MANY-CORE ARCHITECTURE FOR PROGRAMMABLE HARDWARE ACCELERATOR

A Dissertation
Presented to
The Academic Faculty

By

Junghee Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2013

Copyright © 2013 by Junghee Lee

MANY-CORE ARCHITECTURE FOR PROGRAMMABLE HARDWARE ACCELERATOR

Approved by:

Dr. Jongman Kim, Advisor
Assistant Professor, School of ECE
Georgia Institute of Technology

Dr. John Copeland
Professor, School of ECE
Georgia Institute of Technology

Dr. Linda Wills
Associate Professor, School of ECE
Georgia Institute of Technology

Dr. Richard Vuduc
Associate Professor, School of CSE
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
Professor, School of ECE
Georgia Institute of Technology

Date Approved: October 2013

To my parents, my parents-in-law, my wife, and my children

ACKNOWLEDGMENTS

Foremost, I cannot begin to express my thanks to my advisor, Prof. Jongman Kim for his continuous support and encouragement during my graduate study at Georgia Institute of Technology. I have enjoyed working with him and I am glad to finish my Ph.D. degree under his guidance. I would also like to express my deepest appreciation to Prof. Chrysostomos Nicopoulos for his enthusiasm and immense knowledge. His guidance and insightful comments helped me a lot in my research and writing papers. I would also like to extend my sincere gratitute to Dr. Youngjae Kim, who allows me to have invaluable experience. I also learn a lot from him in the research and writing papers.

Besides my advisor, I would like to thank all my committee members, Prof. Linda Wills, Prof. Sudhakar Yalamanchili, who also served on the reading committee, Prof. John Copeland, and Prof. Richard Vuduc for their insightful comments and questions.

Thanks should also go to my lab mates, Prof. Hyung Gyu Lee, Mamadou Diao, Suk Chan Kang, Seungcheol Baek, and Vison Young for their advice, criticism, and helpful reviews. I very much appreciate supports from Dongsik, Sungjin, Yonghee and Haejoon.

Last but not least, I am deeply indebted to my family: my parents, parents-in-law, wife, and two sons. Without their love and support, I could not have accomplished my Ph.D. study.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 ISONET: HARDWARE-BASED JOB QUEUE MANAGEMENT FOR MANY-CORE ARCHITECTURES 3

- 2.1 Preamble 6
  - 2.1.1 The Parallel Programming Model of a modern CMP 7
  - 2.1.2 Motivation for a Conflict-Free Job Queue 8
  - 2.1.3 Motivation for Fault-Tolerance 10
- 2.2 Related Work 11
- 2.3 IsoNet: A Conflict-Free Hardware Job Queue Manager 13
  - 2.3.1 Implementation of the Local Job Queue 14
  - 2.3.2 The IsoNet Selector and Switch 18
  - 2.3.3 Single-Cycle Implementation 20
  - 2.3.4 Scalability Enhancement: Enabling Multiple Job Transfers per Cycle 23
- 2.4 Implementation 26
- 2.5 Supporting Fault-Tolerance 32
  - 2.5.1 Transparent Mode 32
  - 2.5.2 Reconfiguration Mode 33
- 2.6 Evaluation 36
  - 2.6.1 Simulation Setup 36
  - 2.6.2 Performance Analysis 40
  - 2.6.3 Beyond One Hundred Cores: Towards Many-Core CMPs 45
- 2.7 Conclusion 47

CHAPTER 3 SHARDED ROUTER: A NOVEL ON-CHIP ROUTER ARCHITECTURE EMPLOYING BANDWIDTH SHARDING AND STEALING 49

- 3.1 Preliminary Research on Optimal Flit Size 51
  - 3.1.1 Preamble 52
  - 3.1.2 Global Wires 54
  - 3.1.3 Cost of Router 56
  - 3.1.4 Latency 57
  - 3.1.5 Workload Characteristics 59

3.1.6 Throughput 61
3.1.7 Conclusion 63
3.2 Motivation for Channel/Bandwidth Slicing and the Concept of Router Sharding 64
3.3 Related Work 69
3.4 The Sharded Router Architecture - A Sliced NoC Design Employing Bandwidth- and Buffer-Stealing 70
3.4.1 The Baseline NoC Router 70
3.4.2 The Micro-architecture of the Sharded Router 72
3.4.3 The Bandwidth-Stealing Mechanism 76
3.4.4 Replacing Virtual Channels with a Buffer-Stealing Technique . . . 78
3.5 Experimental Evaluation 80
3.5.1 Simulation Framework 80
3.5.2 Performance Evaluation 82
3.5.3 Hardware Cost Analysis 90
3.6 Conclusion 94

