THESIS

HETEROGENEOUS PRIORITIZATION FOR NETWORK-ON-CHIP BASED MULTI-CORE SYSTEMS

Submitted by

Tejasi Pimpalkhute

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2013

Master's Committee:

Advisor: Sudeep Pasricha

Wim Bohm
Anura P. Jayasumana

ABSTRACT

HETEROGENEOUS PRIORITIZATION FOR NETWORK-ON-CHIP BASED MULTI-CORE SYSTEMS

In chip multi-processor (CMP) systems, communication and memory access both play an important role in influencing the performance achievable by the system. The manner in which the network packets (on-chip cache requests/responses) and off-chip memory bound packets are handled, in multi-core environment with several applications executing in parallel, determines end-to-end latencies across the network and memory. Several techniques have been proposed in the past that schedule packets in either an application-aware manner or memory requests in a DRAM row/bank locality-aware manner. Prioritization of memory requests is a major factor in increasing the overall system throughput. Moreover, with the increasing diversity in CMP systems, applying the same prioritization rules to all packets traversing the NoC as is done in the current implementations may no longer be a viable approach.

In this thesis, a holistic framework is proposed that integrates novel prioritization techniques for both network and memory accesses and operates cohesively in an application-aware and memory-aware manner to optimize overall system performance. The application-aware technique makes fine grain classification of applications with a newly proposed ranking scheme. Two novel memory-prioritization algorithms are also proposed, one of which is specifically tuned for high-speed memories. Upon analyzing the fairness issues that arise in a multi-core environment, a novel strategy is proposed and employed system-wide to ensure

fairness in the system. The proposed heterogeneous prioritization framework is validated using a detailed cycle-accurate full system event-driven simulator and shows significant improvement over Round Robin and other recently proposed network and memory prioritization techniques.

ACKNOWLEDGEMENTS

I take this opportunity to thank all the wonderful people who contributed in their own special ways and supported me to attain the completion of this thesis work.

First and foremost, I express heartfelt gratitude towards my advisor, Dr. Sudeep Pasricha who agreed to make me a part of his research group and mentor me for this thesis. It was only his guidance, encouragement and motivation that kept me progressing through this thesis. I had a very little information about Network-on-chip (NoC) and the related research area when I first arrived at CSU for my Master's program. After meeting Dr. Pasricha for the first time, I was really inspired by his work in NoCs and decided to pursue this area. He has been very patient and approachable as well as an amazing guide inspite of being extremely busy. I owe my thesis as well as my learning experience to him.

I would also like to sincerely thank all my colleagues in Multi-core Embedded Computing Systems (MECS) lab- Shirish Bahirat, Nishit Kapadia, Yong Zou, Yi Xiang, Srinivas Desai who provided their helpful feedback from time to time and were instrumental in my learning process. In my initial days in the research group, Nishit, Yong and Yi helped me to get the hang of network-on-chip concepts and the simulation tools.

I am extremely grateful to my committee members- Dr. Anura Jayasumana and Dr. Wim Bohm for agreeing to be on my thesis committee. I appreciate them taking time out of their busy schedule to review my thesis and provide valuable inputs for improvement in this work.

Last, but not the least, I would like to thank my family back in India, who supported my decision of coming to the US for Master's. My deepest gratitude to my mother Savita Pimpalkhute, who has always stood by me and encouraged me for pursuing my dreams. She has been a pillar of strength to me. I would like to dedicate this thesis to my mother. My loving

brother, Omkar Pimpalkhute and my dearest friends, who are also my second family in Colorado, Ketki Kolte, Hrushikesh Kulkarni, Manish Mohanpurkar and Kaustubh Gadkari have been great unconditional support in this process. I thank from the bottom of my heart all these amazing people for their love and support.

DEDICATION

To my mother, Savita

And

Dearest friends and family

Without their incessant support and understanding,

This would have been only a dream for me

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Today's computing chip platforms are becoming increasingly complex with highly parallel architectures and tens to hundreds of components crammed together on a single die. With the number of cores on a chip growing due to the benefits of shrinking nanometer technology, a lot of pressure is put on the interconnection network fabric to handle communication flows from various multi-programmed (i.e., multiple co-running application based) workloads efficiently. In addition, due to the well publicized "memory wall" between processors and main memory, new techniques are needed to overcome mismatches between processor and memory performance. This chapter intends to provide an overview of current challenges faced by today's multi-core systems and the motivation for this research.

1.1 Overview of Multi-core Systems

As Moore's law continues to hold true, we see increasingly denser chips in modern semiconductor technology. Performance gains with single-core computing are limited. Thus, to increase the performance many-fold, parallelism in hardware along with software needs to be exploited. Hence, single-core computing is rapidly being replaced with multi-core computing in today's chip architectures. A modern chip multi-processor (CMP) consists of many cores connected to each other, executing various workloads in parallel.

With the advent of multi-core computing and shrinking nanometer technology, interconnection technology dons a role more significant than ever before. Interconnects are responsible for communication between various processing elements on a chip. As process

technology scales down, gate delays are reducing significantly butwiring delays are not. Hence, there is an inherent limitation to the performance gain achievable by advancing nanometer technology alone. Emerging CMP designs require an interconnection fabric that is modular and scalable.

Several CMPs have been commercially introduced recently that consist of multiple processing elements on a single chip. A few of the more prominent CMP designs are discussed below.

AMD Opteron

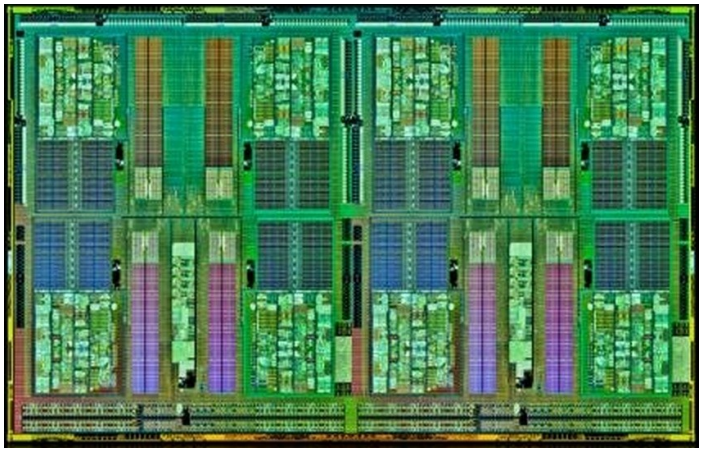


Figure 1.1: AMD Opteron 6200 series processor [33]

Figure 1.1 shows the AMD Opteron 6200 series multi-core system. It consists of 16 cores in the multi-core system and uses the HyperTransport 3.0 interconnect technology. It can operate on clock rates upto 3.3 GHz. This system supports a shared L3 cache and low voltage DDR3 main memories to reduce overall power consumption.

Intel's Single-Chip Cloud Computer (SCC)

Figure 1.2 shows Intel's Single-Chip Cloud Computer (SCC) CMP. It consists of 48 cores on a single chip. The interconnection fabric used to connect all cores to each other and to main memory is a Network-on-chip (NoC) that will be discussed in the following section. The name of the chip is derived from the fact that this chip provides a cluster of compute nodes similar to the "cloud" of computing resources over the internet. Figure 1.3 shows an internal block diagram of SCC. This particular chip has 24 tiles connected together in a 2D mesh NoC topology with two cores per tile. Each core has a private L1 and L2 cache. It has four integrated memory controllers that handle the traffic for off-chip DDR3 memory.

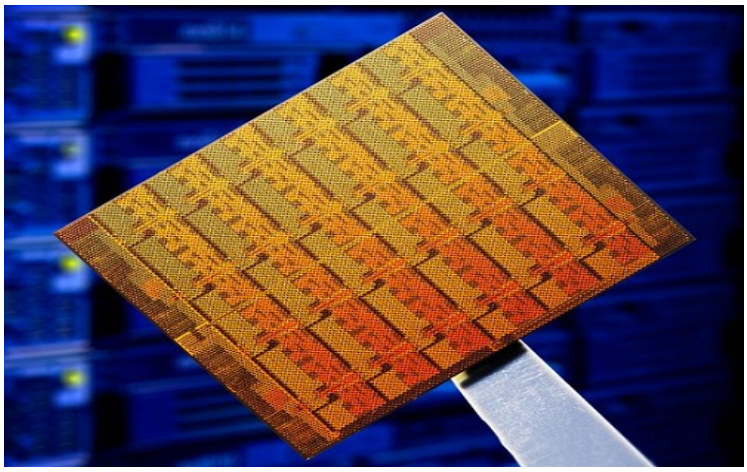


Figure 1.2: Intel's SCC [27]

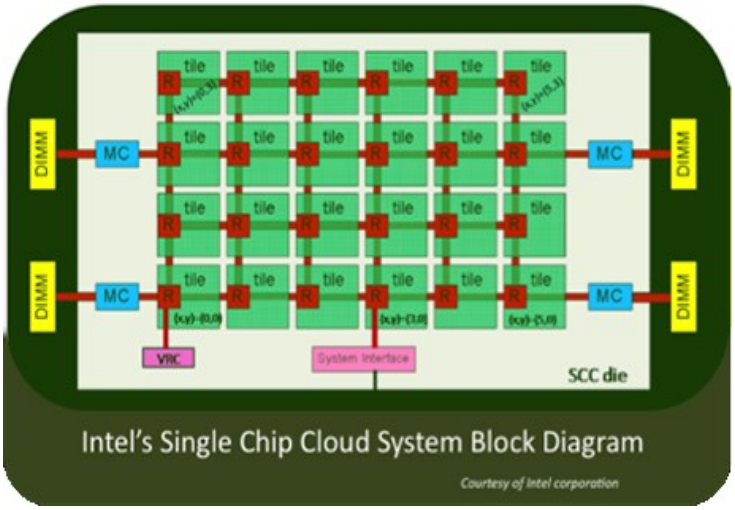


Figure 1.3: Block diagram of Intel's SCC [27]

TILE64 Multi-processor

Figure 1.4 shows the schematic diagram of the Tile64 processor from Tilera. This multi-core system is comprised of 64 cores interconnected in a 2D NoC mesh topology. Each tile represents a general pupose core with L1 and L2 caches as well as a switch to route packets. The cores can operate upto a frequency of 900 MHz. The on-chip processing elements communicate with the off-chip DDR2 memory through four memory controllers connected to the network.

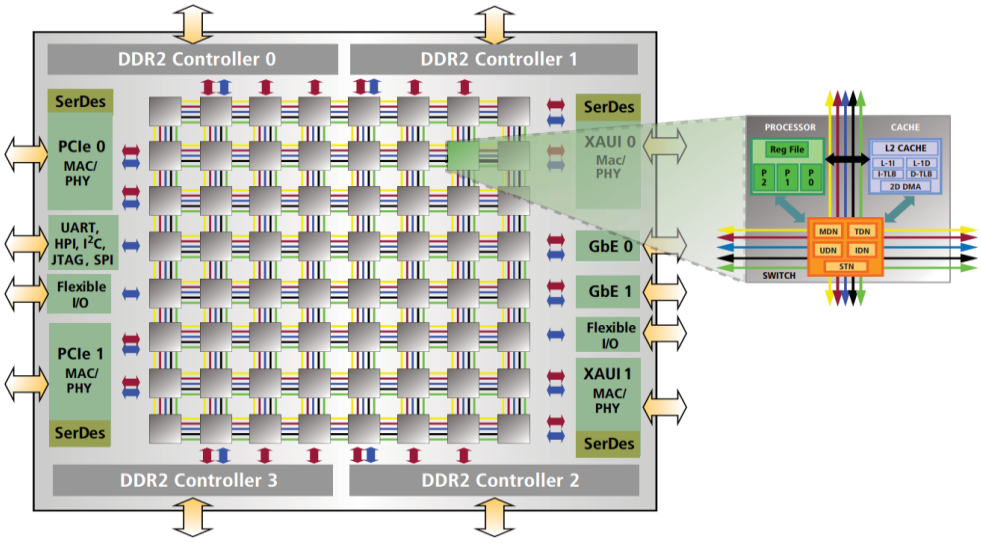


Figure 1.4: Schematic representation of Tile64 processor [43]

Intel's Teraflop Processor

Figure 1.5 shows Intel's research chip- Teraflop processor [45]. It contains 80 tiles interconnected using a NoC fabric in a 2D mesh topology. Each tile consists of a switching element called router and a core with private L1 cache. These cores can operate upto a frequency of 5.7 GHz and can achieve performance upto 1.81 Teraflops.

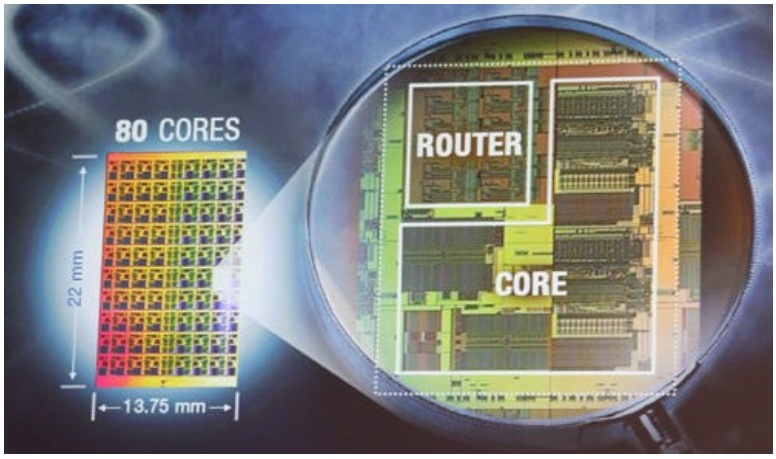


Figure 1.5: Intel's Teraflop processor [45]

As can be observed from the above examples, there are several shared resources in any given multi-core system, such as shared main memory, shared caches, shared interconnects, etc. It is crucial to manage these shared resources efficiently in order to enhance system performance. As we put numerous processing elements together to achieve performance benefits, it puts forth tremendous challenges in architectural design. The main challenges for modern heterogeneous CMPs, which desire optimum performance are:

- Instruction Throughput
- Power
-
- Memory Throughput
- Scalability

Due to the highly adapatable and scalable properties of NoCs, these fabrics are emerging as the most promising interconnect medium for modern CMP systems, such as the ones discussed above. This thesis focuses mainly on NoC-based systems; it sheds light on the current research in this area and presents solutions to overcome some of the key performance challenges pertaining to NoC-based multi-core systems. In the next section, we present a brief overview of NoC interconnection architectures.

1.2 Overview of Networks-on-chip (NoC) architectures

A network-on-chip (NoC) is a type of interconnection fabric for multi-core systems that uses packet-based switching for communication. NoC architectures provide modularity,

scalability and adaptability for intra-chip communication in modern CMPs; hence they are rapidly replacing traditional buses to become the backbone of modern multi-core systems [1][2] as seen in the examples of emerging multi-core chips presented in the previous section. Network-on-chip interconnect technology comprises of three main components-

- **Network Interface:** This is the interface between interconnect and processing elements. It is responsible for converting request/response packets into an appropriate format to be compatible with the processing elements or the interconnect.
- **Router:** This is the packet-switching element in the NoC fabric. A router is responsible for routing packets on a hop-by-hop basis from their source to destination.
- **Links:** Interconnection links are used to connect any two routers or a router and a processing element in a NoC-based system. Links provide a path for packet traversal in the network.

A NoC fabric possesses four key defining characteristics:

- **Topology:** This is the structure in which the processing elements in a CMP are connected to each other. The 2D mesh, as seen in Section 1.1, is the most commonly used topology.
- **Switching Techniques:** The method of allocation of various interconnect resources such as buffers, links, etc. to the communication unit (e.g., message, packet, etc.) is called switching, and several switching techniques can be used (e.g., wormhole packet switching, virtual circuit switchin, etc).

- **Routing Schemes:**The manner in which a route is selected for a packet to traverse the NoC from the source to the destination is governed by the chosen routing scheme. The most commonly used routing scheme for 2D mesh topologies is X-Y routing. In this routing scheme, the packets advance from one router to another in X-direction followed by Y-direction to reach their destination node.
- **Flow Control Techniques:** The mechanism of controlling communication flows between various nodes in the network to regulate the network traffic and reduce congestion or provide fault tolerance is governed by flow control techniques.

A detailed discussion of these NoC components and concepts is presented in Chapter 2.

1.3 Motivation

This thesis focuses mainly on improving overall system performance in multi-core platforms by improving memory throughput as well as communication efficiency. Typically, in today's multi-core systems, there are several applications co-running at any given time and sharing critical resources such as the NoC fabric, caches, memory controller and main memory. Figure 1.6 shows a representation of a typical NoC-based multi-core system. It consists of 16 tiles connected together in a 4×4 2D mesh. The system has a shared main memory, four shared memory controllers, and a shared L2 cache. In such a shared environment, the application packets are constantly in competition to win the race to reach their destination. Consequently, they are always interfering with other application packets resulting in slowdown of applications in the shared environment as opposed to when they are running alone. Determining which one packets gets precedence over another packet has a significant impact on overall system

performance. An inefficient NoC packet prioritization strategy can starve critical applications and reduce system throughput. Thus, it is important that requests are prioritized (i.e., scheduled) in an application-specific manner that is globally optimal, to minimize network latency and benefit overall system performance.

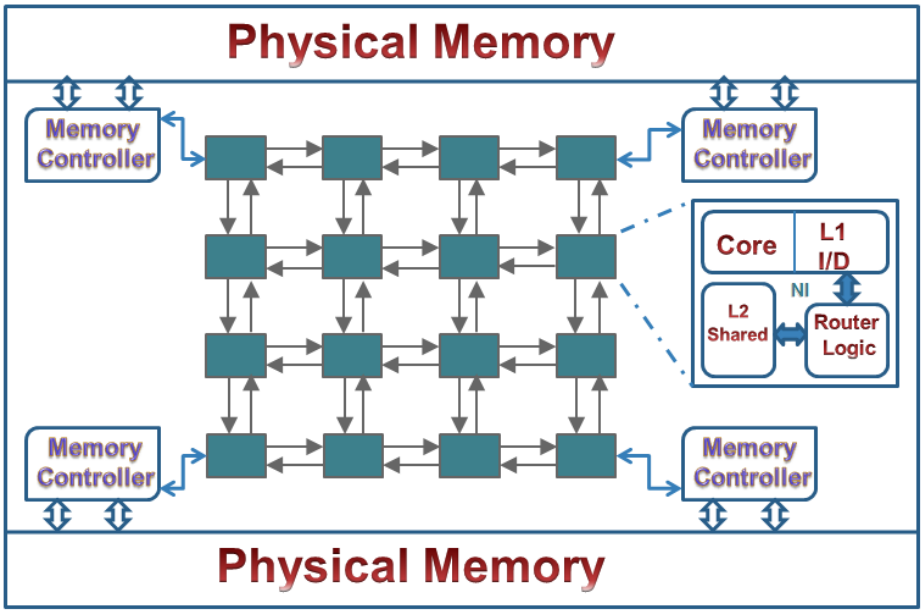


Figure 1.6: NoC-based multi-core system

Application data and instruction requests may encounter misses in the on-chip cache hierarchy, requiring request packets to traverse the NoC to reach the memory controller and subsequently access off-chip main memory. For such requests, in addition to the network latency, a notable main memory latency is added that increases the overall end-to-end packet latency. Memory latency can be elevated if packets experience stalls in the main memory due to bank conflicts and data contention. It is the responsibility of the memory controller to re-order requests going to the main memory so that memory stall cycles are minimized and memory utilization is maximized. The memory controller accomplishes this by keeping track of which

bank is being utilized currently and preparing against bank conflicts by scheduling packets to maximize bank level parallelism (BLP). Intelligent techniques for memory scheduling can improve memory latency and end-to-end latency, thereby significantly improving overall system throughput. Given that the heterogeneity in modern multi-core designs is increasing, these techniques must exploit heterogeneity in application characteristics [1][2], NoC utilization [3] and memory level parallelism (MLP) [4]-[6] to manage resource allocation of NoC and memory subsystems efficiently. In this thesis, we will refer to the on-chip cache packets as *network packets* and off-chip memory packets as *memory packets*.

Importance of NoC Packet Scheduling: The performance of applications is increasingly impacted by communication and memory performance. In particular, the throughput of an application is dependent on how fast its requested packets return to the core. It is very interesting to study the journey of a request packet originating at a core and back to the core and how different parameters affect its end-to-end latency. Figure 1.7 shows a 3×3 NoC-based CMP connected in a 2D concentrated mesh topology. Each router is connected to two cores having private L1 caches and one L2 cache bank through the network interface. In addition, a memory controller is connected to node 0, which is further connected to the main memory. As depicted in Figure 1.7, when a request originates at a core e.g. **A**, initially its private L1 cache is checked, if a miss is encountered, the request is sent to the next level in the memory hierarchy through the network. In Figure 1.7, the packet originating at core A misses its L1 cache and is injected in the network to access shared L2 cache. It traverses the network using the X-Y routing scheme to access a remote L2 cache bank **L**. If the request gets satisfied at **L**, it returns to the core A with the requested data. However, if a miss is encountered, it traverses the network again to go to the

memory controller **M** and further to the main memory. After fetching data from main memory, it traverses the network again to deliver the data to its source core **A**. The network latency of packets in these cases varies due to different hop distances and runtime NoC contention. Further, if a request encounters a miss at an L2 cache bank, it has to reach the memory controller and then access main memory. Once a request reaches the memory controller, it can face delays due to factors such as bank conflicts, read-write turnaround, bus busy etc. Thus, there are several factors contributing to the end-to-end latency of a request. At every step, a request packet experiences slowdown due to interference from other packets and contention. Hence, *we need to prioritize packets such that in situations of contention where several packets are competing for bandwidth, the requests that are most crucial to the overall system throughput can proceed before other requests.*

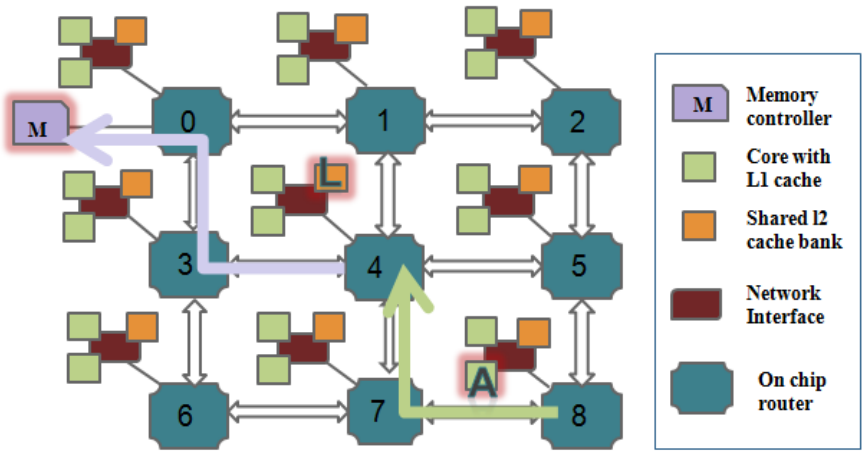


Figure 1.7: Journey of a packet in 3×3 2D mesh CMP

Importance of Memory Level Parallelism (MLP): Instruction Level Parallelism (ILP) has reached its limits because of the growing gap between memory and processor performance [4].

Increasing ILP is no longer beneficial as CPU performance is currently bounded by memory. In contemporary out-of-order superscalar processors even if the instruction issue width is increased or clock frequency is increased, the processor needs to stall until it receives its requested data from the memory. Thus, these optimizations for enhancing ILP would go unused unless memory access becomes faster. But memory performance is not improving as fast as processor performance. The "memory wall" is a major issue facing CMP designers today. The performance gap is increasing by ~50% each year between processors and main memories. Figure 1.8 shows the increasing gap between processor speed and memory speed. With technology shrinking down, processor speed is increasing exponentially but memory speed is not able to cope up with processor data bandwidth demands.

In such a situation, it becomes essential to exploit parallelism in order to improve memory throughput. Modern memories including caches are parallel architectures with multiple banks capable of operating simultaneously. Thus, we can improve memory throughput by maximizing bank level parallelism.

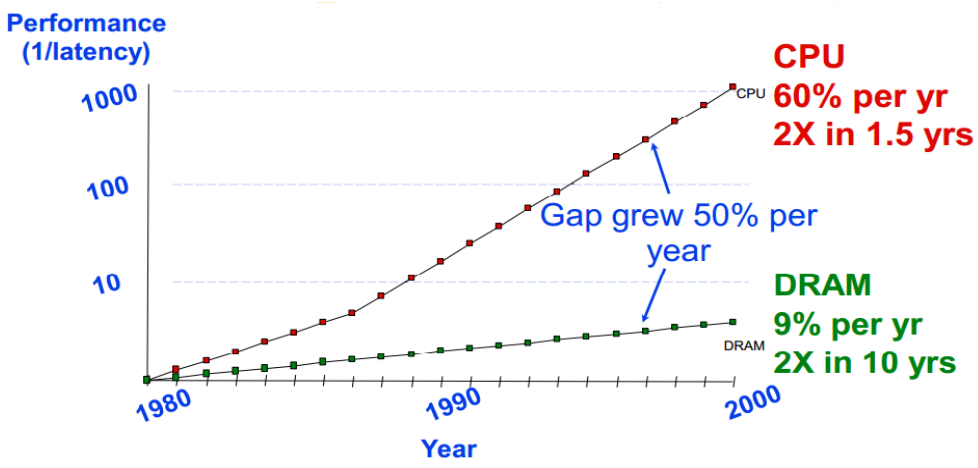


Figure 1.8: Memory wall [46]

Importance of Prioritization: As discussed earlier, it is vital to emphasize the prioritization of network and memory requests to enhance system throughput. However, the key question to be answered is: *at what point should request prioritization occur?* One stage where request prioritization typically occurs in today's systems is in the memory controller. However, out-of-order buffers for request scheduling in these controllers take up a lot of area and add a notable latency, especially when congestion in the network is high or memory-intensive workloads are running. In contrast, NoC routers have simple FIFO input/output buffers, with multiple requests competing for the same output channel simultaneously. As the NoC becomes more and more congested, it is vital to analyze such requests as early as possible, to uncover their impact on individual workloads as well as system performance and appropriately adjust their priorities. Thus, it is more cost-effective to apply *workload-aware prioritization* techniques in the NoC closer to the source rather than prioritizing the same requests at the memory controller. At the same time, improving main memory performance necessitates an increase in row buffer hits and a reduction in memory bank conflicts. This information is more readily available closest to the memory. Rather than relying on costly re-ordering in the memory controller, *memory-aware prioritization* techniques in the NoC can enable more cost-effective distinctions between requests.

1.4 Contributions

In this thesis, we make the following contributions:

- We introduce the novel concept of heterogeneous multi-staged prioritization in NoC intercinnection architectures. We demonstrate that by employing different prioritization criteria in two different parts of the NoC (e.g., part of the network closer to the memory

controller and farther away from the memory controller), overall system performance can be notably improved.

- We propose new algorithms to be employed in each of these two stages of prioritization. The first algorithm, for routers close to their originating cores, prioritizes request packets as per criticality of their applications. The second algorithm, which is for routers in proximity to memory controllers, aims to reduce bank conflict for memory requests as they proceed to the memory controller.
- We propose a new ranking scheme for determining criticality of applications. We evaluate previously proposed ranking metrics and show that our ranking scheme is more accurate.
- We evaluate the constraints on BLP in modern high-speed memories and propose an improved memory scheduling algorithm. We propose to employ our memory-scheduling algorithm in NoC routers in the vicinity of the memory controller (instead of modifying the controller) for better scalability and improved performance.
- We discuss the fairness issues in multi-programmed NoC-based systems and propose a novel anti-starvation algorithm to prevent slowdown of applications and boost overall system performance.
- We compare our work with some of the best performing recent work on packet prioritization and demonstrate the effectiveness of our proposed staged prioritization technique over these prior efforts.

Our experimental studies show consistent improvement with our proposed techniques over a baseline configuration frequently used in CMPs, as well as the best performing prior works in the area of network and memory prioritization.

1.5 Outline

This thesis is organized as follows. We discuss the preliminaries of this research area such as NoC basics, SDRAM operation, memory level parallelism, application characteristics and significance of packet scheduling in detail in Chapter 2. In Chapter 3, we elaborate on the problem statement of this thesis. In Chapter 4, we discuss the details of some of the state-of-the-art work proposed to address some existing challenges in NoC-based multi-core systems. Chapter 5 presents the proposed framework for our novel application-aware and memory-aware techniques and presents a new ranking metric for dynamic classification of applications. Chapter 6 presents an improved memory-aware packet scheduling technique specially tuned for high-speed memories. In Chapters 5 and 6, two different anti-starvation techniques are used for ensuring fairness in NoC-based systems. The experimental results and analysis for our hypotheses in Chapters 5 and 6 are presented along with the corresponding techniques. Finally, in Chapter 7, we present a summary and conclusion of this thesis and discuss future research directions for this work.

