Globally Synchronized Frames for Guaranteed Quality-of-Service in Shared Memory Systems

by

Jae Wook Lee

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author . . . . . .
Department of Electrical Engineering and Computer Science
September 4, 2009

Certified by . . . . . .
Krste Asanović
Associate Professor
Thesis Supervisor

Certified by . . . . . .
Arvind
Professor
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Globally Synchronized Frames for

Guaranteed Quality-of-Service in Shared Memory Systems

by

Jae Wook Lee

Submitted to the Department of Electrical Engineering and Computer Science
on September 4, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Resource contention among concurrent threads on multicore platforms results in greater performance variability of individual threads than traditionally seen with time-multiplexed threads on single-core platforms. This performance variability makes it hard to provide performance guarantees, degrades parallel program performance, and burdens software writers by making performance tuning and load balancing more challenging.

In this thesis, we propose a novel QoS framework, called *Globally Synchronized Frames (GSF)*, to combat the performance variability problem in shared memory systems. We first apply GSF to a multi-hop on-chip network to provide QoS guarantees of minimum bandwidth and maximum delay for each flow sharing the network. The GSF framework can be easily integrated with a conventional virtual channel (VC) router without significantly increasing the hardware complexity. We then present an extended version of GSF framework to provide end-to-end QoS for cache-coherent shared memory systems, which is called GSF memory system (GSFM). GSFM employs a single unified framework to manage multiple heterogeneous bandwidth resources such as on-chip networks, DRAM banks and DRAM channels, to achieve better hardware efficiency and composability towards end-to-end QoS than component-wise QoS approaches.

Finally, we propose the METERG (MEasurement Time Enforcement and Runtime Guarantee) QoS framework. Independent of GSF, the METERG framework provides an easy method to obtain a tight estimate of the upper bound of a program's execution time for a given resource reservation setting. Our approach is based on simple measurement without involving any expensive program analysis or hardware modeling.

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Thesis Supervisor: Arvind
Title: Professor

Acknowledgments

First and foremost, I give thanks to God, my Lord and Savior, for giving me an opportunity to study at MIT and the strength to finish a PhD. In retrospect, my road to a PhD was longer and bumpier than I had imagined, but God used it for His good purpose. After all, my cup overflows with His grace—I am more optimistic, more content, and more grateful in Him than ever before.

I was fortunate to have great mentors around me. Without question, my highest gratitude goes to Prof. Krste Asanović, my thesis supervisor. His keen insight, breath and depth of knowledge, enthusiasm, and efficiency are simply a subject of awe to me at all times. He is accessible and always gives insightful answers to my questions, however inarticulate they may be; there were numerous occasions when I finally understood the deep insight of his answers long after our discussion. He is the best teacher of English writing; his magic hand always turns a poor sentence of mine into a clear one with fewer words. Krste, it was a great privilege to work with you, and thank you for being such a patient and supportive advisor.

I thank Prof. Arvind, my co-supervisor, for accepting me into his group when Krste left for Berkeley and generously supporting me academically, financially, and socially. Without the help of Arvind and Gita, my life at MIT would have been much more difficult. His 60th birthday celebration gave me great inspiration for what I hope to be at his stage of life.

I thank Prof. Srinivas Devadas, my first supervisor, for his academic and financial support for my first three years at MIT and for serving as a thesis committee member. It was great fun to build the first PUF chip under his guidance, and I am proud of the results of our joint work. His door has always been open to me even after I joined Krste's group.

I thank Prof. Victor Zue for his superb guidance as my graduate counselor, and Prof. Anant Agarwal and Prof. Saman Amarasinghe for inviting me to spend a memorable summer in the RAW group. Also, I enjoyed serving as a teaching assistant for Dr. Joel Emer during his first term at MIT. Finally, Prof. Deog-Kyoon Jeong and Prof. Kiyoung Choi at Seoul National University (SNU) first introduced me into the exciting world of computer architecture and VLSI design by giving excellent lectures, providing opportunities

for undergraduate research, and writing recommendation letters, for which I am grateful.

Although it bears my name, this thesis is not solely mine in that many friends and colleagues generously lent their precious time and energy for it. Alfred Man Cheuk Ng, Sarah Bird, Christopher Batten, Charles O'Donnell, Myron King, Nirav Dave, Muralidaran Vijayaraghavan, Abhinav Agarwal, Kermin Elliott Fleming, Asif Khan, Seongmoo Heo, Heidi Pan, Rose Liu, and Ken Barr read various parts of this thesis and gave me invaluable feedback, which I appreciate. Alfred, my "research oracle" and officemate, deserves a special mention. Conversation with him always allows me to understand a problem more deeply, come up with a solution, or both. Sarah Bird at Berkeley helped develop the idea of a GSF memory system. Charlie did many last-minute proofreads for me without complaint. Outside MIT, early feedback on GSF from Kyle Nesbit (Google) and John Kim (KAIST) helped crystallize the main idea of this thesis.

The excellence of MIT education does not solely come from great mentors; I have also benefitted greatly from interactions with my fellow graduate students. Special thanks to the brilliant members of the SCALE Group at MIT, including, Christopher Batten, Heidi Pan, Rose Liu, Seongmoo Heo, Michael Zhang, Jessica Tseng, Albert Ma, Ken Barr, Ronny Krashinsky, and Mark Hampton. For the last several years, Arvind's Group has been my happy home at MIT, and my sincere thanks go to its members, including Alfred Man Cheuk Ng, Nirav Dave, Murali Vijayaraghavan, Abhinav Agarwal, Myron King, Asif Khan, Kermin Elliott Fleming, Ryan Newton, Michael Pellauer, Yuan Tang and Michael Katelman. My graduate school experience is made richer by interacting with other members of the Computer Architecture Group (CAG) and the Computation Structures Group (CSG), including Gookwon Edward Suh, Daihyun Lim, Daniel Rosenband, Michael Taylor, David Wentzlaff, Prabhat Jain, Charles O'Donnell, Jason Kim, Myong Hyon Cho, Keun Sup Shim, and Mieszko Lis.

The First Korean Church in Cambridge (FKCC) was a true blessing to me, where I gained the strength to carry on through fellowship with God and other people. Special thanks to Pastor and Mrs. Taewhan Kim, Mr. and Mrs. Wanhae Kim, Choonkoo Lee and Soonhyo Shin, Mr. and Mrs. Jae Y. Song, Mr. and Mrs. Soochan Bae, Mr. and Mrs. Soohan Kim, Mr. and Mrs. Deukhan Son, Mr. and Mrs. Jaejoung Kang, Mr. and Mrs.

Changsik Song, Mr. and Mrs. Sunho Park, Mr. and Mrs. Mooje Sung, Terrence C. Kim, Sunghoon Kang, Sangmoo Kim, Bo-Eun Ahn, Prof. and Mrs. Sunghun Kim. Also, I thank Rev. Yongha Choi of Boston Milahl for his spiritual guidance and prayer.

The Korean community at MIT EECS helped me overcome difficult times, and we shared ups and downs at graduate school. Thanks to Prof. and Mrs. Jae S. Lim, Soonmin Bae, Hyunsung Chang, Seonghwan Cho, Taeg Sang Cho, Eunjong Hong, Sungho Jo, Jungwoo Joh, Jaeyeon Jung, Adlar Jiwook Kim, Byungsub Kim, Jungwon Kim, Minkyu Kim, Jung Hoon Lee, and Min Park.

I was fortunate to spend two exciting semesters in 2008 at the Parallel Computing Laboratory (Par Lab) at U. C. Berkeley, and am thankful for the hospitality of the Berkeley community. Thanks to the Par Lab members, including, Sarah Bird, Yunsup Lee, Changseo Park, and Youngmin Yi. Outside the Par Lab, I am thankful for the friendship with Dr. and Mrs. Sang Yeol Kim, Dr. and Mrs. Kwangwook Kim, Dr. and Mrs. Hwayong Oh, Jeongjoo Lee, and Kitae Nam.

Thanks to our friendly and caring administrative staff, including Sally Lee, Mary McDavitt, and the late Cornelia Colyer. Soohan Kim, Gilbert D. Nessim, and Srikanti Rupa Avasarala were my "thesis buddies" who were immensely helpful to me in finishing this thesis in time. I thank Julio, my janitor, not only for cleaning up my office but also for tossing random snacks and drinks during many nights when little progress was made.

I thank the late Dr. Yeshik Shin (1969-2004) who was one of my best supporters. I still use her wireless mouse and mug. Dr. Sangoh Jeong is always warm, supportive, and funny, allowing me to blow away my MIT blues. Further, I enjoyed random conversations about computer science with Jin-Young Kim, one of just a few graduates of Yangjae High, my alma mater, in the Boston area.

I am grateful for generous funding support from the Korea Foundation for Advanced Studies (KFAS), Nokia-MIT Collaboration, and the International Computer Science Institute (ICSI). I spent two summers as an intern at Nokia Research Center Cambridge (NRCC), and thank my collaborators including Jamey Hicks, Gopal Raghavan, and John Ankcorn.

This thesis would not have been possible without my family members' prayer, support,

and encouragement. My parents, Kunho Lee and Sunsook Kim, have encouraged me to pursue a PhD since I was a little kid (by calling me “Dr. Lee”), and prayed for me every day. They were confident in me even at times when I lost confidence in myself. I’d like to express my utmost thanks to you, Mom and Dad, for your unchanging love and support. I also thank my sisters and their families, Min-Ah Lee and Sungjin Yoon, Yoona Lee and Junkyu Ahn, for being my lifelong friends and supporters. My two late grandmoms, Ms. Junghee Sohn (1921-2006) and Ms. Jungsook Lee (1921-2004), would be the happiest of anyone for this thesis, and I miss their unconditional love. I am grateful for Aunt Younghee Kim and Uncle Kyu Haeng Lee’s warm support throughout my stay in the United States. I thank my parents-in-law, Chan-Il Lee and Young Zu Kim, for their words of wisdom and heartful encouragement.

Finally, my lovely wife, Jung-Eun, I would not have made it this far without your support, sacrifice, and love. You are the best present I have ever received from God. It was your courage and optimism that kept me going at numerous times when I felt hopeless. Thank you for being the best wife I could ever imagine!

Contents

1 Introduction **23**

1.1 Performance Variability in Multicore Platforms 24

1.2 The Case for Hardware Support for QoS 26

1.3 A Taxonomy of QoS 28

1.4 Design Goals for QoS Mechanisms 32

1.5 Contributions 33

1.6 Thesis Organization and Previous Publications 34

2 Background **37**

2.1 CMP Memory Hierarchy Designs 38

2.2 Fairness Problems in Best-Effort Memory Systems 40

2.2.1 Fairness Problems in On-Chip Networks 41

2.2.2 Fairness Problems in Memory Controllers 42

2.2.3 Fairness Problems in Other Shared Resources 46

2.3 Priority-Based versus Frame-Based Scheduling 47

2.3.1 Priority-Based Scheduling Algorithms 48

2.3.2 Frame-Based Scheduling Algorithms 50

2.4 Component-Wise versus End-to-End QoS 51

2.5 A Survey of Proposals for Network QoS 54

2.6 Arguments for Frame-Based, End-to-End Approaches 57

3 GSF On-Chip Networks: One-Dimensional GSF **61**

3.1 Introduction 62

3.2 Globally-Synchronized Frames (GSF) . . . 63
3.2.1 Global Deadline-Based Arbitration for Bandwidth Guarantees . . . 63
3.2.2 Baseline GSF . . . 65
3.2.3 An Alternative Derivation of Baseline GSF . . . 70
3.2.4 Carpool Lane Sharing: Improving Buffer Utilization . . . 72
3.2.5 Early Frame Reclamation: Increasing Frame Reclamation Rate . . . 73
3.3 Implementation . . . 73
3.3.1 GSF Router Architecture . . . 74
3.3.2 Global Synchronization Network . . . 76
3.4 System-Level Design Issues . . . 77
3.5 Evaluation . . . 78
3.5.1 Simulation Setup . . . 78
3.5.2 Fair and Differentiated Services . . . 79
3.5.3 Cost of Guaranteed QoS and Tradeoffs in Parameter Choice . . . 79
3.6 Summary . . . 83

4 GSF Memory System: Multi-Dimensional GSF 85
4.1 GSF Memory System: an Overview . . . 85
4.2 GSFM Operations . . . 88
4.3 Target Architecture . . . 90
4.4 OS Design Issues . . . 91
4.5 Injection Control and Token Management . . . 93
4.6 Determining the Service Capacity of a Resource . . . 95
4.7 Selecting a Frame Interval . . . 100
4.8 Memory Controller and On-Chip Router Designs . . . 100
4.9 Evaluation . . . 102
4.9.1 Simulation Setup . . . 102
4.9.2 Microbenchmarks . . . 104
4.9.3 PARSEC Benchmarks . . . 108
4.10 Related Work . . . 110

4.10.1 System-wide QoS frameworks and multicore OS 110
4.10.2 QoS-Capable Memory Controllers 111
4.10.3 QoS-Capable Caches and Microarchitectural Components 111
4.11 Summary 111

5 Bridging the Gap between QoS Objectives and Parameter Settings 113
5.1 Introduction 113
5.2 Related Work 115
5.3 The METERG QoS System 117
5.3.1 Overview 117
5.3.2 Safety-Tightness Tradeoff: Relaxed and Strict Enforcement Modes 120
5.3.3 METERG QoS Memory Controller: An Example 121
5.4 Evaluation 123
5.4.1 Simulation Setup 124
5.4.2 Simulation Results 125
5.5 Summary 128

6 Conclusion 129
6.1 Summary of Contributions 130
6.2 Future Work 132

List of Figures

1-1 Utility curves of elastic best-effort and hard real-time applications (taken from [83]). 27

1-2 Characterization of a QoS system with two parameters: the number of static priority levels (P=4 in this example) and bandwidth allocation policy for each priority level. Different priority levels can enforce different bandwidth allocation policies as shown by gray boxes. Better QoS is provided as a flow's policy is placed more up (higher priority) and to the right (more sophisticated QoS bandwidth allocation) in the grid. 29

1-3 Comparison of two QoS systems implementing Expedited Forwarding (EF). The number of static priority levels is two (P=2). 31

2-1 Private versus shared L2 cache designs in a 4-core CMP (modified from a figure in [100]). The private L2 design in (a) treats each L2 slice as a private cache and its last-level private cache (LLPC) is L2, which is also the last-level cache (LLC) on the chip. The shared L2 design in (b) treats all L2 slices as part of a global shared cache, where LLPC (L1) is different from LLC (L2). This thesis addresses QoS problems in bandwidth usage at the resources further out than the last-level private caches (LLPC). 39

2-2 Last-level cache (LLC) miss service path with multi-hop on-chip networks. This thesis focuses on three heterogeneous bandwidth resources on the path: (on-chip) network links, DRAM channels and DRAM banks. 41

2-3 Fairness problem in a simple three-node network. The throughput of a flow decreases exponentially as the distance to the hotspot increases. 42

2-4 Fairness problem in an 8×8 mesh network as in (a). All nodes generate traffic toward a hotspot shared resource located at (8,8) (indicated by arrow) with injection rate of 0.05 (flits/cycle/node), and the bar graph shows accepted service rate per node by the hotspot resource. In (b), locally-fair round-robin (RR) scheduling leads to globally-unfair bandwidth usage, penalizing remote nodes. The minimal-adaptive routing in (c) eliminates the imbalance of traffic between x and y directions in (b), and possibly helps mitigate the problem in a lightly-congested network, but does not fundamentally resolve the problem. 43

2-5 A high-level organization of a modern DRAM system (modified from a figure in [67]). 44

2-6 Benefits of memory access scheduling (an example taken from [81]). It takes 56 cycles to service the 8 memory requests without memory access scheduling, but only 19 cycles with memory access scheduling. 45

2-7 Priority-based versus frame-based queueing and scheduling. Each request is annotated with the queue number it is associated with. 48

2-8 A network node implementing Fair Queueing (FQ) algorithm [21]. 49

2-9 A network node implementing Stop-and-Go scheduling with a double-FIFO structure [27]. 50

2-10 QoS-capable memory controllers and fair networks are not sufficient to provide memory system QoS. In this hotspot example, all sources except (0, 0) which is reserved for the OS, generate memory requests targeting a DRAM bank in the QoS-capable memory controller located at (1, 2). With an on-chip network implementing locally fair arbitration (a), a source's service level depends on distance from the controller. Adding infinite queues at the memory controller (b) would provide a globally fair guarantee. An end-to-end QoS system based on GSFM (c) can provide fair sharing with finite queues at the memory controller. 53

2-11 Component-wise versus end-to-end approaches for QoS with three major operations at each node: packet classification (C), priority assignment (A) and priority enforcement (E). 54

2-12 Four quadrants of the solution space for QoS. 58

3-1 A three-node queueing network with perfect priority queues with infinite capacity. Dotted rectangles are a network component called "MUX", which merges two incoming flows into a single outgoing one. Each packet carries an associated deadline and the priority queue can dequeue the packet having the earliest deadline. The arbiter and the priority queue are assumed to have zero-cycle delay, which implies a packet just generated at any source can be immediately forwarded to the sink at channel rate C through the combinational path if the packet wins all arbitrations on the path. 64

3-2 Per-source accepted throughput at the sink with various arbitration schemes in the network shown in Figure 3-1. Dotted vertical lines indicate minimum injection rate causing congestion. In locally-fair round-robin arbitration in (a), which does not use the deadline field, the throughput of a flow decreases exponentially as the number of hops increases. Age-based arbitration in (b), where deadline is assigned as network injection time, gives fair bandwidth allocation among all flows. With deadline-based arbitration with the policy for bandwidth guarantees in (c), we achieve bandwidth distribution proportional to the ratio of ρ's ($\rho_1 : \rho_2 : \rho_3 : \rho_4 = 0.30 : 0.50 : 0.15 : 0.05$) in face of congestion. 65

3-3 Step-by-step transformation towards a frame-based, approximate implementation of deadline-based arbitration. (a) shows the ideal MUX introduced in Section 3.2.1, which is associated with each physical link. In (b), we group all data entries having a range of deadlines (whose interval is F) to form a *frame*. The frame number is used as a coarse-grain deadline, and we now drop the deadline (or frame number) field because each frame has an associated frame buffer. Finally, we recycle frame buffers, with W active frames, as in (c) to give a finite total buffer size. 66

3-4 A simple chip-wide Stop-and-Go system that provides guaranteed QoS with minimum bandwidth and maximum network delay bound. 71

3-5 Overlapping multiple frames to improve the network utilization. Future frames can use unclaimed bandwidth by the current frame opportunistically. 72

3-6 Frame life time analysis for comparison of frame reclamation rates with and without early reclamation. Although the estimated $e^{MAX} = 1500$ is within 10% of the observed worst-case e^{MAX}, the early reclamation increases the frame reclamation rate by >30%, which leads to corresponding improvement in the average throughput. See Section 3.5.1 for simulation setup. We use a hotspot traffic pattern with injection rate of 0.05 flits/cycle/node. 74

3-7 A GSF router architecture for 2D mesh network. Newly added blocks are highlighted while existing blocks are shown in gray. 75

3-8 Fair and differentiated bandwidth allocation for hotspot traffic. (a) shows fair allocation among all flows sharing a hotspot resource located at (8,8). In (b), we partition a 8×8 CMP in mesh topology into 4 independent groups (e.g., running 4 copies of virtual machine) and provide differentiated services to them independent of their distance from the hotspot. In (c), we partition a 16×16 CMP in torus topology into 2×2 processor groups and allocate bandwidth to four hotspots in a checkerboard pattern. 80

3-9 Average packet latency versus offered load with three traffic patterns. For each traffic pattern, we consider three different synchronization costs: 1 (GSF/1), 8 (GSF/8) and 16 cycles (GSF/16). Network saturation throughput is the cross point with dotted line ($3T_{zero-load}$ line) as in [49]. With GSF/16, network saturation throughput is degraded by 12.1 % (0.33 vs. 0.29) for uniform random, 6.7 % (0.15 vs. 0.14) for transpose and 1.1 % (0.88 vs. 0.87) for nearest neighbor, compared to baseline VC router with iSlip VC/SW allocation. 81

3-10 Tradeoff in buffer organization with hotspot and uniform random traffic patterns. VC buffer configuration is given by $V \times B$. The frame window size (W) is assumed to be V or 2V. For both traffic patterns, having $V \geq 4$ achieves 90% or higher throughput compared to the baseline VC router. Generally, increasing VCs improves the throughput at the cost of increased average latency. 6×5, our default, is a sweet spot for the specific router architecture we use. 82

3-11 Throughput of GSF network normalized to that of the baseline VC router with variable F (frame window size). Two traffic patterns (hotspot and uniform random) and three synchronization costs (1, 8 and 16 cycles) are considered. 83

4-1 (a) A shared memory system with private L2 caches. The injection control is done at the boundary between private and shared resources. (b) Dots in (b) show distinct scheduling/arbitration points in the shared memory system. 86

4-2 Target CMP architecture and L2-memory coherence network organization(C: local L2 cache, H: directory controller, or *home*, M: memory controller). In (a), the individual blocks in the scope of this thesis are shown in gray. . . 91

4-3 Components of Tessellation OS (taken from [58]). The Tessellation OS requires hardware partitioning mechanisms to provide space-time partitioning (STP) to platform users. 92

4-4 Cache coherence protocol operations and their resource usage (C: local L2 cache, H: directory controller, or *home*, M: memory controller). We can infer from this figure the maximum number of coherence messages that one original request into the c2hREQ network can generate. A thin arrow indicates a header-only message and a thick arrow indicates a message with payload data. 94

4-5 A timing diagram of Micron's DDR2-800 part: MT47H128M8. The worst-case service rate is determined by $t_{RC(MIN)}$. ($t_{RCD(MIN)}$: Activate-to-read, $t_{RTP(MIN)}$: Internal read-to-precharge, $t_{RP(MIN)}$: Precharge-to-activate, $t_{RAS(MIN)}$: Activate-to-precharge.) 96

4-6 Datapath of source injection control. Per-resource type (not per-vector element) comparators are needed because a request cannot use more than one elements from each resource type simultaneously. 98

4-7 Injection controller for GSF memory system. 99

4-8 Our proposed frame-based memory scheduler. Components that are needed in conventional priority-based scheduler but not in our scheduler are grayed out for comparison. Each bank maintains not per-flow queues but per-frame queues which are generally more scalable to the number of flows. The scheduler block chooses the request with the earliest frame number from the per-frame queues. 101

4-9 Job placement on 4×4 tiled CMP. This CMP is partitioned into four quadrants (NW, NE, SW and SE). Each tile is labeled with its CPU ID. 104

4-10 Per-source memory access throughput normalized to DRAM channel bandwidth (synthetic benchmarks). Horizontal red lines indicate the guaranteed minimum bandwidth in GSFM for given token allocations. Each configuration is annotated with the overall memory throughput over all threads ("overall") and the throughput ratio between the most and least served threads ("max/min"). 105

4-11 Per-source memory access throughput with `hotspot_channel` and `stream` benchmarks for larger miss status handling registers (MSHR). As a source can inject more requests into the memory system with a larger MSHR, the throughput variation in GSFM decreases, whereas the variation in the best-effort case increases. In both benchmarks, all threads in GSFM receive more than guaranteed minimum bandwidth. 106

4-12 Memory access throughput of a `hotspot_channel` thread with variable number of memory performance hogs (MPHs) and different placement locations. Minimum performance guarantees for foreground process are enforced regardless of the number of background processes or placement locations. 106

4-13 Per-source memory access throughput when only one source is activated. The average throughput degradation caused by QoS support is less than 0.4 % over the four microbenchmarks. 108

4-14 Execution time of PARSEC benchmarks normalized to the execution time with a zero-contention best-effort memory system. The lower, the better. The average slowdown (geometric mean) in a high-contention environment is 4.12 (without QoS) and 2.27 (with QoS). 109

4-15 Cumulative L2 miss count over time. On X axis, the completion time of parallel region for each configuration is marked with a label. The lines for BE/MPH0 and GSFM/MPH0 configurations almost completely overlap. The slope of a line is the L2 miss throughput for the corresponding configuration. 109

5-1 The METERG QoS system. Each QoS block takes an extra parameter (OpMode) as well as a resource reservation parameter ($x_{i,j}$) for each processor. 119

5-2 An example of having longer latency in spite of more bandwidth with a strict time-division multiplexing (TDM) scheduling. 120

5-3 An implementation of the METERG system supporting strict enforcement mode. We use delay queues to meet the latency condition (Condition 2) required by strict enforcement mode. 123
5-4 A simple bus-based METERG system. The memory controller is capable of strict METERG QoS. 124
5-5 Performance of `memread` in various configurations. BE, ENF, and DEP stand for best-effort, enforcement-mode, and deployment-mode execution, respectively. In (a), as the number of concurrent processes increases from 1 to 8, the performance of a process degrades by 46 % without QoS support, but only by 9 % in deployment mode. The estimated performance from a measurement in strict enforcement mode indicates the performance degradation for the given resource reservation to be 31 % in the worst case. In (b), we observe that the performance estimation in strict enforcement mode becomes tighter as the resource allocation parameter (x_1) increases. 126
5-6 Interactions between QoS and Best-effort (BE) processes running `memread`. All QoS processes are running in deployment mode. Even if the number of QoS processes increases, the performance of QoS processes degrades very little as long as the system is not oversubscribed. 126

List of Tables

3.1 Variables and parameters used in GSF. 67

3.2 Default simulation parameters 78

4.1 Four logical networks for MOSI protocol we use for our implementation. . 91

4.2 (Injection) token management table. 94

4.3 Resource usage table for our target CMP architecture and cache coherence protocol. This table illustrates how many service tokens can be consumed from each resource by one GETS, GETX or PUTX request into the c2hREQ network. H stands for header-only packet, P stands for header+payload packet and T stands for memory transaction. N is the number of sources, which is 16 in our target architecture. The numbers in parentheses shows the number of tokens assuming 64-byte cache line size and 64-bit flit size. . 95

4.4 Maximum sustainable service rate from each resource [in tokens] under worst-case access patterns assuming F=10560 [cycles] (= 96 worst-case memory transaction time). For network resources, one token corresponds to one flit. For memory resources, one token corresponds to one memory transaction (read or write). These numbers specify the total number of tokens that can be allocated to all sources (i.e., $\sum_i R_i^j$ must be smaller than the corresponding number for Resource j). 97

4.5 Default simulation parameters 103

Chapter 1

Introduction

As semiconductor technology continues to improve and reduces the cost of transistors, computing devices are becoming pervasive. The difficulty of meeting power-performance goals with single-core designs has led to the adoption of multicore architectures, or chip multiprocessors (CMP), in all scales of systems, from handheld devices to rack-mounted servers. These platforms will be required to support a variety of complex application workloads, with possibly hundreds to thousands of concurrent activities competing for shared platform resources. These concurrent threads (activities) exhibit complex non-deterministic interactions when accessing common system resources, such as the memory system, causing individual thread performance to exhibit much wider variability than they would in a single-core platform. This performance variability makes it hard to provide performance guarantees (e.g., real-time tasks), degrades parallel program performance, and burdens software writers by making performance tuning and load balancing more challenging. As the number of processors increases, problems are usually exacerbated.

In this thesis, we propose a novel Quality-of-Service (QoS) framework, called *Globally Synchronized Frames (GSF),* to improve the performance predictability of one of the most important shared resources in CMPs—the shared memory system. GSF can provide guaranteed QoS for each thread in terms of minimum bandwidth and maximum delay, as well as proportional bandwidth sharing in a cache-coherent shared memory system. GSF employs a single unified framework to manage three important bandwidth resources in a shared memory system: on-chip network links, DRAM channels and DRAM banks. The

GSF framework takes a frame-based approach, which can be efficiently implemented in a resource-constrained on-chip environment.

1.1 Performance Variability in Multicore Platforms

General-purpose computing systems have previously contained one (or at most a few) processors, and have mostly used time-sharing to handle concurrently running tasks. Each task had the whole machine to itself during its time slice, providing a simple form of performance isolation (modulo the edge effect of switching contexts). Once scheduled, a task's execution time is solely determined by the executable itself and the platform on which it is running, both of which can be relatively well analyzed, modeled and/or profiled.

However, the recent move to multicore systems introduces a significant performance variability problem caused by shared resource contention, making it much harder to obtain predictable performance [5]. Future multicore systems will contain dozens and possibly hundreds of cores, and efficient utilization of the resources will require that many tasks run simultaneously via space as well as time sharing. Although current operating systems can certainly space-share the multiple cores of current hardware platforms, the performance of each task is highly dependent on the behavior of other tasks running in parallel. As the number of threads accessing shared resources grows, this performance variability is expected to grow rapidly and make it difficult to reason about program behavior [30, 31, 34, 65]. The number of combinations of co-scheduled tasks increases exponentially with the number of concurrent threads, making it infeasible to perform performance analysis and profiling for all possible usage scenarios.

In addition to the growing number of concurrent threads on a chip, the following technology trends will likely exacerbate the performance predictability problem:

Aggressive resource sharing to reduce hardware cost. The physical proximity of cores on a die enables very aggressive microarchitectural resource sharing, which was not feasible in multi-chip multiprocessors. Shared L2 caches [38, 45] and memory controllers [38, 40, 45, 47, 48] are already commonplace in commercial designs. Kumar et al propose a conjoined architecture for finer-grain sharing of microarchitectural resources

such as floating-point units, crossbar ports, L1 instruction caches, and data caches [50]. Simultaneous multithreading [22] also aims to maximize the utilization of microarchitectural resources such as instruction issue logic and functional units. Higher resource utilization through sharing enables lower-cost design alternatives by saving chip area and energy. However, increased resource sharing leads to increased performance coupling between sharers and wider variability in individual thread performance [73].

Programmable architectures executing a wide variety of workloads. The growing requirements of performance per watt combined with increasing development costs for modern chips favor relatively few highly-optimized general-purpose platforms in all scales [46]. These platforms are required to support a variety of complex application workloads, including both soft real-time and best-effort tasks. For example, a handheld device might support various types of media codecs running alongside best-effort background tasks such as email synchronization and file transfer, on an embedded multiprocessor core [48]. As another example, the provider of a co-located server farm may use OS virtualization to reduce computing costs by supporting multiple customers on a single multiprocessor server with different service-level agreements. These computing platforms are now exposed to more complex and diverse non-deterministic interactions among tasks, which cannot be completely analyzed at platform design time. As a result, application profiling and performance tuning is becoming more challenging.

Minimal resource overprovisioning for energy efficiency. The main driving force for multicore systems is their superior energy efficiency over single-core systems [5]. Energy efficiency is crucial in both large machine room servers and small client devices such as laptops and handhelds—for lower electricity bills in the former and for longer battery life in the latter.

Therefore, designers avoid overprovisioning hardware resources to maximize energy efficiency. Agarwal and Levy propose the "KILL" (Kill If Less than Linear) rule [2] for sizing a resource in a multicore chip. They claim that any resource in a core can be increased in area only when it is justified by an improvement in the core's performance that is at least proportional to the core's area increase. This rule can be applied to other shared resources on the chip. Since common cases receive higher weights in evaluating the over-

all performance improvement, the KILL rule advocates allocating *just enough* resources to maintain good performance in *common* use cases.

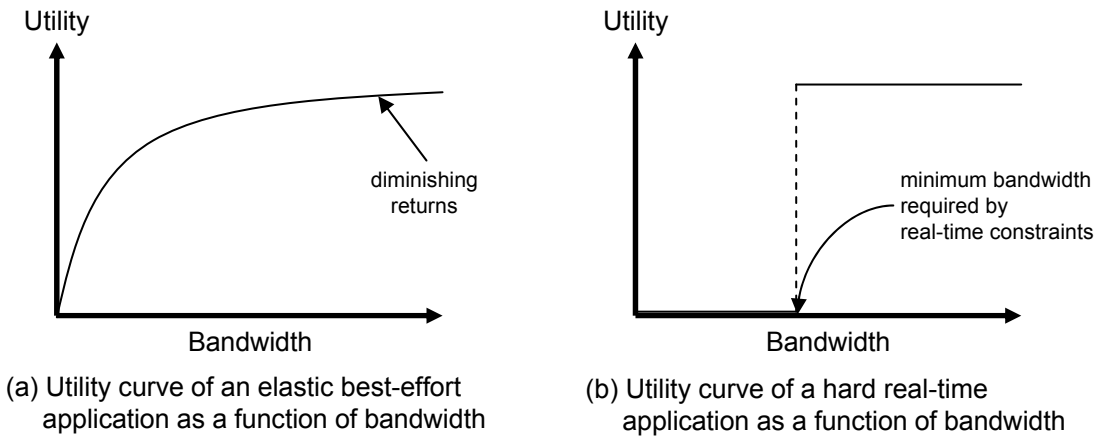
However, a design with minimal resource margins often results in performance vulnerability in less common scenarios. For example, one could aggressively shrink the transaction buffer of each memory controller, assuming a common-case access pattern where memory requests are evenly distributed across all of the memory channels. If the requests are highly imbalanced and directed towards one or just a few memory channels, the system can severely penalize requests from remote nodes because of tree saturation [90], which floods the on-chip network as well as the transaction buffer. In such a case, there is a large variability of memory throughput among concurrent threads depending on their distance to the memory controller(s) in question.

1.2 The Case for Hardware Support for QoS

We believe that future integrated multicore platforms must implement robust Quality-of-Service (QoS) support to maximize their utility. In computer networking, a network that can provide different levels of packet delivery service in terms of latency, bandwidth, or both, is said to support QoS [78]. For CMP systems, *performance isolation*[1] and *differentiated services* are examples of relevant QoS objectives [53]. Performance isolation is the property that a minimum level of performance is guaranteed regardless of other concurrent activities (e.g., preventing denial-of-service attacks to DRAM channels [65]). Differentiated services is the ability to allocate each resource flexibly among competing tasks. We discuss a taxonomy of QoS in Section 1.3.

