UNIVERSITY OF CALIFORNIA SAN DIEGO

On Quality-of-Service and Memory Efficiency in Heterogeneous MPSoCs

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

in

Electrical Engineering (Computer Engineering)

by

Yang Song

Committee in charge:

Professor Bill Lin, Chair
Professor Gert Cauwenberghs
Professor Chung-Kuan Cheng
Professor Ryan Kastner
Professor Alon Orlitsky

2019

The dissertation of Yang Song is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California San Diego

2019

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Professor Bill Lin, for his guidance and support during the last few years. I have very much appreciated his enlightening commentary during hours of discussions on our projects and in the realm of politics. I have also appreciated his efforts to provide financial support and his kind understanding of the difficulties faced by living on university stipends.

I also owe Olivier Alavoine a great debt of appreciation, particularly for his patience as a mentor, his generosity in sharing industry savvy, and his personal wisdom. I cannot thank him enough for his help.

I would also like to thank Kambiz Samadi and Qualcomm Research. Our collaboration during the Qualcomm FMA fellowship program laid the foundation for my PhD research later on.

I would like to thank my doctoral committee members, including Professor Gert Cauwenberghs, Professor Chung-Kuan Cheng, Professor Ryan Kastner, and Professor Alon Orlitsky for their advice and supervision. Doubtless the end product of my dissertation is better because of their involvement.

Last but not the least, I would like to thank Renier Milan for his help in proofreading my manuscripts over the years. I am grateful for his support and encouragement during my studies. Equally, I would also like to thank Emmy and Ricky, my two puppies, for their unconditional love and constant, unwavering (sometimes even annoying) attention during the preparation of my dissertation.

Chapter 3, in part, has been submitted for publication of the material as it may appear in ACM Transactions on Architecture and Code Optimization, 2019. Song, Yang; Alavoine, Olivier; Lin, Bill. Chapter 3 also includes the material as it appears in proceedings of ACM/IEEE Design Automation Conference, 2018. Song, Yang; Alavoine, Olivier; Lin, Bill. The dissertation author was the primary investigator and author of these papers.

Chapter 4, in part, is a reprint of the material as it appears in ACM Transactions on Design Automation of Electronics Systems, 2017. Song, Yang; Samadi, Kambiz; Lin, Bill. Chapter 4 also includes the material as it appears in proceedings of ACM/IEEE Design Automation Conference, 2016. Song, Yang; Samadi, Kambiz; Lin, Bill. The dissertation author was the primary investigator and author of these papers.

Chapter 5, in part, is a reprint of the material as it appears in ACM Transactions on Design Automation of Electronics Systems, 2019. Song, Yang; Alavoine, Olivier; Lin, Bill. Chapter 5 also includes the material as it appears in proceedings of ACM/IEEE Design Automation and Test in Europe, Conference and Exhibition, 2018. Song, Yang; Alavoine, Olivier; Lin, Bill. The dissertation author was the primary investigator and author of these papers.

Chapter 6, in part, is currently being prepared for submission for publication of the material. Song, Yang; Lin, Bill. The dissertation author was the primary investigator and author of this paper.

VITA

2010	Bachelor of Science in Electrical Engineering Shanghai Jiao Tong University
2013	Master of Science in Electrical Engineering Shanghai Jiao Tong University
2019	Doctor of Philosophy in Electrical Engineering (Computer Engineering) University of California San Diego

PUBLICATIONS

Yang Song, Kambiz Samadi, Bill Lin, "Single-Tier Virtual Queuing: An Efficacious Memory Controller Architecture for MPSoCs with Multiple Realtime Cores," ACM/IEEE Design Automation Conference (DAC), 6:1-6:6 (2016).

Yang Song, Kambiz Samadi, Bill Lin, "A Single-Tier Virtual Queuing Memory Controller Architecture for Heterogeneous MPSoCs," ACM Transactions on Design Automation of Electronics Systems (TODAES), 22(3): 56:1-56:23 (2017).

Yang Song, Olivier Alavoine, Bill Lin, "Row-Buffer Hit Harvesting in Orchestrated Last-Level Cache and DRAM Scheduling for Heterogeneous Multicore Systems," Design Automation and Test in Europe, Conference and Exhibition (DATE), 779-784 (2018).

Yang Song, Olivier Alavoine, Bill Lin, "Harvesting Row-Buffer Hits via Orchestrated Last-Level Cache and DRAM Scheduling for Heterogeneous Multicore Systems," ACM Transactions on Design Automation of Electronics Systems (TODAES), 5:1-5:27 (2018).

Yang Song, Olivier Alavoine, Bill Lin, "SARA: Self-Aware Resource Allocation for Heterogeneous MPSoCs," ACM/IEEE Design Automation Conference (DAC), 113:1-113:6 (2018).

Yang Song, Bill Lin, "Uniform Minimal First: Latency Reduction in Throughput-Optimal Oblivious Routing for Mesh-Based Networks-on-Chip," IEEE Embedded Systems Letters (ESL), (2019).

Yang Song, Olivier Alavoine, Bill Lin, "A Self-Aware Resource Management Framework for Heterogeneous Multicore SoCs with Diverse QoS Targets," ACM Transactions on Architecture and Code Optimization (TACO), under review.

Yang Song, Bill Lin, "Improving Memory Efficiency in Heterogeneous MPSoCs through Row-Buffer Locality-Aware Forwarding," ACM Transactions on Architecture and Code Optimization (TACO), under review.

ABSTRACT OF THE DISSERTATION

On Quality-of-Service and Memory Efficiency in Heterogeneous MPSoCs

by

Yang Song

Doctor of Philosophy in Electrical Engineering (Computer Engineering)

University of California San Diego, 2019

Professor Bill Lin, Chair

Due to their energy efficiency, heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) are widely deployed as the engines for modern-day smart devices. As smart devices prevail, heterogeneous MPSoCs have become ubiquitous in our daily lives. These MPSoCs typically integrate a diverse collection of cores, including real-time agents such as the GPU, the DSP and video codec, as well as general-purpose cores such as the CPU. Different from traditional multi-core processors where memory sharing is mostly handled by the cache hierarchy, a heterogeneous system mainly uses DRAM as the medium for data sharing. That makes the memory subsystem, including networks-on-chip (NoC), last-level

caches and memory controllers, a frequently shared resource among heterogeneous cores. As traffic streams travel through the shared memory subsystem, memory interference is inevitable, which is even worsened by their disparate traffic patterns. As a result, the memory subsystem plays a crucial role in heterogeneous systems and has tremendous impact on system performance. In particular, the design goal of the memory subsystem is two-fold: delivering end-to-end Quality-of-Service (QoS) to heterogeneous cores and improving memory efficiency in DRAM.

In this dissertation, we explore several design aspects of the memory subsystem for heterogeneous MPSoCs. First, we propose a self-aware resource allocation framework to enable distributed performance monitoring, priority-based adaptation and instant memory response. The proposed framework meets a diverse range of QoS demands from real-time cores. It can also configure itself in runtime to accommodate best-effort cores and real-time cores that are particularly prone to QoS failure. In the rest of the dissertation, we further look into each major memory fabric. We present the single-tier virtual queuing memory controller that maintains a single tier of request queues and employs an efficacious scheduler that considers both QoS requirements and DRAM bank states. Then we study the orchestration between last-level cache controller and memory scheduler to improve memory visibility and avoid unnecessary precharges and row activations in DRAM. At last, we introduce a locality-aware NoC router to prevent row-buffer locality dilution while providing QoS-aware service to heterogeneous cores.

Chapter 1

Introduction

In the first few decades of semiconductor history, numerous observations on developing technological advancements were made to predict future trends at a similar pace. Among these observations, unquestionably the best-known is Moore's Law. However, lesser known than Moore's Law, but equally influential, Dennard scaling states that, as transistors get smaller, both voltage and current scale down with length so that power density stays constant [12]. These observations provided guidelines for semiconductor roadmaps in early decades.

As Moore's Law continues at a slower speed in this millennium, we have entered a new era where we cannot further scale down threshold voltage without exponentially increasing leakage. As the result, we must hold operating voltage roughly constant, which indicates the breakdown of Dennard scaling and the beginning of dark silicon [50].

In the dark silicon age, an increasing portion of chip area is turned off or underclocked to limit power density. Chip designs are no longer constrained by their area overhead, but the energy cost. To cope with the new reality, the industry first developed multicore processors to increase transistor count without exceeding power budget. However, multicore processors fail to extract more performance from the limited watts per unit area. As the

energy efficiency of chip designs became increasingly important to the industry, a wide variety of heterogeneous multicore systems emerged. In addition to general-purpose cores, heterogeneous systems also include various accelerators optimized for different workloads, such as multimedia applications. Compared with general-purpose cores, these accelerators are much more efficient in terms of performance per watt. Program execution migrates between general-purpose cores and accelerators to minimize energy cost.

Smart mobile devices have stringent energy budgets, the restrictions of which make heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) an ideal solution. The chronicle of smart devices in the last decade is also the evolutionary history of heterogeneous MPSoCs. The industry has seen fierce competition among leading heterogeneous platforms, including Qualcomm Snapdragon [40], Apple A-series [57], Nvidia Tegra [36] and others.

Driven by market demands and process technology, more and more heterogeneous cores are introduced to heterogeneous MPSoCs. As an example, Snapdragon 845 [39], in addition to the CPU, also has various real-time cores, including major agents such as the GPU and the DSP, multimedia cores (such as display control, video codec etc.), and system cores (such as the GPS, cellular modem etc.). One of the latest trends is the integration of a processing unit dedicated to deep learning applications, such as the Neural Engine in Apple A12 Bionic [56] and the NPU in HiSilicon Kirin 970 [55]. In the foreseeable future, heterogeneous MPSoCs will continue to prevail with an increasingly diverse collection of heterogeneous cores.

Apart from the energy benefit, the vast disparity among heterogeneous cores also presents new challenges to chip design. One of the major challenges is memory sharing traditionally conducted through the cache hierarchy in multicore processors. However, unlike the CPU, most heterogeneous cores generate memory access requests in low temporal or low spatial locality thus making caches ineffective. As a result, memory sharing is handled by the main memory, a process that requires heavy involvement from all memory

fabrics. In addition, the new way of memory sharing inevitably exposes traffic streams to each other, subjecting them to memory interference and forcing the memory subsystem to play a more crucial role than ever in heterogeneous MPSoCs.

User experience and battery life are two frequently used performance metrics for smart devices. Similarly, the memory subsystem is also evaluated in two ways: Quality-of-Service (QoS) and memory efficiency. Defined on a per-core basis, QoS refers to differentiated services for heterogeneous cores from the memory subsystem. Additionally, due to physical limitations, DRAM access requires special care from the memory subsystem to avoid energy and time penalties. However, the delivery of QoS and the improvement on memory efficiency can be conflicting objectives, as latency-sensitive memory traffic may not always be friendly to memory efficiency.

Over the last few years, academia has noticed the design challenges and the research opportunities presented by the new demands on the memory subsystem. Nonetheless, heterogeneous MPSoCs remains a generally under explored topic, particularly when compared to the volume of research effort on multicore processors. Apart from being a relatively new area, heterogeneous platforms are proprietary to the developing companies and thus the research has been conducted primarily by the industry. Thanks to our collaboration with Qualcomm, in this dissertation we will look into several aspects of memory subsystem design using Snapdragon as the prototype platform for many of the evaluations.

1.1 Contributions and Organization

The contributions of this dissertation include a memory resource allocation framework and architectural designs of several memory fabrics including the memory controller, the last-level cache and Network-on-Chip (NoC) routers. Note that, as this dissertation contains materials that have been published separately, discrepancy in evaluation methodology

is inevitable among different chapters.

First, we propose the self-aware resource allocation framework for heterogeneous MPSoCs. Intended for multicore processors, prior memory management frameworks induce high computational complexity and only apply to partitionable resources. On the other hand, previous QoS-aware memory fabric designs fail to provide guarantees for end-to-end QoS. In the proposed framework, we apply priority-based adaptation to allow heterogeneous cores to customize performance targets and monitor their own intrinsic health. In response, the system allocates non-partitionable resources based on priorities. Our framework meets a diverse range of QoS demands from heterogeneous cores. Furthermore, we present a runtime scheme to configure priority-based adaptation so that best-effort cores and real-time cores that are particularly prone to QoS failure can both be accommodated accordingly.

Following the resource allocation framework, we continue our research onto architectural designs of the memory subsystem. Previous QoS-aware memory controllers are based on a two-tier queuing architecture that buffers memory transactions and DRAM commands separately. In those designs, QoS-aware policies are applied in the first stage to schedule transactions to the second stage where DRAM commands are served in the first-in-first-out order. Unfortunately, once the scheduled transactions have been forwarded to the command stage, newly arriving transactions that may be more critical cannot be served ahead of those that are already queued in the second stage. To address this pitfall, we propose a scalable memory controller that maintains a single tier of request queues to avoid extra queuing delay after the scheduler, and employs an efficacious scheduler that considers both QoS requirements and DRAM bank states.

For the sake of memory coherence, the last-level cache is often shared among heterogeneous cores, which inevitably leads to more cache misses. As cache misses travel to DRAM, two schedulers are involved respectively in the last-level cache and the memory controller. Conventional designs treat the arbitrations of cache requests and memory

requests as two independent events. However, without orchestration, neither scheduling stage is able to ensure optimal scheduling decisions for memory efficiency. Unnecessary precharges and row activations happen in DRAM while the memory scheduler is unaware of incoming cache misses and DRAM row-buffer states are invisible to the last-level cache. To overcome the limitation, we propose a unified controller with orchestrated schedulers for the last-level cache and DRAM. The memory scheduler harvests row-buffer hit opportunities in cache request buffers during spare time without inducing significant overhead. We further introduce a dynamic orchestrated scheduling policy to improve memory efficiency while achieving target CPU IPC.

Before arriving at the last-level cache and DRAM, memory requests traverse through the NoC from heterogeneous cores. We observe that as memory traffic streams are merged in the network, spatial locality is subject to dilution, resulting in curbed row-buffer hit rate in DRAM. Previous research have discussed the techniques that improve memory visibility to reveal scattered row-buffer hit opportunities to the memory scheduler. However, extending local memory visibility introduces little benefit after the locality has been severely diluted. As the alternative approach, preserving row-buffer locality in the NoC has not been well explored. Therefore, we propose a router design with embedded row-index caches to enable locality-aware packet forwarding. The proposed design requires minor modifications to existing router micro-architecture and can be easily implemented with priority arbiters to integrate quality-of-service support.

The rest of the dissertation is organized as follows:

- Chapter 2 presents an overview of the research background, including memory subsystem basics and DRAM access protocol.

- Chapter 3 presents our self-aware resource allocation framework for heterogeneous cores with distinct notions of QoS sharing non-partitionable resources.

- Chapter 4 presents our single-tier virtual queuing memory controller for efficacious QoS-aware memory scheduling.
- Chapter 5 presents our unified controller for orchestrated last-level cache and DRAM scheduling.
- Chapter 6 presents our locality-aware NoC router to preserve row-buffer locality in memory traffic.
- Chapter 7 concludes this dissertation.

Chapter 2

Background

In this chapter, we will have an overview of memory subsystem basics. Specifically, we will go over principal memory fabrics in the traversal order of memory traffic, and the memory access protocol in DRAM. We will cover the background information to the extent that is necessary for readers to understand the rest of this dissertation. In-depth discussions on memory subsystem design are carried out in the following chapters.

2.1 Memory Subsystem Formation

The classic architecture of a memory subsystem shared by heterogeneous cores is shown in Fig. 2.1. Before accessing DRAM, a memory request travels through three stages in the memory subsystem, including the network-on-chip, the last-level cache and the memory controller. In the rest of this section, we will introduce design principles and classic architectures of these memory fabrics stage by stage.

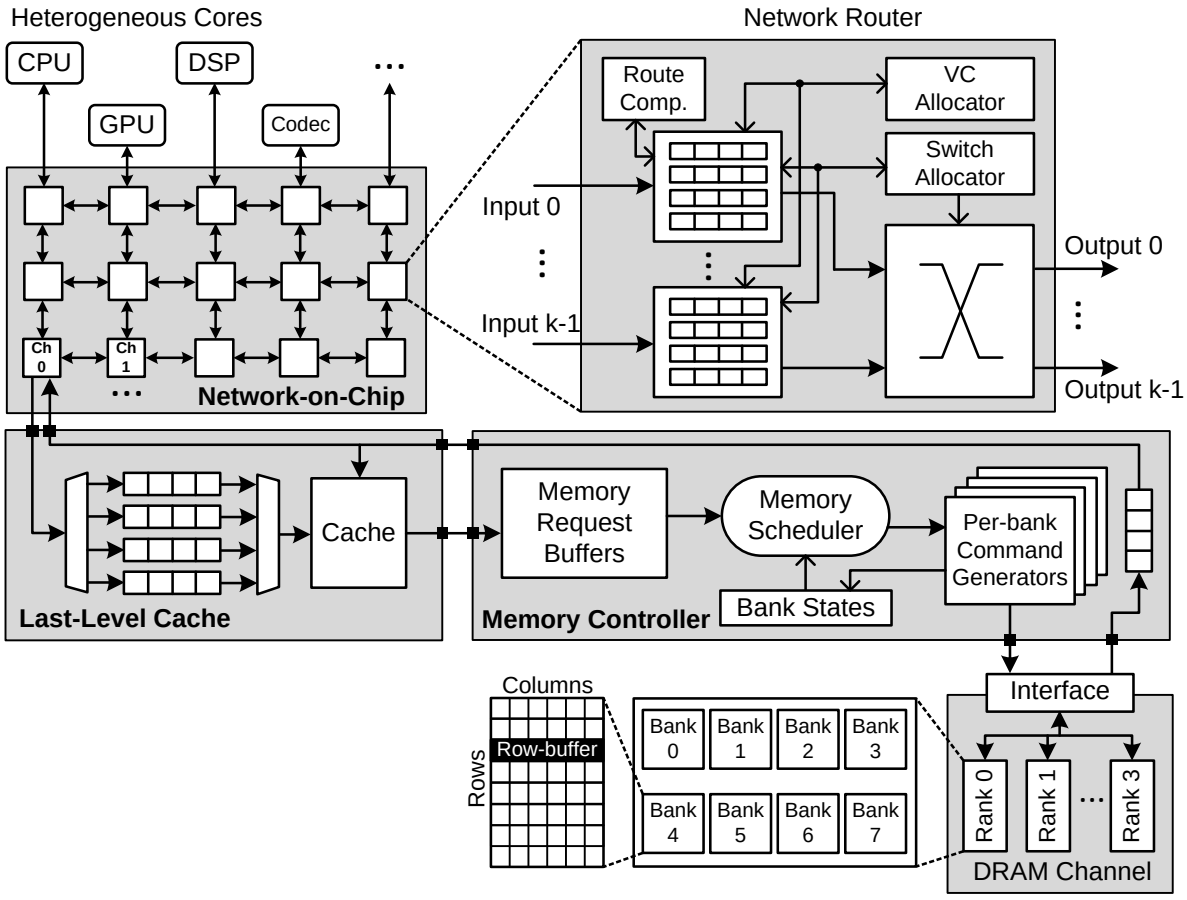


Figure 2.1: Classic architecture of a memory subsystem shared by heterogeneous cores.

2.1.1 Network-on-chip

As the first stage in the memory subsystem, the NoC collects memory traffic from heterogeneous cores and directs the traffic to the last-level cache and the memory controller. When a memory request is injected into the NoC, it is first segmented into network packets if its size exceeds the maximum packet size limit. Then they are further divided into flits which are the basic units for network flow control. As a result, a packet consists of multiple flits, including a head flit that contains the destination address, body flits that carry memory request information and a tail flit that indicates the end of the packet. Upon

ejection, flits and packets are re-assembled into a memory request.

Buffer and link resource allocations, inevitable as memory traffic travels through routers and links in the network, are governed by flow control protocol. State-of-the-art networks often adopt wormhole flow control [10] which manages allocations of buffer storage and link capacity per flit rather than for the entire packet. A flit can depart the current router as long as the next hop provides sufficient buffering capacity for this flit. As a flit traverses a link, it can be further broken down into phits which are the units for link traversal and thus sized by physical link width.

In addition to flow control, the design of a network-on-chip can be decomposed into several building blocks, including topology, routing, and router microarchitecture. We continue the overview on network design based on [23].

Topology and Routing

Network topology is fundamental to network performance because it determines the number of routers a packet must traverse, thus influencing network latency significantly. A topology design can be evaluated from several aspects such as hop count, node degree and path diversity. Based on its connectivity, a network topology can be classified as either direct or indirect. In a direct topology, each terminal node, e.g. a processor core, is connected to a router so that any node can source and sink traffic, as well as switch through traffic from other nodes. In an indirect topology, only terminal nodes are sources and destinations of traffic, whereas intermediate nodes simply switch traffic to and from terminal nodes. As such, packets are switched indirectly through a series of intermediate switch nodes between the source and the destination.

As a typical direct topology, mesh network is frequently adopted in multicore processors due to its advantage in path diversity and the ease of implementation. The symmetric structure of a mesh network makes it a viable choice for platforms where

frequent point-to-point communication happens between homogeneous cores. However, heterogeneous MPSoCs integrate a wide variety of specialized IPC blocks which make regular topologies such as a mesh inappropriate. In the heterogeneous system, not every connection between nodes is necessary because they may not need to communicate with each other. Therefore, often used in heterogeneous MPSoCs, customized topologies are optimized for known communication requirements and network traffic patterns to achieve higher energy efficiency.

Given a network topology, the routing algorithm is used to determine which path a packet will take to reach its destination. Ideally the routing algorithm is able to distribute traffic evenly among the paths to avoid network congestion, as well as minimize per-packet delays. According to how routing decision is made, routing algorithms can be categorized into three classes: deterministic, oblivious and adaptive. Deterministic routing always chooses the same path from A to B. Such a routing algorithm is dimension-ordered routing (DOR) that routes packets from A to intermediate nodes with the same ordinates with B dimension by dimension. On a two-dimensional mesh, DOR is also called XY or YX routing. It is the most commonly used routing algorithm due to its simplicity and minimum hop count. Moreover, oblivious routing forwards packets onto different paths between A and B, but the path selection is performed without regards to network congestion. A more sophisticated routing algorithm can be adaptive, in which the path a packet takes from A to B depends on network traffic situation. In addition, based on hop count, routing algorithms can also be classified as minimal and non-minimal.

Router Pipeline and Microarchitecture

The major components of an NoC router are the input buffers, route computation logic, virtual channel allocator, switch allocator, and the crossbar switch. Router microarchitecture is central to the NoC because it is where routing algorithm, flow control

and buffers/links allocation are implemented. Under high speed clock, NoC routers are usually pipelined to achieve high throughput while performing above tasks. A classic router pipeline includes up to five stages: Buffer Write (BW), Route Computation (RC), Virtual channel Allocation (VA), Switch Allocation (SA) and Switch Traversal (ST). A head flit goes through all stages whereas a body or tail flit only travels through SA and ST stages. In addition to the router pipeline, there is also a Link Traversal (LT) stage where a flit is transported through the link to the next router, the latency of which is determined by flit size and link width.

A head flit, upon arriving at an input port of a router, is first decoded and buffered according to its input Virtual Channel (VC) in the BW pipeline stage. In the RC stage, the routing logic performs route computation to determine the output port for the packet. The header then arbitrates for a VC corresponding to its output port in the VA stage. Upon successful allocation of a VC, the head flit proceeds to the SA stage where it arbitrates for the switch input and output ports. On winning the output port, the flit is then read from the buffer and proceeds to the ST stage, where it traverses the crossbar. Finally, the flit is passed to the next node in the LT stage. Body and tail flits follow a similar pipeline except that they inherit the route and the VC allocated by the head flit. The tail flit, on leaving the router, deallocates the VC reserved by the head flit.

Pipeline optimizations are often applied to shorten router pipeline for head flits. Lookahead routing [13] was proposed to pre-compute routing one hop in advance so that BW and RC can be performed in parallel. Pipeline bypassing technique was introduced for light traffic load to allow a flit to enter the ST stage speculatively if there are no flits ahead of it in the input buffer. At last, speculative SA can be performed in parallel to VA so that the flit directly enters the ST stage following a successful speculation.

The number of pipeline stages determines the minimum number of cycles that it takes a flit to travel through a router. However, under heavy traffic it often takes more

cycles for a flit to go through a router due to the contention from other flits competing for VC and switch resources. Therefore, VC and switch allocators which perform matching between flits and VCs/switch ports have significant influences on the actual delay per router. An allocator that delivers high matching probability translates to more packets succeeding in obtaining VCs and crossbar switch ports, thereby leading to higher network throughput.

Due to the need for pipeline-able allocators, simple separable allocators are typically used, with an allocator being composed of arbiters, where an arbiter is one that chooses one out of N requests to a single resource. For example, in a router with K ports and M VCs per port, we have up to $K \times M$ head flits competing for K VCs from next hops. The allocation can be performed by K $M:1$ input arbiters and K $K:1$ output arbiters. First the input arbiters find a winning VC per input port. Then the output arbiters collect the winners from input ports and find the winning request for each output port. Different arbitration policies can be applied at the arbiters, such as round-robin, first-come-first-serve or priority-based policies. These arbiters and allocators are the implementation basis of QoS-aware routers. We will explore more on network router design in Chapter 6.

2.1.2 Last-Level Cache

Multi-level cache hierarchy is widely deployed on multicore processors as the major conduit for memory sharing among CPU cores. On those platforms low-level caches are inserted between cores and the NoC, and behave as traffic filters that receive memory requests from cores and inject cache coherency traffic into the network. In contrast, cache hierarchy is not applicable to the memory sharing in heterogeneous MPSoCs because most specialized accelerators generate traffic with low memory locality, making caches ineffective. Therefore, heterogeneous cores are often connected to the NoC directly. However, the last-level cache, i.e. the system cache is still needed between the NoC and the memory

controller to reduce the memory access latency of traffic streams with high locality. When a common address space is shared by every heterogeneous core, the last-level cache must capture all memory requests to maintain up-to-date cache lines and manage memory consistency. Since heterogeneous traffic as a whole shows low memory locality, the last-level cache inevitably has high miss rate and behaves similarly to a buffering stage before DRAM access.

A memory subsystem may includes multiple slices of the last-level cache, each one of which is assigned with a unique memory channel and located next to the corresponding memory controller. Inside of the last-level cache, the main component is cache storage whose implementation has been detailed in [16] and is beyond the scope of this dissertation. In addition to cache storage, the last-level cache also includes cache request buffers and a cache controller which arbitrates among buffered requests for cache access. After accessing the last-level cache, memory traffic is converted into cache misses and cache writebacks which will be received as read and write memory requests by the memory controller in the next stage. The impact of memory traffic from the last-level cache on DRAM memory efficiency will be studied in Chapter 5.

2.1.3 Memory Controller

A memory request arriving at the memory controller carries DRAM access information including a physical address and the request type. However, in order to access DRAM, more information is required to generate specific DRAM commands under various DRAM timing constraints. Thus the memory controller is introduced at this stage to fill in the missing information on DRAM access protocol and automate the translation process from memory requests to DRAM commands. In addition, the memory controller is also responsible to manage power state transitions and refresh data arrays in DRAM. To be able to perform above tasks, the memory controller needs per-bank state counters to generate

DRAM commands. A bank state table is necessary to track current states of all memory banks. During memory scheduling, the scheduler scans through the bank state table to identify banks that are ready to receive new commands. To schedule a request to DRAM, the scheduler activates the corresponding state counter to generate commands and updates the state table accordingly.

In the multicore era, the memory controller receives requests from multiple agents. Inevitably the new reality imposes another duty on the controller, that is arbitrating among different traffic streams. As a result, memory scheduling becomes increasingly complicated. Naturally the memory controller evolves into a two-tier architecture performing transaction[1] scheduling and command generating simultaneously. The first tier of the architecture includes multiple transaction queues and the memory scheduler, while the second tier consists of per-bank state counters. The two-tier architecture, as well as its limitation will be further discussed in Chapter 4.

2.2 DRAM Access Protocol

In this section, we introduce memory access protocol of DRAM based on [16]. We will start with an overview of the storage organization and the internal structure of DRAM for a better understanding of the timing constraints on DRAM commands. We will also discuss the impact of these constraints on the memory efficiency.

2.2.1 DRAM Organization

A simplified illustration of DRAM organization has been shown in Fig. 2.1 where the storage in DRAM is organized into multiple hierarchies, including channels, ranks, banks, rows and columns. Fig. 2.2 shows a more detailed view on the internal structure of

[1]In the context of memory scheduling, a memory transaction is the same concept as a memory request. In this dissertation, these two terms are used interchangeably.

DRAM.

A heterogeneous MPSoC may include multiple memory channels which operate on different data buses. Defined by JEDEC standard [20], the data bus in a DDRx memory module is 64-bit wide (or 72-bit wide if the module supports ECC feature). Since memory channels are absolutely independent from each other, they are often embedded at different locations in the SoC and assigned with separate memory controllers for independent memory scheduling.

The organization hierarchy right below channel is memory rank. A memory rank is a set of DRAM chips connected to the same chip select, which are therefore accessed simultaneously. The number of DRAM chips in a rank is determined by their bitwidth. For example, if the data bus of a DRAM chip is 8-bit wide, eight of them need to be aggregated to form a 64-bit data bus (or nine of them if ECC is supported). A channel may include multiple ranks which are accessed independently, although not simultaneously as the data bus is still shared between ranks in the same channel.

Inside of a DRAM chip, memory storage is arranged in multiple banks which are two-dimensional arrays with addressable rows and columns. In addition to DRAM arrays, I/O gating circuitry is used to interface with external data bus and Delay-Lock Loop (DLL) is responsible for the synchronization with input clock signal. In a DRAM array, the most basic storage unit is a DRAM cell where a capacitor encodes one binary digit, whereas the smallest addressable unit is a column whose size is the same as the data bus width of a DRAM chip. To accommodate the bitwidth difference between DRAM cells and chip data bus, a logical memory bank is implemented as multiple physical arrays in parallel. Each one of the physical array produces one bit per cell to match data bus bitwidth and form a column. These arrays share the same control signals and work in lockstep, thereby seen as one logical bank. Note that a single memory request usually accesses multiple columns in a row. For example, to fetch 64 bytes from DRAM through a 64-bit wide data bus, eight

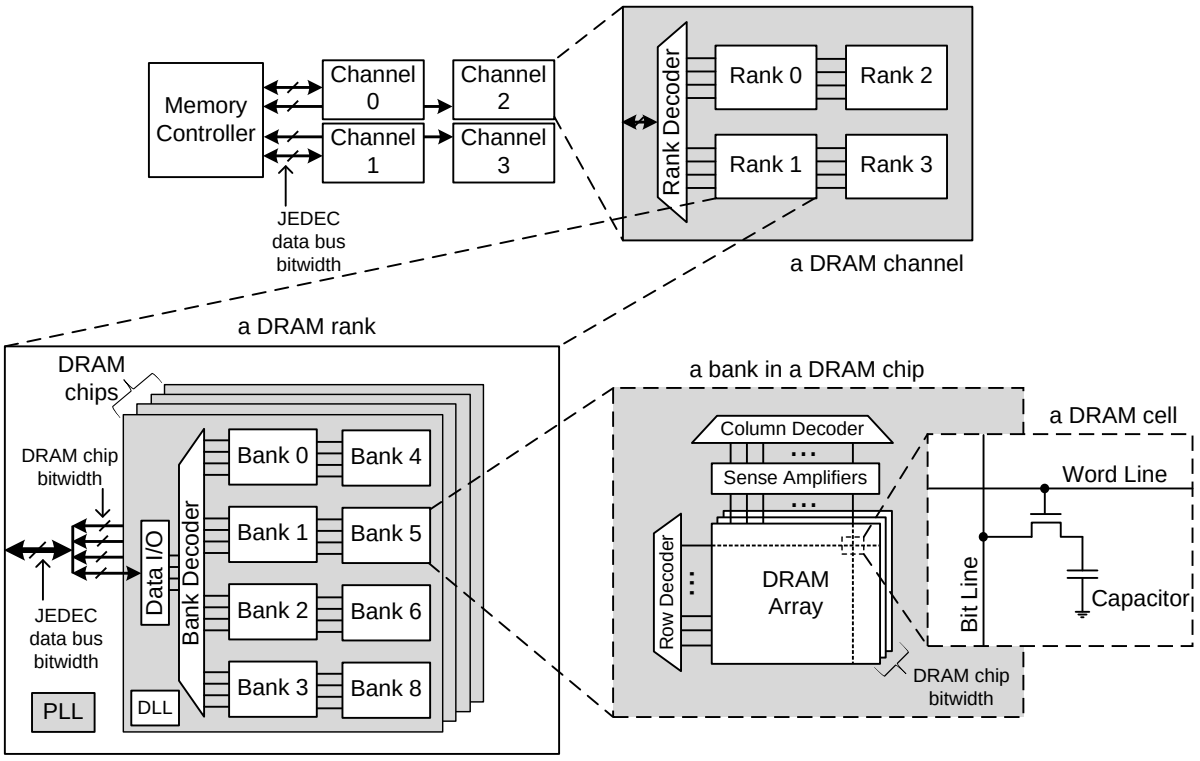


Figure 2.2: Storage organization in DRAM.

consecutive data bursts shall be placed on the bus, each of which comes from one column. This process takes four clock cycles in DDRx DRAMs that support two bursts per cycle.

In a DRAM cell, a capacitor is located at the intersection of a word line and a bit line. The capacitor is connected to the bit line through a transistor controlled by the word line. Cells on the same row share the same word line, while those on the same column share the same bit line. When the voltage on a word line goes high, all of the transistors attached to that word line will be turned on and connect their respective capacitors to bit lines. Since the capacitance of these capacitors is extremely small, sense amplifiers are used to detect any minute voltage changes on bit lines.

To access a column, the associated bit line is first precharged to a certain voltage level. This step is called as precharge operation. The word line corresponding to the row to

be accessed will then be set to high so that the capacitors on this row become connected to bit lines. Every sense amplifier detects voltage change on the bit line and amplifies the tiny change to a full-swing signal. This step is called as row activation. In the last step, column access is performed upon sense amplifiers. Since a row activation operation in essence caches a row of memory at the sense amplifiers until a subsequent precharge command is issued, these amplifiers as a whole are often referred to as a *row-buffer*. A memory bank is associated with one row-buffer, which means only one row can be active at any time per bank. As will be mentioned repeatedly in the following chapters, taking advantage of row-buffer locality and avoiding unnecessary precharge operations are important for memory efficiency improvement.

The electrical charge stored in the storage capacitor will gradually leak away with the passage of time. To ensure data integrity, the stored data value in the DRAM cell must be periodically read out and written back, a process known as refresh. Most DRAM devices use a refresh row address register to keep track of the address of the last refreshed row. Typically, the memory controller sends a single refresh command to DRAM. Then DRAM increments the address in the refresh row address register and goes through a row cycle for all rows with that row address in all of the banks in the DRAM device.

Based on the discussion so far, we can see there are plenty of limitations on the memory efficiency of DRAM. First, since a row-buffer is shared by all rows in the same bank, only one row can be accessed at a time while the others remain closed. Second, periodical refresh operations consume power and interrupt memory banks while they are serving memory requests. In addition, I/O gating and drivers require time to be reversed when a read request is issued after a write request to the same bank. As a result, bus utilization ratio of a DRAM channel can never achieve 100%. Albeit the difficulty, various schemes have been proposed to squeeze as much bandwidth as possible from DRAM, which is one of the goals by our work in addition to providing QoS support to heterogeneous

Table 2.1: Major DRAM command timing constraints.

t_{RCD}	The time interval between row access and data ready at sense amplifiers.
t_{RAS}	The time interval between row access (i.e. row activation) and data restoration in a DRAM array.
t_{CAS}	The time interval between column access and the beginning of data return from DRAM.
t_{BURST}	The time period that data burst occupies on the data bus.
t_{RP}	The time interval that it takes for a DRAM array to be precharged before the next row activation.
t_{WTR}	The minimum time interval between the end of a write data burst and the start of a column read to allow I/O gating to overdrive sense amplifiers.
t_{RFC}	The time it takes to refresh all banks.
t_{RRD}	The minimum time interval between two row activation commands to the same DRAM device.
t_{FAW}	The rolling time window during which a maximum of four-bank activation can be engaged.

cores.

2.2.2 DRAM Timing Constraints

Data bus, I/O gating and row-buffer are commonly shared resources in DRAM. To avoid conflicts when these resources are being shared, there are various timing constraints on DRAM commands that define the minimum time intervals between conflicting operations. Tab. 2.1 shows some of the major timing constraints on DRAM commands. In addition to avoiding resource conflicts, there are also constrains that are introduced to limit peak power consumption in DRAM, such as t_{RRD} and t_{FAW}.

The timing diagram of a single read request is shown in Fig. 2.3, using above timing constraints and assuming the destination row is initially closed. To begin with, row activation command is issued as the first step to access columns in an inactive row. Right after the issuance of row activation command, data in the destination row is brought in from the DRAM cells to the sense amplifiers. t_{RCD} cycles later, data from the requested row

is loaded and stabilized in the sense amplifiers. Only after then can the memory controller place a column read (or a write command) through the command bus. It takes t_{CAS} cycles to fetch data from the target columns and send it to data bus through I/O gating, and another t_{BURST} cycles to propagate the data to the memory controller. Concurrent with the issuance of column access commands, the DRAM device actively restores data from sense amplifiers to DRAM cells (because sharing the stored charge on the bit line is destructive to the voltage level at the cell). Data storage completes in t_{RAS} cycles after the issuance of row activation command. The DRAM cells cannot be precharged until the completion of data storage and precharge operation takes t_{RP} to finish.

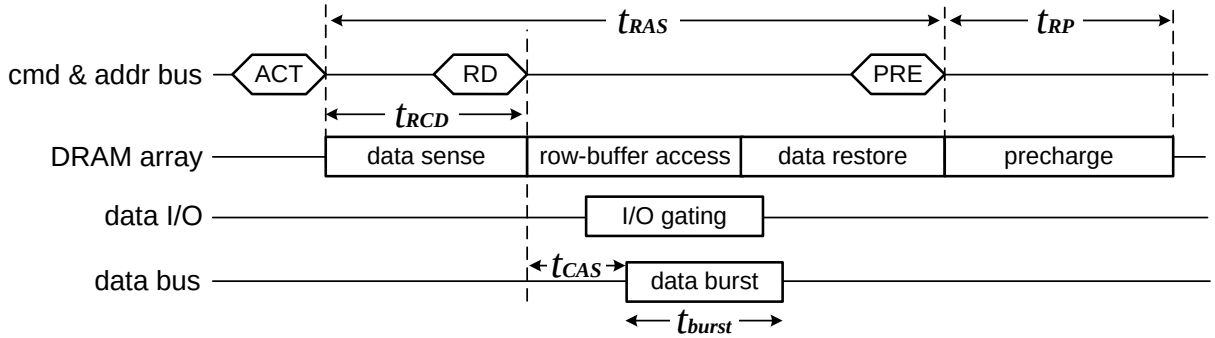


Figure 2.3: Timing diagram of a single read access in DRAM.

In the case of multiple column accesses to the same row, more read (or write) commands can be inserted before the precharge operation. However, to access columns in a different row, precharge operation has to be performed, followed by the activation of the new row, before the column access can take place. Consequently, memory traffic with low row-buffer locality can suffer from excessive delay and energy cost due to the constant repetition of precharge-activation cycles. What's worse, the other timing constraints further limit memory efficiency and eat away available DRAM bandwidth. As for memory scheduling, the central part of the job is to re-order memory requests in such a way that memory delay caused by timing constraints can be hidden and unnecessary energy cost

can be avoided.

2.3 Chapter Summary

In this chapter, we have reviewed the classic architecture of memory subsystems. We have also described the internal organization of DRAM and the timing constraints that were introduced due to the architectural limitations in DRAM. These constraints present design challenges as well as research opportunities for memory scheduling. In the subsequent chapters, we will detail our contributions in memory subsystem design.

Chapter 3

Self-Aware Resource Allocation

3.1 Introduction

In this chapter, we will explore system-wide solutions to provide end-to-end QoS and improve memory efficiency. A simplified architecture of heterogeneous MPSoCs is shown in Fig. 3.1[1], where the memory subsystem is used as the medium for data sharing. As data is being shared through the main memory, memory requests from different cores can easily interfere with each other. For a better understanding of how common the memory interference can be, Fig. 3.2 illustrates the data flow of a camcorder application. As a typical use case, the camcorder application involves up to three tasks, each of which requires participation from different heterogeneous cores. In the data flow, an independent traffic stream is generated every time when the shared data is accessed. As heterogeneous cores are operating in parallel, the memory subsystem serves their traffic streams concurrently. Note that the data flow in Fig. 3.2 only includes major real-time agents. In practice, there are often more heterogeneous cores and traffic streams being active during this process.

[1]Unlike in Fig. 2.1, the last-level cache is omitted here for simplicity because it behaves similarly to a large buffer due to the high miss rate in heterogeneous systems. More discussions on the last-level cache will be found in Chapter 5.

