also use a normal distribution to assign latency values to each router across our NoC architecture. Note that, in systematic process variation, there is a spacial correlation among the assigned values, which needs to be taken into consideration. In our work, we only consider random variations, since our routing scheme works in the same way for the case of systematic variations. Markovsky et al [161] have shown that employing adaptive routing can compensate for performance heterogeneity in NoC architectures, since the heterogenous system abstracts the underlying parametric variation as traffic congestion. Although the network saturates with more traffic by employing adaptive routing, adaptive routing is not the best choice in terms of performance in a network with low traffic.

### 7.3 P-V Aware Source Routing

A parallel application can be represented as a communication task graph (CTG). In a CTG $G = (V, E)$, $V$ corresponds to the set of the graph nodes representing tasks of the application. The communications among different tasks are represented by
the set of the graph edges $E$.

In our NoC based multicore, one of the cores plays the role of the master core running the operating system, and is responsible for mapping new incoming tasks and applications to different processing cores, based on the applications’ CTGs and the implemented mapping algorithm. During the course of execution, active cores (cores that have been assigned tasks) communicate with each other via the NoC using the employed routing algorithm.

In our work, we assume that after a new application is mapped to the physical cores, a path is assigned by our routing algorithm to each communication in the application’s CTG. The packets sent by the source node, traverse this path in the network to reach its destination (this type of routing is called source routing, since this path is determined at the source point). Our routing algorithm computes the best path for each communication between a sender and a receiver based on the inherent speed of the routers (dictated by process variations) and the current traffic pattern. Note that, we assume the OS has the knowledge on the latency of each router from the post-implementation on-chip tests [161].

### 7.3.1 Our Source Routing Algorithm

In this section, we first present our process variation-aware routing scheme and then we discuss its implementation details.

As stated earlier, our scheme is a type of source routing algorithm, i.e, the path traversed by the packets of each communication is specified by the OS after the corresponding application is mapped. Figure 7.1 gives the high-level pseudocode of our proposed algorithm. In this algorithm, $G$ is a graph representing the mapped CTG of an incoming application. To find and assign the best path for each communication, the new communications (that may belong to one or more applications) are sorted according to a parameter called *flexibility* (line 2). Lower the flexibility a communication has, more constrained it is; consequently, we need to start with more constrained communications. Our algorithm starts with the worst flexible communication. In the next section, we discuss how the value of this metric is obtained for each communication. At the next step, the best path, in terms of router latency and traffic contention, from the source to destination is computed (line 5) and the traffic pattern of the network is updated based on the
obtained path (line 8). Finally, the path is assigned (line 9). Note that, we have to ensure that this new assigned path does not create a deadlock in the network with the previously assigned paths. We later discuss the deadlock-freeness of our scheme in Section 7.3.4.

**PATH_ASSIGNMENT**(*G* : MappedTaskGraph)

1. Global pattern
2. List *comm* *sort* = sort *communications*(*G*)
3. for each *comm* *i* in *comm* *sort*
   - *path* *i* = FindBestPath(*G*, *comm* *i*.source, *comm* *i*.dest)
   - if !DeadlockExists(*path* *i*, pattern)
     - UpdateTrafficPattern(*path* *i*, *comm* *i*.rate)
   - AssignPath(*path* *i*, *comm* *i*.source, *comm* *i*.dest)

Figure 7.1. The path assignment algorithm.

Figure 7.2 gives our algorithm that finds the best path for each communication. This algorithm is based on the Dijkstra’s shortest path algorithm and differs from that algorithm in two ways. First, our scheme is a minimum hop routing, i.e, each computed path contains minimum number of hops between the source and destination (lines 11, 12 and 13 satisfy this feature).

**FIND_BEST_PATH**(*G* : MappedTaskGraph, *S* : Source, *D* : Dest)

1. Global pattern
2. for each *comm* *i* in *comm* *sort*
   - *d*[*v*] = ∞; *prev*[*v*] = NULL;
   - *d*[*S*] = 0; *N* = set of all nodes in *G*;
3. while *N* ≠ ∅
   - choose node *w* ∈ *N* such that *d*[*w*] is minimal;
   - remove *w* from *N*;
   - for each neighbor *v* of *w*
     - *z* ← *d*[*w*] + Cost(*w*, *v*, *G*(*w*, *v*));
     - *md* 1 = ManhattanDistance(*v*, *D*)
     - *md* 2 = ManhattanDistance(*w*, *D*)
     - if *z* < *d*[*v*] and *md* 1 < *md* 2
       - *d*[*v*] = *z*; *prev*[*v*] = *w*;
4. return ExtractPath(*D*);