In existing best-effort platforms, individual thread performance is only determined by interactions with other threads, which are generally unpredictable and uncontrollable. While the worst-case performance bound might be derived by assuming that every single access experiences the worst-case contention for a given system configuration, this bound is generally too loose to be useful. Best-effort hardware does not expose fine-grain knobs to

[1] Performance isolation is a vaguely defined term, and its usage is not consistent in the literature [53, 60, 71, 93]. In this thesis, we use both guaranteed services and performance isolation interchangeably.



(a) Utility curve of an elastic best-effort application as a function of bandwidth

(b) Utility curve of a hard real-time application as a function of bandwidth

Figure 1-1: Utility curves of elastic best-effort and hard real-time applications (taken from [83]).

enable the OS to implement flexible resource management policies in a highly-concurrent multicore environment. We advocate hardware support for QoS for the following reasons:

Supporting real-time applications. Real-time applications are a class of applications whose usefulness depends not only on the correctness but also on the timeliness of their results [78]. Figure 1-1 shows the utility curves of an elastic best-effort and a hard real-time application as a function of bandwidth received from a critical resource. The utility of traditional best-effort data applications such as file transfer and emails degrades gracefully as the bandwidth they receive decreases. On the other hand, the utility of real-time applications such as media players and video games, sharply falls to zero as soon as the bandwidth share drops below that needed to meet the real-time constraints. Without hardware QoS support, the only way to meet the performance goals is to overprovision bandwidth, which will lead to an overly conservative rejection of other real-time tasks and/or reduced battery life on a mobile platform. To effectively support real-time tasks in a space-time shared multicore environment, shared resources should be capable of guaranteed QoS in terms of minimum bandwidth and/or maximum latency.

Improving parallel application performance QoS support from shared resources can reduce the variance in communication delays among cooperating threads as well as their waiting time at a synchronization point, which can improve a parallel application's execution time [66]. In addition, a multicore platform with QoS can provide more robust

performance to a parallel application regardless of its placement on the machine because it can receive location-independent fair/guaranteed services from shared resources.

Providing performance isolation and protection for concurrent tasks. It is a possible scenario on a highly-concurrent multicore platform that one resource-hungry task hogs up a critical shared resource and slows down the other concurrent tasks without bound. To provide performance isolation and protection effectively for multiple contending tasks, the platform should be able to partition not only the cores but also the bandwidth of other resources shared in both time and space, such as on-chip interconnects and memory channels.

Easing performance tuning and load balancing. Another important argument for QoS is performance tuning and load balancing [58]. In a highly non-deterministic multicore environment, it is difficult, if not impossible, to tune an application's performance because the observed performance can vary significantly depending on the interactions with other concurrent applications. Also, several types of parallelism such as bulk-synchronous data parallelism and streaming pipeline parallelism are known to work best with load balancing (e.g., equalizing production and consumption rates) made possible by hardware QoS mechanisms.

1.3 A Taxonomy of QoS

We characterize a QoS system with two parameters: the number of static priority levels (denoted by P) and bandwidth allocation policy for each priority level. Although there are many QoS metrics (e.g., bandwidth, latency, jitter, etc.), our primary QoS metric in this section is service bandwidth. We use networking terms such as packets and flows for discussion, but the taxonomy is applicable to any other shared resources (e.g., caches, off-chip memory). (A flow is defined as a distinct sequence of packets between a single source and a single destination.)

Static priority levels. The first axis of QoS characterization is the number of static priority levels. One of the simplest approaches to QoS is to assign a static priority to each flow and give the same priority to all of the packets in that flow. At every scheduling point a request with the highest static priority is always serviced first. In computer networking,

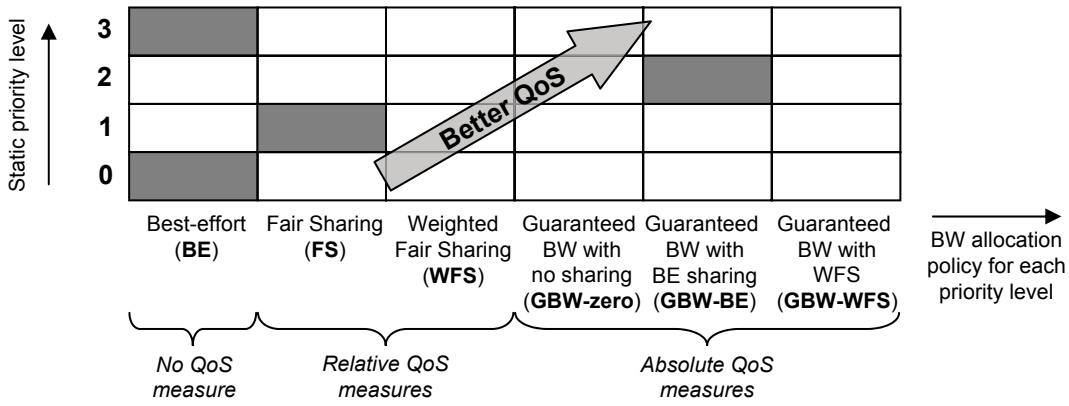


Figure 1-2: Characterization of a QoS system with two parameters: the number of static priority levels (P=4 in this example) and bandwidth allocation policy for each priority level. Different priority levels can enforce different bandwidth allocation policies as shown by gray boxes. Better QoS is provided as a flow's policy is placed more up (higher priority) and to the right (more sophisticated QoS bandwidth allocation) in the grid.

the Expedited Forwarding (EF) class in DiffServ [17] is an example of this type of QoS, where packets marked for EF treatment should be forwarded by the router with minimal delay and loss (i.e., with highest priority). To guarantee this QoS to all EF packets, the arrival rate of EF packets at the router is often strictly controlled so as not to oversubscribe the service capacity for EF-class packets [78]. Realistic implementations of QoS based on static priorities also deal with the priority inversion problem in which a higher-priority packet is blocked by lower-priority ones (e.g., with per-static priority queues) [42].

However, static priority level-only QoS is a very limited form of QoS and inadequate for the demands discussed in Section 1.2. The main drawback of this approach is that QoS is provided only to the flows with the highest priority. There is another orthogonal axis to classify more sophisticated QoS systems—bandwidth allocation policy for each priority level.

Bandwidth allocation policy for each priority level. This policy determines how to distribute service bandwidth among multiple flows in the same priority level. We introduce six bandwidth allocation polices as illustrated in Figure 1-2. These policies are described as follows from the least to the most sophisticated QoS.

- **Best-effort (BE).** No QoS is provided for flows in the priority level. First-Come-

First-Served (FCFS) and input port round-robin (RR) are example scheduling mechanisms that implement this policy.

- **Fair Sharing (FS).** One of the desired QoS properties is fairness among concurrent flows—Flow A receives as much bandwidth as Flow B, for example. Note that it does not specify any bandwidth share in absolute terms; the QoS a flow receives is solely defined in relation to the QoS of the other flows. In other words, FS uses a relative QoS measure. Fair Queueing (FQ) [21] and age-based arbitration [20, 77] are well-known FS mechanisms. FQ provides *min-max fairness* where flows whose injection rate is smaller than its fair share receive the throughput equal to its injection rate, whereas the other flows converge to some throughput value, denoted by α [6].

- **Weighted Fair Sharing (WFS).** WFS is a generalized form of FS. Each of N flows sharing a link has an associated parameter ρ_i where $0 \leq \rho_i \leq 1$ and $\sum_{i=1...N} \rho_i \leq 1$. WFS distributes the available service bandwidth to flows in proportion to the ratio of ρ's. Weighted Fair Queueing (WFQ) and Virtual Clock [98] are examples of WFS mechanisms.

- **Guaranteed Bandwidth with no Sharing (of excess bandwidth) (GBW-zero).** A guaranteed bandwidth policy provide a minimum level of guaranteed bandwidth for a flow regardless of other concurrent activities. For example, guaranteed services can use an absolute measure to describe QoS like "Flow A receives bandwidth of at least 1 Gb/sec." GBW-zero enforces zero sharing of excess bandwidth. That is, any unreserved or unclaimed (=reserved but not used) service slot is wasted. Strict time-division multiplexing (TDM) and multi-rate channel switching [92], where each service slot is allocated to a specific flow and only to this flow, are example QoS mechanisms of the GBW-zero policy. To provide guaranteed bandwidth in a priority level, the absolute amount of injected packets in higher priority levels must be strictly controlled because high priority-level flows can starve low priority-level flows without bound.

- **Guaranteed Bandwidth with Best-effort Sharing (of excess bandwidth) (GBW-**

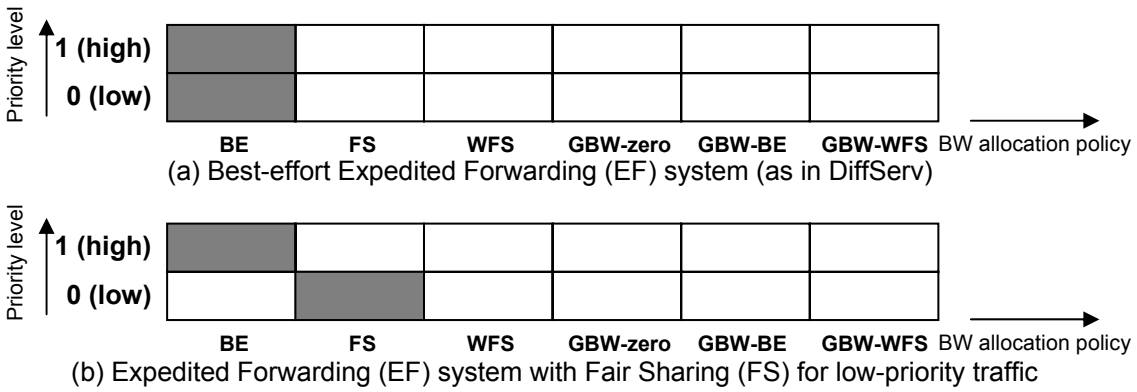


Figure 1-3: Comparison of two QoS systems implementing Expedited Forwarding (EF). The number of static priority levels is two (P=2).

BE). GBW-BE enforces minimum bandwidth guarantees among flows and distribute any unreserved or unclaimed bandwidth with best effort (e.g., round-robin). It is feasible to augment strict TDM with a best-effort scheduler (e.g., round-robin across flows) to reclaim unused bandwidth.

- **Guaranteed Bandwidth with Weighted Fair Sharing (of excess bandwidth) (GBW-WFS).** Our desired QoS in this thesis consists of both guaranteed bandwidth and proportional sharing of excess bandwidth. Guaranteed bandwidth is important to support real-time applications that require strict minimum performance, and proportional bandwidth sharing to maximize the overall system utility by flexible bandwidth allocation. GSF is our proposal to implement the GBW-WFS policy. In a typical setup, we only use one set of ρ's to specify both the minimum guaranteed bandwidth (as a fraction of the link bandwidth) and the proportional share of excess bandwidth.

Different priority levels can enforce different bandwidth allocation policies. For example, Figure 1-3 compares two QoS systems implementing Expedited Forwarding (EF). Both systems have two priority levels, and high-priority packets are always serviced first over low-priority packets. However, they differ in their bandwidth allocation policy for low-priority traffic. The system in Figure 1-3 (a) services low-priority packets with best effort as in DiffServ [12], whereas the system in Figure 1-3 (b) enforces fair sharing among low-priority flows.

1.4 Design Goals for QoS Mechanisms

This section presents a set of design goals in our development of QoS mechanisms for a shared memory system in CMPs. These goals are used as metrics by which we evaluate our proposed QoS mechanisms and other existing mechanisms.

Robust and flexible QoS. The first goal is to provide robust and flexible QoS that performs well over a wide range of usage scenarios. More specifically, we take into account the following:

- It is highly desirable to provide both proportional bandwidth sharing and guaranteed services in terms of minimum bandwidth and maximum delay to memory to meet the complex needs of future CMPs.

- There are many applications that are latency-sensitive, so a QoS mechanism should not increase the average memory access latency when the memory system is not congested. For example, strict time-division multiplexing (TDM) and Stop-and-Go [27] schemes are not acceptable because of their increased average latency in an uncongested environment.

- If the overhead to create a QoS channel is too long, it will penalize a short-lived flow, so it is desirable to reduce setup latency.

High resource utilization. The memory system in a CMP is one of the most important shared resources, and a QoS mechanism for it should achieve a high resource utilization, which is at least comparable to a best-effort memory system. Consequently, *work-conserving* scheduling algorithms (i.e., no idle cycles when a request is waiting to be serviced) are preferred whenever possible. Also, a QoS mechanism that efficiently handles best-effort traffic is desirable to improve utilization.

Low hardware cost. Another important metric is the implementation cost of a QoS mechanism (in terms of an increase in transistor count as compared to a best-effort memory system). A good QoS proposal should lend itself to efficient implementation in a resource-constrained on-chip environment. It should also be scalable to an ever-increasing number

of processor cores. One implication of the scalability requirement is that per-flow queues and data structures should be avoided whenever possible.

Seamless integration to provide an end-to-end QoS. All the components that constitute a shared memory system should be composable to provide a seamless end-to-end QoS in the memory system. These components are expected to implement compatible QoS mechanisms, and every scheduling point should be carefully analyzed to prevent QoS breakdown. Taking into account cache-coherence protocol operations would be an additional benefit.

1.5 Contributions

The single most important contribution of this thesis is a new hardware mechanism to provide proportional bandwidth sharing and guaranteed QoS in a cache-coherent shared memory system. We refer to this mechanism as *Globally Synchronized Frames* (GSF). The goal is to provide an architectural mechanism whereby allocations of shared bandwidth resources (not storage space) in the cache miss path that originated at the last-level private cache (LLPC) can be guaranteed[2] More specifically, there are two contributions related to GSF:

- **GSF on-chip networks**. Although network QoS problems have been well investigated in the networking community, the on-chip environment has different opportunities and challenges compared to the off-chip environment. GSF provides flexible and robust QoS guarantees as well as proportional bandwidth sharing for on-chip networks without significantly increasing the router complexity over best-effort routers. It is our understanding that this work is the first application of global frame-based bandwidth allocation to on-chip networks.

[2]The LLPC is defined as the outermost-level cache that is privately owned by a processor core and a gateway to the shared memory system. The LLPC is not necessarily the last-level cache (LLC). For example, assuming two-level caches, if both L1 and L2 caches are private, the L2 cache is both the LLPC and the LLC. However, if the L2 cache is shared by all cores, the L1 cache is the LLPC but not the LLC. We illustrate this distinction in more detail in Section 2.1.

- **GSF memory system (GSFM) for cache-coherent shared memory**. GSFM is a complete memory hierarchy design with guaranteed QoS, targeted for future many-core platforms. We extend one-dimensional GSF used in the on-chip network into multi-dimensional GSF managing multiple heterogeneous resources. Unlike prior work focusing on designing isolated QoS-capable components, such as caches, memory controllers and interconnects, GSFM is a single unified QoS framework for QoS, encompassing all these components. An advantage of this unified framework is better composability of QoS than component-wise approaches. In general, it is difficult to compose separately designed QoS blocks to provide end-to-end QoS, as they often provide very different service guarantees or accounting mechanisms (e.g., fair bandwidth distribution vs. fair slowdown). GSFM provides guaranteed minimum bandwidth and maximum delay bounds to each core when requesting a service (e.g., cache miss) outside the last-level private cache (LLPC). The hardware cost of GSFM is low because of its frame-based, end-to-end approach.

The final contribution of the thesis is the METERG (MEasurement Time Enforcement and Runtime Guarantee) QoS framework, used to help find a minimal resource reservation setting for a given user-level performance goal:

- **METERG QoS framework**. Assuming QoS support from shared resources, the METERG QoS framework is proposed to help software stacks estimate a minimal bandwidth reservation (e.g., MB/s) required to meet a given performance goal in a user-specified metric (e.g., transactions per second). This framework provides an easy method of obtaining a tight estimate of the lower bound on end-to-end performance for a given configuration of resource reservations.

1.6 Thesis Organization and Previous Publications

Before presenting the main contributions in detail, we first provide background in Chapter 2. This chapter begins with reviewing the basics of CMP memory hierarchy design and identifying the bandwidth resources of interest. We classify the solution space into four

quadrants to discuss the pros and cons of each quadrant.

Chapter 3 introduces one-dimensional GSF and its application to on-chip networks for proportional bandwidth sharing and guaranteed QoS in terms of minimum bandwidth and maximum delay. This chapter revises a previous publication [53]: Jae W. Lee, Man Cheuk Ng, and Krste Asanović, Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks, the Proceedings of the 35th IEEE/ACM International Symposium on Computer Architecture (ISCA-35), Beijing, China, June 2008.

Chapter 4 extends GSF to be multi-dimensional so that it can handle multiple heterogeneous bandwidth resources—network links, DRAM channels, and DRAM banks—in a single unified QoS framework. This chapter revises an unpublished manuscript [52]: Jae W. Lee, Sarah Bird, and Krste Asanović, An End-to-end Approach to Quality-of-Service in Shared Memory Systems, under review for external publication at the time of writing this thesis.

Chapter 5 presents the METERG QoS framework. This chapter revises a previous publication [51]: Jae W. Lee and Krste Asanović, METERG: Measurement-Based End-to-End Performance Estimation Technique in QoS-Capable Multiprocessors, the Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), San Jose, CA, April 2006.

Chapter 6 summarizes the thesis and suggest possible extensions. We discuss further optimization ideas for GSF and software stacks to provide service-level QoS.

Chapter 2

Background

In this chapter we first review the CMP memory hierarchy design, then describe the challenges in supporting QoS using per-component techniques. Conventional best-effort memory systems have been designed to maximize the overall throughput and resource utilization, but have not taken into account fairness issues among sharers. We explain how the current design practice of each component in the memory hierarchy causes fairness problems which, in combination, lead to memory system-wide unfairness. We then introduce four quadrants of solution space according to scheduling granularity (*priority-based* versus *frame-based*) and extent of QoS (*component-wise* versus *end-to-end*). We conclude this chapter with arguments for a frame-based end-to-end approach to provide efficient support for QoS in a resource-constrained on-chip environment.

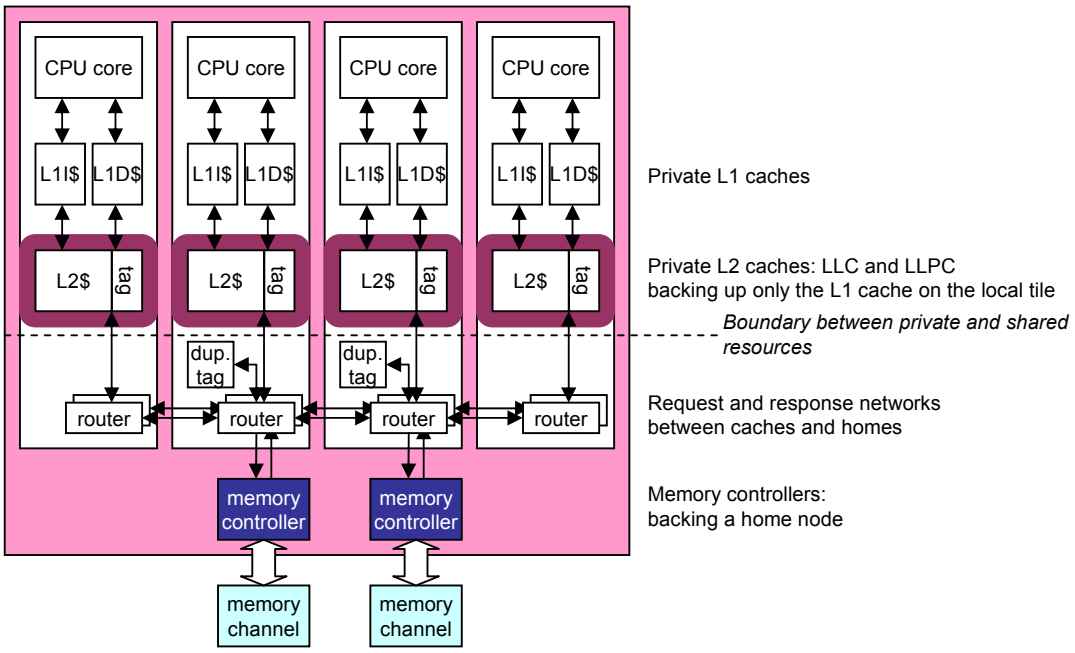
The memory hierarchy of a chip multiprocessor (CMP) is a highly distributed structure, where multiple caches and DRAM memory controllers communicate over multi-hop on-chip networks. To provide seamless QoS guarantees, all the components constituting the memory system should implement an adequate QoS mechanism, as an application's guaranteed service level in memory accesses is determined by the weakest guarantee for any of these components. More specifically, we focus on the three most important bandwidth resources: on-chip network links, DRAM channels and DRAM banks.

2.1 CMP Memory Hierarchy Designs

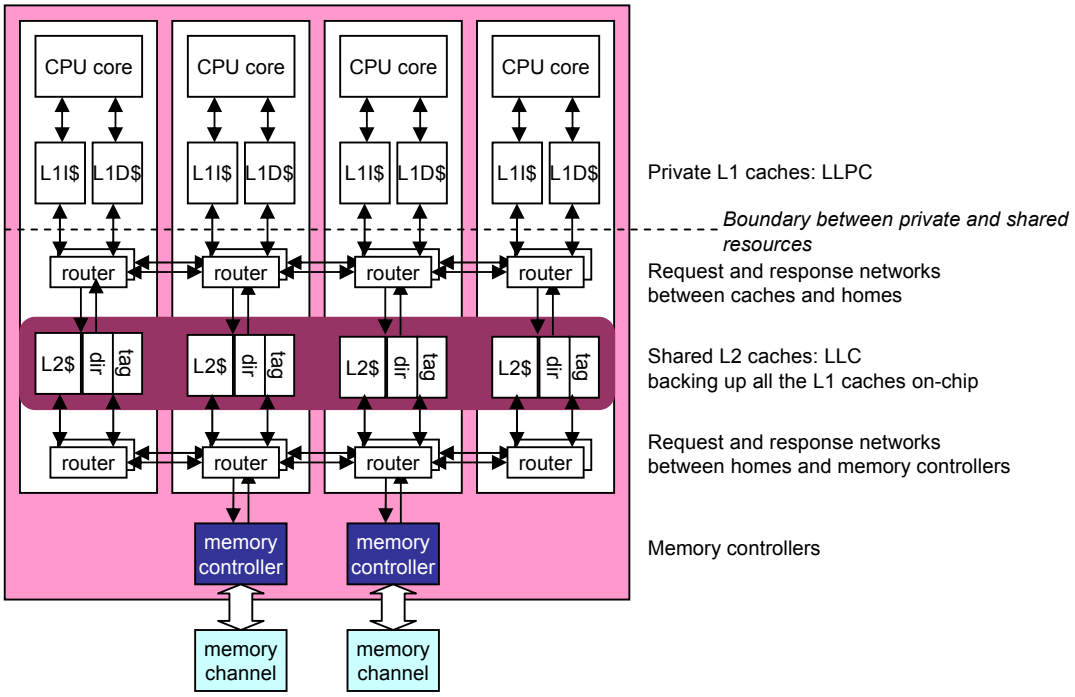
The organization of the memory hierarchy in future CMPs is a major research problem on its own. The primary instruction and data caches are typically kept small and private to each core to give a low hit access latency. The outer caches will likely be divided into multiple *slices* connected over an on-chip interconnect to maximize throughput for a given area and power budget. As the chip size in transistor count grows, both the number of cores and the number of cache slices will increase, making it more challenging to design a distributed memory hierarchy and have it achieve robust performance. Likewise, DRAM controllers will be replicated and distributed across a chip to meet the increasing storage and bandwidth demands.

One straightforward way to build a distributed outer-level CMP cache is to simply shrink the distributed shared memory (DSM) design of a traditional multi-chip multiprocessor, where each processor has a private L2 cache. Figure 2-1 (a) illustrates an example of private L2 caches in a 4-core CMP. Each node, or *tile*, consists of a CPU core, private L1 caches and a slice of L2 cache. Each slice of L2 cache is associated with the CPU core at the same node, and replicates cache lines freely as the core accesses them. We assume a high bandwidth on-chip directory scheme to keep the multiple L2 caches coherent [100], with the directory held as a duplicate set of L2 tags distributed across multiple homes co-located with memory controllers. Cache-to-cache transfers are used to reduce off-chip requests for local L2 misses, but these operations involve a three-hop forwarding: requester to home, home to owner, and owner to the original requester. Because cache lines used by a core are effectively captured into its local L2 slice by replication (shared copy) and migration (exclusive copy), the private L2 scheme gives a low average L2 hit latency. However, static per-core partitioning of L2 cache incurs more capacity misses by allowing multiple copies of the same data to be cached simultaneously, resulting in more expensive off-chip DRAM accesses.

An alternative scheme is a *shared* L2 cache shown in Figure 2-1 (b), where the distributed L2 slices are aggregated to form a single high-capacity shared L2 cache for all nodes. Memory addresses are often interleaved across slices for load balancing, and L2



(a) Private L2 Design



(b) Shared L2 Design

Figure 2-1: Private versus shared L2 cache designs in a 4-core CMP (modified from a figure in [100]). The private L2 design in (a) treats each L2 slice as a private cache and its last-level private cache (LLPC) is L2, which is also the last-level cache (LLC) on the chip. The shared L2 design in (b) treats all L2 slices as part of a global shared cache, where LLPC (L1) is different from LLC (L2). This thesis addresses QoS problems in bandwidth usage at the resources further out than the last-level private caches (LLPC).

hit latency varies according to the number of network hops to the home node and network congestion. The coherence among all L1 caches is maintained by adding additional directory bits to each L2 line to keep track of sharing information of the cache line. Shared L2 caches maximize the L2 caching capacity on a chip because there is no duplication. Unfortunately, the average L2 cache hit latency is usually larger than that of a private cache. There are proposals that aim to benefit from both schemes—data proximity of private L2 and capacity of shared L2 [16, 41, 99, 100]—which we do not discuss here.

This thesis proposes a new unified framework to provide guaranteed minimum bandwidth for each of the distributed last-level private caches (LLPC) to handle misses without degrading the memory system throughput in average-case usage scenarios. As illustrated in Figure 2-1, the LLPC is a gateway to the shared memory system and we address QoS problems in bandwidth usage from the resources at levels further out than the LLPC. Without loss of generality, we assume L2 is the last-level cache (LLC) for the rest of this thesis; a reasoning similar to that between L1 and L2 can be applied between any adjacent levels of outer caches.

2.2 Fairness Problems in Best-Effort Memory Systems

To obtain QoS guarantees from a shared memory system, its individual components need to be aware of the QoS requirement of each sharer and make scheduling decisions accordingly. Figure 2-2 illustrates scheduling/arbitration points while servicing a cache miss at a slice of the last-level cache (LLC) in a CMP platform with multi-hop on-chip networks. Dots in the figure represent distinct scheduling points. The cache miss request first traverses an on-chip network path towards the memory controller. In the network it must go through multiple stages of arbitration (hops) for egress link bandwidth and shared buffers (virtual channels). Once at the memory controller, the request arbitrates for DRAM bank service and channel bandwidth. Finally, the response is sent to the original requester using the response network. Among resources on the LLC miss service path, we are particularly interested in three main bandwidth resources: network links, DRAM channels and DRAM banks.

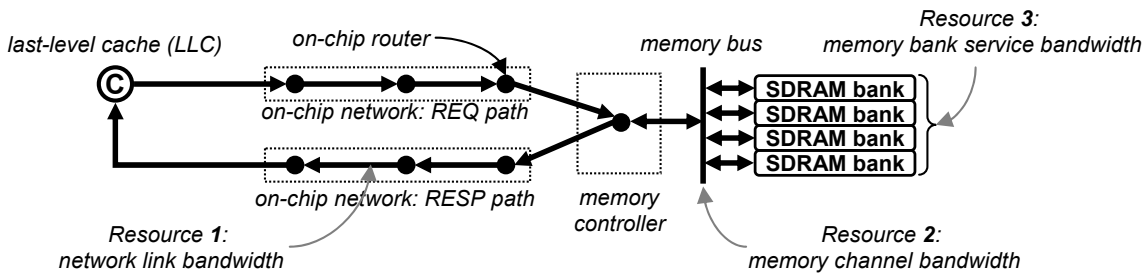


Figure 2-2: Last-level cache (LLC) miss service path with multi-hop on-chip networks. This thesis focuses on three heterogeneous bandwidth resources on the path: (on-chip) network links, DRAM channels and DRAM banks.

Most CMP memory systems to date are *best-effort*; they maximize throughput and minimize latency, but do not address fairness issues among concurrent sharers. However, as the number of CPU cores increases, the performance variation in accessing a shared memory system will increase to make QoS support from the memory system more important. The following subsections discuss fairness problems at individual components constituting a best-effort shared memory system. In so doing, we intentionally use simplified setups to understand the main reasons behind these problems without descending into implementation details.

2.2.1 Fairness Problems in On-Chip Networks

Best-effort on-chip routers do not differentiate flows. (A flow is loosely defined as a distinct sequence of packets between a single source and a single destination.) Instead, most of them implement some variants of locally-fair round-robin arbitration to provide fair service to input ports in allocating buffers and switching bandwidth.

However, local fairness does not imply global fairness. Figure 2-3 shows a simplified example of globally unfair bandwidth sharing. This is a simple three-node network with four flows (whose sources are labeled A, B, C and D) all having a shared destination (labeled "dest"). Assuming the channel rate of C_{ch} [packets/s], the throughput of Flow D is half of the channel rate, because it wins the congested output link with a probability of one half. The throughput of Flow C is a quarter of C_{ch} because it has to go through

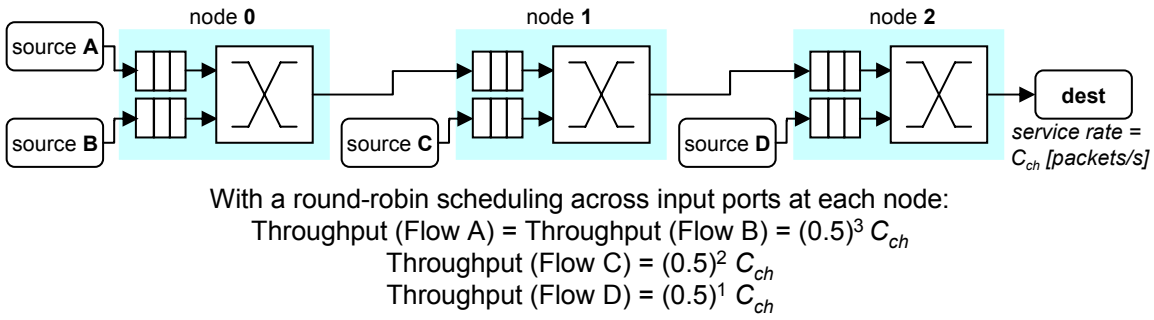


Figure 2-3: Fairness problem in a simple three-node network. The throughput of a flow decreases exponentially as the distance to the hotspot increases.

arbitration twice and has a probability of one half to win each arbitration. Likewise, the throughputs of Flow A and B are each one eighth of C_{ch}. In general, under round-robin arbitration, the degree of uneven bandwidth distribution increases exponentially with the network diameter.

Detailed network simulation with a realistic setup shows this phenomenon as well. We perform simulation in an 8×8 2D mesh network with a hotspot traffic pattern, where all the nodes generate requests to a hotspot node located at (8,8). Figure 2-4 illustrates how the bandwidth in accessing the hotspot is distributed among sharers. X and Y axes show the node index in X and Y directions, and the bar at (x, y) shows the received throughput of the flow whose source is located at (x, y). Round-robin scheduling with dimension-ordered routing (DOR) in Figure 2-4 (b) clearly shows starvation of remote nodes. Adaptive routing [20] can eliminate the imbalance of traffic between X and Y directions and help mitigate the problem in a lightly-congested network, but does not fundamentally resolve the fairness problem, as shown in Figure 2-4 (c). Again, locally-fair round-robin scheduling results in globally-unfair bandwidth usage.

2.2.2 Fairness Problems in Memory Controllers

Figure 2-5 shows a high-level organization of a modern DRAM system. A memory channel consists of one or more Dual In-line Memory Modules (DIMMs). Because one DRAM chip only has a narrow data interface (e.g., 8 bits) due to packaging constraints, multiple DRAM

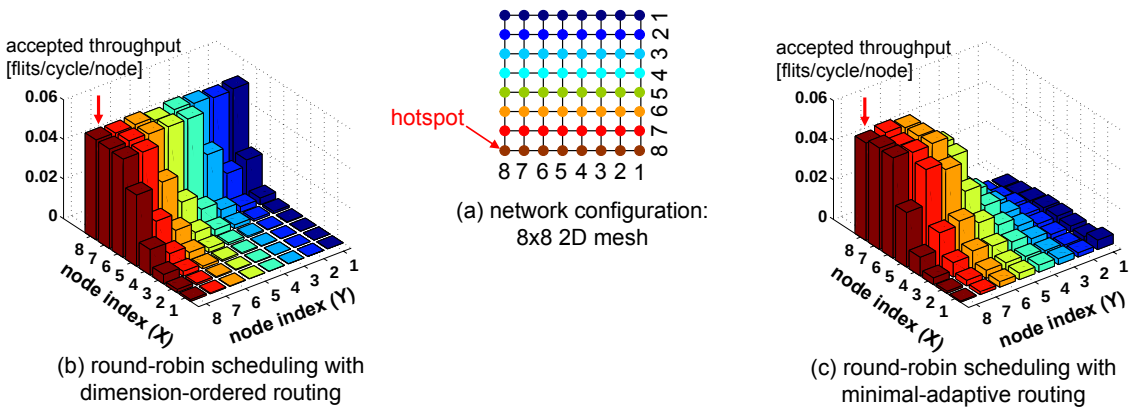


Figure 2-4: Fairness problem in an 8×8 mesh network as in (a). All nodes generate traffic toward a hotspot shared resource located at (8,8) (indicated by arrow) with injection rate of 0.05 (flits/cycle/node), and the bar graph shows accepted service rate per node by the hotspot resource. In (b), locally-fair round-robin (RR) scheduling leads to globally-unfair bandwidth usage, penalizing remote nodes. The minimal-adaptive routing in (c) eliminates the imbalance of traffic between x and y directions in (b), and possibly helps mitigate the problem in a lightly-congested network, but does not fundamentally resolve the problem.

chips are combined and accessed in lock step to provide a wider DRAM channel (e.g., 64 bits) [67].