Chapter 2

Preliminaries

In this chapter, we discuss the basics of networks-on-chip and off-chip memory, mainly, their challenges and optimizations for improvement of overall system performance. We also discuss the significance of application-awareness and memory-awareness for efficient communication in modern multi-core systems.

2.1 NoC Basics

In this section, we will delve into the details of the NoC components and concepts briefly mentioned in Chapter 1. The major building blocks of NoC architecture are described as follows.

Network Interface (NI): Each processing element is connected to the router through a network interface (NI). This NI is responsible for breaking down read/write requests from processors into packets that are further partitioned into smaller data units called *flits* for on-chip traversalA packet is broken down into three types of flits- Header flit, Body flit and Tail flit. Figure 2.1 shows the breakdown of a packet at NI.

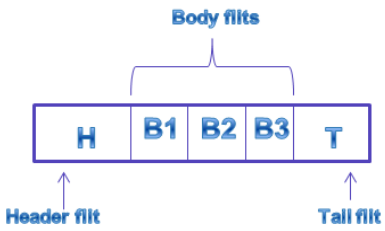


Figure 2.1: Breakdown of a packet at NI

The header flit carries the most significant information regarding destination. The body and tail flits simply follow the header flit in the network. The NI re-integrates the packet structure at the destination after all the flits belonging to the packet have been received from the network. This packet is parsed and the relevant data is then extracted and sent toto the processing element or memory component at the destination.

Router: A router is a key component of the NoC architecture, which directs packets to their destinations. Each router is connected to interconnect links through a certain number of input and output ports. Typically, routers have five input and five output ports in 2D mesh topologies. Figure 2.2 shows a standard router architecture. It has one port each on East, West, North and South directions and also a local port (to connect to a processor or memory) for that node. The flits arrive at these ports through the network interface. A state of the art 5-stage router is shown in Figure 2.3. For a 5-stage virtual channel router, a flit has to undergo the following stages-

- **Buffer Write and Route Compute:** An input flit is stored in the buffer in this stage. There are input buffers at the input port which contain multiple virtual channels. The routing path for this flit determines the output port the flit will be sent to, and this route computation is governed by the routing scheme chosen in the network.
- **VC Allocation:** The next stage is output virtual channel (VC) allocation. Only header flits go through this stage. The stage is responsible for reserving a virtual channel for body and tail flits for the packet to which the header flit belongs. Body and tail flits skip this step and follow the header flit to the next stage.
- **Switch Allocation:** In this stage, all the flits at a particular input port arrive at the

crossbar of the switch and contend for output ports with flits at other input ports. The switch allocation logic determines which flits gets each output port.

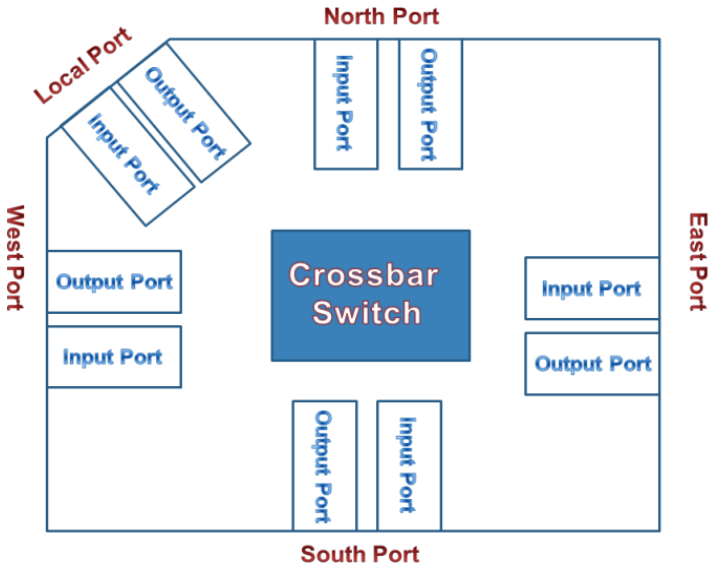


Figure 2.2: NoC router architecture

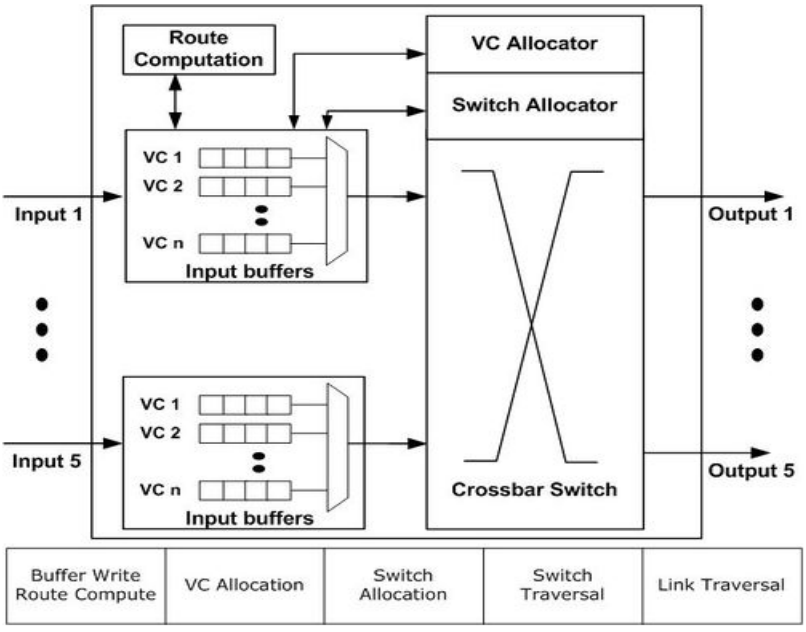


Figure 2.3: 5-stage virtual channel NoC router micro-architecture

- **Switch Traversal:** The winner flit in the earlier stage for each output port advances to the next stage by traversing a switch (crossbar) and arrives at the output port. This stage is called switch traversal.
- **Link Traversal:** After switch traversal, the flit has to traverse the link in order to go to the next router along its destination path.

Apart from the building blocks discussed above, NoCs are characterized by four key concepts:

- Topology
- Switching Techniques
- Routing Schemes
- Flow Control Techniques

These concepts are discussed in detail as follows.

Topology: The processing elements can be connected in different ways by a NoC and this connection structure is referred to as the NoC topology. There are two types of topologies- direct topologies and indirect topologies. In direct topologies, each node has a direct link to its neighbouring nodes. On the other hand, in indirect topologies, each node is connected to an external switch (i.e., router) which in turn has direct links to other switches in the network. Some of the important topologies are shown in Figure 2.4. The topologies- 2D mesh and octagon (Figure 2.4 (a) and (b)) represent direct topologies, whereas, the topologies- k-ary n-fly butterfly and fat-tree (Figure 2.4 (c) and (d)) represent indirect topologies.

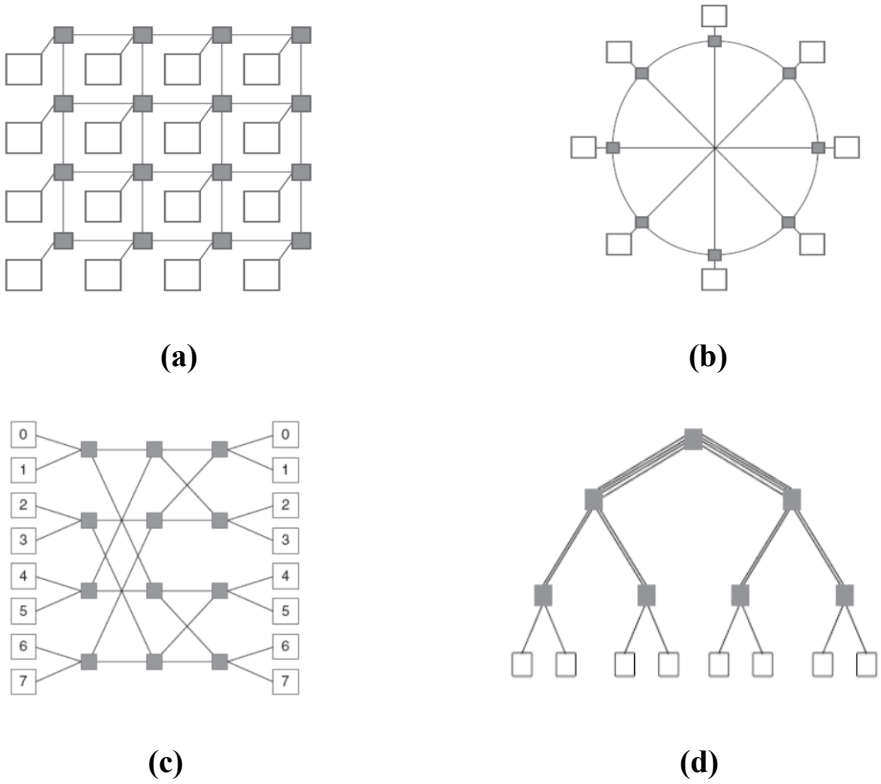


Figure 2.4: NoC topologies: (a) 2D Mesh, (b) Octagon, (c) k-ary n-fly Butterfly, and (d) Fat-tree [3]

Mesh topology is the most widely used topology for NoC-based CMPs. In this thesis, we mainly focus on multi-core processors connected in a 2D mesh topology in a NoC-based architecture.

Switching Techniques: Switching techniques govern the movement of communication messages in the network. There are three main types of switching techniques-

- **Circuit Switching:** In this type of switching technique, a path is first reserved in the network from source to destination before a message is sent through the network. This

path is broken only when complete transfer of the message takes place. This technique is rarely used in NoCs due to its inefficient usage of network resources.

- **Virtual Circuit Switching (VCS):** This technique uses multiple virtual circuits that may share the same physical communication links, to improve network utilization. A scheduling technique such as Time Division Multiplexing (TDM) is used to grant a virtual circuit access to a shared link..
- **Packet Switching:** Instead of reserving a path (circuit) and then transmitting a complete message over the network, in packet switching a message is broken down into multiple packetsthat proceed to traverse the network independently to reach their destination. This is the most cost-effective way to enable inter-node communication in multi-core systems. We adapt a variant of packet switching called wormhole packet switching for the NoC architecture considered in our work in this thesis.

Routing Schemes: The manner in which a route is selected for a packet to traverse the NoC from the source to the destination is governed by the chosen routing scheme. These schemes are meant to perform an error-free, congestion-free, deadlock- and livelock-free, quick as well as efficient routing of packets. There are two major types of routing schemes-

- **Static Routing:** In this type of routing, fixed routing paths are used for data transfers in the network. These routing paths are computed based on static information about the network regarding shortest path from source to destination, etc. These routing schemes are very simple to implement and have lower overhead. Deterministic X-Y routing

scheme (mentioned in Chapter 1) is a static routing technique and most widely used for mesh topology. Therefore we use this routing scheme in our work.

- **Dynamic Routing:** In this type of routing, the routing decisions are changed depending on the current network traffic. These routing schemes incorporate load balancing and fault tolerance mechanisms (for faulty links) in addition to mechanisms to determine the lowest latency paths to the destination. However, these routing schemes have a high hardware complexity and overhead.

Flow Control Techniques: With increasing number of cores on a chip and the number of parallel applications running on them, the network can easily become congested. In order to handle such network congestion as well as any network transmission errors, flow control mechanisms are often employed in the network. Some of the commonly used flow control schemes are- ACK-NACK flow control, Credit-based flow control and Xon/Xoff flow control. Amongst these schemes, credit-based flow control is the most popular. In this technique, a credit count is tracked by an upstream router's virtual channel. With each flit leaving the upstream router, this credit count is decremented. Backpressure is applied when the credit count becomes zero to prevent any further incoming flits – i.e., the upstream router stops injecting flits into the NoC if the credit count becomes zero. When buffers in the virtual channel in downstream routers get freed by forwarding the flits, credits are sent to upstream routers, after which flits injection by these routers can be resumed. We utilize this credit-based flow control technique in our work.

2.2 Multi-Programmed CMPs

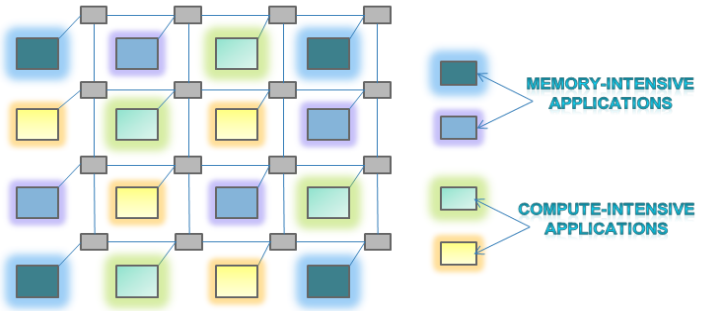


Figure 2.5: Multi-programmed NoC-based CMP

Figure 2.5 shows a representation of a typical NoC-based multi-programmed CMP system. There are 16 cores connected together in a 4×4 2D NoC mesh, running multiple parallel applications independently. This figure will be referred to in the following subsections.

Application Characteristics: We have reached the saturation point for Instruction Level Parallelism (ILP) now and hence use multiple parallel cores for faster turnaround of programs. A multi-core system such as the one shown in Figure 2.5 consists of several diverse applications executing simultaneously. These applications inject packets into the network every time there is a private cache miss. Such application packets may seek data from shared L2 cache or main memory for which they have to traverse the network to reach a remote node. The packets from multiple applications interfere with each other at the routers where they often compete for an output channel at the same time. Such inter-application interference causes unnecessary delays and leads to a deterioration in the system throughput. Hence, it is important to analyse the applications running on the system and make appropriate optimizations to enhance the system performance.

2.2.1 Packet Criticality

In multi-programmed CMPs, system throughput is governed by the harmonious execution of multiple applications running together. In such systems, routers face a lot of contention due to request/response packets of different applications interfering with each other for the same output channels. It is very important to distinguish between packets of different applications as opposed to treating all packets equally, to optimize system performance. The problem of categorizing packet criticality has been studied in a few recent works [16][17][19] that propose various classification metrics. These metrics are summarized below:

- **Packet slack estimation:** In [17], the authors compute the *slack* of a request packet to determine a packet's criticality. Slack is defined as the maximum delay a packet can tolerate in the network without affecting performance at its source core. For example, consider the scenario where a processor has two outstanding load misses that result in successive network request packets Pkt1 and Pkt2. If Pkt2 encounters lower NoC latency and returns with data before Pkt1, it cannot commit its corresponding load until Pkt1 returns. Thus Pkt2 has some slack for which it can be delayed without reducing application performance.The slack is used in arbitration decisions to prioritize packets with smaller slack over others. This metric is based on the nature of a request packet and its destination irrespective of the application it belongs to. But network contention is not considered in the process of estimation, which can lead to incorrect values of packet slack and hence flawed determination of the packet's criticality.
- **Network episode height and length estimation**: An application typically goes through two phases in its execution span- compute phase and network phase. Compute phase is

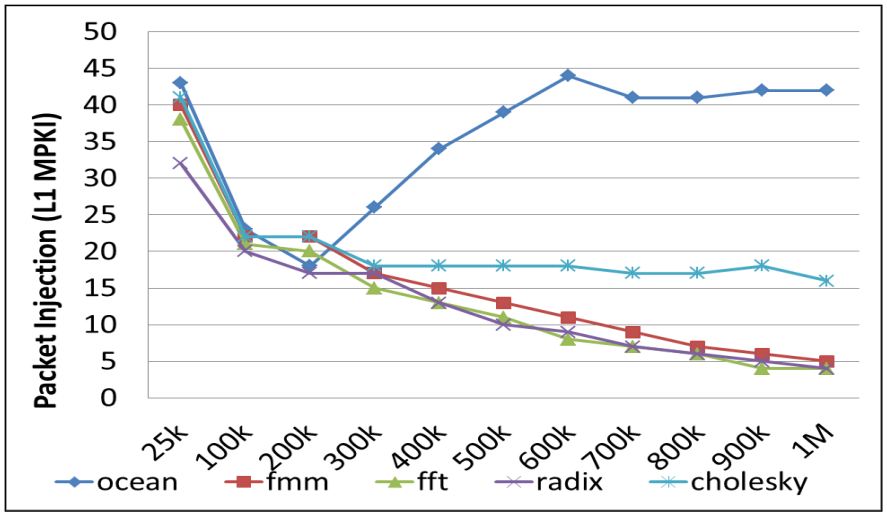
the execution interval in which there are no outstanding requests to the memory from that application. On the contrary, network phase is the execution phase in which the application has pending requests to the memory. In [19], the authors use the metrics-*network episode length* and *episode height* to perform fine grain classification of applications and their requests. The *episode length* is the number of cycles measured from the time of injection of a packet in the network by an application, until all requests for that application is satisfied. *Episode height* is the number of outstanding L1 misses captured for this episode length. The episode length and height value is an indicator of the intensiveness of an application's network-phase, which allows predicting memory intensiveness of an application and its packet criticality. To determine an episode length, a count of cycles is kept for which the MSHR (Miss Status Handling Registers) queue [20] is occupied. However, this metric has some drawbacks. MSHR queues remove an entry only when the request is satisfied [21] (details about MSHR follow later in this section). Hence, if the request is delayed in the network due to contention, it will result in longer network episodes. As a result, there is a high probability of incorrectly projecting that the application is more network-intensive with this approach.

- **Private cache miss count estimation**: In [16], the authors use private cache misses per instructions (MPI) as a ranking metric for determining stall time criticality of an application packet. They form eight clusters of ranks based on the relative MPI of all the applications. The application with the lowest MPI gets the highest rank. This metric is unaffected by the network congestion state and is a reliable indicator of an application's memory-intensiveness. However, it gives us a very broad idea about the nature of an application in terms of its latency criticality and does not capture the

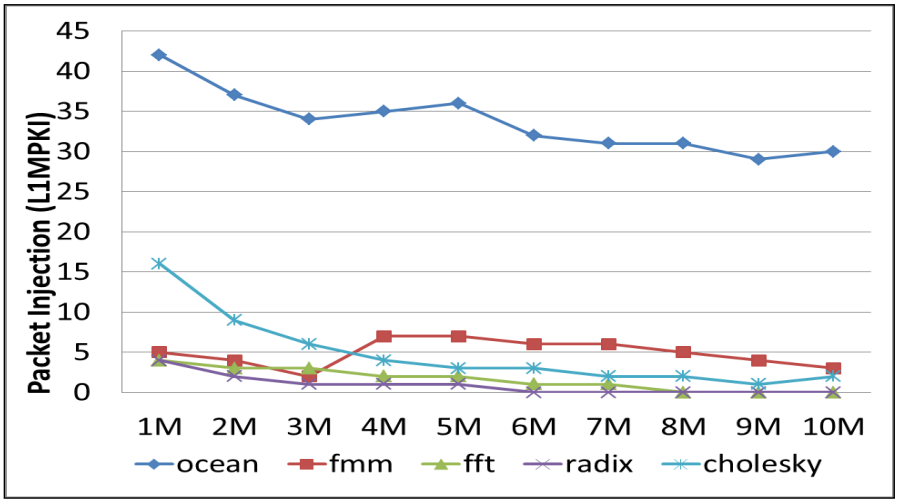
memory level parallelism (MLP) of an application.

Need for a dynamic fine-grain application classification strategy: In a shared CMP with multiple parallel applications, at any given time, two or more packets can be contending for the same resource. Intelligent techniques are required to efficiently allocate resources based on the needs of different applications and their respective network and memory packets. Applications rarely demonstrate invariant characteristics and usually fluctuate between memory-intensive and compute-intensive phases over their lifetime. Figure 2.6 shows packet injection trends in terms of L1MPKI (L1 misses per thousand instructions) for applications from the SPLASH-2 benchmark suite. The benchmarks show varying degree of memory-intensiveness at different times and hence the throughput demands of the applications change as well.

We can observe from Figure 2.6 that some applications (e.g., *fft* and *radix*) have similar packet injection trend, so using L1MPKI alone to distinguish between packets of different applications during scheduling is not sufficient as done in some prior works (e.g., [16]). We need a more sophisticated metric to perform finer grain classification of applications that can exploit unique application-specific characteristics of packet injection. In addition, applications demonstrate varying degrees of memory level parallelism depending on their temporal and spatial localities. A classification metric should be able to exploit this information.



(a)



(b)

Figure 2.6: Packet injection trend for benchmarks from SPLASH-2 suite over a window of (a) 1M, and (b) 10M instructions.

Figure 2.7 shows a typical process to network interface and flow of packets from processor to network. For an application running on Core, it requires some data to be fetched from the memory. First, the private L1 cache is checked for the required data. If there is a L1 miss, the data request is sent to the next level of cache (mostly shared). The request has to go through the MSHR (Miss Status Handling Registers) module attached to the L1 cache's circuitry, before it

can enter into the network to access shared L2 cache. There are multiple MSHRs which form a MSHR queue. MSHR queues are checked on each L1 cache miss. The entry regarding current miss is added to the queue only if a miss to the same block as the current miss does not already exist. The greater the MSHR queue occupancy, the larger is the number of blocks demonstrating MLP in an application [21]. This MSHR queue occupancy also indicates the instantaneous network demands of an application. We term the MSHR queue occupancy as *MLP index*.

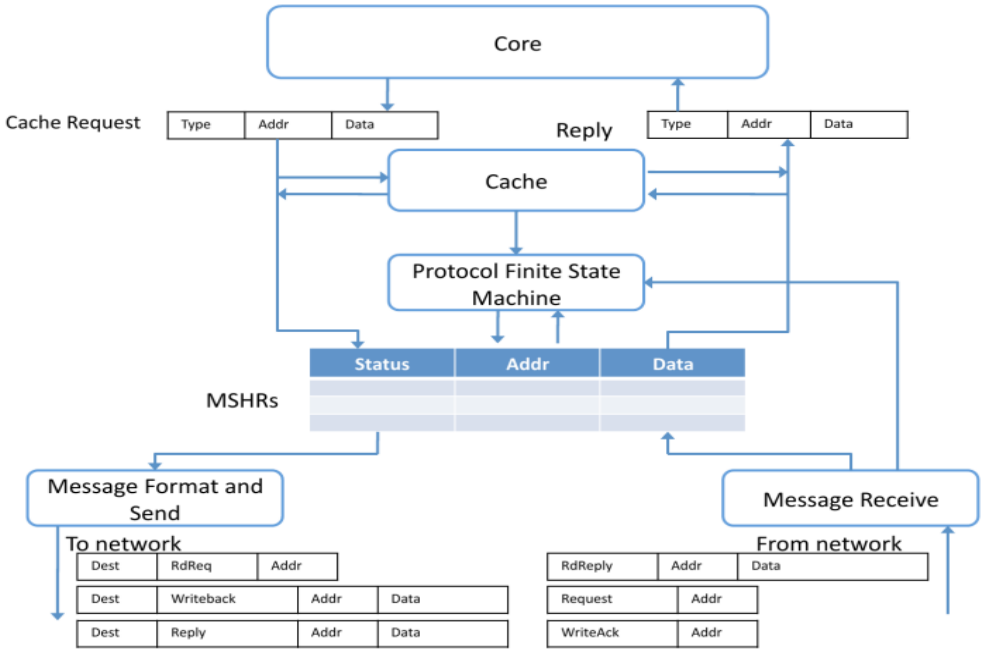
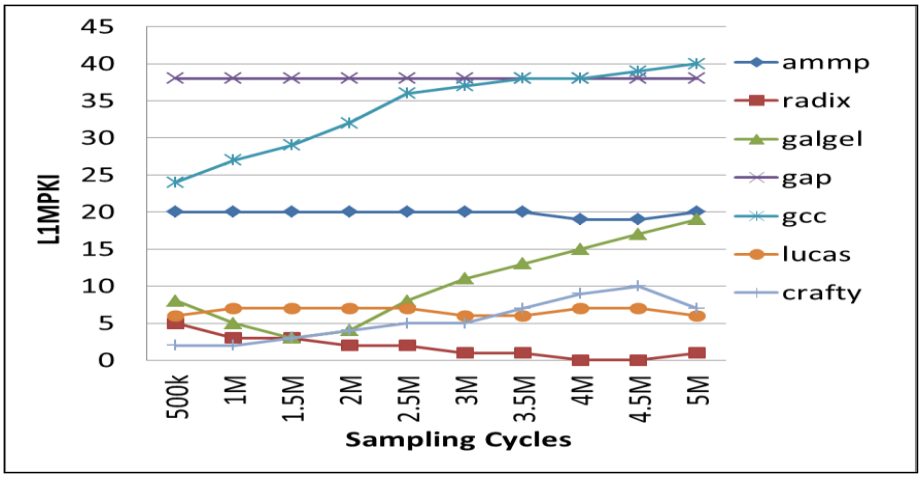


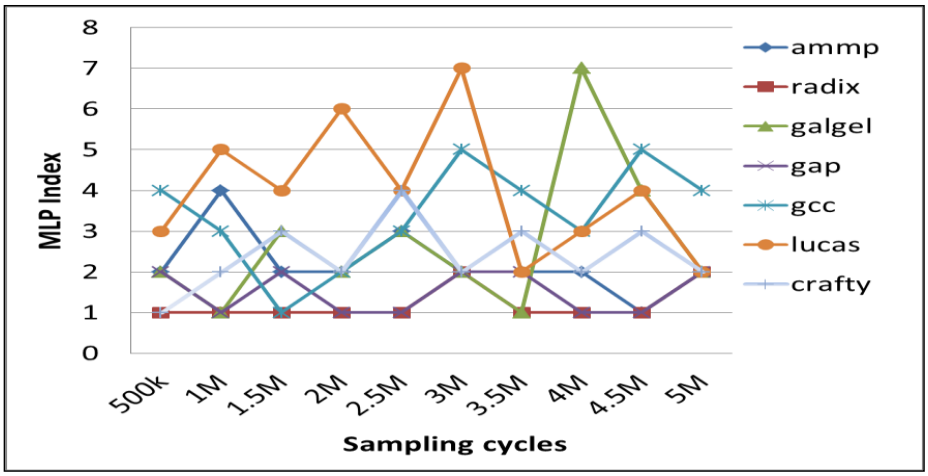
Figure 2.7: Processor-to-network interface [21]

As applications typically demonstrate behavioral variations over the span of their execution, their network demands and therefore criticality also varies over time. To better understand this phenomenon, we captured the L1MPKI as well as number of outstanding requests in the L1 MSHR queue for applications from the SPLASH-2 and SPEC2K benchmark suites. Figure 2.8 (a)-(b) show varying L1 MPKI and MLP index values for the various

applications over a period of 5M cycles. An interesting observation is that even though L1 MPKI may be constant over an interval for an application, its MLP index can change significantly during that time (e.g., for *ammp, lucas,* etc). The results demonstrate a need for fine-grain classification of applications based on their time-varying network demands and MLP.



(a)

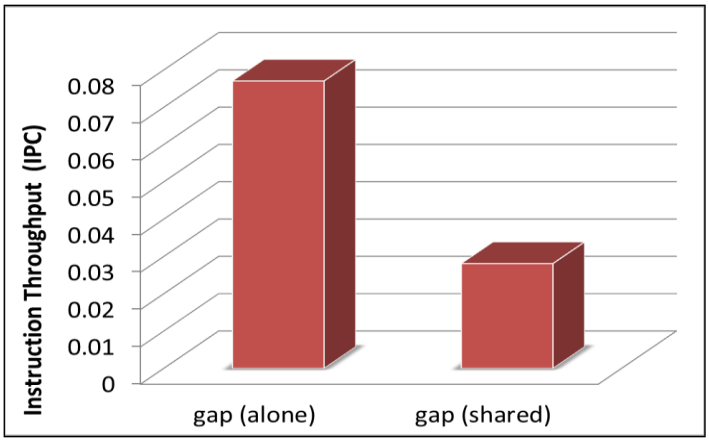


(b)

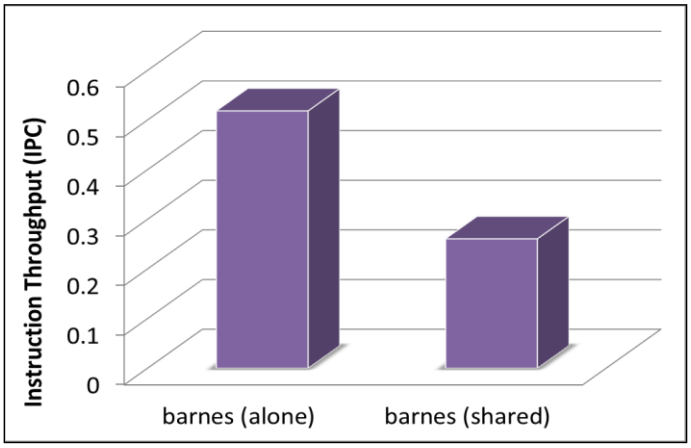
Figure 2.8: (a) L1 MPKI, and (b) number of outstanding requests in L1 MSHR queue over time for SPLASH-2 and SPEC2K benchmarks.