CHAPTER 4 A PROGRAMMABLE PROCESSING ARRAY ARCHITECTURE SUPPORTING DYNAMIC TASK SCHEDULING AND MODULE-LEVEL PREFETCHING 96
4.1 Related Work 97
4.2 Motivational Example 100
4.3 The Execution Model of the Proposed MPPA Architecture 103
4.3.1 Specification 103
4.3.2 Semantics 104
4.3.3 Using the Event-Driven Execution Model 105
4.4 The Hardware Architecture 107
4.5 Architectural Support for the Execution Model 111
4.5.1 Execution Engine 111
4.5.2 Module-Level Prefetching 113
4.5.3 An Event-Driven Execution Example 115
4.6 Experimental Evaluation 117
4.7 Conclusion 121

CHAPTER 5 CONCLUSION 123

REFERENCES 125

LIST OF TABLES

Table 1 Summary of measurements from VLSI implementation 29

Table 2 Overhead of baseline and enhanced IsoNet over Intel's SCC [1] and NVIDIA's GTX 570 [2] 32

Table 3 Simulated system parameters.. 37

Table 4 Profile of RMS benchmarks 39

Table 5 System parameters 55

Table 6 Projection of the power consumption of global wires. [3, 4] 56

Table 7 Profile of the applications in the PARSEC benchmark suite [5]............... 60

Table 8 Simulated system parameters.. 81

Table 9 Summary of the main parameters of the NoC routers. "Baseline" refers to a conventional NoC router implementation, whereas "Proposed" refers to the Sharded Router 82

Table 10 Hardware cost comparison between the Baseline_2 and Proposed_2 designs 93

Table 11 Message types supported by the proposed MPPA architecture................. 112

Table 12 Simulated system parameters.. 117

Table 13 Module execution times for the benchmark applications used.................. 118

LIST OF FIGURES

Figure 1 Abstract illustration of how threads generate and consume jobs 8

Figure 2 A breakdown of the execution time and utilization of Sparse Matrix Vector Multiply 11

Figure 3 System view of the proposed IsoNet load distributor and balancer 15

Figure 4 Illustration of IsoNet Selectors and Switches 15

Figure 5 Block diagram of the architecture of one IsoNet node 16

Figure 6 Load-balancing through interaction between hardware and software 17

Figure 7 Forming a tree-based path between a Source node (S) and a Destination node (D), prior to a job transfer (R: Root node) 20

Figure 8 Longest combinational logic path within IsoNet. The delays shown in the figure are of the enhanced IsoNet implementation 22

Figure 9 Illustration of multiple job transfers per IsoNet cycle 24

Figure 10 Overview of the enhanced IsoNet design that supports multiple job transfers per IsoNet cycle 27

Figure 11 Die micrographs of the resulting VLSI implementation of the baseline IsoNet architecture, assuming a 64-core CMP (8×8 mesh) 29

Figure 12 Reconfiguring the topology and selecting a new root node candidate. . . 36

Figure 13 *Full-system* simulation results for (a) loop-level and (b) task-level parallel benchmarks 41

Figure 14 Profile of the execution time of Gauss-Seidel (GS) 43

Figure 15 Sensitivity analysis of the two benchmark types (loop-level/task-level parallel) on the average job size. The numbers below the x axes of the graphs (4, 8, ..., 64) refer to the number of processing cores 46

Figure 16 *Trace-driven* simulation results for (a) loop-level and (b) task-level parallel benchmarks. The numbers below the x axes of the graphs (128, 256, ..., 1024) refer to the number of processing cores 46

Figure 17 The assumed NoC router architecture and its salient parameters [*v*: number of virtual channels per port, *d*: buffer depth, *c*: physical channel width in bits, *p*: number of ports, *t*: number of pipeline stages] 53

Figure 18 Splitting a packet into flits [h: header overhead, l: payload size, f: flit size, N: number of flits] ... 54

Figure 19 The increasing cost of a router with increasing flit size (width). The reference line indicates a linear increase, whereby the cost increases at the same rate as the flit size ... 58

Figure 20 Overall speedup with increasing flit size (width) ... 60

Figure 21 Physical channel utilization with increasing flit size (width) ... 63

Figure 22 Throughput comparison of one physical network with wide flits vs. two physically separated networks with narrow flits ... 64

Figure 23 Abstract visualization of the size of the two main packet types generated by the MOESI-CMP-directory implementation of the GEMS simulator [6]. In general, the types of messages traversing the NoC of a CMP are dependent on the employed cache coherence protocol ... 65

Figure 24 Conceptual view of the NoC physical channel utilization assuming various router micro-architectural approaches ... 67