At odds with their name, modern DRAMs (Dynamic Random Access Memory) are not truly random access devices (equal access time to all locations) but are 3-D memory devices with dimensions of (bank, row, column). More than one outstanding memory request can be serviced in parallel if their addresses are mapped to different banks. Rows typically store data in consecutive memory locations and are around 1-2 KB in size [63]. The data in a bank can be accessed only from the row buffer, which can contain at most one row. Sequential accesses to different rows within one bank have a high access latency and cannot be pipelined, whereas accesses to different banks or different words within a single row have a low latency and can be pipelined [81].

Typical DRAM accesses are performed in the following way. When a memory request arrives, the memory controller issues an `Activate` command to the corresponding bank to transfer the entire row in the memory array containing the requested address into the bank's row buffer. The row buffer serves as a cache to reduce the latency of subsequent accesses to that row. While a row is active in the row buffer, any number of reads and

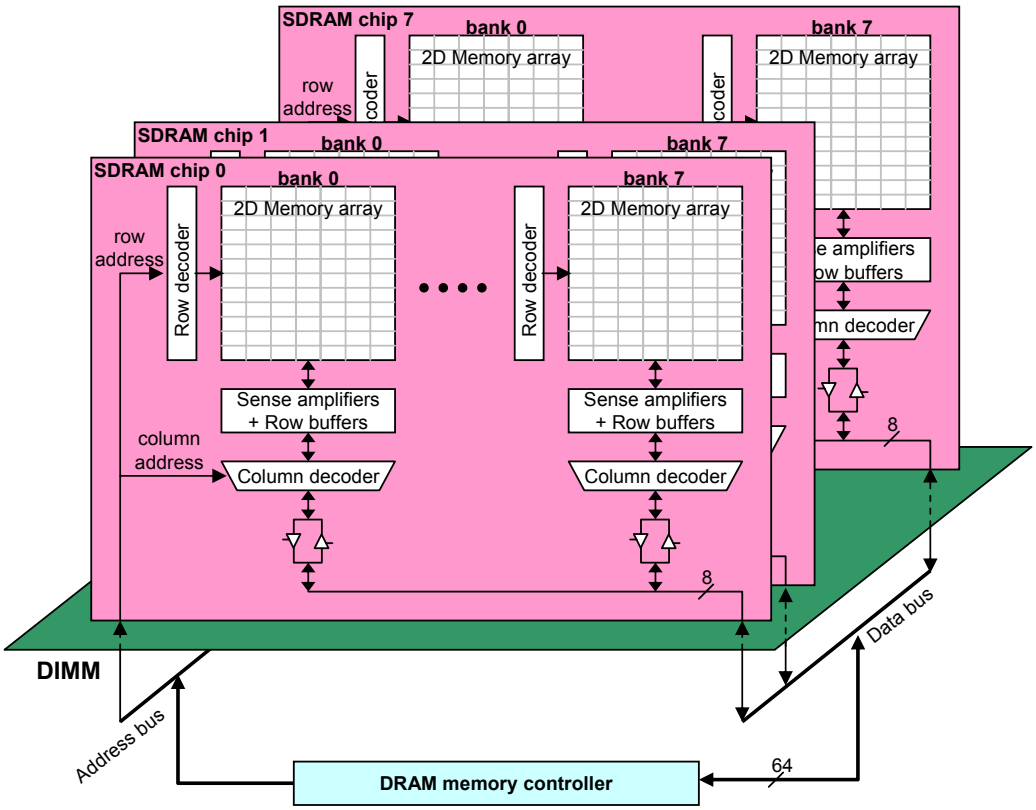


Figure 2-5: A high-level organization of a modern DRAM system (modified from a figure in [67]).

.

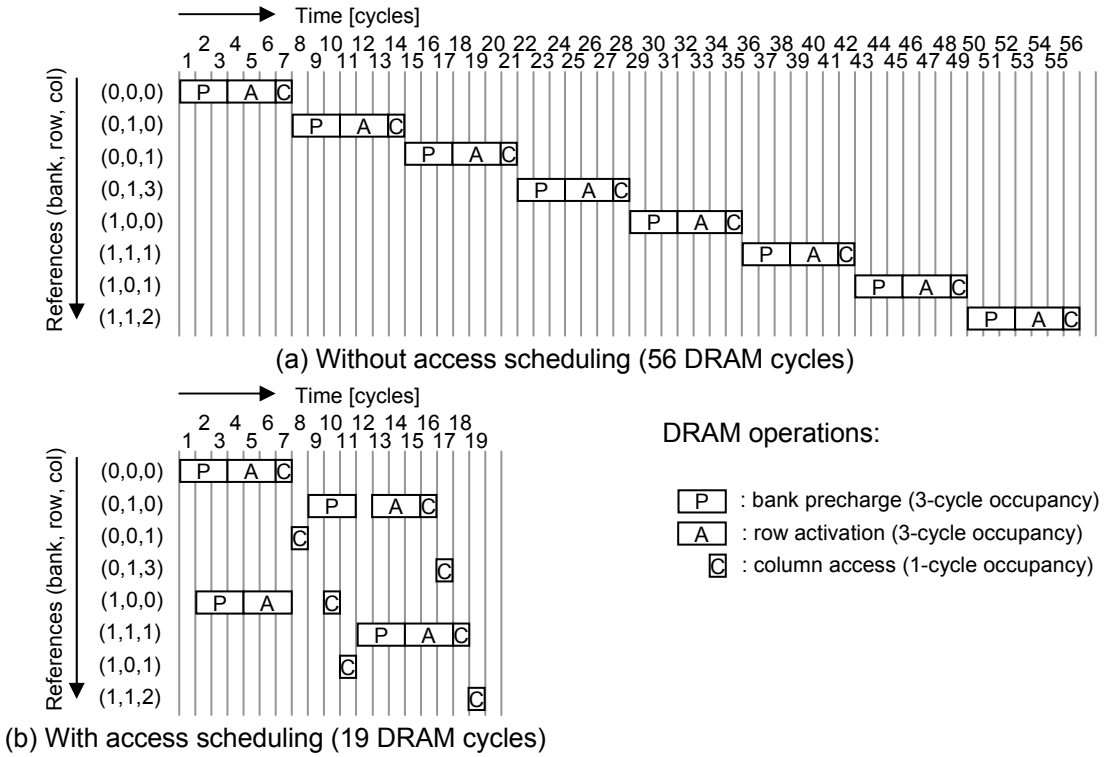


Figure 2-6: Benefits of memory access scheduling (an example taken from [81]). It takes 56 cycles to service the 8 memory requests without memory access scheduling, but only 19 cycles with memory access scheduling.

writes to the row can be performed by issuing (column) `Read` and `Write` commands. After completing the available column accesses, a `Precharge` command is issued to disconnect the cells from the sense amplifier and make the bank ready for a subsequent row activation. Consequently, the amount of time it takes to service a memory request depends on the state of the bank it accesses and falls into three categories [67]: row hit (only column access needed), row closed (row activation and column access needed), and row conflict (precharge, row activation and column access needed).

The 3-D nature of DRAMS makes it advantageous to reorder memory operations to exploit the non-uniform access times of the DRAM. When there are multiple memory requests waiting to be serviced, a *memory access scheduler* [81] can select the next request out of the arrival order to maximize memory throughput. Figure 2-6 shows an example of the potential benefits of memory accessing scheduling.

Rixner et. al. propose an FR-FCFS (First-Ready First-Come-First-Served) scheduling

policy that performs well over a wide range of workloads [81]. The scheduling policy prioritizes among ready requests (i.e., requests that do not violate the DRAM's timing constraints): row-hit requests over row-closed or conflict requests, and older requests over younger ones if their row-hit status is equal.

Although the FR-FCFS scheduling improves the memory system throughput significantly, it does not fairly serve concurrent threads for two main reasons [70]. First, FR scheduling favors a stream of requests that have a high row-hit rate because the next request from the stream will be ready before a request from another stream that accesses a different row in the same bank. Second, in FCFS scheduling a memory request can be blocked for a long time by a bursty stream of requests from another thread just because they happen to arrive earlier than the blocked request. In short, high-demand, bursty threads can make other threads starve in accessing the DRAM system.

2.2.3 Fairness Problems in Other Shared Resources

To provide seamless QoS guarantees, all the scheduling/arbitration points should be carefully designed, for the memory system's guaranteed service bandwidth is determined by the weakest guarantee for any of its shared components. Some of the components have received much attention, such as caches [35, 71, 87], memory controllers [67, 68, 70, 80] and interconnects [10, 29, 53, 95], but others have not, such as directory controllers for on-chip cache coherence and network interfaces interconnecting an on-chip network with distributed caches or memory controllers. In most cases, directories and network interfaces service requests in FIFO (or FCFS) order and are potentially exposed to a similar unfairness problem as in an (FR-)FCFS memory controller. Although it is not clear whether these components can cause a QoS breakdown, it takes more than the abutment of QoS-capable building blocks to build a complete QoS-capable memory system. Also, there is a question about what kind of memory system-wide QoS is provided if QoS-capable components implementing different types of QoS are composed (e.g., fair bandwidth distribution versus fair slowdown).

2.3 Priority-Based versus Frame-Based Scheduling

QoS research has a rich history in computer science. While investigated for decades in other subfields such as computer networking and real-time system, QoS has only recently become an active area of research in computer architecture. Understanding successful solutions proposed in other subfields can give us valuable insights to help devise appropriate QoS mechanisms for future CMP platforms.

In this section and the next, we divide the solution space for QoS into four quadrants according to two orthogonal criteria: scheduling granularity and extent of QoS. The approaches we discuss are independent of resource types and can be applied to any scheduling point. We present just enough details of each class of solutions to understand its pros and cons; we do not mean to provide a comprehensive survey.

The first axis we introduce to divide the solution space is scheduling granularity. Each scheduling point could take either a *(sorted) priority-based* approach or a *frame-based* approach [86]. A typical priority-based scheduler, as shown in Figure 2-7 (a), has per-flow queues. (Again, A flow is loosely defined as a distinct sequence of packets/requests between a single source and a single destination.) Competing requests' priorities are computed based on their classes, arrival time and the requesting flow's bandwidth share, and then compared to determine which request to service next. In contrast, a frame-based scheduler, as shown in Figure 2-7 (b), groups a fixed number of time slots into a frame and controls the bandwidth allocation by giving a certain number of time slots per frame to each requester. The frame-based scheduler has per-frame queues instead of per-flow queues and services requests from the lowest frame number (i.e., earliest time) first. Frame-based scheduling does not preclude multiple virtual channels per frame to prevent head-of-line (HOL) blocking among flows with different destinations that belong to the same frame. Alternatively, we discuss an idea of pooling frame buffers to improve buffer utilization without breaking QoS guarantees in Section 3.2.4.

Generally, frame-based schedulers use less hardware than priority-based schedulers, so they are more suitable for QoS in an on-chip environment. In addition to simplified logic for the scheduler, the number of frames can be made much smaller than the number of flows,

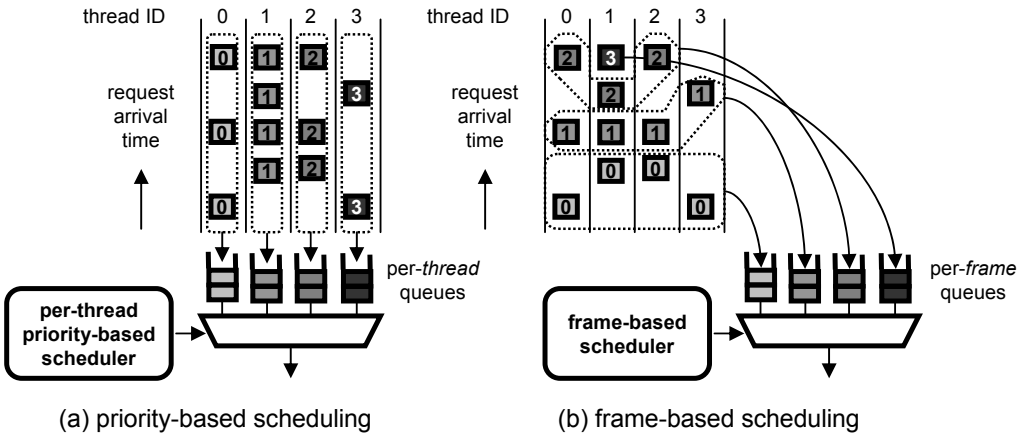


Figure 2-7: Priority-based versus frame-based queueing and scheduling. Each request is annotated with the queue number it is associated with.

which effectively reduces the number of queues needed. The downside of frame-based scheduling is a larger delay bound because frame-based scheduling typically enforces an ordering *across* frames but not *within* a frame. The service order within a frame is usually just FIFO. Therefore, a frame number can be regarded as a coarse-grain priority.

2.3.1 Priority-Based Scheduling Algorithms

We will explain priority-based scheduling approaches with one of the most well-known QoS scheduling algorithms: Fair Queueing [21]. A Fair Queueing (FQ) system requires per-flow queues as shown in Figure 2-8. Conceptually, FQ is like round-robin scheduling amongst queues, except with special care to handle variable packet sizes [6]. The problem with simple round-robin amongst all backlogged queues (i.e., non-empty queues) is that variable packet sizes result in uneven bandwidth distribution; flows with smaller packets receive less service bandwidth.

A FQ algorithm approximates a bit-by-bit round robin (BR). In the BR algorithm, one bit from each flow is forwarded through the output link in a round-robin fashion to fairly distribute the link bandwidth amongst flows. Of course, in real life, packets are not pre-emptible, i.e., once you start sending a packet, you need to send the whole packet out before sending another. BR scheduling is an idealized model that the FQ algorithm emulates.

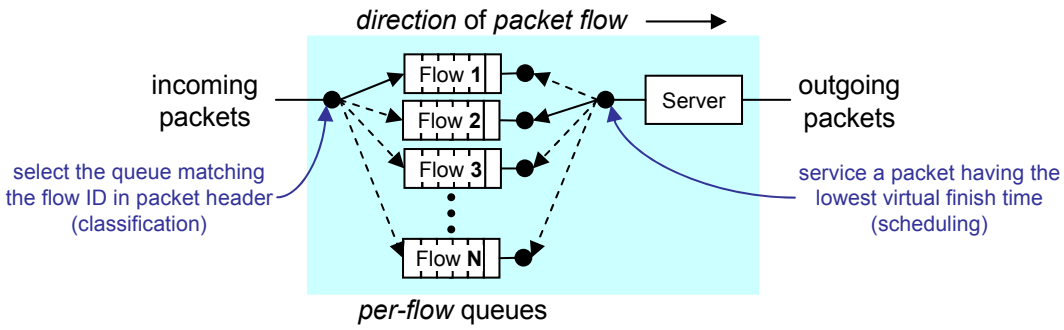


Figure 2-8: A network node implementing Fair Queueing (FQ) algorithm [21].

In the BR algorithm, one complete cycle sending one bit from each of the backlogged queues is called a *round*. Let $R(t)$ be the round number at time t. Suppose the i-th packet of Flow j whose length is p_i^j arrives at time t. Then we can calculate the round number S_i^j in which the packet reaches the head of the queue for Flow j, called the *virtual start time*, as follows:

$$S_i^j = max[R(t), F_{i-1}^j]$$

where F_{i-1}^j is the round number in which the last bit of the previous packet of Flow j is forwarded through the output link, called the *virtual finish time* of the packet. The virtual finish time of the i-th packet of Flow j is given as follows:

$$F_i^j = S_i^j + p_i^j$$

The packet-level emulation of the BR algorithm is realized by sending the packet which has the earliest virtual finish time, which is the FQ algorithm in [21]. This algorithm is also known as Packet Generalized Processor Sharing (PGPS) [76]. This emulation is probably the most intuitive one, but not the only possible one. For example, another version of FQ prioritizes the packet having the earliest virtual start time [8].

There are many priority-based algorithms that share the basic idea of FQ for bandwidth control. Virtual Clock [98] simplifies priority calculations by using the wall-clock time instead of the virtual start time, which unfairly favors bursty traffic over regular traffic. Stochastic Fairness Queueing (SFQ) [61] is a probabilistic variant of FQ and uses a hash to map a source-destination address pair to its corresponding queue, thereby eliminating

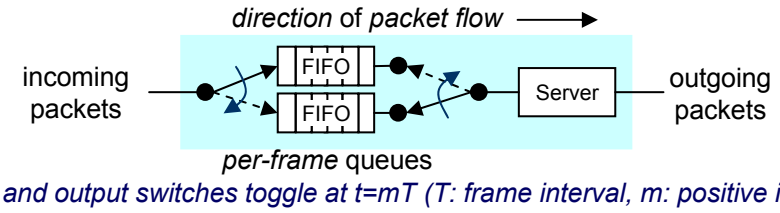


Figure 2-9: A network node implementing Stop-and-Go scheduling with a double-FIFO structure [27].

per-flow queues. Self-Clocked Fair Queueing [28] uses the virtual time (round number) of the currently served flow as system virtual time to simplify the calculation of virtual time.

These priority-based schemes generally require per-flow queues and a tree of comparators to select the highest-priority packet for each packet transmission [42]. This hardware complexity is a significant drawback of the priority-based scheduling algorithms when considering the severe resource constraints in an on-chip environment.

2.3.2 Frame-Based Scheduling Algorithms

In frame-based approaches, time is coarsely quantized into frames. A certain number of packet transmission slots in each frame interval are allocated to each flow in proportion to reserved bandwidth. Unlike strict time-division multiplexing (TDM) where each packet slot is either used by a specific flow or wasted, most of the frame-based algorithms relax flows to use any available slots rather than fixed slots and/or allow frame sizes to vary depending on traffic to improve network utilization [42]. Unlike priority-based algorithms, frame-based algorithms require neither per-flow queues nor a comparator tree to select the highest-priority packet.

Stop-and-Go (SG) scheduling [27] is one of the simplest examples of frame-based scheduling. During each frame, the source of each flow is strictly limited to inject at most as many packets as reserved packet slots. Each switch in the network enforces the following simple rule: transmission of a packet which has arrived at any node during a frame f should always be postponed until the beginning of next frame $(f+1)$. Figure 2-9 illustrates a realization of SG with a double-FIFO structure. During each frame one FIFO queue is filled, while the other is serviced. At the end of each frame the order is reversed. This

arrangement ensures that packets received in a particular frame are not served during that same frame. Unless the total number of reserved packet slots is more than the link's service capacity per frame, requested bandwidth is guaranteed. The network delay of a flow is bounded by $2 \times H \times F$ where H is the number of hops and F is the frame interval. Note that the SG algorithm is *non-work-conserving*; that is, an output link can remain idle even if there are packets waiting to be serviced.

One of the most popular frame-based approaches is Deficit Round Robin (DRR) [84]. In the DRR algorithm, each queue has a deficit counter associated with it. The counter increments at a rate equal to the fair rate of that queue and decrements by the size of each transmitted packet. For each turn, backlogged queues are served in a round-robin manner and a flow can transmit packets as long as its counter is larger than the packet size at the head of its queue. Although this scheme, as described, is not work-conserving, DRR can be made work-conserving by modifying the scheme to allow the counter to decrement to a small negative value [6].

Frame-based algorithms share a common limitation—the coupling between the delay and the granularity of bandwidth allocation. A larger frame size leads to a finer granularity of bandwidth allocation, but increases delay bounds proportionally. However, frame-based algorithms can be implemented using less hardware than priority-based algorithms, which makes them more attractive in an on-chip environment.

2.4 Component-Wise versus End-to-End QoS

The second criterion to classify possible approaches for QoS is the extent of QoS. The extent of a QoS mechanism determines both how much of the system is covered by the QoS mechanism and what type of QoS guarantees are made between end-points. In *component-wise* QoS approaches, a QoS mechanism is contained within each resource or scheduling node, which we call a component; QoS operations are completely defined at a *component* level and QoS guarantees typically made at the same level. Then a global end-to-end QoS is guaranteed by construction. For certain types of QoS (e.g., minimum bandwidth guarantees), if every scheduling point associated with a component in a system guarantees one of

these types of QoS, it is reasonable to expect a thread in the system to receive the same type of guaranteed QoS. In networking terms, component-wise QoS only specifies "per-hop behaviors" (PHBs), which define the behavior of individual routers rather than end-to-end services [78]. If there is any node that does not enforce the same PHBs in a network, it is difficult to predict the end-to-end QoS properties of the network.

Alternatively, the operations of a QoS mechanism can be defined between end-points across multiple components, and its QoS properties described as such—we call this class of approaches *end-to-end* QoS. One good example is age-based arbitration [20, 77]. In one implementation, each request carries a timestamp issued when the request first gets injected into the system and the request with the earliest timestamp wins in any arbitration step at downstream nodes [20]. This global age-based arbitration provides end-to-end QoS properties such as fair bandwidth sharing in steady state and reduced standard deviation of network delay [19].

Although there has been significant earlier work on providing QoS in multicore platforms, researchers have mostly taken component-wise QoS approaches and focused on designing isolated QoS-capable building blocks such as caches [35, 71, 87], memory controllers [67, 68, 70, 80] and interconnects [10, 29, 53, 95]. There is a significant gap between QoS for one component and end-to-end QoS across the entire shared memory system. For example, a system with a QoS-capable memory controller will fail to provide fair memory access bandwidth to different threads if the on-chip network provides only locally fair scheduling, as shown in Figure 2-10. Although the memory controller is capable of QoS, the system cannot provide end-to-end memory bandwidth QoS because the on-chip network does not provide a matching guarantee to transport requests to the hotspot memory controller.

In general, it is difficult to compose separately designed QoS blocks to provide end-to-end QoS, as they often provide very different service guarantees or accounting mechanisms (e.g., fair bandwidth distribution vs. fair slowdown). On the other hand, an end-to-end approach can be extended to multiple resources such as on-chip networks, memory controllers, and caches, to make it easy to reason about the system-wide QoS properties by enforcing the same type of QoS. In addition, end-to-end approaches can potentially reduce

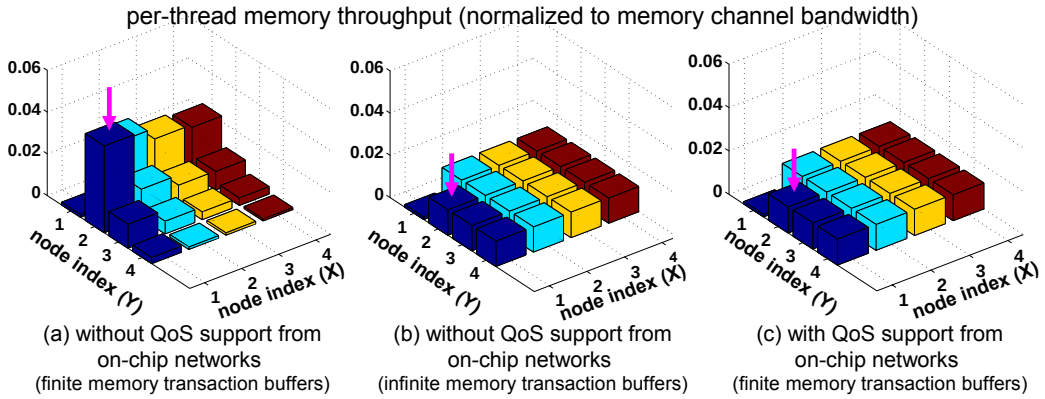


Figure 2-10: QoS-capable memory controllers and fair networks are not sufficient to provide memory system QoS. In this hotspot example, all sources except (0, 0) which is reserved for the OS, generate memory requests targeting a DRAM bank in the QoS-capable memory controller located at (1, 2). With an on-chip network implementing locally fair arbitration (a), a source's service level depends on distance from the controller. Adding infinite queues at the memory controller (b) would provide a globally fair guarantee. An end-to-end QoS system based on GSFM (c) can provide fair sharing with finite queues at the memory controller.

hardware required for QoS scheduling, which makes them more suitable for an on-chip implementation. For example, we compare a component-wise approach (Fair Queueing) and an end-to-end approach (age-based arbitration) in Figure 2-11. The most notable difference is that each component, or scheduling node, in FQ calculates a packet's priority (i.e., virtual finish time) locally, whereas the age-based arbitration scheme effectively offloads the priority assignment stage from each scheduling point to the injection point.

There are challenges that might outweigh the benefits that an end-to-end approach brings. First, it is not easy to maintain the validity of an assigned priority across the entire system. For example, the aforementioned age-based arbitration requires synchronization of all the nodes in the system. Second, some end-to-end approaches are vulnerable to ill-behaved flows. In age-based arbitration, a flow's injection control can manipulate timestamps to gain an unfair advantage over other flows. Consequently, end-to-end approaches are not feasible in a large-scale system, let alone across multiple administrative domains.

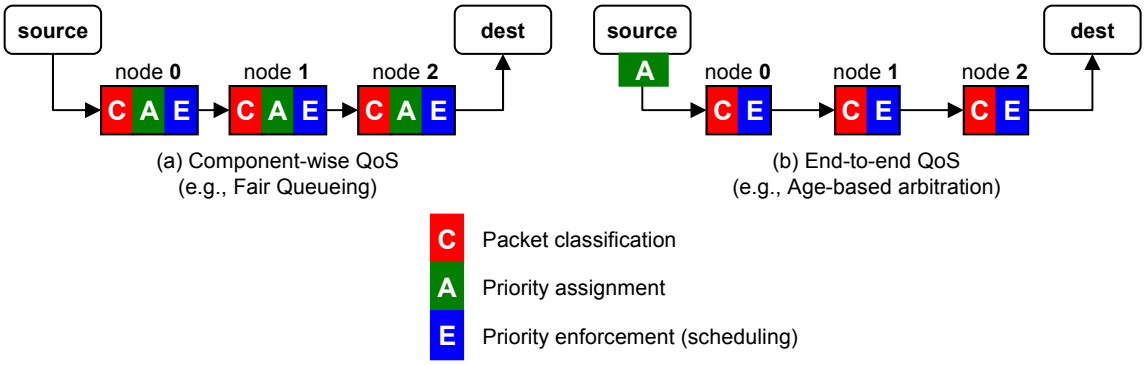


Figure 2-11: Component-wise versus end-to-end approaches for QoS with three major operations at each node: packet classification (C), priority assignment (A) and priority enforcement (E).

2.5 A Survey of Proposals for Network QoS

This section briefly surveys proposals for QoS in the context of on-chip and off-chip networks. We characterize each proposal under the following three criteria:

- **Context** identifies the context in which the proposal was originally developed. The context can be IP/ATM (network diameter: up to thousands of kilometers), multi-chip multiprocessors (network diameter: a few meters – a few kilometers), or on-chip (network diameter: up to a few centimeters) network [79].

- **Scheduling granularity** specifies the granularity of priorities assigned to packets. It can be either priority-based or frame-based as explained in Section 2.3.

- **Extent of QoS mechanism** classifies a proposal to be either component-wise or end-to-end QoS with respect to whether the proposed mechanism is completely contained within a node or not. Both approaches were discussed in detail in Section 2.4.

(Weighted) Fair Queueing [21] / **Virtual Clock** [98] (IP network, priority-based, component-wise). They are developed for QoS in long-haul IP networks where large buffers are available. These achieve fairness and high network utilization, but each router is required to maintain per-flow state and queues which would be impractical in an on-chip network.

Multi-rate channel switching [92] (ATM network, frame-based, component-wise). The source rate of a flow is fixed at an integer multiple of a basic rate before the source starts transmission and remains unchanged for the duration of the flow. Because of the fixed rate, it cannot claim unused bandwidth efficiently, which leads to low network utilization.

Source throttling [90] (multi-chip multiprocessor network, N/A, end-to-end). It dynamically adjusts the traffic injection rate at a source node primarily for congestion control. This keeps the network from suffering overall throughput degradation beyond the saturation point, but does not provide QoS to individual flows.

Age-based arbitration [1, 19, 20, 77] (multi-chip multiprocessor network, priority-based, end-to-end). It is known to provide strong global fairness in bandwidth usage in steady state and reduce standard deviation of network delay. Each packet (or flit) carries information to indicate its age, either a counter updated at every hop [77], or a timestamp issued when the packet first enters the network from which age can be calculated by subtracting from the current time [20]. The oldest packet wins in any arbitration step. This approach lacks flexibility in bandwidth allocation because it does not allow for asymmetric resource allocation, and requires sophisticated logic to handle aging, arbitration and counter rollover.

Rotating Combined Queueing (RCQ) [43] (multi-chip multiprocessor network, priority-based, component-wise). It is designed for a multiprocessor and provides predictable delay bounds and bandwidth guarantees without per-flow queues at intermediate nodes (though it still maintains per-flow statistics).

Each packet in a RCQ network is assigned a local frame number using per-flow statistics upon arrival at every node on the path. The idea of rotating priorities in a set of queues is similar to GSF, but GSF further simplifies the router using global information, which is only feasible in an on-chip environment. Unlike RCQ, the frame number in the GSF is global, which eliminates expensive book-keeping logic and storage at each node.

MetaNet [74] (ATM network, frame-based, component-wise). It is an interesting proposal to implement an frame-based end-to-end QoS mechanism in an ATM network. Two independent networks servicing constant bit rate (CBR) and variable bit rate (VBR) traffic respectively, share physical links to improve the link utilization. Each source is allowed to

inject up to a certain number of packets per frame into the CBR network and the rest of packets are injected into the VBR network. Packets carry a frame number, which is used at a destination to reorder packets taking different paths. The CBR network provides QoS guarantees in terms of minimum bandwidth and maximum delay.

The core idea of MetaNet is to create a distributed global clock (DGC) across the entire network using a distributed clocking protocol. They claim that a DGC running at 1 MHz is feasible for a 100-node system with realistic network parameters, which gives enough resolution for their frame-based QoS. However, the proposal has drawbacks. First, each router should detect malicious changes of the global clock values to police network traffic. Second, the DGCs are only frequency locked, and not phase locked and the link propagation delay on the link is not an integer number of time frames. Therefore, partitioning of the incoming stream of packets into frames should be done carefully to achieve deterministic QoS guarantees. Third, the destination requires a large reorder buffer because traffic is decomposed into the two networks and merged at the destination.

MediaWorm Router [96] (multi-chip multiprocessor network, priority-based, component-wise). The MediaWorm router aims to support multimedia traffic in a multiprocessor cluster, including constant bitrate (CBR), variable bitrate (VBR), as well as best-effort traffic. It uses Fair Queueing [21] and Virtual Clock [98] for packet scheduling, which require to maintain expensive per-flow state and queues.

Æthereal [29] (on-chip network, frame-based, component-wise). It uses pipelined time-division-multiplexed (TDM) circuit switching to implement guaranteed performance services. Each QoS flow is required to explicitly set up a channel on the routing path before transmitting the first payload packet, and a flow cannot use more than its guaranteed bandwidth share even if the network is underutilized. To mitigate this problem, *Æ*thereal adds a best-effort network using separate queues, but this introduces ordering issues between the QoS and best-effort flows.

SonicsMX [95] (on-chip network, frame-based, component-wise). It supports guaranteed bandwidth QoS without explicit channel setup. However, each node has to maintain per-thread queues, which make it only suitable for a small number of threads (or having multiple sources share a single queue).

Nostrum NoC [64] (on-chip network, frame-based, component-wise). It employs a variant of TDM using virtual circuits for allocating bandwidth. The virtual circuits are set up semi-statically across routes fixed at design time, and only the bandwidth is variable at runtime, which is only suitable for application-specific SoCs.

MANGO clockless NoC [11] (on-chip network, frame-based, component-wise). It partitions virtual channels (VCs) into two classes: Guaranteed Service (GS) and Best-Effort (BE). A flow reserves a sequence of GS VCs along its path for its lifetime. Therefore, the number of concurrent GS flows sharing a physical channel is limited by the number of GS VCs (e.g., 8 in [11]).

Clockless NoC by Felicijan et al. [24] (on-chip network, priority-based, component-wise). It provides differentiated services by prioritizing VCs. Though this approach delivers improved latency for certain flows, no hard guarantees are provided.

QNoC [13] (on-chip network, N/A, end-to-end). Its goal is to provide ”fair” services in accessing a hotspot resource, and it takes a source regulation approach. The QNoC approach requires each source to fetch credit tokens from the destination (hotspot) node before sending out payload packets. It requires only minimal modifications to network routers because most of the intelligence is at end nodes.