2.2.2 System Fairness Issues

Modern multi-core systems typically execute multiple applications in parallel. Such applications inject packets into the NoC, where routers handle the journey of these packets to their destination. Routers must frequently cope with contention while granting the output channel to one of the contending packets. In such scenarios of inter-application interference, it is important to distinguish between packets and make an appropriate choice in prioritizing one packet over others. During such packet scheduling that is usually based on a certain criteria (e.g. stall time criticality [16], packet slack [17]), there is a possibility that certain packets get de-prioritized all the time and suffer from starvation. As a result, some application packets get significantly slowed down. Figure 2.9 shows the slowdown experienced by the applications *barnes* and *gap* when they are run together in a multi-programmed environment employing an arbitration strategy as in [16] versus when they are run in a standalone manner with no interference from other applications. *Barnes* is a compute-intensive application while *gap* is memory-intensive, thus *gap* injects more packets into the NoC. When these applications are run together, both applications experience some slowdown. As *gap* is memory-intensive and is frequently de-prioritized by the strategy in [16] over the *barnes* compute-intensive application, its application throughput (instructions per cycle) takes a more significant hit (Figure 2.9). To prevent this slowdown and ensure fairness in the system, anti-starvation strategies are often employed in addition to scheduling algorithms. Some of the standard and previously proposed anti-starvation strategies are discussed below.



(a)



(b)

Figure 2.9: Slowdown for (a) memory-intensive workload (gap) (b) compute-intensive workload (barnes) (standalone vs together).

In [16][17][18], *time-based (TB) batching* is employed to enable fairness among application packets. In time-based batching, a packet is tagged with a *batch id* that is a function of the time when it is injected into the NoC. Requests originating at the same time or within the same batching interval form a batch. At the interval of every T cycles, this batch id is incremented cyclically. With the view of enforcing fairness, when two packets belonging to different batches compete with each other, the packet with lower batch id is given priority over the other packet. The underlying assumption is that the older packet has been starved for a long

time. However, batching is not necessarily an effective way of coping with starvation. For example, with a typically used batching interval of 16,000 cycles, a packet injected at cycle 0 has batch id 0 and a packet injected at cycle 15,998 also has batch id 0. In the following interval, the batch id is incremented and now the packet originating at cycle 16,002 has batch id 1. If the packet originating at cycle 15,998 competes with one originating at cycle 16,002, the former packet will win out even though it may not be starved, rendering the TB batching technique ineffective in such scenarios.

In another similar technique called *packet-based (PB) batching* [16], instead of a time-based threshold, the threshold is based on number of packets injected. Here the batch id is incremented when N packets have been injected into the NoC. This technique has the same disadvantages as those discussed for TB batching. In addition, this technique requires co-ordination among all the nodes to enable a global batching across the system, resulting in a prohibitive communication overhead.

The two batching techniques discussed above are widely used in NoC packet scheduling to enforce fairness but face inherent drawbacks as discussed above. Moreover, both strategies do not distinguish between on-chip cache requests and off-chip memory requests. Thus, higher ranked off-chip memory packets can defeat lower ranked on-chip cache request packets. However, de-prioritizing lower latency on-chip cache requests can hurt overall performance. Thus, more balanced anti-starvation scheduling mechanisms are essential to aggressively improve performance. Chapter 6 presents our proposed anti-starvation mechanism.

2.3 SDRAM Architecture and Memory Level Parallelism

In this section, we will discuss the basic SDRAM operation and how SDRAM characteristics and memory level parallelism can be exploited to improve memory throughput and overall system throughput in turn.

2.3.1 SDRAM Operation

Widely used SDRAMs operate with three main commands: precharge (PRE), activate (ACT) and Read/Write (R/W) [6]. For any request, the SDRAM memory controller needs to know its bank address, row address and column address. When a request arrives at the controller, it checks if the bank being requested is already active and if not, issues an ACT command. The desired row is then copied into a row buffer to access the requested address. Once the data is fetched, a PRE command is issued to restore the row in the row buffer and idle the bank depending on whether the memory implements a closed-page or an open-page policy. Most memories implement an open-page policy to encourage row hits and minimize ACT and PRE commands. Another facet of SDRAMs is that while one bank is busy another bank can still be activated and accessed under certain conditions. It is also desirable for requests approaching memory to do so in a manner that will avoid consecutive requests approaching different rows of the same bank, to reduce memory latency.

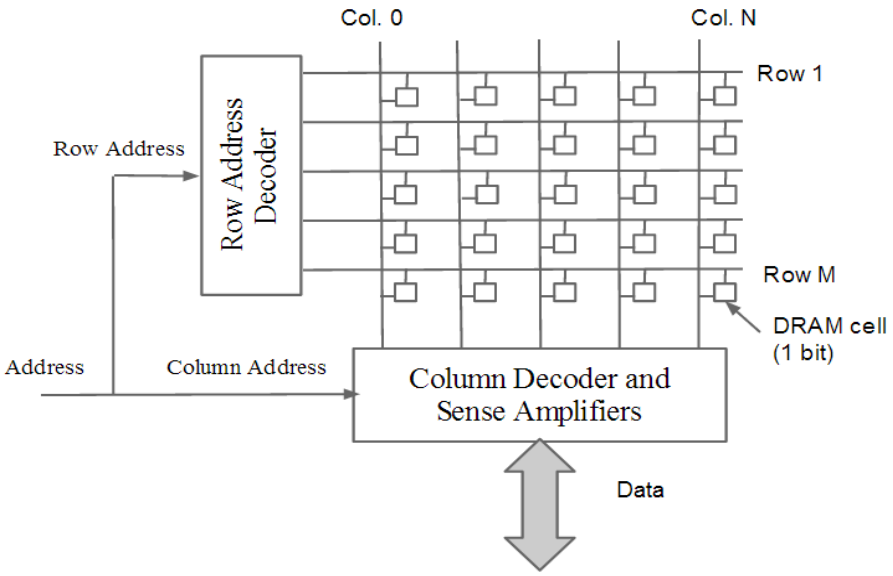


Figure 2.10: DRAM architecture

Figure 2.10 shows a typical DRAM architecture which consists of a 2D array of rows and columns (wordlines and bitlines). Whenever a request arrives at main memory, the address is decoded to identify the row to be addressed. This is called Row Access Strobe (RAS). This row is then copied into the row buffer. The column address is then decoded to know which bitline from the copied row needs to be accessed. This is called Column Access Strobe (CAS). Memory access latency is dependent on how many cycles are taken for the above operations, for any request. Some of the commonly used timing parameters for SDRAM are shown in Table 2.1.

Table 2.1: SDRAM timing parameters

Timing Parameters	Abbreviations	Description
tRAS	Row Access Latency	Cycles consumed for accessing data from a row when it is opened.
tCAS	Column Access Latency	Time for reading data from the specified column in an opened row.
tRCD	Row Cycle Delay	Minimum time between accesses to different rows of the same bank.
tRP	Pre-charge Delay	Cycles to precharge the DRAM bank
tWTR	Write- Read Delay	When read request follows write request to a same memory location, tWTR cycles are required to perform operation.

2.3.2 Typical Memory Controller Functionalities

Before requests go to the main memory, they have to pass through the memory controller. Modern memory controllers perform a variety of functions in addition to generating commands for the main memory. When an off-chip memory request arrives at the memory controller, it translates the request address to main memory address and issues appropriate commands. Some of the typical functionalities of a memory controller are-

- *Issuing DRAM commands* like ACT/PRE/READ/WRITE to activate or de-activate a bank and perform read/write operations;
- *Issuing REFRESH commands* to the DRAM cells so that the written data is retained;
- *Managing power states* of the main memory with the goal of minimizing main memory power consumption;
- *Request Scheduling* of the packets before they go to the main memory to avoid stalls due to bank busy/data contention;

- *Ensuring reliability* of request packets by employing error correction strategies in the memory controller;

Memory controllers are placed on-chip and as the number of functions performed by them increases, their hardware complexity increases. In particular, reorder buffers inside memory controllers take up a lot of area and energy. Hence, efforts are being made to incorporate some of the memory controller functionalities in the on-chip interconnect (mostly routers) in the NoC-based multi-core systems. SDRAM-aware router [13] is one such technique which uses existing resources in a router (input/output queues and switch arbitration) to perform request scheduling so that memory controller is relieved of the expensive reorder buffers. We adapt this goal and attempt to improve overall memory performance by employing more efficient memory scheduling techniques in network routers. We present two such techniques in Chapters 5 and 6.

2.3.3 Bank Level Parallelism

Modern memories are no longer monolithic structures but rather multi-bank architectures that can operate in parallel. The memory structure is shown as follows in Figure 2.11. A memory controller is connected to the main memory through channel. A channel may consist of one or more DIMMs (Dual In-line Memory Module). An SDRAM DIMM represents an independent memory device. Figure 2.12 shows the logical organization of a SDRAM DIMM. A DIMM consists of multiple logical ranks which further consist of banks. Different banks have separate command generator circuitry; hence they can be operated in parallel.

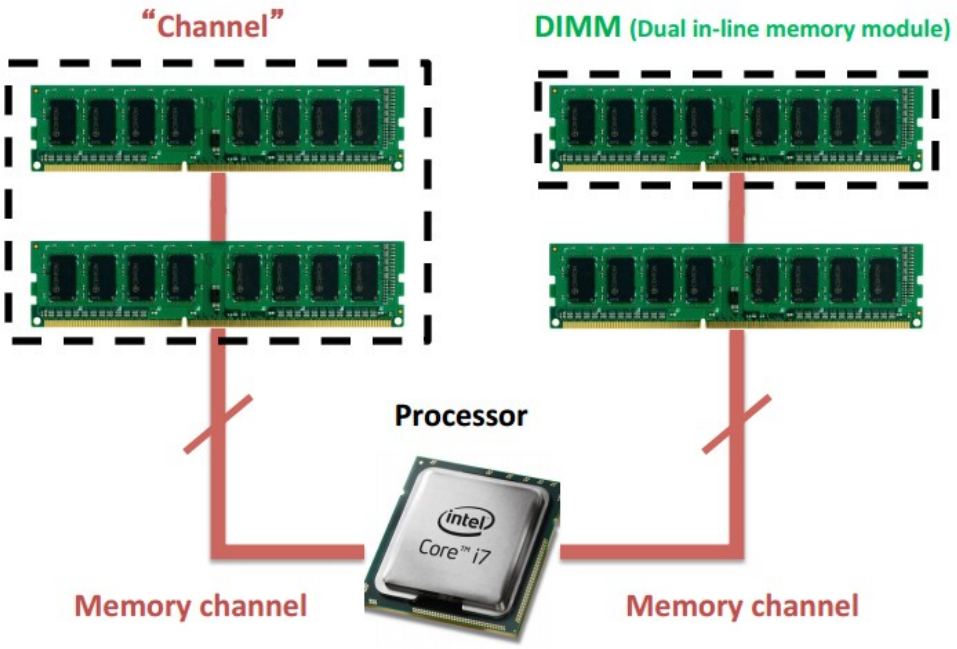


Figure 2.11: Memory subsystem representation [50]

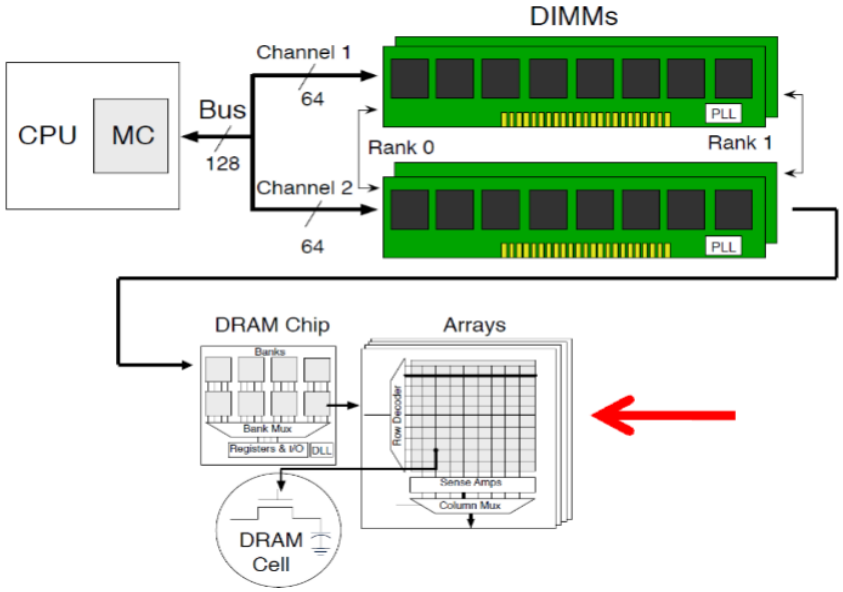


Figure 2.12: Logical organization of a SDRAM DIMM [35]

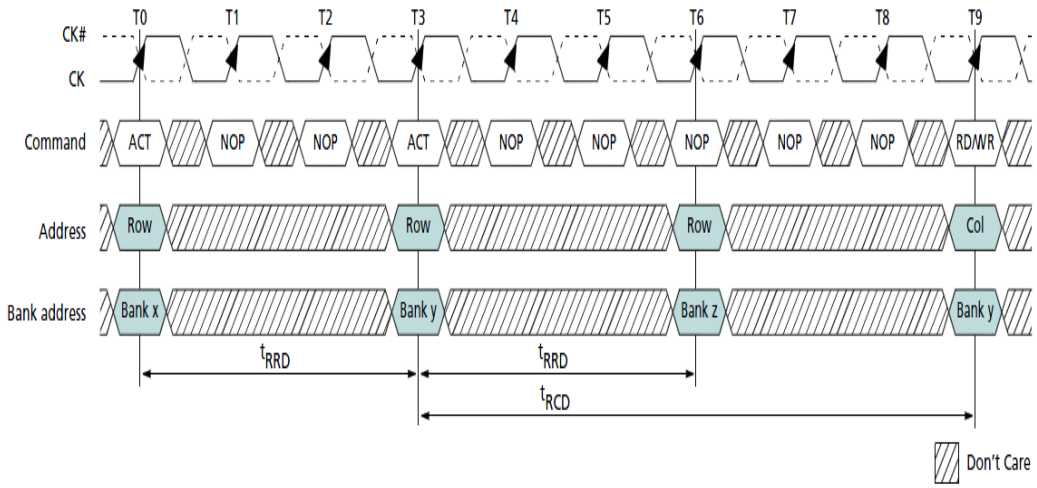


Figure 2.13: Timing constraints- tRRD and tRCD [24]

The timing diagram in Figure 2.13 shows the constraints for operating different banks in parallel and the same bank in succession for different rows. The parameter, tRRD represents the number of cycles it takes to activate two different banks in succession. Figure 2.13 shows that once Bank *x* is activated, Bank *y* can be activated after tRRD cycles whereas Bank *y* can again be activated after tRCD cycles. Usually, tRRD is much lesser than tRCD and hence different banks activated in succession are considered as almost concurrent. However, for higher capacity SDRAMs (> 1GB or eight banks) and high-speed memories, there is a constraint on how many banks can be activated in parallel. In such memories, over a window of tFAW number of cycles, only four banks can be activated. This is to avoid current surge and excessive power consumption. This constraint is termed as tFAW (Four Activate Window). If another bank of the same rank needs to be accessed, it has to wait until tFAW number of cycles or one of the already activated banks are pre-charged. Thus, even if we try to maximize bank level parallelism (BLP) by exploiting spatial locality of packets, packets can still experience head of the queue latency at the memory if they encounter the above-mentioned scenario. The figure below shows the timing representation of the tFAW constraint.

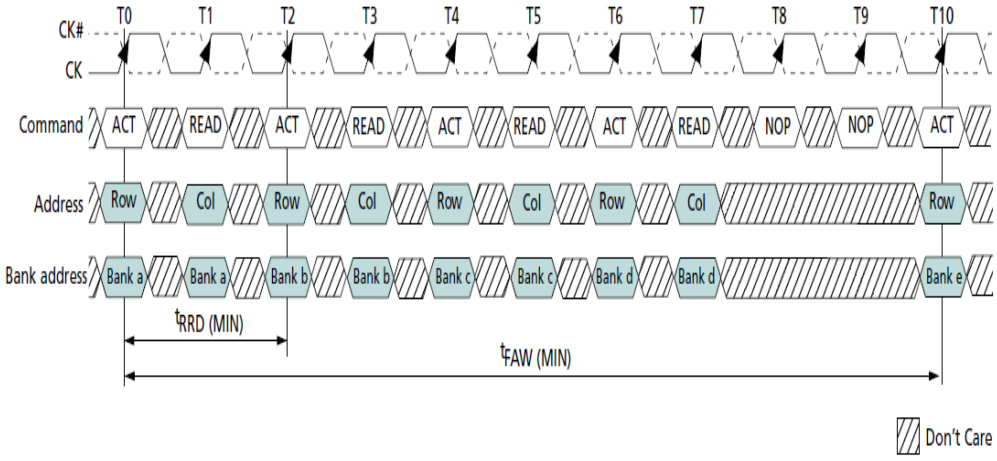


Figure 2.14: Timing constraints- tRRD and tFAW [24]

In Figure 2.14, bank a, b, c and d can be activated concurrently. However, when bank e needs to be activated, there is no bank conflict or data contention issue but it has to wait until tFAW cycles have passed since the activation of bank a. This puts a constraint on maximum bank level parallelism possible. We address this issue in our second memory-scheduling algorithm described in Chapter 6.

Chapter 3

Problem Statement

With the advent of multi-core computing, there is a significant amount of research going on to improve the overall system performance in multi-core chips. As the number of components on a chip increase, the management of shared resources such as interconnects, shared caches, memory controller as well as main memory becomes more critical. Each module in a system is inter-dependent due to communication and therefore it is important to consider on-chip as well as off-chip components while performing synthesis.

The key metrics that determine overall system performance are –

➢ Application throughput	➢ System throughput
➢ Memory utilization	➢ Network utilization
➢ Network latency	➢ Power and Energy consumption
➢ Main memory latency	➢ System fairness

Research in NoC-based multi-core systems has been focused on optimizing one or more of these metrics to enhance the achievable performance. This thesis aims to optimize the following factors to benefit overall system performance in multi-core chip designs:

- Increase system throughput
- Minimize energy consumption
- Minimize main memory latency
- Increase system fairness

A multi-core system executes multiple workloads in parallel, ranging from heavily memory-intensive to memory non-intensive applications. This thesis aims to address the following challenges in order to achieve the abovementioned performance goals:

- **Packet arbitration:** The end-to-end journey of a packet requested by a core plays a significant role in determination of QoS and consequently performance. Every application demands data or instructions from the memory; some applications do so more often than others. In its journey to the destination, a packet might face slowdown due to other packets interfering with it at the routers or memory access delays. Techniques such as prefetching and application- and memory-mapping minimize memory delays by keeping the data needed by any application closer to its core. However, application characteristics often vary with time, so it is difficult to accurately predict what data is needed by the application throughout an application's execution. Efficient packet arbitration can enables high gains in performance by tracking the changing data demands of applications. NoC routers can accelerate or decelerate certain packets in order to reduce memory stalls and benefit the system throughput. This thesis explores the potential of packet arbitration in a NoC-based system.
- **System latency**: The overall system latency of a packet comprises of *network latency* and *memory latency*. Network latency is primarily governed by the on-chip interconnects and is determined by the communication latency of packets traversing the network. Memory latency is governed by the number of stalls faced due to row/bank conflicts and data contention (Read-Write turnaround). System throughput is a function of system latency, and is defined as the total amount of work done (i.e., instructions

executed) over a certain number of cycles. This thesis aims at reducing system latency to improve system throughput.

- **Application-awareness:** Modern multi-core systems execute diverse workloads in parallel to maximize performance. There are two broad categories of applications- *Compute-Intensive and Memory-Intensive*. Compute-intensive applications are latency sensitive and benefit system throughput more as the processors executing these applications spend most of the time on computation rather than waiting for data to be fetched from memory. Memory-intensive applications, on the other hand, are relatively more latency tolerant. These types of applications do not hamper the system performance even if their packets face some delays.
- **Memory-awareness:** Main memory latency is the most significant contributor to system latency. Memory latency can be minimized by mitigating stalls due to bank busy and row conflicts. Also, the spatial or temporal locality of application packets determines their memory access latencies. Hence, memory-state awareness is important in taking scheduling decisions so that bank and row conflicts, which are the main cause of memory access delays, are mitigated. Hence, this thesis exploits memory-awareness in NoC routers to achieve improvement in memory throughput and efficiency.
- **System fairness**: Along with system performance, it is equally important to preserve system fairness. In multi-programmed systems, inter-application interference is inevitable. This results in slowdown of applications. In such cases, when one application is repeatedly prioritized over others, some applications may suffer from starvation and cause a reduction in the overall system performance. Hence, this thesis also aims to establish system fairness while improving system throughput.

- **Heterogeneity in architecture**: With increasing heterogeneity in multi-core systems like AMD Fusion, Tegra SoCs etc. where the system is composed of CPUs, GPUs, image processors, video encoders and other types of processors, it is important that we consider the fact that systems might no longer consist of coherent processing elements. Hence, we need mechanisms that can deal with heterogeneity in terms of workloads executing, NoC router architectures, and heterogeneous memories. We thus introduce the concept of heterogeneous prioritization framework to suit the requirement of modern heterogeneous multi-core computing systems.

This thesis first explores and analyzes application packet criticality and memory access behavior. Based on this analysis, we propose techniques to address the above mentioned challenges leading to performance improvement in terms of system throughput, memory latency, system fairness and energy consumption.

Chapter 4

Related Work

The potential of efficient packet prioritization in influencing overall performance for multi-core systems is well recognized. Several recent efforts have proposed architectural and algorithmic strategies for either NoC packet scheduling or memory-bound packet scheduling. Several other efforts have focused on designing heterogeneous NoC architectures with support for fairness. We summarize a representative subset of these techniques below.

4.1 Heterogeneous NoC Architecture

Several efforts propose to architect NoCs to cope with diverse applications running in parallel and their communication demands. In [39], Grot et al. propose KiloNoC, a topology-aware QoS-enabled NoC architecture. In [19], Mishra et al. propose sub-networks for separating compute-intensive and memory-intensive applications in NoC based systems. For the sub-networks proposed in [19], additional hardware circuitry redirects packets injected into the network into a sub-network based on the application type. In [40], the authors propose using heterogeneous routers that can more effectively match application packet QoS requirements while saving power. In [14], Phadke et al. propose heterogeneous main memory architecture for optimizing latency, bandwidth and power requirements. All of these prior works demand extensive customization of hardware blocks which may be costly or even impractical for commodity processing cores and memory.

4.2 Memory-bound Packet Scheduling Techniques

Several prior efforts such as Thread Cluster Memory Scheduling [1], Stall Time Fair Memory Scheduling [7], ATLAS [8], SMS [9] and PARBS [10] have proposed memory scheduling techniques that modify memory controllers to improve memory access performance. These techniques consider thread criticality and fairness while scheduling requests in the memory controller [1][7][8][10]. However, such scheduling algorithms have a large communication overhead due to limited visibility of the network. These techniques are also unaware of the network and cores executing the applications. The techniques proposed in [11][12] schedule memory requests based on network contention and executing applications, but assume that information about each core is available in the memory controller. Such an assumption entails a huge communication overhead between the memory controller and cores and as such, these techniques are prohibitive and expensive to implement in practice. Main memory requests often face delays due to stalls at the head of the issue queue owing to bank conflicts or data contention. Hence, it is important to re-order these requests, as has been explored in some previous efforts on memory scheduling. For example, in [13], SDRAM-aware NoC routers are proposed that are programmed with the main memory timing parameters, allowing for request re-ordering in a memory-aware manner to improve performance. However, this approach reduces portability as well as scalability in emerging heterogeneous memory systems [14]. The technique in [15] enhances the technique from [13] by distinguishing between demand packets and prefetch packets while reordering requests as per memory state, but the fundamental drawbacks mentioned for [13] still remain. In [42], Ebrahimi et al. propose a new memory-scheduling algorithm for parallel applications aiming to reduce inter-thread interference in the memory system. However, this approach is again agnostic to the network congestion.

4.3 NoC Packet Scheduling Techniques

Das et al. propose NoC scheduling using the ranking metrics stall time criticality [16] and packet slack [17] for co-running applications. Stall time criticality is calculated based on the number of L1 cache misses per thousand instructions (L1MPKI) and packet slack is a measure of delay tolerance of a particular packet in the NoC. However, L1MPKI cannot be used as a fine grain classifier of applications (as discussed in Chapter 2) and this metric does not consider network congestion while determining slack. Therefore, these metrics are not very accurate for ranking application criticality. Recently proposed network scheduling schemes such as [18] aim to bring uniformity in network latencies and main memory bank utilization. In [18], Sharifi et al. propose to apply separate prioritization rules to request and response packets. However, the techniques are application-oblivious and can lead to increased delays in the system while trying to equalize packet latencies. In addition, these techniques prioritize off-chip memory requests and responses only and ignore the on-chip cache requests and responses.

4.4 Fair Scheduling Techniques

Das et al. [16][17] use time-based batching and packet-based batching techniques to ensure anti-starvation in a strict priority enforced environment. In [41], the authors propose a source-based throttling mechanism to ensure fairness in shared CMPs. However, none of these techniques differentiates between on-chip cache requests and main memory requests for their starvation criteria. Our technique presented in Chapter 6 describes a novel anti-starvation method to be applied system-wide to ensure fairness and differentiates between network packets and memory packets.

4.5 Inter-application Interference Reduction Techniques

The TCM technique [1] by Kim et al. proposes a mechanism to determine interference caused by any thread. It uses insertion shuffling between high priority and low priority threads to guarantee fairness amongst all threads (cores). Muralidhara et al. [34] propose having separate memory channels depending on the application type to reduce the interference between heavily interfering applications. In addition, in [19], the authors propose various sub-networks within a network to run a specific type of application, which reduces inter-application interference to a significant extent. Such techniques aim to mitigate inter-application interference to minimize the slowdown of applications running on systems and boost its performance.

In summary, existing memory scheduling techniques are typically network-unaware, whereas existing network scheduling techniques are agnostic to main memory performance, often being analyzed using simple memory models with fixed latencies. However, we believe that as overall system performance is a function of both network latency and memory latency, focusing on one of these aspects alone to improve performance is sub-optimal. Hence, we propose a novel holistic solution for intelligently prioritizing (i.e., scheduling) network packets (on-chip cache requests/responses) and memory packets (off-chip requests/responses) to overcome the key drawbacks of existing techniques and improve overall performance. The optimization goal for techniques mentioned in Section 4.5, i.e., reduction of inter-application interference, is orthogonal to the optimization goals for this thesis and can be used along with our proposed techniques described in Chapters 5 and 6.

Chapter 5

Proposed Network Prioritization Techniques - I

In Chapters 1 and 2, we discussed some key issues and challenges in multi-core NoC-based systems. Chapter 4 presented some relevant research efforts in this area and their drawbacks. In this chapter, we address these drawbacks and propose a NoC framework with heterogeneous prioritization (HEPI). HEPI is comprised of a new dynamic ranking scheme for application classification, an application-aware prioritization algorithm and a memory-aware prioritization algorithm. After evaluating fairness issues in multi-programmed systems and the drawbacks of existing anti-starvation techniques (Section 2.2.2), we also apply a system-wide anti-starvation technique.

Our proposed heterogeneous prioritization (HEPI) framework is inspired by the insight that there is always a race among request packets originating at different cores to reach their destinations. The overall system throughput is impacted by which requests get served faster and allocation of NoC resources plays a significant role in determining this. Request/response packets also often interfere with each other at NoC routers to acquire an output channel. It is the responsibility of *router arbitration mechanisms* to determine which packets get access to an output channel at every cycle. In Round Robin (RR) based arbitration schemes that are widely used in NoCs, the arbitration is fair but packet criticality oblivious. Moreover, for off-chip memory requests, when packets reach the memory controller, a great deal of power and latency is expended in re-ordering these requests to optimally access memory in contemporary memory controller nodes. To overcome both of these limitations, we propose a novel heterogeneous two-stage prioritization technique for packets traversing the NoC. The first stage applies application-

aware prioritization, with packets being prioritized as per their criticality, which in turn depends on their latency sensitivity and MLP characteristics.

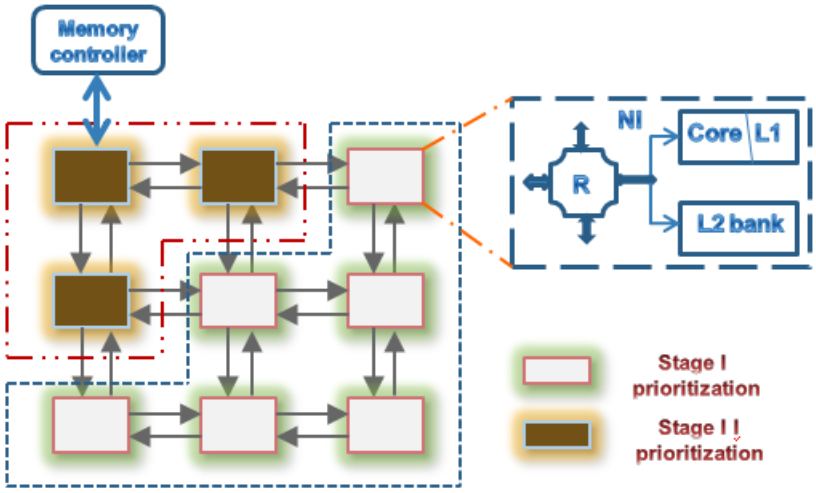


Figure 5.1: Proposed HEPI framework

The second stage applies memory-aware prioritization, with packets being reordered as per memory state to minimize stalls. Figure 5.1 shows how the two types of prioritizations are applied to NoC routers. The routers in close proximity (i.e., at one or less hop distance) from the memory controller employ the second stage prioritization mechanisms while all other routers, relatively farther from memory employ the first stage prioritization mechanisms. As we show later, such a framework can notably improve system throughput and memory access performance.