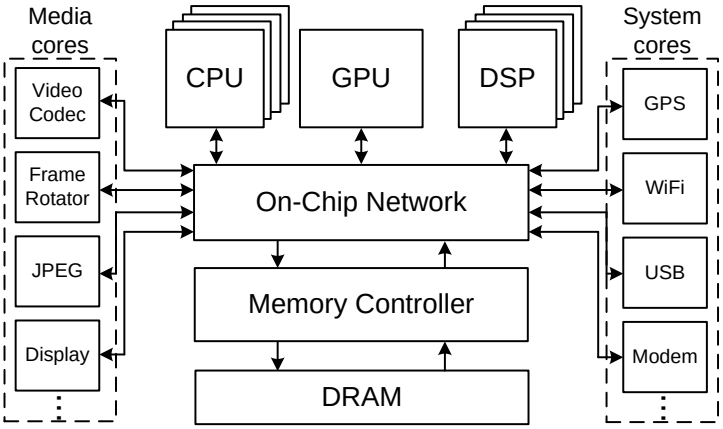


Figure 3.1: Simplified heterogeneous system architecture.

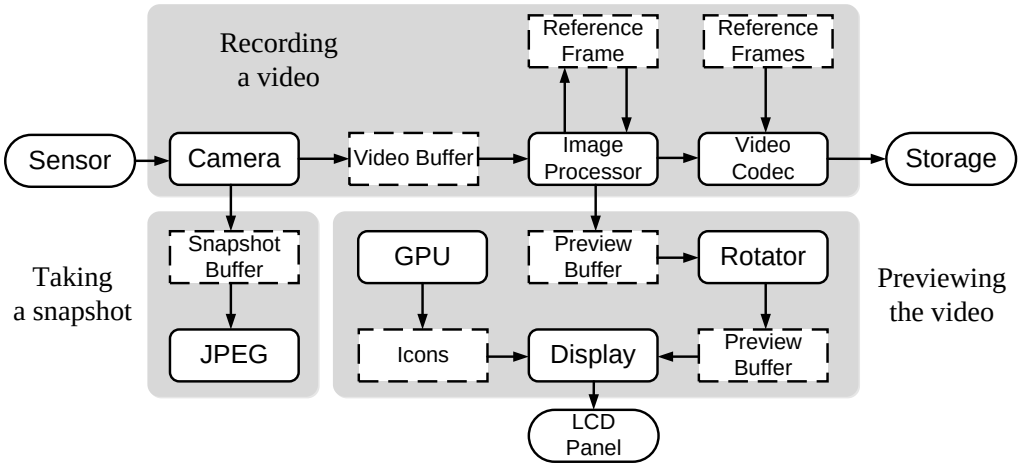


Figure 3.2: The data flow of a camcorder application involving three tasks: recording a video, previewing the video and taking a snapshot. Shared data in the main memory is represented by boxes in dash lines and heterogeneous cores are represented by boxes in solid lines.

Due to the commonality of memory sharing, memory scheduling has great impact on the QoS of heterogeneous cores. Specifically, with ineffective memory scheduling, a real-time core (e.g., the display) may not achieve the target real-time performance due to inadequate memory bandwidth. On the other hand, when latency-sensitive cores such as the DSP share memory with others, they can be easily overwhelmed by real-time cores consuming high bandwidth.

QoS-aware management for specific types of memory resources has been well-studied by previous work [22, 4, 47, 14, 53]. In [22], a QoS-aware scheduling policy was proposed for CPU-GPU systems. The concept of frame progress was introduced to monitor GPU performance. Although the policy can be extended to include more media cores, it cannot be applied to real-time cores whose target QoS cannot be assessed in terms of frame rate. Moreover, holistic memory management frameworks for CPU-centric homogeneous systems have also been explored recently [44, 15, 8]. This series of work typically constructs a management model based on the control theory to partition computing and memory resources. These frameworks accept flexible QoS targets as clients are allowed to define their own target performance. Nonetheless, such type of approaches is performed at a relatively slow timescale (e.g., on the order of milliseconds) due to the computational complexity. In comparison, real-time cores in heterogeneous MPSoCs often demand much more instant response from the memory system. In addition, communication between heterogeneous cores is mainly conducted through shared memory as shown in Fig. 3.2, because multimedia data is generally too large to fit into caches. Therefore, DRAM plays a more crucial role in heterogeneous systems. However, previous frameworks cannot handle DRAM effectively because its bandwidth is not partitionable. The memory access time at DRAM relies on many factors, such as row-buffer locality, bank-level parallelism, write-to-read turnaround times and so on. It takes less time to serve a traffic stream with higher locality and parallelism, and fewer write-to-read turnaround times. Without knowing memory access patterns of all traffic streams, any attempts to partition DRAM bandwidth by specific ratios or rations will be futile. What's worse, the memory access pattern of all traffic streams cannot be easily predicted by the memory subsystem.

So far, there has not been a QoS-aware resource management model for heterogeneous MPSoCs which is capable of allocating non-partitionable resources to fleeting QoS demands. In this chapter, we propose the Self-Aware Resource Allocation (SARA) framework as a

solution. The contributions of our work are summarized as follows.

- We propose a QoS-aware holistic resource management framework for heterogeneous systems. The SARA model accepts diverse notions of QoS and monitors performance distributively with lightweight meters to guarantee end-to-end QoS.
- We introduce priority-based self-adaptations for the management of non-partitionable resources, such as DRAM and the on-chip network, which constitute most of the shared resources in heterogeneous MPSoCs.
- We present a runtime configuration scheme for priority-based adaptation to accommodate heterogeneous sensitivities with respect to memory resources and also to adapt allocations provided to best-effort cores.
- We evaluate the proposed framework using memory traffic of state-of-the-art MPSoCs and show that the proposed SARA model delivers target performance to all cores. In contrast, the performance of critical cores can fall below 10% of their targets without the SARA framework. Furthermore, the SARA framework is evaluated with runtime configuration of priority-based adaptation.

The rest of this chapter is organized as follows: Section 3.2 provides a comprehensive review on related work. Section 3.3 describes the proposed SARA framework. Section 3.4 presents a runtime configuration scheme for priority-based adaptation in the SARA framework. Experimental results and conclusions follow in Section 3.5 and 3.6.

3.2 Related Work

QoS-aware Memory Fabrics: Along with the emergence of heterogeneous SoCs, QoS-aware memory fabrics have been well studied in the recent decade. Staged memory scheduler [4] was the first application-aware scheduler proposed for CPU-GPU systems. The proposed scheduler architecture consists of separate queuing stages for memory transactions

and DRAM commands was introduced. In between two stages a QoS-aware scheduler is deployed. The single-tier virtual queuing memory controller [47] was later proposed to overcome the inefficacy of two-tier controllers in QoS-aware scheduling. A single queuing stage is used to sort per-source traffic flows into per-bank virtual queues once they are received by transaction buffers. The QoS-aware scheduling is involved as the last step in the memory controller to avoid unwanted latency between the scheduler and DRAM. In [22], a QoS-aware scheduling policy was proposed to dynamically balance CPU and GPU bandwidth. The proposed policy tracks GPU workload progress and allocates sufficient bandwidth to the GPU to achieve the target frame rate. Meanwhile best-effort service is provided to the CPU. Furthermore, a deadline-aware scheduler [51] extends the idea of frame progress tracking and takes into account more multimedia cores with target frame rates while improving CPU performance. Besides memory scheduling, QoS-aware on-chip networks [31, 37, 14, 60] and QoS-aware cache management policies [30, 33, 53] have been extensively investigated in recent years as well. We will have more discussions on QoS-aware memory fabrics in the following chapters.

Memory Management Frameworks: Nonetheless, these work cannot guarantee end-to-end QoS because they only deal with certain parts of the memory system. For example, the QoS provided in the memory controller could be deteriorated by the interconnect if it is not applying the same QoS policy. Moreover, implementing a centralized QoS monitor in the memory system can be prohibitive since it needs to collect runtime information from all cores. More limiting, these work assume specific notions of QoS, which is not applicable to modern heterogeneous MPSoCs where the health of different cores is often evaluated by diverse performance objectives.

METE [44] is a multi-level framework for end-to-end resource management based on the control theory. It utilizes runtime information to predict application behaviors. Application controllers calculate the amounts of resources required to achieve target

application performance. A global resource broker determines the final resource partitions for applications. SEEC [15] is a self-aware computing framework designed for a many-core processor. It follows the control loop of observe-decide-act (ODA) for resource allocations. Performance of CPU applications are observed by the decision engine which decides resource partitions using available actions defined by system designers. ARCC [8] is a self-computing framework implemented in the Tessellation many-core OS. It performs a two-level scheduling scheme: first the resource allocation broker distributes global resources and then at user-level scheduling policies are customized separately. CPSoC [43] is a class of sensor-actuator-rich MPSoCs following observe-decide-act paradigm to enable self-awareness. With augmented on-chip and cross-layer sensing and actuation capabilities, it supports adaptation per layer and multiple cooperative ODA loops to achieve multi-objective goals. SPECTR [41] is a new management approach for many-core processors that leverages formal supervisory control theory to decompose complex control problems into sub-problems which are solved by local controllers. A survey has been made covering recent advancements in self-aware platforms [19].

Aforementioned frameworks are aimed at allocating partitionable resources, such as CPU cores and network bandwidth, to applications at software/OS level. They are not suitable for heterogeneous MPSoCs for the following reasons.

First, complicated control models may not be fast enough for heterogeneous cores, because real-time cores often have tight deadlines. For example, the DSP sets limit on memory latency at nanosecond level, but prior frameworks adapt through control theory computations at milliseconds timescales. Only a management framework implemented on the hardware level is able to deliver service that is sufficiently responsive.

More importantly, prior work assume all memory resources can be partitioned using weights. However, the assumption does not apply to DRAM. In DRAM, data storage of a memory bank is organized into rows and columns. Only one row can be active in a bank.

Columns in an open row can be accessed directly without extra operations. Whereas, to access a column in a closed row, the row where this column is located have to be loaded into the row-buffer (i.e. row activation operation) after all other rows have been closed (i.e. precharge operation) [16]. These row activation and precharge operations cause time penalty without contributing to actual data transfers, which makes DRAM access time unpredictable. For example, a traffic stream with higher row-buffer locality suffers from less time penalty and thus achieves lower access latency; a stream with higher bank-level parallelism avoids more precharge operations but accessing unengaged banks. Moreover, there is also time penalty for switching from a write request to a read request. Due to above physical limitations of DRAM, the deliverable bandwidth from DRAM cannot be partitioned without considering the erratic memory access patterns of all streams. To avoid the difficulty for weight-based partitioning approaches, we adopt a priority-based approach for resource allocation. The proposed framework will be detailed in the next section.

3.3 Self-Aware Resource Allocation Framework

The proposed architecture of SARA framework is shown in Fig. 3.3. The resource management model consists of three stages, including distributed monitoring, priority-based runtime adaptation and system response. Next we will go through SARA framework stage by stage.

3.3.1 Distributed Self-Monitoring

In the first stage, each core self-monitors its own performance. Distributed monitoring relieves the memory system from the burden of monitoring cores with various notions of QoS. It is also good for scalability, since a new core can be added or modified without updating the rest of the system. In addition, the self-monitoring provides more accurate

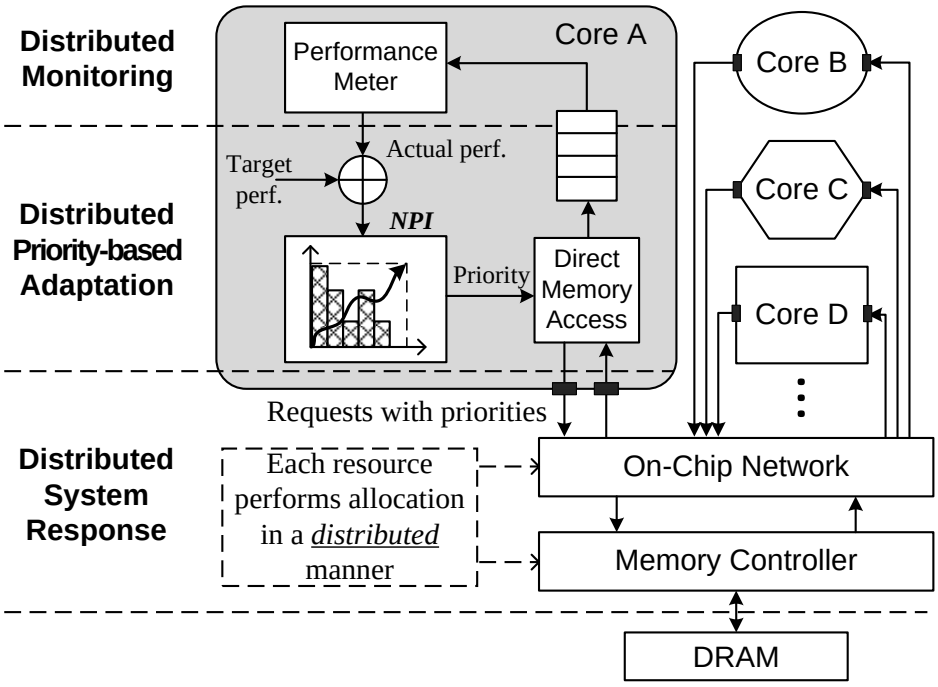


Figure 3.3: The proposed SARA framework for heterogeneous MPSoCs. Each core self-monitors its performance and self-adapts its priority, and each resource performs priority-based allocation in a distributed manner.

feedback on the end-to-end QoS compared with centralized monitoring in the memory system.

Every core customizes its own internal performance meter to measure its own performance or progression against a given target, and the measurement gets normalized into a fractional number called the Normalized Performance Indicator (NPI), which is used as an indicator of the core's intrinsic health. In the DSP, the performance meter monitors the average latency of its transactions, while in the display the meter measures the occupancy level of the read buffer. The deviation from the target performance (e.g., latency, occupancy level, etc) produces the NPI metric. In our framework, each real-time Direct Memory Access (DMA) unit is equipped with a performance meter. Note that there are usually multiple DMAs in a single core. For simplicity, we only show one DMA per core in Fig. 3.3.

3.3.2 Distributed Priority-Based Adaptation

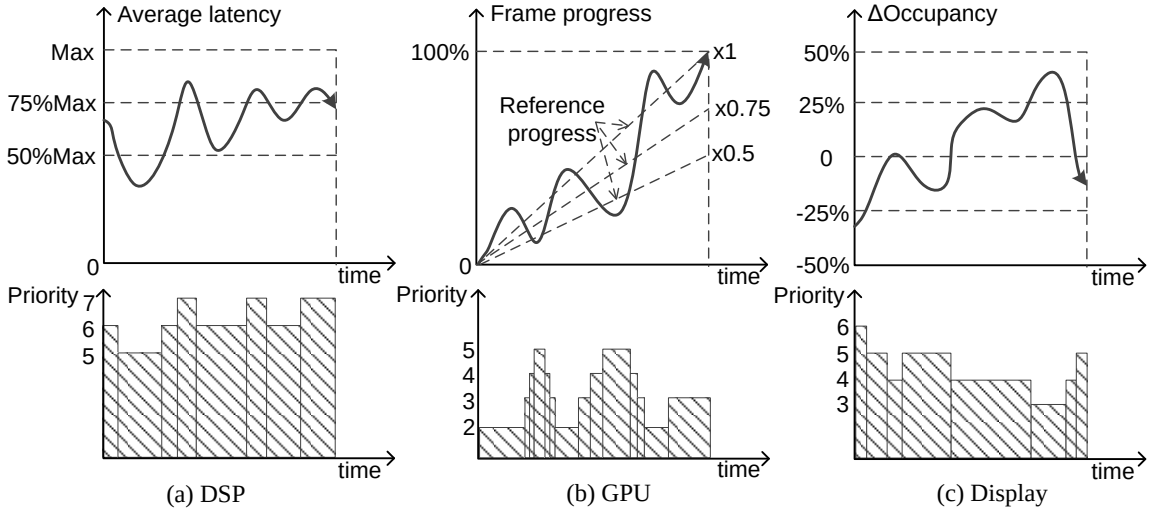


Figure 3.4: Examples of priority-based adaptation in heterogeneous cores, including the DSP, the GPU and the display.

In the second stage, the NPI value delivered by the performance meter is translated into a relative priority level which is attached to memory transactions from the same DMA. As transactions travel to DRAM, their priority levels will be accessed and evaluated by arbiters in the on-chip network and the memory controller. Priority-based arbitrations allow the memory system to provide QoS without specifying the heterogeneous QoS for all cores and DMAs. Same with performance meters, the formulation of the NPI metric and the adaptations of priority can be implemented differently from core to core, depending on the local target performance. Fig. 3.4 shows three examples of priority-based adaptation in different cores.

As for the DSP, the target performance is to have the average memory latency lower than the maximum latency limit. The average latency is measured and compared with a pre-set limit to produce the NPI value (see Eqn. 3.1), which remains above or equal to 1 when the target performance is achieved. This NPI value is then translated to a relative

priority level (Fig. 3.4(a)). The priority level increases along with average latency.

$$NPI_{DSP} = \frac{maximum\ latency\ limit}{average\ latency} \tag{3.1}$$

Similarly, cores requesting for bandwidth produce NPI metrics by computing the ratio between the average and the target bandwidth. However, frame rate differs from bandwidth, because frame size can be variable and thus a constant frame rate can lead to variable bandwidth. Hence frame progress [22, 47] is used instead to produce NPI metrics for frame rate based cores. Take the GPU as an example, the target is to let the frame progress reach 100% as the current frame period comes to an end. The GPU's NPI value is produced at any time by comparing the frame progress with reference progresses which grow proportionally with frame time. The NPI value is then translated to a relative priority level of GPU transactions. Fig. 3.4(b) shows the reference progresses achieving 1, 0.75 and 0.5 times the average data rate of target performance.

$$NPI_{GPU} = \frac{frame\ progress}{reference\ progress} \tag{3.2}$$

In the display, LCD panel reads data from a read buffer at a constant frame rate, while the display controller DMA tries to refill this buffer from DRAM so it never gets empty. Its *health* (see Eqn. 3.3) relies on maintaining the refill rate (R_{refill}) no lower than the read data rate (R_{read}), and can be indicated by the variation of buffer occupancy level ($\Delta occupancy$). Compared with an initial level (e.g. 50%), the lower the occupancy level of this buffer gets, the worse the NPI value becomes, which is in turn translated to a higher priority level (Fig. 3.4(c)).

$$NPI_{display} = \frac{R_{refill}}{R_{read}} = 1 + \frac{\Delta occupancy}{R_{read} \cdot time} \tag{3.3}$$

Intuitively, one might be concerned that every core would intentionally raise its priority to the maximum level to obtain as much resource as possible. Since a priority level is only maximized when the actual performance is far below the target, the pitfall can be avoided by properly setting performance targets. This should be done during design time with exhaustive profiling tests on realistic workloads while priority-based adaptation is enabled. In runtime, pre-determined performance targets are loaded into real-time cores depending on the workload. Note that performance targets of real-time cores should not be programmable because they are limited by hardware performance.

In our evaluations, the priority levels are quantized into 2^k levels, which can be encoded with k bits. We found that $k = 3$ bits provides sufficient granularity in priority levels to produce satisfying results (i.e., the priority levels range from 0 to 7). The translation of NPI value into a priority level can be defined separately for each core, as long as lower NPI value is mapped to a higher priority level. An example of such an NPI translation function is shown in Eqn. 4, in which the priority is adjusted between 0 and 7 in an inverse proportion to NPI value. When NPI value equals or drops below 1, the priority is raised to the maximum level (7). On the other hand, if NPI value grows above 1, the priority is gradually lowered to the minimal level (0).

$$\text{priority}(NPI) = \begin{cases} 7 & NPI \in (0, 1) \\ Round(\frac{7}{NPI}) & NPI \in [1, 14] \\ 0 & NPI \in (14, \infty) \end{cases} \tag{3.4}$$

Note that, instead of [0,7], the upper and lower bounds of priority range can be different from core to core in order to reflect their heterogeneity. Same with performance targets, priority ranges of real-time cores can be pre-tuned per workload during design time. Alternatively, an adaptive configuration scheme is needed to adjust priority ranges according to real-time workloads. This discussion will be continued in Section 3.4.

3.3.3 Distributed System Response

As transactions travel through the memory system, the system responds to QoS demands by providing resource management based on their priority levels. The priority-based management is performed correspondingly in different parts of the memory system. In on-chip network routers, transactions with higher priorities are preferentially selected during switch allocation. In the memory controller, when a priority-based scheduler arbitrates among transactions going to available memory banks, the ones with higher priorities have more chances to be served. An example of such memory scheduling policies is the priority-based round-robin shown in Policy 1. To avoid starvation of transactions with low priorities, the scheduler also needs to consider the aging factor during arbitration. In our evaluations, the scheduler periodically clears the backlog of transactions that have waited for at least T cycles (e.g., $T = 10000$ cycles).

- **Policy 1**: Suppose P_A and P_B are priorities for transactions A and B, if $P_A > P_B$ choose A; if $P_A < P_B$ choose B; otherwise choose between A and B in round-robin manners.

Priorities notify the system whether the cores are in urgent QoS demands. That gives the memory system an opportunity to optimize memory performance without undermining the QoS. Specifically, when transactions are in low urgency, the system can improve memory performance such as row-buffer hit rate, instead of focusing on serving QoS demands.

Row-buffer hits refer to multiple memory accesses to the same active row before it is precharged. Having more row-buffer hits means less time and power are wasted on row activation and precharge operations. Therefore, increasing row-buffer hits helps lower memory latency and improve DRAM total bandwidth.

To increase row-buffer hits, the memory controller re-orders transactions to favor the ones hitting open rows. It may cause degradations to the QoS when the transactions

in high urgency are postponed due to row-buffer hits optimization. Yet, with priorities, the memory controller is aware of the urgency levels of transactions and able to avoid delaying urgent transactions during the optimization. Policy 2 shows an extension of Policy 1 to increase row-buffer hits without QoS degradations. The parameter δ is an adjustable threshold to balance row-buffer hits optimization and QoS-aware scheduling. When the priority level is lower than δ, the scheduler focuses on row-buffer hits, otherwise the QoS comes first. A higher δ value gives more favor to DRAM bandwidth, but also potentially causes more disturbance to the QoS. We found $\delta = 6$ a good setting to achieve high DRAM bandwidth without causing QoS degradations.

- **Policy 2**: Suppose transaction A is going to an active row-buffer and B is not. If $P_A, P_B < \delta$ or $P_A = P_B$, choose A. Otherwise, perform priority-based round-robin.

Priority-based resource allocation handles non-partitionable with little computation in comparison with previous management models [44, 15, 8]. This facilitates instant response from the memory system to QoS demands.

3.3.4 Hardware Implementation

The implementation of the proposed SARA framework includes three parts: the computation of NPI value, the translation of NPI value to a priority level, and the priority-based arbitration in the memory system.

To calculate the NPI, a divider is needed at the performance meter for each DMA. For the translation of the NPI, a mapping function can be stored in a look-up table at each core. Each priority level is assigned with a table entry, and this entry stores the lowest NPI value allowed at that priority level. For example, if priority $= p$ when $NPI \in [u, v)$, the value u will be stored at the entry for p on the look-up table. Note that v will be the lower bound of the NPI for the priority level $p - 1$. Comparators are used to access table entries in parallel. If the given NPI value is not lower than the stored lower bound of NPI

value, the corresponding priority level will be asserted. When multiple priority levels are asserted, the lowest level will be chosen.

Supposed each priority level is encoded into three bits, a look-up table requires $2^3 = 8$ entries and each entry is a register for NPI value. A comparator is paired with each table entry. In total, the implementation only costs the storage of eight registers and eight comparators per core.

In the memory system, 3-bit comparators are required to perform priority-based arbitrations. Since most existing QoS-aware schedulers already provide hardware support for priorities, our framework can be integrated into the memory system without raising complexity.

3.4 Runtime Configuration of Priority-Based Adaptation

The SARA framework provides a platform for heterogeneous cores to define and monitor their performance independently. In addition, the proposed framework also enables fast and consistent responses from memory fabrics through priority-based arbitrations. As the common currency within the framework, the priority attached to memory requests bridges the gaps between distinct notions of QoS and different types of memory resources.

Since the priority plays a central role in the SARA framework, priority-based adaptation is critical in the effectiveness of the framework. An exemplary function which translates given NPI value to a priority level has been shown in Eqn. 3.4. However, a single translation function may not be universally efficient with all cores due to their heterogeneous behaviors. An inefficient translation function may fail to raise the priority high enough to request for sufficient resources. On the other hand, an improperly configured function may also set priority level excessively high, hindering other cores from obtaining

enough resources. The challenges of configuring effective adaptations are specified as below.

Challenge I: Due to different QoS targets, some cores are inherently more sensitive to the updates in memory allocation than the others. For example, the video codec and the DSP are vastly different real-time cores. Their QoS targets are defined respectively in terms of frame rate and memory latency. In order to achieve 30fps, the codec needs to finish a single frame in 33ms which is an abundant amount of time for priority-based adaptation. By comparison, the latency limit of the DSP can be even shorter than $1\mu s$, resulting in little tolerance of allocation failure. If the same NPI translation function is applied to both cores, it could be too conservative for the DSP (i.e. the priority is not high enough) or too aggressive for the codec (i.e. the priority is overly high). Excessive resources achieved by the codec due to the inappropriately high priority level could further lead to more QoS failures at other cores.

Challenge II: For the cores with no QoS targets, its priority level depends on the health status of those with QoS targets. In other words, we provide best-effort service to cores with no QoS targets. The CPU can be such a type of core in that, unlike real-time cores, the CPU may not have a pre-fixed performance target. This does not mean that the CPU is not as critical as others. On the contrary, we want to allocate as much resource as possible to the CPU as long as real-time cores are able to achieve their performance targets. This presents another challenge to our priority-based adaptation, because the priority of the CPU cannot be simply derived through an independent NPI translation function.

Aforementioned challenges can be tackled in two ways. The first approach is to pre-tune NPI translation functions per core during design time through off-line full-system simulations. Pre-tuning has to be done in regard to numerous possible real-time workloads and memory subsystem parameters such as DRAM frequency, because memory competition varies under different scenarios. The other approach is to configure NPI translation functions in runtime according to the healthiness of real-time cores and available memory resources.

In this section, we propose a runtime configuration scheme for the priority-based adaptation. The proposed scheme includes two stages: distributed self-configuration (DSC) and global regulation (GR). In the DSC stage, NPI translation functions are configured separately at each core to reflect its sensitivity to memory allocation. In the GR stage, priority levels of best-effort cores are adapted according to the healthiness of the cores with QoS targets. In the rest of this section, the proposed configuration scheme will be discussed stage by stage.

3.4.1 Distributed Self-Configuration

The DSC stage takes place at each core in parallel. The self-configuration consists of two components: a configurable translation function and a configuration method.

Configurable Translation

To begin with, we convert Eqn. 3.4 into a configurable format shown in Eqn. 3.5 where B_l and B_h are two integer variables used as lower and upper bounds of the priority range. These two variables are intended to be updated in runtime as long as they satisfy $0 \leq B_l < B_h \leq 7$. This way, we can configure a translation function by adjusting its priority range. Note that when B_h is raised to the maximum level and B_l is reduced to the minimum, the translation function regresses to Eqn. 3.4.

$$\text{priority}(NPI) \begin{cases} B_h & NPI \in [0, 1) \\ Round(\frac{B_h - B_l}{NPI}) + B_l & NPI \in [1, 14] \\ B_l & NPI \in (14, \infty) \end{cases} \tag{3.5}$$

According to the discussion in **Challenge I**, we want to raise the priority ranges of cores that are prone to QoS failure and vice versa for robust cores. However, adjusting both bounds at the same time could lead to unexpected results. For example, it is not

obvious which one of $[3, 5]$ and $[2, 6]$ would be a better choice to fix QoS failure. To simplify the adjustment, we increment or decrement one bound at a time. The pseudo-functions for priority range adjustment are shown below. During priority range increment, the upper bound is first raised to the maximum level before the lower bound is increased. As for priority range decrement, we reduce the lower bound to the minimum level before lowering the upper bound. As can be seen, these two priority range adjustment routines are complementary to each other.

Algorithm 1: priority range increment	**Algorithm 2:** priority range decrement
function inc_range(B_l, B_h)	**function** dec_range(B_l, B_h)
if $B_h < 7$ **then**	**if** $B_l > 0$ **then**
B_h++;	B_l-- ;
else if $B_l < B_h - 1$ **then**	**else if** $B_h > B_l + 1$ **then**
B_l++;	B_h--;
end if	**end if**
endfunction	**endfunction**

Configuration Method

The configurable translation function and the configuration routines lay the foundation for runtime configuration of translation functions. The objective of self-configuration is to adjust the priority range of any given core to meet its target QoS without causing excessive interference to others. This includes two sub-tasks: First, the priority range requires increments in the case of QoS failure. Second, the priority range needs decrements to avoid requesting for excessive resources. Before the presentation of our scheme for runtime self-configuration, a few metrics are introduced as below.

- **Failure Rate (FR)**: the percentage of time when NPI is below 1. To compute FR,

the NPI of a real-time core is sampled periodically. FR is the percentage of samples with $NPI < 1$.

- **Healthiness**: a binary state indicating the healthiness of a given core. Different from NPI which indicates a core's health at a specific time instant, healthiness indicates the overall performance within a period of time. A core is considered *healthy* when its FR is below a certain threshold. In our evaluation, we set 3% as the FR threshold, which means a core is *healthy* if its NPI is above or equal to 1 for at least 97% of the time during the measured period.

- **Stasis**: a flag indicating whether self-configuration has reached a stasis. This metric is introduced to evaluate the progress of self-configuration. A stasis is achieved by self-configuration under two circumstances: first, the core is considered in stasis if it is healthy and its priority range is not too high to cause unnecessary competition against others; second, a core is at a stasis if it is unhealthy but its priority range is already at the highest level, i.e. $[6, 7]$. In both circumstances, self-configuration cannot make further progress without violating QoS target or priority limit.

The complete configuration method is shown in Fig. 3.5 where P_{ave} and NPI_{ave} are the average priority level and the average NPI respectively. They are measured along with the failure rate and used to determine whether self-configuration has reached a stasis. As indicated by the colors, self-configuration operations are performed in different timescales from priority-based adaptation. First, periodical sampling is performed to monitor NPI and priority level. Frequent sampling may fail to reflect a core's healthiness because system response to priority-based adaptation takes time, whereas long intervals between samples will slow down the following operations. In our evaluation, sampling is triggered every 100 cycles. P_{ave}, NPI_{ave} and FR can only be evaluated after sufficient samples have been collected, which means evaluation operation is executed in a further larger timescale than

sampling. Evaluations are performed on every 100 samples in our evaluation.

As the last operation, if a stasis has not been reached, the priority range will be adjusted according to the runtime information. The priority range should be incremented, if the core is unhealthy and its priority range is not set at the maximum level. This condition can be easily verified but checking whether $FR > 3\%$ and $[B_l, B_h] = [6, 7]$. As has been discussed, the priority range can also be decremented if a healthy core is causing unnecessary competition against others. P_{ave} is used to quantify the competition pressure that a core is causing to others. If a core is robustly healthy, we want to keep P_{ave} around the center of the full priority range $[0, 7]$, i.e. 3, to avoid pressing too much pressure on others. Since *healthiness* is a binary metric, we still need another metric, NPI_{ave} to quantify the robustness of healthiness. We consider $NPI_{ave} > 2$ as the indication of robust healthiness. Therefore, this condition can be verified by checking whether $FR \leq 3\%$, $NPI_{ave} > 2$ and $P_{ave} > 3$.

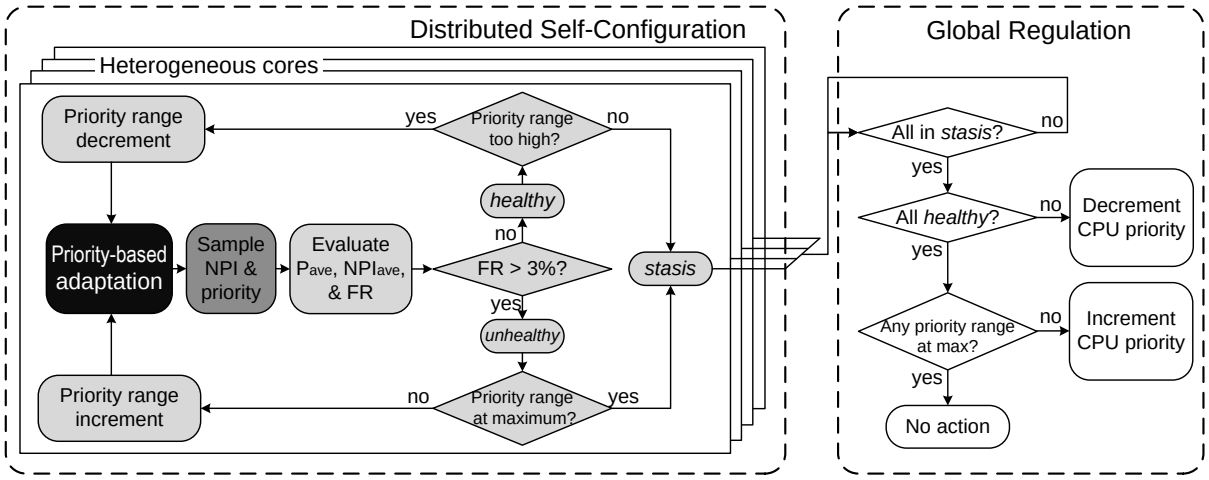


Figure 3.5: Runtime configuration of priority-based adaptation. Operations that are executed in different timescales are labeled in different colors as a darker color indicates a smaller timescale.

3.4.2 Global Regulation

Distributed self-configuration adjusts the priority range of the translation function at each core according to their sensitivities with respect to memory service. It also prevents the core from raising its priority range unnecessarily high to interfere with others. However, **Challenge II** remains to be solved since best-effort cores do not have NPI translation functions for their adaptations.

As has been discussed, the priority of a best-effort core should be set according to the healthiness of the cores with QoS targets. Specifically, the priority of a best-effort core can only be raised when there is no QoS failure at other cores, and it should be lowered if it causes QoS failure to others due to memory interference. A global regulator is introduced to monitor the healthiness of real-time cores and adjust the priority of best-effort cores accordingly. Different from the SARA framework and the self-configuration of translation functions which are implemented in hardware and distributed all over the SoC, this regulator can be deployed in the CPU at OS level, which enables programmable regulation policies for best-effort cores. Note that, the application of global regulation is not limited to the priority of best-effort cores. It can also be applied to best-effort system features such as DVFS. As an example, frequency or supply voltage is gradually reduced to save energy if all real-time cores are stabilized in healthy states.

Fig. 3.5 shows an example of global regulation with the CPU as a best-effort core. The regulation stage is invoked after all real-time cores with QoS targets have settled in stases. If any core is not healthy, the regulator will decrement the priority level of the CPU to release more memory resources. If all cores are healthy and no one has set its priority range to the maximum level (indicating the brink of QoS failure), CPU priority will be incremented. Since the regulation is performed after real-time cores have reached stases, the GR stage is operated at an even larger timescale than priority range adjustment in the DSC stage. This allows algorithms with much higher time complexity than that in

Fig. 3.5 to be deployed at the global regulator. Since prior work have extensively studied the management policies for best-effort cores and services [15, 8], we will not further the discussion on global regulation algorithms.

It is worth noting that an important feature of the SARA framework is decoupling memory allocation into different control layers across hardware and OS levels operating in different timescales, including priority-based adaptation, distributed self-configuration and global regulation. First, priority-based adaptation enables transaction-level responsive memory allocation in hardware. Second, distributed self-configuration adjusts adaptation to fit heterogeneous QoS targets by real-time cores. At last, best-effort cores are handled by global regulation.

3.5 Evaluation

Table 3.1: Memory subsystem simulation configurations.

Memory Traffic	
Case A	All cores are active with DRAM @1866MHz
Case B	Inactive cores include GPS, camera, rotator and JPEG with DRAM @1700MHz
Memory System	
NoC	1GHz clock rate, dimension-order routing, 128-bit link width, 4 VCs per port, 5 buffers per VC, separable VC/switch allocation
MC	42 request reordering buffers, 5 request queues, address mapping scheme (from physical address to DRAM address): channel-rank-row-bank-column
DRAM	2GB memory, 2KB page size, Channels-Ranks-Banks: 2-2-8, CL-tRCD-tRP(cycles): 36-34-34, tWTR-tRTP-tWR(cycles): 19-14-34, tRRD-tFAW(cycles): 19-75

In this section, the proposed SARA framework will first be tested with pre-tuned NPI translation functions to demonstrate its effectiveness in providing target performance

Table 3.2: Summary of heterogeneous cores and their pre-tuned priority ranges.

Name	Target performance	Priority range
GPU	frame rate $\geq$ 30fps	[0, 7]
DSP	average latency $\leq$ 450ns	[5, 7]
Image Processor	frame rate $\geq$ 30fps	[0, 7]
Video Codec	frame rate $\geq$ 30fps	[0, 7]
Rotator	frame rate $\geq$ 30fps	[0, 7]
JPEG	frame rate $\geq$ 10fps	[0, 7]
Camera	buffer occupancy $\leq$ 90%	[3, 7]
Display	buffer occupancy $\geq$ 10%	[3, 7]
GPS	16KB/55KB in 500μs/900μs	[4, 7]
WiFi	read B/W $\geq$ 33MB/s	[3, 7]
USB	total B/W $\geq$ 370MB/s	[2, 7]
Modem	total B/W $\geq$ 700MB/s	[3, 7]
Audio	average latency $\leq$ 1μs	[0, 7]
CPU	N/A	2

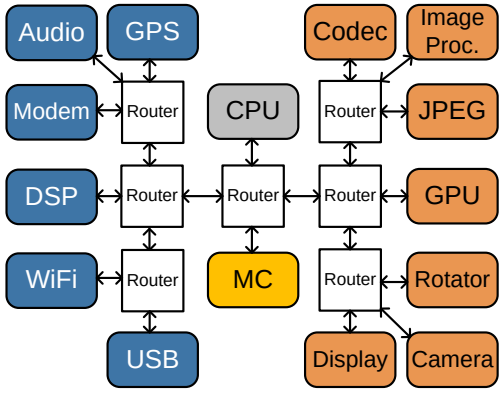


Figure 3.6: Simulated topology of the on-chip network.

to heterogeneous cores. Runtime configuration of priority-based adaptation is not enabled. As for memory traffic, two test cases based on the camcorder dataflow (Fig. 3.2) will be used for demonstration. We will also show that row-buffer hits optimization can be performed efficiently within SARA framework without performance degradations. Furthermore, the SARA framework will be evaluated with runtime configuration of NPI translation functions without any preliminary tuning.

3.5.1 Methodology

The proposed framework is modeled as in Fig. 3.3, where memory traffic from every DMA in heterogeneous cores is generated based on memory specifications of a state-of-the-art MPSoC [39]. Our memory traffic generator models distinct memory access patterns of heterogeneous cores, including memory access rate, address access pattern, outstanding request count and request size etc.. The emulated memory traffic streams are injected into a cycle-accurate on-chip network simulator. The simulated network topology is shown in Fig. 3.6. Lookahead routing is applied in NoC routers with four-stage internal pipelines [9]. As for VC and switch allocations, separable allocators are implemented to perform priority-based arbitration. As the sink for all traffic streams, the memory controller and DRAM are simulated by DRAMSim2 [52] which is integrated into our network simulator. LPDDR4 timing model is applied in DRAMSim2. Tab. 3.1 shows detailed simulation settings. Tab. 3.2 lists the emulated cores and their target performance. The target performance for each core is set according to the camcorder dataflow (Fig. 3.2) running at 30fps. For instance, the frame rotator writes and reads 1080p YUV420 images at 30fps, consuming 89MB/s for write and read requests respectively and 178MB/s in total. For the CPU, we have four CPU cores, each of which is running an application from SPEC2000/2006 benchmark, including mcf, art, sjeng and sphinx3.

3.5.2 SARA Framework with Pre-tuned Configuration

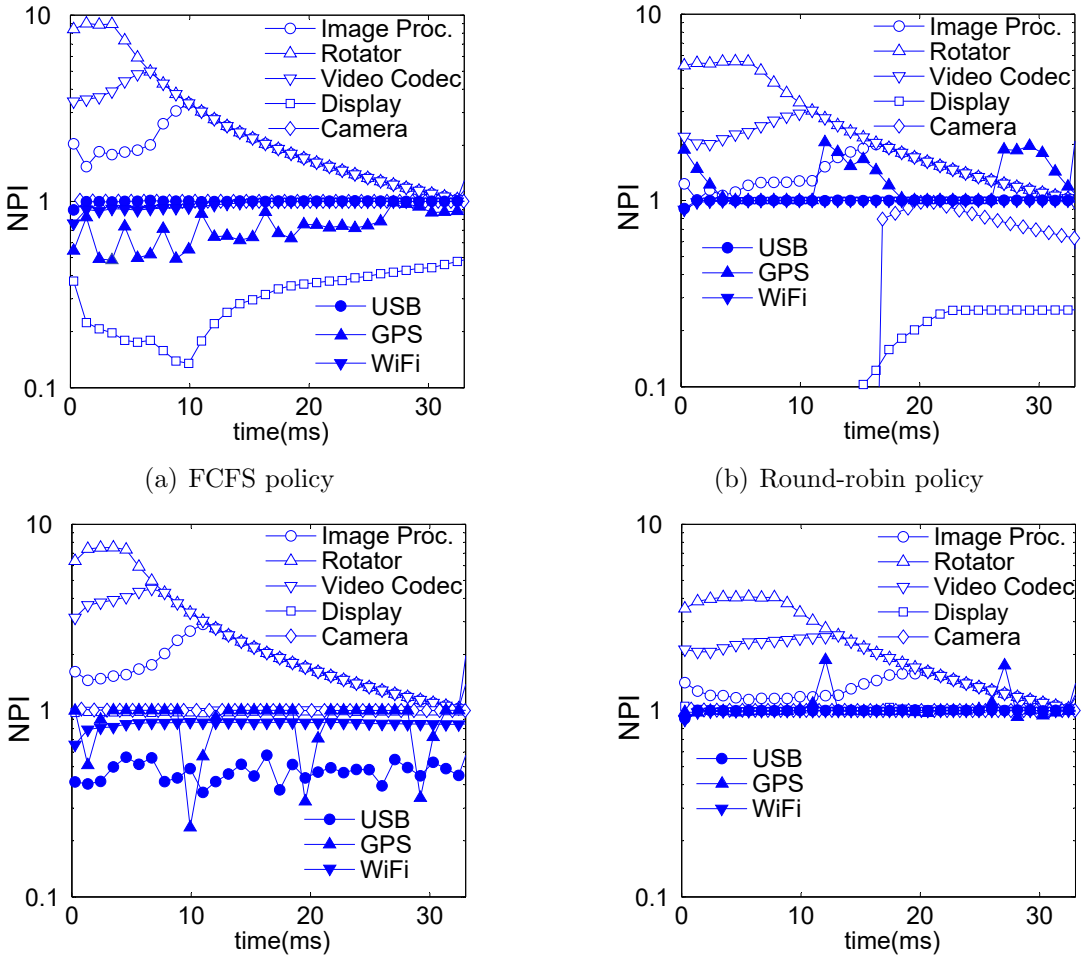
Delivering Target Performance

To begin with, we test the SARA framework in delivering target performance to heterogeneous cores with pre-tuned NPI translation functions. Pre-tuning avoids the necessity of runtime configuration so that the efficacy of SARA framework can be demonstrated without the intricacy of heterogeneous NPI translations. Tab. 3.2 lists the

pre-tuned priority ranges of heterogeneous cores. Their priority ranges are tuned to reflect their sensitivities with respect to memory allocation. Robust real-time cores are initialized with the full priority range from 0 to 7, while failure-prone cores are assigned with higher priority ranges according to their sensitivities. As a best-effort core, the CPU is assigned with a constant priority level of 2 for this test.