However, there are several issues with their approach. First, it does not provide guaranteed QoS. In their approach, the credit recharge is basically asynchronous across all the sources and it is not clear what type of QoS is guaranteed. Second, it requires a sophisticated secondary network (either logical or physical) for credit token request/reply not to slow down the source injection process. Finally, it is more suitable for application-specific SoCs than general-purpose platforms. To set up (and tear down) a QoS flow, the source needs to explicitly request it to the hotspot destination before sending out the first payload packet. It is acceptable in application-specific SoCs where communication patterns are known a priori and do not change over time, but not in general-purpose platforms because it penalizes short-lived bursty flows.

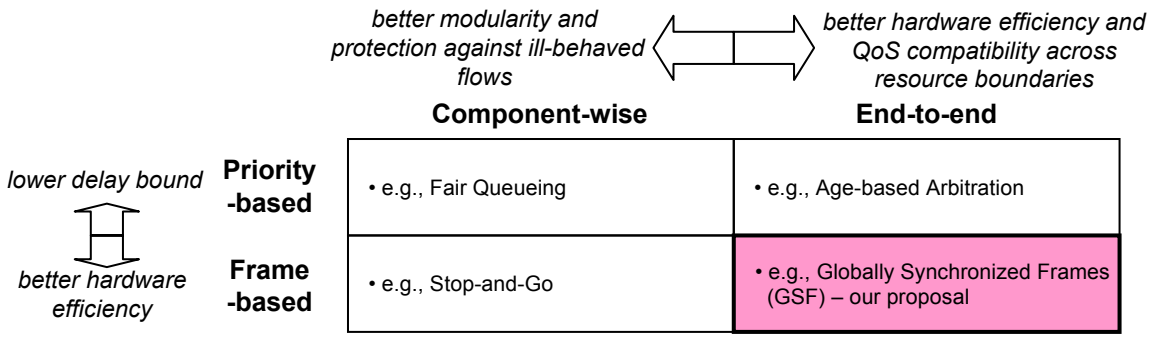


Figure 2-12: Four quadrants of the solution space for QoS.

2.6 Arguments for Frame-Based, End-to-End Approaches

Among the four quadrants of the solution space for QoS in Figure 2-12, we argue for *frame-based*, *end-to-end* approaches for QoS in the resource-constrained on-chip environment. We make separate arguments for each axis in the figure.

Arguments for frame-based approaches: There are two main arguments for frame-based approaches over priority-based approaches in future CMPs. First, future CMP designs will require high energy efficiency from all of their components, and per-flow queues and complex priority enforcement logic will be hard to scale to the number of CPU cores while justifying their hardware cost. Second, unlike off-chip network systems, the memory system of a CMP should provide a very low average memory access time (in the order of tens to hundreds of processor cycles), and hence it cannot afford a heavy-weight QoS algorithm for relatively marginal improvement in the worst-case network latency bound. In short, frame-based approaches are more energy-efficient, scalable and light-weight and better meet the needs of future manycore CMPs.

Arguments for end-to-end approaches: Although end-to-end approaches can potentially reduce the hardware complexity compared to component-wise approaches, they have not been widely deployed in an off-chip environment because of two main challenges: system-wide synchronization and handling of ill-behaved flows. In an on-chip environment, a cheap synchronization among all nodes is feasible to enable the global coordination necessary for an end-to-end approach. It could be achieved through an explicit synchronization (e.g., barrier) network or a system-wide synchronous clock domain. *Policing*

ill-behaved flows is not an issue because the injection control hardware can be under complete control of the chip designer and an OS (or hypervisor) can enforce a strict admission control policy not to oversubscribe any resource in the memory access path.

We believe that frame-based, end-to-end approaches are a promising yet less explored direction towards efficient and robust QoS in the shared memory system of a future many-core CMP. This thesis proposes a new frame-based, end-to-end approach for cache-coherent shared memory systems.

Chapter 3

GSF On-Chip Networks: One-Dimensional GSF

In this chapter we introduce a new frame-based end-to-end QoS framework, called *Globally-Synchronized Frames* (GSF), to provide guaranteed QoS in multi-hop on-chip networks in terms of minimum bandwidth and maximum delay bound. Although there has been significant prior work in QoS support for other building blocks of a CMP memory system, such as memory controllers [67, 68, 70, 80] and shared cache banks [35, 71, 87], there has been less work on QoS support for multi-hop on-chip networks in general-purpose platforms. Without QoS support from an on-chip network, robust memory system-wide QoS is not possible because it can be the weakest link which limits the overall guaranteed service level. For example, allocating a portion of off-chip memory bandwidth at a memory controller is ineffective if the on-chip network does not guarantee adequate bandwidth to transport memory requests and responses. Even in a case where the on-chip network is not a bandwidth bottleneck, tree saturation [90] can produce a tree of waiting packets that fan out from a hotspot resource, thereby penalizing remote nodes in delivering requests to the arbitration point for the hotspot resource.

The GSF scheme introduced in this chapter is *one dimensional* because we deal with only one bandwidth resource: link bandwidth in a single network. In Chapter 4, we will extend it to handle multiple resources in a single unified framework: multiple networks, DRAM channels and DRAM banks. This unified framework is called *multi-dimensional*

GSF.

3.1 Introduction

As illustrated in Figure 2-4, conventional best-effort on-chip networks perform poorly without QoS support when congested. Although the computer networking community has well addressed QoS problems in large-scale IP or multi-chip multiprocessor networks for decades, their solutions cannot be directly applied to on-chip networks because of very different performance requirements and technology constraints. We need an efficient solution that can provide robust and flexible QoS for an ever-increasing number of cores on a chip.

GSF provides guaranteed and differentiated bandwidth as well as bounded network delay without significantly increasing the complexity of the on-chip routers. In a GSF system, time is coarsely quantized into "frames" and the system only tracks a few frames into the future to reduce time management costs. To provide service guarantees, GSF combines injection control and earliest-frame-first scheduling at each scheduling point. Each QoS packet from a source is tagged with a frame number indicating the desired time of future delivery to the destination. At any point in time, packets in the earliest extant frame are routed with highest priority but sources are prevented from inserting new packets into this frame. GSF exploits fast on-chip communication by using a global barrier network to determine when all packets in the earliest frame have been delivered, and then advances all sources and routers to the next frame. The next oldest frame now attains highest priority and does not admit any new packets, while resources from the previously oldest frame are recycled to form the new futuremost frame.

Provided that the pattern of injected packets in each frame does not oversubscribe the capacity of any network link, the system can switch frames at a rate that sustains any desired set of differentiated bandwidth flows with a bounded maximum latency. Note that bandwidth and latency are decoupled in this system, as multiple frames can be pipelined through the system giving a maximum latency of several frame switching times. The scheme does not maintain any per-flow information in the routers, which reduces router complexity and also avoids penalizing short-lived flows with a long route configuration step. The scheme

supports bursty traffic, and allows best-effort traffic to be simultaneously supported with little loss of network utilization compared to a pure best-effort scheme.

3.2 Globally-Synchronized Frames (GSF)

In this section, we present the design of GSF starting from an idealized deadline-based arbitration scheme for bandwidth guarantees. We then transform this scheme step-by-step into an implementable GSF queueing and scheduling algorithm.

3.2.1 Global Deadline-Based Arbitration for Bandwidth Guarantees

GSF was originally inspired by deadline-based arbitration, which is a generalization of age-based arbitration [1, 19, 20, 77]. In age-based arbitration, each packet carries a global timestamp, issued when the packet enters the network, and each arbiter (router) forwards the packet with the earliest timestamp first. Instead of using the timestamp, we allow each source to assign a deadline other than the current time according to a deadline assignment policy. Our premise is that we can achieve a desired flow property, including guaranteed minimum bandwidth, by controlling deadline assignment policy, at least in an idealized setup.

Figure 3-1 shows a network from such an idealized setup, where each queue is a perfect priority queue with infinite capacity, capable of instantly dequeuing the packet with the earliest deadline. Dotted rectangles are a network component which Cruz [18] introduced and named "MUX". Since we assume zero-cycle delay for arbitration and queue bypassing, the entire network conceptually reduces to a single priority queue having four input ports and one output port, with a total ordering among all packets according to their deadlines.

To provide bandwidth guarantees, we assign the deadline for the n-th packet of Flow i (d_i^n) as follows:

$$\boxed{d_i^n(\rho_i) = MAX[current\_time, d_i^{n-1}] + L_i^n/(\rho_i C)}$$

where ρ_i is the guaranteed minimum bandwidth of Flow i represented as a fraction of channel bandwidth C ($0 \leq \rho_i \leq 1$) and L_i^n is the length of the n-th packet of Flow i.

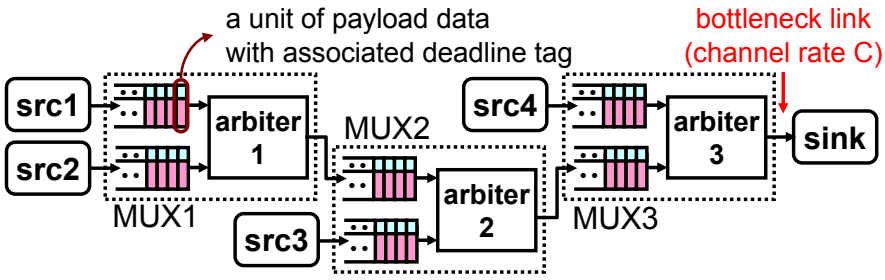
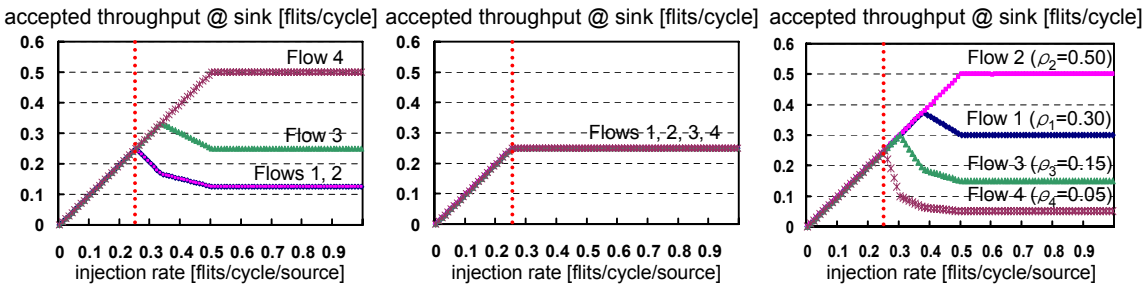


Figure 3-1: A three-node queueing network with perfect priority queues with infinite capacity. Dotted rectangles are a network component called "MUX", which merges two incoming flows into a single outgoing one. Each packet carries an associated deadline and the priority queue can dequeue the packet having the earliest deadline. The arbiter and the priority queue are assumed to have zero-cycle delay, which implies a packet just generated at any source can be immediately forwarded to the sink at channel rate C through the combinational path if the packet wins all arbitrations on the path.

This formula directly follows from what is known as the *virtual finish time* in the Fair Queueing algorithm [21]. The deadline specifies the time when a packet's last bit arrives at the destination if the channel were infinitely divisible and shared by multiple packets simultaneously transmitting according to their guaranteed shares (ρ's), provided we ignore the network traversal delay, which is dependent upon each flow's distance from the destination. Figure 3-2 compares three arbitration schemes: round-robin, age-based and deadline-based with the deadline assignment policy presented above. Note that the deadline-based scheme provides bandwidth distribution to all flows proportional to the ratio of ρ's at all congested links. This result holds even if we have non-zero delay for arbitration and/or queue bypassing as long as the priority queue has an infinite capacity. In such a case, two flows sharing a congested link eventually enter into the steady state of proportional bandwidth sharing after a finite winning (losing) streak by the remote (local) node starting when the two flows first meet at the congested link. The length of the winning (losing) streak is determined by the relative difference of the distance to the congested link.

Although deadline-based arbitration provides minimum bandwidth guarantees to flows using the proposed policy in the idealized setup, there are several issues that make this scheme infeasible to implement. First, the scheme is based on perfect priority (sorting) queues and infinite-sized buffers. Second, there is a large overhead for sending and storing



(a) round-robin (RR) arbitration (b) age-based arbitration (c) deadline-based arbitration

Figure 3-2: Per-source accepted throughput at the sink with various arbitration schemes in the network shown in Figure 3-1. Dotted vertical lines indicate minimum injection rate causing congestion. In locally-fair round-robin arbitration in (a), which does not use the deadline field, the throughput of a flow decreases exponentially as the number of hops increases. Age-based arbitration in (b), where deadline is assigned as network injection time, gives fair bandwidth allocation among all flows. With deadline-based arbitration with the policy for bandwidth guarantees in (c), we achieve bandwidth distribution proportional to the ratio of ρ's ($\rho_1 : \rho_2 : \rho_3 : \rho_4 = 0.30 : 0.50 : 0.15 : 0.05$) in face of congestion.

the deadline along with the payload data. Therefore, we propose a practical implementation approximating the behavior of the ideal deadline-based arbitration, called *baseline GSF*.

3.2.2 Baseline GSF

To make deadline-based arbitration practical, we adopt a frame-based approach [97]. The original deadline-based arbitration is a priority-based approach, where competing packets' priorities are compared to determine which packet is allocated buffer space and switching bandwidth first. In contrast, a frame-based approach groups a fixed number of time slots into a frame and controls the bandwidth allocation by allocating a certain number of flit injection slots per frame to each flow.

Figure 3-3 shows a step-by-step transformation towards a frame-based, approximate implementation of deadline-based arbitration. Frame k is associated with packets whose deadline is in the range of $[kF+1, (k+1)F]$, where F is the number of flit times per frame. The frame number k is used as a coarse-grain deadline. By introducing frames, we enforce an ordering *across* frames but not *within* a frame because the service order within a frame is simply FIFO. The baseline GSF arbitration is shown in Figure 3-3(c), where we have a

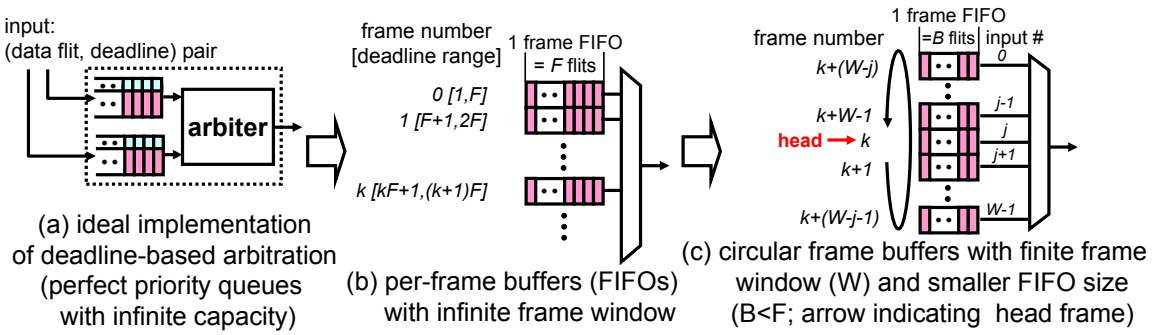


Figure 3-3: Step-by-step transformation towards a frame-based, approximate implementation of deadline-based arbitration. (a) shows the ideal MUX introduced in Section 3.2.1, which is associated with each physical link. In (b), we group all data entries having a range of deadlines (whose interval is F) to form a *frame*. The frame number is used as a coarse-grain deadline, and we now drop the deadline (or frame number) field because each frame has an associated frame buffer. Finally, we recycle frame buffers, with W active frames, as in (c) to give a finite total buffer size.

finite active frame window having W frames (i.e., Frame k through $(k+W-1)$) and each active frame has a dedicated frame buffer (FIFO) whose depth is B flits. The head pointer indicates which frame buffer is currently bound to the earliest frame (frame buffer j in the figure), which we call the *head frame*. Note that the baseline GSF in Figure 3-3(c) is an asymptotic implementation of the ideal deadline-based arbitration in Figure 3-3(a) because the former reduces to the latter as $W \rightarrow \infty$ and $F \rightarrow 1$.

Here is a brief sketch of how the GSF network operates. For each active frame, every flow is allowed to inject a certain number of flits, denoted by R_i for Flow i. Although our scheme is similar to conventional frame-based bandwidth allocation schemes, we allow W frames to overlap at any given time to accommodate bursty traffic while preventing an aggressive flow from injecting too much traffic into the network.

Once a packet is injected into the network, it traverses the network using only the frame buffers for its given frame number. Therefore, there is no possibility of priority inversion, where a lower-priority packet blocks a higher-priority packet. Combined with earliest-frame-first scheduling for bandwidth allocation, the head frame is guaranteed to drain in a finite amount of time because only a finite sum of packets can be injected into a single frame by all flows. The drained head frame buffers across the entire network are reclaimed

Variables	Range	Description
Global parameters and variables*		
W	2 .. ∞	active frame window size
HF	0 .. $(W-1)$	current head frame
F	1 .. ∞	frame size in flits
B	1 .. ∞	frame buffer depth in flits
C	(0, 1]	channel bandwidth in flits/cycle
L^{MAX}	1 .. ∞	maximum packet length in flits
$epoch$**	0 .. ∞	current epoch number
e_k	1 .. e^{MAX}	interval of k-th epoch
e^{MAX}	1 .. ∞	maximum epoch interval (= $\max_{\forall k} e_k$)
T^{epoch}	0 .. e^{MAX}	epoch timer
Per-flow variables		
ρ_i	0 .. 1	a fraction of bandwidth allocated to Flow i (normalized to C)
R_i	0 .. F	flit slots reserved for Flow i in a single frame
IF_i	0 .. $(W-1)$	current injection frame of Flow i
C_i	-L^{MAX} .. R_i	available credit tokens for Flow i to inject flits to IF_i

* Global variable with a subscript i denotes a local copy of the variable maintained by Flow i.
** not implemented in hardware

Table 3.1: Variables and parameters used in GSF.

and allocated to a newer frame synchronously, which is called an *(active) frame window shift*.

We define an *epoch* as the period of time between adjacent frame window shifts, and the interval of the k-th epoch (i.e., the period of time when frame k is the header frame) is denoted by e_k. We also define $e^{MAX} \equiv \max_{\forall k} e_k$. Table 3.1 summarizes variables and parameters used in the GSF algorithm. More detailed description of each network component's operation follows.

Packet injection process: Algorithm 1 describes a packet injection algorithm used by the baseline GSF network. Flow i can inject packets into the active frame pointed to by IF_i as long as it has a positive credit balance ($C_i > 0$) for the frame (Lines 2-4). The flow can go overdrawn, which allows it to send a packet whose size is larger than R_i. As a result, if R_i is not divisible by the packet size, Flow i can inject more flits than R_i into a single frame (bounded by $R_i + L^{MAX} - 1$ flits), where L^{MAX} is the maximum packet size. A flow with negative credit balance cannot inject a new packet until its balance becomes positive again. Since a flow that has injected more flits than its guaranteed share (R_i) into

the current injection frame (IF_i) gets penalized in injecting packets in the following frames accordingly, bandwidth guarantees are preserved over multiple time frames. Allowing a negative credit value is our design decision to maximize the network throughput at the cost of an increase in the fairness observation window size.

If the flow has used up all reserved slots in Frame IF_i, it can use reserved slots further in the future by incrementing IF_i by one (mod W) (Lines 5-13) until it hits the tail of the active frame window. Once the flow uses up all reserved slots in the active frame window, it must stall waiting for a frame window shift to open a new future frame. Note that this injection process is effectively implementing a token-bucket filter [18] with (ρ, σ) = (R_i/e^{MAX}, $R_i * W$) assuming the active frame window shifts at every e^{MAX} cycles.

Algorithm 1 GSF packet injection algorithm into source queue for Flow i ($\oplus_W$: modulo W addition)

Initialize: *epoch* $= 0$, $HF_i = HF = 0$
Initialize: $R_i = C_i = \lfloor \rho_i F \rfloor$
Initialize: $IF_i = 1$
1: AT EVERY PACKET GENERATION EVENT:
2: **if** $C_i > 0$ **then**
3: $SourceQueue_i.enq(packet, IF_i)$
4: $C_i = C_i - packet.size()$
5: **else** {used up all reserved slots in Frame IF_i}
6: **while** $(IF_i \oplus_W 1) \neq HF_i$ and $C_i < 0$ **do**
7: $C_i = C_i + R_i$
8: $IF_i = IF_i \oplus_W 1$
9: **end while**
10: **if** $C_i > 0$ **then**
11: $SourceQueue_i.enq(packet, IF_i)$
12: $C_i = C_i - packet.size()$
13: **end if**
14: **end if**

Switching bandwidth and buffer allocation: Frame buffer allocation is simple because every packet is assigned a frame at the source, which determines a sequence of frame buffers to be used by the packet along the path. There can be contention between packets within a frame but not across frames. In allocating switching bandwidth, we give the highest priority to the earliest frame in the window.

Frame window shifting algorithm: Algorithm 2 shows an algorithm used to shift the active frame window. Source injection control combined with earliest-frame first scheduling yields a finite drain time for the head frame, bounded by e^{MAX}. Therefore, we shift the

active frame window at every e^{MAX} cycles by default. Every flow maintains a local copy (T_i^{epoch}) of the global epoch timer (T^{epoch}) and decrements it at every clock tick (Lines 9-10). Once the timer reaches zero, all the flows synchronously increment the head frame pointer HF_i (mod W) to reclaim the frame buffer associated with the earliest frame.

The frame window shifting algorithm does not allow a flow to inject a new packet into the head frame (Lines 4-7). Otherwise, we would have a very loose bound on the worst-case drain time of the head frame (e^{MAX}), which would degrade network throughput.

Algorithm 2 GSF frame window shifting algorithm ($\oplus_W$: modulo W addition)

Initialize: $T_i^{epoch} = e^{MAX}$
Initialize: $HF_i = HF = 0$
1: FOR ALL FLOWS, AT EVERY CLOCK TICK:
2: **if** $T_i^{epoch} == 0$ **then**
3: $HF_i = HF_i \oplus_W 1$
4: **if** $HF_i == IF_i$ **then**
5: $IF_i = IF_i \oplus_W 1$
6: $C_i = MIN(R_i, C_i + R_i)$
7: **end if**
8: $T_i^{epoch} = e^{MAX}$
9: **else**
10: $T_i^{epoch} = T_i^{epoch} - 1$
11: **end if**

In effect, the GSF network implements W logical networks sharing physical channels, and each logical network is associated with one frame at any point in time. The W logical networks receive switching bandwidth according to priorities which rotate on every frame window shift. A logical network starts as the lowest-priority logical network when it is just assigned to a new frame, and is promoted throughout the lifetime of the frame to eventually become the highest-priority network, after which it finally gets reclaimed for the new futuremost frame.

The baseline GSF network provides the following guaranteed bandwidth to Flow i if (1) none of the physical channels along the path are overbooked and (2) the source queue ($SourceQueue_i$) has sufficient offered traffic to sustain the reserved bandwidth:

$$\text{Guaranteed bandwidth}_i = R_i / e^{MAX}$$

The proof sketch is simple. Flow i can inject R_i flits into each frame, and the network opens a new frame every e^{MAX} cycles. Because the network does not drop any packets

and has a finite buffer size, the guaranteed bandwidth holds. In addition, the worst-case network delay is bounded by We^{MAX} because a packet injected in k-th epoch must be ejected from the network by the beginning of $(k + W)$-th epoch.

Although the baseline GSF scheme provides guaranteed services in terms of bandwidth and bounded network delay, there are several drawbacks to the scheme. First, frame buffers are underutilized, which degrades overall throughput for a given network cost. Second, it is difficult to bound e^{MAX} tightly, which directly impacts the guaranteed bandwidth. Even with a tight bound, it is too conservative to wait for e^{MAX} cycles every epoch because the head frame usually drains much faster.

To address these two issues without breaking QoS guarantees, we propose two optimization techniques: *carpool lane sharing* and *early reclamation of empty head frames*.

3.2.3 An Alternative Derivation of Baseline GSF

Before presenting techniques to optimize the baseline GSF network, we derive the baseline GSF in a different way. This section can be safely skipped without loss of continuity. In Section 3.2.2, we present the baseline GSF as a degenerate form of deadline-based arbitration. Alternatively, we can obtain the baseline GSF network by augmenting a chip-wide Stop-and-Go system with multiple overlapping frames. (Stop-and-Go scheduling is discussed in Section 2.3.2.)

We begin with a simple chip-wide Stop-and-Go system as follows. A frame is coarse quantization of time, say e^{MAX} cycles. The network can transport a finite number of flits for each flow from its source to destination during this period (R_i flits for Flow i). A chip-wide Stop-and-Go system deploys an injection control mechanism at each source as shown in Figure 3-4 (a) with a double-FIFO structure. During the m-th frame interval, the injection control for Flow i drains the flits that were loaded in during the previous frame interval (Frame $(m-1)$) while loading in up to R_i flits that will be injected during the next frame interval (Frame $(m+1)$). A simple admission control prevents any network link from being oversubscribed. That is, it rejects a new flow if the admission of the flow would make the network unable to transport all the injected flits within e^{MAX} cycles. Figure 3-4

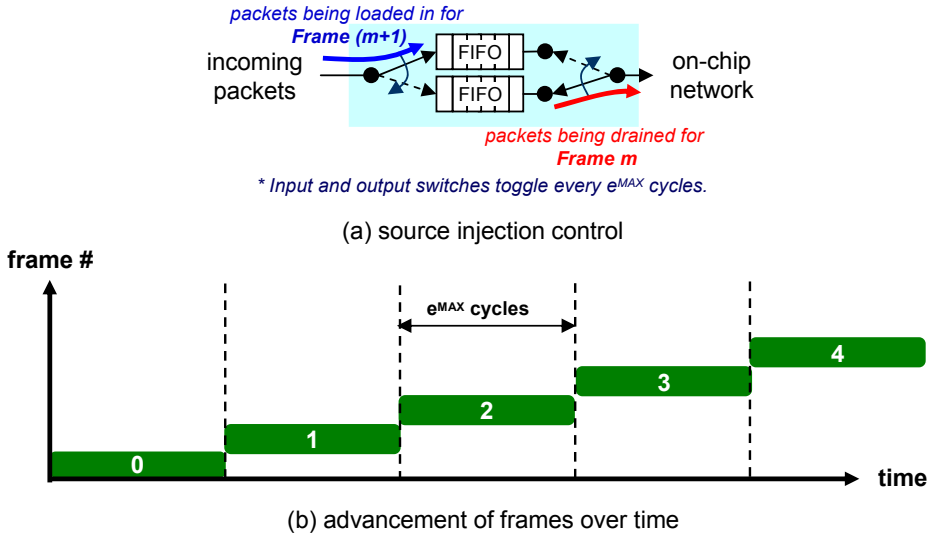


Figure 3-4: A simple chip-wide Stop-and-Go system that provides guaranteed QoS with minimum bandwidth and maximum network delay bound.

(b) shows advancement of frames over time. When a new frame opens, the input and output switches at the injection control toggle, and all the sources start injecting into the network packets loaded in during the previous frame. Bandwidth allocation is controlled by setting R_i parameters appropriately. This simple chip-wide Stop-and-Go system provides QoS guarantees for Flow i with minimum bandwidth of R_i/e^{MAX} and maximum network delay bound of e^{MAX}.

However, the simple chip-wide Stop-and-Go system potentially penalizes bursty traffic in terms of network latency. If more than R_i flits arrive during Frame m, transmission of the extra flits will be postponed until the beginning of Frame $(m+2)$ even if there is unused bandwidth in Frame m and $m+1$. To effectively multiply the flit injection slots of a flow without increasing its bandwidth reservation, we now allow a flow source to inject into the network flits that belong to future frames, and replicate buffers at each node to have separate per-frame buffers. For example, in Figure 3-5 there are three active frames, one current (head) and two future frames, at any point of time. If a flow source drains all flits in Frame 0, it can now inject flits in Frame 1 and 2. Whenever there is a contention for channel bandwidth by multiple flits, older frames always have higher priorities. Note that the drain time of the current frame does not increase because a packet in the current

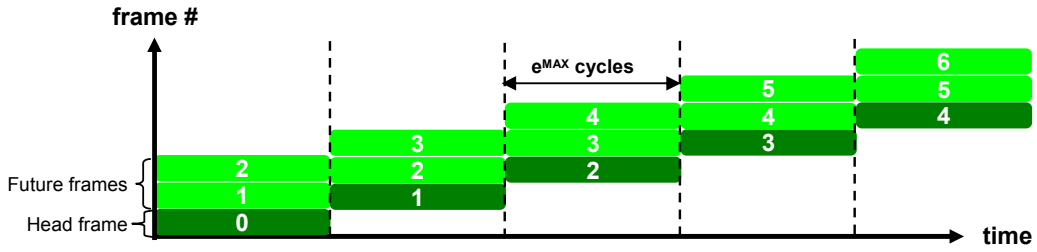


Figure 3-5: Overlapping multiple frames to improve the network utilization. Future frames can use unclaimed bandwidth by the current frame opportunistically.

frame will never be blocked by a lower-priority packet. Future frames can use unclaimed bandwidth by older frames. Another thing to note is that the maximum network delay increases by a factor of three (which is the frame window size) because the lifetime of a frame increases by the same factor.

This chip-wide Stop-and-Go system augmented by multiple overlapping frames effectively implements the baseline GSF as depicted in Figure 3-3 (c). Now we move on to techniques to improve the network utilization of the baseline GSF without degrading QoS guarantees.

3.2.4 Carpool Lane Sharing: Improving Buffer Utilization

One observation in the baseline GSF scheme is that guaranteed throughput does not really depend on the active frame window size, W. The multiple overlapping frames only help claim unused bandwidth to improve network utilization by supporting more bursty traffic. As long as we provide a dedicated frame buffer for the head frame at each router to ensure a reasonable value of e^{MAX}, we do not compromise the bandwidth guarantees.

We propose carpool lane sharing to relax the overly restrictive mapping between frames and frame buffers. Now we reserve only one frame buffer to service the head frame (like a carpool lane), called the head frame buffer, but allow all active frames, including the head frame, to use all the other frame buffers. Each packet carries a frame number (mod W) in its head flit, whose length is $\lceil \log_2 W \rceil$ bits, and the router services the earliest frame first in bandwidth and buffer allocation. The frame window shifting mechanism does not change. Note that head-of-line blocking of the head frame never happens because we map frame

buffers to virtual channels (allocated on a per-packet basis) and the head frame buffer at each router serves as an escape channel for packets that belong to the head frame.

According to our evaluation, the carpool lane sharing scheme increases the overall throughput significantly because more frame buffers are available to a packet at each node. That is, any packet can occupy any frame buffer, except that the head frame buffers are reserved only for packets in the head frame. To support best-effort traffic, we can simply assign a special frame number (W, for example) which represents the lowest priority all the time.

3.2.5 Early Frame Reclamation: Increasing Frame Reclamation Rate

One important factor affecting the overall throughput in the GSF network is the frame window shift rate. According to our analysis, only a small fraction of e_k's ever come close to e^{MAX}, which implies that the head frame buffer is often lying idle waiting for the timer to expire in each epoch.

Therefore, we propose to use a global barrier network to reclaim the empty head frame as quickly as possible. Instead of waiting for e^{MAX} cycles every epoch, we check whether there is any packet in the source or network buffers that belongs to the head frame. If not, we retire the head frame immediately and allocate its associated buffers to the new futuremost frame. Note early reclamation does not break the original bandwidth guarantees, because we always see a net increase, or at worst no change, in available flit injection slots.

Figure 3-6 shows that early reclamation provides over 30% improvement in network throughput in exchange for a small increase in area and power for the barrier network. The barrier network is only a small fraction of the cost of the primary data network, as it uses only a single wire communication tree and minimal logic.

3.3 Implementation

The GSF frame structure fits well into the architecture of a conventional virtual channel (VC) router, requiring only relatively minor modifications. This section discusses the GSF router architecture and the fast on-chip barrier network.

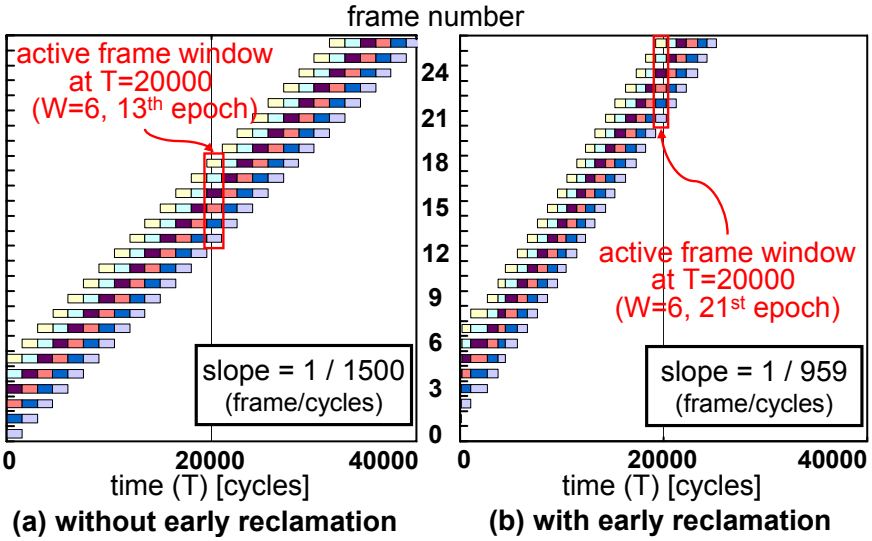


Figure 3-6: Frame life time analysis for comparison of frame reclamation rates with and without early reclamation. Although the estimated $e^{MAX} = 1500$ is within 10% of the observed worst-case e^{MAX}, the early reclamation increases the frame reclamation rate by >30%, which leads to corresponding improvement in the average throughput. See Section 3.5.1 for simulation setup. We use a hotspot traffic pattern with injection rate of 0.05 flits/cycle/node.