In the following subsections, we first describe a time based batch prioritization mechanism used in all NoC routers to prevent packet starvation, followed by details of the Stage I application- aware and Stage II memory-aware arbitration algorithms.

5.1 Time-based Batching

We employ a time-based batching technique that is loosely adapted from [16] to prevent starvation of packets in the NoC. The time-based batching technique is summarized in Figure 5.2.

Time-based batching scheme

Parameter Definitions:
β: number of cycles in a batching interval
K : Maximum Batching levels
CBID: Current Batch Injection ID counter
PBID: Packet's tagged Batch ID

Batch assignment for application packets at core network interface:
At the end of every β cycles-
1: CBID = CBID + 1
2: if (CBID >= K)
3: CBID = CBID % K //wraps around for compact ids
4: PBID = CBID

Relative batch priority of packet p at router during arbitration at cycle T:
RBP (p) = (CBID at T – PBID) % K

Figure 5.2: Time-based batching anti-starvation scheme

After each batching interval β, the packets originating in that interval from a network interface are assigned a batch id. This batch id is a timestamp that can be used in routers to determine how long the packet has spent in the NoC. The batch id counter is incremented at each batching interval until K levels, after which it starts over again (steps 1-3) to maintain compact ids. The network interface (NI) of each core tags a packets' header flit with a batch id before it is injected into the NoC. When any packet *p* arrives at a NoC router, a relative batch priority of *p* (RBP) is computed based on current batch injection id (CBID) and packet batch id (PBID) for *p*. This RBP value is used to include starvation information as part of the prioritization scheme in both stage I and stage II NoC routers as discussed next.

5.2 Stage I Application-aware Algorithm

This algorithm prioritizes packets in NoC routers as per their application-specific criticality and also ensures freedom from starvation. We use our novel ranking scheme to determine application-specific packet criticality and the time-based batching technique (Section 5.2.1) to prevent starvation of packets. The ranking scheme is described as follows.

5.2.1 Proposed Ranking Scheme

Different applications contribute differently to the overall system throughput. If an application requires a lot of its operands to be fetched from memory, it spends most of its time stalling and waiting for the requests to be satisfied by the main memory. Such applications cannot contribute much to the overall system throughput (defined as total number of instructions executed per unit time). We can thus classify such memory-intensive applications as *latency-tolerant* applications that do not impact performance notably even if their packets face a small delay. In contrast, there are some applications that have a very high CPU utilization due to their high compute intensity and that require very few operands to be fetched from memory over long periods. Such compute-intensive applications can be classified as *latency-critical*, as they are very crucial for the performance of the system, i.e., any additional wait time for a core executing this application can make the CPU stall which otherwise it could have spent executing instructions.

In our proposed criticality ranking scheme, applications (and their corresponding packets) are classified dynamically into four categories based on their latency criticality and MLP. The pseudo-code in Figure 5.3 summarizes the classification.

Dynamic application criticality ranking scheme

Counters at core's network interface:
cL1MPKI: counter for average L1 MPKI
cMLPIndex: counter for MLP index

Rank assignment for application packets at core network interface:
At the end of every τ cycles-
case ((cL1MPKI $\leq \tau_0$) AND (cMLPIndex $\leq \tau_1$)) : Rank 0
case ((cL1MPKI $\leq \tau_0$) AND (cMLPIndex $> \tau_1$)) : Rank 1
case ((cL1MPKI $> \tau_0$) AND (cMLPIndex $\leq \tau_1$)) : Rank 2
case ((cL1MPKI $> \tau_0$) AND (cMLPIndex $> \tau_1$)) : Rank 3

cL1MPKI = 0; cMLPIndex = 0 //reset counters

Figure 5.3: Application criticality ranking scheme

At each core's network interface, a count of L1 cache misses is kept to broadly classify an application as latency-critical or latency-tolerant. To perform fine-grain classification of applications as per their memory access patterns and MLP index exhibited, we also keep a count of entries in the L1 MSHR queue. These counters are queried for the purposes of dynamic classification and then reset at each application-profiling interval of τ cycles. At the profiling intervals of τ cycles**,** L1 miss counters and MLP index counters at a core's network interface are checked. If the L1 cache miss count is below a threshold τ_0 then that application is identified as latency-critical and further checked for its MLP index. Intuitively, in the latency-critical class, applications with MLP index lower than a certain threshold τ_1 are indicative of multiple row hit requests or high criticality for an application's compute phase, so we give them the highest priority i.e., rank 0. If a latency-critical application has MLP index greater than τ_1 then it comes immediately next in priority because its MLP is likely to enhance the performance of the system. Similarly, if the L1 cache miss count exceeds the threshold τ_0, then that application is put in the latency-tolerant category and further classified depending on its MLP capabilities as mentioned

for the latency-critical class. In this manner, our approach performs a fine-grain online classification that captures the changing dynamics of application execution over time. Figure 5.4 shows a flowchart for the abovementioned application ranking process. Here, T_0 represents τ_0 and T_1 represents τ_1.

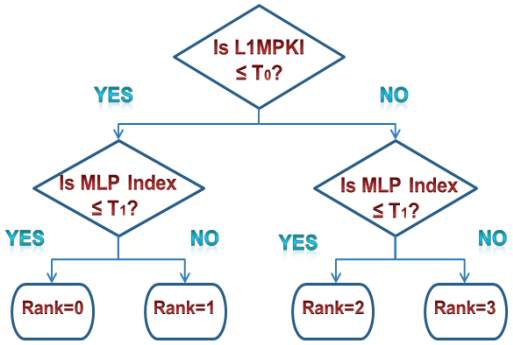


Figure 5.4: Flowchart for application ranking

The network interface (NI) of each core uses our proposed ranking scheme to compute application rank and then tags a packets' header flit with its rank before injecting it into the NoC. The pseudo-code in Figure 5.5 summarizes the Stage I algorithm.

Stage I Application-aware Algorithm

Parameter Definitions
O: set of NoC router output ports
I: set of NoC router input ports
MBP: Maximum Batch Priority *//priority of oldest batch*
PBP: Packet Batch Priority *//batch priority for current packet*

Prioritization of packets at stage I NoC router:

```
1:  for all o: o ∈ O do: {
2:    Initialize min_rank to 5; priority_port to -1; MBP to -K.
3:    for each i: i ∈ I do:  {
4:      if (packet p ∋ p at i ↦ o) {    // packet p at i going to o
5:        if (RBP (p) > MBP) {
6:          MBP = RBP (p)
7:          priority_port = i
8:        } else if (RBP (p) == MBP) {
9:          if (rank_p < min_rank) {
10:           min_rank = rank_p
11:           priority_port = i
12:         } else if (rank_p == min_rank) {
13:           priority_port = (type (p) == L2 packet)? i : priority_port
14:         } // if rank_p > min_rank, do nothing
15:       }// else if (RBP(p) <MBP), do nothing
16:     }
17:   }
18:   if (!(priority_port == -1)) {
19:     priority_port ← grant port o
20:   }
21: }
```

Figure 5.5: Stage I application-aware algorithm

For every output port *o*, the arbiter checks input ports to see if there are packets that need to go to port *o* (steps 3-4). For an input port *i* with a packet *p* destined for output port *o*, the RBP value of *p* is compared against the value stored in the Maximum Batch Priority (MBP) register (steps 5-15). If RBP for *p* is higher than MBP (step 5), it indicates a starved packet that must be given the highest priority, which is done by updating the *priority_port* register with *p*'s corresponding input port *i* (step 6). The value of MBP is also updated with the priority of the oldest batch (step 7). If RBP is equal to MBP (step 8), prioritization occurs as per packet

criticality (steps 9-14). The rank of p ($rank_p$) is first compared against the value stored in the *min_rank* register. If $rank_p$ is less than *min_rank*, the value in *min_rank* is updated and i becomes the priority port (steps 9-11). If $rank_p$ is equal to *min_rank* (step 12), then we have a case where the priority of two packets destined for o is the same. In this case, p is prioritised only if it is an L2 packet, as prioritizing such on-chip packets that can finish quickly over off-chip requests improves system throughput (step 13). Finally, if RBP is less than MBP (step 15), it indicates a situation where another packet set the MBP value and is more severely starved; therefore packet p at the current input port i is not given higher priority (i.e., *priority_port* register is not updated). Once all input ports have been checked, the arbiter grants the priority port access to the output port o (steps 18-20). These steps are performed in parallel in stage I NoC routers for all output ports at every cycle, to determine which packets from the input ports can proceed to their destination output ports.

5.3 Stage II Memory-aware algorithm

This algorithm prioritizes packets in NoC routers so as to avoid bank conflicts and encourage *row hits* and *bank level parallelism* (BLP). One of the major reasons that off-chip memory requests are delayed is due to stalls experienced as a result of bank conflicts. We observed that in addition to prioritizing applications in the network, system performance can be improved if the prioritized requests do not spend time stalling due to such bank conflicts. We thus propose the use of the arbitration inherent in NoC routers to prepare DRAM memory requests to proceed to the memory controller in a manner that prevents bank conflicts and relieves the memory controller from costly reordering of requests to minimize bank conflicts and improve row-buffer hits. We impart this memory-awareness to routers in the proximity of a

memory controller (Stage II routers; Figure 5.1). Much like the Stage I routers, the Stage II routers also use a time-based batching technique (Section 5.1) to prevent packet starvation. The pseudo-code in Figure 5.6 summarizes the Stage II algorithm.

Stage II routers share a small *recently used bank (RUB) table* with information about the main memory state that includes the *N* distinct recently used banks, their corresponding rows and a valid bit to indicate if the corresponding request has been processed. A valid bit value of 0 for an entry indicates that the bank is free whereas 1 indicates that the bank is busy servicing the request. For every output port *o*, the arbiter checks input ports to see if there are packets that need to go to port *o* (steps 3-4). For an input port *i* with a packet *p* destined for output port *o*, we first check if the packet has been starved by applying an RBP check as we did in the stage I prioritization algorithm (steps 5-7), to ensure that no packet suffers from starvation. If packets from the same batch are competing with each other (step 8), then memory packets (i.e. DRAM requests) and network packets (e.g., DRAM responses, shared cache requests/responses or cache coherence data) are handled separately. For a DRAM request, the router queries the RUB table and prioritizes packet *p* under the following conditions (steps 10-12): i) if the entry for *p*'s destination bank is absent in RUB; or ii) if *p*'s destination bank is present in RUB and matches the corresponding row (valid or invalid). The first condition enhances bank level parallelism by accessing multiple banks in parallel; the second condition indicates a row hit. If these conditions are not satisfied, then if a more critical packet (i.e. with lower rank than min_rank) is encountered, it is prioritized if the bank's RUB entry has been invalidated, which indicates that the bank is not busy (steps 13- 15). These conditions ensure freedom from stalls at the DRAM. The priority_port and min_rank registers are updated with the packets *i* and $rank_p$ values if the above conditions are satisfied.

Stage II Memory-aware Algorithm

Parameter Definitions
O: set of NoC router output ports
I: set of NoC router input ports

Prioritization of packets at stage II NoC router:

```
1:  for all o: o ∈ O do :{
2:     Initialize min_rank to 5; priority_port to -1; MBP to -K.
3:     for each i: i ∈ I do: {
4:      if (packet p ∋ p at i ↦ o) {              // packet p at i going to o
5:        if (RBP (p) > MBP) {
6:           MBP = RBP (p)
7:           priority_port = i
8:        } else if (RBP (p) == MBP) {
9:           if ((type (p) == DRAM request) {
10:            if ((dest.bank(p) ∉ RUB) ||      // bank level parallelism
                (dest.{bank, row}(p) ∈ RUB)) { //row hit
11:                 min_rank = rank_p
12:                 priority_port = i
13:            } else if ((dest.bank(p) ∈ RUB) && valid==0)
                  &&(rank_p < min_rank)) {    //bank conflict
14:                 min_rank = rank_p
15:                  priority_port = i
16:           } // else do nothing
17:           } else {                        // for network packets
18:             if (rank_p < min_rank) {             //packet criticality
19:                  min_rank = rank_p
20:                  priority_port = i
21:             } // if rank_p ≥ min_rank, do nothing
22:           }
23:       } // if RBP < MBP do nothing
24:     }
25:    }
26:    if (!(priority_port == -1)) {
27:       priority_port ← grant port o
28:    }
29:  }
```

Figure 5.6: Stage II memory-aware algorithm

If none of the above conditions are satisfied by memory packets at input ports, then the arbiter prioritizes network packets even if they have higher ranks (lower priority) than memory

packets. By de-prioritizing the bank conflict/row conflict memory packets in this manner, head of queue stalls at the memory controller are avoided. The network packets are prioritized based on their ranks (steps 17-20) compared with the value in min_rank. The input port of the highest priority (lowest rank) packet is stored in the priority_port register (step 20). After iterating through all input ports, the output port *o* is granted to the priority_port (steps 26-28). These steps are performed in parallel for all output ports in stage II routers at every cycle, to determine packets from input ports that can proceed to their output ports.

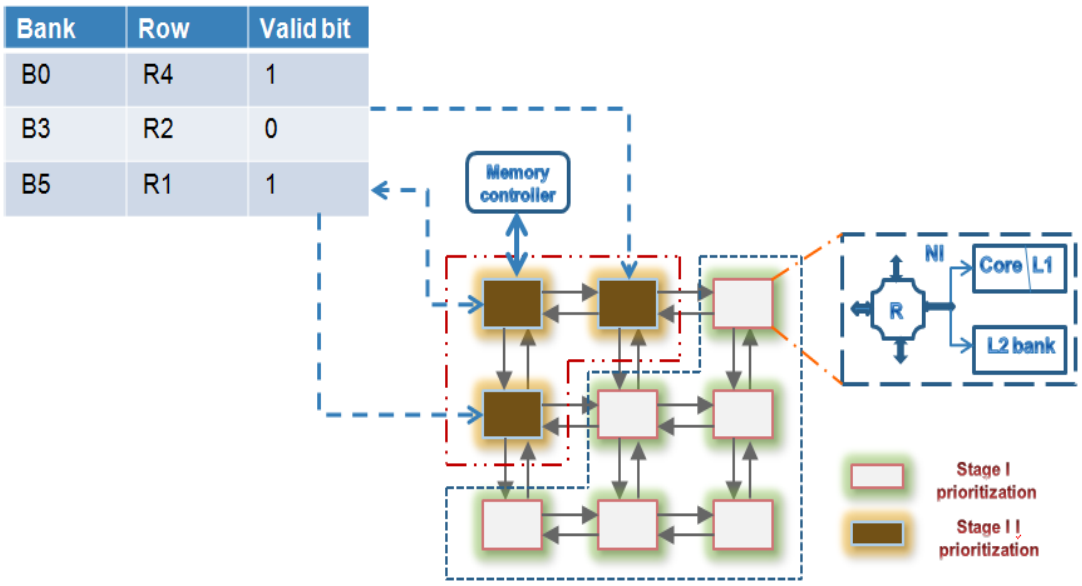


Figure 5.7: Representation of the RUB table

Figure 5.7 shows a representation of the RUB table. All stage II routers have read access to the shared RUB table; however, only the router directly connected to the memory controller node has write access to the table. When a DRAM request wins the arbitration in this router, the RUB table is updated with the request bank and row entries, and the valid bit for the entry is set to 1. If the bank entry for the winner is already present in the RUB table, the row entry is overwritten

and the valid bit is set to 1. The memory controller sends notifications to this router when a bank becomes free following which this router invalidates the bank entry by setting its valid bit to 0. If the table is full, the oldest bank entry with a valid bit of 0 is overwritten with the new request. The size of the table is kept small to allow for fast table lookup and avoid delays during arbitration.

The next section presents experiments performed to validate our hypotheses.

5.4 Experimental Setup

Evaluation Platform: We use the cycle-accurate event-driven GEM5 full system simulator [22] for validating our proposed prioritization framework. GEM5 is a full system simulator providing support for state-of-the-art out-of-order processor cores as well as models for a detailed NoC architecture, cache hierarchy and main memory subsystem [23]. We use the directory-based MESI coherence protocol as our default coherence protocol for the on-chip cache hierarchy. Table 1 shows the configuration of our baseline CMP consisting of a 3×3 concentrated mesh NoC with a concentration factor of four (i.e., four cores connected to each router) for a total of 36 cores on the die. We assume one-to-one application to core mapping with requests from a core allowed to access L2 cache banks at remote cores. Each NoC router has a state-of-the-art 5-stage pipelined implementation. For our baseline CMP, we use one memory controller connected to node zero and assume a NUMA-based system to support scalability. The baseline memory controller serves requests in a first come first served manner.

HEPI Implementation: For our proposed heterogeneous prioritization (HEPI) framework, the network interface for a core requires counters to keep track of a core's L1 misses and outstanding

requests in the MSHR queue at each ranking interval. We empirically choose the ranking interval as τ =100K cycles considering the tradeoff between ranking accuracy and its overhead. The network interface has the (straightforward) logic to compute ranks of applications over a ranking interval, as discussed in Section 5.2.1 and assigns a 2-bit rank to the header flit of each packet for prioritization in downstream routers. Unlike prior work such as STC [16] where at each predefined interval the L1 MPKI values of each core are communicated to a centralized decision logic (CDL) to rank applications, our approach is more scalable and possesses lower overhead as ranking decisions are made locally. Based on the application profiling done for SPEC2K and SPLASH2 benchmarks, the median value of average L1 MPKI was found to be 15 whereas MLP Index was found to be 3. Hence, we set the classification threshold τ_0 as 15 and classification threshold τ_1 as 3. The batching interval value (β) was empirically set to 16K cycles and the maximum batching levels (K) set to 8. Finally, for stage II routers, we chose the RUB table size (N) as 8 entries per rank. This table size is small enough to allow for fast lookup, while also providing sufficient visibility into recently used banks given that modern DRAMs typically possess eight banks per rank.

Table 5.1: Baseline CMP system configuration

CPU configuration	1 GHz; out of order
L1 Cache	I/D-cache 16 KB, 2-way, 2-cycle latency, cacheline 128B, 16 MSHRs
L2 Cache	Unified, 128kB bank shared, 4-way set associative, cacheline 128B
Main Memory	2GB, DDR2-667, 2 ranks/DIMM, 8 banks per rank, detailed memory model using open-row policy and row-interleaving
Router	5-stage virtual channel router; credit-based flow control, 4 VCs per port, 4 buffers/data virtual channel, 1 buffer/ctrl virtual channel
Network Topology	3×3 2D concentrated mesh, concentration factor 4
Routing scheme	Deterministic X-Y

Comparison with Prior Work: We modeled our proposed HEPI approach as well as the best known prior works on NoC and memory prioritization for comparison. The prior works and configurations we compare against include: 1) a baseline localized Round Robin technique (Local RR) that uses a fair round-robin algorithm in all NoC routers; 2) SDRAM-aware router [13] that uses SDRAM-specific timing parameters to determine delay and priority of an off-chip memory request at specific SDRAM-aware routers (SDAR); and 3) Stall Time Criticality (STC) [16] that employs application-aware prioritization based on batching and application ranking as per L1 MPKI at all the NoC routers. The parameters used for STC are 1) batching interval=16000 cycles; 2) ranking interval=350K cycles 3) batching as well as ranking levels=8. For SDRAM-Aware, we use DDR2-667 timing parameters from Micron datasheets [24] and replace three conventional NoC routers in the vicinity of the memory controller with SDRAM-aware routers. These prior techniques have used a trace-based simulator for their evaluation purposes while we use an event-driven simulator with detailed micro-architecture models. All the simulations in our studies were run for at least 180M instructions.

Workloads: We profile 17 diverse applications from the SPLASH-2 and SPEC2K benchmark suites. We use different combinations of these applications to form a multi-programmed workload environment to compare our approach with the baseline system and prior work. We profiled all of the applications and divided them up into two categories based on their compute- and memory-intensity as shown in Table 5.3.

Following table shows the average L1 MPKI of each of the applications profiled from SPLASH2 and SPEC2K benchmark suite. As shown in Table 5.2, *gcc* is the most memory-intensive application amongst above mentioned benchmarks while *radix* is the most compute-intensive benchmark. However, these benchmarks have different degrees of Memory Level Parallelism (temporal locality or spatial locality) at different time intervals. Hence, to demonstrate that two compute-intensive or memory-intensive benchmarks can be totally different in their MLP characteristics, we captured the maximum MLP degree exhibited by any benchmark in a window of 10M cycles. As shown in Table 5.2, *barnes* and *lucas* are both compute-intensive benchmarks (low L1 misses per instruction), however, *lucas* exhibits more spatial locality than *barnes* in a given window of cycles.

We use these benchmarks with multiple combinations to form a multi-programmed workload. We form three categories of multi-programmed workloads- heterogeneous workloads i.e. memory-intensive as well as compute-intensive workloads co-running on a system (case I), homogeneous workload systems comprising all compute-intensive workloads (case II) and all memory-intensive workloads co-running on a multi-core system (case III). We perform experiments for each of these workload cases. Table 5.4 shows how various benchmarks are combined to create the three workload cases. The number next to a benchmark corresponds to its parallelization degree (number of cores it runs on).

Table 5.2: Average L1 MPKI and highest MLP degree exhibited for SPLASH-2 and SPEC2K benchmarks

Benchmark	Average L1 MPKI	Highest MLP exhibited
Radix	0.7296	2
Fft	1.104	3
Swim	1.2605	3
Barnes	2.3959	2
Lucas	6.7095	6
Fmm	8.1248	4
Crafty	10.8114	4
Sixtrack	13.7612	3
Equake	16.1462	5
Ocean	16.7009	4
Galgel	19.1546	7
Ammp	21.2576	4
Mgrid	26.9914	2
Apsi	31.265	2
Gap	38.4858	2
Gcc	47.282	5

Table 5.3: Classification of SPLASH-2 and SPEC2K applications

Compute-intensive	Memory-intensive
ammp, apsi, gap, galgel, gcc, mgrid, ocean	barnes, crafty, fft, fmm, lucas, radix, sixtrack, swim, equake, applu

Table 5.4: Three workload categories for a 36-core CMP

Heterogeneous Workloads (Case I)	ocean(4), gcc(8), apsi(6), swim(8), equake(10)
	ammp(5), apsi(9), radix(10), gap(8),lucas(4)
	crafty(7), gap(9), applu(11), gcc(4), barnes(5)
Compute-Intensive Workloads (Case II)	lucas(12),barnes(12), radix(12)
	fft(9), swim(9), barnes(9), fmm(9)
	equake(12), crafty(12), applu(12)
Memory-Intensive Workloads (Case III)	ocean(9), apsi(9), mgrid(9), gcc(9)
	gcc(16), ammp(8), galgel(12)
	apsi(9), gap(9), mgrid(9), gcc(9)

Evaluation Metrics: We focus on the following performance related metrics:

- **Overall System Throughput**

  We measure overall system throughput as the number of instructions executed in the entire system over duration of certain number of cycles. It represents how the overall system performs with different priority algorithms implemented in terms of latency for a fixed number of instructions.

- **Weighted Speedup**

  This metric is system-level and has significance in a multi-programmed environment. It is defined as:

$$Weighted\ Speedup\ = \sum_{i} (\text{IPC}_{\text{shared}} / \text{IPC}_{\text{alone}})$$

  over all applications, where $\text{IPC}_{\text{shared}}$ and $\text{IPC}_{\text{alone}}$ are the instructions executed per cycle (IPC) when executing multiple workloads and single workloads, respectively. This metric is useful in that it captures inter-application interference.

- **Average Memory Latency**

  We capture average memory latency for packets from the memory controller to the main memory and back. This gives us an estimate of how the prioritization techniques handle bank conflicts and data contention occurring at main memory.

5.5 Experimental Results

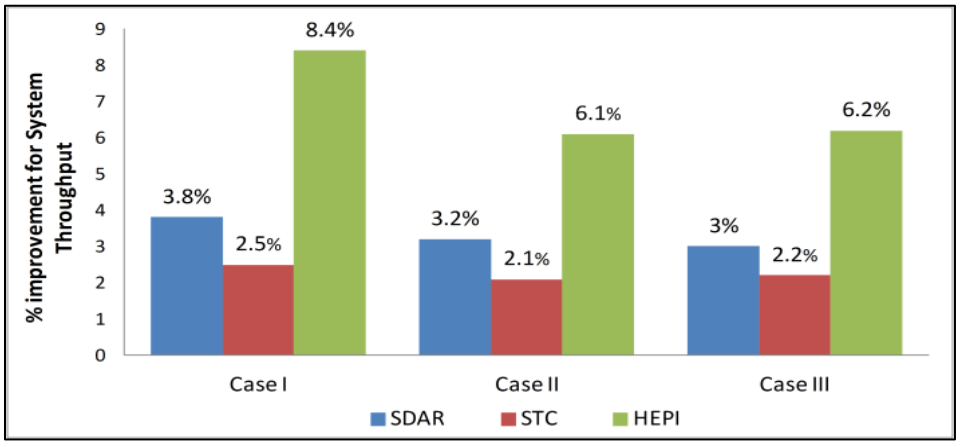
This section presents experimental results that compare our proposed HEPI framework with the following prioritization techniques: 1) baseline Local RR; 2) SDAR [13]; and 3) STC [16]. Modern CMP systems are often multi-programmed, executing either heterogeneous workloads (a mix of compute- and memory-intensive) or homogeneous workloads (only compute-intensive or only memory-intensive).

In the following section (Section 5.5.1), results for system throughput, memory latency and weighted speedup for the comparison against prior prioritization frameworks are presented across the three workload categories. We explore the impact of utilizing a high speed memory (DDR3-1333) for various prioritization techniques in Section 5.5.2. In Section 5.5.3, we also show results for the scalability of our framework for three different network sizes with varying number of cores. Finally, we compare the overhead of our proposed approach and other techniques in Section 5.5.4.

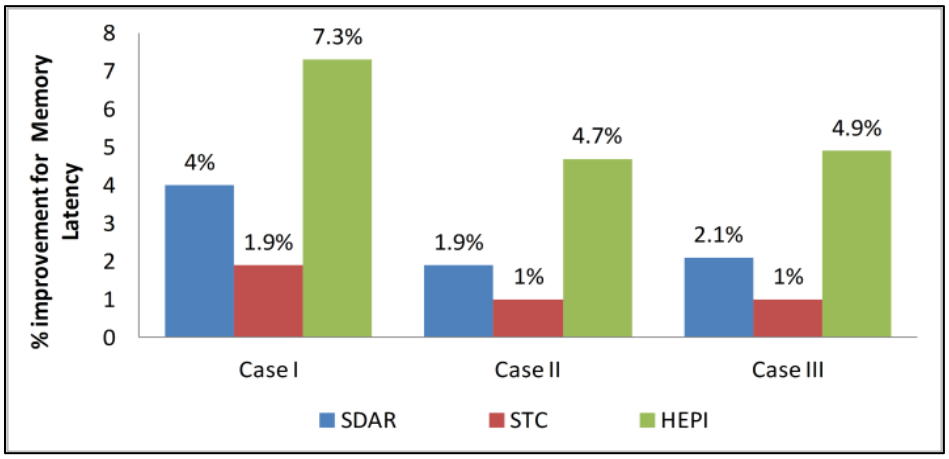
5.5.1 Performance Comparison Across Workload Types

Figure 5.8 shows the comparison between HEPI and other techniques, with averaged results shown for the three workload cases shown in Table 5.5. The numbers over the bars in the figures indicate percentage improvement of the techniques over the baseline Local RR technique.

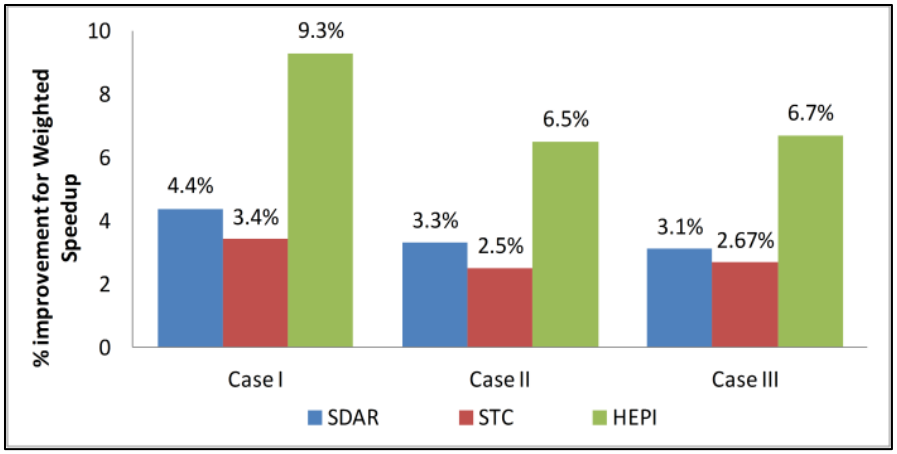
For case I heterogeneous workloads, HEPI shows an improvement over the baseline/SDAR/STC techniques of 8.4%/4.6%/5.9% for system throughput, 7.3%/3.3%/5.4% for memory latency and 9.3%/4.9%/6.1% for weighted speedup, motivating the need for packet prioritization over fairness. The SDAR technique supports memory-aware packet prioritization closer to the memory controller but is oblivious to application-specific packet criticality. Hence, it does not contribute towards improving the system throughput as effectively as the multi-stage HEPI technique does.