For comparison, four arbitration policies are used in the memory controller and on-chip network arbiters, including first-come-first-serve (FCFS), round-robin (RR), a frame-rate-based QoS policy [22, 51] and the priority-based QoS policy (Policy 1). FCFS policy serves all the transactions according to the arrival order. Round-robin policy separates transactions into different queues and serves them in a round-robin fashion. In the memory controller, we have five transaction queues respectively designated to the CPU, the GPU, the DSP, media cores and system cores. Round-robin policy also applies to on-chip network arbiters, as input queues are served in turn. The frame-rate-based QoS policy prioritizes media cores when they are missing real-time deadlines, but otherwise, the policy provides best-effort service to latency-sensitive cores. Furthermore, the priority-based QoS policy compares priority levels for arbitration and uses round-robin as the tiebreaker. The NPI of critical cores during a frame period are shown in Fig. 3.7 when test case A is applied. As explained in Section 3.3.2, the NPI metric reflects performance as higher value indicates better performance. When NPI value drops below 1, it means the target performance is not achieved.

Without reordering memory requests, FCFS policy ends up spending most of the time serving cores consuming high bandwidth. That easily leads to the starvation of latency-sensitive cores. As shown in Fig. 3.7(a), the NPI of the GPS drops below 1 because the GPS is overwhelmed by other system cores sharing the same interconnect, such as the USB. For media cores, the video codec, the rotator and the image processor have all the frame data available at the beginning of a frame period and thus create bursty traffic,



(a) FCFS policy

(b) Round-robin policy

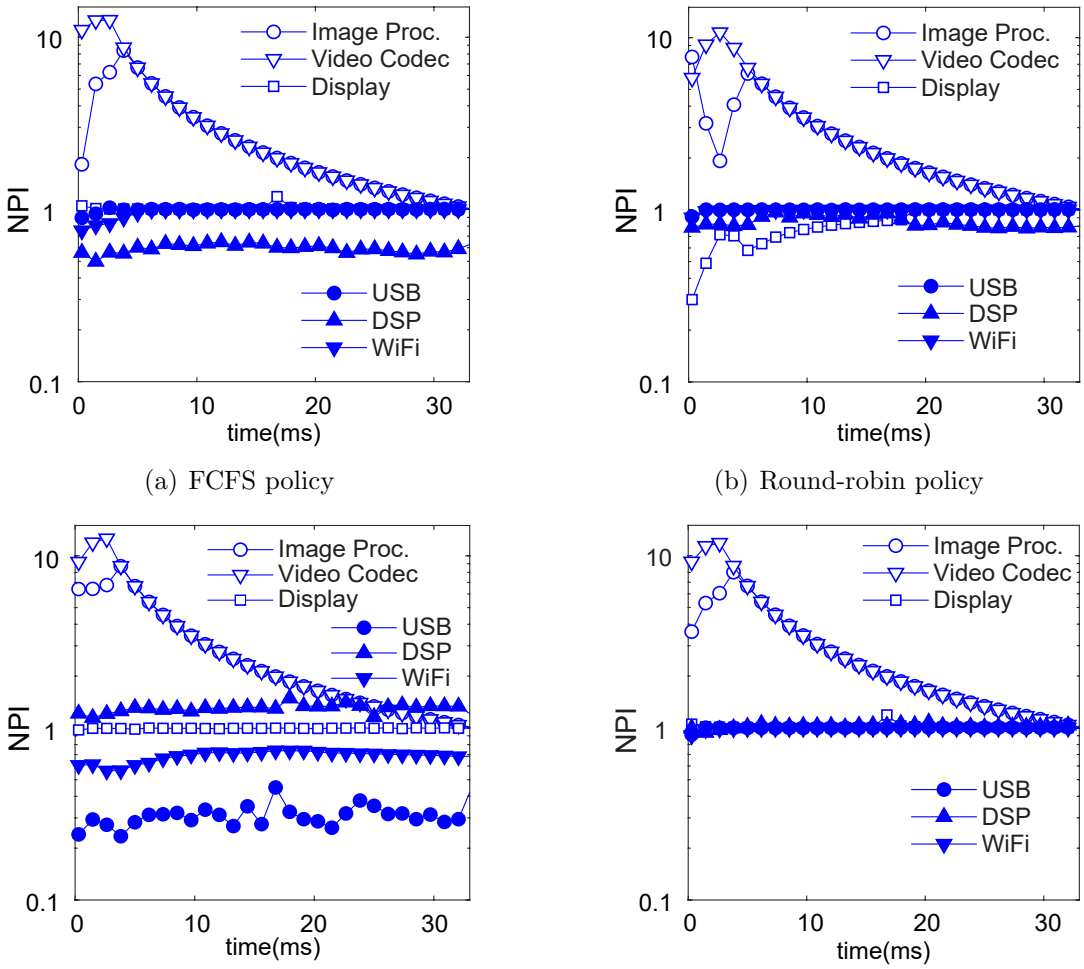
(c) Frame-rate-based QoS policy

(d) Priority-based QoS policy

Figure 3.7: NPI value of critical cores during one frame period (33ms) for test case A with different arbitration policies.

meanwhile the camera and the display generate and consume data at constant rates which are determined by image sensor and LCD panel. In Fig. 3.7(a), media cores with bursty traffic obtain most of the bandwidth in the beginning, resulting in high NPI value. On the other hand, the display fails to achieve the target performance. The display's NPI drops as low as 0.13 which means only 13% of the target performance is achieved.

When round-robin policy is applied, the competition among media cores becomes more intense since they share the same transaction queue in the memory controller. In Fig. 3.7(b), the display and the camera both fail due to the interference from other media



(a) FCFS policy

(b) Round-robin policy

(c) Frame-rate-based QoS policy

(d) Priority-based QoS policy

Figure 3.8: NPI value of critical cores during one frame period (33ms) for test case B with different arbitration policies.

cores. Less than 10% of their target performance is achieved in the worst case. In the meantime, all the system cores meet their target performance because they avoid the interference from media cores by using a separate transaction queue.

The frame-rate-based QoS policy helps all media cores achieve NPI value above 1 in Fig. 3.7(c). However, all system cores fail due to the absence of adaptations for the cores with different QoS targets other than frame rates.

In Fig. 3.7(d), all the cores reach their target performance when QoS-aware scheduling

is performed, because priority-based adaptations help arbiters serve the cores in urgent needs. Note that the NPI of the other cores such as the GPU are not shown because no failure is observed from these cores.

The results by test case B are shown in Fig. 3.8. Similar to Fig. 3.7, the latency-sensitive DSP suffers when FCFS policy is adopted (Fig. 3.8(a)). When round-robin policy is applied (Fig. 3.8(b)), the DSP suffers less since it has its own transaction queue, while the display fails due to the increased interference from other media cores sharing the same transaction queue. Again, the frame-rate-based QoS policy fails to serve non-media cores. At last, the dynamic priorities help the memory system deliver target performance to all cores (Fig. 3.8(d)).

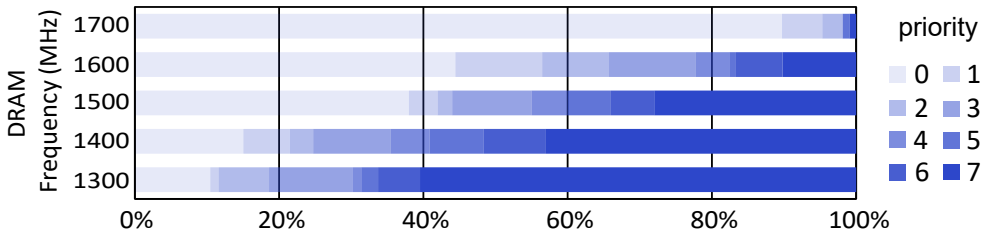


Figure 3.9: Distributions of the image processor's priority levels during one frame period (33ms) with respect to different DRAM frequencies.

Next, we take the image processor from test case A as an example to examine the priority-based adaptation in a single core. Fig. 3.9 shows the distributions of the image processor's priority levels during one frame period, while DRAM frequency decreases from 1700MHz to 1300MHz. Each horizontal bar is designated to a certain DRAM frequency. In a single bar, each block represents the percentage of time during which a certain priority level is adopted. Different shades of blue represent different priority levels, as higher priority levels in darker shades. As shown in Fig. 3.9, when DRAM frequency is set to 1700MHz, for 90% of the time the image processor is adapted to the priority of 0. As frequency decreases, less memory requests can be processed by DRAM. More memory interferences and competitions happen as the result. To maintain target bandwidth, the self-adaptation

leads to a gradual increase in priority levels, which can be observed through the increasing area of blocks in dark shades. When DRAM frequency is lowered to 1300MHz, the image processor has the priority of 7 for 60% of the time. In addition, as frequency decreases, the average bandwidth of the image processor remains above target bandwidth thanks to the priority-based adaptation.

Row-buffer Hits Optimization

As explained in Section 3.3.3, row-buffer hits optimization helps improve available DRAM bandwidth. With the knowledge of heterogeneous cores' urgency levels, the memory controller in the SARA framework is capable of optimizing row-buffer hits without degrading system performance.

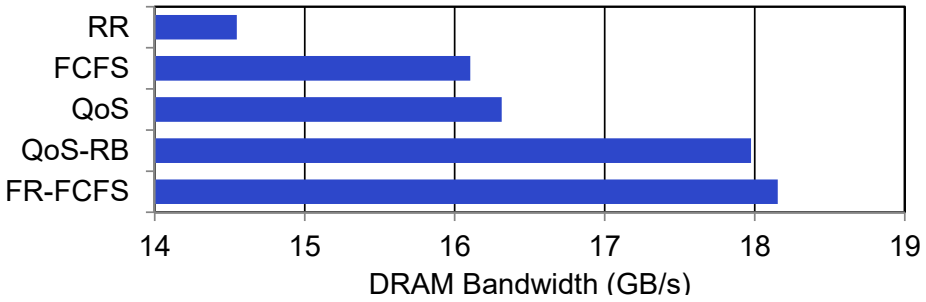


Figure 3.10: Summary of average bandwidth when different scheduling policies applied.

For comparison, we compare with FR-FCFS which prioritizes transactions going to open rows whenever it is possible, and otherwise schedules transactions based on FCFS. FR-FCFS policy is expected to achieve the most row-buffer hits and the highest DRAM bandwidth. Fig. 3.10 shows the average DRAM bandwidth during one frame period when test case A is applied. Four memory scheduling policies are tested, including RR, FCFS, QoS (Policy 1), QoS-RB (Policy 2) and FR-FCFS. Fig. 3.11 shows the NPI of critical cores as QoS-RB and FR-FCFS are adopted.

As expected, FR-FCFS policy achieves the highest bandwidth, whereas performance

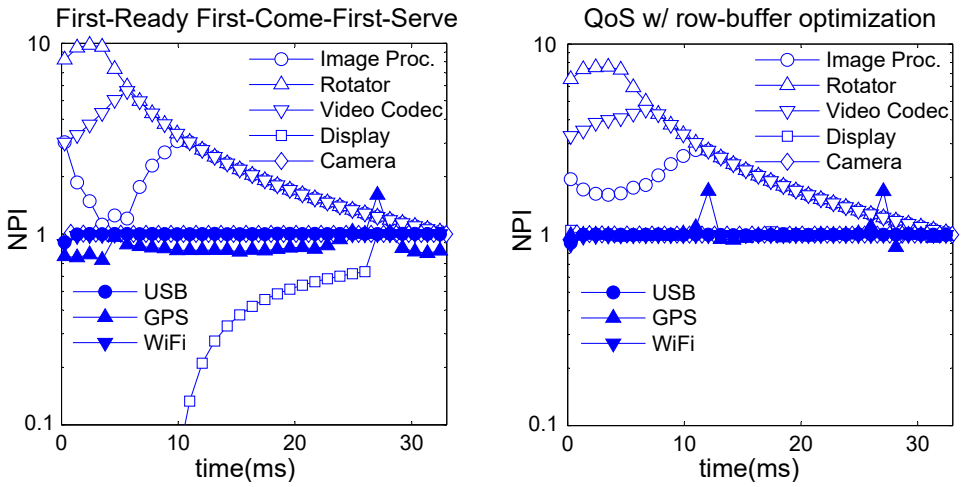


Figure 3.11: NPI value for test case A with respect to FR-FCFS and QoS-RB scheduling policies.

degradations happen to the GPS and the display as the expense. The bandwidth by QoS-RB is slightly lower (by 1%) than FR-FCFS, but much higher than other policies. Specifically, the average DRAM bandwidth obtained by QoS-RB policy is 24%, 12% and 10% higher than RR, FCFS and QoS policies respectively. In the meantime, no performance degradations are caused to heterogeneous cores.

3.5.3 SARA Framework with Runtime Configuration

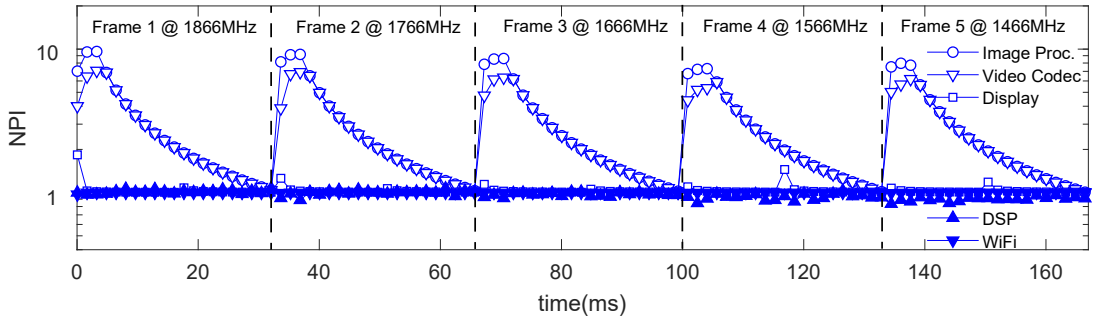
We have shown that SARA framework helps all real-time cores to achieve their target performance given pre-tuned NPI translation functions for each core. However, pre-tuning requires tremendous effort during design time to consider all potential application scenarios. To spare the effort, we have proposed an alternative solution which is a two-stage runtime configuration scheme. In this section, we will demonstrate its effectiveness in comparison to SARA framework without runtime configuration.

As shown in Fig. 3.5, best-effort service is provided to the CPU while guaranteed services are allocated to other real-time cores. Test case A is applied in this evaluation. All real-time cores with QoS targets are initialized with the priority range $[0, 7]$, while

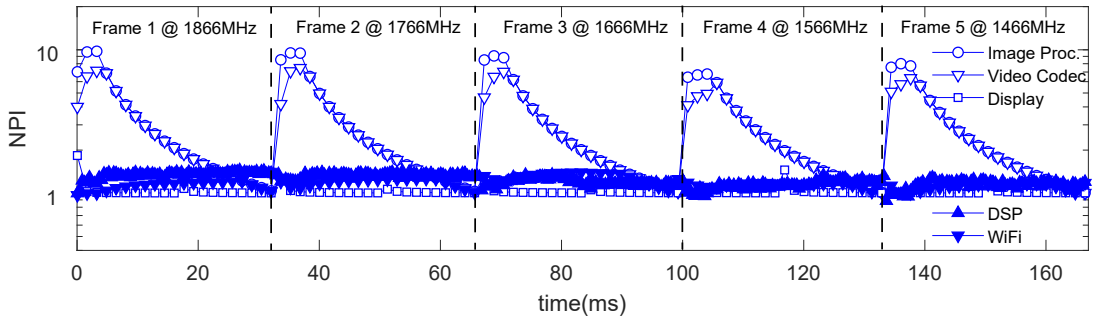
CPU priority level starts with 0. The priority-based QoS policy as in Policy 1 is used in the memory system. For the demonstration of runtime self-configuration, five frames are simulated with different DRAM frequencies. DRAM frequency starts with 1866MHz in the first frame and lowers by 100MHz in each following frame. Although not all of the emulated frequencies are supported by LPDDR4, reducing memory frequency is a straightforward way to intensify memory competitions in runtime for the demonstration of self-configuration.

NPI values of major cores over five frame periods are shown in Fig. 3.12. For comparison, four configuration schemes are implemented, including fully-functional runtime configuration with the DSC stage and the GR stage (Fig. 3.12(d)), semi-functional configuration schemes with the DSC stage and static priority levels ($P_{CPU} = 0$ or 4) for the CPU (Fig. 3.12(c) and 3.12(b)), and no runtime configuration at all with CPU priority at 4 (Fig. 3.12(a)). As can be seen, most real-time cores maintain their NPIs above 1 regardless of DRAM frequency and configuration scheme, which means the full priority range $[0, 7]$ enables effective priority-based adaptation at most cores. However, NPI curves by the DSP show much more variations as different frequencies and configuration schemes are applied. This has to do with the fact that the DSP is highly subject to QoS failure given the same priority range as multimedia cores. Without runtime configuration, as shown in Fig. 3.12(a), the DSP often suffers from $NPI < 1$ which is particularly obvious during the last two frames. With runtime configuration, as shown in Fig. 3.12(b)-(d), failure rate of the DSP is much reduced while its NPI is improved by different extents depending on CPU priority. Note that the figures are drawn at the millisecond timescale to include the total simulation time span of five frames. Distributed self-configuration actually occurs on the order of hundreds of nanoseconds.

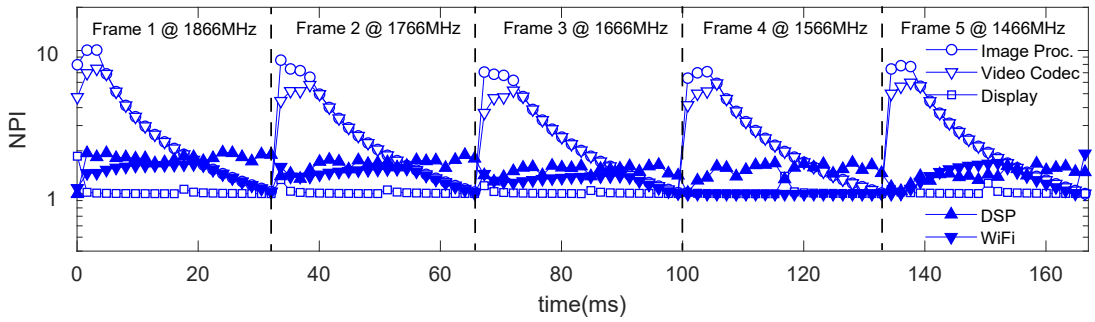
Tab. 3.3 lists the priority range, the average priority level and failure rate of the DSP at the end of each frame. When DSC is disabled, frame rate of the DSP is far beyond



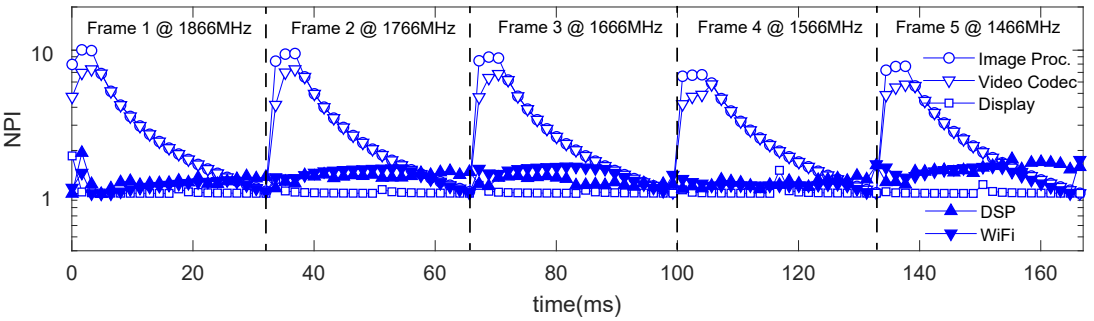
(a) No DSC and $P_{CPU} = 4$



(b) DSC and $P_{CPU} = 4$



(c) DSC and $P_{CPU} = 0$



(d) DSC and GR

Figure 3.12: NPI value of critical cores during five frame periods for test case A. A different DRAM frequency is set during each frame, ranging from 1866MHz to 1466MHz.

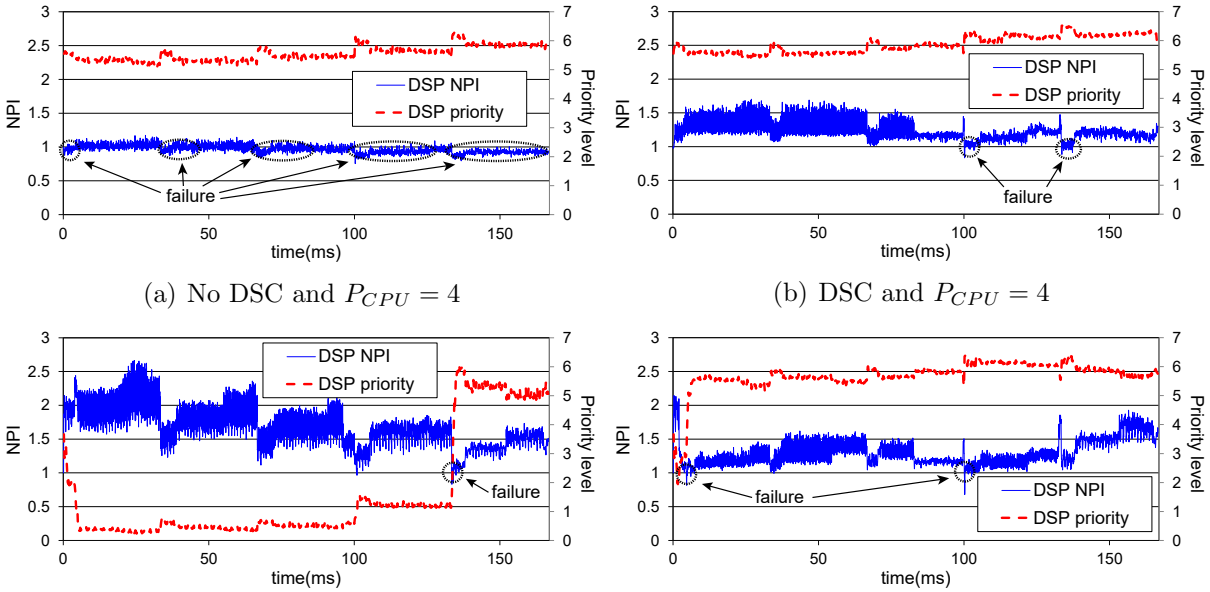
Table 3.3: Results of self-configuration by the DSP per frame, including the final priority range, the average priority level over current frame period and the overall failure rate during each frame.

Frame number	No DSC and $P_{CPU} = 4$			DSC and $P_{CPU} = 4$		
	Range	P_{ave}	FR	Range	P_{ave}	FR
1	[0,7]	5.33	37.11%	[3,7]	5.58	0.66%
2	[0,7]	5.35	46.56%	[3,7]	5.59	0.00%
3	[0,7]	5.50	76.81%	[3,7]	5.82	0.00%
4	[0,7]	5.70	95.05%	[4,7]	6.11	2.95%
5	[0,7]	5.90	99.76%	[4,7]	6.24	4.37%
Frame number	DSC and $P_{CPU} = 0$			DSC and GR		
	Range	P_{ave}	FR	Range	P_{ave}	FR
1	[0,1]	0.60	0.00%	[2,7]	5.09	1.32%
2	[0,1]	0.45	0.00%	[3,7]	5.60	0.02%
3	[0,1]	0.53	0.00%	[3,7]	5.76	0.00%
4	[0,2]	1.22	0.16%	[4,7]	6.10	1.48%
5	[2,7]	5.15	1.47%	[4,7]	5.86	0.00%

3% most of the time. When CPU priority is set at 4, DSC helps to improve the priority level of the DSP by raising the lower bound of priority range, resulting in higher DSP priority on average and much lower failure rate. However, without global regulation, setting CPU priority constantly at 4 still leads to increasing failure rate at the DSP which is up to 4.37% during the last frame. This is because as DRAM frequency decreases, memory bandwidth becomes more limited and the priority of the CPU is no longer appropriate. When CPU priority is set at 0, the DSP experiences fairly low failure rate. Due to the light competition pressure from the CPU, the DSP lowers Its priority range in most frames to release more resources for other cores. When DSC and GR are both applied, failure rate of the DSP remains below 3% in all frames.

To provide more insights into how runtime configuration impacts the DSP and the CPU simultaneously, Fig. 3.13 shows NPI and priority level of the DSP under four different configuration schemes, and Fig. 3.14 shows bandwidth and priority level of the CPU during the same time.

As mentioned, without DSC (Fig. 3.13(a)), DSP NPI falls below 1 most of the time.



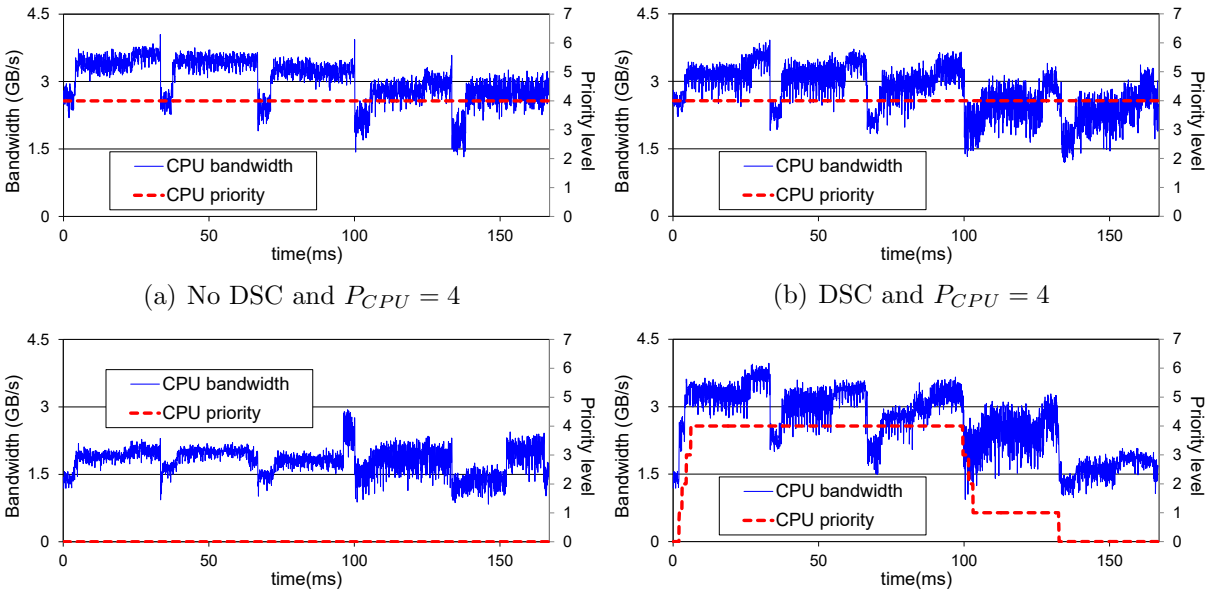
(a) No DSC and $P_{CPU} = 4$

(b) DSC and $P_{CPU} = 4$

(c) DSC and $P_{CPU} = 0$

(d) DSC and GR

Figure 3.13: Runtime NPI and priority level of the DSP with respect to different configuration schemes for priority-based adaptation.



(a) No DSC and $P_{CPU} = 4$

(b) DSC and $P_{CPU} = 4$

(c) DSC and $P_{CPU} = 0$

(d) DSC and GR

Figure 3.14: Runtime bandwidth and priority level of the CPU with respect to different configuration schemes for priority-based adaptation.

When DRAM frequency is reduced to 1466MHz, DSP NPI remains below 1 during the whole frame period. Meanwhile, as shown in Fig. 3.14(a), the CPU achieves the highest bandwidth among four schemes at the expense of DSP NPI, averaging 3.14GB/s over five frames.

When DSC is enabled and CPU priority is still set at 4 (Fig. 3.13(b) and 3.14(b)), DSP NPI settles above 1 most of the time except at the beginning of the last two frames. This is because multimedia cores tend to generate bursty traffic during this time, causing major memory interference to latency sensitive cores. During the same time, CPU bandwidth sees a subtle decrease due to the increased priority level of the DSP. The average CPU bandwidth is lowered to 2.89GB/s.

When DSC is enabled and CPU priority is lowered to 0 (Fig. 3.13(c) and 3.14(c)), the DSP achieves the highest NPI among all configuration schemes, whereas the CPU suffers from the lowest bandwidth which is 1.86GB/s on average.

Finally, when DSC and GR are both applied (Fig. 3.13(d) and 3.14(d)), CPU priority is dynamically adjusted in response to the decreasing DRAM frequency. As the result, DSP failure rate stays below 3% in all frames and the average CPU bandwidth is 2.75GB/s, merely 5% lower than the case where CPU priority is statically set at 4.

In summary, DSC helps the DSP to settle at the healthy state by adjusting its priority range during runtime. Meanwhile, GR increases the bandwidth allocated to the CPU without causing starvation at the DSP. When the proposed runtime configuration scheme is fully enabled, the DSP and the CPU can both be properly handled by the SARA framework.

3.6 Chapter Summary

In this work, we proposed the self-aware resource allocation (SARA) framework for memory management in heterogeneous systems. Lightweight performance meters are distributed in each core to monitor end-to-end QoS with low cost. Priority-based adaptation allows cores to adjust their priority levels according to the observed performance. The memory system with non-partitionable resources responds to QoS requests by performing priority-based management without tremendous overhead. Experimental results show that with properly tuned priority-based adaptation, the SARA framework helps all the heterogeneous cores to achieve their target performance. By comparison, without using priorities, performance of critical cores can drop lower than 10% of the target. Furthermore, memory performance such as row-buffer hits can be optimized within the SARA framework. Up to 24% improvement of total bandwidth is obtained without degrading QoS. Finally, we introduced a runtime configuration scheme for priority-based adaptation in the SARA framework to accommodate best-effort cores and real-time cores that are particularly prone to QoS failure. In our evaluation, the runtime configuration scheme successfully adjusts priority-based adaptation to reduce failure rate and allocates resource to the best-effort core without causing QoS failure.

3.7 Acknowledgement

This chapter, in part, has been submitted for publication of the material as it may appear in ACM Transactions on Architecture and Code Optimization, 2019. Song, Yang; Alavoine, Olivier; Lin, Bill. This chapter also includes the material as it appears in proceedings of ACM/IEEE Design Automation Conference, 2018. Song, Yang; Alavoine, Olivier; Lin, Bill. The dissertation author was the primary investigator and author of these papers.

Chapter 4

Single-Tier Virtual Queuing Memory Controller

4.1 Introduction

In the previous chapter, we have introduced the self-aware memory allocation framework that adopts priority-based adaptation to translate NPI into priority levels at each heterogeneous core. In response, the memory subsystem implements priority-based arbitration inside of each memory fabric. Even though the SARA framework provides an effective foundation for memory management, we still need architectural support from the memory subsystem to enable efficacious memory response. Therefore, in the rest of this dissertation, we will investigate micro-architecture designs of memory fabrics. Particularly, in this chapter, we will study the design of QoS-aware memory controllers.

As the last memory fabric that memory requests visit before accessing DRAM, the memory controller has direct and substantial impact on the memory efficiency in DRAM. Thus early studies on the memory controller were mostly focused on maximizing memory throughput [42]. Along with the prevalence of multicore systems, new challenges

are presented to memory controller design, among which the most difficult one is to relieve memory interference among different traffic streams. In a heterogeneous MPSoC, as the final merging point of heterogeneous traffic streams, the memory controller is where the most memory interference takes place. The interference typically happens between bandwidth-consuming agents and latency-sensitive agents. As an example, the GPU is capable of executing multiple threads in parallel, which often leads to a large number of memory requests and high bandwidth consumption. Additionally, the GPU switches between different threads to hide memory access latency. In contrast, the CPU is highly sensitive to memory latency because memory installs are detrimental to its performance. When the CPU and the GPU share the same memory controller, CPU traffic can be severely delayed by numerous requests from the GPU unless special care is given to the QoS of the CPU. Hence QoS-aware memory controllers are needed to minimize the interference and deliver QoS targets set by heterogeneous cores.

A classic memory controller for multicore platforms comprises a two-tier queuing system, including a per-source queuing transaction stage and a per-bank queuing command stage. The two-tier queuing system is a straightforward solution to deliver high memory throughput. To accommodate heterogeneous traffic, Ausavarungnirun et al. [4] were the first to propose the staged memory scheduler (SMS) in the context of integrated CPU-GPU systems (Fig. 4.1). The proposed SMS architecture also follows the two-tier queuing system approach. In SMS, transactions for the same row-buffer in DRAM are formed into batches to increase row-buffer hits. Due to the lower spatial locality, CPU traffic generates shorter batches. By adopting the Shortest Job First (SJF) policy with certain probability, the scheduler prevents CPU traffic from being overwhelmed by the vast GPU traffic. However, the command stage still uses conventional FIFOs, which degrades the QoS improvement from the transaction stage. In particular, after transactions have been translated into commands and forwarded to the second stage, newly arriving memory transactions that

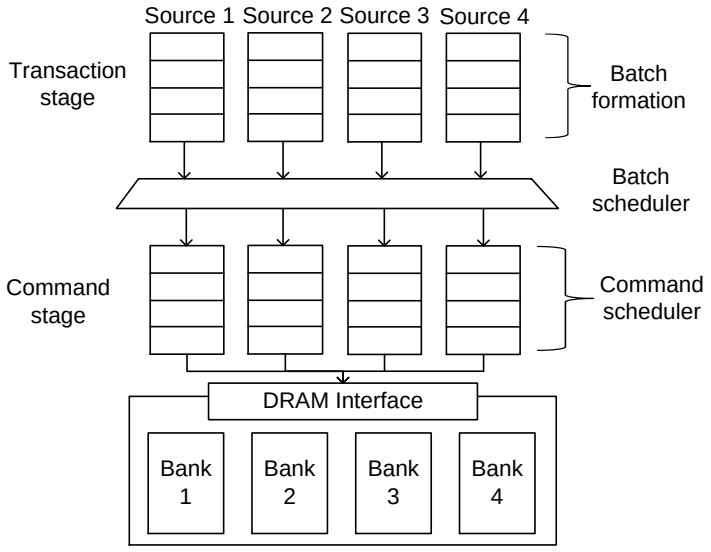


Figure 4.1: Architecture of the two-tier staged memory scheduler.

may be more critical from a QoS perspective cannot be served ahead of those that are already queued at the second stage. As a concurrent work, Jeong et al. [22] introduced a real-time QoS-aware scheduling algorithm for the transaction stage, but it also neglects the queuing effects at the command stage. In addition, various QoS-aware schedulers have been proposed so far on basis of the two-tier queuing architecture.

The limitation of previous work using two-tier queuing systems lies in the unawareness of queuing delays in the command stage which can be much higher than that in the transaction stage. Performance improvement by the transaction scheduler may not be preserved after the command stage. This problem with two-tier queuing is illustrated in Fig. 4.2. The figure depicts a common scenario in which the GPU transaction queue has transactions to send but the CPU has no transactions available. As there is no CPU transaction waiting, a QoS-aware scheduler in a two-tier queuing system would send all the GPU transactions queued at the transaction stage to the command stage in order to maximize memory throughput. Yet any new incoming CPU transaction going to the same queue would have to wait at the end of the same command queue, which could cause

potentially substantial latency to the CPU memory traffic.

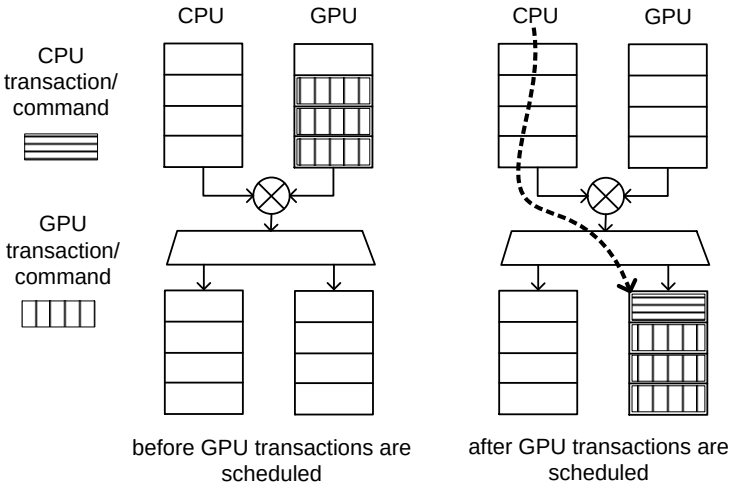


Figure 4.2: An example of transaction scheduling in a two-tier memory controller leading to queuing delay to the CPU.

The problem is further illustrated in Fig. 4.3[1] where queuing delays in the transaction and command stages of a two-tier system are shown respectively. For comparison, three scenarios are simulated: one only has a dual-core CPU; another has a dual-core CPU, a graphic GPU, and a display controller, and uses the SJF policy for scheduling; the last one also has a dual-core CPU, a graphic GPU, and a display controller, but uses the Round-Robin (RR) policy for scheduling. Even though the SJF policy successfully reduces the queuing delay of CPU transactions, the queuing delay of CPU commands remains the same with the result from RR policy, which is much higher than the command delay without real-time cores. In other words the benefit introduced by the SJF policy in the first stage is counteracted by the FIFO queues in the second stage. Therefore, we say that a two-tier queuing system is *not efficacious* in the sense that a QoS-aware scheduler may not produce the desired result.

In this chapter, we propose the Single-Tier Virtual Queuing (STVQ) memory

[1] *mcf-art* and *coc* benchmarks are used respectively for the CPU and the GPU. Experiment setup will be discussed in Section 4.4.

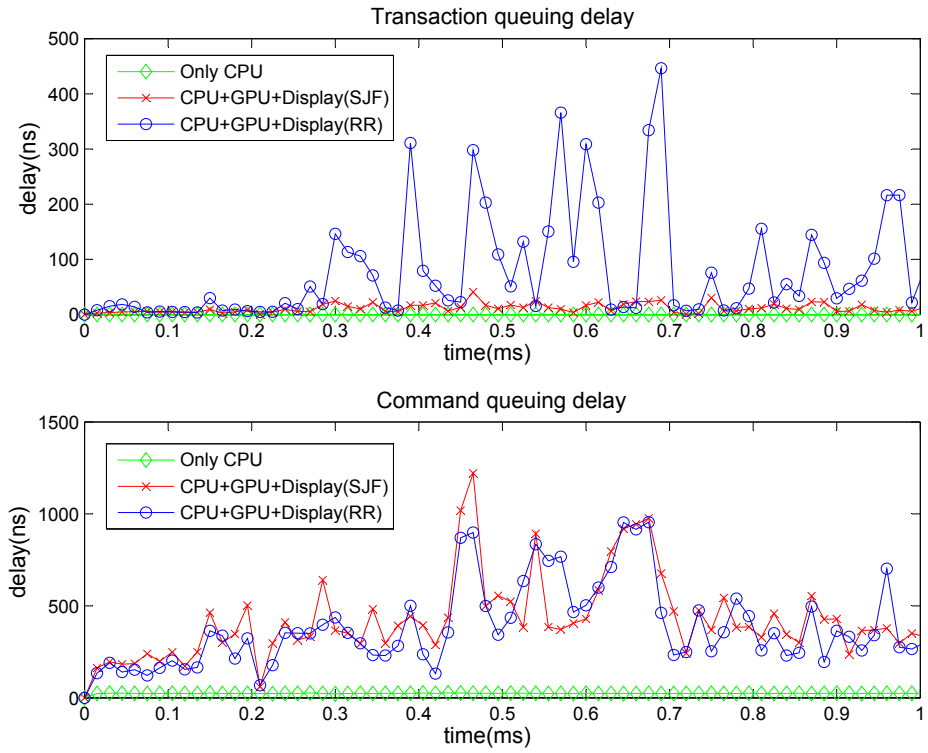


Figure 4.3: Queuing delays of CPU traffic in the transaction and command stages of a two-tier memory controller.

controller to enable efficacious memory scheduling. Without loss of generality, we use memory traffic of the CPU and multmedia cores for our evaluations, because their traffic patterns and QoS targets are representative among heterogeneous cores. In particularly, the CPU is a best-effort core sensitive to latency, whereas multimedia cores request guaranteed services with high bandwidth consumption. The contributions of our work are summarized as follows.