Figure 7.2. Finding the best path based on the Dijkstra’s shortest path algorithm.
Cost($w : Node, v : Node, comm : Communication$)
1  Global pattern
2  return $router[w] . delay + Traffic\_Cost(pattern, comm\_rate)$

**Figure 7.3.** The cost function based on the router delays and the network traffic pattern.

The second difference is the cost of the edge between two adjacent nodes (from node $w$ to node $v$) considered in the algorithm. This cost, as given in Figure 7.3, comprises two parts: 1. Router delays that capture the process variation impacts, and 2. The cost the comes from the communications that have already had this edge in their paths. After a path is assigned to a communication, pattern is updated and the path specification, including its traversed edges and injection rate, is saved. Information provided by pattern is used to estimate the traffic cost of the network edges. For instance, if several communications with different injection rates use edge $w - v$, the traffic delay overhead of this edge on communication $i$ is estimated using Equation 7.1.

$$Traffic\_Cost_{wv}(comm_i) = (1 - \frac{inj_i}{\sum inj_j}) \times router[w].delay,$$

(7.1)

where $inj_i$ is the injection rate of communication $i$. Note that, as the communication injection rate decreases, the delay overhead for that communication increases. This estimated value is added to the router delays as shown in Figure 7.3. In the following sections, we detail the routing process performed in each intermediate router and also discuss how deadlock-freeness of our scheme is guaranteed.

### 7.3.2 Flexibility Metric

As stated earlier, to find and assign routes to the communications represented by a CTG, our scheme starts with the communications that are less flexible. We define flexibility as the number of distinct paths from the source to the destination in our two-dimensional mesh-based NoC with minimum number of hops (i.e., the route length is equal to the Manhattan Distance between the source and destination). For instance, if there is only one path (with minimum number of hops) between the source and destination cores (i.e., they are located on the same row or column in our mesh-based NoC), this communication is selected for the path assignment process earlier. As mentioned earlier, our scheme starts with the communications
that are least flexible. In our experimental evaluations, we show how well this approach can improve the performance of our routing scheme.

In a two-dimensional mesh-based NoC, each core can be represented by a coordinate pair \((x, y)\). In our experiments, we assume that the origin point \((0, 0)\) is the bottom leftmost node and the \(x\) axes is onto the right and the \(y\) axes locates on top. The flexibility value for a communication between two cores \(u = (x_1, y_1)\) and \(v = (x_2, y_2)\) can be computed using the equation shown below:

\[
flexibility(u, v) = \frac{(|x_2 - x_1| + |y_2 - y_1|)!}{|x_2 - x_1|! \times |y_2 - y_1|!}
\]

(7.2)

7.3.3 Routing

A packet reaches its destination when the address of the current router is the same as the packet destination address. If it is not, the packet is forwarded to one of the adjacent nodes based on the route assigned by our path assignment scheme. In addition to the source and destination addresses, this route is also encoded and embedded in the header of the packets.

Figure 7.4 shows the format of a packet in a 2D \(5 \times 5\) mesh based NoC. As can be seen, two parts are added to the packet header: 1. \(cnt\): a counter increased by one at each hop. 2: \(paths\): a bit string indexed by the \(cnt\). At each hop, the bit in \(path\) that is indexed by the \(cnt\) determines the output port that the packet is forwarded to. Since all the paths determined by our routing algorithm contains the minimum number of hops, at most there are two possible output ports at each hop that the packet can be forwarded. As a result, one bit is sufficient for direction selection.

![Figure 7.4. The packet format in a 5 \times 5\ mesh based NoC.](image)

Table 7.1 gives the routing decision at each hop based on \(paths\), the current address and the packet destination address. Note that, in Table 7.1, \(x_{offset} = x_{destination} - x_{current}\) and \(y_{offset} = y_{destination} - y_{current}\).
7.3.4 Handling Deadlocks

Once a path is determined for a new communication, there is a chance that this new route creates a deadlock with other communications already exist in the network. Therefore, we need a mechanism to detect and avoid these potential deadlocks. Dally and Seits [166] have shown how deadlocks can be detected using channel dependency graphs. A channel dependency graph (CDG) is a directed graph, each vertex of which represents a channel in the network. A channel is a physical link and the corresponding Virtual Channel (VC). There is an edge from vertex $u$ to $v$ in this graph, if a communication uses channel $v$ immediately after channel $u$. In our scheme, each link is presented by a vertex in the (CDG), since it is assumed that all the virtual channels are used along the path in each communication. Therefore, the corresponding CDG of a $n \times n$ mesh based NoC contains $2 \times n \times (n-1)$ vertices.