(a)



(b)



(c)

Figure 5.8: % Improvement over Local RR: (a) system throughput, (b) average memory latency, and (c) weighted speedup

In contrast, STC enables application-specific packet prioritization but relies on L1 MPKI only to determine criticality of a packet and does not consider its memory access characteristics. Also, it is inefficient in handling DRAM requests as it is memory-unaware. Hence, it is outperformed by HEPI which not only exploits the unique characteristics of each application to determine packet criticality and better prioritize them upstream in the network via stage I routers, but also maximizes row hits and prevents bank-conflicts downstream in the network (closer to the memory) via stage II routers.

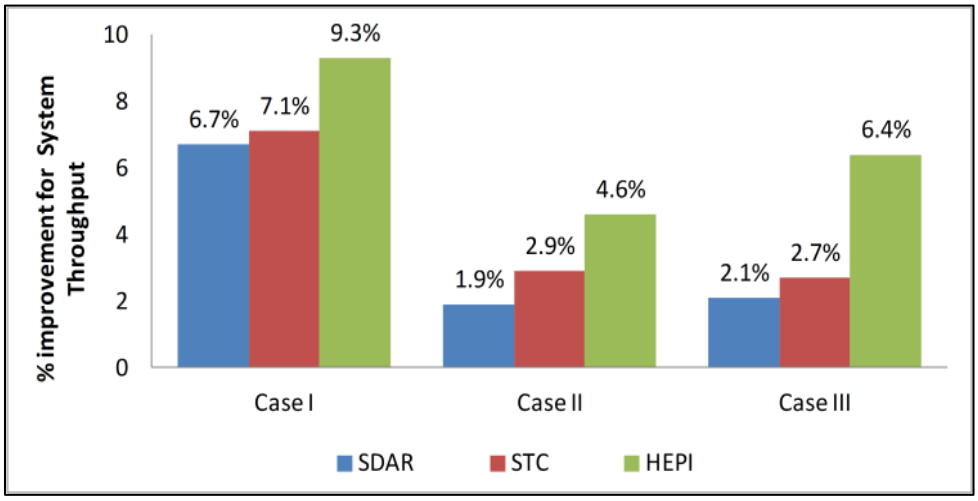
For case II compute-intensive workloads, we can observe that HEPI again outperforms the other techniques, but by a smaller magnitude than for heterogeneous workloads. HEPI shows an average improvement of 6.1%/2.9%/3.9% for system throughput, 4.7%2.8%/3.7% for memory latency and 6.5%/ 3.2%/4% for weighted speedup over the baseline/SDAR/STC techniques. The smaller magnitude of improvement for HEPI with compute-intensive workloads in contrast to heterogeneous workloads is because for compute intensive workloads, network traffic is much lower, which reduces opportunities for applying prioritization and consequently diminishes the

benefits of using the HEPI technique. Moreover, as all the workloads have similar behavior, there is little scope for distinguishing between applications in this case. Finally, for case III memory-intensive workloads, we see an average improvement of 6.2%/3.2%/4% for system throughput, 4.9%/2.8%/3.9% for memory latency and 6.7%/3.6%/4.1% for weighted speedup over the baseline/SDAR/STC techniques. Intuitively, with higher network traffic under memory-intensive workloads, the magnitude of improvement for HEPI is also slightly larger than for compute-intensive case II workloads with low network traffic. The improvements for case III workloads are, however, lower than that for case I workloads. This is because, case I workloads are heterogeneous and thus, there is more opportunity to identify packets from different types of applications, rank them and exploit stage I routers to benefit criticality of applications; which is not as effective with homogeneous case III workloads.

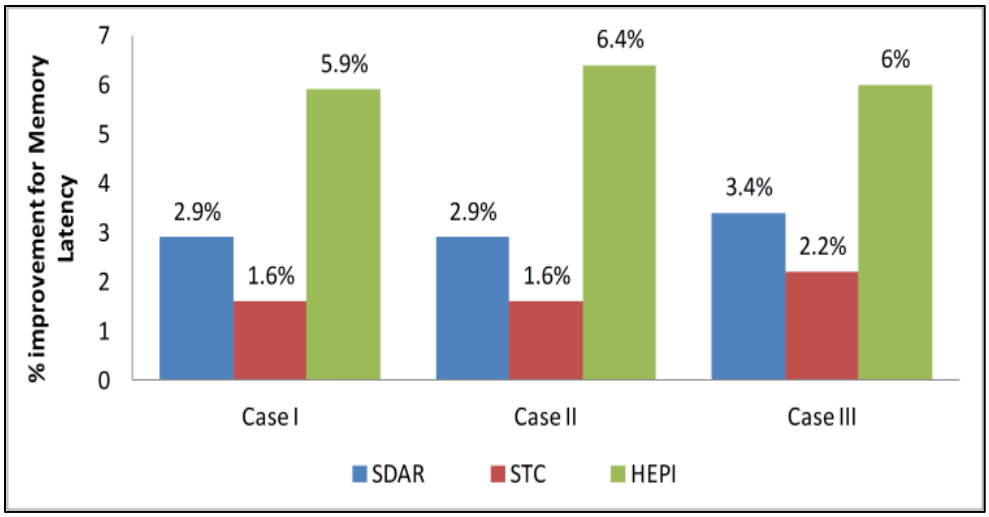
5.5.2 Performance with Different Main Memory

Next, we were interested in understanding the impact of changing the main memory type and observing if the performance benefits of HEPI still hold with respect to the other prioritization techniques. We modeled the high speed DDR3-1333 with parameters taken from Micron datasheets [26] and used it to replace the baseline DDR2-667 main memory model [24] in our simulation framework. Figure 5.9 shows the results for system throughput, average memory latency and weighted speedup for the high speed DDR3 memory. The results show the percentage improvement of the techniques over the baseline Local RR scheme and are averaged for the three workload cases in Table 5.5. We observed that the improvements with HEPI were more pronounced when using the high speed DDR3 memory, than with the DDR2 memory. This is because for high speed memories with a faster clock frequency, bank conflicts can become a

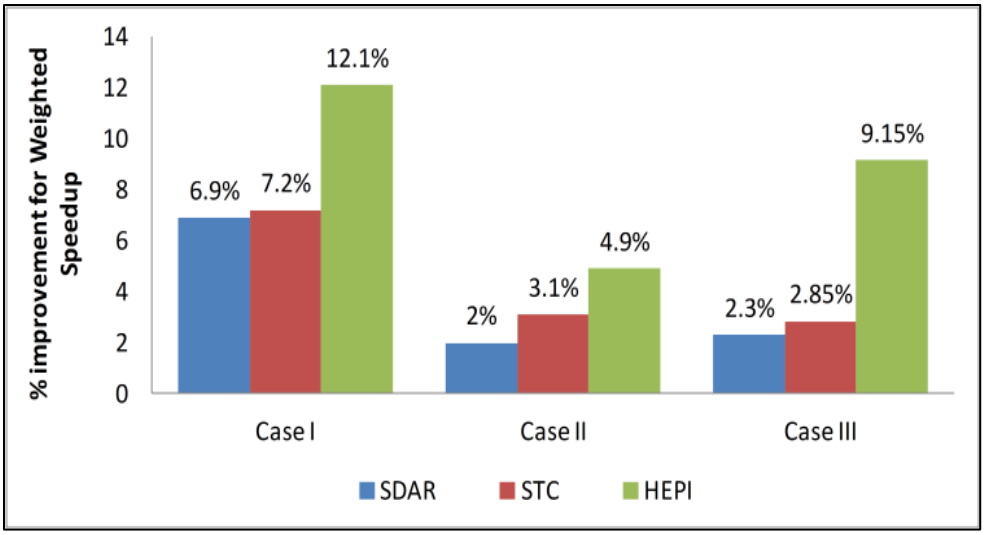
major bottleneck due to larger number of pre-charge and activate cycles. The stage II prioritization algorithm in HEPI is able to better route requests to the main memory than other techniques, allowing prioritization of requests such that accesses to free banks and row hits are maximized. The better performance for the stage II prioritization in HEPI is the main reason that it achieves greater improvements with DDR3 than with DDR2.



(a)



(b)

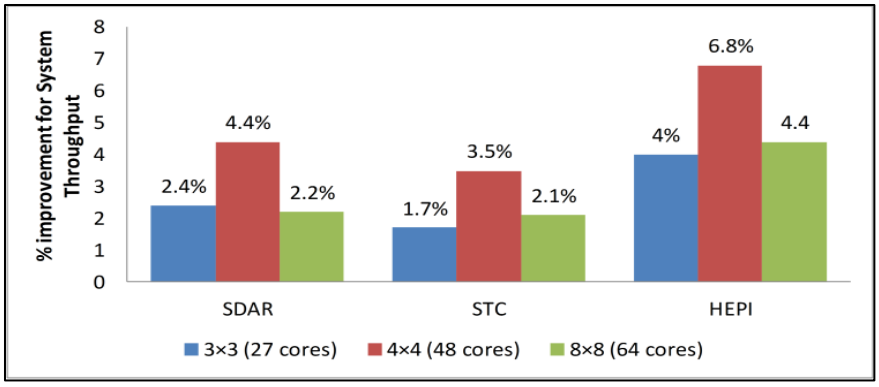


(c)

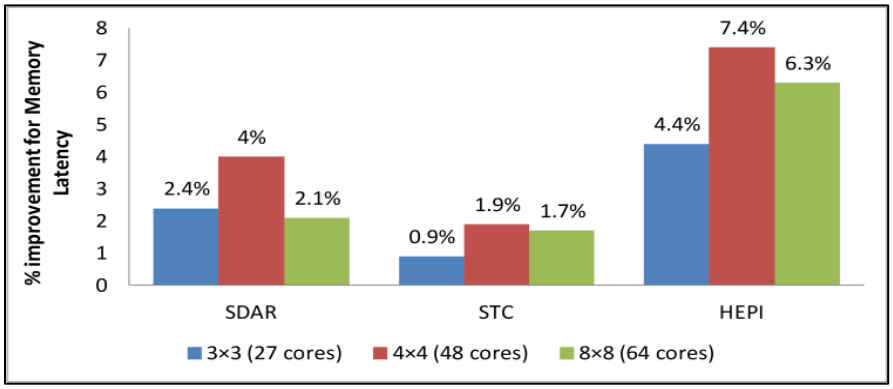
Figure 5.9: % Improvement over Local RR for DDR3-1333: system throughput, (b) average memory latency, and (c) weighted speedup

5.5.3 Platform Scalability Analysis

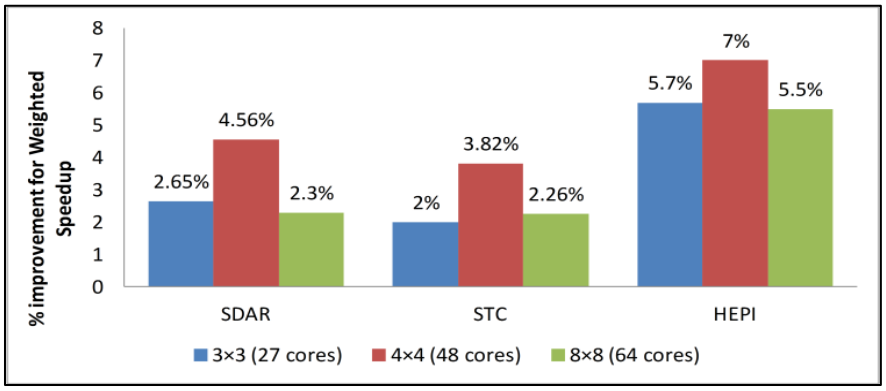
In this set of experiments, we were interested in observing the impact of scaling the platform complexity (mesh size, core count, memory controller count) on the performance of HEPI and other prioritization techniques. We explored performance for three different network sizes: i) 3×3 mesh with a concentration factor of three (27 cores) and one memory controller, ii) 4×4 mesh with concentration factor of three (48 cores) and one memory controller; and iii) 8×8 mesh with concentration factor of one (64 cores) with two memory controllers on diagonally opposite sides. Figure 5.10 shows the percentage improvement for system throughput, average memory latency and weighted speedup over the baseline Local RR technique, for the three different platform complexities.



(a)



(b)



(c)

Figure 5.10: % Improvement over Local RR for different platform complexities: (a) system throughput, (b) average memory latency, and (c) weighted speedup

The results are averaged over the three heterogeneous workload configurations in case I

from Table 5.5 (results for case II and case III workloads showed similar trends). It can be observed that HEPI scales well with increasing core counts, network complexity and memory controller complexity, showing notable improvements over other techniques. With higher network traffic and more opportunity to prioritize requests, the results are more pronounced. Hence, we get the best improvements for the 4×4 configuration with 48 cores as it has the most NoC traffic.

5.5.4 Overhead Analysis

This final set of results attempts to quantify and compare the overheads of implementing the HEPI prioritization technique with the implementation overhead for other prioritization techniques. Figure 5.11 shows the overall NoC fabric area for the baseline Local RR, SDAR, STC and HEPI frameworks for the 32nm CMOS process technology node and the 36 core CMP configuration. Not surprisingly, the baseline technique having a NoC router with a simple round robin prioritization mechanism has the lowest area overhead. For the SDAR technique, the SDRAM-Aware NoC router has memory-specific information tables at each SDRAM-aware router and also requires buffers to store complete addresses of the previous arbitration winners. These additional components drive up its area overhead compared to the baseline case. STC routers require buffers to store L1 MPKI information for all the cores and central decision logic to calculate ranks for each application dynamically, hence this technique has a very high area overhead. In contrast, HEPI has a lower area overhead than for STC and SDAR as it performs ranking and application-centric prioritization locally with simple circuitry in stage I. HEPI stage II routers have low overhead as they require only a small RUB table shared by multiple routers and minimal circuitry to query/update the table.

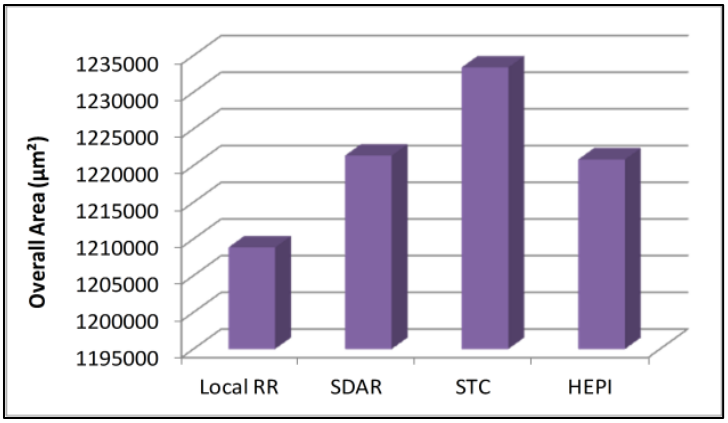


Figure 5.11: Overall area estimation

Thus, our experimental results show that the proposed HEPI framework outperforms fair prioritization techniques by up to 12.6% as well as other application-specific techniques from prior work by up to 6.9% for various multi-programmed workloads. HEPI also shows consistent improvements with increasing platform complexity. Furthermore, the benefits achievable with HEPI are improved when emerging high speed memories are utilized. Given its competitive area overhead compared to other competing prioritization techniques, HEPI provides an attractive alternative for inclusion in emerging CMPs with multi-programmed workloads.

Chapter 6

Proposed Network Prioritization Techniques-II

Chapter 5 presented an application-aware and a memory-aware packet prioritization algorithm. The memory-aware algorithm prioritizes packets to enhance bank level parallelism and consequently memory throughput. However, as discussed in Chapter 2, high-speed memories place a constraint on BLP degree of any memory device to avoid power spikes. We therefore enhanced our work presented in Chapter 5 to create an improved memory-aware algorithm for high-speed memories considering this constraint. The architectural proposition of heterogeneous prioritization framework is retained but the framework presented in this chapter employs a different memory-aware prioritization technique and a new anti-starvation technique. We will refer to the framework described in this chapter as the Dual Scheduled Packet Framework (DSPK).

DSPK is composed of two main components: *(i)* a packet classification technique that is application-aware at the first level and memory-aware at the second level; and *(ii)* an anti-starvation technique employed system-wide to prevent unfairness in the system due to strict priority-based scheduling. Details of our framework are presented next.

6.1 Anti-starvation Technique

On-chip L2 cache request packets have a lower overall latency than off-chip memory packets. If the on-chip cache request packets encounter a hit, they can contribute to overall performance (system throughput) to a much greater extent than packets going off-chip. However, when one application is ranked higher than the other, there are instances when L2 packets of

lower ranked applications consistently lose out to off-chip memory requests of higher ranked applications. Such L2 packets will get a "starved" status only after a significant amount of time has elapsed (corresponding to the batch interval size) if the batching strategies – time-based batching or packet-based batching described in Chapter 2 are employed. We conjecture that there should be different starvation criteria for L2 requests and off-chip requests. We, therefore, propose an ***anti-starvation technique*** that decides the "starved" status of a packet depending on not only how long it has been delayed but also depending on its destination. The network interface tags a packet with *initial request time* as soon as it is injected in the NoC. In the switch allocation stage at each NoC router, *current delay* faced by the packet is calculated by subtracting its *initial request time* from current time. We set separate thresholds for L2 and off-chip requests in each router that are calculated based on the running average of *current delay* values for the L2 and off-chip destined packets seen by a router in the last N cycles. The pseudo code in Figure 6.1 describes our threshold determination process. After keeping track of average L2 request and memory request delays for N cycles, thresholds for L2 packets (α_0) and memory packets (α_1) are stored at each router to determine starvation conditions. If the *current delay* of a network or memory packet exceeds the relevant threshold at a router, then the packet is considered *starved* and immediately granted the output channel over other higher ranked packets. Figure 6.2 describes identification of starved packets. The next subsection discusses how we classify and rank packets.

Anti-starvation Strategy Threshold Computation

Parameter Definitions:
Set of all packets: P
Set of all routers: R
α_0 : Threshold for L2 cache requests
α_1 : Threshold for memory requests

Compute average packet delay at each r:
∀ r: r ∈ R
∀ p: p ∈ P at r **do**:
While (current_cycle != N) **do**:
    if (type (p) == L2 cache request) **then**
        l2_lat += (current_cycle – initial_request_cycle)
        l2_pkt ++
    else //if p is a memory request)
        mem_lat += (current_cycle – initial_request_cycle)
        mem_pkt++
    end if
end while

average_delay_0 = l2_lat/l2_pkt // Average L2 delays
average_delay_1 = mem_lat/mem_pkt // Average memory delays

α_0 = (average_delay_0 * 1.5)
α_1 = (average_delay_1 * 1.5)

Figure 6.1: Pseudo-code for threshold determination in anti-starvation technique

Anti-starvation Strategy

Apply the following anti-starvation check at the router:

∀ r: r ∈ R **do**:
∀ p: p ∈ P at r **do**:
Case based on **type(p)**
  Case DRAM request:
    if (delay (p) ≥ α_1) **then**
      packet p is starved
    end if
  Case L2 request:
    if (delay (p) ≥ α_0) **then**
      packet p is starved
    end if
  Default:
    packet is not starved
End Case

Figure 6.2: Pseudo code for anti-starvation technique

6.2 Application- and Memory-aware Packet Scheduling

Applications are frequently classified based on their memory access behavior either as *memory-intensive* if they spend most of their execution time communicating with memory or as *compute-intensive* if they spend most of their time performing computations and not accessing memory as often. Intuitively, in scenarios with multiple co-running applications, the *compute-intensive* applications contribute to a greater extent towards system throughput (i.e., total instructions executed per cycle across all cores in the system). Therefore, packets from these applications should be given higher priority during scheduling. But, it is also important to consider the varying degrees of memory-intensiveness and memory level parallelism (MLP) exhibited by different applications over their lifetime. To capture these unique application-specific characteristics while assigning ranking (priority) weights to packets from applications, we propose using a two-stage approach.

In the *first stage*, we dynamically measure the memory-intensiveness of an application at

runtime using the metric average L1 misses per Kilo Instructions (L1MPKI). If the average L1MPKI over a time window is less than a threshold T_{S1}, it can be categorized as a compute-intensive application whose packets deserve higher priority (weightage). On the other hand, if average L1MPKI is less than the threshold, the application is labeled as memory-intensive and its packets assigned lower weights. In the *second stage* of classification, we consider the memory level parallelism (MLP) capabilities of an application to assign its final weights. We measure the MLP of an application using the outstanding count in the request queues of L1 MSHRs (Miss Status Handling Registers) [20]. The length of the MSHR queue has been shown to be directly proportional to MLP exhibited by the application [21]. If this length is less than a certain threshold T_{S2}, the application has lower MLP and is more critical. Thus, applications with low L1MPKI and low MLP are assigned the highest weight (3 on a scale of 0-3). This is followed by applications with low L1MPKI and higher MLP being assigned a weight of 2. Memory-intensive applications with low MLP are assigned a weight of 1 and the applications in the remaining categories are assigned the lowest weight of 0. Counters at each nodes network interface keep track of average L1MPKI of the application running at that node and the length of the MSHR queue over a time interval. The counters are reset and weights are assigned at the start of every new interval (every 100K cycles). Header flits of packets injected into the NoC have a 2-bit field that contains the assigned weight.

The application-aware strategy is presented in Figure 6.3. At each arbitration cycle, the router iterates through input ports to find a suitable request to be granted output channel. While iterating through input port i, if there is a packet p such that it desires output channel o (step 6), then the packet p should satisfy certain constraints in order to gain priority over packets at other input ports. If packet p is starving i.e. p being a L2 request packet exceeds threshold α_0 or p being

a memory request exceeds threshold α_1, then it is immediately granted channel o because a starving packet takes priority over other packets (steps 8 to 12). If the packet is not starved, the priority is given on the basis on weights assigned to the packets.

DSPK I Application-aware Algorithm

Parameter Definitions:
O: set of NoC router output ports
I: set of NoC router input ports

Prioritization of packets at stage I NoC router:

```
1:  for each o ∈ O do: {
2:    max_weight = -1        // initialization value
3:    priority_port = -1     //initialization value
5:    for each i ∈ I do:  {
6:     if (packet p ∋ p at i ↦ o) {    // packet p at i going to o
8:       if ((type (p) == L2 cache request)
           && (delay (p) ≥ α0){
9:          i ← grant port o
10:      } else if ((type (p) == memory request)
           && (delay (p) ≥ α1)) {
11:         i ← grant port o
12:      } else {                 // no packet is starving
13:        if (weightp > max_weight) {
14:          max_weight = weightp
15:          priority_port = i
16:        } else if (weightp == max_weight) {
17:          priority_port = (type (p) == L2 request) ? i : priority_port
18:        } // if weightp < max_weight, do nothing
19:      }
20:    }
21:  }
22:  if (!(priority_port == -1)) {
23:      priority_port ← grant port o
24:  }
25: }
```

Figure 6.3: Pseudo code for DSPK-I application-aware technique

The value for priority_port stored in register is to determine the highest priority port. If the weight of packet p ($weight_p$) is greater than value stored in max_weight register, then the value

of max_weight and priority_port are updated. If the weights are equal then the priority_port is updated only if p is a L2 request packet since on-chip cache request packets have a faster turnaround time. Finally, the port with highest priority (priority_port) wins the arbitration. We implement this two-stage weighing based classification at the NoC routers two or more hops away from the memory controller. This technique at this level is referred to as ***DSPK-I***.

Most of the network packets are handled by DSPK-I. However, the request packets that miss on-chip caches and must traverse the NoC again to go to off-chip memory must be treated differently, to prepare them against bank conflicts and row misses. Hence, a second-level scheduling algorithm must be applied on these memory packets. This algorithm is employed at NoC routers that are a single hop away from the memory controller. The technique at this level is referred to as ***DSPK-II***. The primary focus here is to schedule memory packets to exploit bank level parallelism and maximize row hits while taking into account the constraints in modern high speed memories.

The pseudo-code in Figure 6.4 summarizes DSPK-II scheduling algorithm. DSPK-II NoC routers maintain a Used Bank Row (UBR) table for each rank. This table maintains the bank and row history for a maximum of N previous requests for each rank. It has a valid bit associated with each entry to know if the bank is still processing the request. In addition, we also need to consider that no more than four banks can be activated in a rank in a window of T cycles for modern DDR3 memories (as discussed in Chapter 2). We employ n-bit counters for each logical rank at DSPK-II routers to count up to T cycles. The value of T is set to tFAW (a maximum 5-bit value specified in DDR3 datasheets) when the number of UBR table entries for any rank becomes equal to four. The value of this tFAW counter is decremented every cycle until it becomes 0 to indicate that BLP can be exploited (step 2). For a packet p at input port i

contending for output port o, starvation check is applied and granted channel immediately

DSPK II Memory-aware Algorithm

Initialization of parameters:
Set of Output ports: O
Set of Input ports: I
ctFaw: counters for counting tFaw cycles
Prioritization of packets at stage II routers:

```
1:  for all o ∈ O do:{
2:      if (ctFaw !=0) ctFaw --
3:      priority_port = -1; max_weight = -1          //initialization
4:      for each i ∈ I do: {
5:        if (packet p ∋ p at i ↦ o) {          //p at i maps to o
6:          if (type (p) == DRAM request) {
7:            if ((delay (p) ≥ α1)) {
8:              i ← grant port o
9:            } else {
10:                 if (((dest.bank(p) ∉ UBR) &&
                        ((sizeof (UBR) ≤ 4) || (ctFaw == 0)))          //BLP
                      || (dest.{bank, row}(p) ∈ UBR)                   //row hit
                    || ((dest.bank(p) ∈ UBR) && valid==0)  &&
                      (weight_p > max_weight))) {                      //bank conflict
11:                     priority_port = i
12:                     max_weight = weight_p
13:           } // else do nothing
14:         } else {                    // for network packets
15:           if ((delay (p) ≥ α0)) {
16:             i ← grant port o
17:           } else {
18:             if (weight_p > max_weight) {          //packet criticality
19:                 priority_port = i
20:                 max_weight = weight_p
21:             } // if max_weight ≥ weight_p , do nothing
22:           }
23:         }
24:       }
25:   }
26:   if (!(priority_port == -1)) {
27:     priority_port ← grant port o
28:   }
29: }
```

Figure 6.4: Pseudo code for DSPK-II memory-aware technique

if identified as starved (steps 7-8, 15-16). When memory requests are contending at a DSPK-II

router for an output channel towards the memory controller, entries in the UBR table are checked (step 10). If the entry is absent and if the BLP degree is not maximized (i.e. tFAW constraint applicable), then the packet is prioritized (priority_port and max_weight registers updated with values *i* and $weight_p$) to benefit from bank level parallelism (step 11-12). Packet is also prioritized if there is a possibility of row hit i.e. the bank entry and row entry matches or if a more critical DRAM request has invalid entry in UBR table. Otherwise, the bank is declared as *busy* and the packet is held at the router granting the output channel, with preference given to other network packets with applicable check for weights (steps 18-21). This prevents head of the queue stalls at the memory controller due to bank busy requests.

Figure 6.5 summarizes how our framework is applied to a NoC-based CMP. The figure shows a 3×3 mesh NoC with a concentration degree of two (i.e., two cores/router) and how our proposed DSPK-I and DSPK-II scheduling techniques are applied to various NoC routers. In addition, the proposed anti-starvation technique is applied to all NoC routers. Together, these techniques constitute a holistic framework for addressing the scheduling challenges facing modern NoC-based multi-core systems.