- We identify the bottleneck of two-tier queuing memory controllers in relieving memory interference and delivering target performance to heterogeneous cores with disparate traffic patterns and QoS targets.
- We propose the single-tier virtual queuing memory controller without command queuing delays to enable efficacious memory scheduling.

- We adopt separable allocation to implement memory scheduling in the proposed memory controller in order to achieve high scalability.

- We evaluate the proposed design and show that the STVQ memory controller successfully reduces slowdown of the CPU caused by the memory interference from real-time cores. Compared with prior QoS memory controllers, our approach reduces CPU slowdown by up to 13.9%, with an average reduction of 6.1%, while ensuring that all real-time cores are able to meet their frame rate targets.

The rest of this chapter is organized as follows: Section 4.2 briefly reviews related work. Section 4.3 describes our proposed single-tier virtual queuing memory controller architecture. Experimental results and conclusions follow in Sections 4.4 and 4.5, respectively.

4.2 Related Work

FR-FCFS policy [42] has been widely used for memory scheduling to deliver high memory throughput, but it assigns more priorities to memory-intensive applications as it prioritizes requests that result in high row-buffer hits. Application-aware scheduling for CPU-only systems has drawn more attention in recent years [27, 28, 17]. ATLAS [27] prioritizes applications with low memory intensity to improve system throughput at the expense of applications with high memory intensity. Thread cluster memory scheduling [28] dynamically clusters applications into low and high memory-intensity clusters and achieves high system performance and fairness simultaneously. Dual-criticality memory controller [17] handles memory contention between real-time and high performance applications by bank separation, assuming no shared memory between different types of applications. Minimalist Open-page policy [25] adopts a fair DRAM address mapping scheme to avoid row-buffer locality starvation and to increase bank level parallelism. On basis of that

the scheduler prioritizes requests with a higher criticality, and then it prioritizes requests that are hitting an open page. Similarly, parallelism-aware batch scheduling [34] batches requests in time order before prioritizing row-hit requests within each batch to circumvent starvation.

Staged memory scheduler [4] was the first application-aware scheduler proposed for CPU-GPU systems. A two-tier staged memory architecture (Fig. 4.1) is used to decouple the memory scheduling task into three basic steps, including batch formation, batch scheduling and command scheduling. The batch scheduler switches between two policies which are the Shortest Job First (SJF) and Round-Robin (RR). Due to the distinct characteristics of CPU and GPU memory traffic, SJF policy gives higher priority to the CPU while RR policy favors the GPU. By adjusting the probability of SJF policy, the priority of CPU is modulated to balance system performance and fairness for the CPU-GPU system. However, even though the CPU can achieve sufficient bandwidth through the batch scheduler, there is no guarantee for the latency because the queuing delay in the FIFO command scheduler is uncertain.

In [22], a QoS-aware scheduling policy was proposed to dynamically balance CPU and GPU bandwidth. The proposed policy allocates the minimum bandwidth to the graphic GPU to meet the target FPS. To attain that, two notions for GPU's frame progress were introduced by (4.1),

$$\begin{aligned} P_A &= \frac{number\ of\ tiles\ rendered}{total\ number\ of\ tiles}, \\ P_E &= \frac{elapsed\ time\ in\ current\ frame}{target\ frame\ time}. \end{aligned} \tag{4.1}$$

Here, P_A is the actual progress of GPU workload in the current frame and P_E is the expected progress. Their QoS policy prioritizes the GPU when P_A is lagging behind P_E. Progress notions can be extended to include other real-time cores such as an HD decoder

whose progress metric can be expressed as follows:

$$P_A = \frac{number\ of\ bytes\ (or\ pixels)\ processed}{total\ number\ of\ bytes\ (or\ pixels)}. \tag{4.2}$$

Nonetheless, this QoS policy focuses on transaction-level scheduling and optimizes CPU bandwidth instead of CPU latency, a more critical performance metric. This may cause performance degradation in a memory scheduler with the two-tier queuing system. Similarly, a deadline-aware scheduler was introduced in [51] to translate target frame rates into real-time deadlines for multimedia cores. The proposed scheme bares the same limitation as in [22].

4.3 Single-Tier Virtual Queuing Memory Controller

The STVQ memory controller for a DRAM channel with N memory banks and K independent traffic flows is shown in Fig. 4.4. Four major components reside in the STVQ controller, including single-tier virtual queues (VQ), bank arbiter, transaction scheduler and per-bank command generators. In this section, we will go through design details of the proposed architecture.

4.3.1 Interface for Incoming Traffic Flows

At the input of the memory controller, incoming memory traffic flows from different sources are separated so that QoS-aware scheduling policies can be applied by the transaction scheduler. A real-time core can generate more than one independent traffic flows. In a tile-based graphic GPU, a complete frame is divided into a few tiles which can be rendered in parallel. A typical rendering procedure goes through the geometry shader, rasterizer

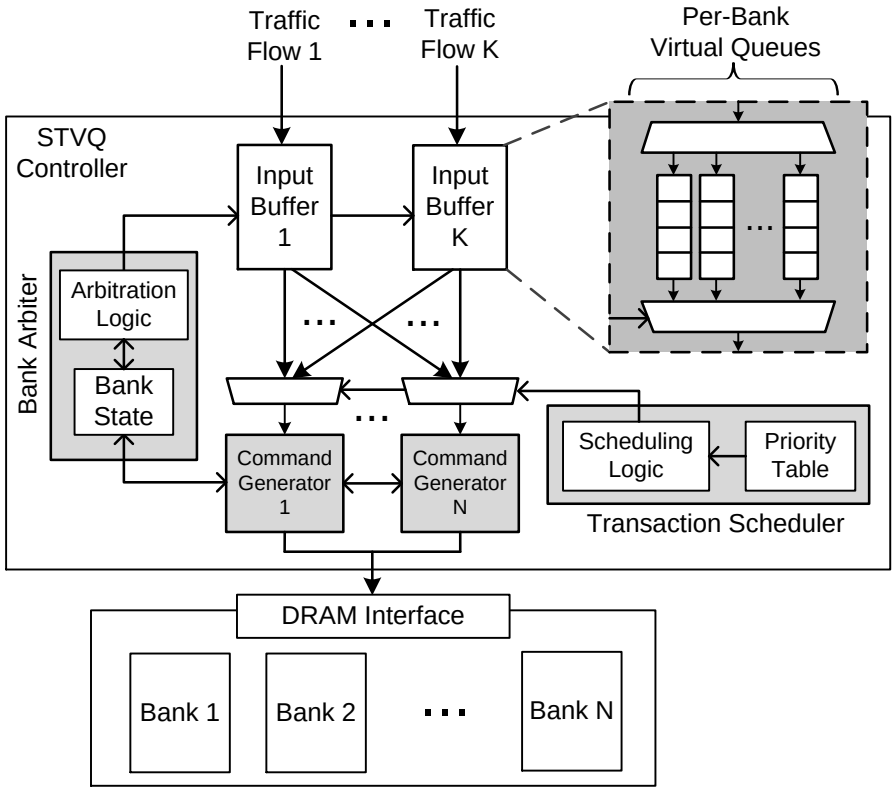


Figure 4.4: Architecture of the single-tier virtual queuing memory controller.

and fragment shader etc. until the tile cache is flushed into the framebuffer in the external memory. The memory traffic generated during the rendering procedure of the current tile, which is mostly input traffic of texture, does not depend on the flushing traffic of previous tiles. Thus GPU memory traffic can be split into rendering and flushing traffic flows. Similarly, the video codec generates reading and writing traffic flows to meet the target frame rate respectively. The generation of real-time traffic can be shown as ideal pipelines in Fig. 4.5 where the delays between consecutive stages are omitted. Each traffic flow is scheduled in parallel in the memory controller. In our experiment, real-time traffic flows are independent from each other during memory scheduling while traffic from different CPU cores are combined together. Although separating CPU traffic per core would allow differentiated scheduling policies for CPU cores, we do not apply that in this chapter

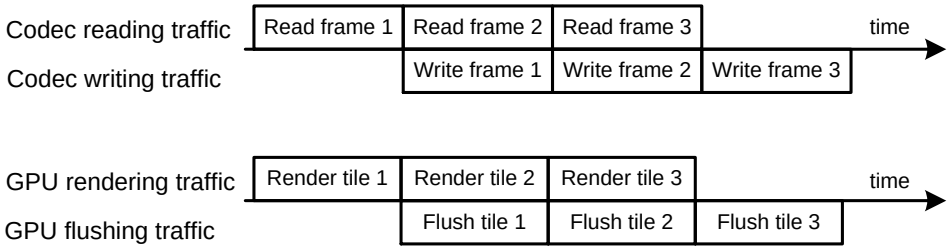


Figure 4.5: Real-time input and output traffic flows of the GPU and video codec.

because our current focus is to reconcile the conflicts between CPU and real-time traffic flows in the memory controller.

4.3.2 Single-Tier Virtual Queues

In the first stage of the STVQ memory controller, transactions from the same traffic flow are stored in the same input buffer. An input buffer contains a series of per-bank virtual queues. Upon arrival, transactions are sorted into virtual queues based on their destination banks. Per-bank virtual queues avoid head-of-line blocking when the transaction scheduler forwards requests to different banks.

Since in every clock cycle, the memory controller can only send one command to DRAM and receive one transaction from each traffic flow, multiple reading or writing does not happen to an input buffer. This allows per-bank virtual queues for the same traffic flow to share one buffer without raising storage cost. In the implementation, the logical partitions for the virtual queues can be statically allocated. In this case, we simply require a simple address lookup table to record the head and tail pointers for each virtual queue in the input buffer. When a transaction for a particular traffic flow arrives at the corresponding input buffer, the address lookup table can be consulted to identify the tail memory location of the corresponding virtual queue based on the destination memory bank. Similarly, when a transaction needs to be removed from the virtual queue, the address

lookup table can be consulted to identify the head pointer. The corresponding head and tail pointers can be updated accordingly when a transaction is removed from or inserted into the corresponding virtual queue.

In a conventional memory controller, the two-tier queuing system guides memory traffic from per-source buffers to per-bank buffers. In spite of its high scalability, it would take extra effort to unify the scheduling and queuing policies in two stages. By consolidating two-tier queues into single-tier queues in the proposed architecture, we enable memory scheduling to be performed effectively without any intermediate delays between the scheduler and DRAM.

4.3.3 Separable Allocation

To schedule a transaction to DRAM, the STVQ controller selects among per-flow per-bank queues, which is an allocation problem where the access to DRAM interface is granted to one of the virtual queues. Similar to *separable allocators* in NoC routers [5], we achieve high scalability by performing allocation with two separable steps: first choosing a memory bank, and then choosing a traffic flow.

In the first step, the bank arbiter assigns DRAM access to one of the memory banks. Then virtual queues for this bank are activated in all input buffers and contend for the designated command generator which will later convert the chosen transaction into DRAM commands. Since the command generator may receive requests from multiple traffic flows, in the second step, the transaction scheduler arbitrates among available traffic flows at the command generator. Only then can the chosen virtual queue send a transaction to the command generator.

By using separable allocation, the STVQ controller solves a simple N:1 or K:1 arbitration problem at a time, where N is the number of banks and K is the number of traffic flows. For example, if there are 4 traffic flows and 8 memory banks, that requires the

STVQ controller to perform an 8:1 arbitration to choose a bank, and then a 4:1 arbitration to select a traffic flow. The use of separable allocation significantly reduces the complexity of the arbitration problem.

4.3.4 Bank Arbiter

In a DRAM device, memory banks operate separately at the same time. For example, bank 1 can be read while bank 2 is being written. Yet since they share the same interface to the memory controller, only one memory bank can receive DRAM command in the same clock cycle. Also, the memory bank is not available for more commands until the current operation has finished. Therefore, the bank arbiter is responsible to find available memory banks and assign one of them with the permission to access DRAM interface.

To achieve this, the arbiter records bank states in a state table and tracks the availability of each memory bank. The arbitration logic selects a winner from available banks based on a certain arbitration policy. In our experiment, the bank arbitration policy does not have notable impact on the performance. Thus we adopt the round-robin policy for simplicity. After arbitration, the bank arbiter sends the bank selection signal to all input buffers. At each input buffer, the selection signal activates the corresponding virtual queue for the winning bank and forwards the request to the command generator. Meanwhile, the command generator designated to the winning bank is also activated.

4.3.5 Transaction Scheduler

The transaction scheduler arbitrates among requests from different traffic flows at the command generator after bank arbitration. Then the winning queue forwards a transaction to the command generator. Prior QoS-aware schedulers or memory management frameworks can be applied here. For example, the STVQ memory controller can be integrated within our SARA framework by using a priority arbiter as the scheduling logic. In that case, a

priority table is needed to record the priority levels of requests residing at the head of all virtual queues. During transaction scheduling, the scheduler checks on the priority table and searches for the request with the highest priority level. After transaction scheduling, if there is a transaction selected and removed from its virtual queue, the priority table will be updated with the change.

4.3.6 Command Generators

After bank arbitration and transaction scheduling, only one transaction arrives at the command generator designated to the destination memory bank. To execute this memory transaction, the command generator generates commands to operate DRAM while following the DRAM memory-access protocol [16] that has been introduced in Chapter 2. Thanks to the command generators, the bank arbiter and transaction scheduler are able to perform allocation without dealing with the intricate DRAM access protocol. In addition, per-bank command generators communicate with the bank state table in the bank arbiter to update the bank availability information. In the STVQ memory controller, a command generator is implemented as a state machine which is similar to the command schedulers in prior work [4, 42, 27, 28, 17].

4.3.7 Hardware Implementation

Virtual Queues

As explained, virtual queues of the same traffic flow share the same input buffer. Therefore, for an STVQ memory controller with N memory banks and K traffic flows, K buffers are needed. By comparison a two-tier memory controller shown in Fig. 4.1 requires $N + K$ buffers because all the queues can be active at the same time. Moreover, command queues in the two-tier system can take up plenty of storage, because every transaction

generates up to three DRAM commands. By avoiding buffering DRAM commands, more storage can be saved in the single-tier design. The storage of an input buffer can be equally allocated to per-bank virtual queues. To record the addresses of N virtual queues in each buffer, N registers are needed. Note that all input buffers can share the same address table for virtual queues.

Bank Arbiter

The logic of bank arbitration is similar to previous memory schedulers. To implement round-robin policy, the arbiter maintains a register recording the last served memory bank. A small state table is used to track availability of memory banks. Each bank has a one-bit flag to indicate its availability.

Transaction Scheduler

Since the arbitration for transactions is performed for a single memory bank each time, one transaction scheduler is enough in our design. As mentioned in Section 4.3.5, to integrate the STVQ controller into the SARA framework, we need a priority table to store the priority levels of memory requests at the head of each virtual queue. If a priority level is encoded in three bits, we need a total of $3NK$ bits in the priority table.

Command Generators

A command generator is implemented for each bank as a state machine performing scheduling based on various timing and power constraints, which is identical to previous schedulers [4, 42, 27, 28, 17]. As suggested in [61] we actually only need four command generators, because no more than four memory banks can be active at the same time due to the constraint t_{FAW}. Four command generators are shared by active memory banks. Yet to implement that more arbitration logic is required to allocate command generators

to active memory banks. Thus in our experiment we use per-bank command generators.

Hardware Cost

The hardware parameters of the STVQ memory controller employed in our experiment are configured as follows. We use an STVQ controller operating DRAM with eight memory banks, and serving six traffic flows including two flows from the graphic GPU, two from the video codec, one from the display control and one from CPU cores. Each virtual queue reserves 8 entries in the input buffer. With 48 virtual queues in total, the STVQ controller can buffer up to 384 transactions which is about the same buffer capacity as in prior two-tier designs [4]. In the priority table, an entry is assigned to each virtual queue and each entry is a priority level encoded three bits. Thus the priority table stores 144 bits in total. Storage overhead by the STVQ controller is summarized in Tab. 4.1.

Table 4.1: Storage overhead by the STVQ controller with six traffic flows ($K = 6$) and eight DRAM banks ($N = 8$).

Item	Storage overhead
Per-bank VQ	8 entries
Input buffer	$N \cdot VQ\_Size = 64$ entries
VQ address table	$N \cdot log_2 Buffer\_Size = 48$ bits
Bank state table	$N = 8$ bits
Priority table	$3NK = 144$ bits

4.4 Evaluation

In this section, the proposed STVQ memory controller will be first evaluated on QoS-aware bandwidth allocation in comparison with previous memory controllers for heterogeneous memory traffic. Then CPU performance will be further examined in terms of memory latency and IPC slowdown rate. Overhead analysis of the STVQ memory controller follows in the end.

Table 4.2: Simulation parameters.

CPU Parameter	Description
Clock speed	1GHz
L1 cache	32KB private 4-way
L2 cache	1MB shared 16-way
GPU Parameter	Description
Unified shader cores	8
Clock speed	800MHz
DRAM Parameter	Description
Volume	4GB
I/O bus clock	666.67MHz
CL-tRCD-tRP	10-10-10
tWTR-tRTP-tWR	5-5-10
tRRD-tFAW	4-20
Channels-Ranks-Banks	1-1-8

4.4.1 Methodology

Our simulation platform for the STVQ memory controller is based on DRAMSim2 [52], on top of which a heterogeneous system is simulated by a trace-based simulator, including a dual-core CPU, a graphic GPU, a video codec, and a display controller. Our DRAM timing model is configured according to the DDR3 SDRAM-1333 specification. The simulation parameters are listed in Tab. 4.2. Two test cases on the real-time traffic are evaluated. In particular, the simulated real-time cores include either the GPU and display control, or the video codec and display control.

For CPU test traces, we have four applications from SPEC 2000/2006 benchmarks with different memory intensities, including *art*, *mcf*, *hmmer* and *sphinx3*. GPU memory traces are extracted from the instrumented softpipe driver in the open-source OpenGL library Gallium3D [1, 3, 2]. Four representative mobile games are selected from different genres. The memory trace of video codec is generated from a commercial MP4 parser. More benchmark descriptions can be found in Tab. 4.3.

Table 4.3: Benchmark descriptions.

GPU benchmark	Description	FPS
coc	strategy game	50
fruit	action game	60
minecraft	sandbox game	60
transformer	shooting game	60
Video benchmark	Description	FPS
interstellar, knight	4k ultra HD movie trailer	24
CPU benchmark	Description	Bandwidth
art	image recognition	1.06 GB/s
mcf	combinatorial optimization	0.94 GB/s
hmmer	biosequence analysis	0.18 GB/s
sphinx3	speech to text program	0.005 GB/s

4.4.2 Bandwidth Allocation

To begin with, we test the STVQ memory controller with the CPU, the GPU and display control. For the dual-core CPU we use *mcf* and *art* benchmarks, and *coc* for the GPU. Bandwidth allocation for the CPU and the GPU during one frame period (20 ms) by the STVQ memory controller is shown in Fig. 4.6 (blue curve with circular markers). For comparison two other schemes are implemented: one only has CPU cores generating memory requests (green curves with diamond markers); the other applies SJF policy (in favor of CPU) in a two-tier memory controller (red curves with cross markers). The applied SJF policy limits batch size to 4 entries and sets the threshold age to 200 cycles. A batch is formed under three circumstances: 1) an incoming transaction accesses a different row; 2) the oldest transaction in the batch has exceeded the threshold age; 3) the batch size limit has been reached. Among available batches, SJF scheduler chooses the smallest and the oldest one.

With no concerns for command queuing delay, the two-tier memory controller with an SJF scheduler allows the GPU to quickly finish all the transactions in current frame time and consume most of the bandwidth in the first 3ms. At the same time applications on the CPU hardly make any progress, suffering from a delay lasting almost 3ms which can

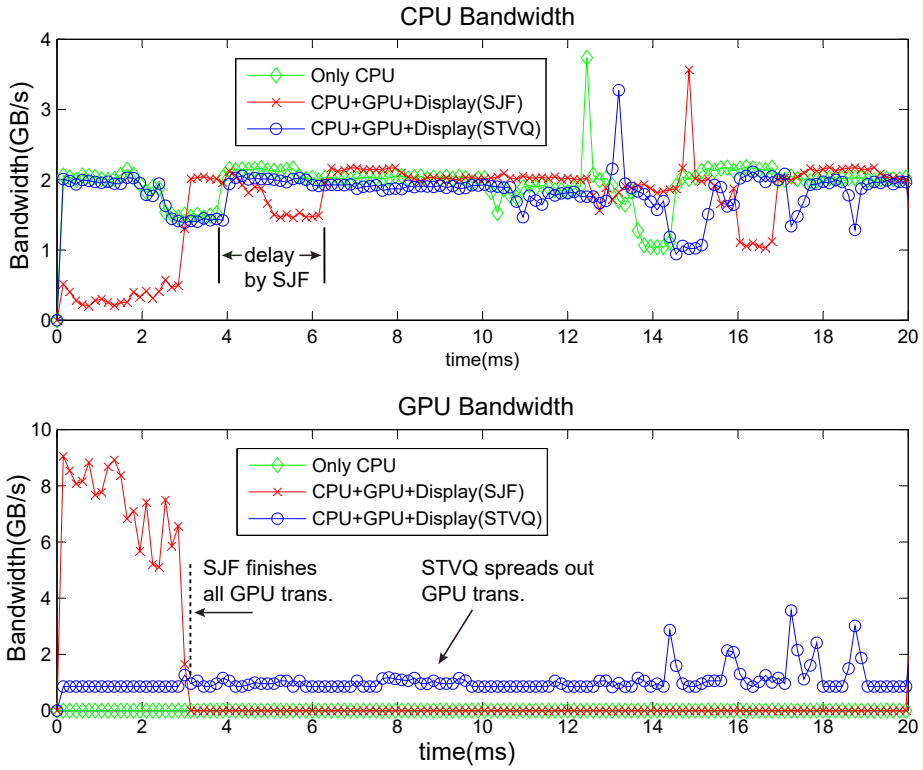


Figure 4.6: Bandwidth allocation for the CPU and the GPU.

be clearly observed between the red and green curves in the CPU bandwidth diagram. By comparison, the STVQ memory controller keeps GPU bandwidth low and steady most of the time to alleviate memory contention against the CPU. In the GPU bandwidth diagram, spikes appear near the end of the frame period when the actual frame progress of GPU workload is behind the expected progress and thus the GPU receives higher priority in order to finish current frame in time. The CPU bandwidth diagram shows that the results of the CPU-only system and heterogeneous system with the STVQ controller stay close to each other, because the STVQ memory controller protects CPU traffic from the excessive interference from the GPU. For clarity, the bandwidth of the display control is not shown in Fig. 4.6 in that it is very similar to the GPU's.

Furthermore, we test the STVQ memory controller with the CPU, the video codec

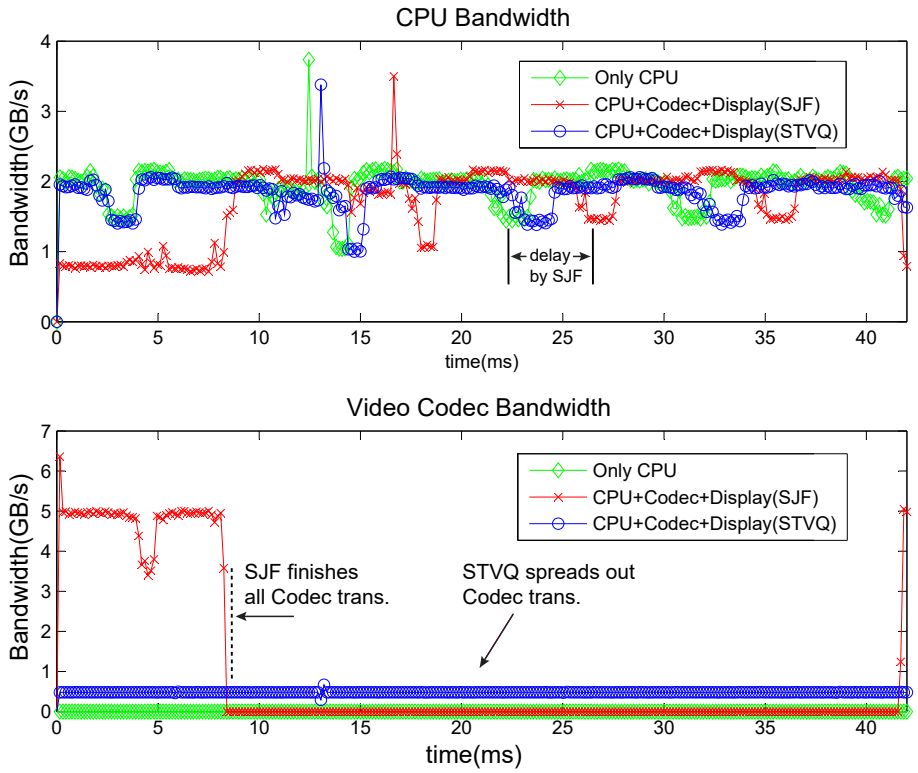


Figure 4.7: Bandwidth allocation for the CPU and video codec.

and the display control. Benchmarks *mcf* and *art* are applied on CPU cores again, while the 4k HD movie clip *knight* is used to generate memory traces for the video codec. The bandwidth allocation by the STVQ controller during one frame period (42 ms) is shown in Fig. 4.7. Similar to the previous example, the two-tier scheduler using SJF policy gives more bandwidth to real-time cores in the beginning. As 4k movie has more pixels in each frame than GPU applications (1280×720 pixels), it takes DRAM more time to finish serving the video codec in Fig. 4.7 compared with Fig. 4.6. The delay caused to CPU applications by the two-tier controller can be observed by comparing CPU bandwidth curves of the SJF and the CPU-only scenario. The STVQ controller ends up with a much smaller delay for CPU traffic by evenly distributing bandwidth consumption of the video codec over the whole frame period.

4.4.3 CPU Performance

Execution Time

We first compare the STVQ controller with the SJF scheduler in reducing the latency for CPU traffic. Fig. 4.8 shows the normalized execution time by the STVQ controller to finish 7.5 million instructions by each CPU application. The normalization is performed on basis of the execution time of the SJF scheduler to finish the same number of instructions. As shown, the normalized execution times all remain below 1 which means the STVQ controller takes less time to execute the same number of CPU instructions compared with the SJF scheduler. On average the STVQ controller saves 6% execution time.

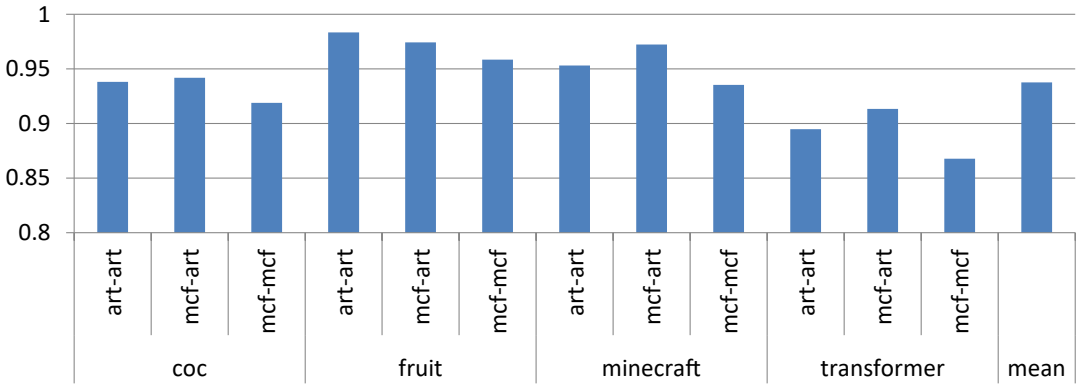
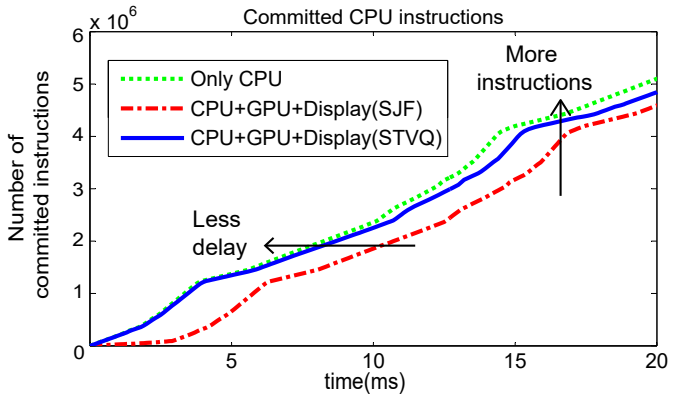


Figure 4.8: Summary of normalized execution times by the STVQ controller.

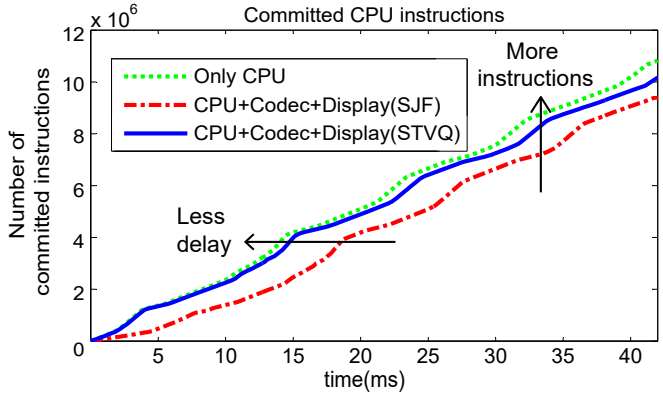
IPC Slowdown

Due to memory interference, the CPU suffers from IPC slowdown when sharing memory with real-time cores, which means fewer CPU instructions are executed during the same period of time. The reduction of IPC slowdown rate is a good indication on the efficacy of QoS-aware memory controllers, as a lower slowdown rate indicates less interference imposed on the CPU. Therefore, we evaluate the STVQ controller in terms of IPC slowdown reduction as the next step. The same comparison schemes are evaluated,

including the CPU-only system and the heterogeneous system with a two-tier scheduler applying SJF policy.



(a) under interference from the GPU and display control



(b) under interference from video codec and display control

Figure 4.9: Committed CPU instructions during one frame period under memory interference from real-time cores.

Fig. 4.9 shows the number of committed CPU instructions during one frame period. Two combinations of real-time cores are simulated: Fig. 4.9(a) shows the CPU under interference from the GPU and display control memory traffic; Fig. 4.9(b) shows the CPU affected by video decoder and display control. Same to the last section, benchmarks *mcf-art*, *coc* and *knight* are applied respectively to the CPU, the GPU and video codec. As expected, the CPU-only system executes the most instructions over the same period compared with other scenarios. Meanwhile the STVQ memory controller commits more

instructions than the two-tier memory controller. Note that the horizontal distance between two curves shows the time difference after finishing the same number of instructions, which indicates the delay of CPU traffic caused by memory interference. As can be seen, the STVQ memory controller results in less delay than the two-tier scheduler. In both cases, the two-tier scheduler leads to huge delays to the CPU in the beginning. In contrast, the STVQ controller has the CPU traffic delay increase gradually over time by alleviating more memory conflicts.

To further evaluate IPC slowdown, we quantify the slowdown rate as the ratio of the baseline IPC, when CPU has exclusive access to memory, to the IPC when CPU shares memory with real-time cores. Due to the memory interference, the slowdown rate is inevitably bigger than 1, and a lower value indicates less interference. Apart from the proposed STVQ memory controller, we test three other scheduling policies applied at the two-tier memory scheduler, including RR policy, SJF policy and the QoS policy proposed in [22] and extended to support multiple real-time cores which are served in round-robin manner when the CPU has low priority.

The summary of slowdown rates of all tested scenarios during 50ms is shown in Fig. 4.10. As expected, RR policy results in the worst CPU performance because it favors bandwidth-intense traffic and real-time traffic consumes far higher bandwidth than that of the CPU transactions. Furthermore, SJF and QoS share similar results, because in both cases CPU latency reduction by the transaction stage in the two-tier memory controller is cancelled by the extra queuing delay in the command stage. Finally, the STVQ memory controller achieves the best performance in all tested scenarios except for the case with *art-art* and *knight*. In that case the slowdown rate by STVQ is a bit higher than SJF and QoS. This happens when adverse address mapping leads to plenty of bank conflicts between real-time and CPU traffic. Instead of letting bank conflicts happen all at once, the STVQ memory controller spreads out the conflicts over time, which in total may cause

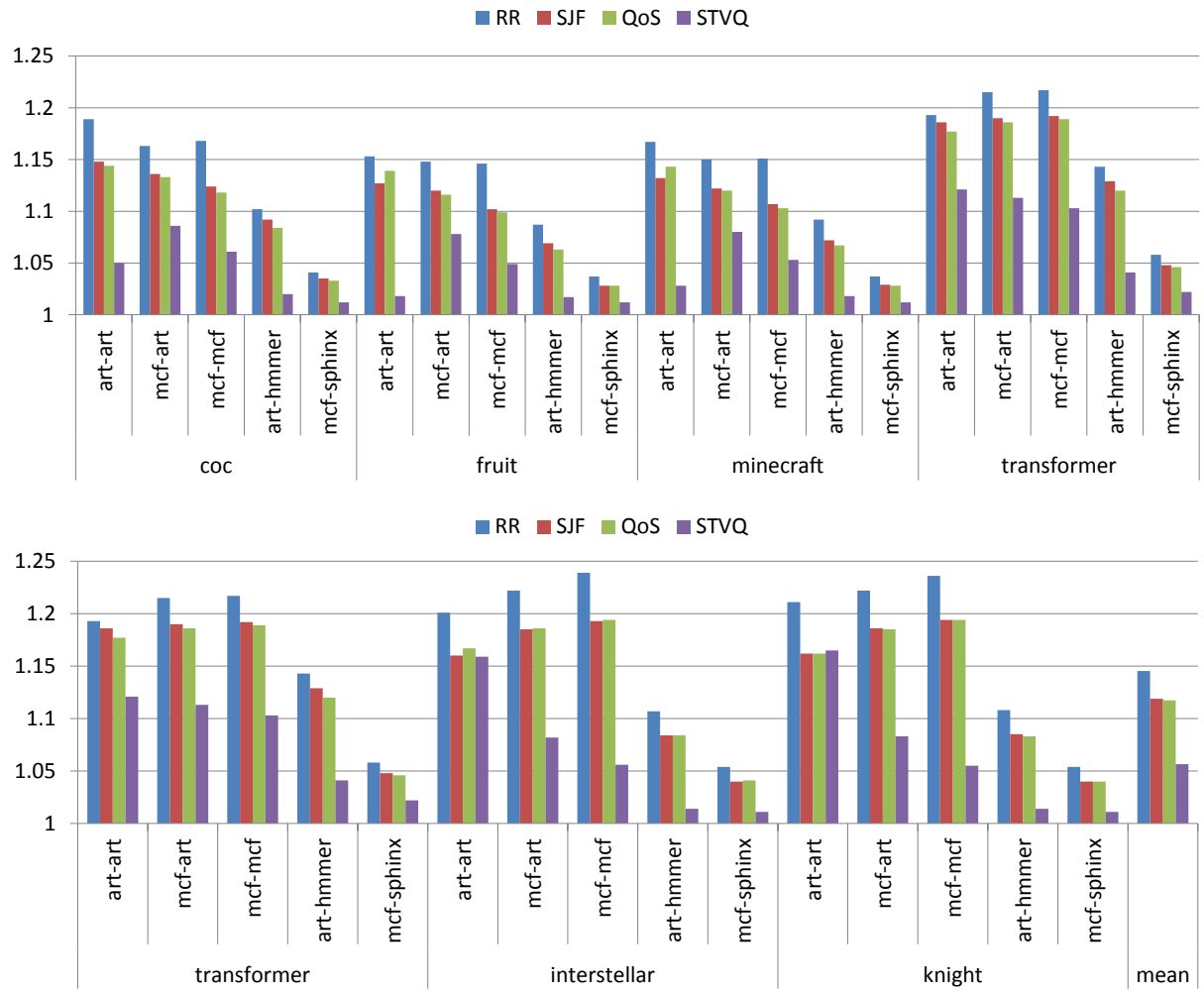


Figure 4.10: IPC slowdown rate summary.

more latency to CPU traffic than the reduction on queuing delay. However, adverse address mapping takes place rarely in our experiment even with random address allocation. With careful address mapping between physical and DRAM addresses, severe bank conflicts can be avoided. Compared with the best performances by prior schedulers, the STVQ memory controller lowers the slowdown rate by 6.1% on average. The highest reduction of slowdown rate by STVQ is 13.9% (*mcf-mcf* and *knight*).

Frame time slacks ($:= \frac{1}{FPS_{target}} - \frac{1}{FPS_{actual}}$) by the STVQ controller are summarized in Tab. 4.4. All slacks are non-negative, which means real-time cores meet their target

Table 4.4: Frame time slack summary of the STVQ memory controller.

Real-time benchmark	CPU benchmark	Frame time slack (μs)	Real-time benchmark	CPU benchmark	Frame time slack (μs)
coc	art-art	0.0	transformer	art-art	0.3
	mcf-art	0.2		mcf-art	0.1
	mcf-mcf	0.2		mcf-mcf	0.1
	art-hmmer	0.0		art-hmmer	0.0
	mcf-sphinx	0.2		mcf-sphinx	0.1
fruit	art-art	0.6	interstellar	art-art	18.1
	mcf-art	0.3		mcf-art	12.0
	mcf-mcf	0.2		mcf-mcf	18.1
	art-hmmer	0.4		art-hmmer	5.9
	mcf-sphinx	0.3		mcf-sphinx	5.9
minecraft	art-art	0.1	knight	art-art	6.6
	mcf-art	0.6		mcf-art	13.4
	mcf-mcf	0.4		mcf-mcf	26.8
	art-hmmer	0.0		art-hmmer	6.5
	mcf-sphinx	0.2		mcf-sphinx	6.5

frame rates in all scenarios. In addition, if the frame period is allowed to be late for a few microseconds, which is unnoticeable for human eyes, even lower IPC slowdown rates can be achieved by the STVQ controller.

Fig. 4.11 shows IPC slowdown rate difference between STVQ and SJF as DRAM frequency increases from 666.67MHz to 1GHz. The difference is calculated by subtracting the IPC slowdown rate of STVQ from that of SJF. A positive value in difference means SJF has higher slowdown rate. Therefore, as shown in Fig. 4.11, SJF has higher slowdown rates in all scenarios. However, the difference in slowdown rates shrink as DRAM frequency increases, because under higher frequency DRAM processes requests faster, leading to less waiting time in the command stage where most CPU latency in the two-tier memory controller is generated.

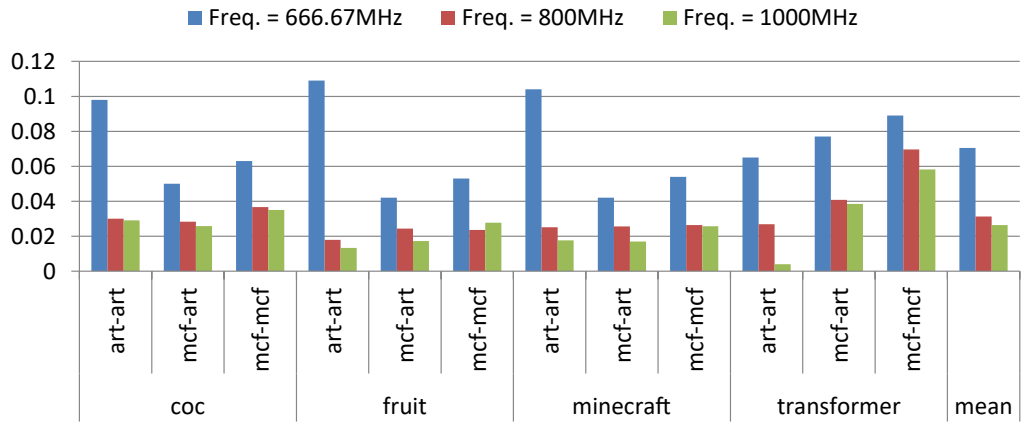


Figure 4.11: Differences of IPC slowdown rates between STVQ and SJF as DRAM frequency increases.

4.4.4 Leakage and Area Overhead

For overhead analysis, the STVQ memory controller is implemented in RTL including the scheduling logic and buffers. For comparison, we have also implemented with the two-tier staged memory scheduler [4] which has the lowest overhead among previous schedulers. Same input buffer size is applied to both memory controllers. Synthesis is performed at 666.67MHz with TSMC GP 65nm technology node library. In both designs, more than 90% of leakage power and area overhead are induced by buffers. Tab. 4.5 presents the power and area overhead results which are normalized to the overhead by the staged memory scheduler. As shown, the STVQ controller saves 8% leakage and 28% area overhead. Overhead savings by the STVQ controller mainly come from the removal of command queues, whereas adding virtual queues does not increase buffer cost since per-bank queues share the same physical memory. In summary, the STVQ memory controller achieves more efficient scheduling with less overhead.

Table 4.5: Power and area overhead compared with the staged memory scheduler.