A deadlock exists in the network if a cycle can be found in the CDG. To avoid it, the cycle dependency needs to be broken. The mechanism presented in [167] that breaks cycles by adding new channels limits the number of virtual channels that can be used by each communication.

After finding a path and before assigning it, the corresponding CDG is updated and checked for any cycles. If a cycle is detected, the procedure to find the best route is invoked again but not considering the router with the highest delay in the route. Consequently, the new discovered path is different from the previous one that may lead to breaking the cycle. If the new route does not solve the problem, another node is selected and removed in the path assignment algorithm. If no deadlock-free paths found (a very rare situation), the cycle is broken by limiting the virtual channels that two overlapping communications in the cycle may use.

<table>
<thead>
<tr>
<th>$x_{offset} \geq 0$</th>
<th>$y_{offset} \geq 0$</th>
<th>$x_{offset} \geq 0$</th>
<th>$y_{offset} \leq 0$</th>
<th>$x_{offset} \leq 0$</th>
<th>$y_{offset} \leq 0$</th>
<th>$x_{offset} \leq 0$</th>
<th>$y_{offset} \geq 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>North</td>
<td>0</td>
<td>South</td>
<td>0</td>
<td>South</td>
<td>0</td>
<td>North</td>
</tr>
<tr>
<td>1</td>
<td>East</td>
<td>1</td>
<td>East</td>
<td>1</td>
<td>West</td>
<td>1</td>
<td>West</td>
</tr>
</tbody>
</table>

Table 7.1. The routing Decisions.
7.4 Experimental Evaluation

7.4.1 Setup

Figure 7.5 illustrates the high-level view of our simulation framework. This framework comprises five main modules. In the following discussion we explain briefly the functionality of each component: (i) CTG Mapping: This module takes the new parallel application represented by a CTG as an input and maps it to the nodes of our NoC based multicore. The output of this module shows the communications between different physical cores in our mesh-based NoC. (ii) P-V Delay Mapping Generator: This module generates a delay map based on a normal distribution, with a mean value of $\sigma$ and a variance of $\mu$ for the routers in our NoC to simulate the process variation effects. (iii) Path Assignment: Our proposed routing scheme is implemented in this module. (VI) Traffic Generator: The packets injected to the network are generated by the traffic generator module. This module needs the CTG and the application mapping to generate the network traffic for various communications. (V) NoC Simulator: The core of our simulation framework is the NoC simulator that we employed to evaluate our routing scheme. This simulator uses the paths produced by our scheme to route the packets of different communications. Table 7.2 gives the system configuration parameters of the simulator.

<table>
<thead>
<tr>
<th>NoC Topology</th>
<th>2D mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoC Dimension</td>
<td>5 x 5</td>
</tr>
<tr>
<td>Switching Technique</td>
<td>wormhole</td>
</tr>
<tr>
<td>Packet Size</td>
<td>5 flit</td>
</tr>
<tr>
<td>Buffer Size</td>
<td>2 flit</td>
</tr>
<tr>
<td>Flit Size</td>
<td>128 bit</td>
</tr>
</tbody>
</table>

Table 7.2. System configuration parameters.
7.4.2 Results

7.4.2.1 Results with Different Mappings

Figure 7.6 gives the CTGs of our four applications from the E3S benchmark suite [162] (auto-indust-mocsyn). We assume that these applications are mapped and start their executions at a same time. Therefore, our scheme considers all of the communications in these graphs for the path assignment process.

We performed a Monte Carlo analysis to evaluate our proposed algorithm. In this analysis, at each experiment instance, a random router latency mapping and a random application mapping are generated and used during the simulation. As stated earlier, based on these mappings, our routing scheme assigned routes to different communications.