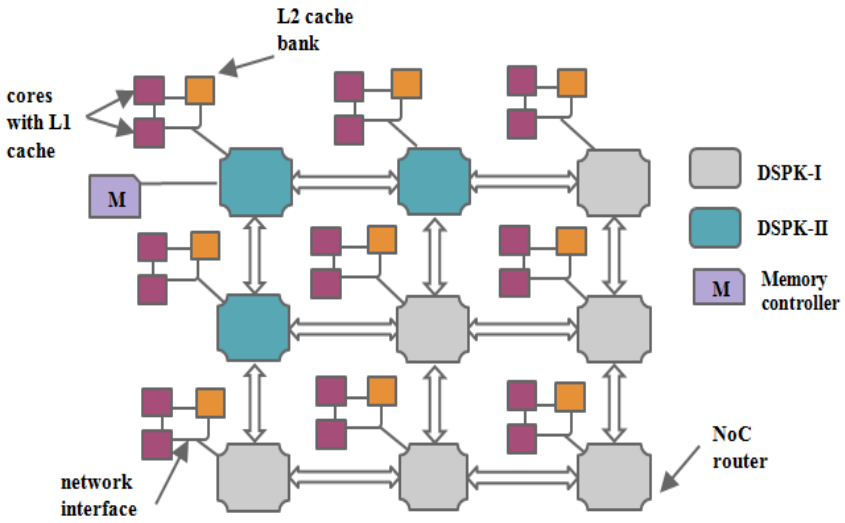


Figure 6.5: Proposed framework applied to a NoC-based CMP

6.3 Experimental Setup

We use the cycle-accurate event-driven GEM5 simulator [22] for validating our proposed packet scheduling framework. GEM5 is a full system simulator providing support for state-of-the-art out-of-order cores as well as models for a detailed NoC architecture, cache hierarchy and main memory subsystem [23]. We use the directory-based MESI coherence protocol as our default coherence protocol for the on-chip cache hierarchy. Table 6.1 shows the configuration of our baseline CMP consisting of a 3×3 concentrated mesh NoC with a concentration factor of three (i.e., three cores connected to each router) for a total of 27 cores on the die. We assume one-to-one application to core mapping with requests from a core allowed to access L2 cache banks at a remote core. Each NoC router has a state-of-the-art 5-stage pipelined implementation. For our baseline CMP, we use one memory controller connected to node 0 and assume a NUMA configuration to support scalability. The default memory controller services requests in a first come first served manner.

Table 6.1: Baseline CMP configuration

CPU	1 GHz; out of order;128 instruction queue
L1 Cache	I/D-cache 16 KB, 2-way, 2-cycle latency, cacheline 128B, 16 MSHRs
L2 Cache	Unified, 128kB bank shared, 4-way set associative, cacheline 128B
Main Memory	2GB, DDR3-1333, 2 ranks/DIMM, 8 banks per rank, detailed memory model using open-row policy and row-interleaving
NoC Router	5-stage virtual channel router; credit-based flow control, 4 VCs per port, 4 buffers/data virtual channel, 1 buffer/control virtual channel
NoC Topology	3×3 2D concentrated mesh, concentration factor 3
Routing scheme	Deterministic X-Y

For our proposed framework, we empirically set the value of threshold T_{S1} as 15 and T_{S2} as 3 in DSPK-I routers. For DSPK-II routers, we set a UBR table size of 8. For the anti-starvation technique, we keep a track of packets arriving at each router over a window of 4096 cycles. We modeled our proposed approach as well as the best-known prior works on NoC and memory scheduling for comparison. The prior works and configurations we compare against are: 1) a baseline technique (Round-Robin) that uses a fair round-robin scheduling algorithm in all NoC routers and is widely used in CMPs today; 2) a memory-aware technique [13] that uses SDRAM-specific timing parameters to determine delay and priority of an off-chip memory request at specific routers (MAT); and 3) an application-aware technique (AAT) [16] that employs application-aware scheduling based on batching and application ranking (using L1MPKI values) at all the NoC routers. The parameters used for AAT are: batching interval=16000 cycles; ranking interval=350K cycles; batching as well as ranking levels=8. For MAT, we use DDR3 timing parameters from the Micron datasheet [26] and replace three conventional NoC routers in the vicinity of the memory controller with memory-aware routers. These prior techniques have used a trace-based simulator for their evaluation purposes while we use an event-driven

simulator with detailed micro-architecture models. All the simulations in our studies were run for at least 150M instructions. For workloads, we considered 17 diverse applications from the SPLASH-2 [28] and SPEC2K [47] benchmark suites. We first profiled all of the applications and separated them into two categories: compute-intensive and memory intensive. Benchmarks with average L1MPKI values less than *equake* are classified as compute-intensive while the remaining benchmarks are classified as memory-intensive. Figure 6.6 shows the L1MPKI and memory level parallelism (MLP) exhibited by the benchmarks. We then used different combinations of these benchmarks to create several multi-programmed workloads. Table 6.2 shows the benchmark combinations we created for our experimental studies. Workloads 1 and 4 are formed by mixing compute-intensive and memory-intensive benchmarks; whereas workloads 2, 3 and 5 represent homogeneous workloads with either all-compute-intensive or all-memory-intensive benchmarks co-running on the system. The number next to a benchmark refers to the degree of parallelization (i.e. the number of cores it runs on).

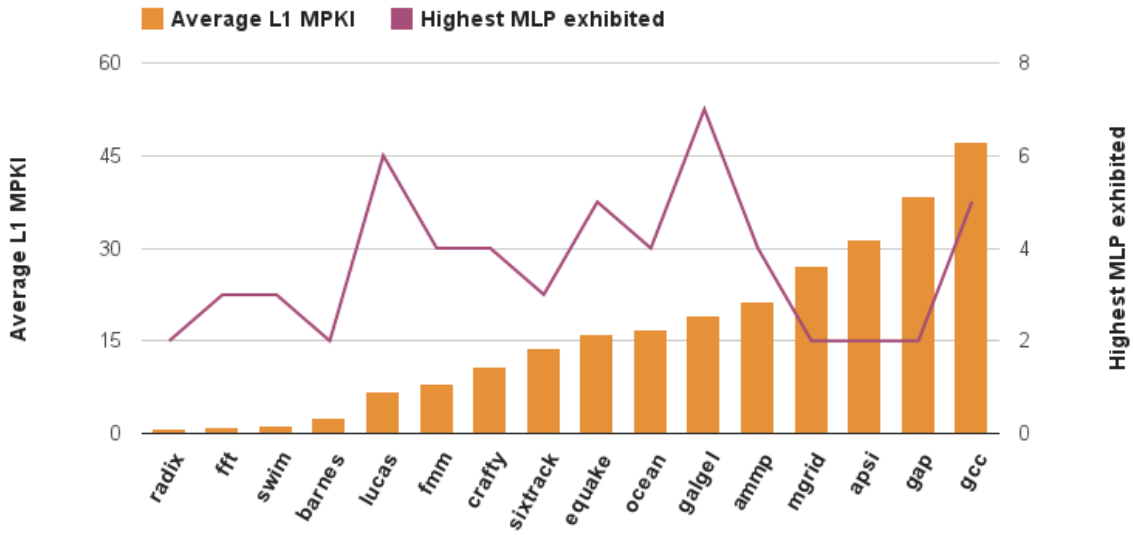


Figure 6.6: L1MPKI and MLP results of application profiling for benchmarks from the SPLASH-2 and SPEC2K suites

Table 6.2: Workloads for execution on baseline CMP

Workload 1	ocean(4), gcc(4), apsi(6), swim(8), applu(5)
Workload 2	lucas(9),barnes(9), radix(9)
Workload 3	gcc(5), ammp(9), galgel(7), gap(6)
Workload 4	crafty(5), gap(7), fft(6), gcc(5), barnes(4)
Workload 5	equake(9), crafty(9), applu(9)

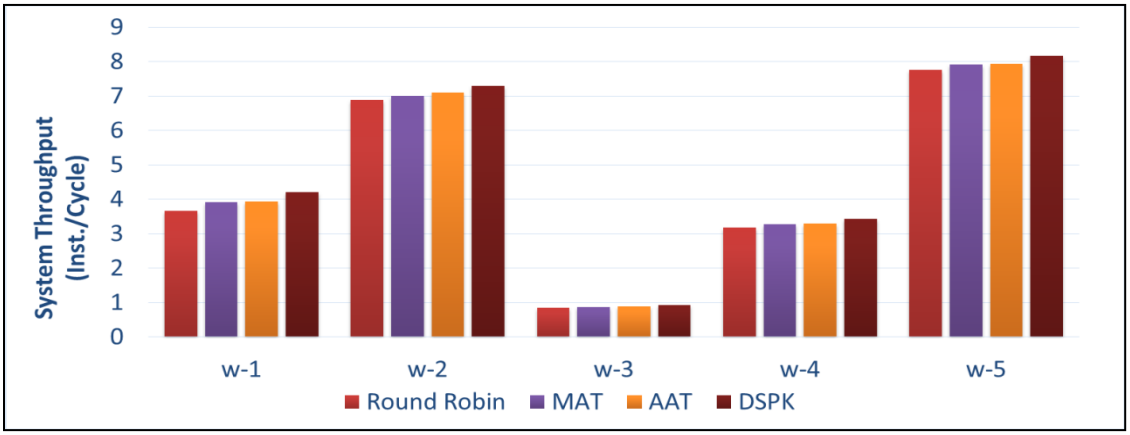
6.4 Experimental Results

In this section, we provide results for our experimental evaluations for the baseline Round Robin, Application-Aware technique (AAT) [16] and Memory-Aware technique (MAT) [13] compared to our proposed technique (DSPK). In Section 6.4.1 we present comparison results for our baseline CMP. In Section 6.4.2, we evaluate the impact of memory speed scaling on the effectiveness of scheduling techniques. In Section 6.4.3, we explore the scalability of our technique for different network sizes. In Section 6.4.4, we compare the energy consumption for the different scheduling techniques. Finally, in Section 6.4.5, we evaluate the area overhead of

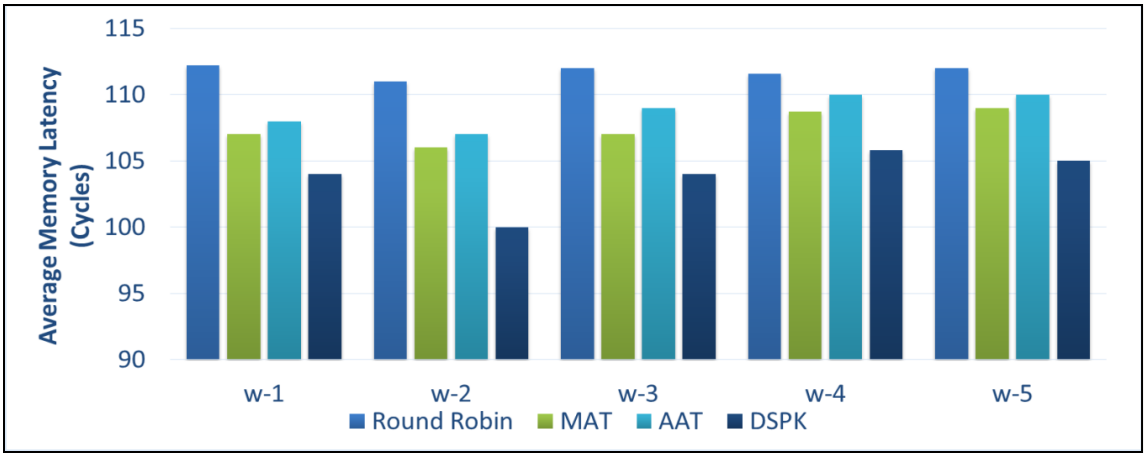
the NoC routers of the techniques considered.

6.4.1 Multi-programmed Workload Evaluation on Baseline CMP

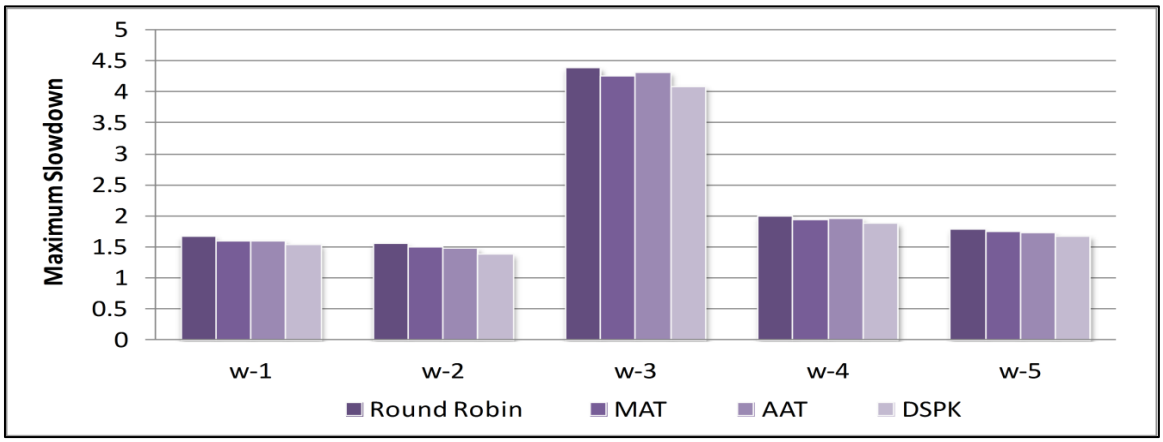
With the growing heterogeneity of applications co-running on modern multi-core systems, it is important that the packet scheduling techniques perform well for various workload combinations. Figure 6.7 shows the performance comparison and presents results for system throughput and average memory latency for the various scheduling techniques. It can be seen that our proposed DSPK framework outperforms the previously proposed scheduling techniques from [13] and [16] and the baseline Round Robin scheme for all workload combinations.



(a)



(b)



(c)

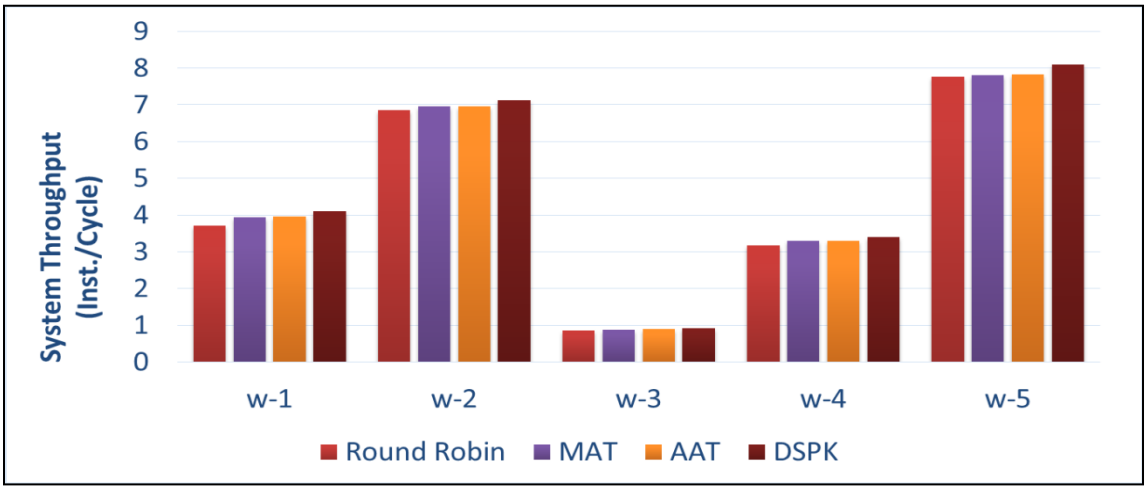
Figure 6.7: Performance evaluation (a) system throughput, (b) average memory latency, and (c) maximum slowdown with DDR3-1333 for workloads from Table 6.2.

This improvement is due to the comprehensive nature of DSPK in scheduling both network packets and memory packets efficiently by considering application-specific and memory-specific characteristics of the system, unlike any of the other techniques. DSPK improves system throughput on an average by 14.4%/7.4%/6.8% and average memory latency by 11%//6%/7% over the baseline Round Robin, MAT and AAT techniques, respectively. To evaluate our fairness technique, we compare the maximum slowdown experienced by the applications in a multi-

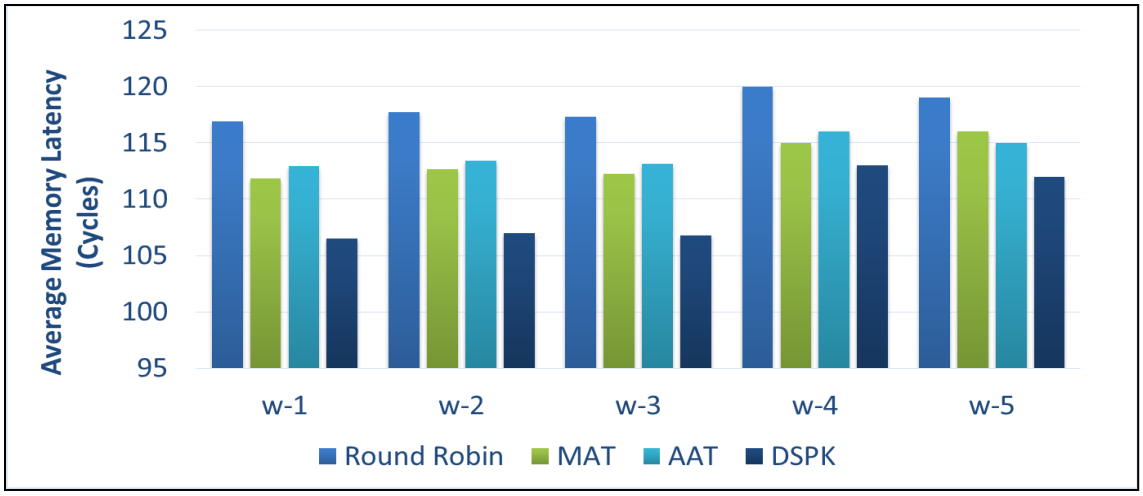
programmed environment. We found that our fairness technique improves the system fairness by 11.5%/7.1%/6.4% over Round Robin, MAT and AAT respectively.

6.4.2 Memory Speed Scaling Evaluation

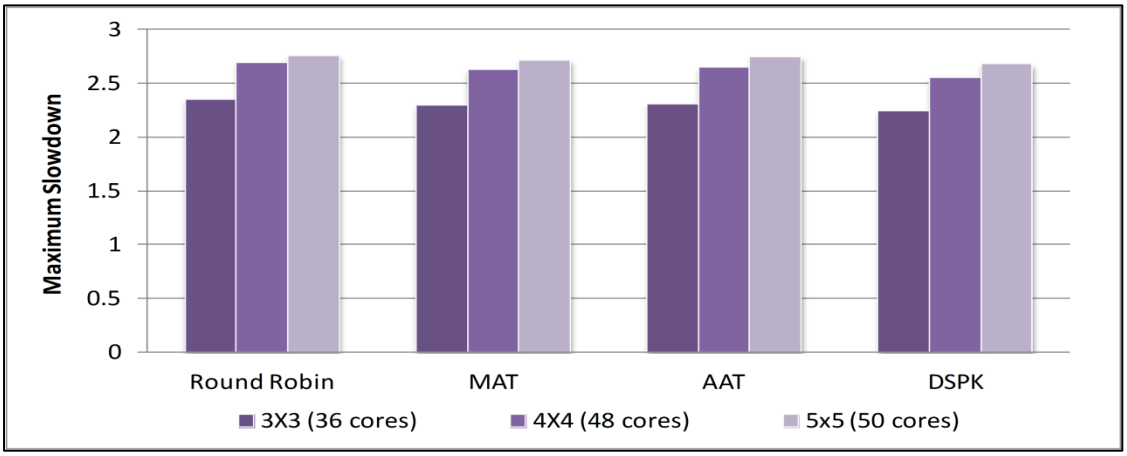
Today's multi-core systems employ high-speed memories to help overcome the "memory wall". However, as memory speed (clock frequency) increases, the number of cycles taken for any memory operation also rises. Any memory-aware scheduling technique should comply with the characteristics and constraints of the memory architecture being utilized in the system. To check for applicability and suitability of our scheduling technique for higher speed memories, we compared its performance to that of other scheduling techniques for the baseline CMP system running with a faster memory model DDR3-1600 [26], instead of the baseline DDR3-1333 model.



(a)



(b)



(c)

Figure 6.8: Memory speed scaling evaluation (a) system throughput, (b) average memory latency, and (c) maximum slowdown for DDR3-1600 with baseline CMP system

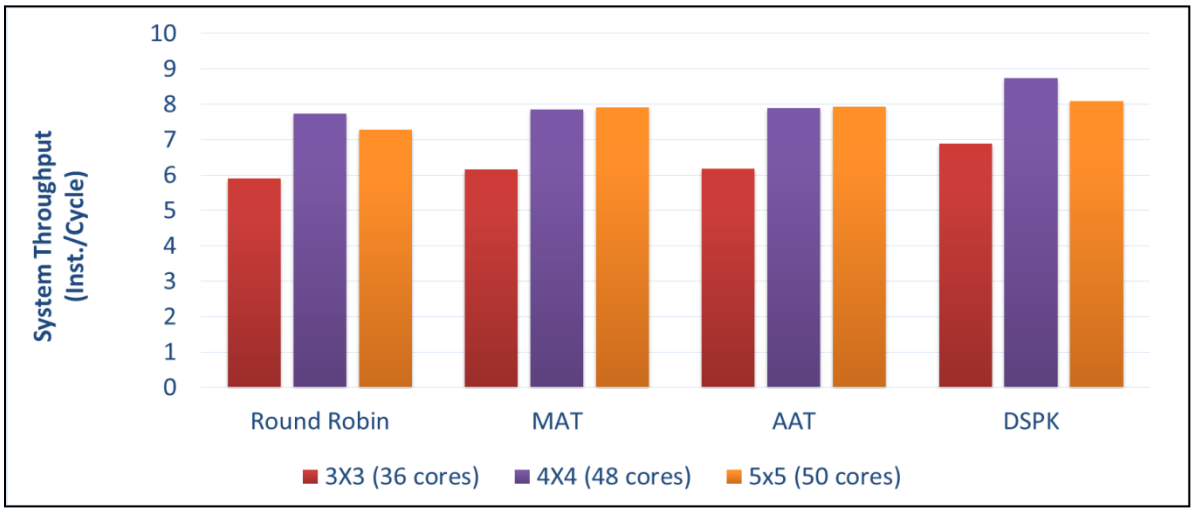
Figure 6.8 shows the results of this comparison study. We observed that the performance of MAT takes a dip with the higher speed memory, because its memory-aware scheduling algorithm does not consider the four activate window constraint (discussed in Chapter 2) while aiming to maximize bank level parallelism. Packets using the MAT technique face frequent head of the queue stalls at the memory controller, thereby increasing its memory latency and lowering its system throughput. AAT is agnostic to memory and hence it achieves the some performance

improvement on the basis of application-aware scheduling. However, DSPK outperforms all the techniques as it is prepared for these issues and handles memory packets and network packets very efficiently ensuring fairness in the system. DSPK achieves an average improvement of 10.3%/4.6%/3.6% for overall system throughput and 10%/5.3%/6% for average memory latency and 11%/6.7%/7.2% for fairness over the baseline Round Robin, MAT and AAT techniques, thereby proving the superior memory speed scalability of our technique.

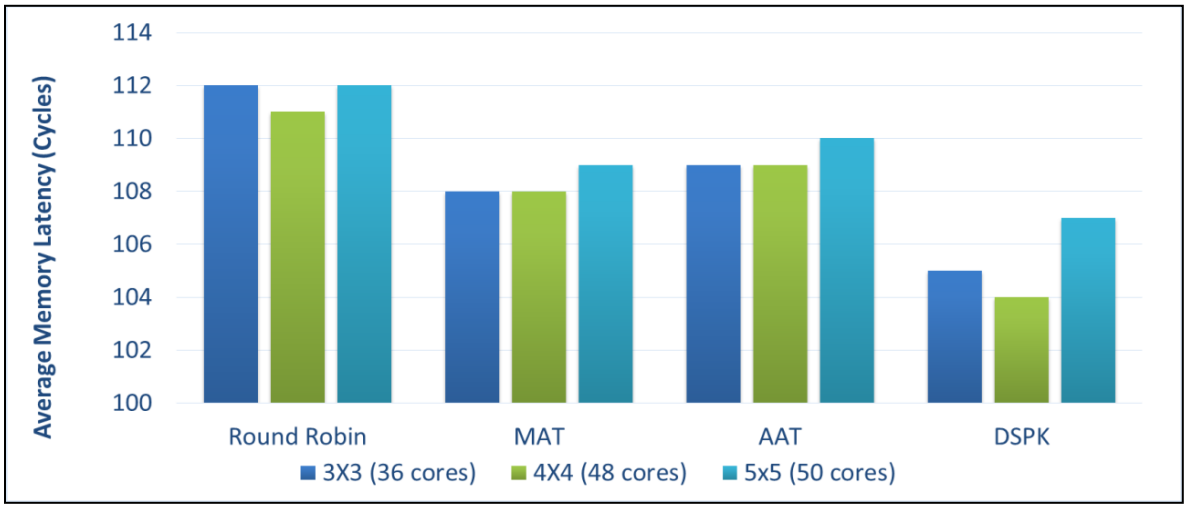
6.4.3 Network Size Scalability Evaluation

Next, we were interested in observing how our technique scales with increasing system complexity and for larger network sizes. Therefore, we studied three different CMP platform complexities with varying network sizes: a 3×3 concentrated NoC with concentration degree of 4 (36 cores), a 4×4 concentrated NoC with concentration degree of three (48 cores) and a 5×5 concentrated NoC with concentration degree of two (50 cores). Figure 6.9 shows the results for system throughput and memory latency averaged over all workloads from Table 6.2 adapted for the three different platforms considered. We observed that our proposed DSPK framework works even better when congestion in the network is higher, as it creates many more opportunities to classify packets and schedule them efficiently. We also noted that AAT ranks applications according to their relative L1MPKI requiring co-ordination among all the nodes in the system to enable a global batching; hence for workloads running on larger network sizes, the technique has a higher overhead making it less effective. MAT focuses only on memory packets with the help of its three memory-aware routers while ignoring network packets, which results in lower performance for larger network sizes with more network packet dominated communication flows. DSPK has an edge over all of these techniques owing to its holistic nature, providing an

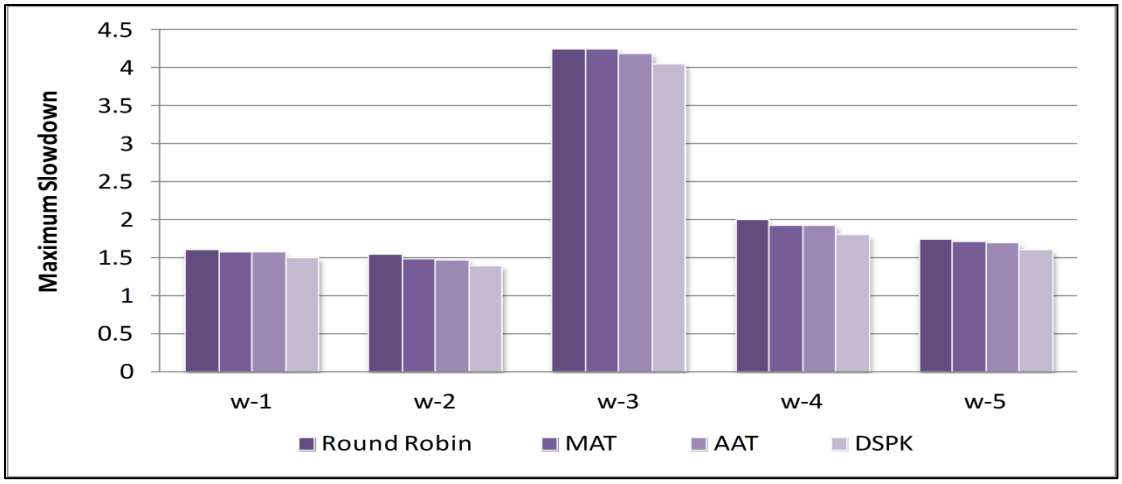
average improvement of 16.7%/11.8%/11.4% for overall system throughput, 6.7%/3.8%/4.8% for average memory latency and 5.5%/2.7%/3.5% for network slowdown over the baseline Round Robin, MAT and AAT techniques for the three CMP platforms with DDR3-1333 memory.



(a)



(b)



(c)

Figure 6.9: Network scalability evaluation (a) system throughput, (b) average memory latency, and (c) maximum slowdown for 3×3 (36 core), 4×4 (48 core) and 5×5 (50 core) CMPs

6.4.4 Energy Consumption Evaluation

We were also interested in evaluating the energy consumption of DSPK and comparing it with other previously proposed scheduling techniques. We calculated the power for out-of-order cores, on-chip network and memory (caches and DDR3 DRAM) using McPAT [48], Orion 2.0 [29] and CACTI 4.0 [49]. We carried out our experiments on the baseline 3×3 CMP platform with 27 cores, for three different workload combinations consisting of compute-intensive workloads (w-2), memory-intensive workloads (w-3) and a mix of these two types of workloads (w-1). Figure 6.10 presents the energy consumption results from this evaluation study. We observed that the baseline Round Robin scheduling based configuration consumes the highest energy due to its application and memory-oblivious nature, which increases execution time considerably (even though its power overhead is lower than the other techniques due to the simplicity of its implementation). MAT improves upon the baseline Round Robin technique as it schedules packets in a memory-operation friendly manner, thereby lowering memory latency and

saving memory energy. AAT is unaware of the memory but does application-aware scheduling efficiently for w-1 and w-2 workloads where network packets are in the majority, while for w-3 its energy consumption increases as the memory packets in the network increase and AAT is unable to handle them efficiently. DSPK, on the other hand, has lower energy consumption than all of these techniques as it is able to handle all kinds of workloads efficiently. We observe an average improvement of 10%/5%/4.7% for overall energy consumption with DSPK over the baseline Round Robin, MAT and AAT techniques.