Memory controller	SMS	STVQ
Leakage(normalized)	1	0.92
Area(normalized)	1	0.72

4.5 Chapter Summary

In this chapter, we show that the existing approaches for heterogeneous MPSoCs memory scheduling based on two-tier queuing architectures have the disadvantage of large queuing delays in the FIFO command stage, which may counteract the benefits of QoS-aware scheduling at the transaction stage. To enable efficacious scheduling, we propose a novel memory architecture with a single-tier virtual queuing system. Separable allocation is introduced to our memory controller to implement highly scalable scheduling logic for the single-tier of queuing stage. According to the experimental results, the STVQ memory controller achieves a lower IPC slowdown rate compared with previous memory schedulers for heterogeneous systems. Among the evaluated benchmarks, IPC slowdown is reduced by up 13.9%, with an average reduction of 6.1%. This reduction is achieved with no frame drops or negative frame slacks for real-time cores.

4.6 Acknowledgement

This chapter, in part, is a reprint of the material as it appears in ACM Transactions on Design Automation of Electronics Systems, 2017. Song, Yang; Samadi, Kambiz; Lin, Bill. This chapter also includes the material as it appears in proceedings of ACM/IEEE Design Automation Conference, 2016. Song, Yang; Samadi, Kambiz; Lin, Bill. The dissertation author was the primary investigator and author of these papers.

Chapter 5

Orchestrated Last-Level Cache and DRAM Scheduling

5.1 Introduction

Cache was originally designed for the CPU to harness its memory locality and reduce memory stalls caused by DRAM access. Compared to CPU traffic, the memory traffic by real-time cores generally shows much lower locality. However, regardless of the discrepancy in memory locality, the last-level cache is still often shared within a heterogeneous MPSoC when real-time cores and the CPU access the same address space. This is because, in order to guarantee memory coherence, memory requests to the same address have to traverse through the same data path in the memory subsystem. As a result, the last-level cache becomes an important shared resource in heterogeneous MPSoCs.

Same as the memory controller, memory interference is a concern for the last-level cache when it is shared by multiple agents. QoS-aware management of the last-level cache has been extensively covered by previous work [30, 33, 53] to deal with such interference. What's more, since the last-level cache is merely one block away from DRAM, it also has

significant influence on the memory efficiency in DRAM. Specifically, the memory traffic flowing out of the last-level cache consists of cache writebacks and cache misses, forming into write and read traffic respectively to DRAM. Prior work have discussed memory-aware cache replacement and eviction policies [48, 26, 21] that reorder cache writebacks to improve memory efficiency. However, last-level cache misses, i.e. read requests to DRAM, have long been ignored when it comes to their potential for memory efficiency improvement.

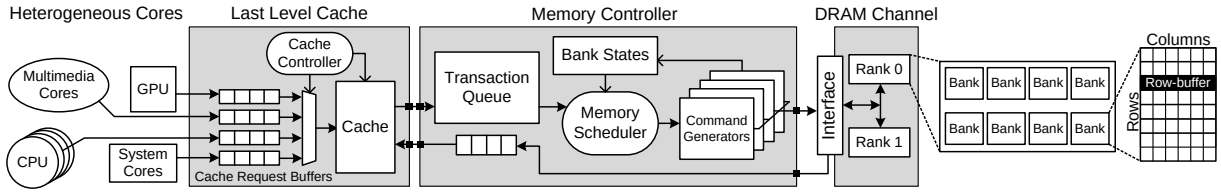


Figure 5.1: Traditional architecture of the memory subsystem that performs uncoordinated scheduling at the last-level cache and the memory controller. Private caches and on-chip interconnects are omitted.

As cache missing traffic travels through the last-level cache and the memory controller, these two stages are usually seen as separate units that make independent scheduling decisions (as shown in Fig. 5.1). Specifically, incoming cache requests are invisible to the memory scheduler. On the other hand, the information on DRAM bank states is unavailable to the cache controller. The limited memory visibility in both ways leads to uncoordinated scheduling decisions. These decisions may eventually suppress memory efficiency because neither controller is aware of the opportunities for memory efficiency improvement on the other side. In addition, the increased cache miss rate by heterogeneous traffic brings more read requests to DRAM from the last-level cache, making the problem of uncoordinated scheduling even worse.

In this chapter, we propose a unified memory controller for the last-level cache and DRAM. The goal of the unified memory controller is to provide more visibility to memory scheduling and orchestrate scheduling decisions by the cache controller and the memory

scheduler. The contributions of our work are summarized as follows.

- We identify the limitations of separate scheduling stages for the last-level cache and DRAM. With limited visibility on both sides, the last-level cache controller and the memory controller are not able to make the best decisions to improve memory efficiency.

- We propose the design of a unified memory controller with orchestrated schedulers for the last-level cache and DRAM. Given access to cache request buffers, the memory scheduler gains more visibility and harvests more fleeting row-buffer hit opportunities, without raising hardware complexity.

- We present a dynamic policy for the orchestrated scheduling in the unified controller. The dynamic policy balances the priorities of CPU requests and row-buffer hits in runtime to achieve CPU IPC targets while improving memory efficiency.

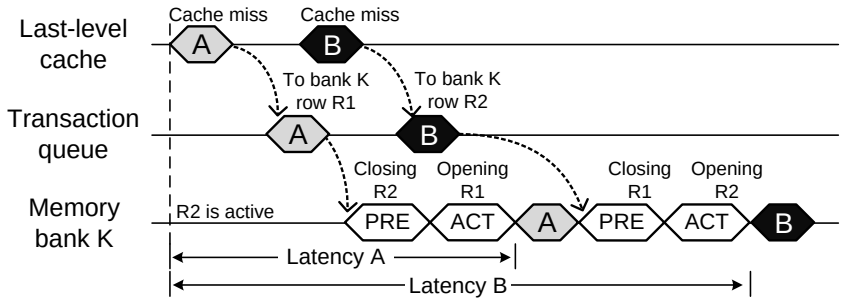
- We evaluate the proposed design using memory traffic of a next-generation heterogeneous multicore SoC and show that row-buffer hits and DRAM bandwidth are improved by 22% and 16.8% respectively on average by the unified controller. DRAM energy can be saved by up to 29.7% with comparable CPU IPCs achieved. We also test different scheduling policies to explore the balance between CPU IPC and DRAM bandwidth. We demonstrate that the presented dynamic policy helps the unified controller to achieve target IPC and improve bandwidth at the same time.

The rest of this chapter is organized as follows: Section 5.2 describes the background and motivation of our work. Section 5.3 presents the architecture of the proposed unified memory controller. Section 5.4 introduces a dynamic scheduling policy for the unified controller. Section 5.5 elaborates evaluation results of the unified controller in comparison with the conventional design. A thorough review of related work and conclusions follow in Sections 5.6 and 5.7.

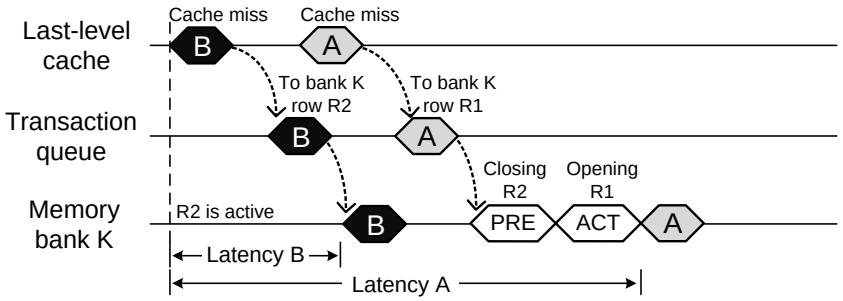
5.2 Background and Motivation

Architectures of the last-level cache and the memory controller have been widely discussed in recent years. Most prior work tackle these two units as independent memory fabric [42, 25, 4, 30, 53, 33]. In contrast, fewer work have explored the interaction between the last-level cache and the memory controller. Stuecheli et al. [48] were the first to coordinate memory scheduling with cache writeback traffic. Their *virtual write queue* is implemented by LRU cache lines in the last-level cache, as an extension to the transaction queue in the memory controller. The virtual write queue improves the scheduler's visibility into memory locality and thus leads to better memory scheduling efficiency. Kim et al. [26] proposed a new scheduling scheme for the last-level cache in consideration of DRAM rank power states. The objective is to reduce DRAM power state transitions induced by evicted cache lines. Jeon et al. [21] extended the virtual write queue to include read requests when clustering memory requests and cache writebacks for open row-buffers. A comprehensive exploration of DRAM-aware eviction policies was also provided by the authors.

Aforementioned work focus on the DRAM-aware management of last-level cache writebacks. However, cache misses can also affect the efficiency of memory scheduling. Fig. 5.2 illustrates an example in which cache access ordering determines the number of row activations and precharges in DRAM. Assume cache requests A and B are both waiting at the last-level cache in the beginning. The target addresses of requests A and B are located in the same memory bank (bank K) but different rows ($R1$ and $R2$ respectively). The target row by request B, row $R2$, is initially active. In the case of cache misses for both requests, if request A is served first by the last-level cache (in Fig. 5.2(a)), it will arrive at the transaction queue earlier than request B. Without knowing the existence of request B, the memory scheduler precharges (i.e. closes) row $R2$ and activates (i.e. opens) row $R1$ for request A. When request B arrives later, the scheduler will have to close row $R1$ and re-open $R2$. In the other scenario (in Fig. 5.2(b)), request B accesses the last-level cache



(a) Request A is served first.



(b) Request B is served first.

Figure 5.2: An example of cache access ordering affecting memory efficiency. Destination of request A: bank K row $R1$. Destination of request B: bank K row $R2$. Initial active row in bank K: row $R2$.

first and gets to be scheduled to row $R2$ before this row is closed for request A, avoiding extra precharge and activation operations. By comparison, memory scheduling is less efficient in the first scenario because the unnecessary precharge and activation operations result in higher DRAM power and average memory latency. Yet, this can be prevented if row-buffer states are visible to the last-level cache or incoming cache requests are visible to the memory scheduler so that orchestrated scheduling decisions can be made to ensure request B arrives before row $R2$ is closed.

What's worse, the problem depicted in Fig. 5.2 is fairly common in heterogeneous systems in comparison with multicore processors, due to their disparate memory traffic patterns. First, DRAM bandwidth sees a much higher demand in a heterogeneous system, because most data sharing happens in the main memory. Second, cache miss rate is much

higher in a heterogeneous system, because the traffic from multimedia real-time cores shows low temporal locality. At last, memory traffic shows much higher row-buffer locality in a heterogeneous system, because multimedia data, say pixels in a single frame, tends to be stored in consecutive addresses. As a result, compared with the memory traffic in a multicore processor, the traffic in a heterogeneous system consists of many more cache misses with much higher row-buffer locality. Although last-level cache bypassing [33, 7, 24, 32] by real-time traffic has been proposed to avoid polluting the last-level cache with non-reusable data and thus to reduce cache misses, bypassing cannot fundamentally avoid this issue.

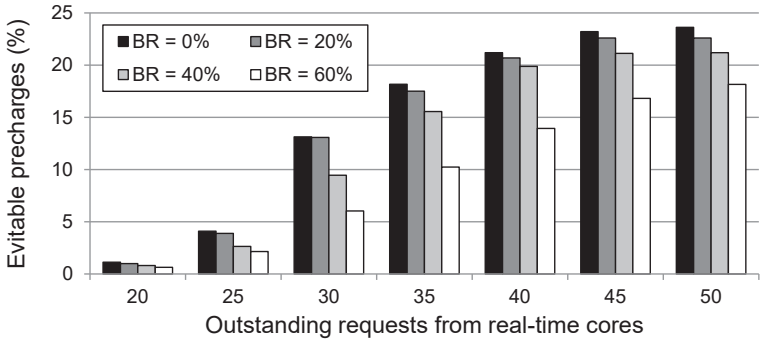


Figure 5.3: Percentage of evitable precharges vs. the number of outstanding real-time requests and cache bypassing ratio (BR) for real-time traffic. Scheduling decisions for the last-level cache and DRAM are independent from each other.

We identify a precharge operation as *evitable* when there are last-level cache requests which target at the row to be precharged and will turn out to be cache misses later on. For example, the precharge operation for $R2$ in Fig. 5.2(a) is evitable because a potential row-buffer hit by request B is waiting at the last-level cache.

The number of evitable precharges indicates the potential for memory efficiency improvement if more visibility is given to the scheduler. Fig. 5.3[1] shows the percentage of precharge operations that are evitable as the number of outstanding real-time requests and

[1]CPU test case 4 and real-time application GPU_shader are used for simulation (Tab. 5.3).

real-time cache bypassing ratio[2] vary. As can be seen, more evitable precharges happen when more outstanding real-time requests are injected. Without cache bypassing (BR = 0%), the percentage of evitable precharges saturates around 23% after the number of outstanding real-time requests reaches 50, which is not too much considering that there are plenty of heterogeneous cores, including memory-intense ones such as the GPU. Even when 60% of real-time traffic bypasses the last-level cache, the percentage of evitable precharges can still reach up to 18% given enough outstanding requests.

5.3 Unified Memory Controller with Orchestrated Schedulers

In this section, we present the unified memory controller with improved memory visibility and orchestrated scheduling for the last-level cache and DRAM. Architecture of the unified controller is shown in Fig. 6.4. Similar to the conventional design (Fig. 5.1), most memory traffic, except cache bypassing traffic, still travels through cache request buffers, the last-level cache, the transaction queue and DRAM. Yet the memory scheduler now has access to cache request buffers so that row-buffer hit opportunities can be detected and harvested as early as the cache requests arrive in the memory subsystem. A short-cut for cache requests, i.e. the *fast lane*, is introduced to expedite the delivery of potential row-buffer hits. More details follow in the rest of this section.

5.3.1 Memory Scheduling and Row-buffer Hit Harvesting

In the unified memory controller, the memory scheduler works in two modes: memory scheduling and row-buffer hit harvesting.

[2]Cache bypassing ratio is computed as the number of cache bypassing requests divided by the total number of memory requests.

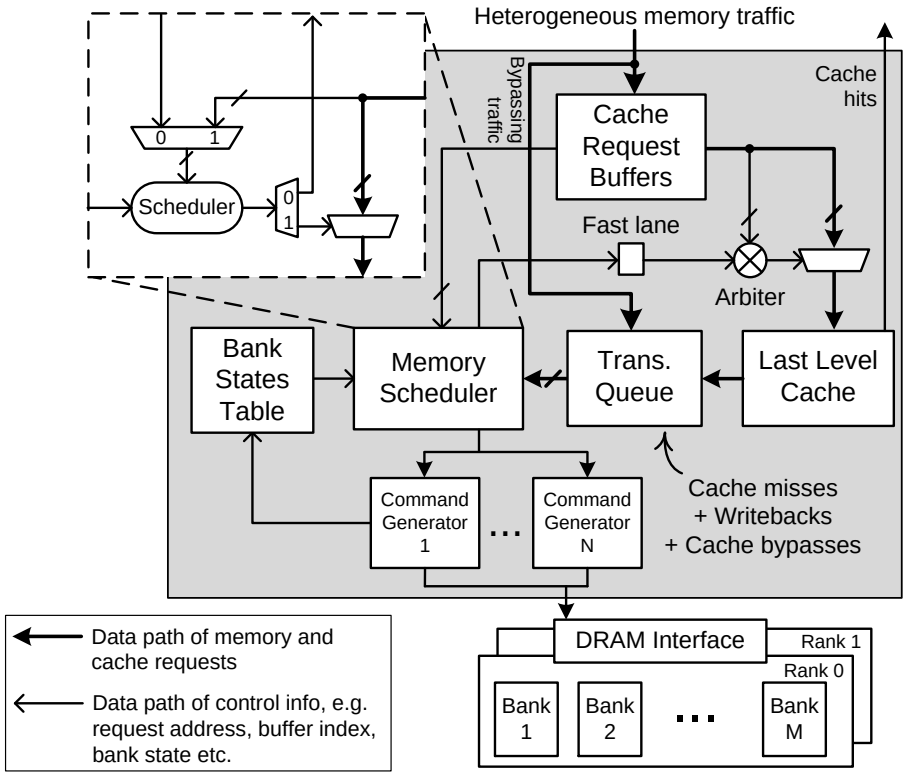


Figure 5.4: Architecture of the unified controller for the last-level cache and DRAM.

In the scheduling mode, the scheduler performs the conventional memory scheduling task. Specifically, the scheduler arbitrates among requests in the transaction queue including cache misses and cache writebacks from the last-level cache, and cache bypassing memory requests. This process requires information on the states of DRAM banks and row-buffers, which is maintained on the bank states table. To avoid unnecessary precharges and row activations, preferentially a request for an idle bank and an active row is chosen. The chosen request will be forwarded to one of the per-bank command generators. The command generator will convert the selected request into one or multiple commands to operate the DRAM bank, according to the low-level DRAM timing constraints. Note that even though the memory scheduler is capable of performing scheduling at every cycle, it takes time for DRAM to execute each operation command. Besides, there are constraints on how often one can activate row-buffers in order to limit dynamic power [16]. Therefore, most of

the time the scheduler is waiting to issue new operations. We utilize these idle cycles by introducing the row-buffer hit harvesting mode to the scheduler.

In the harvesting mode, the scheduler scans through extant read requests in cache request buffers. If the target address of a read request is located in an open row or a row to be opened within t_{miss} cycles, the scheduler will forward this request to the last-level cache through the fast lane. Here, t_{miss} is the cache miss latency of the last-level cache. If more than one such requests are found, the oldest one will be selected. If no such requests can be found or the fast lane is busy, no cache requests will be harvested by the scheduler. To minimize transmission latency, only one buffer is used on the fast lane so that a cache request becomes visible to the last-level cache as soon as it is harvested by the scheduler. If the harvested request turns out to be a cache miss, it will arrive at the transaction queue t_{miss} cycles after being harvested, and then become available for memory scheduling. Note that, during row-buffer hit harvesting, the harvested request still resides in the cache request buffer. It is actually the buffer index that is forwarded to the fast lane. Later on, the last-level cache will use this buffer index to locate the harvested request among cache request buffers.

The purpose of having the harvesting mode is to capture cache requests that potentially will result in row-buffer hits and reduce their cache access latency so that they can finish cache lookup before the open rows are closed. Moreover, to avoid limiting memory throughput we do not hold the row-buffer open for the cache request on the fast lane in case it will end up as a cache hit.

5.3.2 Orchestrated Cache Request Scheduling

As mentioned, when the buffer index of the harvested request appears on the fast lane, the last-level cache uses this index to fetch the harvested request. Since row-buffer hit harvesting is not being performed all the time, the scheduler cannot deliver a cache

request whenever the last-level cache is ready. Therefore, we still need a local arbiter at the last-level cache to arbitrate among requests from cache request buffers in case the fast lane is empty.

At the last-level cache, orchestration with the memory scheduler is done by scheduling the harvested cache request for cache access. If memory efficiency is the first concern, harvested requests should be given higher priority over other cache requests. An example of such scheduling policies is to always look up the cache request from the fast lane first, otherwise find one from cache request buffers based on a local scheduling policy.

To implement orchestrated cache request scheduling, the cache arbiter first reads the buffer index on the fast lane; if the buffer is empty, the local arbitration logic will be triggered to determine the next cache request to access the last-level cache. In either case, the output of the arbiter is the buffer index of the chosen request. In the last step, the request associated with the buffer index will be fetched and sent to the last-level cache for access.

5.3.3 Hardware Implementation and Cost

In the implementation, memory scheduling and row-buffer hit harvesting are implemented on the same scheduler (shown in Fig. 6.4) in order to re-utilize the arbitration logic. The logic contains a series of magnitude comparators, with multiple inputs and a single output. Inputs to the arbitration logic includes not only scheduling information, such as target addresses and request types etc., but also buffer indices of input requests. After arbitration, the buffer index of the selected request is produced at the output. Apart from slightly different policies, the major difference between the arbitrations for scheduling and harvesting are the input sources and the output destinations. For row-buffer hit harvesting, the scheduler collects information on cache requests as the input, and outputs the buffer index of the harvested cache request to the fast lane. For memory scheduling, the scheduler

receives memory requests from the transaction queue at input, and uses the resulting buffer index to select a request for the command generator.

Multiplexers are needed to adjust input and output data paths to the scheduler depending on the working mode. In Fig. 6.4, indices 1 and 0 at multiplexers represent the scheduling and harvesting modes respectively. To switch between two working modes, the scheduler only needs to adapt the selection signals at the multiplexers for outputs and inputs.

In addition, the fast lane contains a small buffer to record the buffer index of the harvested cache request. The bit-width of a buffer index is determined by the number of cache request buffers. If there are 40 cache request buffers, we need a 6-bit buffer on the fast lane. Overall, the implementation of the orchestrated schedulers do not require sophisticated logic with significant cost.

5.4 Dynamic Orchestrated Scheduling

So far we have discussed improving row-buffer hits in the unified memory controller by harvesting row-buffer hit opportunities in cache request buffers. However, memory efficiency is not the only concern in system design. The health of heterogeneous cores, especially the CPU, is also a critical metric for system performance. When row-buffer hit harvesting is performed, traffic flows with high cache miss rates and high row-buffer locality receive the most improvements on row-buffer hits. Such traffic flows are usually generated from multimedia cores in the heterogeneous system. By comparison, CPU traffic has much lower cache miss rate, and the scarce cache misses from the CPU typically show much lower memory locality. As a result, CPU traffic may suffer from deprivation during row-buffer hit harvesting, unless proper care is given to CPU traffic by the scheduler.

To treat CPU traffic with different priorities in relative to memory efficiency, different

scheduling policies can be applied at the cache arbiter and the memory scheduler. Tab. 5.1 lists three possible scheduling policies and their priority hierarchies. The arbitration process at an arbiter or a scheduler includes a series of comparisons. Requesters are evaluated and compared in different aspects, based on the priority levels of these aspects from high to low. The comparison process continues until one requester, if any, stands out from the others. For example, in Policy A, the arbiter first checks whether any of the given requesters is going to an open row. If only one such requester is found, it will be chosen as the arbitration winner. Otherwise, if multiple requesters are found, the arbiter will continue with them and check if any of them is a read request. If there are still multiple requesters left, the arbiter will select the one with the earliest arrival time.

Table 5.1: Policies for memory and cache request scheduling with different priority levels for the CPU.

Scheduling policy	Priority levels from high to low			
Policy A	Row-buffer hit	Read request	Older request	
Policy B	Row-buffer hit	CPU request	Read request	Older request
Policy C	CPU request	Row-buffer hit	Read request	Older request

Among the policies listed in Tab. 5.1, Policy A does not distinguish CPU requests from others and focuses on row-buffer hit improvement. It is expected to lead to the highest improvement in memory efficiency at the expense of the most CPU IPC degradation. Furthermore, Policy C serves CPU requests as the first priority, which is good for CPU performance but can also suppress memory efficiency when row-buffer hit harvesting is preempted by CPU traffic. In addition, Policy B gives CPU requests the second highest priority right after row-buffer hitting requests, as a trade-off between Policy A and C.

To orchestrate arbitrations at the cache arbiter and the memory scheduler, the same scheduling policy is applied to both with little difference in implementation: to identify a potential row-buffer hit, the memory scheduler matches open row addresses with the target addresses of existing requests, whereas the cache arbiter directly checks on the harvesting result on the fast lane. Since in Policy C a CPU request has a higher priority than the

request for an open row, the harvested request on the fast lane can be preempted by a cache request from the CPU. To avoid holding up the fast lane, the result on the fast lane will be discarded once the preemption happens. Note that it is the buffer index of the harvested request that is stored on the fast lane. Therefore, to discard the harvesting result, the cache arbiter only needs to erase the buffer index on the fast lane while the previously harvested request is still residing at a cache request buffer and available to be harvested again if the target row remains open.

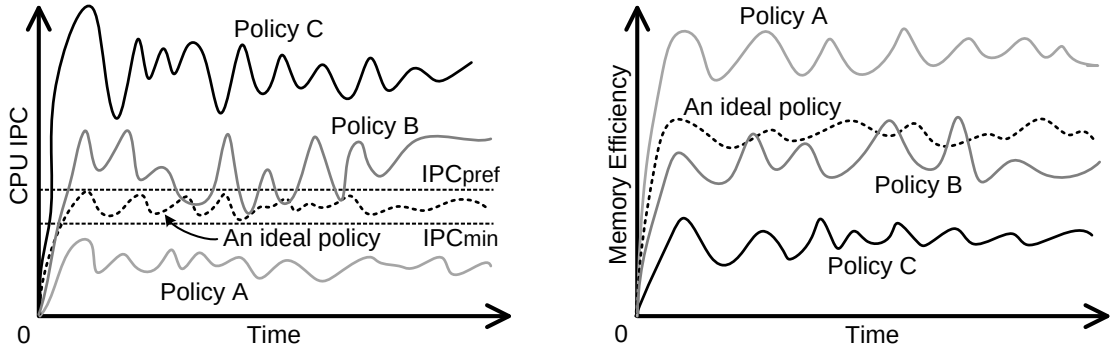


Figure 5.5: Real-time CPU IPCs and memory efficiency under different scheduling policies.

Aforementioned policies assign static priority levels to CPU traffic during orchestrated scheduling so that the impact of row-buffer hit harvesting on the CPU can be managed to different extents. However, static scheduling policies may not lead to the optimal results without taking into account of the target performance by the CPU. Fig. 5.5 illustrates CPU IPCs and memory efficiency in real-time under different scheduling policies. Memory efficiency can be interpreted in terms of row-buffer hit rate or DRAM bandwidth. The IPC is targeted to settle between two IPC references, IPC_{min} and IPC_{pref}. These IPC references respectively represent the minimal IPC that is acceptable and the IPC that is preferred by the application. As have been discussed, Policy A leads to high memory efficiency but low IPC while Policy C leads to high IPC but low efficiency. In this example, insufficient IPC is achieved by Policy A. Although the IPC by Policy C is well above the

target range, the excessive IPC improvement limits memory efficiency. In addition, Policy B achieves a more balanced result than Policy A and C, but there is no guarantee for CPU IPC to settle within the target range under Policy B. Ideally, we need a policy that dynamically adjusts scheduling priorities to help the CPU to meet the target IPCs without unnecessarily containing memory efficiency. Such a dynamic policy should be able to adapt in real-time, because the interaction between CPU traffic and memory efficiency varies over time and differs per application.

Prior work on memory resource management [44, 15] have proposed frameworks that dynamically adjust memory management policies to help the CPU to meet target performance. For example, in [44] CPU application controllers are used to monitor applications' health, indicated by their average IPCs. According to the average IPCs, the framework adapts management policies and memory configurations to achieve the target IPCs which are given to the controllers as references. The proposed unified controller can be integrated into such frameworks where the policy for the orchestrated scheduling is adapted according to the average IPC. This way, row-buffer harvesting is performed without impeding CPU's workload.

To demonstrate the capability of our unified controller in improving memory efficiency while allowing CPU IPC to achieve the targets, we introduce a dynamic orchestrated scheduling scheme in the unified controller as shown in Algorithm 3. The CPU evaluates the average IPC periodically and compares with the reference IPCs. If the average IPC is above the preferred IPC, Policy A is adopted to focus on row-buffer hit harvesting. If the average IPC is below the minimal IPC, Policy C is applied for the orchestrated scheduling to give CPU traffic the highest priority. If neither of above cases happens, Policy B is used to improve memory efficiency while giving CPU requests higher priority than real-time requests.

Note that, besides CPU IPC, the orchestrated scheduling can be adapted based

on any performance targets of any heterogeneous cores. However, providing a full-system performance monitor for heterogeneous cores is beyond the discussion in this work.

Algorithm 3: A dynamic policy for orchestrated scheduling.

Require: two reference CPU IPCs, i.e. IPC_{min} and IPC_{pref}

1: evaluate the average CPU IPC in the past epoch, IPC_{ave}
2: **if** $IPC_{ave} > IPC_{pref}$ **then**
3: adopt policy A
4: **else if** $IPC_{ave} > IPC_{min}$ **then**
5: adopt policy B
6: **else**
7: adopt policy C
8: **end if**

5.5 Evaluation

In this section, the proposed unified memory controller will be first tested in comparison with the conventional design with independent schedulers. Performance metrics of the memory subsystem, including row-buffer hits, DRAM bandwidth and energy will be evaluated. The unified controller will also be compared with memory-aware cache writeback schemes. Furthermore, the impact of row-buffer hit harvesting on CPU IPC will be explored. The dynamic scheduling policy introduced in Section 5.4 will be used to demonstrate the potential of the unified controller in improving memory efficiency while meeting IPC targets. Sensitivity analysis on several design parameters of the unified memory controller follows in the end.

5.5.1 Methodology

The unified memory controller is implemented on basis of DRAMSim2 [52] using LPDDR4 timing model [20]. The LPDDR4 model is chosen because it is widely paired with state-of-art heterogeneous MPSoCs [39, 35]. We perform cycle-accurate simulations of the

Table 5.2: Simulation parameters.

Number of Outstanding Requests			
CPU cores	16	Heterogeneous cores	50
Last-level Cache			
Size	2MB	Associativity	8
Miss latency	3 cycles	Hit latency	4 cycles
Request buffers	40	Frequency	2GHz
Memory Controller			
Read queue size	30	Write queue size	30
DRAM			
Volume	4GB	Max I/O bus freq.	1866MHz
CL-tRCD-tRP (cycles)	36-34-34	tWTR-tRTP-tWR (cycles)	19-14-34
tRRD-tFAW (cycles)	19-75	Channels-Ranks-Banks	1-2-8
Address mapping scheme		Channel-Rank-Row-Bank-Column	

Table 5.3: Benchmark descriptions.

CPU	Description
case 1	art–mcf–sjeng–sphinx3
case 2	art–mcf–sjeng–hmmer
case 3	mcf–sjeng–sphinx3–hmmer
case 4	art–mcf–sphinx3–hmmer
Real-time	Description
camcorder a	Camcorder application in UHD format at 60FPS with snapshot (one picture per second)
camcorder b	Camcorder application using dual cameras at 30FPS
display_panel	Dump display panel refreshing in WQXGA resolution at 60FPS (RGB888 pixel format)
GPU_shader	GPU shader composition of 4 ARGB layers

memory subsystem, including the last-level cache and DRAM. The simulation parameters are listed in Tab. 6.1. On top of the simulator, a heterogeneous system is emulated with four CPU cores and real-time cores such as the GPU and the video decoder etc. Each CPU core has one application from SPEC 2000/2006 benchmarks running and generating traffic for the last-level cache. For real-time cores, we collect memory traffic per data stream generated by all real-time cores in a next-generation heterogeneous multicore SoC [39] while running real life multimedia applications. To emulate cache bypassing for real-time traffic, we randomly pick real-time data streams whose requests will always bypass the

last-level cache. To achieve a specific bypassing ratio (BR), say 20%, we pick 20% of data streams from real-time cores and route their memory traffic directly to the transaction queue. Detailed descriptions of CPU test cases and real-time applications are listed in Tab. 5.3. For the evaluations in Section 5.2-5.4, Policy A in Tab. 5.1 is applied in both the unified controller and the conventional design. Note that since the cache arbiter in the conventional design is not aware of row-buffer states, it will ignore the row-buffer hit factors in Tab. 5.1 during arbitration.

5.5.2 Row-buffer Hits and DRAM Bandwidth Improvement

To begin with, we test the unified memory controller in terms of evitable precharges. Same as in Fig. 5.3, CPU test case 4 and real-time application *GPU_shader* are applied. As shown in Fig. 5.6, the unified controller lowers the percentage of evitable precharges to below 14% when there is no cache bypassing traffic. As more real-time traffic bypasses the last-level cache, there are fewer row-buffer hit opportunities in cache request buffers. Therefore, the percentage of evitable precharges lowers as the bypassing ratio increases. When the ratio is 60%, about 10% precharges are evitable. In comparison with the conventional approach with separate scheduling stages (Fig. 5.3), the unified controller reduces the percentage of evitable precharges by about 10% in all cases.

Fig. 5.7 shows the summary of row-buffer hit rates by cache requests that are harvested in the unified controller. This hit rate is calculated as the ratio of harvested requests hitting open row-buffers to all the harvested cache requests. Without cache bypassing, the row-buffer hit rate reaches 95% in almost all cases. Note that 100% hit rate is difficult to achieve because a harvested cache request may end up as a cache hit, or another request closes the target row-buffer while the harvested request is waiting to be scheduled. The latter cause becomes more common as cache bypassing traffic grows. As can be see, the hit rates decreases as the bypassing ratio increases. Yet we still can achieve

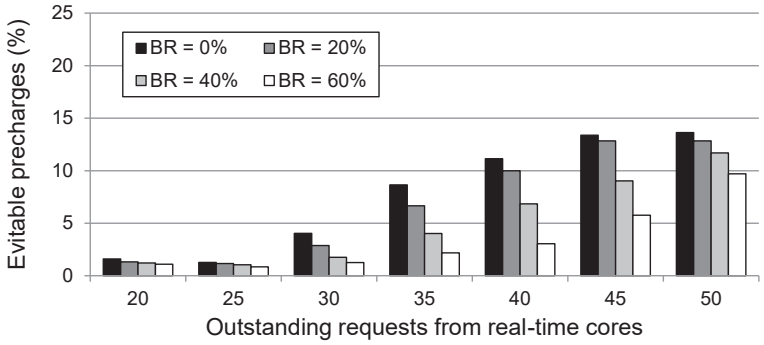


Figure 5.6: Percentage of evitable precharges vs. the number of outstanding real-time requests and cache bypassing ratio (BR) for real-time traffic. The unified memory controller is deployed.

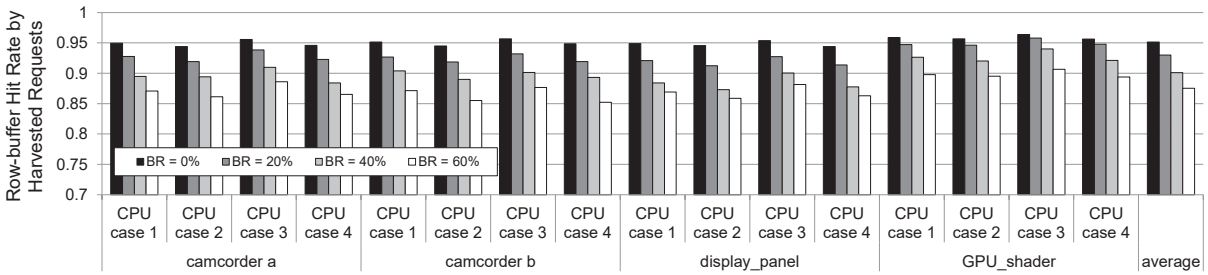


Figure 5.7: Summary of *row-buffer hit rates* by harvested cache requests by the proposed memory controller in one million cycles.

more than 85% hit rate in every case.

Fig. 6.9 shows the summary of normalized row-buffer hits per activation by the unified controller in comparison with the conventional design with separate scheduling units. In each test case, the memory subsystem is simulated for one million cycles. To perform normalization, the average number of row-buffer hits per activation by the unified controller is evaluated and divided by the result of the conventional controller. Without cache bypassing, the unified controller improves the number of row-buffer hits per activation by 22% on average. Since last-level cache bypassing reduces the opportunities of row-buffer hit harvesting in cache request buffers, the improvement of row-buffer hits by the unified controller decreases as bypassing ratio increases. Yet, the average number of row-buffer

hits per activation by the unified controller can still be 17.6% higher than the conventional design, when 60% real-time traffic bypasses the last-level cache. Similar results can be observed in DRAM bandwidth. Fig. 5.9 shows the summary of normalized DRAM bandwidth by the unified controller. Without cache bypassing, the average improvement of DRAM bandwidth is 16.8%. The improvement shrinks to 13.9% as bypassing ratio is raised to 60%.

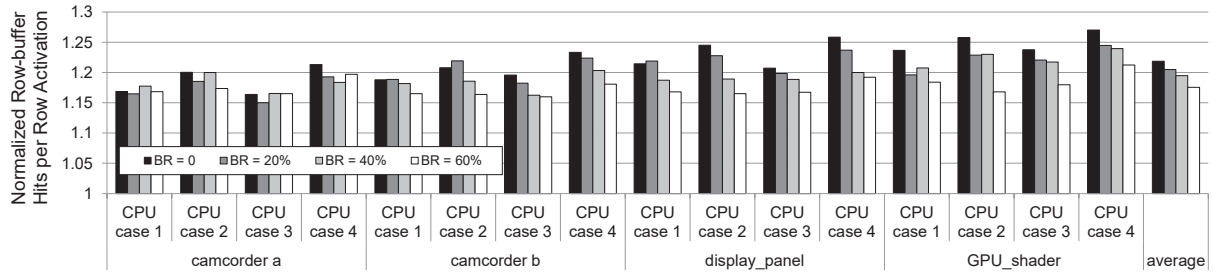


Figure 5.8: Summary of *normalized row-buffer hits per activation* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM.

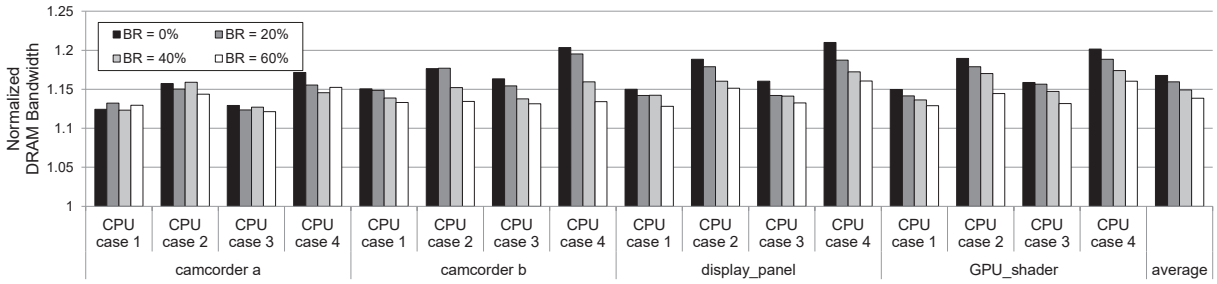


Figure 5.9: Summary of *normalized DRAM bandwidth* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM.

5.5.3 DRAM Energy Saving

Thanks to higher row-buffer hits and bandwidth, the unified controller helps DRAM finish processing the same number of memory requests with less time and power. We use

an open source tool named DRAMPower [6] for command trace based power and energy estimation. Tab. 5.4 lists the total DRAM energy savings for all test cases by the unified memory controller compared with the baseline controller with independent schedulers. Policy A (Tab. 5.1) is applied to both controllers. The proposed controller achieves lower DRAM energy in all cases. In the best case, the proposed controller saves DRAM energy by 29.7%.

Table 5.4: Summary of DRAM energy savings by the unified controller for processing 50000 memory requests.

CPU	Real-time	Energy saving	Real-time	Energy saving
case 1	camcorder a	26.7%	camcorder b	29.7%
case 2		26.3%		29.2%
case 3		21.9%		24.8%
case 4		26.7%		29.7%
case 1	display_panel	22.5%	GPU_shader	21.1%
case 2		24.6%		20.4%
case 3		20.4%		13.1%
case 4		27.4%		20.8%

5.5.4 Memory-aware Cache Writebacks

As mentioned in Section 2, prior work have explored memory-aware writeback policies to improve memory efficiency. Although these work share the same objective with our unified memory controller, their approach is very different in that these writeback policies deal with write memory traffic (i.e. evicted cache lines) to DRAM. In contrast, our unified controller is focused on read memory traffic (i.e. cache misses) to DRAM. Furthermore, memory-aware writeback policies only handle LRU cache lines inside the LLC, whereas the orchestrated cache request scheduling in the unified controller is applied to cache requests waiting outside the LLC. Thanks to the differences, the unified controller can be combined with memory-aware writeback policies so that both write and read memory traffic can be optimized for memory efficiency.

To test memory-aware writebacks on the unified controller, we implement Eager

Read/Write Clustering (ERWC) [7] scheme in which LLC writebacks are triggered eagerly when the corresponding rows are activated by the memory scheduler for read or write transactions. Fig. 5.10 shows the summary of normalized row-buffer hits per activation in different scenarios. No cache bypassing is applied. All results are normalized to the number of row-buffer hits by the conventional design without any memory-aware writeback policy. For comparison, three scenarios are implemented, including the unified controller without memory-aware writebacks, the unified controller with ERWC policy in the LLC and the conventional design with memory-aware writebacks.

Without memory-aware writebacks, the unified controller brings more improvement compared with the conventional design with ERWC writeback policy. This has to do with the pattern of memory traffic in heterogeneous systems. Most heterogeneous traffic is generated from multimedia real-time cores and shows low memory locality. The result is high cache miss rate and heavy cache miss traffic, which gives more opportunities to row-buffer hit harvesting in the unified controller. On the other hand, memory-aware writeback policies rely on the assumption that the rows opened for memory transactions can be utilized by writebacks. However, in heterogeneous MPSoCs, this does not happen as often as in CPU-centric homogeneous systems. Due to high cache miss rate, memory traffic to DRAM consists mostly of read requests, which rarely share the same row with write requests (i.e. LLC writebacks). For example, the video decoder reads frames to be decoded and writes decoded frames at the same time. Since un-decoded and decoded frames are stored in different files, it is unlikely for them to share the same row in DRAM, which means read requests and write requests from the video decoder are unlikely to hit the same open rows. Moreover, when ERWC scheme is implemented in the unified memory controller, more row-buffer hits are collected as the result, compared with the unified controller without memory-aware writebacks. This is because when ERWC scheme and the unified controller are combined, both write and read memory traffic are optimized for

memory efficiency.