3.3.1 GSF Router Architecture

Figure 3-7 shows a proposed GSF router architecture. This router implements both carpool lane buffer sharing and early frame reclamation on top of the baseline GSF. Starting from a baseline VC router, we describe various aspects and design issues in the GSF router.

Baseline VC router We assume a three-stage pipelined VC router with lookahead routing [25] as our baseline. The three stages are next-hop routing computation (NRC) in parallel with virtual channel allocation (VA), switch allocation (SA) and switch traversal (ST).

Added Blocks Each router node keeps a local copy of the global head frame (HF) variable. This variable increments (mod W) at every frame window shift triggered by the global barrier network. Each VC has a storage to maintain the frame number (mod W) of the packet it is servicing. The frame number at each VC is compared against HF to detect any packet belonging to the head frame. Then the global barrier network gather information to determine when to shift the frame window appropriately.

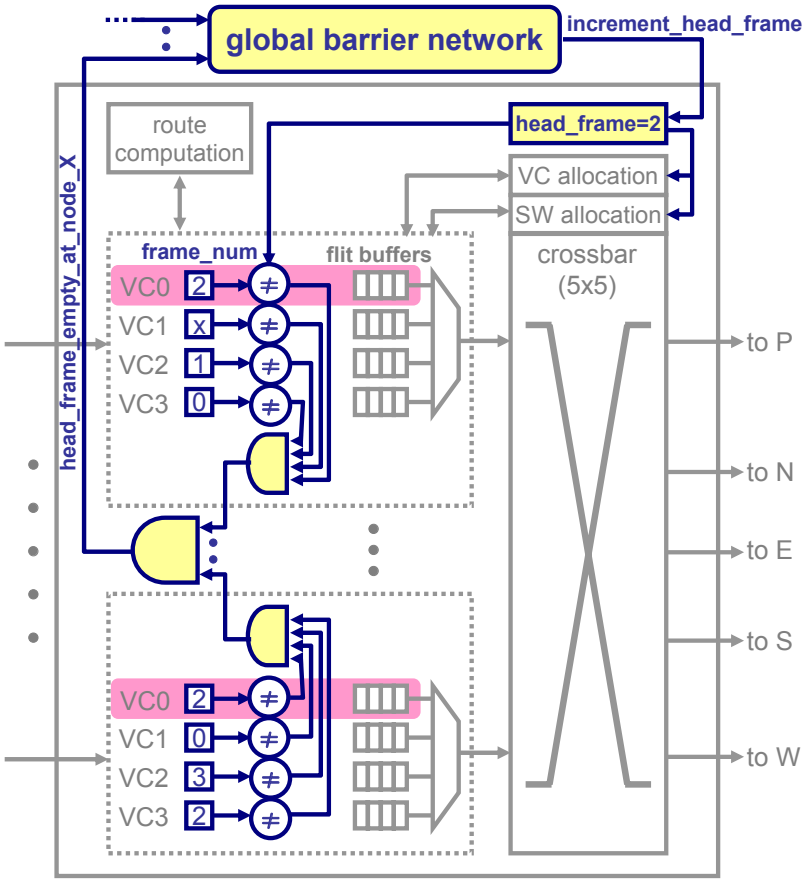


Figure 3-7: A GSF router architecture for 2D mesh network. Newly added blocks are highlighted while existing blocks are shown in gray.

Next-Hop Routing Computation (NRC) In order to reduce the burden of the VA stage, which is likely to be the critical path of the router pipeline, we precalculate the packet priority at this stage. The packet priority can be obtained by $(frame\_num - HF)$ (mod W). The lowest number has the highest priority in VC and SW allocation. When calculating the routing request matrix, NRC logic is responsible for masking requests to VC0 from non-head frames, because VC0 is reserved for the head frame only.

VC and SW allocation VC and SW allocators perform priority-based arbitration, which selects a request with the highest priority (the lowest number) precalculated in the previous NRC stage. The GSF router uses standard credit-based flow control.

3.3.2 Global Synchronization Network

The latency of the global synchronization network affects the overall network throughput because higher latencies leave VC0 (the carpool channel) idle for longer. Although our proposed router is generally tolerant to the latency of the barrier network if the frame size (F) is reasonably large, the impact is more visible for a traffic pattern having a high turnaround rate of frame buffers.

One way to achieve barrier synchronization is to use a fully-pipelined dimension-wise aggregation network [90]. In this network, assuming a 2D mesh, the center node of each column first collects the status of its peers in the same column. Then, it forwards the information to the center node of its row where the global decision is made. A broadcast network, which operates in reverse of the aggregation network, informs all nodes to rotate their head frame pointers (HF). For k-ary n-cube (or mesh) network, the latency of the synchronization will be $2n\lceil\frac{k-1}{2}\rceil$ cycles assuming one-cycle per-hop latency.

Alternatively, we can implement a barrier network using combinational logic which might take multiple fast clock cycles to settle. This requires significantly less area and power, and provides lower latency as cross-chip signal propagation is not delayed by intermediate pipeline registers. If propagation takes N fast clock cycles, we could sample each router's state and read the barrier signal every N cycles to determine when to perform a frame window shift. For evaluation, we use a variable number of cycles up to $2n\lceil\frac{k-1}{2}\rceil$

cycles.

3.4 System-Level Design Issues

A QoS-capable on-chip network sits between processors running the software stack and shared platform resources. This section addresses issues in interacting with these two ends of the system to provide robust QoS support.

Admission control: Admission control is a software process that should guarantee that no channel in the network is oversubscribed. That is, suppose $S_c = \{i_1, i_2, \cdots, i_n\}$ is a set of flows that pass through Channel c. Then $\forall$ Channel c in the network, $\sum_{i \in S_c} R_i \leq F$ should hold. To keep network utilization high, each flow can be granted more than the minimum number of slots required where possible, as the maximum number of flits in flight from flow i at any given time is upper bounded by WR_i.

A request for a new flow is accepted only if all the channels along the path of the flow have enough extra bandwidth. If a new flow enters into a previously reserved channel, the software stack may need to redistribute the excess injection slots according to its excess bandwidth sharing policy.

For a given set of flows in the network, the guaranteed share of Flow i (R_i) is limited by the most congested channel on its path. The maximum fair allocation of bandwidth can be achieved in the following way. (1) For Flow i, identify the most congested channel (i.e., shared by the most number of flows) on the flow's path. Assume this channel is shared by M_i flows. (2) Set R_i to be F/M_i.

Note that our GSF scheme does not require any explicit channel setup, and so only the R_i control register at each source must be changed. If there are multiple clock domains, possibly with dynamic voltage-frequency scaling (DVFS), any channel c should provide at least the sum of guaranteed bandwidths on the channel to preserve QoS guarantees.

Specifying bandwidth requirements: To specify requested bandwidth, one can use either a relative measure (e.g., 10 % of available bandwidth) as in [72] or an absolute measure (e.g., 100 MB/s). If a relative measure ρ_i is given, R_i can be set to be $\rho_i F$. If an absolute measure BW (in flits/cycle) is used, R_i can be set to be ($BW * e^{MAX}$).

e^{MAX} is a function of traffic pattern, bandwidth reservation, frame size, packet size, global synchronization latency, and so on, and it is generally difficult to obtain a tight bound. At this point, we rely on simulation to get a tight bound. That is, we keep the largest e^{MAX} observed for a given traffic pattern and bandwidth reservation, and add a safety margin to estimate the true e^{MAX}.

3.5 Evaluation

In this section, we evaluate the performance of our proposed GSF implementation in terms of QoS guarantees and average latency and throughput. We also discuss tradeoffs in the choice of network parameters.

3.5.1 Simulation Setup

Table 3.2 summarizes default parameters for evaluation. We implemented a cycle-accurate network simulator based on the *booksim* simulator [91]. For each run, we simulate 0.5 million cycles unless the simulation output saturates early, with 50 thousand cycles spent in warming up.

Simulation parameters	Specifications
Common parameters	
Topology	8×8 2D mesh
Routing	dimension-ordered
Router pipeline (per-hop latency)	VA/NRC - SA - ST (3 cycles)
Credit pipeline delay (including credit traversal)	2 cycles
Number of VCs per channel (V)	6
Buffer depth (B)	5 flits / VC
Channel capacity (C)	1 flit / cycle
VC/SW allocation scheme	iSlip [62] (baseline) or GSF
Packet size	1 or 9 flits (50-50 chance)
Traffic pattern	*variable*
GSF parameters	
active_frame_window_size (W)	same as number of VCs
frame_size (F)	1000 flits
global_barrier_latency (S)	16 cycles

Table 3.2: Default simulation parameters

We use an 8×8 2D mesh with four traffic patterns where the destination of each source at Node (i, j) is determined as follows: hotspot ($d_{(i,j)} = (8, 8)$), transpose ($d_{(i,j)} = (j, i)$), nearest neighbor ($d_{(i,j)} = (i+1, j+1)$ (mod 8)) and uniform random ($d_{(i,j)} = (\texttt{random()}, \texttt{random()})$). Hotspot and uniform random represent two extremes of network usage in terms of load balance for a given amount of traffic. The other two model communication patterns found in real applications, e.g. FFT for transpose, and fluid dynamics simulation for nearest neighbor.

3.5.2 Fair and Differentiated Services

We first evaluate the quality of guaranteed services in terms of bandwidth distribution. Figure 3-8 shows examples of fair and differentiated bandwidth allocation in accessing hotspot nodes. Figures 3-8 (a) and (b) illustrate QoS guarantees on a 8×8 mesh network and Figure 3-8 (c) on a 16×16 torus network. In both cases, the GSF network provides guaranteed QoS to each flow. We are able to achieve this without significantly increasing the complexity of the router partly because the complex task of prioritizing packets to provide guaranteed QoS is offloaded by the source injection process, which is globally orchestrated by a fast barrier network. We make the case for using a simple secondary network (barrier network) to control a primary high-bandwidth network to improve the efficiency of the primary network (i.e., to provide more sophisticated services in our case).

For all the simulation runs we performed, we confirmed that bandwidth is shared among all flows in compliance with the given allocation. Therefore, for the rest of this section, we focus on non-QoS aspects such as average throughput and tradeoffs in parameter choice.

3.5.3 Cost of Guaranteed QoS and Tradeoffs in Parameter Choice

The cost of guaranteed QoS with GSF is additional hardware including an on-chip barrier network and potential degradation of average latency and/or throughput. Unless a router has *a priori* knowledge of future packet arrivals, it must reserve a certain number of buffers for future high-priority packets although there are waiting packets with lower priorities. This resource reservation is essential for guaranteed QoS but causes resource underuti-

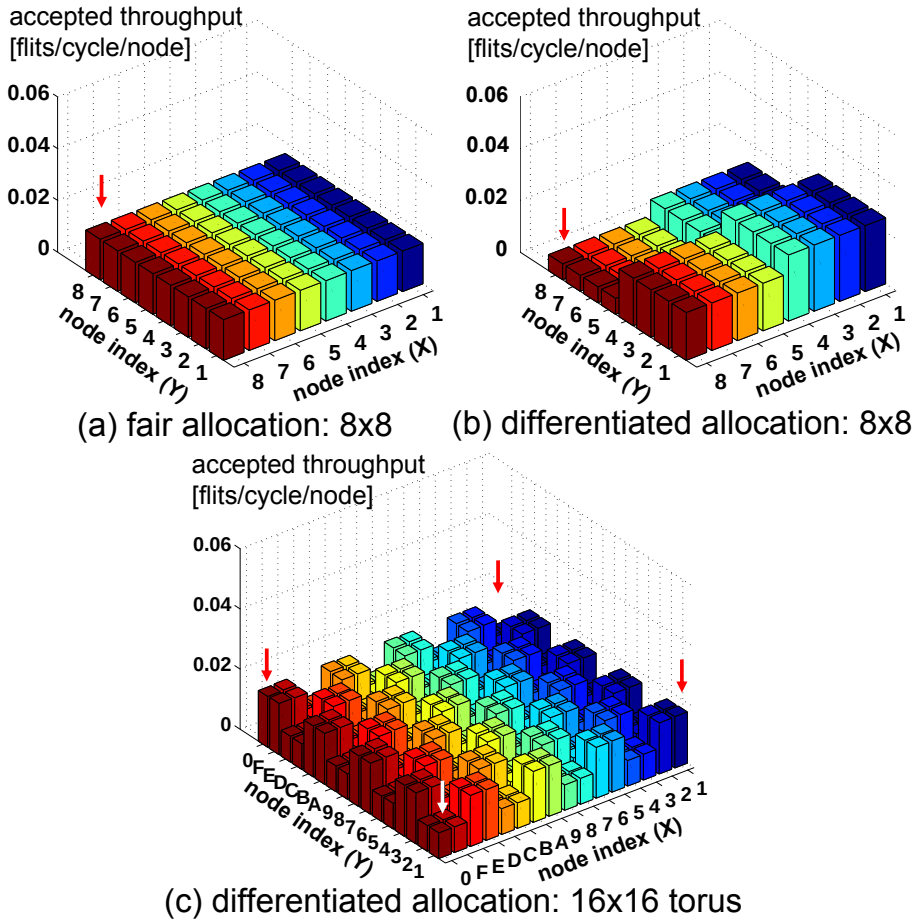


Figure 3-8: Fair and differentiated bandwidth allocation for hotspot traffic. (a) shows fair allocation among all flows sharing a hotspot resource located at (8,8). In (b), we partition a 8×8 CMP in mesh topology into 4 independent groups (e.g., running 4 copies of virtual machine) and provide differentiated services to them independent of their distance from the hotspot. In (c), we partition a 16×16 CMP in torus topology into 2×2 processor groups and allocate bandwidth to four hotspots in a checkerboard pattern.

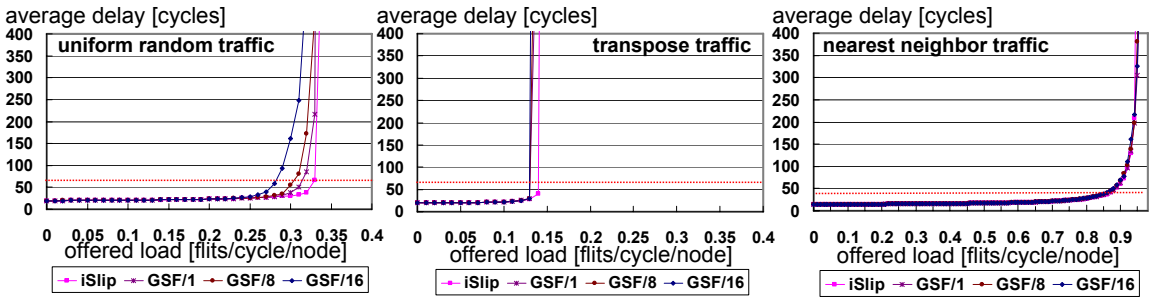


Figure 3-9: Average packet latency versus offered load with three traffic patterns. For each traffic pattern, we consider three different synchronization costs: 1 (GSF/1), 8 (GSF/8) and 16 cycles (GSF/16). Network saturation throughput is the cross point with dotted line ($3T_{zero-load}$ line) as in [49]. With GSF/16, network saturation throughput is degraded by 12.1 % (0.33 vs. 0.29) for uniform random, 6.7 % (0.15 vs. 0.14) for transpose and 1.1 % (0.88 vs. 0.87) for nearest neighbor, compared to baseline VC router with iSlip VC/SW allocation.

lization, degrading average-case performance. Therefore, it is our primary design goal to provide robust average-case performance over a wide range of network configurations and workloads. Note that robust performance by the GSF framework entails setting QoS parameters appropriately for a given workload.

Figure 3-9 shows the average latency versus offered load over three different traffic patterns. For uniform random traffic, we allocate $\lfloor F/64 \rfloor = 15$ flit injection slots per frame to each source (not to each source-destination pair), and these slots are shared by all packets from the same source. For the other traffic patterns, each source-destination pair is regarded as a distinct flow and allocated flit slots considering the link sharing pattern.

We first observe that the GSF network does not increase the average latency in the uncongested region. The network saturation throughput, which is defined as the point at which packet latency is three times the zero-load latency (as in [49]), is degraded by about 12% in the worst case. The performance impact of barrier synchronization overhead is the most visible in uniform random traffic because it has the lowest average epoch interval ($e^{AVG} \equiv (\sum_{k=0}^{N-1} e_k)/N$). Assuming 16-cycle barrier synchronization latency ($S = 16$), S/e^{AVG} ratios are 0.32, 0.15 and 0.09 for uniform random, transpose and nearest neighbor, respectively.

There are two main reasons for degradation of network saturation throughput: underuti-

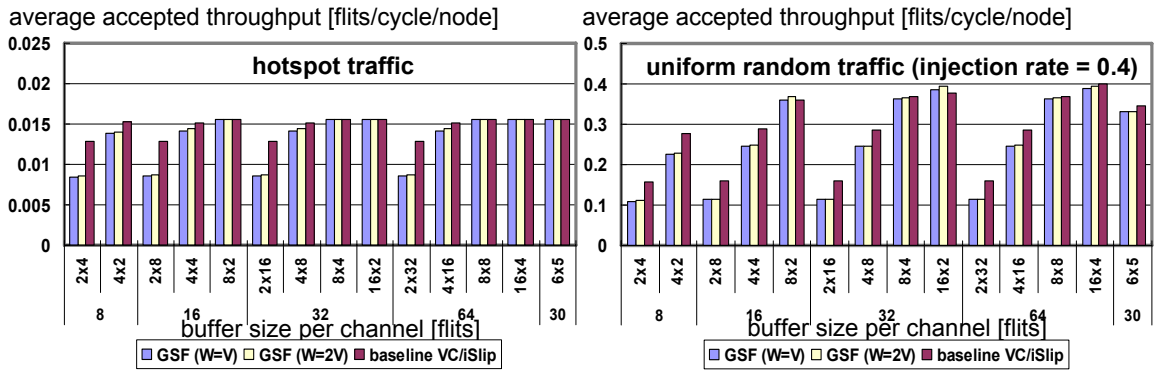


Figure 3-10: Tradeoff in buffer organization with hotspot and uniform random traffic patterns. VC buffer configuration is given by $V \times B$. The frame window size (W) is assumed to be V or 2V. For both traffic patterns, having $V \geq 4$ achieves 90% or higher throughput compared to the baseline VC router. Generally, increasing VCs improves the throughput at the cost of increased average latency. 6×5, our default, is a sweet spot for the specific router architecture we use.

lization of the head frame VC (VC0) and finite frame window. Because VC0 at each node is reserved to drain packets in the head frame, only (V-1) VCs are available for packets in the other active frames. The finite frame window prevents a flow from injecting more traffic than its reserved flit slots in the active frame window even when there are unclaimed network resources.

Figure 3-10 explains the impact of these two factors on average accepted throughput. With a small number of virtual channels, e.g., V=2, the throughput gap between GSF and baseline is dominated by underutilization of VC0 and increasing the window size from V to $2V$ does not improve throughput much. As the number of VCs increases, the gap narrows, and the performance gain from a wider window becomes more significant. To have enough overlap of frames to achieve over 90% of the throughput of the baseline VC router, the number of VCs should be at least four in this network configuration. With 8 VCs, the GSF achieves a comparable network throughput at the cost of increased average latency. We choose the 6×5 (virtual channels $\times$ buffers) configuration by default.

In choosing the window size (W), a larger window is desirable to overlap more frames, thereby increasing overall network throughput. However, the performance gain only comes with a cost for more complex priority calculation and arbitration logic. According to our

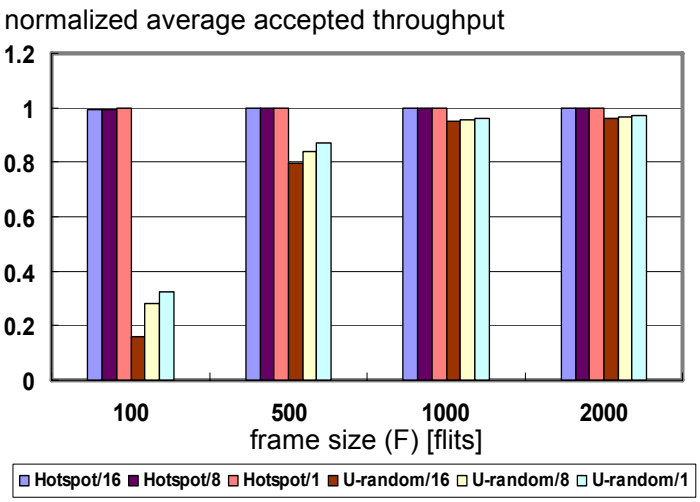


Figure 3-11: Throughput of GSF network normalized to that of the baseline VC router with variable F (frame window size). Two traffic patterns (hotspot and uniform random) and three synchronization costs (1, 8 and 16 cycles) are considered.

simulation, increasing W to be larger than $2V$ gives only marginal throughput gain. Therefore, we choose the frame window size (W) to be V by default as a reasonable design tradeoff.

In Figure 3-11, we explore the choice of frame size (F). A long frame (whose size is $\geq$ 1000 in this configuration) amortizes the overhead of barrier synchronization and effectively increases the size of injection window to support more bursty traffic, which is likely to improve the network throughput. The downside is larger source buffers and potential discrimination of remote nodes within a frame. The choice depends on workloads, synchronization overhead and system size.

3.6 Summary

In this chapter we introduce GSF to provide guaranteed QoS from on-chip networks in terms of minimum bandwidth and maximum delay bound. We show that the GSF algorithm can be easily implemented in a conventional VC router without significantly increasing its complexity. This is possible because the complex task of assigning priorities to packets for guaranteed QoS is pushed out to the source injection process at the end-points. The

global orchestration of the end-points and intermediate nodes is made possible by a fast barrier network feasible in an on-chip environment. Our evaluation of the GSF network shows promising results for robust QoS support in on-chip networks. In the next chapter we extend the GSF framework into multiple dimensions to handle multiple networks as well as heterogeneous bandwidth resources (e.g., DRAM channels and banks) in a single unified framework.

Chapter 4

GSF Memory System: Multi-Dimensional GSF

In this chapter we propose a Globally Synchronized Frame Memory System (GSFM) to provide guaranteed end-to-end QoS from a cache-coherent shared memory system by extending the GSF framework introduced in Chapter 3. Although GSF is shown to be effective in providing QoS for network bandwidth, it is only designed for point-to-point open-loop communication in a single network.

Unlike prior work focusing on designing isolated QoS-capable components, such as caches, memory controllers and interconnects, we take an end-to-end approach to provide guaranteed bandwidth to a cache-coherent shared memory system. GSFM manages multiple heterogeneous bandwidth resources in the memory system using a single unified QoS framework. GSFM provides a guaranteed minimum fraction of global memory system bandwidth to each core, including dynamic cache misses and coherence traffic, and also bounds maximum delay, without degrading throughput for typical usage scenarios. GSFM works seamlessly with the cache coherence protocol and requires little additional hardware.

4.1 GSF Memory System: an Overview

We present and evaluate the GSFM system using a system with private last-level caches (LLCs). Although several recent machines have shared LLCs, private LLCs can pro-

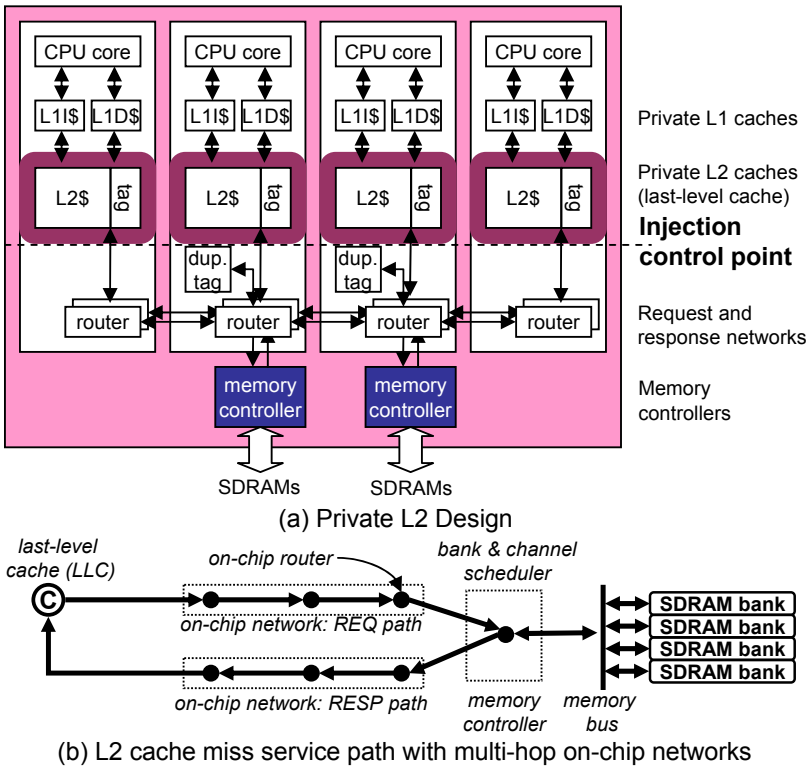


Figure 4-1: (a) A shared memory system with private L2 caches. The injection control is done at the boundary between private and shared resources. (b) Dots in (b) show distinct scheduling/arbitration points in the shared memory system.

vide performance advantages for some workloads and are featured on the new manycore Larrabee design, for example. We later discuss how GSFM can be extended to shared last-level caches.

The goal of GSFM is to provide guaranteed minimum memory bandwidth and maximum latency bound for each of the distributed outermost-level caches that are private (assumed to be L2 caches for the rest of this chapter) without degrading memory system throughput in typical usage scenarios. We call these outermost-level private caches *last-level private caches* (LLPCs). We assume L2 caches have non-uniform distance to memory controllers and that they are connected by scalable multi-hop on-chip networks as in Figure 4-1. Although we focus on three main bandwidth resources—DRAM banks and channels, and multi-hop on-chip interconnects—in this thesis, the framework can be easily extended to other bandwidth resources such as directory controllers and network interfaces. Instead of implementing QoS for each of these resources in isolation, GSFM offers a single frame-based unified framework to seamlessly provide end-to-end QoS in terms of memory access bandwidth for distributed caches. Time is coarsely quantized into frames, and each of the resources can service a certain amount of requests *within a single frame interval*. This amount is the *service capacity* of the resource. A unit of service from each resource is called a *token*. The service capacity of each component is calculated and represented in terms of tokens during the frame interval.

The operating system decides on the QoS guarantees for each application and sets the maximum token allocation for that application. Token allocations can be varied to provide differentiated services to different applications. The total token allocation given to all applications for a resource can not exceed the service capacity of the resource. Since GSFM uses worst case for accounting, all sources are guaranteed to receive their allocation during every frame interval. The L2 cache controller checks each request and determines how many tokens the request could take on each resource and deducts this from the source's allocation. If the request exceeds the allocation on a resource, it can not be injected until a future frame interval. GSFM takes into account cache-coherence operations in a shared memory system to eliminate the possibility that unusual coherence behavior could cause another application to be denied its guaranteed service.

While this worst case accounting might appear to result in low memory system utilization, GSFM allows applications to inject requests into future frames to use the excess bandwidth unclaimed by the current frame. This provides weighted sharing of excess bandwidth based on the source's allocation and efficient utilization of the system components.

4.2 GSFM Operations

The three main bandwidth resources GSFM manages are network links, DRAM banks and DRAM channels. To distribute the shared service capacity of each resource, we introduce the notion of a token as a unit of service for each resource. One token in a network link corresponds to one flit time, and one token in DRAM banks and channels to one memory transaction time. The OS distributes the tokens of each resource to L2 caches according to its QoS policy. An injection control mechanism at each L2 source node controls bandwidth allocation by restricting the number of memory requests that the source can inject into each frame. The injection control for each source keeps track of the source's usage of each resource using a per-resource token management table. As a source consumes its tokens in each frame for a resource, its injection frame number for the resource is incremented and it begins injecting requests into the next future frame, up to the maximum number of active future frames $(W - 1)$ in the system.

Each QoS request from a source could be tagged with a vector of frame numbers for the resources it will potentially use: the corresponding frame number will be used for scheduling decisions at each resource. For example, an application could have remaining tokens for an earlier frame for the network but only a later frame for the memory channel. In this case the request would move through the network at the priority of the earlier frame number and across the memory channel at the priority of the later frame number. At every scheduling point, requests with the earliest frame number are serviced with highest priority. Because we use modulo-W number to represent a frame number, it only adds only a few bits per resource in the header. This frame number can be regarded as a coarse-grain deadline.

In practice, we tag each request with the highest frame number among all the resources

it will potentially use to reduce hardware and bandwidth overhead. This tagging does not break QoS guarantees because it is impossible for a source to inject more requests than its per-frame allocation into any frame for any resource. There might be slight underutilization of resources, but our evaluation in Section 4.9 shows a minimal impact on performance.

To determine when all requests in the head frame have been serviced, GSFM exploits fast on-chip communication by using a global barrier network. This barrier operation will invoke advancement of all QoS building blocks in the memory system to the next set of frames synchronously. Unlike the GSF on-chip network [53], the barrier network does not have to check all the routers and other QoS building blocks in the system except source nodes because the transaction entry buffer at each source keeps track of all outstanding requests and their frame numbers. We call this operation a *frame window shift*. At every frame window shift, the next oldest frame now attains highest priority and does not admit any new request into it, while resources from the previously oldest frame, such as per-frame buffers and service bandwidth, are recycled to form the new futuremost frame. Sources are not allowed to inject new requests into the head frame to ensure the drain time of the head frame is tightly bounded.

Provided that the sum of distributed injection tokens per frame for each resource does not exceed the resource's service capacity and the number of future frames ($W - 1$) is large enough to saturate each resource's service capacity, Source i's guaranteed minimum bandwidth in accessing resource j is given as follows:

$$\boxed{\text{Guaranteed bandwidth}_i^j = R_i^j / F[tokens/cycle]}$$

where R_i^j is the number of tokens allocated to Source i for Resource j every frame. This follows from (1) proportional bandwidth sharing for the resource is guaranteed across sources within a frame according to their R_i^j's because a request is never blocked by lower-priority (later frame) requests, and (2) the resource's service capacity is larger than ($\sum_i R_i^j$).

Therefore, the minimum guaranteed bandwidth from the entire memory system is given by:

$$\boxed{\text{Guaranteed bandwidth}_i^{mem} = MIN_{\forall j}[R_i^j * b_j / F][bytes/cycle]}$$

where b_j is bytes per token for Resource j. This formula simply states that the memory system's guaranteed service bandwidth is determined by the weakest guarantee for any of its shared resources. The bottleneck resource is determined by Source i's memory access patterns. For example, if all memory requests are designated to a DRAM bank, it will be the bottleneck resource. If memory requests are uniformly distributed over DRAM banks, a DRAM channel may become the bottleneck resource. These guarantees hold only if all scheduling points in the memory access path implement per-frame queues to prevent priority inversion. In addition, the queues between these blocks and other building blocks such as directory controllers and network interfaces should also implement per-frame queues and earliest-frame-first scheduling.

4.3 Target Architecture

For the remainder of this chapter, we assume a specific target CMP architecture to explain our design decisions. Figure 4-2 (a) shows our target 16-core tiled CMP configuration. The network topology is 4×4 2D mesh. Both L1 and L2 caches are private to each processor. There are two memory controllers placed to the east of Node 11 and to the west of Node 4. Two directory controllers are co-located with memory channels to keep track of all on-chip cached lines for cache-coherence operations.

Figure 4-2 (b) shows a logical diagram of the memory subsystem and Table 4.1 summarizes four logical networks used to implement a MOSI protocol: c2hREQ (cache-to-home request), h2cRESP (home-to-cache response), h2cREQ (home-to-cache request) and c2cRESP (cache-to-cache response). This protocol is taken from the GEMS toolset [59]. These logical networks can be mapped to either physical or virtual networks. We choose to map them to four distinct physical networks by their functionality to achieve better performance isolation, similar to the Tilera Tile64 [7]. We believe that the GSFM idea can be applied to time-shared virtual networks overlaid on fewer physical networks as well.

In this target architecture, GSFM manages the following bandwidth resources: four physical networks, two DRAM channels and eight banks per DRAM channel (16 banks in total).

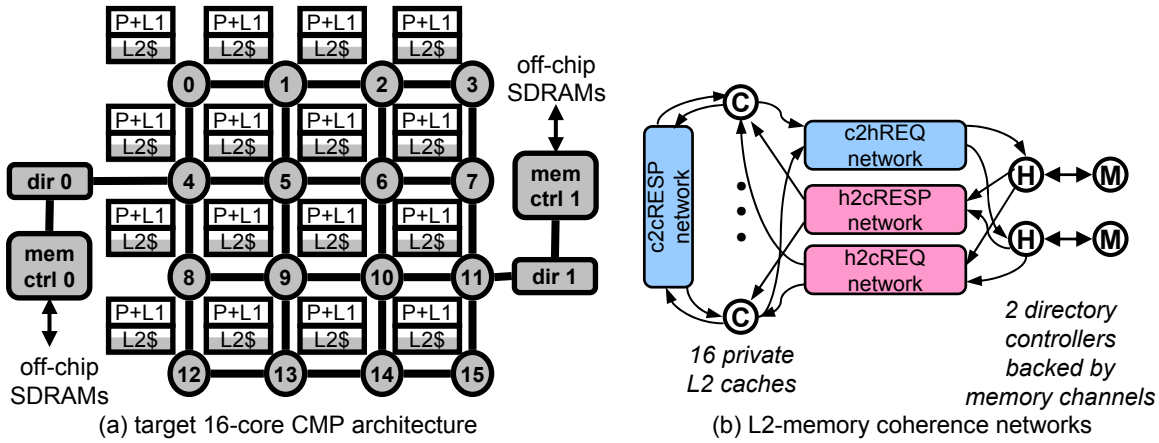


Figure 4-2: Target CMP architecture and L2-memory coherence network organization(C: local L2 cache, H: directory controller, or *home*, M: memory controller). In (a), the individual blocks in the scope of this thesis are shown in gray.