Figure 25 A conceptual overview of the baseline router's micro-architecture. This is a typical input-buffered NoC router design, where the Virtual Channel (VC) buffers employ a *parallel* (rather than serial) FIFO implementation. The FIFO order is maintained by the pointer logic controlling the input DEMUX and output MUX ('4' and '5' in diagram above) ... 73

Figure 26 A conceptual overview of the Sharded Router's micro-architecture. The proposed design has 4 physically separated networks (called "slices") and each network has a physical channel width of 32 bits. In this case, each slice has two Virtual Channel (VC) FIFO buffers ... 74

Figure 27 An example illustration of the Sharded Router's bandwidth-stealing mechanism. Flits residing in Slice 0 may "steal" the physical channel bandwidth of idle Slices 1 and 2, thus fully utilizing the available physical links ... 76

Figure 28 The datapath of flits stealing bandwidth from other (idle) slices. In this example, three flits depart VC0 of a particular slice, in the same clock cycle, by stealing bandwidth from two other slices. The flits are redirected to their original VC and slice upon arrival at the downstream router ... 77

Figure 29 An example illustration of the Sharded Router's buffer-stealing technique. The 'B' flits in Router 0 can temporarily "steal" the buffer of Slice 2 in Router 1 to bypass the HoL blocking incurred by the 'A' flits. The 'B' flits can then return to their original slice (Slice 1) in downstream Router 2 80

Figure 30 Performance comparison under two synthetic traffic patterns 84

Figure 31 Performance comparison with *enlarged* baselines having deeper buffers (Baseline$_b$ and Baseline$_{4b}$) and more VCs (Baseline$_v$ and Baseline$_{4v}$). . . 86

Figure 32 Performance comparison with wider physical channel widths........ 87

Figure 33 Physical channel utilization. The "Effective" utilization curve is the *real* utilization of the baseline router design, when the non-utilized bits within a flit are accounted for in the calculations 87

Figure 34 The performance contributions of the two stealing techniques employed in the Sharded Router architecture. The "Sliced" curve refers to a bare-bones sliced (sharded) router with no stealing mechanisms........ 89

Figure 35 Performance evaluation using a full-system, execution-driven simulation framework running real multithreaded applications from the PARSEC benchmark suite [5] on a 64-core CMP........ 91

Figure 36 Sensitivity analysis on the injection rate of the *additional dummy traffic* injected alongside the real application traffic of the multithreaded workload. The multithreaded benchmark used here is blackscholes 91

Figure 37 Performance comparison in terms of *time* (instead of *cycles*), in order to account for the longer critical path in the proposed router. One clock cycle in the baseline router is 1.36 ns, while that of the proposed router is 1.46 ns (as per the hardware synthesis results of Table 10) 94

Figure 38 A high-level overview of a processor architecture employing a Massively Parallel Processing Array (MPPA) as a programmable hardware accelerator 98

Figure 39 Illustration of the parallelism exhibited in the quicksort algorithm........ 101

Figure 40 Inefficiency of the SIMD model for applications with irregular computation kernels........ 103

Figure 41 Module diagram of the quicksort algorithm, as specified using the proposed event-driven execution model........ 106

Figure 42 The proposed MPPA microarchitecture consists of several identical *tiles* interconnected using an on-chip interconnection network 108

Figure 43 Block diagram of a *single* core tile of the many-core MPPA architecture shown in Figure 42 109

Figure 44 Sequence diagram of the prefetching process of the proposed MPPA architecture. Notice how prefetching can hide both the overhead of the execution engine and the access latency to the device memory 114

Figure 45 Illustrative example of an event-driven execution of the quicksort algorithm 116

Figure 46 Average access times of the scheduler (normalized to the average execution time of the modules), and average utilization of the processing elements (i.e., the core tiles) 120

Figure 47 The impact on performance of the number of core tiles designated to serve as part of the *execution engine*. “Util(k)” and “Execution time(k)” denote the tile utilization and the total execution time, respectively, when the number of core tiles devoted to the execution engine is k. The benchmark used is CED 120

CHAPTER 1
INTRODUCTION

As the further development of single-core architectures faces seemingly insurmountable physical and technological limitations, computer designers have turned their attention to alternative approaches. One such promising alternative is the use of several smaller cores working in unison as a programmable hardware accelerator. The most popular and widely used embodiment of this concept is the graphics processing unit (GPU). While initially devised as a graphics-only co-processor, it is now envisioned as a powerful processor that can undertake more diverse duties. This realization has given rise to the emerging paradigm of general-purpose computing on GPUs (GPGPU). A programmer may now use the GPU as a general-purpose accelerator. The latest Sandy Bridge microarchitecture [7] of Intel and the Fusion (Llano) [8] architecture of AMD both integrate a GPU engine on the same die as the general-purpose CPU cores. It is clear that the vast - and, as yet, largely untapped - potential of hardware accelerators (such as the GPU) is coming to the forefront of computer architecture. To be more general, the accelerator consisting of many small cores is called as a massively parallel processing array (MPPA) [9] in this thesis.