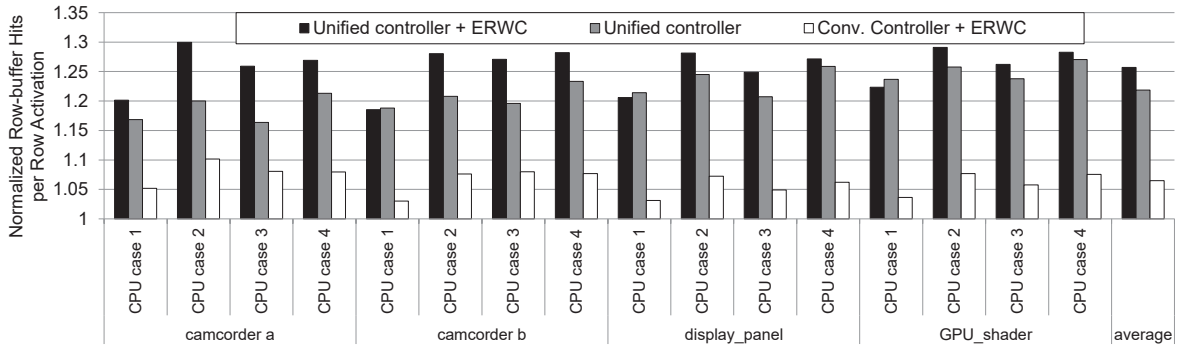


Figure 5.10: Summary of *normalized row-buffer hits per activation* in one million cycles.

5.5.5 CPU IPC

On CPU IPC, the proposed controller with orchestrated schedulers has mixed impacts. The summary of CPU IPC by the unified controller, normalized with respect to the conventional controller, is shown in Fig. 5.11. Without cache bypassing, CPU IPC is suppressed by the unified controller as the average IPC drops 10%. This is because multimedia memory traffic from real-time cores often manifests higher spatial locality than CPU traffic. When row-buffer hit harvesting is performed, real-time traffic receives more service than CPU traffic which is slowed down as the result. However, when cache bypassing is applied, CPU traffic, faced with less competitions at the last-level cache, receives more benefits from row-buffer hit harvesting. Hence CPU IPC is gradually improved as bypassing ratio increases. When the ratio is 60%, the average CPU IPC by the unified controller ends up 10% higher than the conventional controller. Overall, performance of the unified controller and the conventional controller are comparable in terms of CPU IPC.

The mixed impacts on CPU IPC happen when the applied scheduling policy at the last-level cache does not distinguish CPU traffic from the rest. However, this can be improved by assigning higher priority to CPU traffic while searching for row-buffer hit

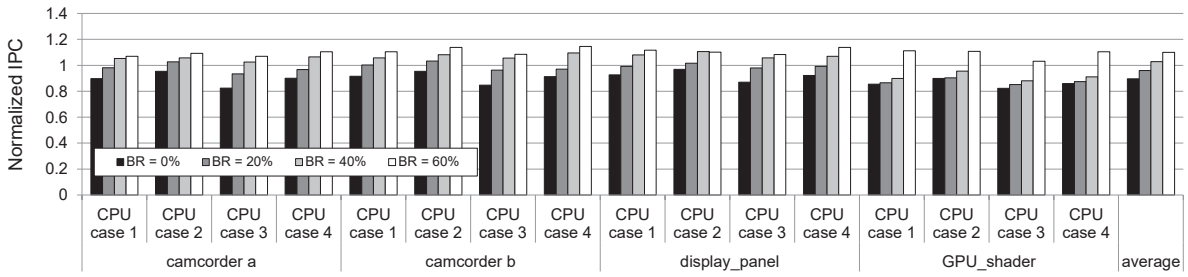


Figure 5.11: Summary of *normalized CPU IPCs* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM. **Policy A** is applied to both designs.

opportunities or scheduling memory requests. So far Policy A in Tab. 5.1 has been applied. Next, to improve CPU IPC on the proposed architecture, we apply Policy B where CPU requests are prioritized second only to the requests hitting open rows.

Fig. 5.12 shows the summary of CPU IPC by the unified controller using scheduling Policy B, normalized with respect to the conventional controller with scheduling Policy A. Thanks to the prioritization of CPU traffic, CPU IPC by the unified controller is higher than the conventional controller in all cases. The average IPC improvement ranges from 9.9% to 38% as the bypassing ratio increases. Fig. 5.13 shows the same comparison in terms of normalized DRAM bandwidth. As can be seen, less improvement on DRAM bandwidth is achieved compared with Fig. 5.9. Lower than 5% average improvement is observed for all cases. By assigning higher priority to CPU requests, the unified controller reshapes memory traffic with more requests showing low spatial locality. As a result, DRAM bandwidth has less potential for improvement, even though requests for active row-buffers still have the highest priority.

Next, we compare two memory controller designs with Policy B applied to both for scheduling, which means both controllers assign more priority to CPU traffic. Normalized CPU IPC and DRAM bandwidth by the unified controller are shown in Fig. 5.14 and Fig. 5.15. The unified controller shows improvement on CPU IPC by 10% - 29.8% on

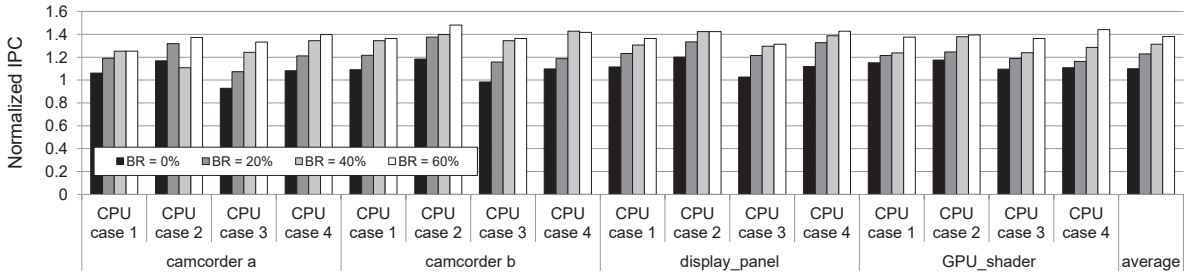


Figure 5.12: Summary of *normalized CPU IPCs* in one million cycles by the proposed memory controller using **Policy B** compared with the conventional design with **Policy A**.

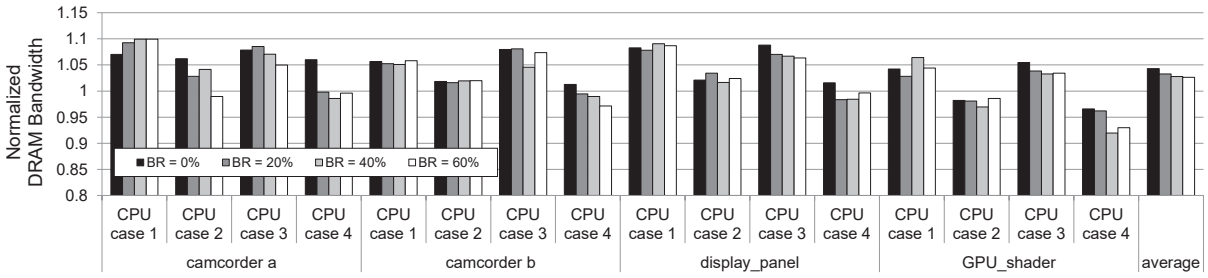


Figure 5.13: Summary of *normalized DRAM bandwidth* in one million cycles by the proposed memory controller using **Policy B** compared with the conventional design with **Policy A**.

average depending on the bypassing ratio. The improvement is slightly lower than that in Fig. 5.12, which is reasonable because the conventional design now attains higher CPU IPC using Policy B. Yet, the unified controller still has much better IPC under the same CPU-aware policy because row-buffer hit harvesting helps reduce average memory latency which relieves the contention pressure against CPU traffic. Meanwhile, since low-locality CPU traffic is adverse to row-buffer hit rate, by adopting Policy B the conventional design suffers loss on DRAM bandwidth. Hence more improvement on DRAM bandwidth by the unified controller is observed in Fig. 5.15 compared with Fig. 5.13. Above 5% improvement on average DRAM bandwidth is achieved regardless of bypassing ratio.

To further explore the impact of prioritizing CPU traffic, we test Policy C as in Tab. 5.1 which assigns the highest priority to CPU requests. In this policy, a CPU request going

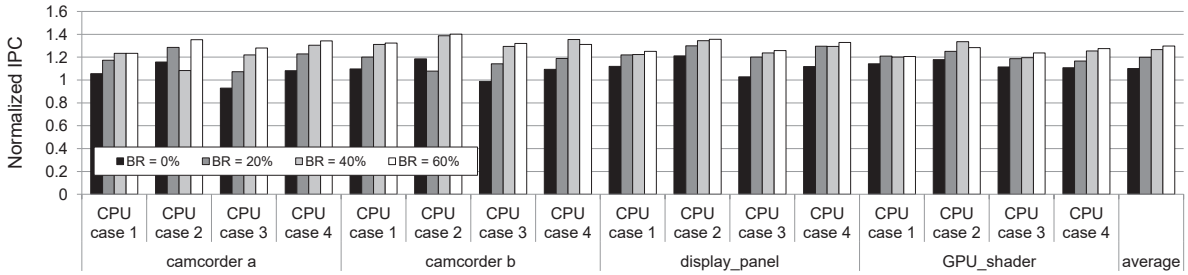


Figure 5.14: Summary of *normalized CPU IPCs* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM. **Policy B** is applied to both.

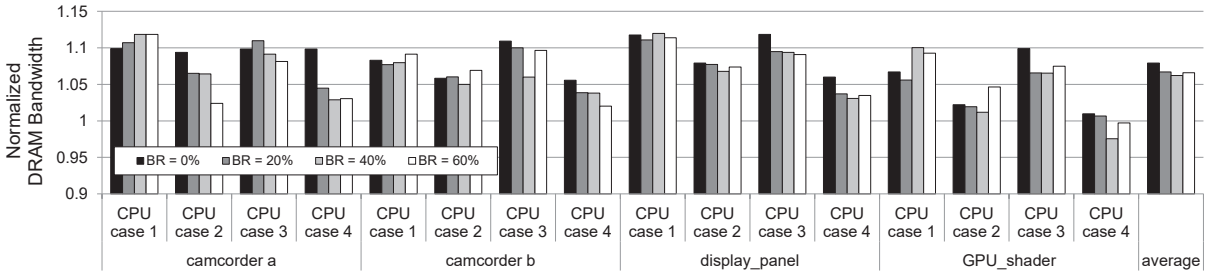


Figure 5.15: Summary of *normalized DRAM bandwidth* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM. **Policy B** is applied to both.

to a closed row-buffer precedes a real-time request for an open row-buffer. The updated CPU IPC and DRAM bandwidth by the unified controller are shown in Fig. 5.16 and Fig. 5.17.

When CPU requests are assigned with the highest priority level, the unified controller further facilitates CPU traffic by harvesting row-buffer hit opportunities. As shown in Fig. 5.16, when the bypassing ratio is 0%, the unified controller improves CPU IPC by 15.4% on average, which is about 50% relatively more compared with the case where Policy B is used (10% average improvement when BR = 0% in Fig. 5.14). The difference of CPU IPC improvement under Policy B and C shrinks as cache bypassing ratio increases. When the bypassing ratio is 60%, IPC improvements under Policy B and C are 29.8% and 31.5% respectively. The shrinking margin is caused by the fact that cache bypassing is beneficial

to CPU IPC as shown in Fig. 5.11, since it removes memory contention at the last-level cache. Therefore, the prioritization of CPU traffic has less effect on IPC when a large portion of real-time traffic bypasses the last-level cache.

As for DRAM bandwidth, the unified controller does not introduce any noticeable improvement under Policy C. As shown in Fig. 5.17, the average DRAM bandwidth by the unified controller shows smaller than 1% difference from the conventional design regardless of cache bypassing ratio. Since Policy C gives the highest priority to CPU traffic, real-time traffic is suppressed. Row-buffer hit harvesting in the unified controller reduces the latency for CPU requests, but fails to capture row-buffer hit opportunities from real-time traffic. As a result, the overall DRAM bandwidth remains the same between the unified controller and the conventional controller.

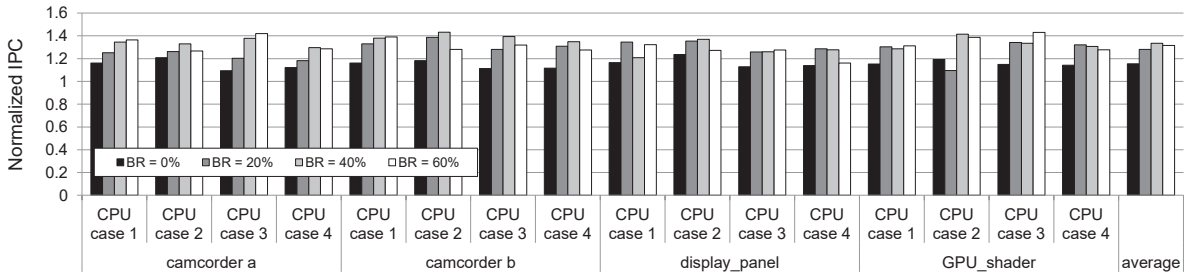


Figure 5.16: Summary of *normalized CPU IPCs* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM. **Policy C** is applied to both.

In summary, without the consideration of CPU traffic, the proposed controller exerts mixed impacts on the IPC depending on the cache bypassing ratio. A higher bypassing ratio leads to better CPU IPC as CPU traffic receives more benefits from the row-buffer hit harvesting. When CPU traffic is prioritized during row-buffer hit harvesting and memory scheduling, the unified controller achieves better IPC than the conventional design, although the improvement on DRAM bandwidth reduces. In the extreme case where CPU requests have higher priority than row-buffer hits, the unified controller achieves up to

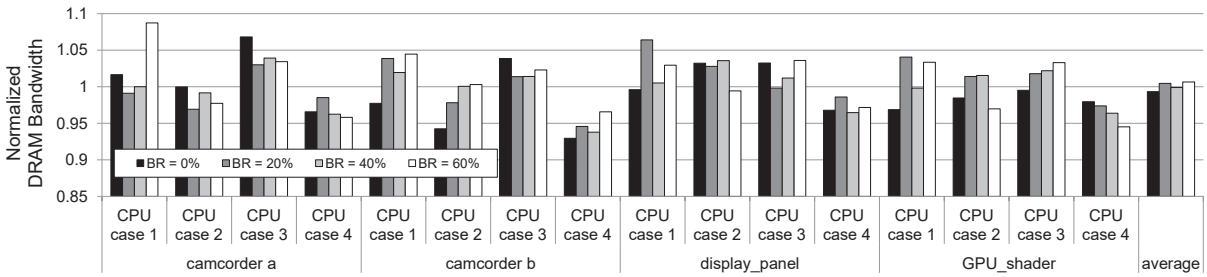


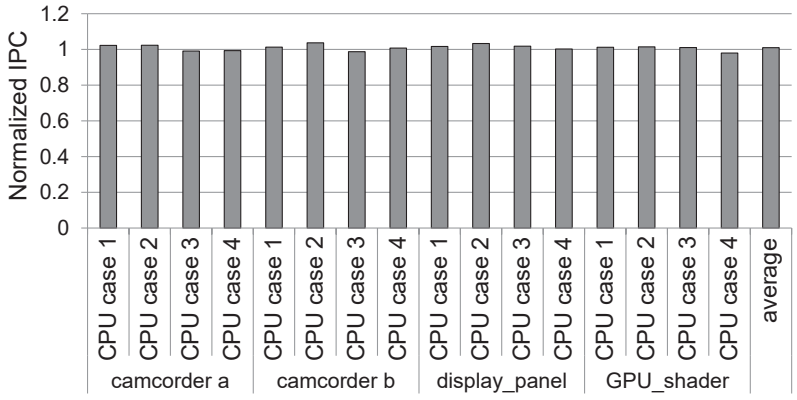
Figure 5.17: Summary of *normalized DRAM bandwidth* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM. **Policy C** is applied to both.

31.5% improvement on CPU IPC whereas the average DRAM bandwidth remains almost the same as the conventional design.

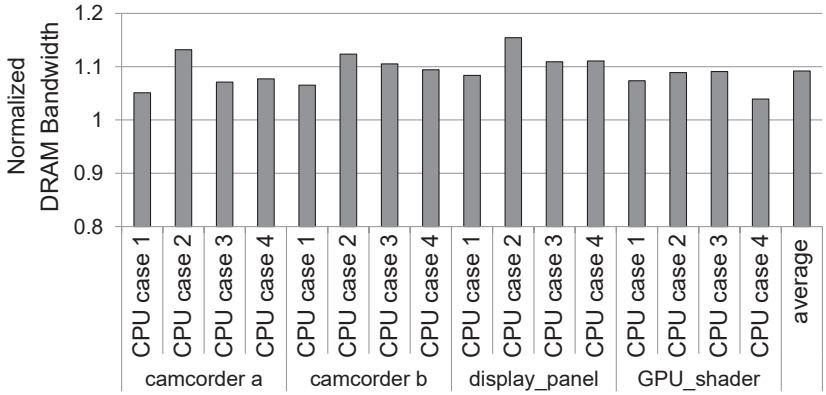
5.5.6 Dynamic Scheduling

Experimental results in the previous section demonstrate the potential of the unified controller in improving CPU IPC and DRAM bandwidth. Static scheduling policies in Tab. 1 help the unified controller to achieve either higher bandwidth or higher IPC than the conventional design. However, since CPU traffic has low row-buffer hit rate, prioritization of CPU requests cannot be done without limiting row-buffer hits. On the other hand, row-buffer hit harvesting inevitably slows down CPU traffic. Therefore, CPU IPC and memory efficiency are two disparate objectives for the schedulers. To balance the priority levels between the two, a dynamic policy for the orchestrated scheduling in the unified controller was introduced in Section 4. In this section, we will evaluate this dynamic policy in terms of its efficacy in delivering target CPU IPC and improving DRAM bandwidth at the same time.

For the evaluation, we measure the average IPCs by the conventional design under Policy A and use them as IPC targets. For each test case, we set IPC_{pref} and IPC_{min} in the dynamic policy (Algorithm 1) respectively as $\times 1.03$ and $\times 0.97$ of the average IPC



(a) Normalized IPC



(b) Normalized bandwidth

Figure 5.18: Summaries of *normalized IPC* and *normalized DRAM bandwidth* by the unified memory controller with the dynamic orchestrated scheduling policy. All results are normalized to those by the conventional design applying Policy A. No cache bypassing is applied.

by the conventional design. That means we allow ±3% difference from the IPC by the conventional design. Fig. 5.18 shows both summaries of normalized IPC and normalized bandwidth by the unified controller. All results are normalized to those by the conventional design. As can be seen, the normalized IPC remains around 1 in all cases, which means the dynamic policy is effective in delivering target IPCs. On average, the normalized IPC is 1.01. Meanwhile, we still achieve higher bandwidth than the conventional design with an average bandwidth growth by 9.2%. By comparison, when Policy A was previously applied to both designs (as shown in Fig. 5.9 and Fig. 5.11), the bandwidth was improved by

16.8% at the expense of an average IPC reduction by 10%; when Policy B was applied to the unified controller and Policy A was adopted in the conventional design (as shown in Fig. 5.12 and Fig. 5.13), the average IPC by the unified controller was improved by 9.9% but the improvement on bandwidth shrank to only 4.3%. The dynamic policy avoids either inadequate or excessive IPCs while improving bandwidth by a decent margin.

5.5.7 Sensitivity Analysis

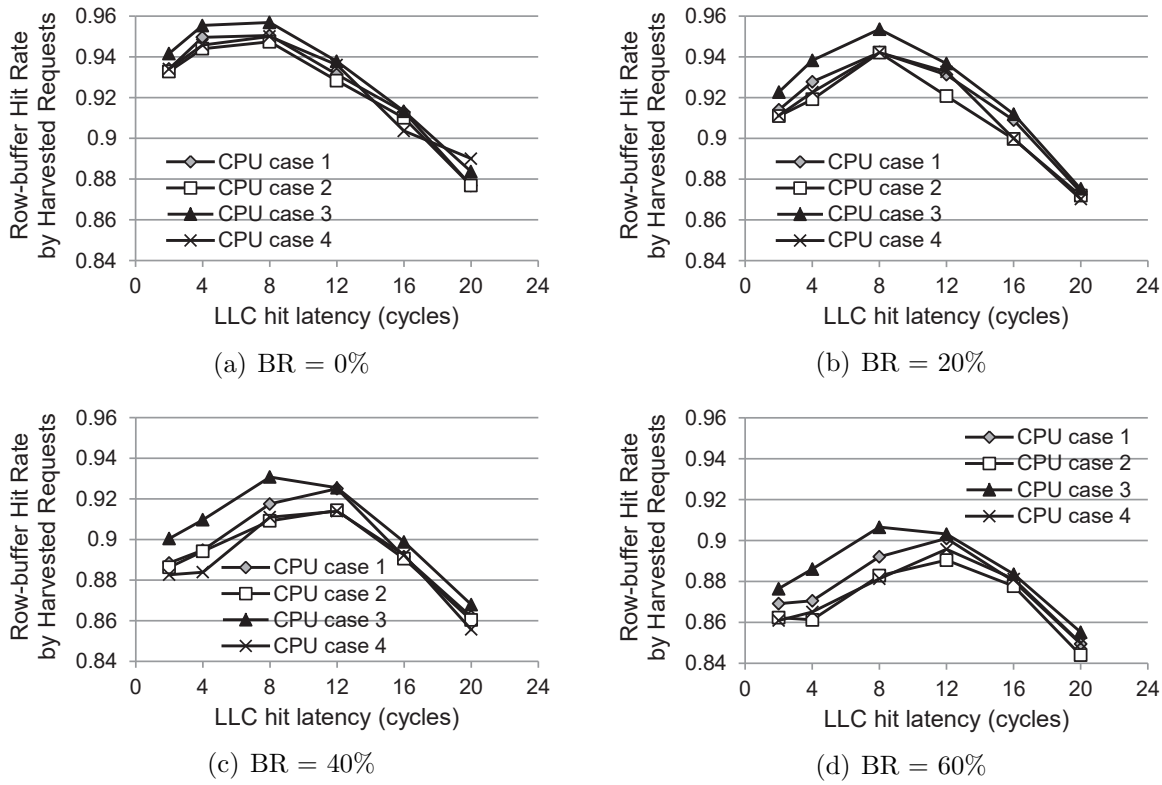
To have a better understanding of the proposed unified controller, we will explore different aspects of architecture design, including last-level cache latency, request storage volume and bank-level parallelism, and evaluate their impacts on the performance of the unified controller.

Last-Level Cache Latency

Row-buffer hit harvesting enhances row-buffer hit rate by facilitating cache requests that are likely to hit open row-buffers. However, the harvested requests still need to be checked by the last-level cache before, if necessary, moving to the next stage for memory scheduling. Last-level cache latency can be critical in determining the effectiveness of row-buffer hit harvesting.

Fig. 5.19 shows the row-buffer hit rates by harvested cache requests with respect to different last-level cache latencies and cache bypassing ratios of real-time traffic. Each sub-figure shows the row-buffer hit rate as a function of last-level cache latency under a certain bypassing ratio. The horizontal axis represents the latency of cache hits while cache miss latency is set to 75% of cache hit latency in the experiment. Four CPU test cases and the real-time application *camcorder a* are simulated. Policy A (Tab. 5.1) is used for scheduling.

As can be seen, the row-buffer hit rate has a non-linear correlation with the cache



(a) BR = 0%

(b) BR = 20%

(c) BR = 40%

(d) BR = 60%

Figure 5.19: Summary of *row-buffer hit rates* by harvested cache requests by the proposed memory controller with different cache latencies and bypassing ratios. Real-time application *camcorder a* is used for simulation.

latency. In all cases, the highest hit rates are obtained when cache hit latency is set between 8 and 12 cycles. Row-buffer hit rate suffers loss when cache hit latency is either higher or lower than that range. When last-level cache latency is too high, it takes a long time for the harvested cache request to be validated for memory scheduling. There is a high chance that the corresponding row-buffer is no longer open, because the memory controller may close open rows when there are no requests for them. When cache latency is too low, cache misses that are not going to open row-buffers pass through the last-level cache quickly, which intensifies the memory contention at the memory controller. Meanwhile, the requests that travel through the fast lane do not have much advantage over the others. As a result, the row-buffer hit rate by harvested cache requests is suppressed.

Moreover, cache bypassing limits row-buffer hit rate by harvested requests by introducing more memory contention at the memory controller. As shown in Fig. 5.19, the overall row-buffer hit rate drops as the bypassing ratio rises. The drop is particularly obvious when cache latency is low, because during this stage the memory contention is the major factor suppressing hit rate.

Last-Level Cache Size

Besides last-level cache latency, the size of last-level cache also plays a role in the effectiveness of row-buffer hit harvesting. A bigger size leads to a higher hit rate and less cache misses. Since the row-buffer hit harvesting is only effective when the harvested requests reach open rows in DRAM, having less cache misses results in less performance improvement from row-buffer hit harvesting. Fig. 5.20 shows the summary of normalized row-buffer hits per activation by the unified memory controller when the LLC is set to different sizes ranging from 1MB to 16MB. As expected, the number of row-buffer hits decreases as LLC size is increased. Note that this result does not rule out the necessity of row-buffer hit harvesting, because the LLC is usually small due to the limited power and area overhead that can be afforded on an MPSoC. For example, a state-of-art MPSoC, Snapdragon 845 [39] has only 2MB in its LLC (L3 cache).

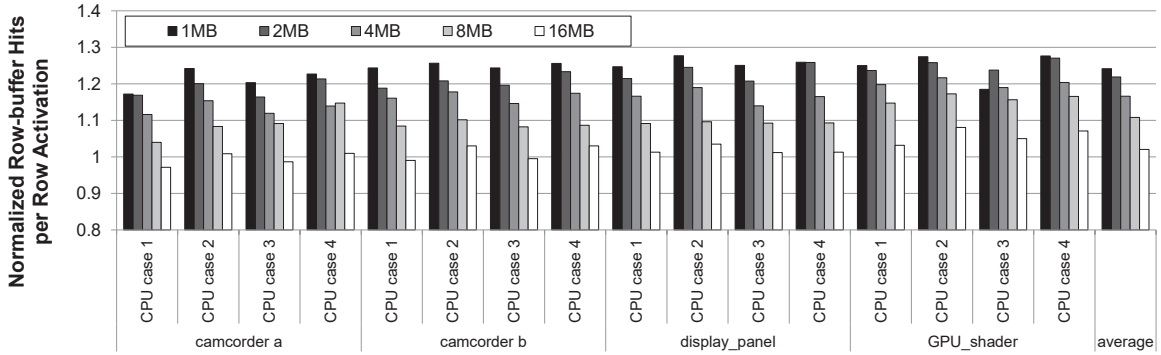
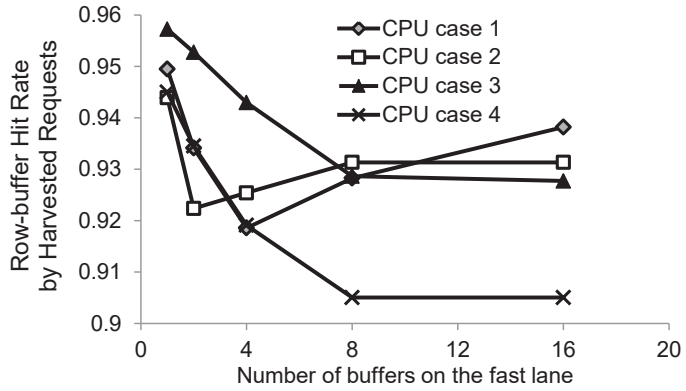


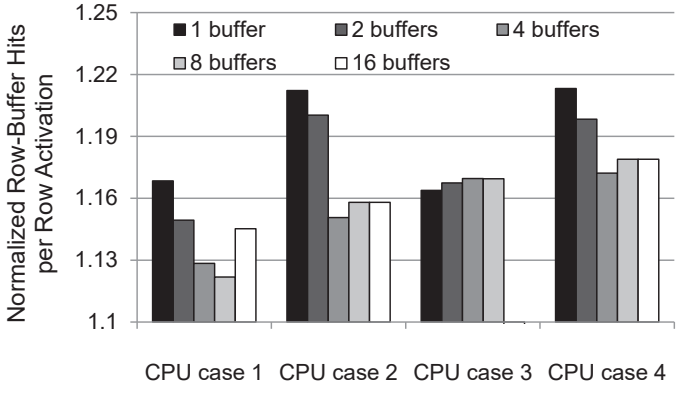
Figure 5.20: Summary of *normalized row-buffer hits per activation* in one million cycles by the proposed memory controller with different LLC sizes. No cache bypassing is applied.

Fast Lane Buffers

As mentioned in Section 3.1, we only use one buffer on the fast lane to limit the queuing delay of harvested cache requests. Yet, intuitively, having more fast lane buffers might facilitate row-buffer hit harvesting since more capacity would be given to potential row-buffer hits. To explore the impact of fast lane buffer capacity, we evaluate the row-buffer hit rate by harvested cache requests and the number of row-buffer hits in DRAM respectively, with respect to different numbers of buffers on the fast lane, as shown in Fig. 5.21.



(a) row-buffer hit rates by harvested cache requests



(b) normalized row-buffer hits

Figure 5.21: Summary of *row-buffer hit rates* by harvested cache requests and *normalized row-buffer hits per activation* by the proposed memory controller with different numbers of buffers on the fast lane. Real-time application *camcorder a* is used for simulation. No cache bypassing is applied.

As can be seen in Fig. 5.21(a), when the number of buffers is increased from 1 to 4, the row-buffer hit rate by harvested requests decreases in all CPU test cases. This is because having more buffers leads to higher delay on the fast lane. While harvested cache requests wait on the fast lane, their target rows may be closed by the scheduler, which makes the row-buffer harvesting futile. However, in CPU test cases 1 and 2, the hit rate increases later on when more buffers are inserted; in CPU test cases 3 and 4, the hit rate hardly decreases any more once the buffer count is raised above 4. Depending on the traffic pattern, the fast lane may collect multiple requests for the same open row rather than multiple requests for different open rows. In the former case, these requests virtually form into a batch. Even if their target row is closed for the first request in the batch, the following requests will end up as open row hits. This way we can still achieve high row-buffer hit rates. Besides, the more buffers we have on the fast lane, the more possible it is for harvested requests to form into batches.

In Fig. 5.21(b), a similar phenomenon can be observed in terms of the number of row-buffer hits. In most cases, the number of row-buffer hits sees an initial drop when more buffers are introduced to the fast lane. However, as the number of buffers is further increased, the number of row-buffer hits eventually gets higher in all cases. Although the row-buffer hit rate by harvested requests is lower than the hit rate when the fast lane has only one buffer, since there are more harvested requests, we still end up having more row-buffer hits.

Request Buffers

Since the motivation behind the unified memory controller is to improve the scheduler's visibility into memory locality, the efficacy of the unified controller depends on two factors: the memory/DRAM request storage and the cache request storage. These two factors respectively determine how much visibility the memory scheduler already has and

how much more visibility the scheduler can obtain from the harvesting mode.

Fig. 5.22 shows the percentage of row-buffer hits by harvested cache requests with respect to different numbers of buffers for cache and memory requests. Note that the number of memory request buffers refers to the size of each transaction queue in the memory controller, assuming write and read queues share the same size. The percentage of harvested row-buffer hits is calculated as the ratio of the number of row-buffer hits by harvested requests to the total number of row-buffer hits. A higher percentage (represented by a lighter shade in Fig. 5.22) means more contribution is made by the harvesting mode to row-buffer hits. Four CPU test cases and the real-time application *camcorder a* are simulated. Each node in Fig. 5.22 represents the average result of all test cases with respect to specific numbers of buffers for cache and memory requests. Policy A (Tab. 5.1) is used for scheduling. The rest of experiment settings can be seen in Tab. 6.1.

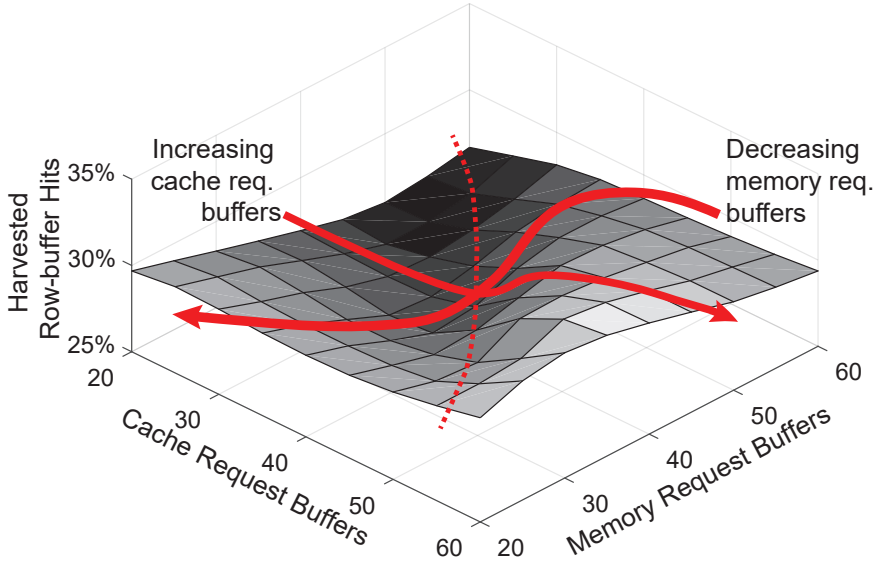


Figure 5.22: The percentage of harvested row-buffer hits with different number of buffers for cache and memory requests. Real-time application *camcorder a* is used for simulation. No cache bypassing is applied.

As shown, under a certain number of memory request buffers, the percentage of harvested row-buffer hits rises as the number of cache request buffers increases. This is

because more cache request buffers means more visibility can be gained through row-buffer hit harvesting. As a result, a higher percentage of row-buffer hits is discovered through the harvesting. On the other hand, given the number of cache request buffers, the overall percentage of harvested row-buffer hits increases as the number of memory request buffers decreases. This is expected because with fewer memory request buffers, the scheduler has less visibility and relies more on the contribution from row-buffer hit harvesting. However, the percentage of harvested row-buffer hits does not always increase along the decrement of memory request buffers. As shown in Fig. 5.22, the percentage of harvested row-buffer hits goes through a downward slope in the midst of an upward trend. In the meantime, the percentage takes a leap on the other dimension along with the increment of cache request buffers. A "fold" is formed as the result.

With a closer look, one can tell the fold is located along $\text{Buff}_{\text{mem}} + \text{Buff}_{\text{cache}} \in [70, 80]$. This formation has to do with the memory locality pattern of the real-time application *camcorder a*. Due to the inherent composition of the heterogeneous system, such as the structure of interconnects and the frequencies of real-time cores etc., the memory traffic arriving at the last-level cache manifests a recurring pattern in terms of spatial locality. In the case of *camcorder a*, many row-buffer hit opportunities show up every 70-80 requests. That means when the total number of request buffers lies in this range, many row-buffer hit opportunities are arriving in cache request buffers, which greatly augments the impact of row-buffer hit harvesting.

Bank-Level Parallelism

As discussed in Section 5.2, increasing row-buffer hits helps removing evitable precharge operations and improving DRAM bandwidth. The underlying assumption is that we do not have unlimited bank-level parallelism. Since every memory bank operates independently, requests for different banks can be processed in parallel. The more memory

banks we have, the higher bank-level parallelism can be achieved. If there are very few banks, the bank-level parallelism is limited and requests often need to access the same memory bank. That is when row-buffer hit harvesting becomes important. However, if we have plenty of memory banks, the bank-level parallelism is high. Hence when bank conflicts happen, we can easily hide memory stalls by scheduling requests to other banks. As a result, DRAM bandwidth does not suffer much loss from evitable precharge operations. This points out the fact that the bank-level parallelism plays a role in determining the impact of the unified controller in improving DRAM bandwidth.

Fig. 5.23 shows normalized DRAM bandwidth by the proposed memory controller with different number of ranks and banks in DRAM. The normalized bandwidth is calculated as the ratio of DRAM bandwidth by the unified controller to the bandwidth by the conventional design with respect to the same test case. In each case, the total DRAM volume is the same, i.e. 4GB. The number of rows per bank is adjusted to accommodate the change in the number of ranks and banks. Only one memory channel is used.

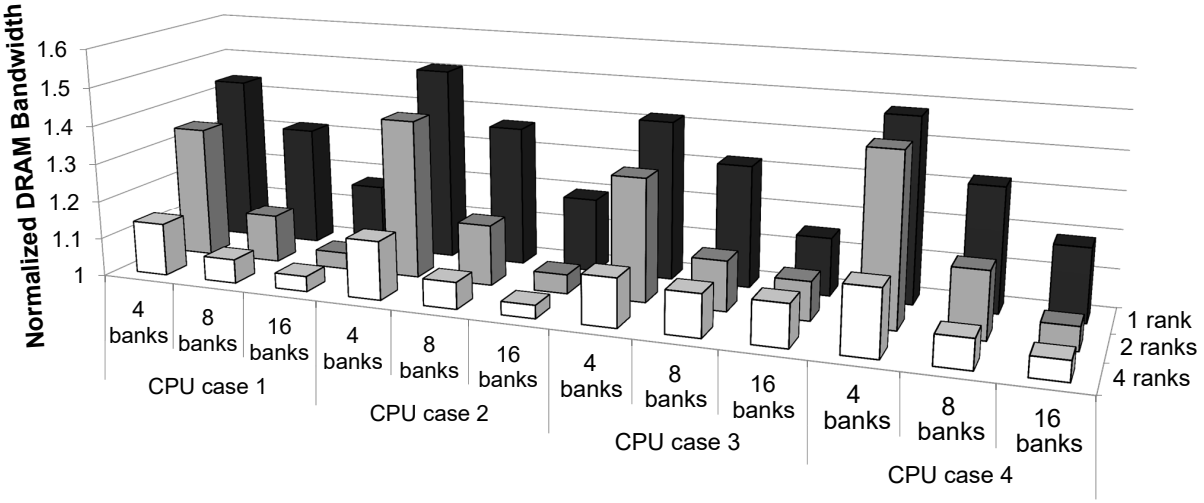


Figure 5.23: Normalized DRAM bandwidth by the proposed memory controller with different numbers of ranks and banks (per rank), normalized with respect to the conventional design. Real-time application *camcorder a* is used for simulation. No cache bypassing is applied.

As shown in Fig. 5.23, the normalized bandwidth remains above 1 meaning the unified controller results in higher bandwidth than the conventional design in every case. Moreover, the normalized bandwidth varies between 103.4% and 150.8% as different numbers of ranks and banks are used. As expected, the normalized bandwidth increases as the total number of banks ($= ranks \times banks/rank$) is reduced and so is the bank-level parallelism. That means the unified controller brings more benefit when the bank-level parallelism is lessened. For each CPU test case, the best normalized bandwidth appears when there are 4 banks in total (1 rank and 4 banks/rank); the worst normalized bandwidth is obtained when there are 64 banks (4 ranks and 16 banks/rank). Note that as the bank-level parallelism decreases, both the unified and the conventional memory controllers observe loss in DRAM bandwidth. Yet the loss in the unified controller is less than the other. Thus the normalized bandwidth ends up getting higher.

One may also observe that between the cases that have the same number of banks in total but different configurations (e.g. 1 rank and 8 banks/rank vs. 2 ranks and 4 banks/rank), the normalized DRAM bandwidth is similar except that the case with more ranks tends to be slightly better. This is because we use the higher bits in physical address to index ranks (as shown in Tab. 6.1). Among different ranks, memory traffic is not as interleaved as among banks. Hence the test case with more ranks but fewer banks tend to be more limited in terms of parallelism, which leads to higher normalized bandwidth by the unified controller.

Note that when there are 4 ranks and 16 banks/rank the normalized bandwidth is almost 1, meaning that the unified controller hardly has any advantage over the conventional design. However, such a configuration is not practical as the superior bank-level parallelism comes at the price of excessive DRAM power. In fact, it is not supported by JEDEC standard for LPDDR4 [20]. Therefore, we still need to increase row-buffer hits to cope with the limited bank-level parallelism.

5.6 Related Work

Memory-aware Cache Writeback: Traditionally cache writebacks are triggered only by cache line evictions. When cache misses happen, cache writebacks are also induced to make room for new cache lines. However, these writebacks can easily cause contentions against the fetching of new cache lines. The Eager Writeback [29] was proposed to re-distribute and balance traffic by writing dirty cache lines to memory prior to their eviction so that the writebacks are less likely to impede the fetching of cache miss data. The dirty cache line at LRU state will be written back once the bus is idle. The Virtual Write Queue (VWQ) [48] was designed to coordinate memory scheduling with cache writebacks. As an extension to the write queue in the memory controller, the LRU cache lines in the last-level cache are visible to the memory controller and available for memory scheduling along with write requests. Writebacks are triggered when the target row of these dirty LRU cache lines is being accessed so that these scheduled writebacks will not cause extra row activations in DRAM. The Last-Write Predictor Guided (LWPG) writeback policy was proposed in [54]. The authors defined last-write blocks as cache blocks that will not be written again before being evicted to memory. A low overhead last-write predictor keeps track of last-write blocks which are available for memory scheduling without incurring extra memory write requests. Eager Read/Write Clustering (ERWC) [21] further reduces the number of row activations by clustering both write and read requests with cache writebacks to the same row-buffer. Eager writebacks are triggered by row activations regardless of the cause.