Network	Source	Destination	Traffic pattern	Coherence messages	Remarks
c2hREQ	L2 caches	directories	many-to-few	GETS, GETX, PUTX	frame_num assigned here
h2cRESP	directories	L2 caches	few-to-many	DATA, ACK#, WB_ACK	
h2cREQ	directories	L2 caches	few-to-many	FWD_GETS, FWD_GETX, INV	never used with payload
c2cRESP	L2 caches	L2 caches	many-to-many	DATA, INV_ACK	all responses go to the original requester

Table 4.1: Four logical networks for MOSI protocol we use for our implementation.

4.4 OS Design Issues

Admission control is performed by the operating system. Applications requiring QoS guarantees must make a request to the OS. Once the request is made the OS then determines if there are enough resources available to meet the QoS requirements for the applications without oversubscribing any resources and violating another application's QoS. If there aren't enough resources then the request must be rejected. However, if the resources are available then the application is admitted. Once the application is admitted into the QoS system, the operating system sets the token allocations, the GSFM hardware handles enforcement of the allocation through token management and injection control. See Section 4.5 for more details.

A manycore OS can effectively support space-time partitioning (STP) on top of the

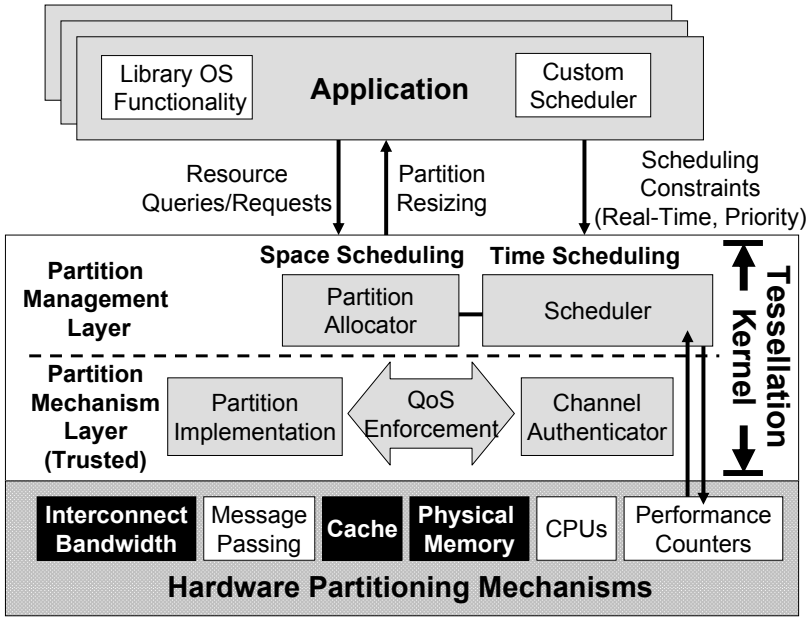


Figure 4-3: Components of Tessellation OS (taken from [58]). The Tessellation OS requires hardware partitioning mechanisms to provide space-time partitioning (STP) to platform users.

GSFM hardware that enforces the token allocations. Recent proposals for manycore OS's build on partitioning capabilities not only of cores and memory space but also of other shared resources such as caches, interconnect bandwidth and off-chip memory bandwidth [31, 58, 73]. Figure 4-3 illustrates one proposal advocating STP [58] for performance, energy efficiency, service-level QoS and fault isolation in a manycore client OS. A partition is an abstraction for resource allocation and scheduling (comparable to a process in a conventional OS), which contains a subset of physical machine resources such as cores, cache, memory, guaranteed fractions of memory or network bandwidth, and energy budget. STP demands both fair and differentiated bandwidth allocations to concurrent threads accessing shared resources.

Many different forms of software interfaces, APIs and policies for QoS guarantees can be realized on top of STP. One option is for the application to specify higher level performance metrics such as video frames/second, and the runtime system would translate these into memory system requirements, possibly using the mechanism in [51]. Another option is for the application to directly specify the resources it would like such as bytes/second.

The application could also specify nothing and just allow the runtime system to learn from its behavior and set QoS requirements accordingly.

4.5 Injection Control and Token Management

Injection control processes requests and determines if they are allowed to be sent to the memory system. Injection control is done at the sources of c2hREQ network because that is where new requests are generated and retired after being serviced.

Whenever a new request arrives, the injection control checks in its token management table to see if there are enough tokens available for *each* of the resources that can be potentially used by this request. If the source has used up its injection tokens within the window of future frames for *any* of the resources that can be potentially used by the request, the source must hold injection until a new frame opens and injection tokens are recharged. The request is billed for the worst-case resource usage for the request type regardless of whether it actually uses all the paid time slots or not. This conservative token billing guarantees that it is safe to inject the request without concerns for degrading other sharers' QoS. If the request does not take the worst-case path as charged, it will help reduce the drain time of the frame it belongs to and open a new futuremost frame early to recharge every source with their token allocations per frame for all the resources. We explored the option of providing refunds to sources for unused resources. However, we found this significantly complicated the hardware for little performance gain since our approach reclaims frames early and allows sources to inject into future frames. Table 4.2 summarizes the variables in the token management table.

To make a decision for injection control, the source should also know how much service bandwidth (in tokens) can be consumed from each resource by a request being injected into the c2hREQ network. In the MOSI protocol we are using, there are only three request types: `GETS`, `GETX` and `PUTX`. `GETS` is invoked by a load or instruction fetch miss, `GETX` by a store miss, and `PUTX` by eviction of a dirty cache line.

The maximum number of consumed tokens from each resource for each request type is summarized in Table 4.3. Figure 4-4 enumerates possible usage scenarios of the memory

Variables	Description
Global parameters and variables	
W	active frame window size [in frames]
HF	current head frame number
F	frame interval [in cycles]
Per-source Token management variables	
R_i^j	Source i's injection tokens per frame for Resource j
IF_i^j	Source i's current injection frame for Resource j
C_i^j	Source i's available injection tokens for Resource j into the current injection frame

Table 4.2: (Injection) token management table.

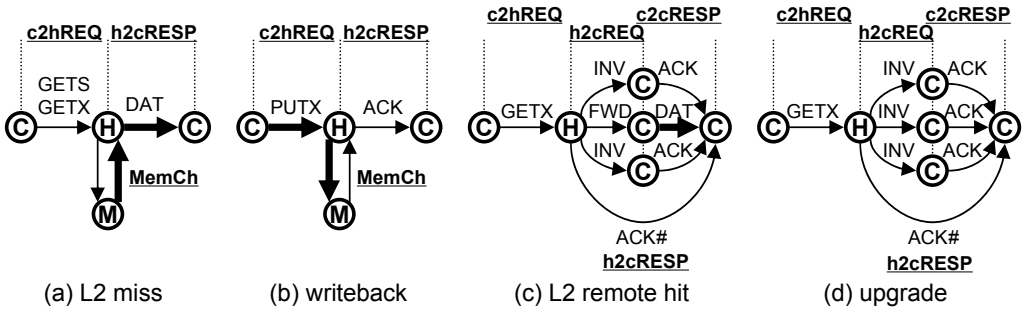


Figure 4-4: Cache coherence protocol operations and their resource usage (C: local L2 cache, H: directory controller, or *home*, M: memory controller). We can infer from this figure the maximum number of coherence messages that one original request into the c2hREQ network can generate. A thin arrow indicates a header-only message and a thick arrow indicates a message with payload data.

system resources to obtain this table. For example, the worst-case (maximum) usage of the c2cRESP network by a `GETX` request happens when the line is shared by all the other caches (remote hit as shown in Figure 4-4 (c)). In that case, the owner forwards the up-to-date data to the requester (1P) and all the other sharers send an `INV_ACK` message to the requester ($(N-2)$H). Note that this table is dependent on the network configuration such as topology, flit size, and multicast support.

For example, for a new `GETX` request whose address is mapped to Bank #0 at Memory Controller #1 to be injected, the source should have at least 1, 9, 16, 23, 1, 1 injection tokens for the corresponding resources c2hREQ.mem1, h2cRESP.mem1, h2cREQ.mem1, c2cRESP, mem1.Bank0, mem1.Channel, respectively, along the memory access path. Oth-

Request	Networks				Memory	
	c2hREQ	h2cRESP	h2cREQ	c2cRESP	memory bank	memory channel
GETS	1H (1)	1P (9)	1H (1)	1P (9)	1T (1)	1T (1)
GETX	1H (1)	1P (9)	N*H (16)	$(N-2)$H+1P (23)	1T (1)	1T (1)
PUTX	1P (9)	1H (1)	0 (0)	0 (0)	1T (1)	1T (1)

* An artifact of unoptimized protocol. Could be reduced by one.

Table 4.3: Resource usage table for our target CMP architecture and cache coherence protocol. This table illustrates how many service tokens can be consumed from each resource by one GETS, GETX or PUTX request into the c2hREQ network. H stands for header-only packet, P stands for header+payload packet and T stands for memory transaction. N is the number of sources, which is 16 in our target architecture. The numbers in parentheses shows the number of tokens assuming 64-byte cache line size and 64-bit flit size.

erwise, it would stall. Note that we maintain separate token counters for each memory channel in the c2hREQ network (h2cRESP and h2cREQ networks) because a memory request takes a different path depending on its destination (source) memory controller. The c2cRESP network maintains only one counter at each source because all the messages are designated to the original requester. A distinct counter is maintained for each memory channel (mem[01].Channel) and each memory bank (mem[01].Bank[0–7]) in our 16-tile target architecture from Section 4.3 (we assume eight banks per memory channel). We explain how to assign token counter values according to a QoS policy in the next section.

4.6 Determining the Service Capacity of a Resource

To determine how many tokens per frame are available for the OS to allocate, we calculate how many cycles a token is worth for each resource. Design specifications can be used to determine the throughput of each resource. Note that the token time should be measured under the worst-case usage scenario for each resource because we want to provide a minimum guarantee of service. This requires calculating the maximum sustainable throughput for each resource if the memory system is bottlenecked by that resource. Only allowing worst case allocation of tokens doesn't mandate low system utilization. Since the

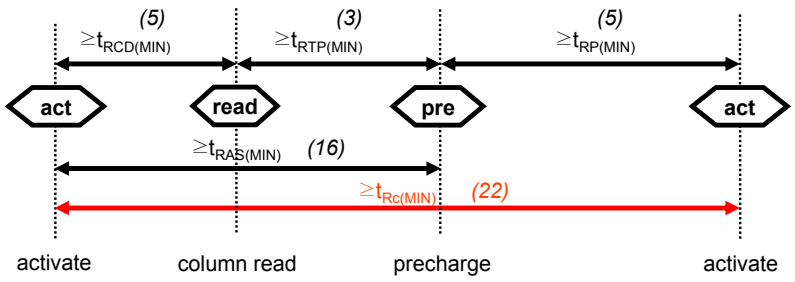


Figure 4-5: A timing diagram of Micron's DDR2-800 part: MT47H128M8. The worst-case service rate is determined by $t_{RC(MIN)}$. ($t_{RCD(MIN)}$: Activate-to-read, $t_{RTP(MIN)}$: Internal read-to-precharge, $t_{RP(MIN)}$: Precharge-to-activate, $t_{RAS(MIN)}$: Activate-to-precharge.)

resources may be able to process requests faster than their worst case, frames would just be completed more quickly than the nominal frame interval, and the system would move onto servicing future frames allowing full utilization of the memory system. Since GSFM provides weighted sharing of the excess bandwidth, the applications will still maintain the same priority when using the extra requests.

We perform this worst-case analysis for each resource as follows:

DRAM bank: Figure 4-5 shows timing constraints of a DDR2-800 part [63]. The maximum sustainable bandwidth of a DRAM bank is limited by $t_{RC(MIN)}$ (activate-to-activate to same bank). In this case, one transaction (one token) takes 22 SDRAM cycles, which corresponds to 110 processor cycles in a 2 GHz processor.

Memory channel: One memory transaction takes $BL/2$ SDRAM cycles in data bus, where BL is a burst length. Assuming the burst length of 8, it corresponds to 20 processor cycles.

Network link: Nominally, it takes one processor cycle to transport one flit, but bubbles can be introduced in the worst case traffic pattern. The token time is highly dependent upon buffer organization, flow control, routing efficiency, etc. We assume the utilization of 0.9 for the most congested link in the all-to-one traffic pattern. Note that this number is different from saturation throughput, which is measured by the injection rate which makes the network in question saturated. Therefore, the token cycle for network resources is 1/0.9=1.1 processor cycles. The source injection algorithm keeps track a request's worst case behavior on each link of the network for that given source. The injection is rejected

if the source has used its allocation on any of the links in the path traversed by the request, which we calculate assuming deterministic routing (not adaptive). In our topology, all memory requests need to go through the links to the memory controller which means we only need to keep track of the egress link to each memory controller node, since no other link can be oversubscribed if that one is not.

According to the worst-case analysis above, DRAM banks are the resource with the highest processor cycles per token. If we want to set the frame interval (F) to distribute 6 tokens for each DRAM bank to all sharers, it will correspond to 6 [tokens/source] * 16 [sources] * 110 [cycles/token] = 10560 [cycles]. Given this frame interval, we can calculate how many tokens per frame are available for distribution for each resource, which is shown in Table 4.4. If we distribute these tokens to all sharers in a fair manner, each source will have per-frame injection token vector R_i^j as follows:

$$R_i^j = (\overbrace{\overbrace{594, 594}^{c2hREQ}, \overbrace{594, 594}^{h2cRESP}, \overbrace{594, 594}^{h2cREQ}, \overbrace{594}^{c2cRESP}, \overbrace{6, \cdots, 6}^{memBank}, \overbrace{33, 33}^{memChannel}}^{\text{22 elements in total}})$$

Network				Memory	
c2hREQ	h2cRESP	h2cREQ	c2cRESP	mem bank	mem channel
9504	9504	9504	9504	96	528

Table 4.4: Maximum sustainable service rate from each resource [in tokens] under worst-case access patterns assuming F=10560 [cycles] (= 96 worst-case memory transaction time). For network resources, one token corresponds to one flit. For memory resources, one token corresponds to one memory transaction (read or write). These numbers specify the total number of tokens that can be allocated to all sources (i.e., $\sum_i R_i^j$ must be smaller than the corresponding number for Resource j).

Note that we can use a register file for each resource type in implementing the injection control logic because only one element per each resource type can be ever used by a request (e.g., a request cannot be sent to Memory Channel #0 and #1 simultaneously) as shown in Figure 4-6.

We can reduce the complexity of the injection control logic further. Considering this token distribution and our conservative billing to each request, it becomes evident that only three resources can ever stop a request from being injected regardless of traffic patterns—DRAM banks and channels, and the c2cRESP network. This does not mean that one of

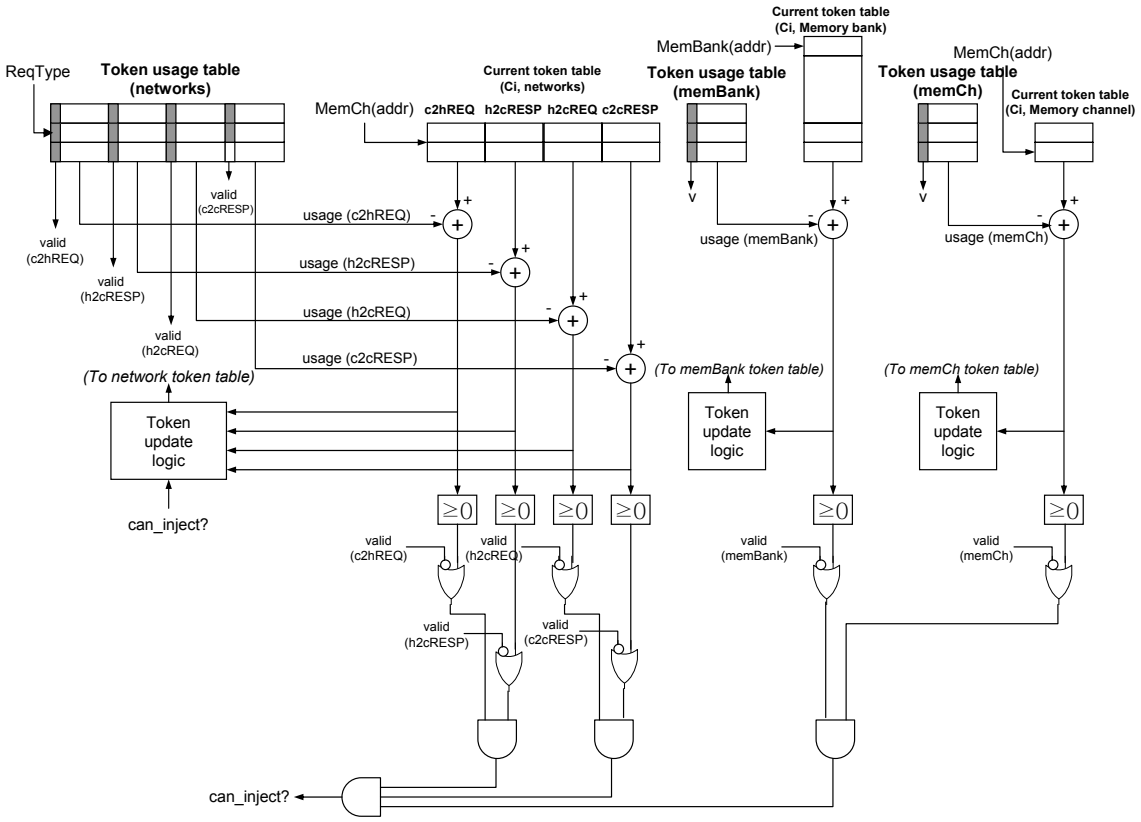


Figure 4-6: Datapath of source injection control. Per-resource type (not per-vector element) comparators are needed because a request cannot use more than one elements from each resource type simultaneously.

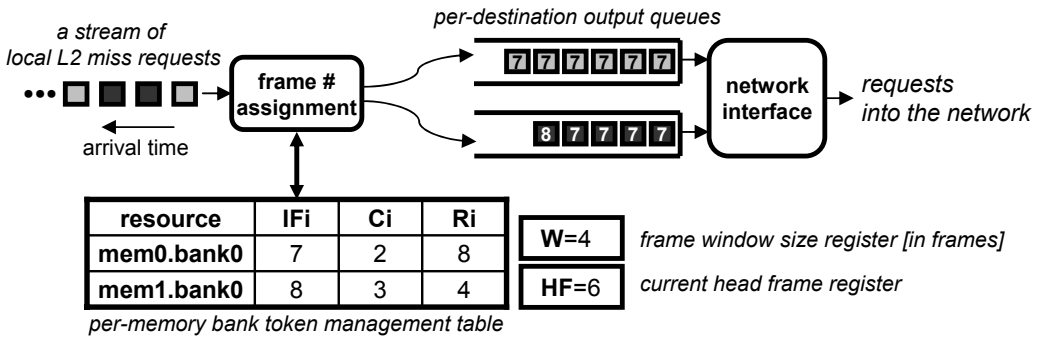


Figure 4-7: Injection controller for GSF memory system.

these three resources is always a bottleneck for the system—just that with worst case billing it is impossible to saturate any resource without saturating one of these resources. As a result we need to only perform injection control considering these tokens. Therefore, instead of maintaining a R_i^j vector of 22 elements, we discard the elements from the other resources to reduce the vector length to 17. Therefore, the injection process only needs to perform three comparator operations (i.e., one per each resource type) for a request to make an injection decision.

Figure 4-8 illustrates an injection controller reflecting this decision. For illustrative purposes, we assume all requests go to either Bank #0 of Memory Channel #0 (`mem0.bank0`) or Bank #0 of Memory Channel 1 (`mem1.bank0`) and show only these two counters in the figure. This source has used up its injection tokens in Frame 7 for `mem1.bank0` and keeps injecting requests using future frames (Frame 8). It can do so until it uses up all the injection tokens for the current active future frames (Frames 7-9). Remember that we do not allow a source to inject a new request into the head frame to keep the head frame drain time tight.

With our general framework for representing resources, we believe that GSFM can be easily extended to a shared L2 cache configuration. In this case, L1 caches become the last-level private caches (LLPC) and the points of injection control. Now cache coherence matters between L1 and L2 caches instead of between L2 caches and directory controllers co-located with DRAM controllers. The most notable difference is that each source needs to maintain more token counters because of the increase in the number of home nodes. In

one design, the vector length of R_i^j increases from 17 to 61 in the same hardware configuration (still in the range of hundreds of bytes per source), and the comparator count from 3 to 8. However, the basic idea is unchanged.

4.7 Selecting a Frame Interval

Another design issue in GSFM is how to set the frame interval (F). This parameter determines the bandwidth allocation granularity and the worst-case delay bound. The smaller the frame interval is, the tighter the delay bound is. However, the bandwidth allocation will be more coarse-grained because there are a smaller number of tokens to distribute to sources. Therefore, it is good to set F large enough that even the slowest resource has enough tokens per frame to distribute to sharers. The frame interval can also affect the amount of hardware resources since the frame interval multiplied by the window size determines how many messages can be in flight at one time. The total messages capable of being in flight simultaneously should be large enough to efficiently use all of the memory bandwidth. If frame intervals are small then more frames will be needed in the window which require more bits on packets and more buffers.

4.8 Memory Controller and On-Chip Router Designs

Like other resources, QoS memory controllers can be implemented with either priority-based or frame-based scheduling. Most of the recent proposals take a priority-based approach [67, 70, 80]. Note that these priority-based controllers invariably require per-thread queues and comparator logic to compare assigned priorities. We are aware of only one frame-based QoS memory controller, parallelism-aware batch scheduler (PAR-BS) [68]. In the PAR-BS approach, a memory controller forms a batch (frame) as packets arrive, and schedules request from the oldest batch first. It allows in-batch reordering of requests but not across-batch reordering to provide fairness.

GSFM requires memory controllers to implement per-frame queues and earliest-frame-first scheduling. Figure 4-8 illustrates our proposed GSFM memory controller. Compared

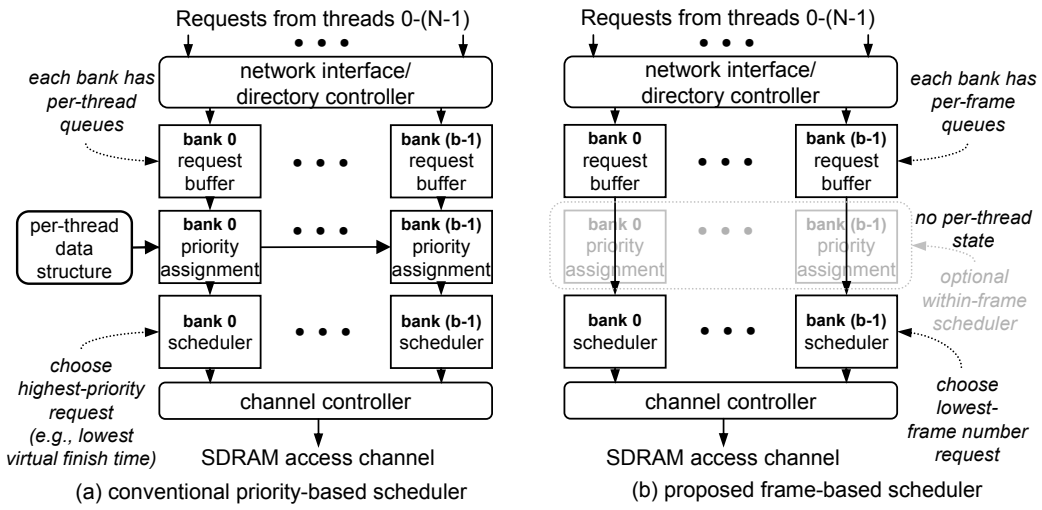


Figure 4-8: Our proposed frame-based memory scheduler. Components that are needed in conventional priority-based scheduler but not in our scheduler are grayed out for comparison. Each bank maintains not per-flow queues but per-frame queues which are generally more scalable to the number of flows. The scheduler block chooses the request with the earliest frame number from the per-frame queues.

to the priority-based scheduler in Figure 4-8, the proposed scheduler provides QoS without having per-thread queues or bookkeeping logic. The baseline bank and channel controller simply implements the earliest-frame-first scheduling of GSFM with a closed-row policy (close the open page as soon as possible), but it does not preclude the scheduler from implementing sophisticated in-batch scheduling as in the PAR-BS memory controller. Compared to PAR-BS, we offload the batching process from the memory controller and performed this as part of the source injection process.

For multi-hop on-chip networks, the GSF router detailed in Chapter 3 can be used for GSFM. The GSF router already implements per-frame queues and earliest-frame-first scheduling. GSFM even further simplifies the router because the check for head frame drainage can be done at the sources in the c2hREQ network. Therefore, the global barrier network does not have to check any router to determine when to make a frame window shift.

A network can require point-to-point FIFO delivery of messages for the correct operation of the cache-coherence protocol. The h2cREQ network is such an example in

our setup. For the h2cREQ network, a FIFO scheduling overrides the earliest-frame-first scheduling to ensure in-order delivery of protocol messages. There is no impact on QoS guarantees because the h2cREQ network only delivers header-only messages and never becomes a bandwidth bottleneck.

Finally, the carpool lane sharing to improve the network buffer utilization discussed in Section 3.2.4 should be applied carefully. This technique improves buffer utilization in the GSF network by reserving one virtual channel (VC) for the head frame use only but allowing the other VCs to be shared by all frames on first-come-first-served basis. This optimization allows priority inversion between non-head frames, and it can theoretically degrade the worst-case guaranteed bandwidth for Flow i to be R_i/e^{MAX} (instead of R_i/F) where e^{MAX} is the worst-case drain time of the head frame. However, in practice, the memory system utilization is usually bottlenecked by the source injection process at each processing node with a limited number of outstanding requests, and this optimization improves the throughput of each flow by increasing the frame reclamation rate. Therefore, we turn it on by default.

4.9 Evaluation

In this section, we evaluate the performance of our proposed GSFM implementation using a multiprocessor simulator with detailed timing models for caches, memory controllers and interconnects. We present per-thread memory access throughput and execution time to show robust guaranteed QoS provided by GSFM and the overall memory throughput to show its minimal performance overhead compared to a best-effort memory system.

4.9.1 Simulation Setup

Our target architecture is a 4×4 tiled CMP as detailed in Section 4.3. Table 4.5 summarizes our default parameters for the GSFM implementation. We partition the 16 cores into four quadrants as shown in Figure 4-9 to evaluate the impact of job placement. Typically, we place the application of interest on one quadrant and a variable number of background tasks, or memory performance hogs (MPHs), on the other three. We reserve one core, CPU

0 by default, for OS operations, so the maximum number of concurrent user threads is 15.

Simulation parameters	Specifications
Cache/memory parameters	
Processor clock frequency	2 GHz
L1 I-cache	16 KB private, 16 MSHRs 2 ways, 64-byte cache lines
L1 D-cache	same as L1 I-cache
L2 Cache	512 KB private, 64-byte cache lines 32 ways, 32 trans. buffer entries
SDRAM configuration	DDR2-800, 2 channels 1 rank/channel, 8 banks/rank
Memory controller	128 trans. buffer entries (16/bank), 64 write buffer entries, closed row policy
Memory channel bandwidth	8.6 GB/s/channel
SDRAM access latency	110 processor cycles
Network parameters	
Number of physical networks	4
Topology	4×4 2D mesh
Routing	dimension-ordered
Router pipeline (per-hop latency)	BW/RC -VA - SA - ST - LT (5 cycles)
Number of VCs per channel (V)	3 (1 for carpool lane in GSF)
Flit size	8 bytes
Buffer depth (B)	4 flits / VC
Channel capacity (C)	1 flit / cycle
VC/SW allocation scheme	round-robin or GSF
GSF parameters	
Frame window size (W)	4 frames
Frame interval (F)	10560 cycles (=96 worst-case memory transaction time)
Global barrier latency	8 cycles

Table 4.5: Default simulation parameters

Simulation is done by using the GEMS toolset from Wisconsin [59] which provides a detailed memory hierarchy and network model. GEMS is implemented on top of the Simics full-system multiprocessor simulator [94]. As a tradeoff between simulation speed and accuracy, we implement our own processor model with perfect value prediction [56]. The processor can issue up to four instructions every cycle assuming a perfect register value predictor; it stalls only when the miss status handling registers (MSHR) are full. This model is very optimistic, but we can interpret the cycle count in this setup as a performance upper bound (execution cycle lower bound) when an application's performance is completely memory-bound. Similar approaches were previously taken in simulating on-chip

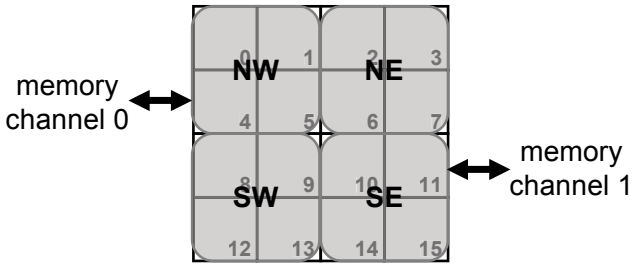


Figure 4-9: Job placement on 4×4 tiled CMP. This CMP is partitioned into four quadrants (NW, NE, SW and SE). Each tile is labeled with its CPU ID.

networks [44] and memory controllers [81].

We use both synthetic microbenchmarks and more realistic multithreaded benchmarks. We first present the results of various stress tests of the proposed memory system using microbenchmarks, which are followed by the results using the PARSEC benchmark suite [9].

4.9.2 Microbenchmarks

We use synthetic benchmarks to validate the simulator and obtain insights into GSFM system performance. The four synthetic benchmarks we use are: `hotspot_bank` (all load requests designated to Bank #0 at Memory Channel #0), `hotspot_channel` (all load requests designated to Memory Channel #0 but evenly distributed to 8 banks), `stream` (sequential memory accesses with a step of cache line size) and `random` (generating loads to randomly-generated addresses). `hotspot_bank` is the true worst case in accessing the shared memory system because a DRAM bank is the slowest component (i.e., having the least service capacity), and there is no overlap between any pair of memory requests.

Figure 4-10 shows per-source (or per-thread) memory access throughput normalized to the DRAM channel bandwidth. GSFM greatly improves fairness in accessing shared hotspot resources among individual threads without degrading the overall throughput. Without QoS support, the throughput difference between the most and least served `hotspot_bank` threads is more than a factor of 43. With GSFM's QoS support, the difference reduces to less than 0.02, and all threads receive more than their guaranteed minimum throughput shown by a horizontal red line in the figure. Furthermore, there is only negligible over-

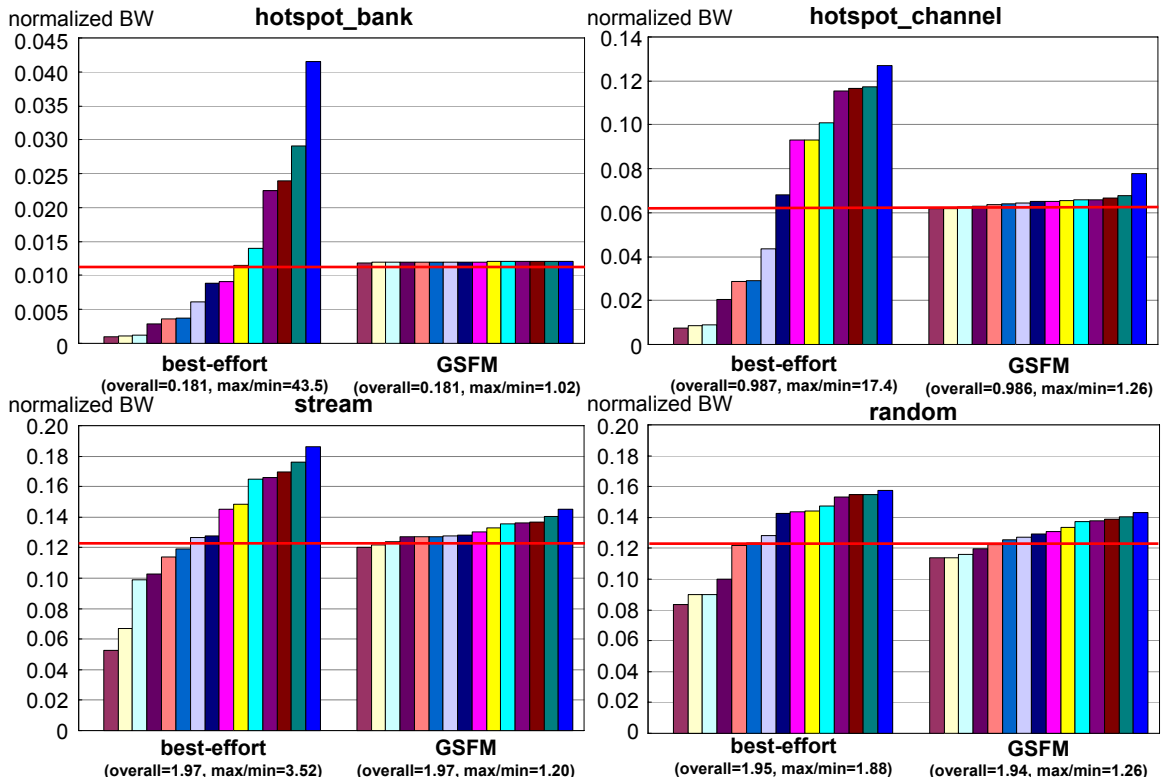


Figure 4-10: Per-source memory access throughput normalized to DRAM channel bandwidth (synthetic benchmarks). Horizontal red lines indicate the guaranteed minimum bandwidth in GSFM for given token allocations. Each configuration is annotated with the overall memory throughput over all threads ("overall") and the throughput ratio between the most and least served threads ("max/min").