Our metric for the performance comparison of different routing schemes is the average latency of the packets sent from the sources until they arrived at their destinations. We measure the value of this metric as the rates of the packets injected into the network increase. The injection rate of a communication depends on the frequency of the processing cores and the bandwidth requirement of the running application. Equation 7.3 shows how the injection rate can be computed for a communication.

\[
\text{inj. rate}_i = \frac{BW \times 1}{\text{packet.size}},
\]  

(7.3)

where \(\text{inj. rate}_i\) is the injection rate of communication \(i\) (packet / cycle), \(BW\) represents the bandwidth requirement of that communication (MB/sec) and \(\text{packet.size}\)
Figure 7.7. The average packet latency vs. injection rate for 8 different application mappings and router delay mappings.

is the size of packets (Byte). For instance, if the injection rate is 0.01 packet/cycle, the working frequency is 400MHz and the packet size is 90B, the bandwidth requirement can be computed as 360 MB/sec.

We tested our scheme for 50 different mappings (application and latency mappings), and compared our scheme with one deterministic routing algorithm (X-Y routing [168]) and one adaptive algorithm (Duato’s algorithm [168]) in terms of packet latency and network saturation point. Figure 7.7 plots the average packet latencies when three routing schemes are employed for eight different mappings, as the communication injection rate increases. As one can observe, the impact of the routing schemes varies across different mappings. In Figures 7.7 (a), (d), (f) and (h), employing our scheme reduces the average packet latencies and also extends the network saturation point. However, our scheme does not always perform better than the adaptive routing in extending the network saturation point. Although our scheme reduces the packet latencies when the saturation points not reached yet in Figures 7.7 (b), (e), (g), the adaptive routing works better for the network with higher traffic due to the ability to balance the network load. In such cases, the adaptive routing can be employed if the network is under high traffic pressure. Note that, our scheme improves the performance of the communications in all the mappings before the saturation points are reached.
Figures 7.8 and 7.9 plot the average packet latency improvement for these different routing schemes, when the packets are injected to the network with different rates. These charts show the averages and the variances of the improvements achieved for 50 different mappings. As can be seen, our scheme can achieve the average improvement up to 28% as compared with the two other routing mechanisms.

The network saturation point is another metric that we considered for performance comparison of different schemes. The network saturation point is the lowest packet injection rate in which the packet latencies start to converge to unlimited bounds. In other words, the saturation point shows how much traffic the network can endure. Figure 7.10 plots the average saturation points when different routing schemes are employed. Our scheme increased the saturation point by 10% and 30% compared to X-Y and adaptive routings.

7.4.2.2 Impact of Communication Volume

As the number of communications increases, our routing scheme offers more performance gains as compared to the other routing algorithms. As an example, assuming that mapping 1 and mapping 2 are two different application mappings (illustrated in Figure 7.11) for the applications in Figure 7.6. Figure 7.13 plots the average packet latencies as the three routing schemes employed when the packet injection rates are fixed at 0.028 packet/cycle for three cases: 1. One instance of the applications is mapped and executed using mapping 1; 2. One instance of the applications is mapped and executed using mapping 2; and 3. Two instances of
Figure 7.11. Router delay and task graph mappings ((a,b) indicates that node a and node b from task graphs shown in Figure 7.6 have been mapped to this core).

Figure 7.12. Application mappings. Each number corresponds to a task in Figure 7.6.

7.4.2.3 Path Reassignment

As stated earlier, our proposed path assignment algorithm is invoked when an incoming application is about to start its execution. Suppose that, at time $t_0$, two applications (App1 and App2) start their executions. The path assignment algorithm is invoked to assign routes to the applications’ communications. At time $t_1$ a new application (App3) is about to start its execution. At this time, we have two options as far as route assignment is concerned. One option is to consider only
the communications of App3 in the path assignment algorithm and the other option is to consider all running applications. In the latter case, the pre-assigned routes may change, since the algorithm is executed for all communications. Further, once one or more applications terminate, the assigned routes can be recomputed and potentially modified.

For example, assume that the four applications shown in Figure 7.6 start their executions at time $t_0$. At time $t_1$ App1, 2 and 4 terminate. In this case, the communications in App3 can be recomputed. We tested this scenario for a network with a router delay mapping shown in Figure 7.11 and the application mapping given in Figure 7.12(a). Figure 7.14(a) illustrates the routes for the communications of App3 when other applications running, and Figure 7.14(b) shows the new paths after the termination of the other applications. As can be observed, one of the routes has been changed by the path reassignment process.