There are many challenges that must be addressed for the MPPA to be realized in practice. In this thesis, the following challenges, which are vital for fully utilizing tens or hundreds of cores in the MPPA, will be studied.

Through empirical studies, it was observed that there is significant variation in utilization of the processing elements when the multithreading programming model is adopted. Imbalanced distribution of workloads across the MPPA constitutes wasteful use of resources, which results in degrading the performance of the system. It was reported that the existing software-based load-balancing techniques do not scale well with an increasing number of cores [10]. In this thesis, a hardware-based load-balancing technique is proposed.

It is a firm trend that the number of cores keeps increasing. To facilitate efficient communication among ever increasing number of cores, a scalable communication network is imperative. Packet switching networks-on-chip (NoC) is considered as a viable candidate for scalable communication fabric. The size of flit, which is a unit of flow control in NoC, is one of important design parameters that determine latency, throughput and cost of NoC routers. How to determine an optimal flit size is studied in this thesis and a novel router architure is proposed, which overcomes a problem related with the flit size.

This thesis also includes a new execution model and its supporting architecture. An event-driven model that is an extension of hardware description language (HDL) is employed as an execution model. The dynamic scheduling and module-level prefetching for supporting the event-driven execution model are evaluated.

CHAPTER 2

ISONET: HARDWARE-BASED JOB QUEUE MANAGEMENT FOR MANY-CORE ARCHITECTURES

The last few years have witnessed a paradigm shift in computer architecture away from complex, high-frequency, deeply pipelined wide-issue microprocessor cores to arrays of leaner, simpler and smaller cores working in tandem. Recent trends in both industry and academia are indicating that the number of on-chip processing cores will continue to rise in the foreseeable future.

The most popular programming model for multicore systems is *multithreading*, whereby a programmer can parallelize an application by spawning a separate thread for each parallel task. Thread creation, however, comes at a cost, which becomes difficult to amortize as the granularity of exploitable parallelism wanes. In applications characterized by *fine-grained parallelism* [11], the execution time of each thread is relatively short. As used in this thesis, the term *fine-grained parallelism* refers to parallel applications that consist of very small parallel tasks [11]. Due to the small size of these tasks, the overhead of spawning new threads and, subsequently, context switching between them becomes unwarranted [11]. Rather than creating new *threads*, the application may instead generate *jobs* in order to reap the benefits of fine-grained parallelism. It is important to note the fine distinction between the two above-mentioned terms: a *thread* comprises a set of instructions and states of execution of a program, while a *job* is composed of a set of data to be processed by a thread.

Existing techniques for implementing the job queue face scalability issues as the number of processing cores grows into the many-core realm (i.e., tens or even hundreds of cores). Recent research [10] has also indicated that previously proposed software-based job queue managers and load distribution techniques face scalability issues on GPU, which currently have more than one hundred cores (even though the definition of a "core" in the

GPU domain is different than in chip-level multiprocessors (CMPs)). Those techniques are predominantly software-based and are supported by general-purpose atomic operations in hardware. Existing mechanisms are not scalable, principally because of *conflicts*. Conflicts occur when multiple cores are trying to access the job queue simultaneously. To protect the data structure of the job queue from corruption, a lock mechanism is used, so that only one core may update the data structure at any given time. Lock-free mechanisms [12, 13, 14] alleviate the inefficiencies of locking by not blocking other cores when one core is updating the data structure. However, if a conflict is detected, the task queue should be updated again. In the presence of multiple, repeated conflicts, the task queue must, consequently, be updated several times. This pathological scenario may incur even more overhead than the lock mechanism. Thus, lock-free mechanisms do not solve the issue of conflicts in the task queue. Hardware-based approaches [11, 15] can reduce the probability of conflicts, but they cannot eliminate conflicts, either.

One more vital requirement in modern CMPs is fault-tolerance. It is a well-known fact that as technology scales toward conventional physical limits, transistors are steadily becoming more unreliable [16]. It is expected that in the many-core designs of the near future some provisions must be made to tolerate the presence of faulty processing elements. Within the context of job queue management, the aforementioned existing hardware-based techniques [11, 15] do not provide any fault-tolerance capability. More specifically, in the case of Carbon [11] - the current state-of-the-art in hardware-based job queue management - a centralized global task unit (GTU) is employed, which may potentially constitute a single point of failure. A fault within the GTU may inhibit the whole system from utilizing the hardware-based load-balancing mechanism.