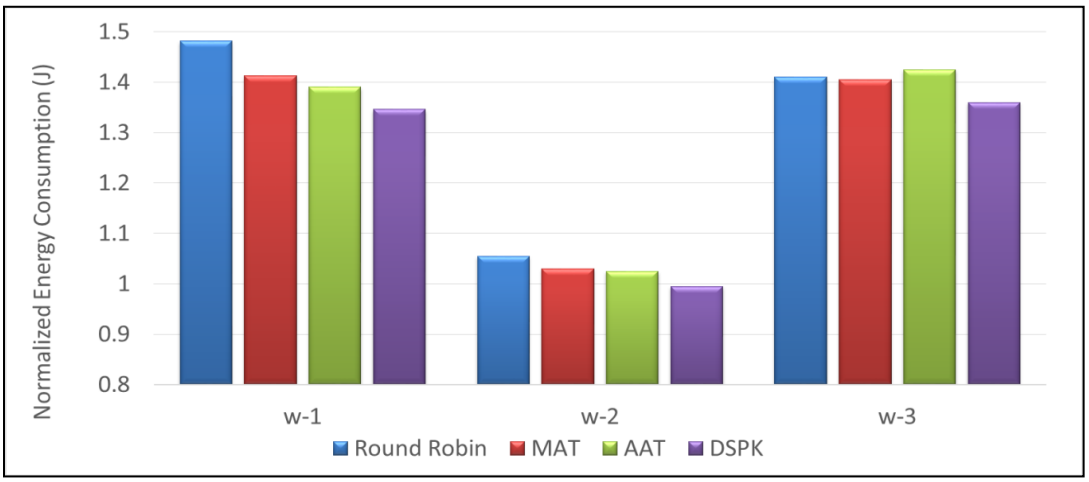
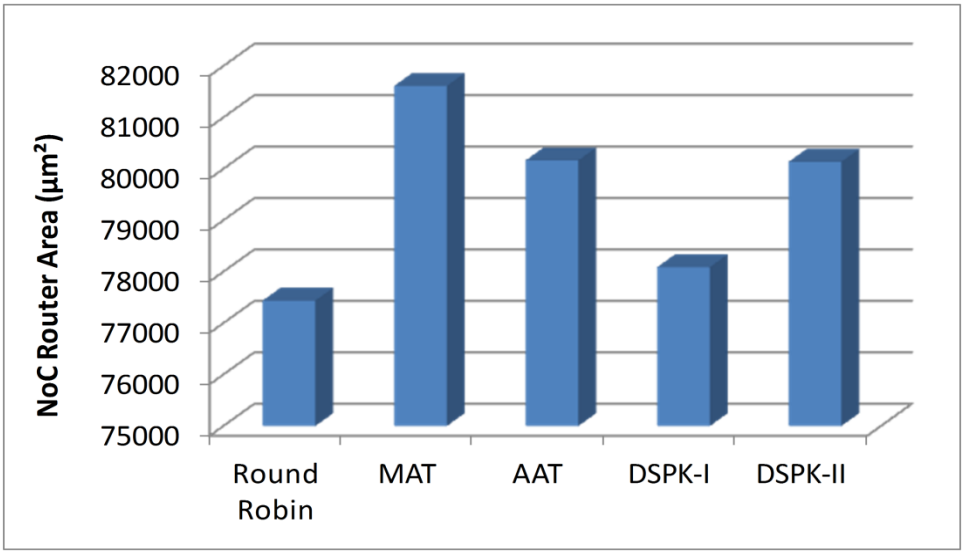


Figure 6.10: Normalized energy consumption across mixed (w-1), compute-intensive (w-2) and memory-intensive (w-3) workloads.

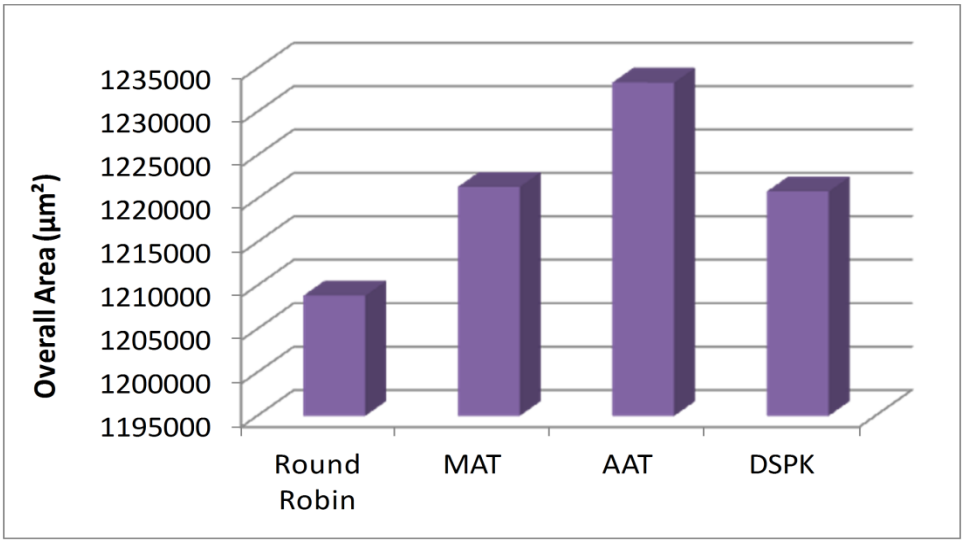
6.4.5 Area Overhead Estimation

Lastly, we evaluated the area overhead for our technique in comparison with others techniques. Figure 6.11 (a) shows the area of the intelligent routers in DSPK and other scheduling techniques for the 32nm CMOS technology node. The Round Robin scheme is the simplest and therefore not surprisingly has the lowest area overhead. MAT keeps memory-specific information at each router and stores the complete addresses of winners from previous arbitrations which increases its overhead. AAT stores the L1 MPKI of each core and has the

circuitry to compute ranks of all the cores, which significantly increases its area overhead. DSPK uses small counters locally and also operates on a smaller header flit compared to AAT (as it does not carry a larger batch id and only needs 2-bit rank data).



(a)



(b)

Figure 6.11: Area estimation for scheduling techniques: (a) NoC router area, and (b) overall NoC fabric area.

Therefore, its overhead is lower than the AAT technique. We also calculated the total area of the NoC fabric with each of the scheduling techniques and found that DSPK adds only 9% to the overall NoC area over the baseline Round Robin scheme and 5.6% more area over the MAT scheme. When compared to the AAT scheme, DSPK has 4.8% lower area overhead. Thus, our DSPK framework improves system throughput and reduces memory latency and energy consumption at the cost of a slight increase in the area.

Our proposed scheduling framework (DSPK) was shown to achieve up to 16.7%, 11% and 10% improvements in system throughput, average memory latency and energy consumption, respectively, as compared to the other previously proposed scheduling techniques. Given its scalability and superior performance across various workload types, we believe that our approach is an attractive option for future multi-core NoC-based systems executing multiple and diverse applications.

Chapter 7

Conclusion and Future Work

This chapter highlights the key contributions of the thesis, summarizes how our proposed prioritization framework addresses the previously discussed challenges and can be effective in achieving maximum performance benefits, and finally describes scope for future extensions to our work.

7.1 Summary

As the number of processing elements on a die are rapidly growing, interconnects are becoming increasingly crucial for overall system performance along with the memory subsystem. Interconnects play a significant role in determining maximum achievable performance as inter-processor and processor-memory communication is dependent on it. The main contribution of this thesis is the design of novel heterogeneous prioritization techniques for network and memory packets in NoCs. Chapter 1 introduced this thesis and described in detail our motivation to pursue this area of research. Chapter 2 provided the essential background relevant to this research area including NoC basics, multi-programmed CMPs, SDRAM functionality, etc. In Chapter 3, the thesis optimization goals and parameters were specified. A representative subset of relevant research in this area was presented in Chapter 4 as related work, along with a discussion of some of the existing issues and challenges in the area. To address these issues and achieve performance goals, we proposed a holistic heterogeneous prioritization framework. Within the umbrella of this heterogeneous prioritization framework, an application-aware prioritization technique, application classification metric, two memory-aware

prioritization techniques and an anti-starvation technique were proposed and presented in Chapters 5 and 6. In these chapters, the experimental results and analyses were also presented.

7.2 Conclusion

In conclusion, our proposed heterogeneous prioritization framework has addressed the packet scheduling challenge faced by state-of the-art NoC-based CMPs. We discussed the shortcomings with some of the best performing recently proposed works on packet scheduling. To overcome these shortcomings, we proposed a holistic prioritization framework to optimize the end-to-end latency of packets in CMPs with multiple co-running applications. Due to the multi-level nature of our prioritization approach, it can intelligently identify the specific network- or memory-characteristics of a packet and give it an appropriate priority in arbitration decisions. Two different packet prioritization approaches were proposed and applied to NoC routers, depending on whether the routers are nearer or farther away from the off-chip memory subsystem. A new ranking scheme for classifying an application's criticality was also proposed. We also proposed a novel anti-starvation mechanism for establishing fairness in multi-programmed workload based systems.

Our experiments were performed using a full-system, cycle-accurate event-driven simulator that validated our motivation and intuition. We evaluated our proposed framework for system throughput, average memory latency, and energy consumption compared to other previously proposed scheduling techniques. In a nutshell, the best improvements achieved by our proposed techniques over the baseline Round Robin prioritization mechanism and prior works are summarized in table 7.1.

Table 7.1: Key improvements of proposed framework

Prioritization Techniques	System Throughput	Memory Latency	Energy Consumption
Baseline Round Robin	16.7%	11%	10%
State-of-the-art techniques	11.8%	7%	5%

Given its scalability and superior performance across various workload types, we believe that our proposed prioritization framework is an attractive option for future multi-core NoC-based systems executing multiple and diverse applications.

7.3 Future Work

As discussed in Chapter 4, a significant amount of research is being done for NoC-based multi-core systems. In Chapters 5 and 6, a holistic solution was proposed to resolve the performance challenges related to on-chip and off-chip memory communication as well as fairness issues faced by modern multi-core systems executing parallel applications. However, this work can certainly be extended to improve achievable system performance. Some of the possible research directions to extend this work are presented below-

- **Finer-grain application classification strategy-** Our application-aware strategy uses a packet-criticality and destination-based approach to prioritize applications. We use four ranking levels to classify applications. An alternate method for application classification can be devised. Having more number of ranking levels can ensure finer classification among application packets and enable higher scalability. However, it comes with an additional hardware cost and area overhead.

- **Adaptability with emerging memory technologies-** Two memory-aware strategies to be implemented in NoC routers are proposed in this thesis. We have considered the DRAM memory models- DDR2-667, DDR3-1333 and DDR3-1600. Emerging alternate memory models like Spin-Torque Transfer RAM (STT-RAM) and Phase Change RAM (PCRAM) can be implemented to see if the proposed techniques are adaptable and effective for memory hierarchies with these new memories.
- **Power-aware optimizations-** We evaluated energy consumption for some recently proposed techniques and our proposed framework. We achieved reduction in energy consumption due to reduced system latency. It would be interesting to devise some power-aware optimizations for NoC routers in conjunction with the proposed framework for further benefits in power and energy.
- **Eliminating inter-application interference-** There are several research papers that focus on eliminating or mitigating inter-application interference [19][34]. Since we make minimal hardware alterations, our proposed work cannot eliminate inter-application interference although it handles it by prioritizing one application over another. With customization of hardware, inter-application interference could be prevented to improve system performance more aggressively at the cost of more complex hardware and higher power dissipation.
- **Combination with application mapping techniques-** An application mapping technique can be integrated with the proposed framework to obtain higher gains in performance. With intelligent mapping of the applications to the cores, the proposed dual stage NoC routers will be more productive in terms of handling memory and on-chip network traffic.

This area of NoC and memory optimizations has a large scope for future research and can enable huge performance gains with appropriate system design tradeoffs. The above mentioned directions are not exhaustive by any means and represent some of the multiple ways in which future research can alleviate bottlenecks in the NoC and memory subsystems for multi-core chip platforms.

References

[1] Y. Kim, M. Papamichael, O. Mutlu and M. Harchol-Balter, “Thread Cluster Memory Scheduling: Exploiting differences in Memory Access Behavior.”, Proc. MICRO-43, 2010.

[2] Y. Jin, R. Wang, W. Choi and T.Pinkston, “Thread Criticality Support in On-Chip Networks”, Proc. NoCArc 2010.

[3] S. Pasricha and N. Dutt, “On-Chip Communication Architectures”, Morgan Kauffman, Apr 2008.

[4] A. Glew, “MLP Yes! ILP No! Memory Level Parallelism, or, Why I No Longer Worry About IPC”, Proc. ASPLOS WACI Session, 1998.

[5] N. Dutt, “Memory-aware NoC Exploration and Design”, Proc. DATE 2008, pp. 1128-1129.

[6] S. Rixner, W. Dally, U. Kapasi, P. Mattson and J. Owens, “Memory Access Scheduling,” Proc. ISCA 2000.

[7] O. Mutlu and T. Moscibroda, “Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors”, Proc. MICRO-40, 2007.

[8] Y. Kim, D. Han, O. Mutlu and M. Harchol-Balter, “ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers”, Proc. HPCA-16, 2010.

[9] R. Ausavarungnirun, K. Chang, L. Subramanian, G.Loh and O. Mutlu, “Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems”, Proc. ISCA 2012.

[10] O. Mutlu and T. Moscibroda, “Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems”, Proc. ISCA-35, 2008.

[11] Z. Fang, X. Sun, Y. Chen and S. Byna, “Core-Aware Memory Access Scheduling Schemes”, Proc. IEEE International Parallel and Distributed Processing Symposium, (IPDPS) 2009, pp. 1-12.

[12] D. Kim, S. Yoo and S. Lee, “A Network Congestion-Aware Memory Controller”, Proc. NOCS 2010.

[13] W. Jang and D. Pan, “An SDRAM-Aware Router for Networks-on-Chip”, Proc. DAC 2009, pp. 800-805.

[14] S. Phadke and S. Narayanasamy, “MLP-aware Heterogeneous Memory System”, Proc. DATE 2011, pp. 1-6.

[15] W. Jang and D. Pan, “Application-Aware NoC Design for Efficient SDRAM Access”, Proc. DAC 2010, pp. 453-456.

[16] R. Das, O. Mutlu, T. Moscibroda and C. Das, “Application-Aware Prioritization Mechanisms for On-Chip Networks”, Proc. MICRO-42, 2009, pp. 280-291.

[17] R. Das, , O. Mutlu, T. Moscibroda and C. Das, “Aérgia: Exploiting Packet Latency Slack in On-Chip Networks”, Proc. ISCA 2010.

[18] A. Sharifi, E. Kultursay, M. Kandemir and C. Das, “Addressing End-to-End Memory Access Latency in NoC-Based Multicores”, Proc. MICRO 2012, pp. 294-304.

[19] A. K. Mishra, O. Mutlu and C. Das, “A Heterogeneous Multiple Network-On-Chip Design: An Application-Aware Approach”, Proc. DAC 2013.

[20] D. Kroft, “Lock-up Free Instruction Fetch/pre-fetch Cache Organization”, Proc. ISCA 1981, pp. 81-87.

[21] N. Jerger and L. Peh, “On-Chip Networks”, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers 2009.

[22] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill and D. Wood, “The Gem5 Simulator”, ACM SIGARCH Computer Architecture News, vol. 39, Issue 2, May 2011, pp. 1-7.

[23] N. Agarwal, , T. Krishna, L. Peh and N. Jha, “GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator”, ISPASS 2009.

[24] “Micron datasheet DDR2-667”, http://download.micron.com/pdf/datasheets/modules/ddr2/HTF16C128_256x64H.pdf

[25] S. Eyerman and L. Eeckhout, “System-Level Performance Metrics for Multiprogram Workloads”, IEEE Micro, May-June 2008, pp. 42-53.

[26] “Micron datasheet DDR3-1333”, http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf

[27] “Intel’s SCC”, http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html

[28] S.C. Woo, S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations”, Proc. ISCAS 1995.

[29] A. B. Kahng, B. Li, L. Peh, and K. Samadi, “ORION 2.0: A Power-Area Simulator for Interconnection Networks”, Trans. Very Large Scale Integration (VLSI) Systems, 20(1), Jan. 2012.

[30] W. J. Dally and B. Towles, “Route Packets, Not Wires: On-Chip Interconnection Networks”, Proc. DAC, Jun. 2001, pp. 684-689.

[31] L. Benini and G. De Micheli, “Networks on Chips: A New SoC Paradigm”, Computer, vol. 35, no. 1, Jan. 2002, pp. 70-78.

[32] W. J. Dally and B. Towles, “Principles and Practices of Interconnection Networks”, Morgan Kaufmann, 2003.

[33] “AMD Opteron 6200 series processors”, http://www.amd.com/US/PRODUCTS/SERVER/PROCESSORS/6000-SERIES-PLATFORM/6200/Pages/6200-series-processors.aspx

[34] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir and T. Moscibroda, “Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning”, Proc. MICRO-44, 2011, pp. 374-385.

[35] “Main Memory lecture by Prof. S. Pasricha, Colorado State University”, http://www.engr.colostate.edu/~sudeep/teaching/%20ppt_554/lecture05-dram.pdf

[36] J. Balfour and W. J. Dally, “Design Tradeoffs for Tiled CMP On-Chip Networks”, Proc. ICS 2006, pp. 187-198.

[37] S. Sair and M. Chamey, "Memory Behavior of the SPEC2000 Benchmark Suite", IBM Thomas J. Waston Research Center Technical Report RC-21852, October 2000.

[38] V. Cuppu, B. Jacob, B. Davis and T. Mudge, “A Performance Comparison of Contemporary DRAM Architectures”, Proc. ISCA 1999.

[39] B.Grot, J. Hestness, S. Keckler and O. Mutlu, “Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees”, Proc. ISCA 2011.

[40] A.Mishra, N. Vijaykrishnan and C. Das, "A Case for Heterogeneous on-Chip Interconnects for CMPs", Proc. ISCA 2011, pp. 389-400.

[41] E. Ebrahimi, C. Lee, O. Mutlu, Y. Patt, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems", Proc. ASPLOS 2010.

[42] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. Lee, J. Joao, O. Mutlu and Y. Patt, "Parallel Application Memory Scheduling", Proc. MICRO-44, 2011, pp. 362-373.

[43] "Tile64 processor", http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf

[44] "Interconnection Networks lecture by Prof. S. Pasricha, Colorado State University", http://www.engr.colostate.edu/~sudeep/teaching/lectures_452.htm

[45] "Intel's Teraflop", http://static.trustedreviews.com/94/836ed2/0e6e/3477-80core.jpg

[46] "Memory wall", http://www.engr.colostate.edu/~sudeep/teaching/ppt_554/lecture03-review2.pdf

[47] **"**SPEC2K benchmark suite**",** http://www.spec.org/cpu2000/

[48] S. Li, J. Ahn, J. Brockman and N. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures", Proc. MICRO-42 2009.

[49] "CACTI 4.0", http://www.hpl.hp.com/research/cacti/

Appendix

Disclaimer: The information presented in Appendices I and II is taken from a few sources along with my own experiences. The main motive is to put useful information together and make it easy for anyone who intends to use the GEM5 platform for research pertaining to NoC-based systems. Please note that GEM5 is a vast full system simulator, so it is not possible to include every detail in the appendices. Please refer to the sources cited for more information.

Appendix A

GEM5 Usage Guide

GEM5 is an event-driven full system simulator which consists of state-of-the-art processors, private and shared caches, on-chip interconnect and detailed memory subsystem. This appendix intends to provides basic pointers for setting up GEM5.

1. Download GEM5

GEM5 is an open source simulation software. It can be downloaded freely on a linux-based machine from here. It is advisable to download using mercurial so that it can be updated with newer versions easily. Before compiling Gem5, these softwares must be installed on your system. Use SWIG 1.3.34 and gcc 4.5 versions in case there are any issues compiling it.

GEM5 overview: In order to start simulating with GEM5, let us have a look at the top Level GEM5 structure. Figure A.1 shows high level GEM5 structure. It has two modules for modeling system architecture- Simple and Ruby. Simple model is basic high level modeling whereas Ruby model is more detailed and state-of-the-art model for CMPs. Both Simple and Ruby models have network and memory models. Ruby model comprises of Garnet network and ruby memory model.

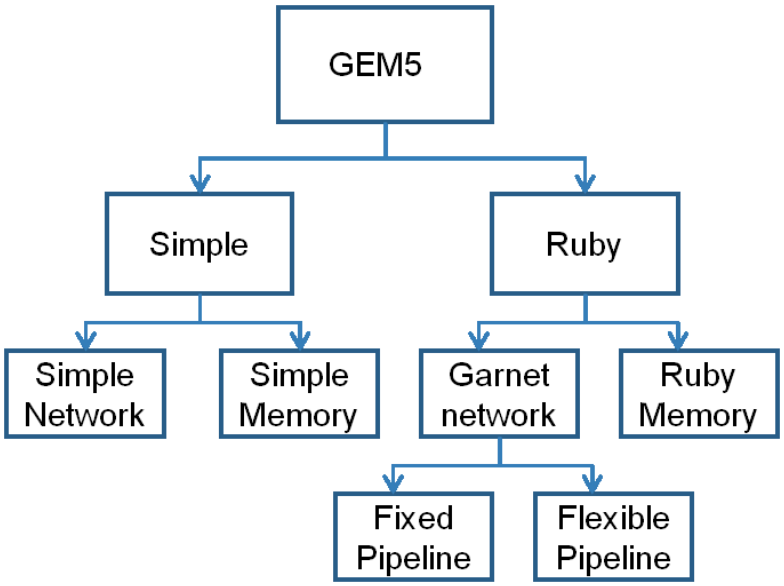


Figure A.1: Top level GEM5 overview.

There are two main modes in which Gem5 can operate:

- **FS (Full System) mode:** As the name suggests, this mode operates as a full system in the form of a virtual machine. However, this mode is very slow for running applications and can be avoided unless research is related to kernel operations and ISA optimizations. More details about running this mode can be found here.
- **SE (System Emulation) mode:** This is a simpler and faster way of simulations in Gem5. Most of the times it is sufficient to use SE mode as we deal with application level performance. As the name suggests, this mode emulates the full system by emulating system calls by kernel instead of actually booting the kernel.

We will study mainly SE mode with Ruby model for implementing contemporary techniques.

2. Compiling GEM5

Once the local GEM5 repository is created and required softwares are installed on the system, GEM5 is ready to be compiled.

Command for Compiling Gem5:

```
%scons build/<arch>/gem5.<binary> -j4
```

Replace <arch> by any architecture (of your choice) as mentioned below. For binaries, replace it preferably by gem5.opt (for faster compiling) or gem5.debug (slower compiling but has debug information).

Architecture capabilities: GEM5 provides platform for following mainly used processor architectures:

- **ALPHA:** This is the most established architecture in GEM5 with least bugs.
- **ARM:** Developed thoroughly using ARM Realview development suite. Useful for embedded mobile computing related studies.
- **X86:** PC-based architecture platform.
- **SPARC:** Based on Sun T1 architecture.

In addition to choosing the processor architecture, it is beneficial to choose the cache coherence protocol for using Ruby model. GEM5 provides the following primary cache coherence protocols:

- **MESI CMP DIRECTORY**: This is a directory-based protocol. In this coherence protocol, each core has a private L1 cache and subsequent caches are shared.
- **MOESI HAMMER**: This is a snooping protocol. This protocol has private L1 and L2 caches.

Depending upon one's requirement about directory-based or snoop-based protocols or private L2 or shared L2 caches, one can choose the appropriate cache coherence protocol. It should be specified at the time of compiling GEM5. A typical command line to use can be-

```
%scons build/ALPHA_MESI_CMP_directory/gem5.opt \
PROTOCOL=MESI_CMP_directory \ RUBY=TRUE
```

3. Running Simulations on GEM5

Once GEM5 is compiled successfully, the binary gem5.opt/gem5.debug generated can be used to run simulations.

3.1 Simulation commands

You can try your first simulation by this simple command line.

```
%./build/ALPHA/gem5.opt configs/example/se.py  \
-c tests/test-progs/hello/bin/alpha/linux/hello
```

In addition to the above command, there are several system specific options that can be specified via command line to create a user defined system. These options can be viewed using:

```
%./build/ALPHA/gem5.opt -h
```

```
%./build/ALPHA/gem5.opt configs/examples/se.py -h
```

3.2 Running Benchmarks on GEM5

GEM5 supports multi-core computing and can be used to execute standard benchmarks from benchmark suite like SPLASH2, PARSEC, SPEC benchmarks, etc. Pre-cross-compiled Splash2 binaries for ALPHA platform can be downloaded from here.

Parsec and original Splash2 benchmarks can be downloaded from here and can be cross compiled as per desired platform. The SPEC benchmarks require license and needs to be purchased. GEM5 provides utility to run SPEC benchmarks. These documents provides some useful information about specifying command line options for SPEC benchmarks- SPECINT and SPECFP.

GEM5 uses script to run desired benchmarks as per user defined system configurations. Examples of such scripts are in gem5/configs/example directory.

Running GEM5 with multiple workloads:

Following is an example of command line specified for running multi-programmed workloads. An example code of script for using below mentioned command line is provided in Appendix B.

```
%./build/ALPHA_MESI_CMP_directory/gem5.opt
configs/example/se_multiprogram.py -n 8 --cpu-type=detailed
--clock=1GHz --topology=MeshDirCorners --caches
--cacheline_size=128 --l2cache --num-l2caches=64
--l1d_size=16kB --l1i_size=16kB --l2_size=128kB
--l1d_assoc=2 --l1i_assoc=2 --l2_assoc=2 --num-dirs=4
--mesh-rows=2 --ruby --garnet-network=fixed
--benchmark="ammp;apsi;radix;gap;crafty;barnes;ocean;gcc"
-I 400000000
```

The results of simulations are output to the folder gem5/m5out. Files generated for results are-

- **config.ini** – This file provides information about connection tree of the system i.e. the way in which one module in the system is connected to another and specifications of each module e.g. size, version, capacity, etc.
- **stats.txt** – This file provides detailed information about each individual processor executing workload e.g. total simulation time, IPC of application running on any core, etc.
- **ruby.stats** – This file is generated only when '--ruby' option is specified in the command line while running GEM5. It provides statistics for memory and network

usage e.g. number of L1 misses, number of main memory requests, network utilization, virtual channel load, etc.

Appendix B

GEM5 Source Code Modification Guide

Being a full system simulator, GEM5 has a vast code base consisting of models for CPU, ISA, kernel, interconnect, cache, memory and other microarchitecture. In order to use GEM5 for research, one needs to understand the existing codebase and modify it. The official website fof GEM5 provides general information about the simulator but not really code specific. This appendix aims provides pointers to the code structure of GEM5 to facilitate modifying the code. Hopefully, this will be helpful to understand the simulator model flow and ease the pain of hacking GEM5 code ☺.

GEM5 has two major code components:

1. Configuration code and
2. Source code

Section 1 provides information regarding configuration code and Section 2 sheds light on GEM5 source code. GEM5 is comprised of C++ code for modeling system level components like cpu, caches, network, memory, etc and bound by a python wrapper to facilitate communication between these modules.

1. GEM5 Configuration code

GEM5 cofiguration code is written in python. This code is located in gem5/configs.

Directory Structure: Figure below shows sub-folders of configs directory containing configuration code.