Aforementioned memory-aware cache writeback policies are summarized in Tab. 5.5 in comparison with the orchestrated cache request scheduling in our unified memory controller. These policies are characterized in terms of when they are triggered, where they are applied and the type of memory requests that are generated as the result. As shown, previous writeback policies are either triggered by cache eviction or row activation in DRAM. By comparison, orchestrated cache request scheduling is performed when the

cache controller searches for cache access requests. In addition, these writeback policies are applied to dirty cache lines which will be converted into write requests to DRAM, whereas the orchestrated scheduling is applied to cache access requests which, if turn out to be cache misses, will be converted into read requests. As can be seen, the orchestrated cache scheduling and memory-aware writeback policies are complementary to each other, since they are applied in different parts of the LLC and generating different types of requests to DRAM.

Table 5.5: Summary of memory-aware cache writeback policies in comparison with orchestrated cache request scheduling.

	VWQ	LWPG	ERWC	Orchestrated scheduling
When	Eviction/Activation	Eviction	Activation	Cache access
Where	Cache lines	Cache lines	Cache lines	Cache request buffers
Request type	Write	Write	Write	Read

Cache Bypassing: Similar to memory scheduling, last-level cache management also has to deal with the interference among heterogeneous cores. While the CPU relies on the last-level cache to reduce memory latency, real-time cores can often degrade cache hit rate due to their distinct traffic pattern. Heterogeneous LLC Management (HeLM) [33] collects runtime information to measure applications' cache sensitivity and tolerance against memory access latency. Those that are categorized as cache insensitive and latency tolerant will bypass the last-level cache. This work assumes no shared data among heterogeneous cores. Another cache management policy which is named coordinated bypassing and warp throttling (CBWT) [7] was proposed for the GPGPU. The authors noticed that under pure cache bypass policies, the massive amount of misses and bypassing requests may lead to on-chip resource congestion and limit the performance benefit. Therefore, the proposed bypass policy is coordinated with warp throttling and adaptively controls the parallelism when contention or congestion happens. To improve GPU performance, the Memory Request Prioritization Buffer (MRPB) [24] performs request reordering and cache bypassing so that requests are released into the cache in a more cache-friendly order. A

software-based adaptive cache bypassing framework for GPUs was proposed in [32] to adjust the bypass degree to match the size of applications' runtime footprints. In [58], a compiler framework was proposed to analyze GPU code and judiciously select global loads for cache access or bypass. In [59], a coordinated static and dynamic cache bypassing optimization framework for GPUs was proposed. This framework not only identifies the global loads that indicate strong preferences for caching or bypassing in compile-time, but also modulates the ratio of thread blocks that use or bypass the cache in run-time.

5.7 Chapter Summary

In this chapter, we showed that existing controllers for the last-level cache and DRAM with independent schedulers often lead to unnecessary precharges and row activations in DRAM, due to their limited visibility into memory traffic. These evitable operations introduce extra delays and power, limiting memory efficiency. To improve the visibility for memory scheduling, we proposed the unified memory controller with orchestrated schedulers for the last-level cache and DRAM. The unified memory controller harvests more row-buffer hits and achieves higher DRAM bandwidth compared with the conventional design. According to our evaluations, the unified controller increases the number of row-buffer hits and DRAM bandwidth by 22% and 16.8% respectively on average. With up to 60% of real-time traffic bypassing the last-level cache, row-buffer hits and bandwidth can still be improved by 17.6% and 13.9% respectively on average. Further, the proposed unified controller saves DRAM energy by up to 29.7% while achieving comparable CPU IPC. We also explored possible scheduling policies that improve CPU IPC for the unified controller, which indicates the unified controller's potential to achieve improvements in both memory efficiency and CPU IPC. In addition, we introduced a dynamic policy for the orchestrated scheduling in the unified controller to achieve target CPU IPC while improving memory

efficiency. At last, sensitivity analyses were conducted to provide more insight into the unified controller design.

5.8 Acknowledgement

This chapter, in part, is a reprint of the material as it appears in ACM Transactions on Design Automation of Electronics Systems, 2019. Song, Yang; Alavoine, Olivier; Lin, Bill. This chapter also includes the material as it appears in proceedings of ACM/IEEE Design Automation and Test in Europe, Conference and Exhibition, 2018. Song, Yang; Alavoine, Olivier; Lin, Bill. The dissertation author was the primary investigator and author of these papers.

Chapter 6

Locality-Aware Forwarding in Networks-on-Chip

6.1 Introduction

As the first memory fabric that memory requests visit in the memory subsystem, the NoC collects traffic streams from heterogeneous cores and forwards them to the last-level cache or the memory controller. Due to its vicinity to heterogeneous cores, providing QoS support in the network is a major concern and thus has drawn plenty of attention in recent years [31, 14, 37, 60]. Yet, because of its distance from DRAM, the NoC is rarely seen as a memory fabric that can have great impact on the memory efficiency. While memory efficiency studies are extensively conducted on the memory controller [42, 25, 4] and the last-level cache [48, 26, 21, 45], little attention has been paid to the NoC.

In the previous chapter, we have discussed improving memory efficiency through the extension of memory visibility to the scheduler. Specifically, the more requests that are visible to the scheduler, the more likely that it can find a request for an open row in DRAM. The same belief also motivated prior work on the last-level cache writeback policies

[48, 26, 21]. These work extend the candidate pool for memory scheduling into the last-level cache to offer more visibility of memory traffic to the scheduler. However, since the memory controller and the last-level cache only hold a small number of memory requests compared with those being transported in the NoC, improving local memory visibility alone may bring limited improvement. When memory locality is low, potential hits for open rows are often waiting in NoC routers which are beyond the visible range of the scheduler even given local visibility extension. As the transporter of memory requests, the NoC plays a unique role in preserving row-buffer locality during traffic forwarding. Unfortunately, there have been very few studies on the NoC [18, 38] discussing the improvement of row-buffer locality for the benefit of memory efficiency.

In this chapter, we propose a locality-aware NoC router design with lightweight row-index caches for row-buffer locality-aware traffic forwarding. The goal of the proposed design is to shorten the delay between requests for the same row so that they can arrive at the memory controller within a visible distance. Meanwhile, the proposed design is capable of providing QoS support. The contributions of our work are summarized as follows.

- We show the challenge of improving memory efficiency in heterogeneous MPSoCs, and identify the limitation of previous approaches. Without managing memory locality through the NoC, local extension of memory visibility introduces trivial improvement.
- We propose the design of a locality-aware NoC router architecture. The proposed architecture is equipped with row-index caches to track row indices that have appeared in recently forwarded packets. These lightweight caches enable efficient locality-aware forwarding in NoC routers and can be implemented with extant priority-based allocators.
- We introduce a locality-aware allocation scheme that improves row-buffer locality and provides QoS support at the same time.

- We evaluate the proposed design on several NoC topologies with the memory traffic of a state-of-art heterogeneous MPSoC [39]. The results show that our locality-aware NoC router achieves higher row-buffer hit count and lower memory latency than previous memory-aware methods, in addition to providing QoS guarantees. Compared with QoS-aware routers, the proposed design increases row-buffer hits by 58.32% and reduces average memory latency by 14.45%.

The rest of this chapter is organized as follows: Section 6.2 describes the background and motivation of our work. Section 6.3 details our router design with locality-aware forwarding and QoS support. Section 6.4 presents the evaluation results. A review of related work and conclusions follow in Sections 6.5 and 6.6.

6.2 Background and Motivation

Memory subsystem design for heterogeneous MPSoCs has been extensively investigated by previous work. A major challenge being tackled by this body of literature is the delivery of quality-of-service (QoS) to heterogeneous cores in consideration of the discrepancy in their memory access patterns, data access rates and performance targets.

Previous work on NoC designs have introduced QoS awareness to routers and allocators. One of the earliest studies is the globally-synchronized frames [31]. The authors batch network flits into frames with a constant size. QoS guarantee is provided by allocating a specific fraction of a frame to each traffic flow. During VC/switch allocations, flits from older frames are prioritized over those from younger frames. As a refined version of [31], the locally-synchronized frames [37] deploys frame-based scheduling at the granularity of router links to enable efficient flow control. The preemptive virtual clock [14] proposed a cycle-based framing strategy. Routers maintain per-flow bandwidth consumption counters which are reset at the end of each frame period. Packets' priorities are evaluated according

to on their bandwidth consumptions in relative to the provisioned bandwidth. Recently, a QoS-aware scheduling scheme for heterogeneous multicore systems was presented in [60]. Asynchronous batch scheduling is performed at each router to form mini-batches and enable differentiated bandwidth allocations to heterogeneous cores. Inside a batch, CPU packets are prioritized over GPU packets, and read packets are prioritized over writes. Note that priority arbiters are a necessity for QoS-aware router implementations. While QoS awareness in the NoC has been well explored, memory performance, often in conflict with the QoS, is usually ignored by NoC designs.

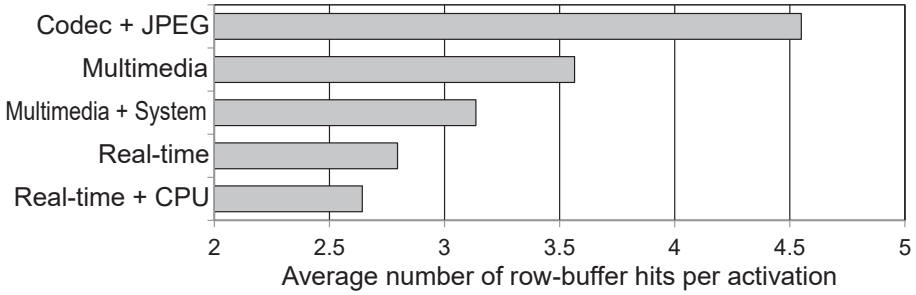


Figure 6.1: Average number of row-buffer hits per row activation with respect to different number of heterogeneous cores being served in the memory subsystem. A single arbiter applying round-robin policy is used to forward memory traffic to the memory controller.

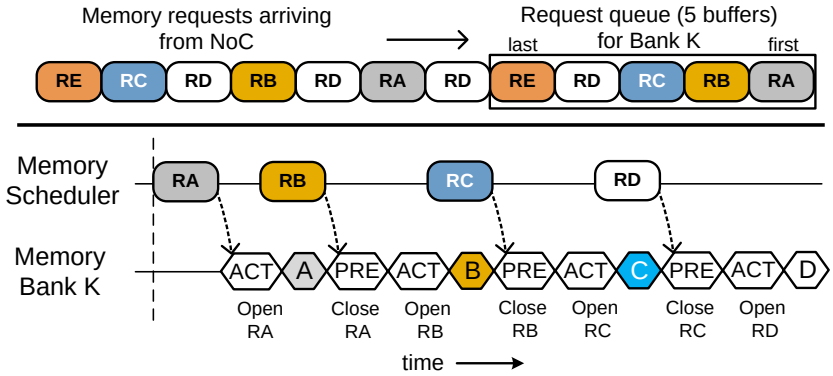
Multimedia data constitutes a large portion of memory traffic in heterogeneous MPSoCs. This part of memory traffic typically possesses high spatial locality which is beneficial to row-buffer hit rate in DRAM. However, as traffic flows merge into a single stream to DRAM, they often dilute each other. Without proper care, memory locality is quickly lost during the dilution, resulting in fewer row-buffer hits and lower memory efficiency in DRAM. Fig. 6.1[1] shows the average number of row-buffer hits with respect to different numbers of heterogeneous cores sharing the memory subsystem. As more cores are included, the average number of row-buffer hits drops rapidly from 4.55 to 2.64.

[1]Emulated real-time cores are listed in Tab. 6.3. CPU test case 1 (Tab. 6.2) and high performance mode (Tab. 6.3) are used for simulation. Experiment setup is detailed in Section 6.4.

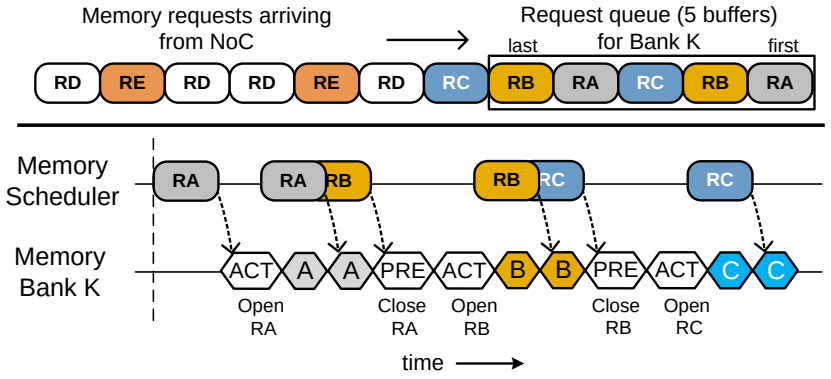
To further explain this phenomenon, Fig. 6.2 illustrates two scenarios with different degrees of memory locality leading to different memory efficiencies. Assume there are five request buffers designated to each memory bank. Only requests in these buffers are visible to the scheduler which applies FR-FCFS policy for memory scheduling. In both scenarios, the memory controller receives the same requests from the NoC for memory bank K but in different orders. The memory traffic in Fig. 6.2(a) has low row-buffer locality, meaning that requests for the same rows are separated far apart. As a result, the memory scheduler cannot find the opportunity to access any active row. Instead, the scheduler keeps precharging the row-buffer and serves memory requests in their arrival order. In the other scenario shown in Fig. 6.2(b), memory requests for the same rows stay close enough so that they are visible to the scheduler at the same time. Two row-buffer hits are found after each row activation. By comparison, the second scenario is more desirable because less delay and energy are induced. However, the first scenario is more common in heterogeneous systems due to the locality dilution.

The locality dilution can be tackled in two ways: 1) increase memory visibility and make more requests visible to the scheduler so that the dilution can be tolerated; 2) preserve memory locality and prevent requests for the same row from being dispersed. The first approach has been well explored by previous work [48, 26, 21, 45], whereas the second approach has drawn little attention in recent years [18, 38].

Fig. 6.3 shows the results from varying memory visibility and locality. In the simulation, a single arbiter fetches requests from cores to the memory controller in a round-robin fashion. Different extents of memory visibility are realized by different numbers of request buffers in the memory controller. To vary memory locality, we modify round-robin policy at the single arbiter so that a requestor can be served multiple times in consecutive cycles. The goal is to create bursty traffic so that memory locality can be partially preserved during arbitration. Fig. 6.3 includes the results from applying different burst lengths.



(a) low memory locality and low memory efficiency



(b) high memory locality and high memory efficiency

Figure 6.2: An example of memory locality of NoC traffic affecting memory efficiency in DRAM. Requests are labeled by indices of their target rows. Those for the same row are painted in the same color.

The maximum burst length refers to the maximum number of times that a requestor can be served in a row, if possible, before the next requestor takes its turn. A longer burst preserves more memory locality.

As shown, the average number of row-buffer hits is improved trivially as the buffers are increased from 10 to 80 regardless of burst length. This is because in a multicore system with numerous traffic flows, the memory locality is too diluted to be captured by a reasonable number of request buffers. By comparison, creating bursty traffic provides a more noticeable impact on row-buffer hits. Nonetheless, the average number of row-buffer hits is quickly saturated once the maximum burst length increases above 4, because

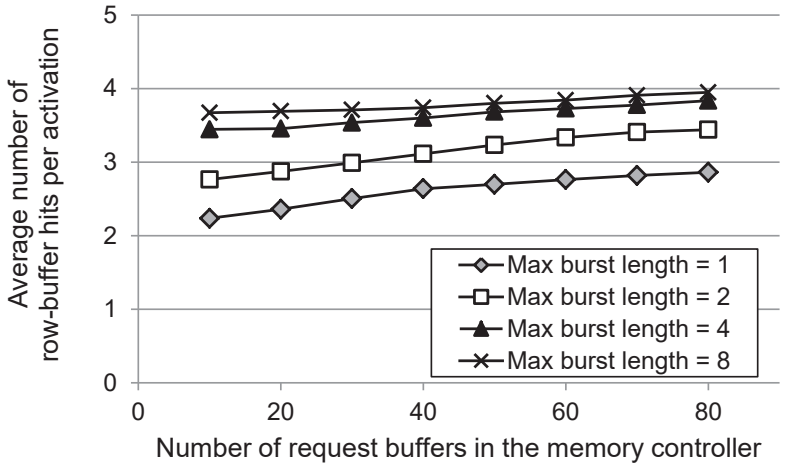


Figure 6.3: Average number of row-buffer hits per row activation with respect to different numbers of request buffers (memory visibility) in the memory controller, and different maximum burst lengths (memory locality). Experiment setup is the same as in Fig. 6.1.

creating bursty traffic blindly introduces pollution from traffic flows with low memory locality. Overall, preserving row-buffer locality of memory traffic shows more potential than increasing memory visibility. The locality preservation should be performed in the NoC where the dilution happens.

Prior work on improving memory efficiency through the NoC were focused on memory-aware router design [18, 38]. Jang et al. [18] proposed a DRAM-aware flow control protocol. In this protocol, VC allocators prioritize the packets that travel to the same row as the preceding packet and avoid allocating a virtual channel to any packet which can cause bank conflict with its predecessor. It facilitates batch formation of packets for the same row-buffer at source. However, it is not clear how this protocol accommodates various notions of QoS in modern heterogeneous systems. Pimpalkhute et al. [38] introduced a DRAM-aware packet scheduling scheme in the routers that are one-hop away from the memory controller. Although the proposed scheme expands the memory visibility from the memory controller into its neighbor routers, it is still unable to prevent locality dilution that happens near the source. Moreover, this scheme cannot be easily integrated into

extant QoS-aware designs. In this work, we propose a locality-aware NoC router design to preserve row-buffer locality of memory traffic without degrading the QoS for heterogeneous cores.

6.3 Locality-aware Forwarding

6.3.1 Preserving Row-Buffer Locality

As discussed in the previous section, the memory traffic coming out of a single real-time core usually shows high row-buffer locality. However, the locality is often diluted when different memory streams are merged into one, as they travel through the same links in the NoC. To alleviate the dilution, we prefer requests for the same rows to stay close to each other during transmission. To achieve that, we introduce the *row-index caches* to NoC routers. A row index refers to the address bits that locate the target row in DRAM, including the bits indexing the channel, the rank, the bank and the row of the destination address. As the name suggests, row-index caches record row indices of recently forwarded packets that are going to DRAM. By referring to these caches, allocations can be performed in consideration of row-buffer locality.

Caching of row indices is done separately for each output port to preserve the locality of the memory traffic traveling through the same link. When a packet is requesting for VC/switch allocation, the allocator checks on the cache with the row index of this packet. If there is a cache hit, the allocator will prioritize this packet to shorten its queuing latency so that it can stay in a visible distance from the previous packet going to the same row. If it turns out to be a cache miss, it means either this row index has not appeared before or it has appeared a while ago. In either case, this packet will not be prioritized. This simple approach reorders memory streams to improve row-buffer locality without keeping track of DRAM row-buffer states or timing constraints. In addition, it is oblivious of NoC topology.

A locality-aware policy for packet forwarding in the network is specified as below.

- **Locality-aware RR policy**: If packet A hits the row-index cache and B does not, choose A. Otherwise, perform round-robin arbitration among all requestors.

Since cache implementation techniques have been well studied so far, integrating caches into NoC routers does not require sophisticated architectural modifications. Unlike conventional caches which keep a tag and the data for each cache line, row-index caches only store tags, which further lowers implementation complexity.

6.3.2 Accessing Row-Index Cache

A straightforward way to implement row-index caches is to insert a cache at every output port. Packets requesting for an output port access the associated cache. However, when there are more than one packets requesting for the same output port, multiple accesses will happen to the same cache which cannot be done in a single clock cycle. To avoid cache access conflicts, we adopt per-input row-index caches instead so that cache access can be performed immediately at input once a packet arrives. Inside the cache, each cache set is designated to a distinct output port to allow independent caching per output. These input caches are identical in that cache sets for the same output port remain the same in all caches. Once an allocation is determined by the allocator, all row-index caches will be refilled with the same entry, i.e. the allocatee's row index, in the designated cache set.

When multiple allocation decisions are made with respect to different output ports, these decisions will trigger multiple refills to every input cache. To circumvent frequent cache refills, only VC allocation decisions are broadcast to row-index caches. A cache refill queue is located before every row-index cache to buffer refills from VC allocator that cannot be written into the cache at current cycle. In addition, to allow concurrent cache access and refill, an SRAM with one read and one write ports is used to implement a row-index cache.

6.3.3 Integrating QoS Support

Without regard to the QoS, locality-aware forwarding will easily delay memory streams with low memory locality but high time criticality, leading to worsened system performance. To ensure locality-aware forwarding is not performed at the expense of system performance, it is necessary to integrate QoS support into our locality-aware router.

Previous work on QoS-aware NoC [31, 14, 37, 60] have used priority arbiters to realize QoS support in NoC routers. Specifically, in these approaches packets are categorized in distinct frames or classes which are served based on their priority levels. There have also been work [46, 51] adjusting priority levels of network packets according to requestors' QoS experiences. In [46], the priority of a packet is set to reflect the requestor's actual performance in relative to its target QoS. The resulting priority level is attached to the packet and can be used for priority-based round-robin (specified below) in the memory subsystem.

- **QoS-aware RR policy**: Suppose P_A and P_B are priorities for transactions A and B, if $P_A > P_B$ choose A; if $P_A < P_B$ choose B; otherwise choose between A and B in round-robin manners.

To combine QoS support with locality awareness, we need to introduce row-index cache into extant priority schemes. This can be done by assigning extra credits to the priority levels of requestors that hit row-index caches. An example of such a locality-aware priority scheme is shown below.

- **Locality-aware QoS policy**: Suppose packet A hits the row-index cache and B does not. If $P_A, P_B < \delta$ or $P_A = P_B$, choose A. Otherwise, perform QoS-aware RR.

The parameter δ is an adjustable threshold used to balance locality-aware forwarding and QoS-aware allocation. When the priority level is lower than δ, the allocator focuses on row-buffer locality, otherwise QoS support comes first. A higher δ value gives more

favor to row-buffer locality, but also potentially causes more disturbance to the QoS. Given a priority range of 0 to 7, we found $\delta = 6$ a good setting to improve memory efficiency without causing QoS degradations.

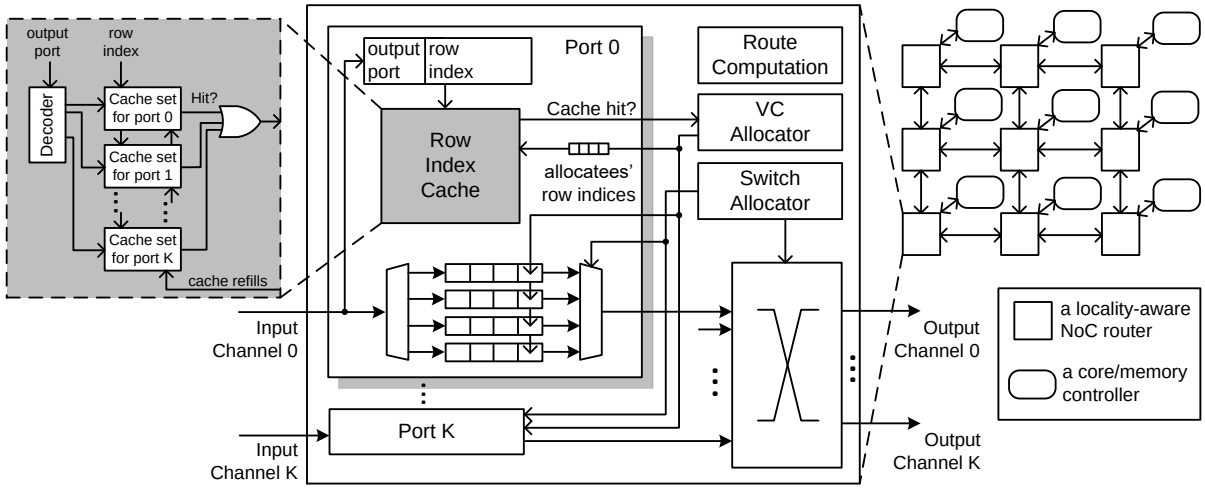


Figure 6.4: Microarchitecture of the locality-aware NoC router with per-input row-index caches.

6.3.4 Router Microarchitecture

The proposed microarchitecture is based on a canonical NoC router design [9]. The pipeline of the canonical NoC router consists of four stages: buffer write (BW)/route computation (RC), VC allocation (VA), switch allocation (SA) and switch traversal (ST). Link traversal (LT) typically takes another cycle as the flit is forwarded to the next router. The router adopts lookahead routing to determine output ports one hop earlier. Fig. 6.4 shows the microarchitecture of the locality-aware NoC router. Compared with the canonical router design, the locality-aware router is equipped with per-input row-index caches to track row-buffer locality in recent packets. It also requires priority arbiters to enable locality-aware VC/switch allocations. The updated pipeline stages will be specified in the rest of this section.

Row-Index Cache Access: Once a head flit arrives at the input buffer, its destination information, including output port index and DRAM row index, is used to access the row-index cache associated with this input port. Cache access by the newly-buffered head flit is concurrent with route computation, which means no extra cycles are introduced to the router pipeline by cache access. Since a row-index cache only performs tag comparisons without data transfers, cache access can be finished in one cycle and the result will be available for VC allocation in the next cycle. When there is no incoming head flit, cache access can still be performed on extant head flits in case the results will be different due to recent cache refills from VC allocator.

Locality-aware Allocation: Cache access result for a packet is encoded into a one-bit flag and collected by the VC/switch allocator, along with the priority level of this packet. Given above information, allocation policies with the awareness of both locality and QoS can be applied at this stage.

Row-Index Cache Refill: After VC Allocation is finished, allocatees' row indices are broadcast to every input cache and buffered in the cache refill queue if they cannot be written back in current clock cycle. In parallel with cache access, cache refill is performed in every cycle as long as the refill queue is nonempty. The classic Least-Recently Used (LRU) replacement policy is applied in row-index caches. In case of overflow at a cache refill queue, the oldest refill will be discarded to be consistent with the LRU policy.

6.3.5 Implementation Cost

Compared with canonical routers, the extra implementation cost of the locality-aware router mainly comes from the storage overhead of row-index caches. In a memory subsystem with 2GB memory and 2KB page size, we need $log(2G/2K) = 20$ bits to encode a row index. Thus a row-index cache stores $20N_{way}N_{port}$ bits in total, in which N_{way} is the number of cache ways per set/output and N_{port} is the number of router ports. Assume the

same amount of storage is implemented in the cache refill queue, the locality-aware router requires an extra storage of $40N_{way}N_{port}$ bits per input port.

6.4 Evaluation

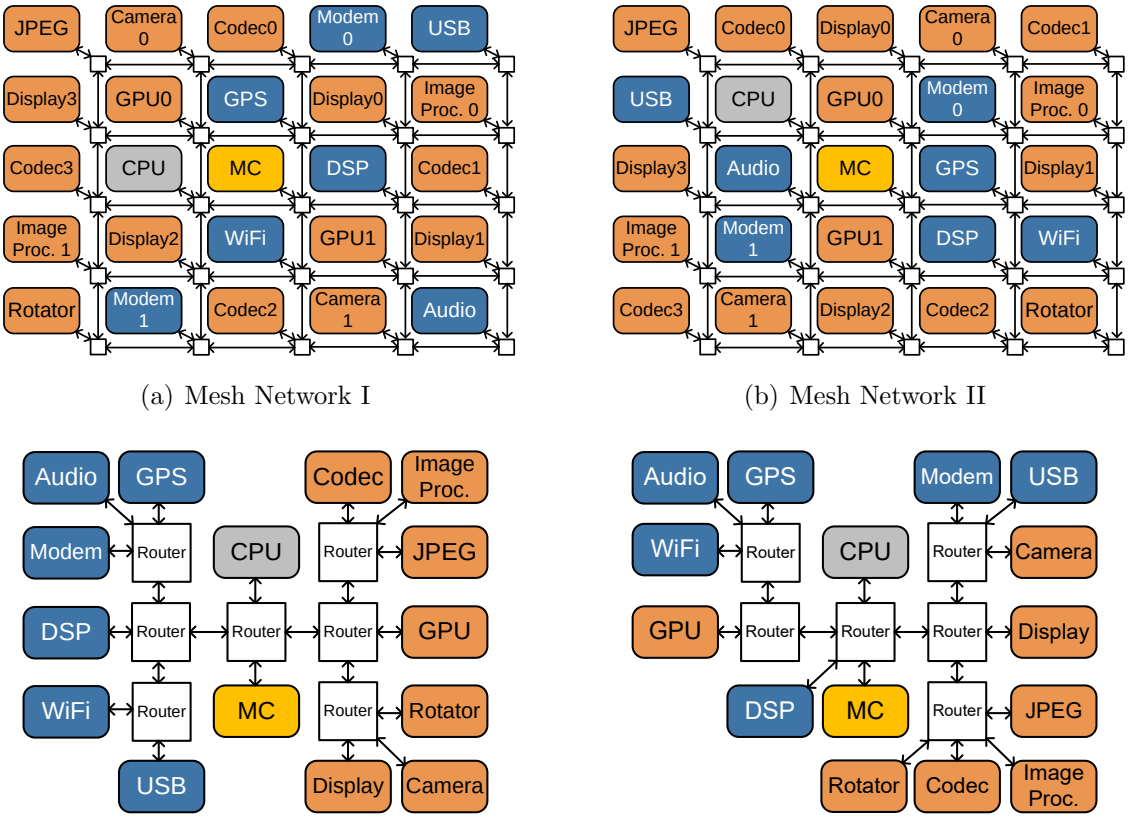
In this section, the proposed NoC design with locality-aware forwarding will first be compared with prior memory-aware routers without integrating QoS support. Performance metrics of memory efficiency, including row-buffer locality and memory latency will be evaluated. Furthermore, the proposed design will be demonstrated in its capability of improving row-buffer locality while providing QoS guarantee. In addition, the energy savings and overhead by the proposed design will be assessed.

Table 6.1: Memory subsystem simulation configurations.

NoC	1GHz clock rate, 5x5 mesh and asymmetric networks, dimension-order routing, 128-bit link width, 4 VCs per port, 5 buffers per VC, separable VC/switch allocator
MC	40 request buffers, FR-FCFS policy, address mapping scheme (from physical address to DRAM address): channel-rank-row-bank-column
DRAM	1866MHz clock rate, 2GB memory, 2KB page size, Channels-Ranks-Banks: 1-2-8, CL-tRCD-tRP(cycles): 36-34-34, tWTR-tRTP-tWR(cycles): 19-14-34, tRRD-tFAW(cycles): 19-75

6.4.1 Methodology

For evaluation, the proposed architecture shown in Fig. 6.4 is implemented in a cycle-accurate NoC simulator. Dimension-order routing and wormhole flow control are applied in the NoC. We assume a memory request, i.e. an NoC packet, contains 64 byte and an NoC flit contains 128 bits. Once injected into the NoC, a packet is converted into four flits to carry data and one head flit encoded with destination information. In NoC



(a) Mesh Network I

(b) Mesh Network II

(c) Asymmetric Network I

(d) Asymmetric Network II

Figure 6.5: Summary of tested NoC topologies, including two 5×5 mesh networks and two asymmetric networks.

routers, we adopt the four-stage pipeline. Each input port of the router contains four virtual channels and a row-index cache. Each virtual channel buffers up to five flits. The row-index cache contains a 4-way cache set for each output port. A memory controller (MC), simulated by DRAMSim2 [52] with LPDDR4 timing model, is integrated into the NoC simulator. A single memory channel with 2GB address space is shared by all heterogeneous cores. Detailed simulation configurations are depicted in Tab. 6.1.

Memory traffic in the heterogeneous system is generated per-core and per-DMA based on a state-of-the-art MPSoC [39]. For the CPU, we emulate four cores sharing a 2MB L2 cache. Each CPU core executes an application from SPEC 2000/2006 benchmarks and

Table 6.2: CPU benchmark descriptions.

case 1	art–mcf–sjeng–sphinx3
case 2	art–mcf–sjeng–hmmer
case 3	mcf–sjeng–sphinx3–hmmer
case 4	art–mcf–sphinx3–hmmer

injects memory traffic into the NoC. Different combinations of CPU applications are tested as listed in Tab. 6.2. For real-time traffic, we model the traffic pattern for each DMA, taking into account of their memory locality and memory access rates. The bandwidth requested by each real-time DMA is calculated assuming a camcorder application is running at 30fps. For example, the frame rotator reads and writes 1080p YUV420 images at 30fps, consuming 89MB/s for read and write requests respectively and 178MB/s in total.

Since most real-time cores do not operate all the time, we emulate real-time traffic in two separate modes including high performance and low power modes. In high performance mode, more real-time cores are active and more traffic is generated than in the other mode. Tab. 6.3 shows the summary of emulated real-time cores with their bandwidth consumptions and their statuses in different emulation modes.

Table 6.3: Summary of the emulated real-time cores, their bandwidth consumptions and their statuses in high performance (HP) and low power (LP) modes.

Real-time cores		Bandwidth	HP	LP
GPU		2GB/s	on	on
DSP		365MB/s	on	on
Multimedia	Video Codec	374MB/s	on	off
	JPEG Decoder	432MB/s	on	off
	Display	3.928GB/s	on	on
	Camera	1.08GB/s	on	on
	Frame Rotator	178MB/s	on	off
	Image Processor	445MB/s	on	off
System	USB	372MB/s	on	on
	GPS	71MB/s	on	on
	Modem	705MB/s	on	off
	WiFi	33MB/s	on	on
	Audio DSP	250MB/s	on	off

Evaluations are conducted upon different network topologies, including mesh and asymmetric networks as shown in Fig. 6.5. For the mesh networks, heterogeneous cores are evenly distributed on a 5×5 mesh with a MC in the center. Some major real-time cores are divided into smaller pieces with equal bandwidth to balance traffic load on the mesh. We test two layouts (Fig. 6.5(a) and Fig. 6.5(b)) of heterogeneous cores on the mesh network. Note that in heterogeneous MPSoCs, DMAs belonging to the same core are usually clustered and sharing the same interface to the NoC. Therefore we also test asymmetric networks with undivided real-time cores. In Fig. 6.5(c) real-time cores are placed based on their functions. In Fig. 6.5(d) heterogeneous cores are positioned in imitation of the layout of a commercial MPSoC [39].

6.4.2 Row-buffer Locality

To begin with, we evaluate the efficacy of our locality-aware router in improving row-buffer locality before considering QoS guarantee. Mesh network I and CPU test case 1 are used in simulation. Real-time cores are set to the high performance mode. The first 10,000 memory requests arriving at the MC are collected. For each request, we record its arrival time and the destination row in DRAM. Then we check whether the destination row has been targeted by previous requests. If so, we compute the arrival time difference between current request and the last request that has targeted at the same row. We use this arrival time difference to measure row-buffer locality, as a smaller arrival time difference indicates higher locality.

Fig. 6.6 shows the distribution of requests over arrival time difference up to 300 cycles. The requests with arrival time difference above 300 cycles and those targeting at rows that have not been accessed before are not included in the figure. In our locality-aware NoC routers, two allocation policies are tested: Locality-aware RR policy prioritizes packets hitting row-index caches, otherwise it performs round-robin arbitrations; Locality-aware

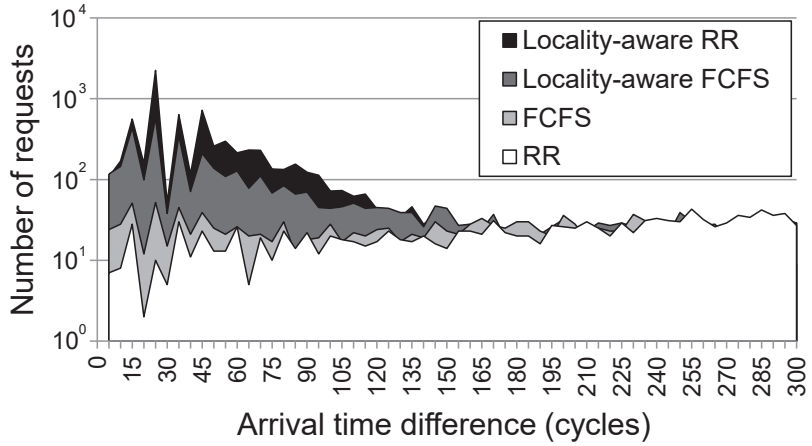


Figure 6.6: Distribution of memory requests over arrival time difference ranging from 1 to 300 cycles.

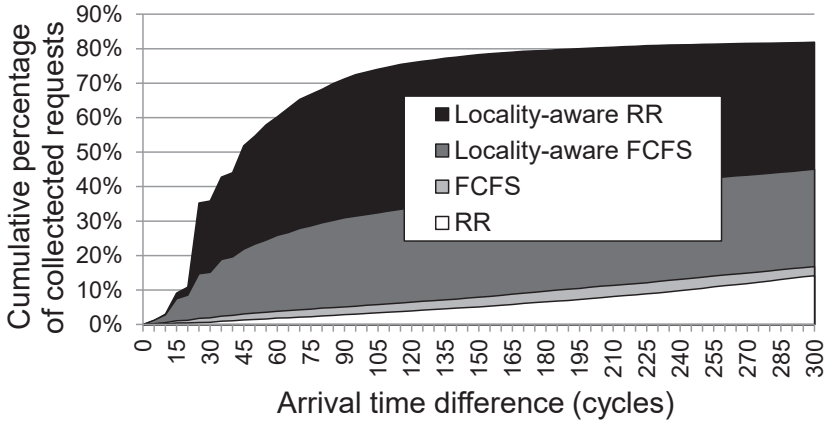


Figure 6.7: Cumulative percentage of 10,000 collected memory requests over arrival time difference from 1 to 300 cycles.

First-Come-First-Serve (FCFS) policy selects the oldest request in terms of waiting time. To prioritize row-index cache hits, Locality-aware FCFS counts 100 extra cycles during waiting time evaluation. Therefore, a row-index cache hit can be chosen if it is less than 100 cycles younger than the oldest pending request. For comparison, conventional RR and FCFS policies without locality awareness are also implemented. As shown in Fig. 6.6, RR policy results in the least number of requests within 100 cycles, meaning RR arbitrations in the NoC cause severe dilution to row-buffer locality. FCFS policy leads to higher locality

than RR, because FCFS preserves the arrival order of NoC packets in each router, which helps to relieve dilution to memory streams with high data rates. In contrast, locality-aware RR and FCFS policies obtain much higher row-buffer locality as many more requests settle within 100 cycles.

Cumulative percentage of collected requests over arrival time difference is shown in Fig. 6.7. RR and FCFS policies result in gradual increase in the percentage. In both cases, less than 20% of collected requests eventually settle within 300 cycles. Locality-aware RR and FCFS have much better results, whereas Locality-aware RR leads to even higher cumulative percentage than Locality-aware FCFS. This is because row-index cache hitting requests have the highest absolute priority under Locality-aware RR policy. As the result, 81.78% of collected requests have arrival time difference below 300 cycles, and 73.25% of requests have the time difference below 100 cycles.

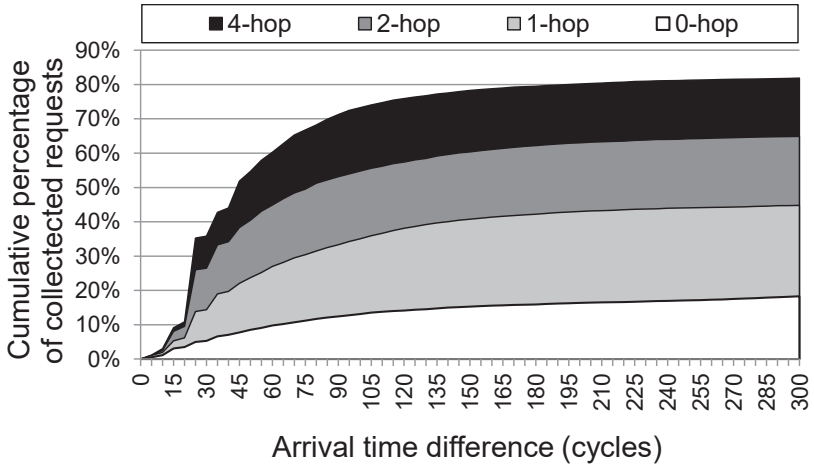


Figure 6.8: Cumulative percentage of collected memory requests over arrival time difference. Different numbers of hops around the MC are applying locality-aware forwarding.

To have a further look at how locality-aware forwarding improves row-buffer locality, we apply Locality-aware RR policy in different numbers of routers around the MC while RR policy is applied in the rest of the network. Fig. 6.8 shows cumulative percentages under different numbers of locality-aware routers. In each case, X-hop means only the routers

within a radius of X hops around the router for the MC perform locality-aware forwarding. In particular, 0-hop means only the router directly connected to the MC applies Locality-aware RR. When $X = 4$, the whole mesh network is applying locality-aware forwarding. As can be seen, when only the router for the MC performs locality-aware forwarding, less than 20% of collected requests arrives at the MC within 300 cycles from previous requests for the same rows. As locality awareness is introduced to more routers around the MC, the cumulative percentage shows an obvious increase.