In an effort to provide both a *scalable* and a *fault-tolerant* job queue manager, IsoNet[1] is proposed, which is an efficient hardware-based dynamic load distribution engine that enhances concurrency control and ensures uniform utilization of computational resources.

[1]The prefix in the name IsoNet is derived from the Greek “isos”, meaning “equal”. Hence, IsoNet is a *network* that maintains *equal* load between the processing cores

This engine is overlaid on top of the existing CMP infrastructure, it is completely independent of the on-chip interconnection network, and it is transparent to the operation of the system. The hardware takes charge of managing the list of jobs for each processing element and the transferring of job loads to other processing elements in order to maintain balance. More importantly, it also provides extensive fault tolerance to the operation of load balancing.

The main contributions of this work are:

- A very lightweight *micro-network of on-chip load distribution and balancing modules* - one for each processing element in the CMP - that can rapidly transfer jobs between any two cores, based on job status. This IsoNet network uses its own clock domain and can run at speeds that are significantly lower than the CPU speed, thus requiring scant energy resources.

- Whereas the proposed IsoNet mechanism is centralized in nature (i.e., it relies on a single node to coordinate job transfers), any node in the system can undertake the coordinators duties in case of malfunction or partial failure. This dynamic load distributor is architected in such a way as to avoid a single point of failure. Furthermore, we demonstrate why the chosen centralized scheme is *scalable even for designs with more than a thousand cores*, while achieving much higher efficiency and accuracy in concurrency management than a distributed approach.

- The hardware-based load distributor provides extensive *fault-tolerance support*. The proposed architecture can handle two different kinds of fault: malfunctioning processing cores (CPUs) and malfunctions within the IsoNet network itself. Through a Transparent Mode of operation, non-functioning CPU cores are *hidden* from the load distribution mechanism, while a Reconfiguration Mode reconfigures the IsoNet fabric in order to bypass faulty load-balancing nodes. In both cases, seamless load distribution and balancing operations are ensured even in the presence of faults.

- A complete hardware implementation of the proposed design is presented. The IsoNet architecture is fully implemented in a HDL and it is then passed through a detailed application-specific integrated circuit (ASIC) design flow using commercial standard-cell libraries in 45 nm technology.

- In order to ascertain the validity of our experimental results, we employ a full-system simulation framework with *real application workloads* running on a commodity operating system with only partial modifications. We evaluate the scalability of various dynamic load balancers in CMPs with 4 to 64 processing cores. For long-term scalability analysis, we also employ a cycle-accurate trace-driven simulator for experiments with CMPs up to 1024 cores.

This chapter is organized as follows. The following section presents a preamble on the typical CMP programming model and provides motivation on why a hardware-based dynamic load balancer is imperative in order to fully exploit the resources of the CMP. Section 2.2 summarizes the most relevant related work in the domain of job queue management. In Section 2.3, the proposed IsoNet architecture is presented and analyzed. Section 2.4 analyzes the hardware cost of the IsoNet mechanism through actual implementation. IsoNet's extensive fault-tolerant features are described in Section 2.5, while Section 2.6 describes the experimental methodology of this work and the accompanying results. Finally, concluding remarks are made in Section 2.7.

2.1 Preamble

This section briefly describes the typical parallel programming model of a CMP and culminates with the motivation for the proposed IsoNet mechanism by presenting some key limitations of existing techniques.

2.1.1 The Parallel Programming Model of a modern CMP

A very popular programming model for multicore chips is *multithreading*. A programmer can extract parallelism by segregating the code into independent parts and creating distinct threads for each such part. Creating multiple threads as a means to exploit parallelism may become problematic in applications characterized by what is known as *fine-grained parallelism*. The threads of such applications tend to have very short execution times. Thus, creating new threads and context-switching between them incurs significant overhead [11]. A typical and well-known set of benchmarks that exhibit abundant fine-grained parallelism is the recognition, mining, and synthesis (RMS) benchmark suite [17]. RMS comprises a collection of important emerging applications [11].

Loop-level parallel benchmarks in RMS are independent of the input data. The execution times of all threads are expected to be almost the same. Therefore, threads can be partitioned at compile-time and static load-balancing can often outperform dynamic load-balancing. However, simulation results indicate that there still exists significant *variation* in the execution times of the threads, which is incurred by factors *outside* of the application. Specifically, background operating system tasks - such as interrupt service routines and daemons - frequently preempt threads and delay their execution. Such preemption affects the execution time, if static load-balancing is employed, because the execution time is bounded by the slowest thread. Since fine-grained parallel benchmarks are dominated by short jobs, they are especially vulnerable to such interruptions. The need for *dynamic* load balancing is clearer in the case of *task-level parallel* benchmarks. Since their execution is depedent on the input data, which cannot be determined at compile-time, the load should be maintained at *run-time*.