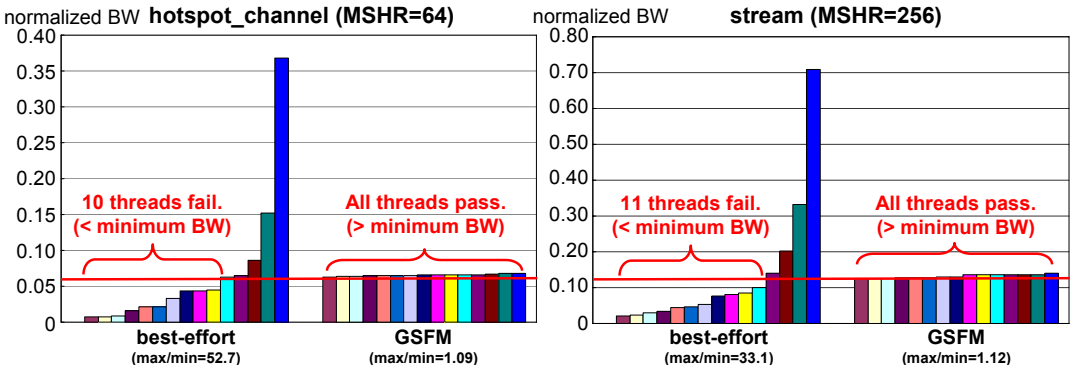


Figure 4-11: Per-source memory access throughput with `hotspot_channel` and `stream` benchmarks for larger miss status handling registers (MSHR). As a source can inject more requests into the memory system with a larger MSHR, the throughput variation in GSFM decreases, whereas the variation in the best-effort case increases. In both benchmarks, all threads in GSFM receive more than guaranteed minimum bandwidth.

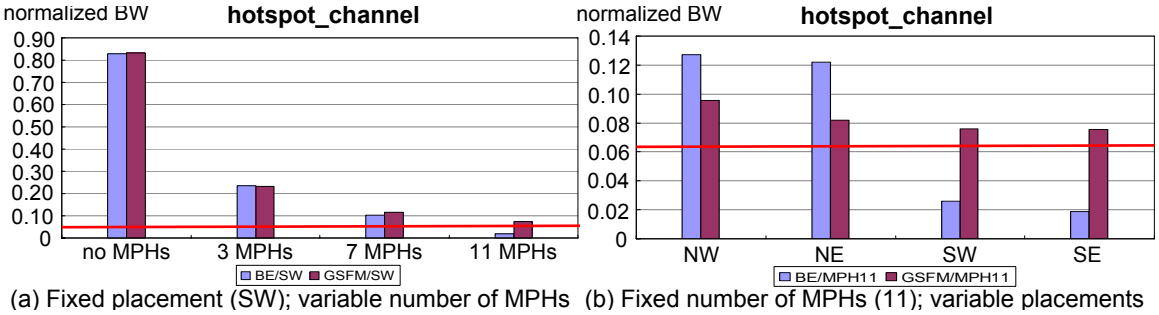


Figure 4-12: Memory access throughput of a `hotspot_channel` thread with variable number of memory performance hogs (MPHs) and different placement locations. Minimum performance guarantees for foreground process are enforced regardless of the number of background processes or placement locations.

all throughput degradation for all four benchmarks. `hotspot_channel` benchmark has higher guaranteed throughput (shown by a red line) because its requests are uniformly distributed over multiple DRAM banks to make the hotspot memory channel a bandwidth bottleneck. `Stream` and `random` have twice as much guaranteed throughput as `hotspot_channel` because their requests are uniformly distributed over two memory channels.

The degree of unfairness for `hotspot_channel` and `stream` benchmarks in GSFM is higher than in `hotspot_bank`. Even worse, the least served two or three threads in these benchmarks receive slightly lower bandwidth than the guaranteed minimum. This

unfairness is an artifact of non-ideal source injection mechanism. A processor can have only a finite number of outstanding requests, usually limited by the size of miss status handling registers (MSHR). Under an extremely long latency due to heavy contention, the MSHR (whose size is 16 by default) is not large enough to fill all available injection slots, and the injection rate is affected by the round-trip latency of memory requests. This non-ideal injection mechanism is not an issue specific to GSFM but a common issue that can happen in other heavily-contended memory systems with insufficient latency tolerance. As shown in Figure 4-11, this unfairness generally decreases as we increase the MSHR size. For example, the throughput ratio in `hotspot_channel` reduces from 1.26 to 1.09 as we increases the MSHR size from 16 to 64, and the ratio in `stream` from 1.20 to 1.13 as we increases the MSHR size from 16 to 256. More importantly, all threads in GSFM receive more than guaranteed minimum bandwidth in accessing the memory system, whereas most of the threads in a best-effort memory system fail.

Figure 4-12 shows a `hotspot_channel` thread (a foreground process) achieves minimum guaranteed throughput regardless of (a) how many memory performance hogs (MPHs) are running concurrently as background processes, or (b) which quadrants foreground/background processes are placed on. For the rest of this section, we use the `hotspot_channel` benchmark as default MPH, which happens to be the same as the foreground process in this case. In a highly-congested environment as in (b), process placement can have a significant impact on the performance of an application. Without QoS support, for a given set of applications, the throughput of a `hotspot_channel` thread can differ by a factor of 6.7 (NW versus SE) depending on how far it is placed from the hotspot resource. Our GSFM memory system not only mitigates this performance variability but also guarantees minimum throughput to the foreground `hotspot_channel` thread independent of its distance to the hotspot resource.

One concern with GSFM is that it might slow down an aggressive thread by source throttling even if there is no contention for the memory system. However, our experiment shows this impact is minimal with a reasonable frame window size (4 in our case). In Figure 4-13, we place a microbenchmark thread on one CPU and observe its throughput with and without QoS while keeping the other CPUs idle except for OS operations mostly

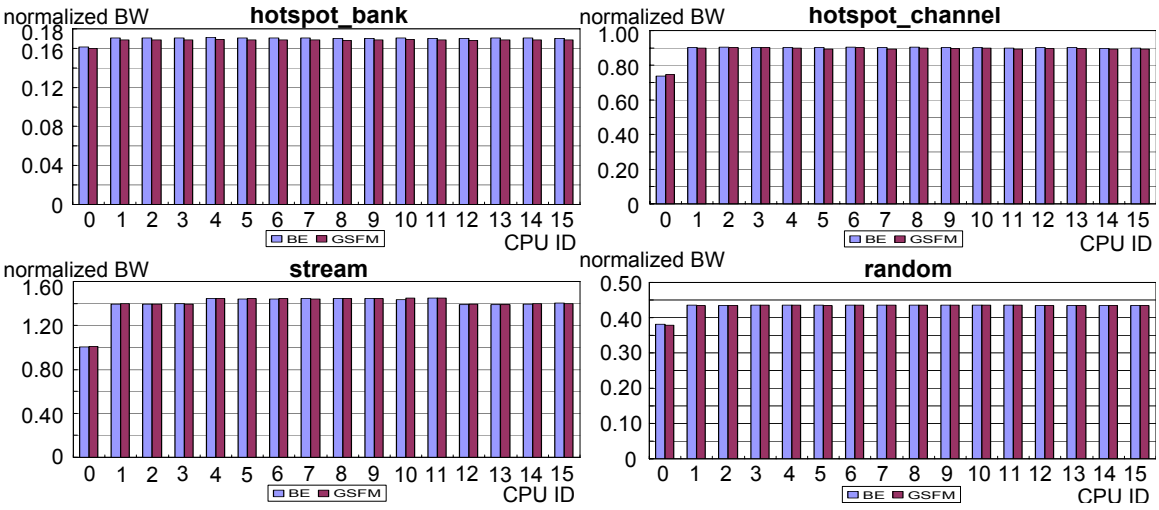


Figure 4-13: Per-source memory access throughput when only one source is activated. The average throughput degradation caused by QoS support is less than 0.4 % over the four microbenchmarks.

on CPU #0. The throughput degradation of the individual thread caused by the QoS support is less than 0.4 % on average compared to the best-effort memory system.

4.9.3 PARSEC Benchmarks

We evaluate GSFM with four multithreaded benchmarks from the PARSEC benchmark suite: `streamcluster`, `fluidanimate`, `swaptions` and `freqmine`. The number of threads for these applications is fixed to four, and we place them on the southeast (SE) quadrant. For each application, we skip the uninteresting parts such as thread creation and initialization routines and execute the entire parallel region between `__parsec_roi_begin()` and `__parsec_roi_end()` functions with `sim-small` input sets.

Figure 4-14 shows the effectiveness of GSFM's QoS support. We observe the execution time of each application in three different degrees of contention: zero (without any MPH), low (with 3 MPHs) and high (with 7 or 11 MPHs). In `streamcluster`, the overall execution time increases by 87% without QoS but only by 21% with QoS. The average slowdown (geometric mean) in a high contention environment ("high") compared to contention-free ("zero") execution over the four benchmarks is 4.12 without QoS, but only 2.27 with QoS.

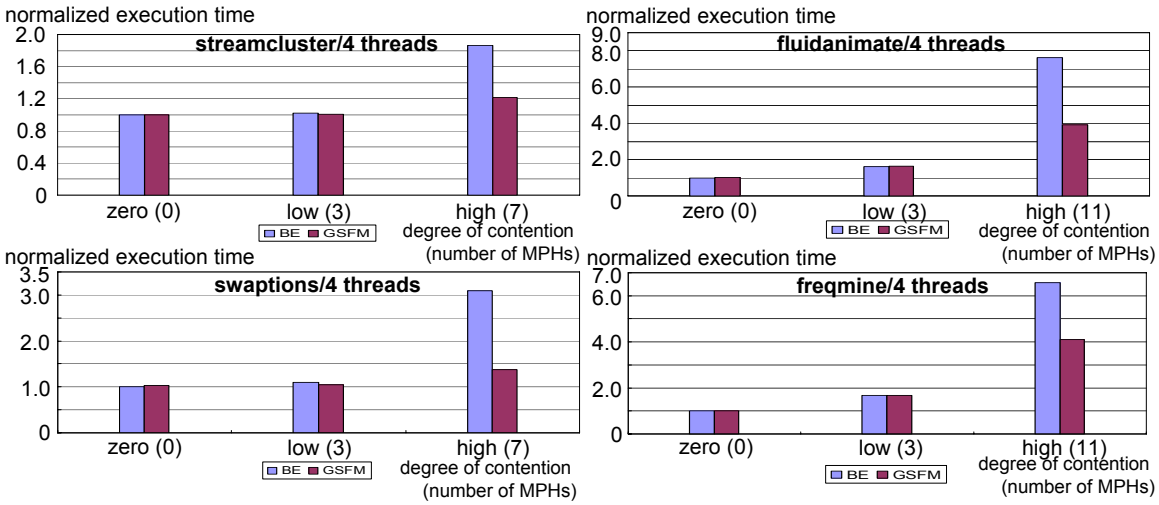


Figure 4-14: Execution time of PARSEC benchmarks normalized to the execution time with a zero-contention best-effort memory system. The lower, the better. The average slowdown (geometric mean) in a high-contention environment is 4.12 (without QoS) and 2.27 (with QoS).

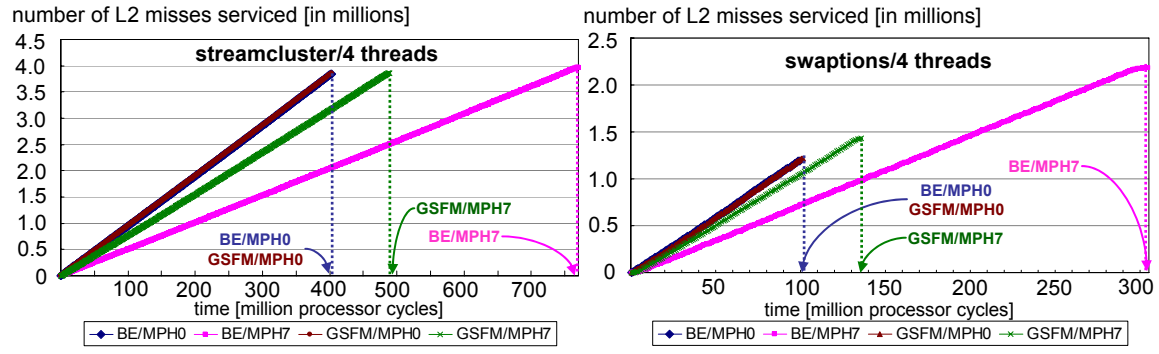


Figure 4-15: Cumulative L2 miss count over time. On X axis, the completion time of parallel region for each configuration is marked with a label. The lines for BE/MPH0 and GSFM/MPH0 configurations almost completely overlap. The slope of a line is the L2 miss throughput for the corresponding configuration.

Unlike microbenchmarks, it is hard to define steady-state throughput for these applications because of their complex phase behaviors and generally lower misses per instruction. Instead, we show cumulative L2 miss counts over time for two applications while varying the degree of contention (zero, low and high) to illustrate their bandwidth usage, which is sampled every one million cycles and summed over four worker threads. Our optimistic processor model makes graphs look almost linear partly because an application's data dependency does not affect its request injection pattern. The slope of a line is the L2 miss throughput for the corresponding configuration. Under high contention, the GSFM configuration (GSFM/MPH7) receives much more bandwidth (higher slope) than the best-effort configuration (BE/MPH7). In `streamcluster`, the total number of L2 cache misses in the parallel region is almost the same over all the configurations, which suggests the difference in execution time results from the difference in average memory access latency. (`Fluidanimate` and `freqmine` follow the same bandwidth usage pattern.) On the other hand, `swaptions` has significantly different L2 miss counts among configurations, where large variations in memory access latency lead to different execution paths and/or OS intervention patterns even if all configurations started execution from the same architectural state.

4.10 Related Work

QoS support in multicore platforms is gaining more and more attention as the number of processor cores on a single die increases and on-chip resources are more physically distributed. We will briefly survey QoS research in a multicore environment.

4.10.1 System-wide QoS frameworks and multicore OS

Recently, researchers have proposed multicore OS architectures that build upon hardware QoS/partitioning support like GSFM to ease resource management in a multicore platform [58, 72]. Virtual Private Machine (VPM) [72] is an example of component-wise QoS approaches, where each resource is required to implement its own QoS. Herdrich et. al. proposes a source throttling approach by clock modulation and frequency scaling to prior-

itize high-priority tasks in using the shared memory system [33]. In their work, injection control is done at the processor pipeline instead of the network interface in GSFM. However, their QoS does not provide any service guarantees.

4.10.2 QoS-Capable Memory Controllers

Nesbit et al. first brought up fairness issues in memory scheduling and proposed the Fair Queueing Memory System (FQMS) [70]. They took a priority-based approach, which requires per-thread queues and data structures. Other proposals with similar approaches have followed FQMS, often with different QoS or fairness metrics. Mutlu et al. [67] proposed a Stall-Time Fair Memory Scheduler (STMS) to equalize the DRAM-related slowdown among competing threads, and have recently proposed another memory controller named parallelism-aware batch scheduler (PAR-BS) [68]. PAR-BS is the only memory scheduler that takes a frame-based approach of which we are aware. They form batches (frames) locally, but our frames are global. Also, their in-batch scheduling can be easily applied to the GSFM memory controller to further improve memory throughput.

4.10.3 QoS-Capable Caches and Microarchitectural Components

Researchers have also investigated QoS support for other on-chip shared resources. Iyer [35] addresses the QoS issue in shared caches of CMP platforms. Kalla et al. [39] allocate shared resources to concurrent threads on a simultaneous multithreaded (SMT) processor, such as instruction issue bandwidth, based on a fixed static priority. Virtual Private Cache [71] addresses QoS issues in accessing shared on-chip L2 caches in CMP. Combined with our work, these schemes can provide seamless end-to-end performance guarantees to applications.

4.11 Summary

In this section, we have presented a frame-based end-to-end solution to provide QoS for the memory hierarchy in a shared memory system. Using source injection, which calculates the

total potential resource usage of a request, we are able to provide fairness and differentiated services as a guaranteed fraction of the total bandwidth. The frame-based aspect of our system allows us to simplify our routing hardware and scheduling logic, and reduce buffers, while still providing a latency bound for requests. By allowing sources to inject into future frames, applications can take advantage of unused bandwidth in the system and not be punished for bursty access behavior. Careful design of the network and system allows the source injection control to be simplified significantly.

Quality of Service is rapidly increasing in importance for single chip systems as the number of core scale up in manycore designs. Applications must be provided with minimum service guarantees despite which other applications are running on the machine concurrently to prevent starvation, long latency, and unpredictable performance on these manycore platforms. Local QoS at shared resources will not provide the global QoS necessarily to meet this goal, so end-to-end systems such as GSFM will be necessary additions to future manycore architectures.

Chapter 5

Bridging the Gap between QoS Objectives and Parameter Settings

In this chapter we switch to a different QoS problem. The question is how to find a minimal resource reservation setting (e.g., memory bandwidth) for a given user-level performance goal (e.g., execution time, transactions per second). Finding a minimal resource reservation is crucial not to waste platform resources by overprovisioning. To address the issue, we introduce the METERG (MEasurement-Time Enforcement and Run-Time Guarantee) QoS system that provides an easy method of obtaining a tight estimate of the lower bound on end-to-end performance for a given configuration of resource reservations. Our goal is to provide a well-controlled QoS environment to support real-time tasks and ease performance tuning. Note that the METERG framework does not require GSF and that it can be applied to any QoS-capable platform resource.

5.1 Introduction

We believe future integrated CMP platforms must implement robust QoS support. A QoS platform we envision has hardware-enforced resource allocation controlled by an operating system, which allocates resources and performs admission control to ensure QoS resources are not oversubscribed. When starting, an application that requires QoS guarantees should express its resource and QoS requirements to the OS.

Even though QoS-capable hardware components provide performance isolation and allow tighter bounds on a task's execution time, a major problem still remains: users would like to request only the minimal resources needed to meet the desired performance goal. That is, translating a high-level performance goal for a complex application (e.g., transactions per second) into minimal settings for the hardware QoS components (e.g., memory bandwidth and latency) is challenging and usually intractable.

One approach is for the user to try to measure the performance of their code running on the target system with varying settings of the resource controls. Unfortunately, a conventional QoS-capable shared resource usually distributes unused resources to the sharers so as to maximize the overall system's throughput. Consequently, even with an identical setting of resource reservation parameters, the observed completion time of an instruction sequence fluctuates widely depending on how much additional resource it receives. Even if competing jobs are run to cause severe resource contention, we cannot safely claim the observed performance is really representative of worst case contention.

To address this problem, we propose a new measurement-based technique, METERG (Measurement-Time Enforcement and Run-Time Guarantee), where QoS blocks are modified to support two modes of operation. During performance measurement, resource guarantees in the QoS blocks are treated as an upper bound, while during deployment, resource guarantees are treated as a lower bound. For example, if a thread reserves a bandwidth share of 1 Gb/s in accessing a shared memory, it receives *at most* 1 Gb/s bandwidth in enforcement mode, whereas it receives *at least* 1 Gb/s in deployment mode. The METERG QoS system enables us to easily estimate the maximum execution time of the instruction sequence produced by a program and input data pair, under the worst-case resource contention. In this way, we can guarantee measured performance during operation.

Unlike static program analysis techniques, which cover all possible input data, the METERG methodology is based on measurement of execution time for a certain input set. However, this is still useful, especially in soft real-time systems, where we can use representative input data to get a performance estimate and add a safety margin if needed. In a handheld device, for example, a small number of deadline violations may degrade end users' satisfaction but are not catastrophic. Such potential violations are tolerable in

exchange for the reduced system cost of shared resources. In contrast, hard real-time systems, such as heart pacemakers, car engine controllers, and avionics systems, justify the extra cost for dedicated resources with possibly suboptimal throughput.

5.2 Related Work

The end-to-end performance (e.g. execution time) of a program running on a multiprocessor is the result of complex interactions among many factors. We identify three major factors causing non-deterministic performance. First, performance depends on the input data to the program, as this determines the program's control flow and data access patterns. Second, even if we run the program with identical input data multiple times, the performance of each run can vary widely because of the different degrees of contention in accessing shared system resources. Third, *performance anomalies* [69, 89] in microprocessors also affect the end-to-end performance.

The first factor has been actively investigated in the real-time system community in Worst Case Execution Time (WCET) studies [3, 4, 32, 55, 85, 101]. Researchers have made efforts to estimate a program's WCET across all possible input data. The third factor is not problematic as long as a processor is *performance monotonic*, i.e., longer access time to a resource always leads to equal or longer execution time. Performance monotonicity holds for simpler processors, but not for some complex out-of-order processors. A code modification technique [89], or an extra safety margin can be added to the estimated lower bound of performance to cope with performance non-monotonicity. We do not cover this issue further in this thesis. The METERG QoS system reduces the non-determinism caused by the second factor: resource contention.

The goal of QoS research is to limit the impact of the non-determinism coming from resource contention. Although there are numerous proposals which aim to bound the non-determinism of individual resources, their link to the user-observable end-to-end performance still remains largely unclear. Therefore, users often have to rely on the high-level knowledge of a program in setting up the resource reservation parameters appropriately to meet a certain end-to-end performance goal.

One interesting approach to fill the semantic gap between user-level performance goal and resource reservation setting, is to design hardware components that take as an input each application's performance goal in a metric specified by a user. For example, Cazorla et al [15] propose an SMT processor architecture whose QoS metric is instructions-per-cycle (IPC). In this processor, a dynamic adjustment is made to allocation of shared resources such as renaming registers, instruction/load/store queues, L2 cache, to meet the IPC goal set by a user. One drawback of their proposal is that the adjustment is made based on observations during the previous sampling period, which could be in a different program phase with completely different IPC characteristics. Mutlu et al [67] propose a Stall-Time Fair Memory Controller which equalizes the slowdown of execution time caused by resource contention at the memory controller among multiple threads. However, QoS metrics in this class of approaches are too high-level to implement flexible policies in response to the system's various workloads. We need a better separation between mechanisms and policies [54] for efficient yet flexible QoS support.

Note that our work differs from real-time scheduling research at OS level [23, 36, 57, 88]. In the conventional formulation of the real-time scheduling problem, it is assumed that, for a given set of n tasks ($T_1, T_2, \cdots, T_n$), the execution time of each task (or its upper bound) is known in advance. Then each task is scheduled for execution at a particular time. In contrast, our work provides performance guarantees to ensure the estimated execution time of each task is not violated in practice because of severe resource contention. Our work is independent of the scheduling algorithm.

In order to estimate a lower bound on end-to-end performance, existing frameworks for WCET estimation can be used [3, 4, 32, 55, 85, 101]. By assuming all memory accesses take the longest possible latency, the execution time, given worst-case resource contention, can be calculated. However, use of WCET is limited. Most WCET estimation techniques require several expensive procedures: elaborate static program analysis, accurate hardware modeling, or both. These techniques cannot be easily applied, if at all, to complex hardware components and programs.

5.3 The METERG QoS System

We first overview the METERG QoS system with two operation modes: enforcement and deployment. Further, we propose two possible ways to implement the enforcement mode: strict and relaxed. The strict enforcement mode trades tightness of the performance estimation for safety, and vice versa. Finally, we illustrate the proposed framework with a METERG QoS memory controller as an example.

5.3.1 Overview

In the METERG QoS system, each process requiring QoS support (QoS process for brevity) can run in two operation modes: *enforcement* and *deployment*. In enforcement mode, a QoS process cannot take more than its guaranteed resource share from each QoS block even when there are additional resources available. In deployment mode, however, the process is allowed to use any available additional resources (we do not discuss policies for sharing excess resources among multiple QoS processes in this thesis) [1]. If a QoS block supports the two operation modes as described above, we call it a *METERG QoS block*. If every shared resource within a system is capable of METERG QoS, the system is termed a METERG system.

With a METERG system, a user first measures execution time of a given code sequence with a given input in enforcement mode with given resource reservation parameters. The METERG system then guarantees that a later execution of the same code sequence in deployment mode will perform as well as or better than the previous execution in enforcement mode, provided the same parameters are used for resource reservation. An example usage scenario might take the following steps:

- **Step 1** Given an input set and a performance goal for execution time (ET_{GOAL}), the user runs the program with the input set in enforcement mode. The system provides APIs to set resource reservation parameters to arbitrary values. (Setting these parameters to reasonable values can reduce the number of iterations in Step 2.)

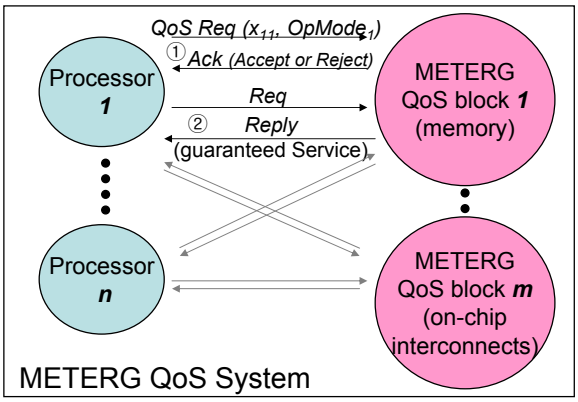
[1] In other setups, the two QoS modes, enforcement and deployment, are called *Maximum* and *Minimum* Virtual Private Machines (VPM) in [73], and WCET Computation and Standard Execution Modes in [75], respectively. Both frameworks [73, 75] were published after the METERG framework [51].

- **Step 2** Iterate Step 1 with various resource reservation vectors to find a minimal resource request setting that still meets the performance goal. If the measured execution time (ET_{MEAS}) is smaller than $ET_{GOAL} - ET_{MARGIN}$ (where ET_{MARGIN} is a safety margin, if needed), the user may try a smaller resource reservation vector. If not, they may increase the amount of reserved resources. We believe that this process can be automated, but do not consider an automated search in this thesis.

- **Step 3** The estimated minimal vector of resource reservation is stored away, for example, as an annotation in the binary. Since resource reservation vectors are only partially ordered (i.e., vector A is smaller than vector B if and only if every element in vector A is smaller than its corresponding element in vector B), there can be multiple minimal vectors, or pareto-optimal points, meeting the performance goal, to enable better schedulability for the program.

- **Step 4** Later, when the program is run in deployment mode, the OS uses a stored resource reservation vector to configure the hardware accordingly. Note that the system may reject the resource reservation request if oversubscribed. If the request is accepted, the execution time (ET_{DEP}) is guaranteed to be no greater than $ET_{GOAL} - ET_{MARGIN}$. Any performance slack ($ET_{GOAL} - ET_{MARGIN} - ET_{DEP}$) of the program in runtime can be exploited to improve the system throughput or reduce energy consumption as described in [3].

Figure 5-1 depicts the METERG system model. There are n processors and m METERG QoS blocks (e.g., interconnects, DRAM memory channels, caches, I/Os). These QoS blocks are shared among all n processors, and can reserve a certain amount of resource for each processor to provide guaranteed service (e.g., bandwidth, latency). Unlike conventional guaranteed QoS blocks, they accept an extra parameter, $OpMode_i$, from a processor to request the QoS operation mode. If the processor requests enforcement mode, the strict upper bound on runtime usage of a resource is enforced in every METERG QoS block. If the processor requests deployment mode, it can use additional otherwise unclaimed resources.



Phase ①: Resource reservation phase
(Setting up a service level agreement)
Phase ②: Operation phase
(Requests are serviced with bandwidth/latency guarantees.)

Figure 5-1: The METERG QoS system. Each QoS block takes an extra parameter (OpMode) as well as a resource reservation parameter ($x_{i,j}$) for each processor.

We assume that the j-th METERG QoS block (where $1 \leq j \leq m$) maintains a resource reservation vector of n real numbers, $(x_{1,j}, x_{2,j}, \ldots, x_{n,j})$, where the i-th entry ($x_{i,j}$) specifies the fraction of available resource reserved for processor i. We use a single number per processor for simplicity. For example, a shared memory bus may take the vector to determine the fraction of available bandwidth it will allocate to each processor, and set up the bus arbiter properly (time sharing). Another example could be a shared cache to determine how many ways or sets it will allocate to each processor (space sharing). The range of $x_{i,j}$ is between 0 and 1 by convention, where $x_{i,j}$=1 indicates that the i-th processor will monopolize all of the available resource from the QoS block, and $x_{i,j}$=0 indicates no guaranteed share for the i-th processor or only a *best-effort* service from the QoS block. Note that guaranteed services can be provided only to a limited number of sharers meeting the condition $\sum_{i=1}^{n} x_{i,j} \leq 1$.

Before further discussion, we present a set of assumptions for the rest of this chapter:

- We limit ourselves to dealing with *single-threaded* programs in the multiprogrammed environment. We do not consider interprocess communication or shared objects among multiple threads running on different processors.

- We assume a single-threaded processor core, where a scheduled process (thread)

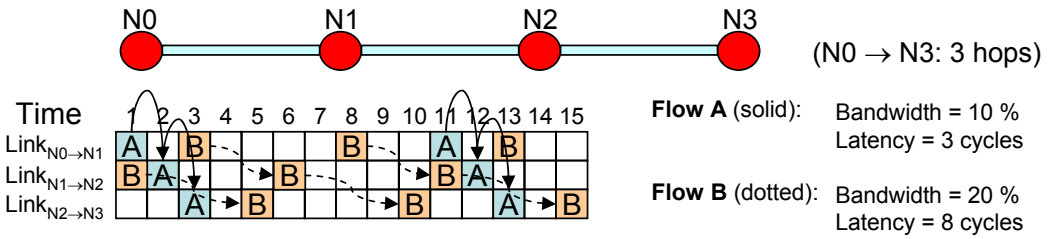


Figure 5-2: An example of having longer latency in spite of more bandwidth with a strict time-division multiplexing (TDM) scheduling.

completely owns the microarchitectural resources within the core. Consequently, there is no interference such as cache contamination between multiple threads that run on the same processor by time multiplexing. Combined with the previous assumption, this implies a one-to-one relationship between running processes and processors.

- We neither consider the performance variation from OS operations (e.g., flushing cache or branch predictor states after a context switch), nor address OS design issues (e.g., admission control, scheduling). A scheduled program runs to completion without preemption by other processes.

- We are not concerned about the performance variation coming from the processor's initial state, whose performance impact can be easily amortized over a program's longer execution time in most cases.

5.3.2 Safety-Tightness Tradeoff: Relaxed and Strict Enforcement Modes

We propose two types of enforcement mode, *relaxed* and *strict*, to reflect a tradeoff between *safety* and *tightness* of performance estimation. The estimated execution time should not be violated (safety), but be as close as possible to typical execution times (tightness). So far, we have accounted for the amount of allocated resources (i.e., bandwidth), but not the access latency to each QoS block. However, the execution time of a program can be dependent on latency as well, for example. If a program without much memory-level parallelism (MLP) [26] cannot overlap multiple cache miss requests to hide the latency of

an individual request.

Although higher bandwidth generally leads to lower average memory access latency, there are no guarantees for an individual memory request always taking fewer cycles than *any* requests with less bandwidth reserved. In Figure 5-2, we show an example of two network flows between Node 0 (N0) and Node 3 (N3) with a simple fixed frame-based scheduling. Although Flow B receives twice as much bandwidth as Flow A, Flow A has a lower latency because of its time slots across nodes are well aligned like synchronized traffic lights. This longer latency could lead to a longer execution time for a given instruction sequence. Therefore, we introduce strict enforcement mode to enforce constraints on both bandwidth and latency. Condition 1 below (Bandwidth) is met by both relaxed and strict enforcement modes, but Condition 2 (Latency) is met only in strict enforcement mode.

Condition 1. *For a given processor i and* $\forall j (1 \leq j \leq m)$*, and over any finite observation window,*

$$Bandwidth_{DEP}(x_{i,j}) \geq Bandwidth_{ENF}(x_{i,j})$$

Condition 2. *For a given processor i,* $\forall j (1 \leq j \leq m)$*, and any pair of corresponding memory requests in two modes,*

$$MAX\{Latency_{DEP}(x_{i,j})\} \leq MIN\{Latency_{ENF}(x_{i,j})\}$$

Assuming performance monotonicity of a processor, we can safely claim that the estimated performance lower bound in strict enforcement mode will not be violated in deployment mode. The estimated lower bound of execution time in strict enforcement mode is safer than that in the relaxed enforcement mode but is not as tight.

5.3.3 METERG QoS Memory Controller: An Example

We can build METERG QoS blocks meeting Condition 1 by slightly modifying conventional QoS blocks supporting per-processor resource reservation. The QoS scheduling algorithm for each time (scheduling) slot is quite simple as follows: (1) If the owner process of the slot has a ready request, the process takes the slot. (2) If not, the scheduler picks a

ready request in a round-robin fashion among processes in deployment mode. That is, a process in enforcement mode cannot participate in reclaiming unclaimed slots.