Figure 7.14. App2 communication routes.
7.4.2.4 Impact of the Flexibility Metric

As we discussed before, to assign routes to the communications of an incoming application, the communications first are sorted based on their flexibility values. To quantify the impact of this parameter on the efficiency of our scheme, we conducted an experiment. Figure 7.12(b) and Figure 7.11(a) show the application mappings for the graphs in Figure 7.6 and routers’ delay mapping we considered in this experiment, respectively. Figure 7.15 plots the average packet latencies as the injection rates increase, for these two cases: when the communications are selected to be assigned based on their flexibility values and when they are chosen randomly. As one can observe, considering the flexibility metric helps the network to handle more traffic before the saturation point.

7.5 Conclusion

We proposed a process variation-aware routing scheme to enhance the speed of the communications in an NoC based architecture. In our proposed scheme, we consider two significant factors: the performance variations of the routers and also the network traffic pattern in order to reduce the potential traffic congestions. It is important to note that our routing scheme is a topology independent scheme and can be employed for NoCs with different topologies. Our experimental evaluation reveals that the proposed scheme can reduce average packet latencies up to 28% as compared to adaptive and dimension-order routings.
Conclusions

8.1 Summary of Contributions

In this research, we focus on proposing novel schemes to dynamically manage various available shared resources in emerging multicore systems, while targeting three main goals: (1) Maximizing the overall system performance, (2) Meeting end-to-end QoS targets defined by the system administrator and (3) Optimizing power and energy consumption. We consider a wide range of available resources including cores, shared caches, off-chip memory bandwidth, on-chip communication resources and power budgets. Further, in my research towards achieving our goals, we employ formal control theory as a powerful tool to provide the high-level performance targets through dynamically managing and partitioning the shared resources.

Figure 8.1 shows the high level view of our main goals and the available resources that we consider and target in each chapter of my dissertation. A summary of the main contribution of each chapter is as follows:

**Chapter-2.** In this chapter, we propose a control theory centric scheme, called METE, to partition multiple shared resources in a multicore machine among concurrently executing applications over the course of execution. In the current implementation of METE, we consider three types of shared resources: processing cores, shared cache space, and off-chip memory bandwidth. Assuming that each running application has a performance target to be satisfied, our main goal is to provide applications with sufficient resources to achieve the specified targets.

**Chapter-3.** We propose two network prioritization schemes that reduce the end-
to-end memory access latency in multicores in this chapter. Our first scheme addresses the latency variance in memory accesses that belong to the same application, and expedites memory response messages that experience high latencies by prioritizing them. This reduces the number of off-chip requests with high latencies and achieves a more uniform memory latency pattern. Our second scheme improves memory performance by prioritizing memory request messages that are destined for idle banks over other requests. This optimization increases bank level parallelism and improves memory utilization.

**Chapter-4.** The main contribution of the work in this chapter is a two-level power distribution strategy oriented towards maximizing system performance under a specified power budget. The results collected from our experiments with 6 workloads formed using 16 applications selected from a set of 30 applications clearly demonstrate the importance of power distribution at both the levels.

**Chapter-5.** In this chapter, we propose an agile DVFS technique for dynamically modulating the frequency of memory components based on the sensitivity of running applications. The proposed HiPEMM technique utilizes the DMPKC parameter for monitoring the memory sensitivity of applications for dynamically partitioning the applications to different groups, and modulating the frequency of each group for optimizing energy conservation at the cost of minimal performance penalty.
Chapter-6. We introduce a novel way of managing shared caches for obtaining the best performance characteristics from multiprogrammed multithreaded applications in this chapter. Our proposed scheme (CCS) prioritizes the slowest thread in each multithreaded application and improves performance of each multithreaded application. We also ensure that overall system performance is improved by managing the eviction priorities of different application threads in the shared cache.

Chapter-7. In this chapter, we propose a process variation-aware routing scheme to enhance the speed of the communications in an NoC based architecture. In our proposed scheme, we consider two significant factors: the performance variations of the routers and also the network traffic pattern in order to reduce the potential traffic congestions. It is important to note that our routing scheme is a topology independent scheme and can be employed for NoCs with different topologies.
Bibliography


[104] Chen, M., X. Wang, and X. Li (2011) “Coordinating processor and main memory for efficient server power control,” in SC.


Vita
Akbar Sharifi

Akbar Sharifi a Ph.D. candidate in the Department of Computer Science and Engineering at Pennsylvania State University. Prior to joining PSU in Fall 2008, he received a Masters of Science in Computer Engineering from Sharif University of Technology in 2008 and a B.S in Computer Engineering from University of Tehran in 2005.