Job queueing is an alternative way to exploit fine-grained parallelism. The applications simply creates jobs, instead of threads. As previously mentioned, a *thread* is a set of instructions and states of execution of a program, while a *job* is a set of data that is processed by a thread. This concept is illustrated by Figure 1. Note that only one thread

is created for each core at initialization time. Each thread then generates and consumes jobs during run-time, whenever necessary. Because there is only one thread per core and threads are not created, nor destroyed during run-time, the overhead of creating threads and context-switching between them can be reduced.

The main issue with this technique is the inevitable load imbalance among cores, which must be handled by the *job queue*. The job queue can be implemented as a number of distinct distributed queues, with each core having its own local queue. The number of jobs in the local queues may not always be uniform, because of load imbalance. In order to maintain balance and evenly distribute work between cores, jobs must be transferred between the various queues. However, once a job is fetched by a thread, it is *not* transferred until it is completed. The unprocessed jobs in the queue may be transferred to another core's queue before commencement of execution. Thus, the job transfer does not have an adverse impact on cache performance.

2.1.2 Motivation for a Conflict-Free Job Queue

While the introduction of a job queue would reduce the overhead incurred by run-time thread creation and context-switching, we have observed that job queues are marred by *conflicts*. In this context, conflict means idle time for a processor that is forced to wait before accessing the job queue. In fact, the presence of conflicts in the job queue is the major performance bottleneck as the number of processors increases. Conflicts occur when

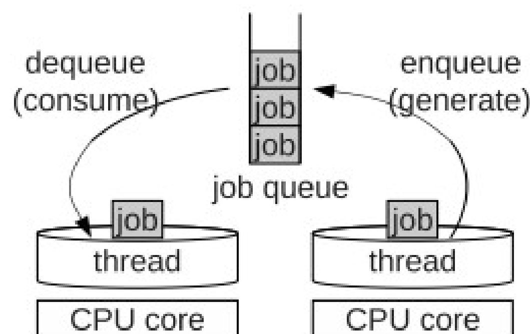


Figure 1: Abstract illustration of how threads generate and consume jobs.

multiple processors try to update the same data structure. To ensure the integrity of the data structure, only one processor is allowed to update it, and the others must wait. This situation still occurs when utilizing the popular technique of *job stealing*, as will be described in Section 2.2. If a processor tries to steal a job from a local job queue, which happens to be simultaneously accessed by another processor, then a conflict will occur. Even if the job queue is implemented in hardware (see section 2.2 for an example), conflicts are still inevitable, because even a hardware implementation cannot allow multiple processors to update the same data structure simultaneously.

The probability of conflict is affected by the duration and the frequency of the update. The longer it takes to perform an update of the data structure, the higher the probability of conflict is. The update duration can be reduced by implementing the job queue in hardware. Furthermore, the more frequently a queue is updated, the more conflicts tend to occur. One way to alleviate this issue is to distribute the queues in order to reduce the chance of multiple entities attempting to update the same data structure. Finally, the execution time of jobs also affects the frequency of updating, because the job queue needs to be accessed more frequently if it holds shorter jobs.

Fine-grained parallelism is highly vulnerable to conflicts, because the associated job sizes are very small, as previously explained. The short job sizes cause frequent accesses to the job queue. Figure 2 illustrates the conflict issue through a breakdown of the execution time of Sparse Matrix Vector Multiply, which is a prime example of an algorithm that can be implemented using multithreading. The results of Figure 2 are derived from simulations of CMPs with processing cores ranging from 4 to 64. In each configuration, there are as many threads as there are processing cores, and a total of 65,536 jobs are sorted before completion. The job queue is distributed and implemented in software, with each core having its own job queue. Job queue management is implemented with the job-stealing technique. The details of the simulation environment will be presented later on in Section 2.6.

The height of each bar in Figure 2 indicates the average execution time for all threads.

The bottom part of each bar corresponds to the pure processing time spent on the jobs, the middle part represents the time spent on stealing a job from a job queue, and the top part corresponds to the waiting time due to conflicts. In order to steal a job, a thread needs to visit the local queues of other threads one by one. Since the local queues are implemented in software, they are located in memory and one can allocate a local queue to every thread. For fair comparison with hardware-based queues, we allocate one local queue to every core. The time taken to visit the other local queues and to access a queue that has a spare job is included in the middle segment of each bar in Figure 2, while all conflicts that occur during the accessing of the queue are included in the top segment of each bar.