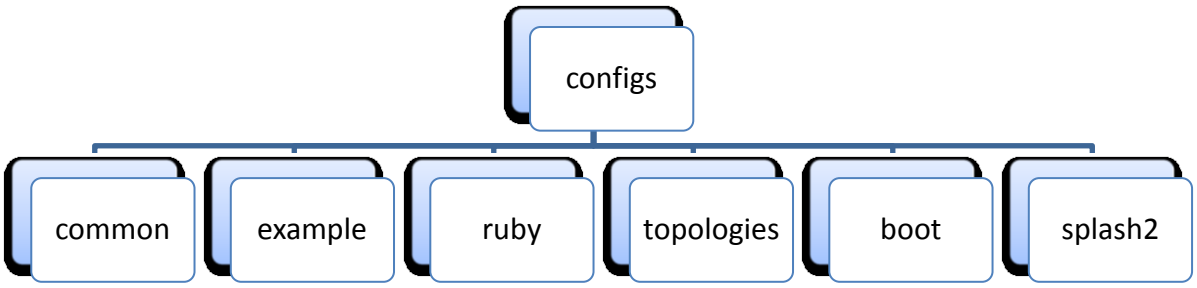
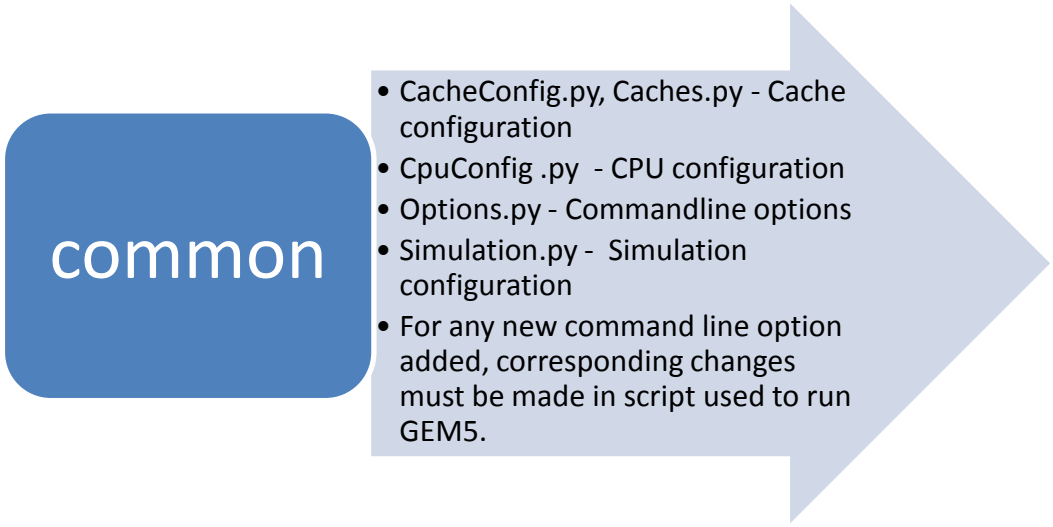
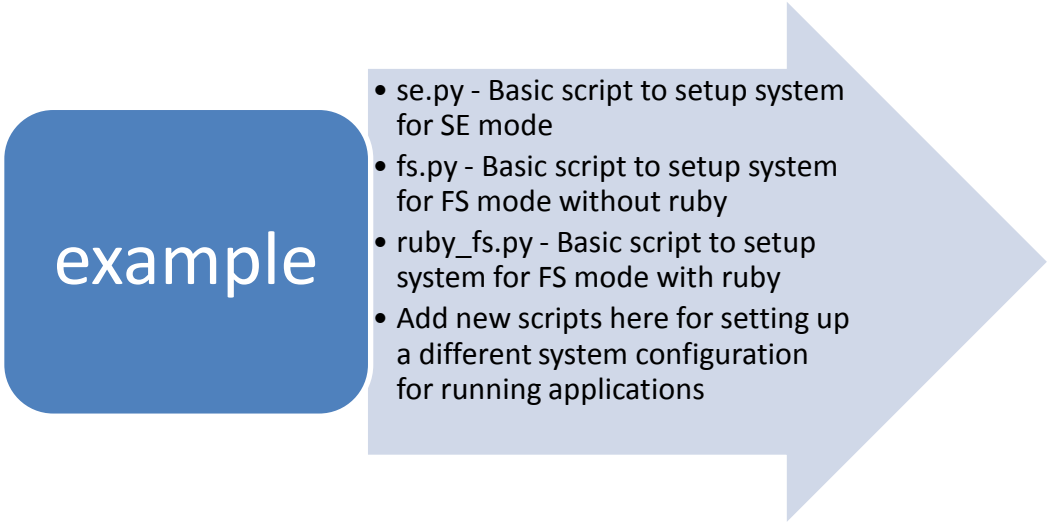
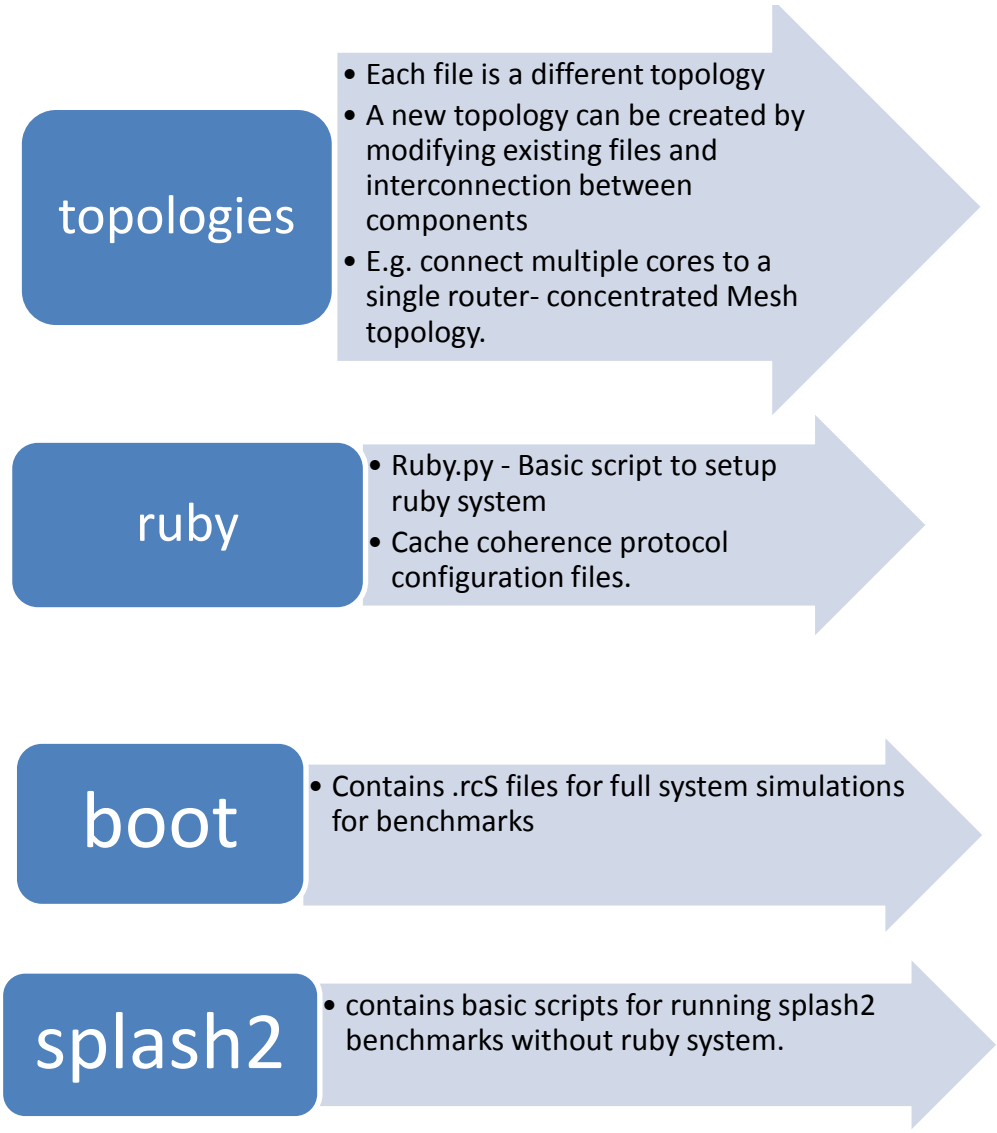


Figure B.1: Configuration directory structure







We provide a sample script to run multi-programmed workloads using Ruby system in SE mode. It includes options for benchmarks from SPLASH2 and SPEC2K benchmark suite. However, this is a sample file and in order to use directly with your GEM5 environment a few changes will be needed in configs/common/Options.py and configs/ruby/Ruby.py.

B.1 se_multiprogram.py

```
# Author: Tejasi Pimpalkhute
#
# Simple script for multi-programmed workloads
# "se_multiprogram.py"

import optparse
import sys

import m5
from m5.defines import buildEnv
from m5.objects import *
from m5.util import addToPath, fatal

addToPath('../common')
addToPath('../ruby')
addToPath('../topologies')

import Options
import Ruby
import Simulation
import CacheConfig
from Caches import *
from cpu2000 import *

def get_processes(options):
    """Interprets provided options and returns a list of processes"""

    multiprocesses = []
    inputs = []
    outputs = []
```

```
errouts = []
pargs = []

workloads = options.cmd.split(';')
if options.input != "":
    inputs = options.input.split(';')
if options.output != "":
    outputs = options.output.split(';')
if options.errout != "":
    errouts = options.errout.split(';')
if options.options != "":
    pargs = options.options.split(';')

idx = 0
for wrkld in workloads:
    process = LiveProcess()
    process.executable = wrkld

    if len(pargs) > idx:
        process.cmd = [wrkld] + pargs[idx].split()
    else:
        process.cmd = [wrkld]

    if len(inputs) > idx:
        process.input = inputs[idx]
    if len(outputs) > idx:
        process.output = outputs[idx]
    if len(errouts) > idx:
        process.errout = errouts[idx]

    multiprocesses.append(process)
    idx += 1
```

```
    if options.smt:
        assert(options.cpu_type == "detailed" or options.cpu_type == "inorder")
        return multiprocesses, idx
    else:
        return multiprocesses, 1


parser = optparse.OptionParser()
Options.addCommonOptions(parser)
Options.addSEOptions(parser)

#Benchmark options
parser.add_option("--rootdir1",
                  help="Root directory of Splash2 Benchmarks",
                  default="splash2/codes")
parser.add_option("--rootdir2",
                  help="Root directory of Spec2K Benchmarks",
                  default="spec2k")
parser.add_option("-b", "--benchmark",
                  help="Benchmark to run")


if '--ruby' in sys.argv:
    Ruby.define_options(parser)

(options, args) = parser.parse_args()

if args:
    print "Error: script doesn't take any positional arguments"
    sys.exit(1)
```

```
# --------------------
# Define Splash2 Benchmarks
# ====================
class cholesky(LiveProcess):
    cwd = options.rootdir1 + '/kernels/cholesky'
    executable = options.rootdir1 + '/kernels/cholesky/CHOLESKY'
    cmd = ['CHOLESKY', #, '-p' +  str(options.num_cpus),
            options.rootdir1 + '/kernels/cholesky/inputs/tk25.O']

class fft(LiveProcess):
    cwd = options.rootdir1 + '/kernels/fft'
    executable = options.rootdir1 + '/kernels/fft/FFT'
    cmd = ['FFT', '-m18'] #, '-p', str(options.num_cpus), '-m18']

class LU_contig(LiveProcess):
    executable = options.rootdir1+ '/kernels/lu/contiguous_blocks/LU'
    cmd = ['LU'] #, '-p', str(options.num_cpus)]
    cwd = options.rootdir1 + '/kernels/lu/contiguous_blocks'

class LU_noncontig(LiveProcess):
    executable = options.rootdir1 + '/kernels/lu/non_contiguous_blocks/LU'
    cmd = ['LU'] #, '-p', str(options.num_cpus)]
    cwd = options.rootdir1 + '/kernels/lu/non_contiguous_blocks'

class Radix(LiveProcess):
    executable = options.rootdir1 + '/kernels/radix/RADIX'
    cmd = ['RADIX', '-n524288'] #, '-p', str(options.num_cpus)]
    cwd = options.rootdir1 + '/kernels/radix'

class barnes(LiveProcess):
    executable = options.rootdir1 + '/apps/barnes/BARNES'
    cmd = ['BARNES']
```

```
    input = options.rootdir1 + '/apps/barnes/input' #+ str(options.num_cpus)
    cwd = options.rootdir1 + '/apps/barnes'

class fmm(LiveProcess):
    executable = options.rootdir1 + '/apps/fmm/FMM'
    cmd = ['FMM']
    #if str(options.num_cpus) == '1':
    input = options.rootdir1 + '/apps/fmm/inputs/input.2048'
    #else:
     #          input    =    options.rootdir1    +    '/apps/fmm/inputs/input.2048.p'    +
str(options.num_cpus)
    cwd = options.rootdir1 + '/apps/fmm'

class Ocean_contig(LiveProcess):
    executable = options.rootdir1 + '/apps/ocean/contiguous_partitions/OCEAN'
    cmd = ['OCEAN'] #, '-p', str(options.num_cpus)]
    cwd = options.rootdir1 + '/apps/ocean/contiguous_partitions'

class Ocean_noncontig(LiveProcess):
    executable = options.rootdir1 + '/apps/ocean/non_contiguous_partitions/OCEAN'
    cmd = ['OCEAN'] #, '-p', str(options.num_cpus)]
    cwd = options.rootdir1 + '/apps/ocean/non_contiguous_partitions'

# --------------------
# Define Spec Benchmarks
# ====================

#class mcf(LiveProcess):
#    executable = options.rootdir2 + '/spec-alpha/mcf00.peak.ev6'
#    input = options.rootdir2 + '/000.input/CINT2000/181.mcf/data/test/input/inp.in'
#    cmd = ['mcf00.peak.ev6', '> inp.out 2> inp.err']
#    cwd = options.rootdir2
```

```
class bzip2(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/bzip200.peak.ev6'
    input                    =                    options.rootdir2                    +
'/000.input/CINT2000/256.bzip2/data/test/input/input.random'
    cmd = ['bzip200.peak.ev6', '> input.random.out 2 > input.random.err']
    cwd = options.rootdir2

class swim(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/swim00.peak.ev6'
    input = options.rootdir2 + '/000.input/CFP2000/171.swim/data/test/input/swim.in'
    cmd = ['swim00.peak.ev6', ' > swim.out 2> swim.err']
    cwd = options.rootdir2



class apsi(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/apsi00.peak.ev6'
    input = options.rootdir2 + '/000.input/CFP2000/301.apsi/data/test/input/apsi.in'
    cmd = ['apsi00.peak.ev6', '> apsi.out 2> apsi.err']
    cwd = options.rootdir2

class equake(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/equake00.peak.ev6'
    input = options.rootdir2 + '/000.input/CFP2000/183.equake/data/test/input/inp.in'
    cmd = ['equake00.peak.ev6', '> inp.out 2> inp.err']
    cwd = options.rootdir2

class gcc(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/gcc00.peak.ev6'
    input = options.rootdir2 + '/000.input/CINT2000/176.gcc/data/test/input/cccp.i'
    cmd = ['gcc00.peak.ev6']
    cwd = options.rootdir2
```

```
class gzip(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/gzip00.peak.ev6'
    input                    =                    options.rootdir2                    +
'/000.input/CINT2000/164.gzip/data/test/input/input.compressed'
    cmd = ['gzip00.peak.ev6', '> input.compressed.out 2> input.compressed.err']
    cwd = options.rootdir2



class mgrid(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/mgrid00.peak.ev6'
    input = options.rootdir2 + '/000.input/CFP2000/172.mgrid/data/test/input/mgrid.in'
    cmd = ['mgrid00.peak.ev6', '> mgrid.out 2> mgrid.err']
    cwd = options.rootdir2

class applu(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/applu00.peak.ev6'
    input = options.rootdir2 + '/000.input/CFP2000/173.applu/data/test/input/applu.in'
    cmd = ['applu00.peak.ev6',  '> applu.out 2> applu.err']
    cwd = options.rootdir2

class wupwise(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/wupwise00.peak.ev6'
    input                    =                    options.rootdir2                    +
'/000.input/CFP2000/168.wupwise/data/test/input/wupwise.in'
    cmd = ['wupwise00.peak.ev6', '> wupwise.out 2> wupwise.err']
    cwd = options.rootdir2

class twolf(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/twolf00.peak.ev6'
    input                    =                    options.rootdir2                    +
'/000.input/CINT2000/300.twolf/data/test/input/test.pin'
```

```
    cmd = ['twolf00.peak.ev6']
    cwd = options.rootdir2

class gap(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/gap00.peak.ev6'
    input = options.rootdir2 + '/000.input/CINT2000/254.gap/data/test/input/test.in'
    cmd = ['gap00.peak.ev6', '-l', './', '-q', '-m', '64M', '> test.stdout 2>
test.err']
    cwd = options.rootdir2

class crafty(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/crafty00.peak.ev6'
    input = options.rootdir2 +
'/000.input/CINT2000/186.crafty/data/test/input/crafty.in'
    cmd = ['crafty00.peak.ev6', '> crafty.out 2> crafty.err']
    cwd = options.rootdir2

class perlbmk(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/perlbmk00.peak.ev6'
    input = options.rootdir2 +
'/000.input/CINT2000/253.perlbmk/data/test/input/test.in'
    cmd = ['perlbmk00.peak.ev6', '-I.', '-I./lib test.pl', '> test.out 2> test.err']
    cwd = options.rootdir2

class vortex(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/vortex00.peak.ev6'
    input = options.rootdir2 +
'/000.input/CINT2000/255.vortex/data/test/input/lendian.raw'
    cmd = ['vortex00.peak.ev6', '> vortex.out 2> vortex.err']
    cwd = options.rootdir2

class lucas(LiveProcess):
```

```
    executable = options.rootdir2 + '/spec-alpha/lucas00.peak.ev6'
    input                   =                   options.rootdir2                   +
'/000.input/CFP2000/189.lucas/data/test/input/lucas2.in'
    cmd = ['lucas00.peak.ev6', '> lucas2.out 2> lucas2.err']
    cwd = options.rootdir2

class galgel(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/galgel00.peak.ev6'
    input                   =                   options.rootdir2                   +
'/000.input/CFP2000/178.galgel/data/test/input/galgel.in'
    cmd = ['galgel00.peak.ev6', '> galgel.out 2> galgel.err']
    cwd = options.rootdir2

class facerec(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/facerec00.peak.ev6'
    input                   =                   options.rootdir2                   +
'/000.input/CFP2000/187.facerec/data/test/input/test.in'
    cmd = ['facerec00.peak.ev6', '> test.out 2> test.err']
    cwd = options.rootdir2

class fma3d(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/fma3d00.peak.ev6'
    input = options.rootdir2 + '/000.input/CFP2000/191.fma3d/data/test/input/fma3d.in'
    cmd = ['fma3d00.peak.ev6', '> test.out 2> test.err']
    cwd = options.rootdir2

class sixtrack(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/sixtrack00.peak.ev6'
    input                   =                   options.rootdir2                   +
'/000.input/CFP2000/200.sixtrack/data/test/input/inp.in'
    cmd = ['sixtrack00.peak.ev6', '> inp.out 2> inp.err']
    cwd = options.rootdir2
```

```
class ammp(LiveProcess):
    executable = options.rootdir2 + '/spec-alpha/ammp00.peak.ev6'
    input = options.rootdir2 + '/000.input/CFP2000/188.ammp/data/test/input/ammp.in'
    cmd = ['ammp00.peak.ev6', '> ammp.out 2> ammp.err']
    cwd = options.rootdir2

#class art(LiveProcess):
#   executable = options.rootdir2 + '/spec-alpha/art00.peak.ev6'
#                                   #input        =        options.rootdir2        +
'/000.input/CFP2000/179.art/data/test/input/lucas2.in'
#     cmd = ['art00.peak.ev6', '[-startx 134]', '[-starty 220]', '-endx 139', '-endy
225', '-stride 2', '-scanfile /000.input/CFP2000/179.art/data/test/input/c756hel.in',
'-trainfile1   #/000.input/CFP2000/179.art/data/test/input/a10.img',   '-objects   1   >
test.out 2> test.err']
#    cwd = options.rootdir2

#===========================================================================#
np = options.num_cpus
# --------------------
# Pick the correct Splash2 Benchmarks
# ====================
def pick_benchmark(workload):
  if  workload == 'cholesky':
    bench = cholesky()
  elif workload == 'fft':
    bench = fft()
  elif workload == 'LUContig':
    bench = LU_contig()
  elif workload == 'LUNoncontig':
    bench = LU_noncontig()
  elif workload == 'radix':
```

```
  bench = Radix()
elif workload == 'barnes':
  bench = barnes()
elif workload == 'fmm':
  bench = fmm()
elif workload == 'ocean':
  bench = Ocean_contig()
elif workload == 'OceanNoncontig':
  bench = Ocean_noncontig()
elif workload == 'mcf':
  bench = mcf()
elif workload == 'swim':
  bench = swim()
elif workload == 'bzip2':
  bench = bzip2()
elif workload == 'apsi':
  bench = apsi()
elif workload == 'equake':
  bench = equake()
elif workload == 'gcc':
  bench = gcc()
elif workload == 'mgrid':
  bench = mgrid()
elif workload == 'wupwise':
  bench = wupwise()
elif workload == 'applu':
  bench = applu()
elif workload == 'gzip':
  bench = gzip()
elif workload == 'crafty':
  bench = crafty()
elif workload == 'perlbmk':
```

```
    bench = perlbmk()
  elif workload == 'gap':
    bench = gap()
  elif workload == 'twolf':
    bench = twolf()
  elif workload == 'lucas':
    bench = lucas()
  elif workload == 'art':
    bench = art()
  elif workload == 'vortex':
    bench = vortex()
  elif workload == 'galgel':
    bench = galgel()
  elif workload == 'facerec':
    bench = facerec()
  elif workload == 'fma3d':
    bench = fma3d()
  elif workload == 'sixtrack':
    bench = sixtrack()
  elif workload == 'ammp':
    bench = ammp()

  else:
    print >> sys.stderr, """The --benchmark environment variable was set to something
improper.
Use Cholesky, FFT, LUContig, LUNoncontig, Radix, Barnes, FMM, OceanContig,
OceanNoncontig, Raytrace, WaterNSquared, or WaterSpatial"""
    sys.exit(1)
  return bench

multiprocesses = []
numThreads = 1
```

```
if options.benchmark:
    workloads = options.benchmark.split(';')
    if len(workloads) >= 1:
      for workload in workloads:
          process = pick_benchmark(workload)
          multiprocesses.append(process)
          print process
    #else :
      #multiprocesses.append(workloads)

if options.bench:
    apps = options.bench.split("-")
    if len(apps) != options.num_cpus:
        print "number of benchmarks not equal to set num_cpus!"
        sys.exit(1)

    for app in apps:
        try:
            if buildEnv['TARGET_ISA'] == 'alpha':
                exec("workload = %s('alpha', 'tru64', 'ref')" % app)
            else:
                exec("workload = %s(buildEnv['TARGET_ISA'], 'linux', 'ref')" % app)
            multiprocesses.append(workload.makeLiveProcess())
        except:
            print >>sys.stderr, "Unable to find workload for %s: %s" %
(buildEnv['TARGET_ISA'], app)
            sys.exit(1)
elif options.cmd:
    #multiprocesses, numThreads = get_processes(options)
    workloads = options.cmd.split(';')
    if len(workloads) > 1:
```

```
        #process = []
        inputs = []
        outputs = []
        errouts = []
        #workload = []
        wrkld_idx = 0

        if options.input != "":
            inputs = options.input.split(';')
        if options.output != "":
            outputs = options.output.split(';')
        if options.errout != "":
            errouts = options.errout.split(';')

        for workload in workloads:
            process = LiveProcess()
            process.executable = workload
            process.cmd = workload + " " + options.options
            if inputs and inputs[wrkld_idx] :
                process.input = inputs[wrkld_idx]
            if outputs and outputs[wrkld_idx]:
                process.output = outputs[wrkld_idx]
            if errouts and errouts[wrkld_idx]:
                process.errout = errouts[wrkld_idx]
            multiprocesses.append(process)
            wrkld_idx += 1

#else:
#   print >> sys.stderr, "No workload specified. Exiting!\n"
#   sys.exit(1)
```

```
(CPUClass, test_mem_mode, FutureClass) = Simulation.setCPUClass(options)
CPUClass.clock = options.clock
CPUClass.numThreads = numThreads

# Check -- do not allow SMT with multiple CPUs
if options.smt and options.num_cpus > 1:
    fatal("You cannot use SMT with multiple CPUs!")

np = options.num_cpus
system = System(cpu = [CPUClass(cpu_id=i) for i in xrange(np)],
                physmem =SimpleMemory(range=AddrRange("3072MB")),
                membus = CoherentBus(), mem_mode = test_mem_mode)

# Sanity check
if options.fastmem:
    if CPUClass != AtomicSimpleCPU:
        fatal("Fastmem can only be used with atomic CPU!")
    if (options.caches or options.l2cache):
        fatal("You cannot use fastmem in combination with caches!")

num_processes = len(multiprocesses)

if (num_processes == 1):
   system.cpu[00].workload = multiprocesses

for i in xrange(np):
    if options.smt:
        system.cpu[i].workload = multiprocesses
    #elif len(multiprocesses) == 1:
    #    system.cpu[i].workload = multiprocesses[0]
    elif len(multiprocesses) == np:
        system.cpu[i].workload = multiprocesses[i]
```

```
    else:
        system.cpu[i].workload = multiprocesses[i % (num_processes)]

    if options.fastmem:
        system.cpu[i].fastmem = True

    if options.checker:
        system.cpu[i].addCheckerCpu()

    system.cpu[i].createThreads()

if options.ruby:
    if not (options.cpu_type == "detailed" or options.cpu_type == "timing"):
        print >> sys.stderr, "Ruby requires TimingSimpleCPU or O3CPU!!"
        sys.exit(1)

    # Set the option for physmem so that it is not allocated any space
    system.physmem.null = True

    options.use_map = True
    Ruby.create_system(options, system)
    assert(options.num_cpus == len(system.ruby._cpu_ruby_ports))

    for i in xrange(np):
        ruby_port = system.ruby._cpu_ruby_ports[i]

        # Create the interrupt controller and connect its ports to Ruby
        # Note that the interrupt controller is always present but only
        # in x86 does it have message ports that need to be connected
        system.cpu[i].createInterruptController()

        # Connect the cpu's cache ports to Ruby
```

```
        system.cpu[i].icache_port = ruby_port.slave
        system.cpu[i].dcache_port = ruby_port.slave
        if buildEnv['TARGET_ISA'] == 'x86':
            system.cpu[i].interrupts.pio = ruby_port.master
            system.cpu[i].interrupts.int_master = ruby_port.slave
            system.cpu[i].interrupts.int_slave = ruby_port.master
            system.cpu[i].itb.walker.port = ruby_port.slave
            system.cpu[i].dtb.walker.port = ruby_port.slave
else:
    system.system_port = system.membus.slave
    system.physmem.port = system.membus.master
    CacheConfig.config_cache(options, system)

root = Root(full_system = False, system = system)
m5.disableAllListeners()
Simulation.run(options, root, system, FutureClass)
```

2. GEM5 Source Code

For implementing an existing work or new techniques in GEM5, modification of source code is required. This section of Appendix B gives a brief overview of GEM5 source code structure and modification guidelines.

Being a full system simulator GEM5 has separate modules for CPU, caches, on-chip interconnects, memory controller and main memory. Consider the following path for further references:

$ROOT_PATH = gem5/src

$RUBY_PATH = $ROOT_PATH /mem/ruby

$NETWORK_PATH = **$RUBY_PATH**/network/garnet/fixed-pipeline

$MEMORY_PATH = **$RUBY_PATH**/system

$ROOT_PATH/cpu: It has the source code for cpu functionality which includes- Timing Simple CPU, Inorder CPU and Out-of-Order CPU models.

$RUBY_PATH: It has the most crucial source code to handle all requests which are originated at CPU and need to access on-chip caches (private or shared), on-chip interconnects and/or memory.

$RUBY_PATH/network: It contains the code for Garnet and Simple network-on-chip. Within Garnet, models for fixed pipeline and flexible pipeline are provided. Fixed pipeline has 5-stage virtual channel router and flexible pipeline router has flexible stages for router which can be configured in GarnetRouter.py.

$NETWORK_PATH: This directory has code for NoC components. The relevant code for NoC modules is mentioned in the table.

Table B.1: Source code for NoC modules

NoC Components	NoC module code	Comments
Input Port	InputUnit_d.*	Structure of input buffers and input port for routers can be modified.
Output Port	OutputUnit_d.*	Structure of output buffers and output port for routers can be modified.
Virtual Channels	VCallocator_d.*, VirtualChannel_d.*, OutVcState_d.*	Number/structure of virtual channels (data and command), VC allocation process can be modified.
Flits	flit_d.*, flitBuffer_d.*	Addition/deletion of a field for flits, information carried by flits can be changed.
Network links	GarnetLink_d.*, CreditLink_d.*, NetworkLink_d.*	Link bandwidth, link scheduling, flow control techniques can be modified with these files.
NoC Router (Garnet)	GarnetRouter_d.py, Router_d.*, RoutingUnit_d.*	Main router configuration, functionalities of a router, storage buffers, etc. can be modified.
Network Interface	NetworkInterface_d.*	NI can be altered by adding new functionalities or modifying existing ones.
Switch Allocation	SWallocator_d.*	Router arbitration code for switch traversal.

$MEMORY_PATH: This directory has code for memory components. The relevant code for memory modules is mentioned in the table.

Table B.2: Source code for memory modules

Memory Components	Memory module code	Comments
MemoryController	MemoryControl.*, RubyMemoryControl.*	MemoryControl is a parent class from which RubyMemoryControl is derived. RubyMemoryControl is a handle for modifying memory controllers only for ruby based memory systems. Memory configurations can be changed in RubyMemoryControl.py.
Ruby System configuration	RubySystem.py	This is a configuration file for ruby system setup (system with ruby components).
Ruby Port	RubyPort.*	This port connects the CPUs to the Ruby system consisting of caches, cache coherence protocol, NoC and memory subsystem. More information can be found here.
Sequencer	Sequencer.*	This is a entry point for the messages coming into the Ruby jurisdiction. Details can be found here.
Cache	Cache.py, CacheMemory.*	These files help modify cache structure and configuration.

***Note:** To add or delete a module to a subsystem- network or memory, the corresponding files must be added to Sconscript files belonging to that module. E.g., for addition of a new garnet network submodule in $NETWORK_PATH, $NETWORK_PATH/Sconscript must be modified.*