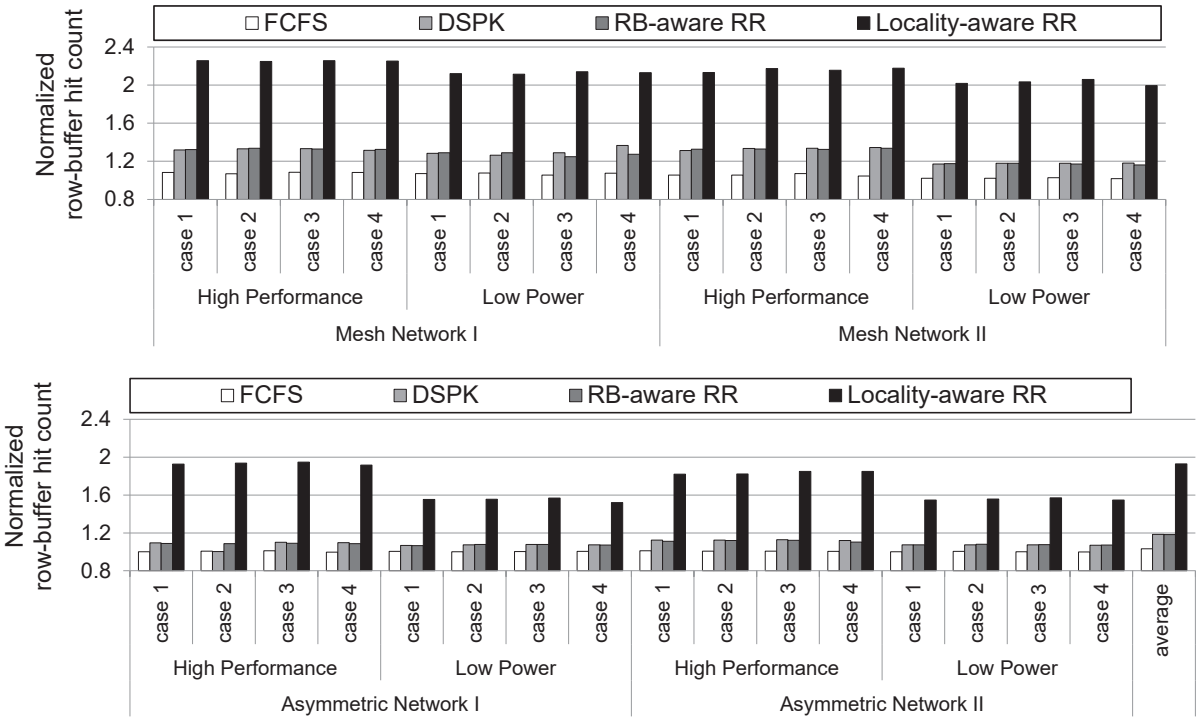


Figure 6.9: Normalized row-buffer hit count per row activation by different allocation policies. All results are normalized by the row-buffer hit count achieved by RR policy. One million DRAM clock cycles are simulated.

A direct result from improving row-buffer locality is the increase of row-buffer hits per row activation. Fig. 6.9 summarizes the average row-buffer hits per activation of all test cases/modes and networks. All results are normalized by the average hits by RR policy. The normalized number indicates the improvement on row-buffer hits over RR

policy. Besides FCFS and locality-aware RR, we also implement two other VC/switch allocation policies for comparison. First, we compare with the Dual Scheduled Packet (DSPK) [38] scheme, which prioritizes requests to idle banks and open rows to improve bank-level parallelism and row-buffer hit rate at the same time. It is implemented in routers within one-hop distance from the MC. In other routers, RR policy is applied. In [38], it was shown that DSPK outperforms previous approaches, such as [18]. Second, we introduce a simplified version of DSPK in which routers (that are located within a one-hop radius of the MC) track row-buffer states and prioritize requests for open rows such that potential row-buffer hits have the highest priority. We refer to this method as row-buffer-aware round-robin (RB-aware RR) policy.

Several observations can be made from Fig. 6.9. First, CPU applications show trivial impacts on row-buffer hits, because most memory traffic comes from real-time cores. Second, under most policies (except FCFS), more improvement on row-buffer hits happens in high performance mode. This is because in this mode more memory requests are generated, as well as the locality dilution. In addition, mesh networks show more potential in row-buffer hit improvement, because memory traffic travels through more routers and suffers from more dilution in meshes.

Among the tested policies in Fig. 6.9, FCFS policy has the lowest normalized hit count which remains around 1 in all cases. That means FCFS and RR policies share similar row-buffer hit rates since both of them are ignorant of memory locality. The normalized hit counts by DSPK and RB-aware RR policies stay close to each other in all cases, because these two policies are similar and only implemented in part of the NoC. Compared with RR policy, DSPK and RB-aware RR policies improve average row-buffer hit count by 18.5% and 18.19% respectively. Instead of tracking row-buffer states, locality-aware RR policy focuses on improving row-buffer locality which is effective regardless of router location and NoC topology. As the result, locality-aware RR has the highest hit count improvement in

every case. On average, row-buffer hit count by Locality-aware RR policy is 93.01% higher than RR and 62.88% higher than DSPK.

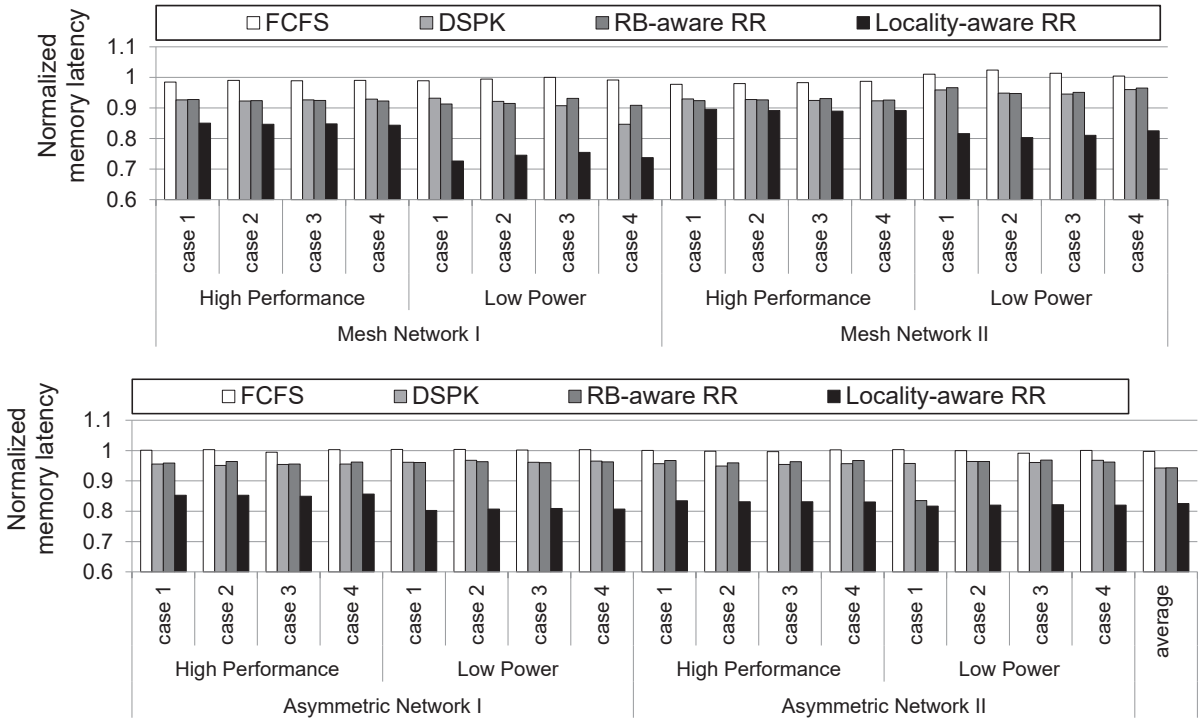


Figure 6.10: Normalized average memory latency by different allocation policies. All results are normalized by the average latency achieved by RR policy. One million DRAM clock cycles are simulated.

Moreover, increasing row-buffer hits helps to reduce memory latency by avoiding the latency penalty caused by unnecessary activation and precharge operations. The summary of normalized average memory latency is shown in Fig. 6.10. Locality-aware RR results in the lowest memory latency on average, which is 17.44% lower than RR and 12.31% lower than DSPK.

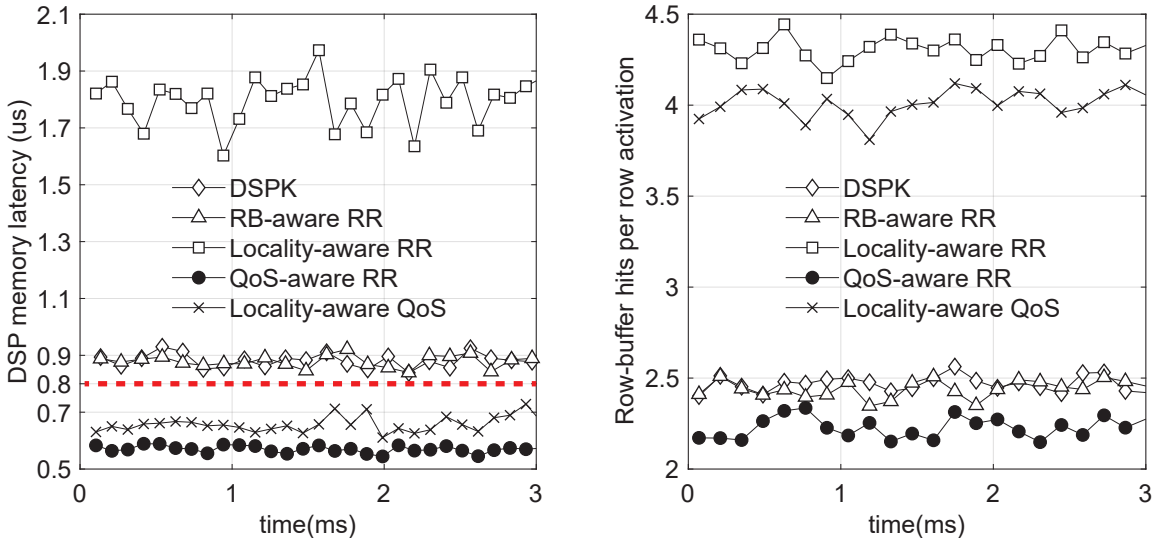
6.4.3 Row-buffer Locality with QoS Support

So far we have demonstrated the efficacy of our locality-aware NoC router in improving memory efficiency. However, in heterogeneous MPSoCs, QoS is a critical metric

for user experience which can be harmed by memory efficiency improvement. For example, the DSP generates random memory access requests, which means it is subject to extra delay when linear memory accesses are prioritized to improve memory efficiency. What's worse, the DSP is latency-sensitive and requires its average memory latency to be limited within a certain threshold. Without regard to the DSP's QoS, i.e. memory latency, improving row-buffer locality could bring more damage than benefits.

As discussed in Section 6.3.3, prior work have proposed frameworks to adjust priority levels of memory requests based on their QoS targets. We have also showed that QoS support can be integrated into locality-aware routers so that memory efficiency is improved without violating QoS guarantees. To evaluate the memory efficiency improvement under QoS support, we implement the Locality-aware QoS policy from Section 6.3.3, as well as the QoS-aware RR policy. In our implementation, QoS priority levels are generated at source as proposed in [46]. The latency sensitivity and the random memory access pattern make the DSP an ideal example for the demonstration of achieving QoS target during locality-aware forwarding. Therefore, we use DSP memory latency as the QoS target to be evaluated. In our evaluation, the average memory latency limit of the DSP is set to $0.8\mu s$.

Fig. 6.11 shows real-time DSP memory latency and row-buffer hits per row activation in DRAM under different allocation policies, including DSPK, RB-aware RR, Locality-aware RR, QoS-aware RR and Locality-aware QoS. Asymmetric network I and CPU test case 4 are used in simulation. Coherent with Fig. 6.9, DSPK and RB-aware RR policies result in similar row-buffer hit counts which are much lower than Locality-aware RR. Since DSP memory requests are de-prioritized during locality-aware forwarding, Locality-aware RR policy also leads to the highest DSP memory latency. Meanwhile, neither DSPK or RB-aware RR policies limit DSP latency within $0.8\mu s$ since they do not provide QoS support. QoS-aware RR achieves the lowest latency for the DSP but also the lowest row-buffer hit count since it does not take into account memory efficiency. At last, Locality-aware



(a) DSP memory latency over time

(b) row-buffer hit count over time

Figure 6.11: DSP memory latency and row-buffer hit count over time. The maximum limit of DSP latency is 0.8 μs. Asymmetric network I and CPU test case 4 are used for simulation. Real-time cores are set to HP mode.

QoS limits DSP latency below the given threshold while obtaining a decent amount of improvement on row-buffer hit count. The average row-buffer hit count is merely 7.09% lower than Locality-aware RR and 80.40% higher than QoS-aware RR.

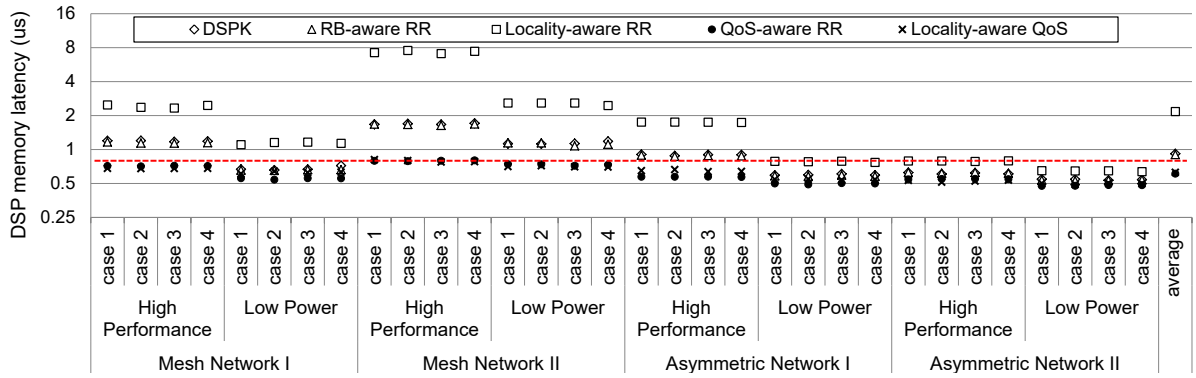


Figure 6.12: Average memory latency of the DSP by different allocation policies. The maximum limit of DSP latency is 0.8 μs.

Fig. 6.12 summarizes the average memory latency of the DSP in all test cases.

Without QoS support, DSPK, RB-aware RR and Locality-aware RR often fail to meet the latency requirement of the DSP. This is especially obvious in mesh networks where memory requests travel through more routers to DRAM. DSP memory latency also tends to be higher when real-time cores are in high performance mode due to the intensified memory competition. Only QoS-aware RR and Locality-aware QoS policies achieve the target QoS for the DSP in all cases. In addition to meeting QoS target, Locality-aware QoS also brings considerable improvement on row-buffer hit rate. Fig. 6.13 shows the row-buffer hit counts by Locality-aware QoS policy normalized by those from QoS-aware RR. All normalized hit counts are above 1 meaning row-buffer hit rate improvement is seen in every test case. On average, row-buffer hit count is improved by 58.32%. In the best case, the improvement is up to 81.88%. As discussed in previous section, row-buffer locality improvement helps to reduce average memory latency for all cores. Fig. 6.14 lists the normalized memory latency by Locality-aware QoS normalized by the results from QoS-aware RR policy. Locality-aware QoS achieves lower latency in all cases, leading to an average latency reduction by 14.45%.

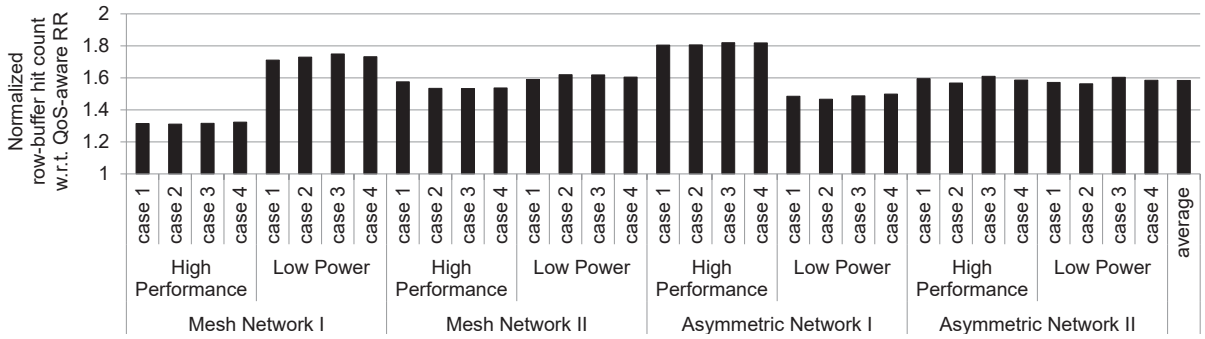


Figure 6.13: Row-buffer hit counts by Locality-aware QoS policy, normalized to the results by QoS-aware RR policy. A normalized hit count higher than 1 indicates improvement in row-buffer locality.

In summary, the locality-aware router is shown to be capable of providing QoS support and improving row-buffer locality at the same time.

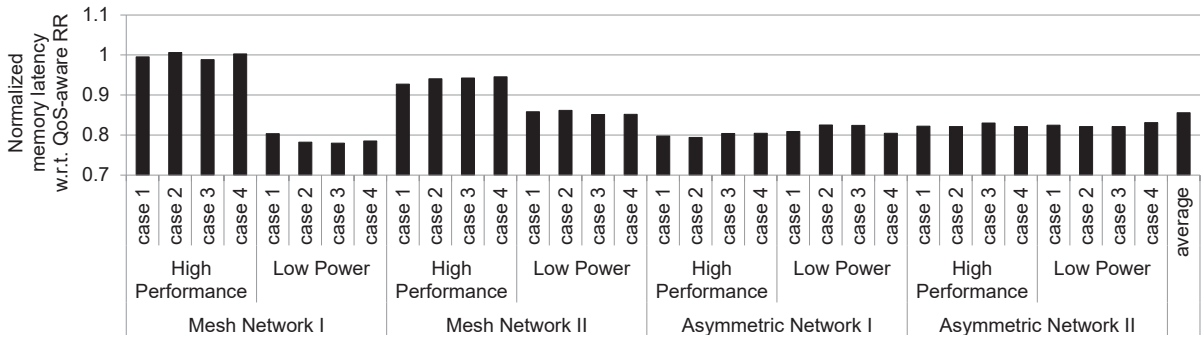
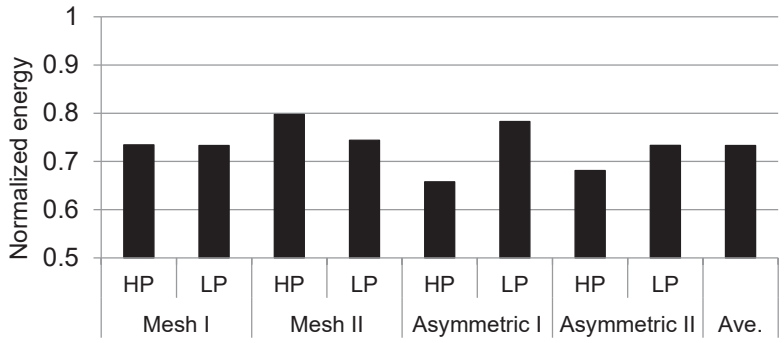


Figure 6.14: Average memory latency by Locality-aware QoS policy, normalized to the results by QoS-aware RR policy. A normalized latency lower than 1 indicates reduction in memory latency.

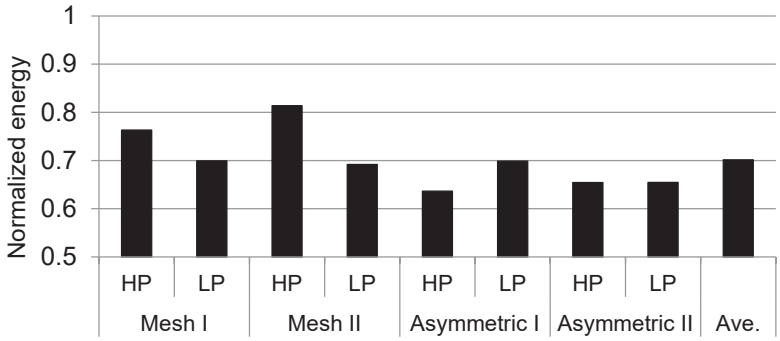
6.4.4 Energy Saving and Overhead

The energy consumed by DRAM can be split into two parts: operation energy and background energy [11]. Operation energy refers to the energy cost related to the execution of DRAM commands, including activation, precharge and column read/write. Whereas background energy is consumed at a constant rate regardless of DRAM operations, such as the energy cost by phase-locked loop. With more row-buffer hits, fewer activation and precharge operations are needed to finish the same amount of memory accesses, which means less operation energy cost. Furthermore, since memory latency is reduced as a result of increased row-buffer hits, it takes shorter time to serve the same number of memory requests in DRAM, which leads to lower background energy.

Fig. 6.15 summarizes the DRAM energy by Locality-aware QoS policy. The energy costs are normalized to those by QoS-aware RR policy to show the energy reduction introduced by locality-aware forwarding. CPU test case 1 is used for all test cases. We adopt an open source tool named DRAMPower [6] for command trace based DRAM energy estimation. The memory subsystem is simulated until one million bytes have been accessed in DRAM. The energy cost during this period is shown, including operation energy (Fig. 6.15(a)) and background energy (Fig. 6.15(b)). Compared with QoS-aware RR policy,



(a) Normalized operation energy



(b) Normalized background energy

Figure 6.15: Summary of normalized DRAM energy cost (operation energy and background energy) by different allocation policies after one million bytes have been accessed in DRAM. All results are normalized by the energy cost achieved by QoS-aware RR policy.

Locality-aware QoS lowers operation energy by 26.70% and background energy by 29.84% on average, respectively.

The proposed locality-aware router design achieves substantial DRAM energy savings by creating more row-buffer hits and reducing the energy required to serve the same number of memory requests. However, the locality-aware design also adds extra energy cost to router microarchitecture. To estimate the energy overhead in NoC routers, we use DSENT [49] with a 45nm technology node library. We extend the power model in DSENT to include row-index caches. Compared with a canonical NoC router, the locality-aware design increases area and power by 11.54% and 12.10% respectively. Nonetheless, given that DRAM energy consumption is far higher than the energy consumption of an NoC

router, we still see a substantial net improvement in energy savings. In Fig. 6.16, we show the combined energy cost in DRAM and NoC routers when locality-aware RR policy is applied in our locality-aware NoC routers. The results are normalized by the combined energy cost when the QoS-aware RR policy is applied in canonical NoC routers. Although locality-aware NoC routers incur some energy overhead, the DRAM energy savings are substantially more. Therefore, we still see a substantial net improvement in energy savings when considering both DRAM and NoC energy. As shown in Fig. 6.16, the combined energy cost by locality-aware RR still shows an average net reduction by 27.82%.

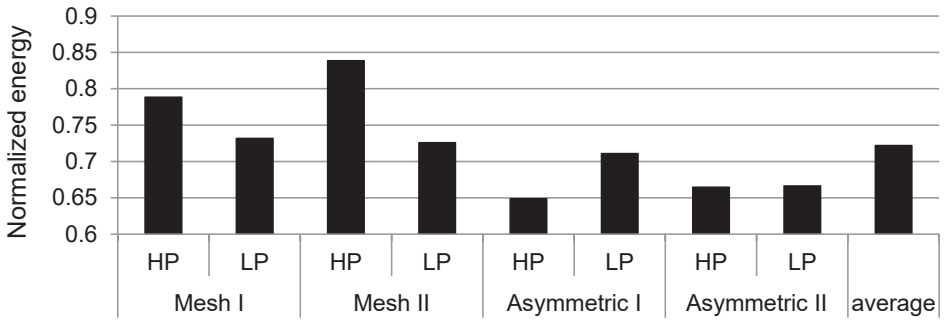


Figure 6.16: Combined energy cost (in DRAM and NoC routers) by Locality-aware QoS policy in locality-aware routers after one million bytes have been accessed in DRAM. All results are normalized by the combined energy cost by QoS-aware RR policy applied in canonical routers.

6.5 Related Work

QoS-aware Networks-on-Chip: Globally Synchronized Frames (GSF) [31] is one of the earliest approaches that adopts a frame-based scheduling scheme to provide throughput guarantees to network traffic. In GSF, time is coarsely quantized into frames and each flow can reserve a fraction in the frame to inject data flits. Frames are prioritized according to their ages and the flits belonging to older frames are chosen over those in younger frames. The limitation of GSF is two-fold: the hardware complexity due to the

recycling of frames and source buffering, and the coarse granularity of global scheduling. To overcome the limitations of GSF, LOFT scheme [37] localizes the frame-based scheduling. In LOFT, each output link of a router manages frame recycles its own frames independently, which enables more flexibility and less hardware cost. Preemptive Virtual Clock (PVC) [14] takes a different approach to circumvent the issues in GSF. In PVC, frames are defined as a fixed number of cycles so that no frame recycling is needed. Moreover, each router maintains a table of bandwidth counters for network traffic flows. Rate-based scheduling is performed at each router in reference to the bandwidth counters. The OSCAR scheme [60] was recently proposed for CPU-GPU systems. To accommodate the vastly different traffic from the CPU and the GPU, the authors deployed asynchronous batch scheduling. Every router bundles packets into mini batches, inside which specific provisions are allocated to the CPU and the GPU. Packets from older batches are scheduled ahead of younger batches. Inside a batch, packets are reordered so that higher priority is given to CPU requests and read requests.

Memory-aware Networks-on-Chip: Studies on memory-aware NoC designs are relatively limited. A DRAM-aware flow control protocol was proposed in [18], assuming the memory controller does not posses scheduling capability or request buffers. In the proposed protocol, VC allocators compare the destinations of buffered packets with the last scheduled packet. The priority level of a packet is evaluated based on the hypothetic memory scheduling interval after the previous packet. The packet that goes to the same row with its predecessor needs the least scheduling time interval and thus achieves the highest priority. This way, packets coming from the same source and to the same row in DRAM naturally form into batches along the route. The authors in [38] introduced the Dual Scheduled Packet (DSPK) framework which includes two types of routers. The first type of router (DSPK-I) deals with core-to-core communication traffic, and prioritizes latency-sensitive packets according to their L1 cache miss rates and memory-level parallelism. The

second type of router (DSPK-II) is particularly designed for the traffic to off-chip memory. A DSPK-II router collects bank state information and prioritizes memory packets that are beneficial to DRAM efficiency. Since a DSPK-II router requires real-time information on bank states, it has to be placed near the memory controller. In the proposed framework, DSPK-II routers are used within a one-hop radius of the memory controller.

Memory Visibility Extension: Besides NoC designs, there have also been many research about increasing the visibility for the memory scheduler in order to improve DRAM efficiency. As the last stage for memory requests before the memory controller, the last-level cache is seen as the opportunity for visibility extension. The Virtual Write Queue (VWQ) [48] was designed to coordinate memory scheduling with cache writebacks. As an extension to the write queue in the memory controller, the LRU cache lines in the last-level cache are visible to the memory controller and available for memory scheduling along with write requests. Writebacks are triggered when the target row of these dirty LRU cache lines is being accessed so that these scheduled writebacks will not cause extra row activations in DRAM. The Last-Write Predictor Guided (LWPG) writeback policy was proposed in [54]. The authors defined last-write blocks as cache blocks that will not be written again before being evicted to memory. A low overhead last-write predictor keeps track of last-write blocks which are available for memory scheduling without incurring extra memory write requests. Eager Read/Write Clustering (ERWC) [21] further reduces the number of row activations by clustering both write and read requests with cache writebacks to the same row-buffer. Eager writebacks are triggered by row activations regardless of the cause. Lately a unified memory controller [45] was proposed to orchestrate cache request arbiter with memory scheduler so that potential row-buffer hits can be harvested during cache request arbitration.

6.6 Chapter Summary

In this chapter, we show the deficiency of heterogeneous MPSoCs in memory efficiency due to memory locality dilution. Previous work have studied memory visibility extension, while the preservation of memory locality in the NoC remains to be explored. In response to the challenge, we proposed a NoC router design with locality-aware packet forwarding. Row-index caches are introduced to track memory locality in recent packets. Locality-aware forwarding can be implemented on basis of priority arbiters. It is compatible with extant QoS-aware router designs. In our evaluations, the proposed design is proven to be more effective in improving memory efficiency compared with prior memory-aware approaches, while providing QoS support. Compared with QoS-aware routing without special care to row-buffer locality, the proposed design greatly improves row-buffer hits by 58.32% and reduces memory latency by 14.45% on average. Considering both energy overhead and savings, the locality-aware router brings about an average net reduction in energy cost by 27.82%.

6.7 Acknowledgement

This chapter, in part, is currently being prepared for submission for publication of the material. Song, Yang; Lin, Bill. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Conclusion

In the age of dark silicon, heterogeneous MPSoCs are widely deployed, with an increasingly diverse collection of specialized real-time accelerators, due to their advanced energy efficiency. Unlike traditional multicore processors, heterogeneous MPSoCs conduct data sharing through the main memory, a process which involves close participation from all fabrics in the memory subsystem. Consequently, the memory subsystem has more influence on system performance than in prior platforms.

As memory requests traverse through the memory subsystem, they are in constant competition for shared resources such as on-chip buffers and DRAM. The distinct memory access patterns and performance targets of these requests makes memory interference even worse. These conditions make it necessary to introduce QoS awareness to the memory subsystem to ensure heterogeneous cores achieve performance targets under the interference. Moreover, due to the inherent limitations in DRAM architecture, the memory subsystem needs to pay special attention to memory efficiency in order to extract the maximum possible bandwidth from DRAM. As a result, memory subsystem design for heterogeneous MPSoCs is a two-dimensional design problem with frequently conflicting objectives, namely end-to-end QoS and memory efficiency.

In this dissertation, we have examined memory subsystem design from four different aspects. First, in Chapter 3 we investigated memory management solutions and proposed the self-aware resource allocation framework that applies priority-based adaptation to translate various notions of QoS into relative priority levels, thereby enabling the distribution of performance monitoring and memory allocation. Second, in Chapter 4 we studied the limitations of two-tier memory controllers in QoS-aware scheduling and proposed the single-tier virtual queuing memory controller that implements efficacious memory scheduling with high scalability. Third, in Chapter 5 we explored the extension of memory visibility and proposed a unified controller that orchestrates arbitrations at the last-level cache and DRAM to improve memory efficiency. Finally, in Chapter 6 we analyzed the locality dilution in the NoC and proposed a locality-aware network router that forwards packets while providing locality preservation as well as QoS support.

Bibliography

[1] Gallium3D. http://en.wikipedia.org/wiki/Gallium3D/.

[2] Arnau, J.-M., Parcerisa, J.-M., and Xekalakis, P. Parallel frame rendering: Trading responsiveness for energy on a mobile gpu. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (2013), PACT '13, IEEE, pp. 83–92.

[3] Arnau, J.-M., Parcerisa, J.-M., and Xekalakis, P. Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (2013), ICS '13, ACM, pp. 37–46.

[4] Ausavarungnirun, R., Chang, K. K.-W., Subramanian, L., Loh, G. H., and Mutlu, O. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2012), ISCA '12, IEEE Computer Society, pp. 416–427.

[5] Becker, D. U., and Dally, W. J. Allocator implementations for network-on-chip routers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), SC '09, ACM, pp. 52:1–52:12.

[6] Chandrasekar, K., Weis, C., Li, Y., Goossens, S., Jung, M., Naji, O., Akesson, B., Wehn, N., and Goossens, K. DRAMPower: Open-source DRAM power and energy estimation tool, 2014.

[7] Chen, X., Chang, L.-W., Rodrigues, C. I., Lv, J., Wang, Z., and Hwu, W.-M. Adaptive cache management for energy-efficient gpu computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO '14, IEEE Computer Society, pp. 343–355.

[8] Colmenares, J. A., Eads, G., Hofmeyr, S., Bird, S., Moretó, M., Chou, D., Gluzman, B., Roman, E., Bartolini, D. B., Mor, N., Asanović, K., and Kubiatowicz, J. D. Tessellation: Refactoring the os around explicit resource

containers with continuous adaptation. In *Proceedings of the 50th Annual Design Automation Conference* (New York, NY, USA, 2013), DAC '13, ACM, pp. 76:1–76:10.

[9] Dally, W., and Towles, B. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[10] Dally, W. J., and Seitz, C. L. The torus routing chip. *Distributed Computing 1*, 4 (Dec 1986), 187–196.

[11] David, H., Fallin, C., Gorbatov, E., Hanebutte, U. R., and Mutlu, O. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing* (New York, NY, USA, 2011), ICAC '11, ACM, pp. 31–40.

[12] Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (Oct 1974), 256–268.

[13] GALLES, M. Scalable pipelined interconnect for distributed end-point routing : The sgi spider chip. *Proceedings of the International Symposium on High-Performance Interconnects* (1996), 141–146.

[14] Grot, B., Keckler, S. W., and Mutlu, O. Preemptive virtual clock: A flexible, efficient, and cost-effective qos scheme for networks-on-chip. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO '09, ACM, pp. 268–279.

[15] Hoffmann, H., Holt, J., Kurian, G., Lau, E., Maggio, M., Miller, J. E., Neuman, S. M., Sinangil, M., Sinangil, Y., Agarwal, A., Chandrakasan, A. P., and Devadas, S. Self-aware computing in the angstrom processor. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 259–264.

[16] Jacob, B., Ng, S., and Wang, D. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[17] Jalle, J., Quiñones, E., Abella, J., Fossati, L., Zulianello, M., and Cazorla, F. J. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *Proceedings of the Real-Time Systems Symposium* (Dec 2014), RTSS '14, pp. 207–217.

[18] Jang, W., and Pan, D. Z. An SDRAM-aware router for networks-on-chip. In *Proceedings of the 46th Annual Design Automation Conference* (New York, NY, USA, 2009), DAC '09, ACM, pp. 800–805.

[19] Jantsch, A., Dutt, N., and Rahmani, A. M. Self-awareness in systems on chip - a survey. *IEEE Design Test 34*, 6 (Dec 2017), 8–26.

[20] JEDEC. Low Power Double Data Rate 4 (LPDDR4).
https://www.jedec.org/standards-documents/docs/jesd209-4b, 2017.

[21] Jeon, M., Li, C., Cox, A. L., and Rixner, S. Reducing DRAM row activations with eager read/write clustering. *ACM Trans. Archit. Code Optim. 10*, 4 (Dec. 2013), 43:1–43:25.

[22] Jeong, M. K., Erez, M., Sudanthi, C., and Paver, N. A QoS-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 850–855.

[23] Jerger, N. E., Krishna, T., Peh, L.-S., and Martonosi, M. *On-Chip Networks: Second Edition*. Morgan & Claypool, 2017.

[24] Jia, W., Shaw, K. A., and Martonosi, M. MRPB: Memory request prioritization for massively parallel processors. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture* (Feb 2014), HPCA '14, pp. 272–283.

[25] Kaseridis, D., Stuecheli, J., and John, L. K. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), MICRO '44, ACM, pp. 24–35.

[26] Kim, S., Kim, S., and Lee, Y. Dram power-aware rank scheduling. In *Proceedings of the International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2012), ISLPED '12, ACM, pp. 397–402.

[27] Kim, Y., Han, D., Mutlu, O., and Harchol-Balter, M. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture* (Jan 2010), HPCA '10, pp. 1–12.

[28] Kim, Y., Papamichael, M., Mutlu, O., and Harchol-Balter, M. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '10, IEEE Computer Society, pp. 65–76.

[29] Lee, H.-H. S., Tyson, G. S., and Farrens, M. K. Eager writeback - a technique for improving bandwidth utilization. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture* (New York, NY, USA, 2000), MICRO '00, ACM, pp. 11–21.

[30] Lee, J., and Kim, H. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2012), HPCA '12, IEEE Computer Society, pp. 1–12.

[31] Lee, J. W., Ng, M. C., and Asanovic, K. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2008), ISCA '08, IEEE Computer Society, pp. 89–100.

[32] Li, A., van den Braak, G.-J., Kumar, A., and Corporaal, H. Adaptive and transparent cache bypassing for gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 17:1–17:12.

[33] Mekkat, V., Holey, A., Yew, P.-C., and Zhai, A. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (Piscataway, NJ, USA, 2013), PACT '13, IEEE Press, pp. 225–234.

[34] Mutlu, O., and Moscibroda, T. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (2008), ISCA '08, IEEE Computer Society, pp. 63–74.

[35] NVIDIA. Tegra x1.

http://www.nvidia.com/object/tegra-x1-processor.html, 2015.

[36] Nvidia. Tegra.

https://www.nvidia.com/object/tegra.html, 2019.

[37] Ouyang, J., and Xie, Y. Loft: A high performance network-on-chip providing quality-of-service support. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Dec 2010), MICRO '10, pp. 409–420.

[38] Pimpalkhute, T., and Pasricha, S. Noc scheduling for improved application-aware and memory-aware transfers in multi-core systems. In *Proceedings of the 27th International Conference on VLSI Design and the 13th International Conference on Embedded Systems* (Jan 2014), pp. 234–239.

[39] Qualcomm. Snapdragon 845.

https://www.qualcomm.com/products/snapdragon-845-mobile-platform, 2017.

[40] Qualcomm. Snapdragon.

https://www.qualcomm.com/snapdragon, 2019.

[41] Rahmani, A. M., Donyanavard, B., Mück, T., Moazzemi, K., Jantsch, A., Mutlu, O., and Dutt, N. SPECTR: Formal supervisory control and coordination for many-core systems resource management. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 169–183.

[42] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2000), ISCA '00, ACM, pp. 128–138.

[43] Sarma, S., Dutt, N., Gupta, P., Venkatasubramanian, N., and Nicolau, A. Cyberphysical-system-on-chip (CPSoC): A self-aware mpsoc paradigm with cross-layer virtual sensing and actuation. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition* (March 2015), DATE '15, ACM, pp. 625–628.

[44] Sharifi, A., Srikantaiah, S., Mishra, A. K., Kandemir, M., and Das, C. R. METE: Meeting end-to-end qos in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev. 39*, 1 (June 2011), 13–24.

[45] Song, Y., Alavoine, O., and Lin, B. Row-buffer hit harvesting in orchestrated last-level cache and dram scheduling for heterogeneous multicore systems. In *Proceedings of the Design, Automation Test in Europe Conference* (March 2018), DATE '18, pp. 779–784.

[46] Song, Y., Alavoine, O., and Lin, B. SARA: Self-aware resource allocation for heterogeneous mpsocs. In *Proceedings of the 55th Annual Design Automation Conference* (New York, NY, USA, 2018), DAC '18, ACM, pp. 113:1–113:6.

[47] Song, Y., Samadi, K., and Lin, B. Single-tier virtual queuing: An efficacious memory controller architecture for mpsocs with multiple realtime cores. In *Proceedings of the 53rd Annual Design Automation Conference* (New York, NY, USA, 2016), DAC '16, ACM, pp. 6:1–6:6.

[48] Stuecheli, J., Kaseridis, D., Daly, D., Hunter, H. C., and John, L. K. The virtual write queue: Coordinating dram and last-level cache policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 72–82.

[49] Sun, C., Chen, C.-H. O., Kurian, G., Wei, L., Miller, J., Agarwal, A., Peh, L.-S., and Stojanovic, V. DSENT - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of the 2012 IEEE/ACM 6th International Symposium on Networks-on-Chip* (Washington, DC, USA, 2012), NOCS '12, IEEE Computer Society, pp. 201–210.

[50] Taylor, M. B. Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 1131–1136.

[51] Usui, H., Subramanian, L., Chang, K. K.-W., and Mutlu, O. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Trans. Archit. Code Optim. 12*, 4 (Jan. 2016), 65:1–65:28.

[52] Wang, D., Ganesh, B., Tuaycharoen, N., Baynes, K., Jaleel, A., and Jacob, B. DRAMsim: A memory system simulator. vol. 33, ACM, pp. 100–107.

[53] Wang, P.-H., Li, C.-H., and Yang, C.-L. Latency sensitivity-based cache partitioning for heterogeneous multi-core architecture. In *Proceedings of the 53rd Annual Design Automation Conference* (New York, NY, USA, 2016), DAC '16, ACM, pp. 5:1–5:6.

[54] Wang, Z., Khan, S. M., and Jiménez, D. A. Improving writeback efficiency with decoupled last-write prediction. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2012), ISCA '12, IEEE Computer Society, pp. 309–320.

[55] WikiChip. Kirin 970 - HiSilicon.

https://en.wikichip.org/wiki/hisilicon/kirin/970, 2017.

[56] WikiChip. A12 Bionic - Apple.

https://en.wikichip.org/wiki/apple/ax/a12, 2018.

[57] Wikipedia. Apple-designed processors.

https://en.wikipedia.org/wiki/Apple-designed_processors, 2019.

[58] Xie, X., Liang, Y., Sun, G., and Chen, D. An efficient compiler framework for cache bypassing on gpus. In *Proceedings of the International Conference on Computer-Aided Design* (Piscataway, NJ, USA, 2013), ICCAD '13, IEEE Press, pp. 516–523.

[59] Xie, X., Liang, Y., Wang, Y., Sun, G., and Wang, T. Coordinated static and dynamic cache bypassing for gpus. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture* (Feb 2015), HPCA '15, pp. 76–88.

[60] Zhan, J., Kayiran, O., Loh, G. H., Das, C. R., and Xie, Y. Oscar: Orchestrating stt-ram cache traffic for heterogeneous cpu-gpu architectures. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Oct 2016), MICRO '16, pp. 1–13.

[61] Zhang, T., Xu, C., Chen, K., Sun, G., and Xie, Y. 3D-SWIFT: A high-performance 3D-stacked wide IO DRAM. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI* (New York, NY, USA, 2014), GLSVLSI '14, ACM, pp. 51–56.