Obviously, in a system with *scalable* job queue management, the time spent on processing jobs should dominate the execution time, regardless of the number of processing cores. However, one can clearly see that the percentage of time wasted on conflicts keeps increasing steadily with the number of CPUs. Extrapolating this worrisome trend, conflicts will eventually dominate the execution time when the number of processors exceeds 64. Hence, job queue conflicts are expected to become show stoppers and precipitate into a major performance bottleneck as we transition into the many-core era. The need for a *conflict-free* job queue management system to balance job load across a large number of processors is imperative and of utmost importance, if architects wish to achieve scalable designs with hundreds of cores.

2.1.3 Motivation for Fault-Tolerance

As technology scales relentlessly into unchartered territories, it has been shown that the probability of faults increases exponentially [16]. Given that IsoNet is targeted at CMPs with tens or even hundreds of processing cores, the transistor counts involved will be in the range of billions. At such immense integration densities, fault-tolerance will not be a *luxury*, but, instead, a *necessity*. Systems will be expected to tolerate malfunctioning

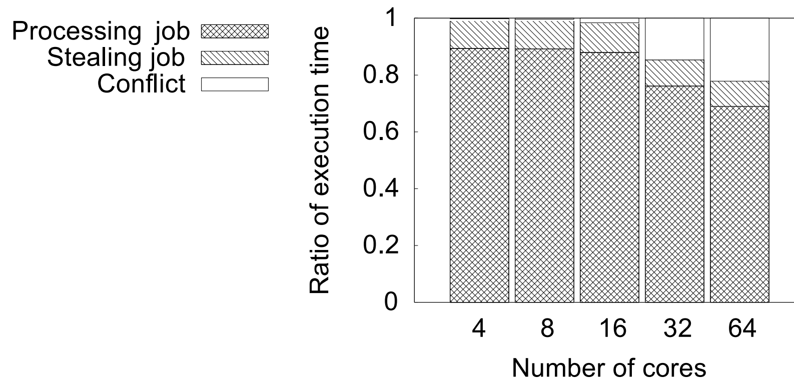


Figure 2: A breakdown of the execution time and utilization of Sparse Matrix Vector Multiply.

elements and provide reliable computation amidst unreliable constituent components. Potential faults in the load-balancing hardware could wreak havoc in the overall system operation, since performance will degrade substantially due to inefficient utilization of the hardware. It is, therefore, vital for the hardware-based mechanism to be equipped with a proper protection scheme. Moreover, if the load-balancing hardware cannot function properly in the presence of faulty processor cores, the system may even produce incorrect results. The load distribution engine should be able to handle both CPU faults and faults within its own hardware.

2.2 Related Work

Job queues may either be centralized or distributed. For centralized job queues, a synchronization mechanism is critical, which plays a key role in overall performance, because of the possibility of conflict when multiple processing elements try to update the shared job queue simultaneously. The simplest way of synchronizing accesses is to use locks or semaphores. However, being a blocking method, this technique blocks all other processing elements while one processing element accesses the job queue, regardless of whether the others are actually trying to update the job queue or not. More importantly, the performance of a lock-based centralized task queue implementation does not scale well with the

number of processing elements, as shown by experiment [10, 18]. Non-blocking methods [12, 13, 14], on the other hand, allow multiple processing elements to access the task queue concurrently. Only if they actually try to update the task queue at the same time do they keep retrying until all of them succeed one by one. Even though these methods reduce conflicts to some degree, it has been shown that they are also not scalable [10, 18].

The alternative is to distribute the job queue: each processing element gets its own local job queue. In this case, load balance is maintained by job-stealing [19, 20]. If a processing element becomes idle and there is no job in its job queue, it tries to steal a job from another queue. In this way, conflicts are drastically reduced, while load balance is also maintained. However, worst-case execution time may grow proportional to the number of processing elements, because of the way the search for job stealing is performed, which also affects the average execution time. A stealing node has to visit nodes sequentially, until a job is found. This process may require the traversal of many nodes when jobs are not evenly distributed.

Another category of related work is hardware implementation of the job queue. However, most of the work in this field concentrates on the hardware implementation of a scheduler [21, 22]. In other words, the focus has been on implementing in hardware the scheduling policies that are traditionally handled by software for scheduling *threads*. Load distribution and balancing of *jobs* are not addressed.

Carbon [11] implements the job queue in hardware by employing centralized task queues (contained in GTU). To hide latency between the queues and the cores, Carbon uses task pre-fetchers and small associated buffers close to the cores (called local task units (LTUs)). However, as the cores scale to well over one hundred, conflicts at GTU are expected to still be excessive, because Carbon does not address conflicts. More importantly, though, Carbon does not provide any fault-tolerance capabilities. On the contrary, IsoNet provides extensive protection from both CPU faults and faults within the load distribution micro-network itself.