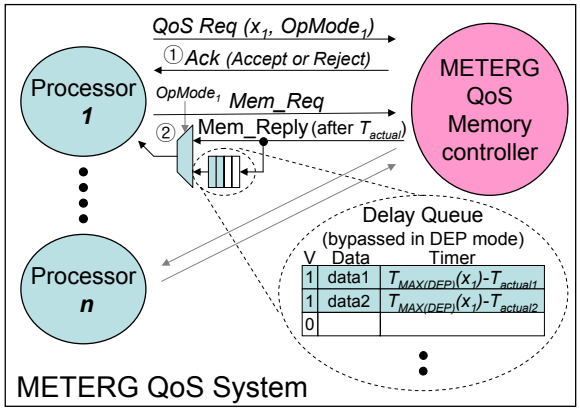
Latency guarantees are more difficult to implement than bandwidth guarantees. Fortunately, although conventional latency guarantees require a certain time bound not to be exceeded (absolute guarantee), the METERG system requires only the inequality relationship in Condition 2 to hold (relative guarantee). Given METERG QoS blocks capable of bandwidth guarantees, we can implement the METERG system supporting strict enforcement mode by inserting a delay queue between every QoS block-processor pair on the reply path.

Figure 5-3 depicts an example of a METERG QoS memory controller (possibly combined with a shared bus) supporting strict enforcement mode. For the rest of this chapter, we drop the QoS block identifier in resource reservation parameters, for we only have one shared resource for illustrative purposes, i.e., $x_i \equiv x_{i,j}$. Because the memory block is capable of resource reservation, we can assume that the latency to this block in deployment mode is upper bounded. The upper bound, $T_{MAX(DEP)}(x_1)$, is determined by the resource reservation parameter x_1. Note that the bounded latency is not guaranteed in enforcement mode, because the program receives no more resource than specified by x_1.

The delay queue is used only in enforcement mode; it is simply bypassed in deployment mode. If a memory access in enforcement mode takes T_{actual} cycles, which is smaller than $T_{MAX(DEP)}(x_1)$ due to lack of contention, the bus or network interface (NI) places the reply message in the delay queue with the entry's timer set to $T_{MAX(DEP)}(x_1) - T_{actual}$. The timer value gets decremented every cycle. The NI will defer signaling the arrival of the message to the processor until the timer expires. Hence, the processor's observed memory access latency is no smaller than $T_{MAX(DEP)}(x_1)$ and Condition 2 holds.

Because the processor needs to allocate a delay queue entry before sending a memory request in enforcement mode, a small delay queue may cause additional memory stall cycles. However, this does not affect the safety of the measured performance in enforcement mode, but only its tightness.

The memory access time under the worst-case contention for a given resource reservation parameter (x_1), $T_{MAX(DEP)}(x_1)$, depends on the hardware configuration and the



(Only the operations from Processor 1 are shown.)

Phase ①: Resource reservation phase
(Setting up a service level agreement)
Phase ②: Operation phase
(Requests are serviced with bandwidth/latency guarantees.)

Figure 5-3: An implementation of the METERG system supporting strict enforcement mode. We use delay queues to meet the latency condition (Condition 2) required by strict enforcement mode.

scheduling algorithm. In some systems, the latency lower bound is known or can be calculated [14, 43, 78]. For example, for Flow A in the setup shown in Figure 5-2, it is straightforward. Because it uses simple frame-based scheduling with ten time slots in each frame and at least one out of every ten is given to the process, each hop cannot take more than 10 time slots (if there is no buffering delay caused by a large buffer). Therefore, the worst-case latency between Node 0 and 3 will be 30 time slots. If the estimation process is not trivial, however, one may use an observed worst-case latency with some safety margin.

5.4 Evaluation

As a proof of concept, we present a preliminary evaluation of the METERG system. We evaluate the performance of a single-threaded application with different degrees of resource contention on our system-level multiprocessor simulator.

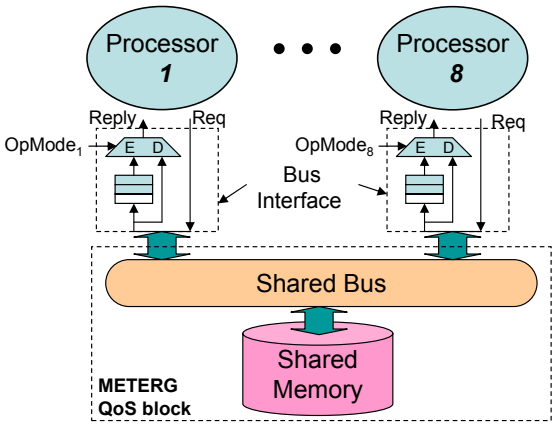


Figure 5-4: A simple bus-based METERG system. The memory controller is capable of strict METERG QoS.

5.4.1 Simulation Setup

We have added a strict METERG memory system to a full-system execution-driven multiprocessor simulator based on Bochs IA-32 emulator [37]. Figure 5-4 depicts the organization of our simulated platform. Although our simulator only supports the METERG QoS in the memory controller, we believe that the QoS support can be extended to other shared I/O devices (e.g., disk, network).

Our processor model is a simple in-order core capable of executing one instruction per cycle, unless there is a cache miss. A cache miss is handled by the detailed memory system simulator and takes a variable latency depending on the degree of contention. The processor's clock speed is five times faster than the system clock (e.g., 1 GHz processor with 200 MHz system bus, which is reasonable in embedded systems). The processor has a 32-KB direct-mapped unified L1 cache, but no L2 cache. The cache is blocking, so there can be at most one outstanding cache miss by any processor.

Our detailed shared memory model includes primary caches and a shared bus interconnect, and we have augmented it with the METERG QoS support. We have used a simple magic DRAM which returns the requested value in the next bus cycle; otherwise, long memory access latencies of the detailed DRAM, combined with our blocking caches, would make the memory bandwidth underutilized. Note that QoS support is meaningful only when there is enough resource contention. Severe memory channel contention

is possible in multiprocessor embedded systems where resources are relatively scarce and bandwidth requirements for applications (e.g., multimedia) are high.

The shared bus interconnect divides a fixed-size time frame into multiple *time slots*, which are the smallest units of bandwidth allocation, and implements a simple time division multiplexing (TDM) scheme. For example, if Processor 1 requests QoS with the resource allocation parameter (x_1) of 0.25, one out of every four time slots will be given to the processor. Hence, the access time is bounded by $\lceil 1/x_1 \rceil$=4 time slots in this case. An unclaimed time slot can be used by any other processors not in enforcement mode (work conserving).

We use a synthetic benchmark, called `memread`, for our evaluation to mimic the behavior of an application whose performance is bounded by the memory system performance. It runs an infinite loop which accesses a large memory block sequentially to generate cache misses, with a small amount of bookkeeping computation in each iteration.

5.4.2 Simulation Results

Performance Estimation

Figure 5-5 compares the performance of `memread` in various configurations with different operation modes ($OpMode$), degrees of contention, and resource allocation parameters (x_1). We use instructions per cycle (IPC) as our performance metric and all IPCs are normalized to the best possible case (denoted by `BE-1P`), where a single `memread` in best-effort mode monopolizes all the system resources. In best-effort (BE) mode, all processes run without any QoS support. In enforcement (ENF) or deployment (DEP) mode, only one process runs in QoS mode (either enforcement or deployment), and the remaining concurrent processes run in best-effort mode to generate memory contention. The figure depicts the single QoS process' performance.

In Figure 5-5(a), we first measure performance with varying degrees of resource contention. Without any QoS support (denoted by `BE`), we observe the end-to-end performance degradation of a single process by almost a factor of 2, when the number of concurrent processes executing `memread` increases from 1 (`BE-1P`) to 8 (`BE-8P`).

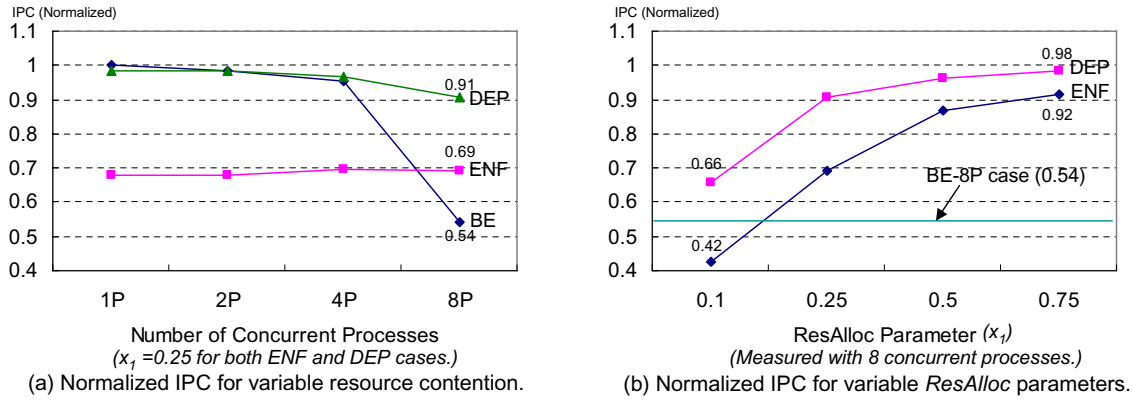


Figure 5-5: Performance of `memread` in various configurations. BE, ENF, and DEP stand for best-effort, enforcement-mode, and deployment-mode execution, respectively. In (a), as the number of concurrent processes increases from 1 to 8, the performance of a process degrades by 46 % without QoS support, but only by 9 % in deployment mode. The estimated performance from a measurement in strict enforcement mode indicates the performance degradation for the given resource reservation to be 31 % in the worst case. In (b), we observe that the performance estimation in strict enforcement mode becomes tighter as the resource allocation parameter (x_1) increases.

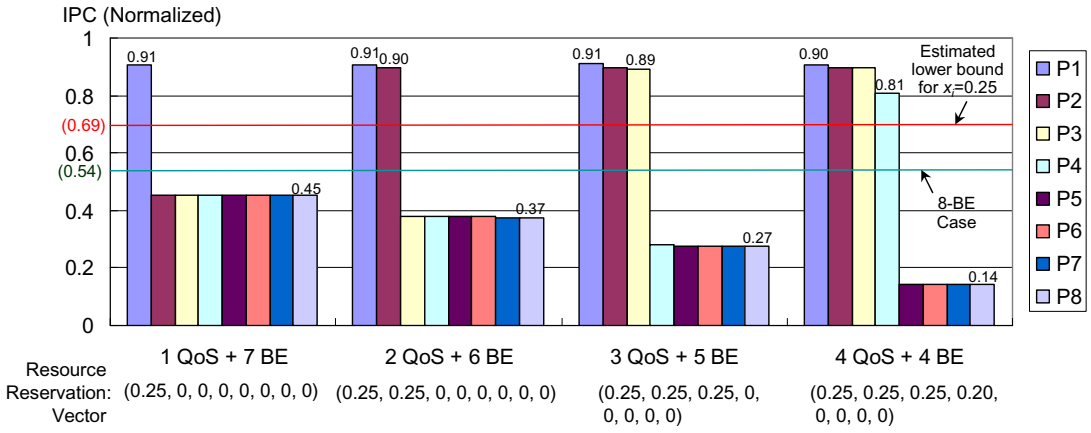


Figure 5-6: Interactions between QoS and Best-effort (BE) processes running `memread`. All QoS processes are running in deployment mode. Even if the number of QoS processes increases, the performance of QoS processes degrades very little as long as the system is not oversubscribed.

On the other hand, a QoS process in either enforcement or deployment mode is well protected from the dynamics of others. At any given time, a process in deployment mode always outperforms its counterpart in enforcement mode for a given resource allocation parameter (x_1). There is a significant performance gap between the two to give a safety margin for estimated performance. The performance gap can be explained by two factors. First, there is extra bandwidth given to the process in deployment mode, which would not be given in enforcement mode. Second, regardless of the actual severity of contention, every single memory access in enforcement mode takes the longest possible latency. Note that, although rare, this could happen in a real deployment-mode run. Hence, an enforcement-mode execution provides a safe and tight performance lower bound for a given x_1. Because a memory access in enforcement mode always takes the worst possible latency, there is little variation of the performance across 1 (`ENF-1P`) through 8 (`ENF-8P`) concurrent processes.

In Figure 5-5(b), we run the simulation with different resource reservation parameters. We observe that as we increase the parameter value, the performance gap between the two modes shrinks. This is because extra bandwidth beyond a certain point gives only a marginal performance gain to the QoS process in deployment mode, but improves the performance in enforcement mode significantly by reducing the worst-case memory access latency.

Interactions among Processes

Because we have dealt with a single QoS process so far, a question naturally arises about the interactions of multiple concurrent QoS processes. Figure 5-6 shows the performance variation when multiple QoS processes are contending against each other to access the shared memory.

The figure demonstrates that, even if we increase the number of QoS processes from one to four, the performance of QoS processes in deployment mode degrades very little (by less than 2 %) for a given parameter (x_i=0.25) and the performance lower bound estimated by an enforcement-mode execution is strictly guaranteed. The amount of reserved resource for each process is given in the resource allocation vector. Note that x_i=0 means no resource

is reserved for processor i (best-effort), and that we use x_4=0.20 rather than x_4=0.25 in the case of 4 QoS + 4 BE so as not to starve the best-effort processes.

As we increase the total amount of resources reserved for QoS processes, the performance of best-effort processes is degraded as expected. We observe that the system provides fairness so that their execution time differs only by less than 1 %. Fairness is also provided to the QoS processes having the same resource reservation parameter.

5.5 Summary

Although conventional QoS mechanisms effectively set a lower bound on a program's performance for a given resource reservation parameter, it is not easy to translate a user-level performance metric into a vector of hardware resource reservations. To facilitate performance estimation, we argue for having every QoS-capable component support two operation modes: enforcement mode for estimating end-to-end performance and deployment mode for maximizing performance with a guaranteed lower bound. Our approach does not involve any expensive program analysis or hardware modeling often required in conventional approaches for performance estimation. Instead, we use simple measurement. Its effectiveness is demonstrated by multiprocessor simulation with a bandwidth-intensive synthetic benchmark.

We discuss GSFM and METERG QoS system independently. However, an ideal QoS memory system we envision implements both GSFM and METERG QoS to provide guaranteed QoS and make it easy to find a minimal resource reservation for a given performance goal specified in a user metric. We might be able to implement enforcement mode in GSFM by strict rate control at an injection point, but further investigation is needed (e.g., the impact of running enforcement-mode and deployment-mode threads concurrently on the overall memory system throughput).

Chapter 6

Conclusion

The general-purpose computing industry has reached a consensus that its future lies in parallel computing to sustain its growth in the face of ever increasing demands for performance per watt [5]. Moore's Law will be translated into an increase in the number of cores on a single die, and highly parallel manycore systems will soon be the mainstream, not just in large machine room servers but also in small client devices, such as laptops, tablets, and handhelds [58]. The number of concurrent activities will continue increasing, and there is a growing need to support complex multiprogrammed workloads possibly composed of software components (e.g., virtual machines, shared libraries, applications, tasks) with diverse performance requirements. The importance of QoS support in such integrated platforms is increasing accordingly.

In this thesis, we have divided the solution space for QoS into four quadrants and made arguments for frame-based, end-to-end approaches to minimize hardware usage in a resource-constrained on-chip environment. We have proposed the GSF framework as an instance and first applied it to an on-chip network. We have extended the framework into a cache-coherent shared memory system to provide end-to-end QoS in handling cache misses which originated at the last-level private cache (LLPC). Finally, we have proposed the METERG QoS framework to bridge the gap between user-level QoS objectives and QoS parameter settings. We believe a system supporting this types of QoS will be commonplace on future CMPs.

6.1 Summary of Contributions

GSF, the main contribution of this thesis, takes an frame-based, end-to-end approach to QoS. Frame-based approaches eliminate per-flow (or per-thread) queues and reduce the complexity of scheduling logic at cost of discarding a precise ordering among requests, thereby leading to a longer latency upper bound. End-to-end approaches potentially reduce hardware usage further by offloading part of QoS scheduling operations (e.g., priority assignment and computation) from each scheduling node, but are not always feasible due to synchronization and policing costs.

We first apply GSF to an on-chip network. To summarize, the GSF on-chip network we propose is:

- **Guaranteed QoS-capable in terms of minimum guaranteed bandwidth and maximum delay.** The GSF on-chip network provides not only proportional bandwidth sharing but also guaranteed QoS. GSF can support best-effort traffic with little performance loss.

- **Flexible.** Both fair and differentiated bandwidth allocations are supported. When a new flow is admitted into the network, no explicit channel setup along the path is required; only the injection node needs to be reconfigured. Therefore, short-lived flows are not penalized.

- **Robust.** By controlling the frame size in the network, we can make a tradeoff between the granularity of bandwidth allocation and the overall network throughput. According to our simulation in a 8×8 mesh network with various traffic patterns, the degradation of the network saturation throughput is less than 5 % on average (with 12 % in the worst).

- **Simple.** We show that the GSF algorithm can be easily implemented in a conventional VC router without significantly increasing its complexity. This is possible because the complex task of prioritizing packets to provide guaranteed QoS is pushed out to the source injection process at the end-points. The end-points and network are

globally orchestrated by a fast barrier network made possible by the on-chip implementation.

GSF memory system (GSFM) extends one-dimensional GSF in an on-chip network into multi-dimensional GSF so that GSFM can handle multiple heterogeneous bandwidth resources—network links, DRAM channels, and DRAM banks—in a single unified QoS framework. The main contributions of GSFM are summarized as follows:

- *GSFM is the first to provide end-to-end QoS guarantees across the entire cache-coherent memory hierarchy for CMPs beyond component-wise QoS from individual building blocks such as caches, memory controllers and interconnects.* GSFM provides both guaranteed minimum bandwidth for a high-throughput processor and proportional sharing of excess bandwidth. GSFM can reduce the throughput ratio of most-over-least served threads from over 43 to 1.02 in one case and from 16.6 to 1.18 on average over four microbenchmarks.

- *GSFM shows the feasibility of low-cost QoS support in a CMP shared memory system without any significant degradation of the overall memory system throughput using global frame-based queueing and scheduling.* GSFM simplifies hardware by pushing out the complex task of prioritizing messages to end-points to avoid burdening each arbitration point with complex QoS mechanisms. The overall throughput degradation caused by the QoS support is almost negligible, and the throughput degradation of the individual thread is less than 0.4% on average compared to a best-effort memory system.

Finally, we propose the METERG QoS framework with which we argue for having every QoS-capable component support two operation modes to facilitate performance estimation: enforcement mode for estimating end-to-end performance and deployment mode for maximizing performance with a guaranteed lower bound. Although conventional QoS mechanisms effectively set a lower bound on a program's performance for a given resource reservation parameter, it is not easy to translate a user-level performance metric into a vector of hardware resource reservations. Our approach does not involve any expensive

program analysis or hardware modeling often required in conventional approaches for performance estimation. Instead, we use simple measurement. As proof of concept, we have implemented a multiprocessor system simulator and estimated the lower bound on the execution time of a bandwidth-intensive synthetic benchmark.

6.2 Future Work

Recently, QoS support for multicores has become an active research area in the field of computer architecture. Avenues for further research are plentiful. A partial list of future work is as follows:

Evaluation of GSFM in Other CMP Configurations

Although our results with GSFM look promising, more sensitivity analysis of GSFM should be performed with different topologies (e.g., a ring topology as in Intel's Larrabee [82]) and system sizes. The scalability of GSFM with an increasing number of cores should be better understood.

Finding an optimal set of GSFM parameters for a given system configuration is a challenge yet to be addressed. A systematic way can be devised to determine GSFM parameters such as frame window size (W), frame interval (F), and source queue size to achieve high memory system throughput while keeping the hardware cost minimal.

Optimization of GSFM

There are several optimization ideas to boost the overall memory system throughput and/or reduce the hardware complexity of GSFM without degrading QoS. *Sender-side request coalescing* is an example. When requests to shared memory are stalled because of using up all injection tokens, a source injection control can coalesce multiple requests to adjacent addresses into one jumbo request to reduce network traffic and simplify a memory access scheduler.

Another example is *token refund*. The worst-case billing of GSFM at injection time potentially leads to the low utilization of memory system if the number of overlapping future frames is not large enough. The idea of token refund is that a reply message delivers the precise resource usage information of the original requester, and that an injection control recharge injection credit tokens accordingly. Design tradeoffs with these optimization ideas need to be investigated further.

METERG Implementation on GSFM and Other Shared Resources

In this thesis we discuss GSFM and METERG QoS system independently. It is an open question whether and how GSFM can implement the METERG QoS. We might be able to implement enforcement mode with strict rate control at an injection point, but further investigation with detailed simulation is needed. An ideal QoS memory system we envision implements both GSFM and METERG QoS to provide guaranteed QoS and make it easy to find a minimal resource reservation for a given performance goal specified in a user metric.

Furthermore, it is feasible to apply the METERG methodology to other shared resources such as I/O devices, special-purpose accelerators, and microarchitectural resources (e.g., instruction issue logic and functional units in a multi-threaded core).

QoS-Capable OS/Hypervisor Design

There are a lot of design issues of software stack on top of GSFM such as admission control, APIs for resource reservation, and reasonable performance abstractions for programmers. There is a significant gap between a user's performance specification and QoS resource reservation parameters. Although current programming models provide a measure for program correctness, there is no corresponding performance model with which a user or programmer can reason about his programs performance behavior for a given resource reservation setting.

An adaptive QoS resource scheduler to maximize the system throughput without breaking QoS guarantees to concurrent processes is another research agenda. One interesting idea would be exploiting performance counters to characterize an application and/or to

implement online feedback mechanisms to assist scheduling decisions.

In a METERG QoS system, we assume a simple non-preemptive process scheduler. It is still unclear how a METERG QoS system can provide performance guarantees to an application (e.g., meeting a deadline) across multiple scheduling quanta with a preemptive process scheduler. To improve the usability of METERG QoS, this issue needs to be addressed in the future.

Bibliography

[1] Dennis Abts and Deborah Weisser. Age-based packet arbitration in large-radix k-ary n-cubes. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[2] Anant Agarwal and Markus Levy. The KILL rule for multicore. In *DAC '07: Proceedings of the 44th annual conference on Design Automation*, pages 750–753, New York, NY, USA, 2007. ACM.

[3] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil, Eric Rotenberg, and Frank Mueller. Virtual simple architecture (VISA): Exceeding the complexity limit in safe real-time systems. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 350–361, New York, NY, USA, 2003. ACM Press.

[4] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding Worst-case Instruction Cache Performance. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, December 1994.

[5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] Hari Balakrishinan. Lecture 8: Scheduling for fairness: Fair queueing and CSFQ. Lecture Notes for MIT 6.829: Computer Networks, 2005.

[7] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 processor: A 64-core SoC with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb. 2008.

[8] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 143–156, New York, NY, USA, 1996. ACM.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and Compilation Techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[10] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1):1, 2006.

[11] Tobias Bjerregaard and Jens Sparso. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1226–1231, Washington, DC, USA, 2005. IEEE Computer Society.

[12] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An architecture for differentiated service. RFC Editor, 1998.

[13] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *J. Syst. Archit.*, 50(2-3):105–128, 2004.

[14] Brian Case. First Trimedia chip boards PCI bus. *Microprocessor Report*, Nov 1995.

[15] Francisco J. Cazorla, Alex Ramírez, Mateo Valero, Peter M. W. Knijnenburg, Rizos Sakellariou, and Enrique Fernández. QoS for High-Performance SMT Processors in Embedded Systems. *IEEE Micro*, 24(4):24–31, 2004.

[16] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA '06: Proceedings of the 33rd annual International Symposium on Computer Architecture*, pages 264–276, Washington, DC, USA, 2006. IEEE Computer Society.

[17] David D. Clark and Wenjia Fang. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Trans. Netw.*, 6(4):362–373, 1998.

[18] Rene L. Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[19] W. J. Dally. Virtual-channel flow control. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):194–205, 1992.

[20] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[21] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications Architectures & Protocols*, pages 1–12, New York, NY, USA, 1989. ACM.

[22] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[23] Ali El-Haj-Mahmoud and Eric Rotenberg. Safely exploiting multithreaded processors to tolerate memory latency in real-time systems. In *CASES '04: Proceedings of the 2004 international conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 2–13, New York, NY, USA, 2004. ACM Press.

[24] T. Felicijan and S. B. Furber. An asynchronous on-chip network router with quality-of-service (QoS) support. In *Proceedings IEEE International SOC Conference*, 2004.

[25] Mike Galles. Spider: A high-speed network interconnect. *IEEE Micro*, 17(1):34–39, 1997.

[26] Andy Glew. MLP yes! ILP no! ASPLOS Wild and Crazy Idea Session, 1998.

[27] S. J. Golestani. A stop-and-go queueing framework for congestion management. *SIGCOMM Comput. Commun. Rev.*, 20(4):8–18, 1990.

[28] S.J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOM '94. 13th Annual Joint Conference of the IEEE Computer and Communication Societies. Proceedings., IEEE*, pages 636–646 vol.2, Jun 1994.

[29] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, 2005.

[30] Fei Guo, Hari Kannan, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. From chaos to QoS: case studies in CMP resource management. *SIGARCH Comput. Archit. News*, 35(1):21–30, 2007.

[31] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Christopher A. Healy, Robert D. Arnold, Frank Mueller, Marion G. Harmon, and David B. Walley. Bounding Pipeline and Instruction Cache Performance. *IEEE Trans. Comput.*, 48(1):53–70, 1999.

[33] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. Rate-based QoS techniques for cache/memory in CMP platforms. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, pages 479–488, New York, NY, USA, 2009. ACM.

[34] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *PACT '06: Proceedings of the 15th international conference on Parallel Architectures and Compilation Techniques*, pages 13–22, New York, NY, USA, 2006. ACM.

[35] Ravi Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS '04: Proceedings of the 18th annual International Conference on Supercomputing*, pages 257–266, New York, NY, USA, 2004. ACM.

[36] Rohit Jain, Christopher J. Hughes, and Sarita V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *IEEE Real-Time Systems Symposium*, 2002.

[37] K. Lawton. Bochs: The cross-platform IA-32 emulator. `http://bochs.sourceforge.net`.

[38] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March/April 2004.

[39] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[40] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March/April 2003.

[41] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, New York, NY, USA, 2002. ACM.

[42] Jae H. Kim. *Bandwidth and Latency Guarantees in Low-Cost, High-Performance Networks*. PhD thesis, University of Illinois, Urbana-Champaign, 1997.

[43] Jae H. Kim and Andrew A. Chien. Rotating Combined Queueing (RCQ): Bandwidth and latency guarantees in low-cost, high-performance networks. In *ISCA '96: Proceedings of the 23rd annual International Symposium on Computer Architecture*, pages 226–236, New York, NY, USA, 1996. ACM.

[44] John Kim, William J. Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. *SIGARCH Comput. Archit. News*, 35(2):126–137, 2007.

[45] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[46] Ronny Krashinsky. *Vector-Thread Architecture and Implementation*. PhD thesis, Massachusetts Institute of Technology, 2007.

[47] K. Krewell. Sun's Niagara pours on the cores. *Microprocessor Report*, 18(9):11–13, September 2004.

[48] Kevin Krewell. ARM opens up to SMP. *Microprocessor Report*, May 2004.

[49] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express virtual channels: towards the ideal interconnection fabric. In *ISCA '07: Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 150–161, New York, NY, USA, 2007. ACM.

[50] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-core chip multiprocessing. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 195–206, Washington, DC, USA, 2004. IEEE Computer Society.

[51] Jae W. Lee and Krste Asanović. METERG: Measurement-based end-to-end performance estimation technique in QoS-capable multiprocessors. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 135–147, Washington, DC, USA, 2006. IEEE Computer Society.

[52] Jae W. Lee, Sarah Bird, and Krste Asanovic. An end-to-end approach to quality-of-service in a shared memory system. In preparation for external publication.

[53] Jae W. Lee, Man Cheuk Ng, and Krste Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 89–100, Washington, DC, USA, 2008. IEEE Computer Society.

[54] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 132–140, New York, NY, USA, 1975. ACM.

[55] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS)*, pages 97–108, 1994.

[56] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society.

[57] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[58] Rose Liu et al. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the first Workshop on Hot Topics in Parallelism (HotPar)*, Berkeley, CA, 2009.

[59] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, September 2005.

[60] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. In *ISCA '07: Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 46–56, New York, NY, USA, 2007. ACM.

[61] P.E. McKenney. Stochastic fairness queueing. In *INFOCOM '90. Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. Proceedings., IEEE*, pages 733–740 vol.2, Jun 1990.

[62] Nick McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Trans. Netw.*, 7(2):188–201, 1999.

[63] Micron Inc. DDR2-800 Datasheet. Part number MT47H128M8. `http://www.micron.com`.

[64] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, page 20890, Washington, DC, USA, 2004. IEEE Computer Society.

[65] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.

[66] Ronald Mraz. Reducing the variance of point-to-point transfers for parallel real-time programs. *IEEE Parallel Distrib. Technol.*, 2(4):20–31, 1994.

[67] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160, Washington, DC, USA, 2007. IEEE Computer Society.

[68] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.

[69] N. Kushman. Performance nonmonotonicities: A case study of the ultraSPARC processor. Technical Report MIT/LCS/TR-782, 1998.

[70] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

[71] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. In *ISCA '07: Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 57–68, New York, NY, USA, 2007. ACM.

[72] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private machines: A resource abstraction for multicore computer systems. In *University of Wisconsin - Madison, ECE TR 07-08*, 2007.

[73] Kyle J. Nesbit, Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and James E. Smith. Multicore resource management. *IEEE Micro*, 28(3):6–16, 2008.

[74] Y. Ofek and M. Yung. The integrated Metanet architecture: a switch-based multimedia LAN for parallel computing and real-time traffic. In *INFOCOM '94. 13th Annual Joint Conference of the IEEE Computer and Communication Societies. Proceedings., IEEE*, pages 802–811 vol.2, Jun 1994.

[75] Marco Paolieri, Eduardo Qui nones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA '09: Proceedings of the 36th annual International Symposium on Computer Architecture*, pages 57–68, New York, NY, USA, 2009. ACM.

[76] Abhay K. Parekh and Robert G. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Trans. Netw.*, 2(2):137–150, 1994.

[77] Randal S. Passint, Gregory M. Thorson, and Timothy Stremcha. United States Patent 6674720: Age-based network arbitration system and method, January 2004.

[78] L. L. Peterson and B. S. Davie. *Computer Networks: a Systems Approach*. Morgan Kaufmann Publishers Inc., 2003.

[79] Timothy Mark Pinkston and Jose Duato. *Interconnection Networks, Computer Architecture: A Quantitative Approach 4th Edition, Appendix E*. Morgan Kaufmann Publishers Inc., 2003.

[80] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 245–258, Washington, DC, USA, 2007. IEEE Computer Society.

[81] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 128–138, New York, NY, USA, 2000. ACM.

[82] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[83] Scott Shenker. Fundamental design issues for the future Internet (invited paper). *IEEE Journal on Selected Areas in Communications*, 13(7):1176–1188, 1995.

[84] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM '95: Proceedings of the conference on applications, technologies, architectures, and protocols for computer communication*, pages 231–242, New York, NY, USA, 1995. ACM.

[85] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES*

'01: Proceedings of the 2001 international conference on Compilers, Architecture, and Synthesis for Embedded Systems, pages 132–140, New York, NY, USA, 2001. ACM Press.

[86] Dimitrios Stiliadis and Anujan Varma. Design and analysis of frame-based fair queueing: a new traffic scheduling algorithm for packet-switched networks. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 104–115, New York, NY, USA, 1996. ACM.

[87] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[88] J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems*, Washington, DC, USA, 1996. IEEE Computer Society.

[89] Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, Washington, DC, USA, 1999. IEEE Computer Society.

[90] Mithuna Thottethodi, Alvin R. Lebeck, and Shubhendu S. Mukherjee. Self-tuned congestion control for multiprocessor networks. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 107, Washington, DC, USA, 2001. IEEE Computer Society.

[91] Brian Towles and William J. Dally. Booksim 1.0. `http://cva.stanford.edu/books/ppin/`.

[92] Jonathan S. Turner. New directions in communications (or which way to the information age). *IEEE Communications*, 24(10):8–15, October 1986.

[93] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192, New York, NY, USA, 1998. ACM.

[94] Virtutech Inc. Simics 3.0 multiprocessor simulator. `http://www.simics.net`.

[95] Wolf-Dietrich Weber, Joe Chou, Ian Swarbrick, and Drew Wingard. A quality-of-service mechanism for interconnection networks in system-on-chips. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1232–1237, Washington, DC, USA, 2005. IEEE Computer Society.

[96] Ki Hwan Yum, Eun Jung Kim, Chita R. Das, and Aniruddha S. Vaidya. MediaWorm: A QoS capable router architecture for clusters. *IEEE Trans. Parallel Distrib. Syst.*, 13(12):1261–1274, 2002.

[97] Hui Zhang and Srinivasan Keshav. Comparison of rate-based service disciplines. In *SIGCOMM '91: Proceedings of the conference on Communications Architecture & Protocols*, pages 113–121, New York, NY, USA, 1991. ACM.

[98] L. Zhang. Virtual Clock: a new traffic control algorithm for packet switching networks. In *SIGCOMM '90: Proceedings of the ACM symposium on Communications Architectures & Protocols*, pages 19–29, New York, NY, USA, 1990. ACM.

[99] Michael Zhang and Krste Asanovic. Victim migration: Dynamically adapting between private and shared CMP caches. Technical Report MIT-CSAIL-TR-2005-064, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, October 2005.

[100] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA '05: Proceedings of the 32nd annual International Symposium on Computer Architecture*, pages 336–345, Washington, DC, USA, 2005. IEEE Computer Society.

[101] N Zhang, A Burns, and M Nicholson. Pipelined processors and worst case execution times. Technical Report University of York - CS 198, 1993.