Somewhere in-between software- and hardware-based techniques sits a *software-mostly*

approach [15]. The authors of this work introduce asynchronous direct messages (ADM), a general-purpose hardware primitive that enables fast messaging between processors by bypassing the cache hierarchy. By exploiting ADM, the proposed mechanism achieves comparable performance with hardware-based approaches, while providing the added flexibility of software-based scheduling algorithms. However, ADM does not address conflicts, nor fault-tolerance.

Finally, the Rigel architecture [23] incorporates hierarchical distributed queues to feed one thousand cores with jobs. Its application programming interface (API) is an extension of Carbon [11], but Rigel relies mostly on software with minimal support from hardware (the task queues and associated management are implemented in software). Although this software-managed architecture is shown to scale with up to a thousand cores *for some applications*, the design is *not* evaluated with benchmarks exhibiting fine-grained parallelism, which suffer from excessive conflicts.

2.3 IsoNet: A Conflict-Free Hardware Job Queue Manager

IsoNet consists of a number of load-balancing nodes (one such node for each processing element in the system), arranged as a mesh-based micro-network overlaid on top of the existing CMP infrastructure, as shown in Figure 3 (the IsoNet nodes are the small squares labeled as “I”). Note that IsoNet is a distinct micro-network that is totally independent of any existing on-chip interconnection network. In other words, the load-balancing mechanism does not interfere with the activities of the CMP interconnection backbone.

Each IsoNet node comprises three main modules: (1) a Dual-Clock Stack, (2) a Switch, and (3) two Selectors, as shown in Figure 4 (the micro-architecture of one IsoNet node is depicted in Figure 5). The Selectors’ job is to choose the source and destination nodes of the next job to be transferred, as part of load balancing. Two Selectors are needed, one - the Max Selector - to choose the node with the largest job count (i.e., the source of the next job transfer) and one - the Min Selector - to choose the node with the smallest job count

(i.e., the destination of the next job transfer). The Switch configures itself in such a way as to make a path between the source and destination nodes. The Dual-Clock Stack is the task queue, where jobs are stored. As the name suggests, the Dual-Clock Stack has two clock domains: one is for the Switch and the other is for the CPU subsystem. This characteristic allows IsoNet to accommodate processing elements with different operating frequencies. If a node is chosen by the Selector to be a source or a destination, its Switch is configured to route information to the Dual-Clock Stack.

Note that the clock of the IsoNet micro-network is independent of the CPU clock domain to not only allow various CPU operating frequencies, but also to be able to operate at a frequency that is much lower than typical CPU frequencies. All IsoNet nodes are implemented in a single clock domain and are assumed to be synchronous.

Our hardware implementation of the load distribution and balancing engine is logically and physically located outside of the CPU core. In other words, IsoNet is a kind of peripheral, much like direct memory access (DMA) controllers, from the perspective of the CPUs. Processing cores are treated as black boxes; thus, the proposed engine can be retrofitted to any kind of CPU architecture.

2.3.1 Implementation of the Local Job Queue

The local task queue is managed by the Dual-Clock Stack. It has two interfaces; one is for the CPU and the other is for the Switch of the load balancer. As previously mentioned, the two interfaces may use different clocks. However, because of this asynchronous capability, the stack was designed in such a way as to prevent the so called metastability problem, by using various circuit techniques [24]. The circuit implementation of the Dual-Clock Stack is beyond the scope of this paper.

IsoNet maintains balance through a close interaction between hardware and software, as illustrated abstractly in Figure 6. The figure also shows the principal difference between existing software-based load-balancing techniques and our hardware-based approach. More

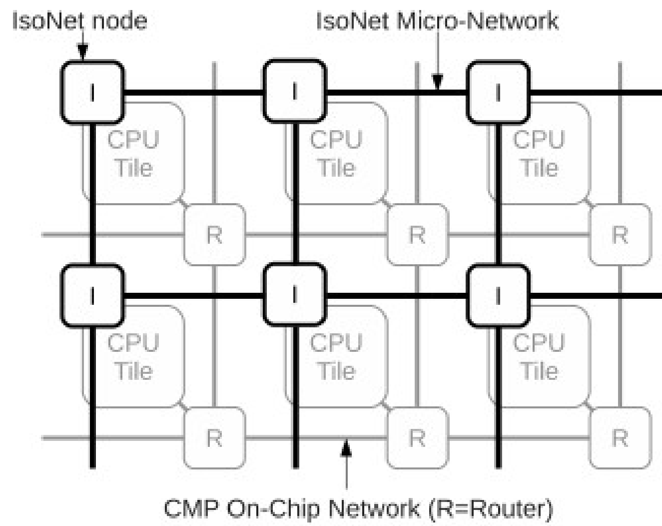


Figure 3: System view of the proposed IsoNet load distributor and balancer.

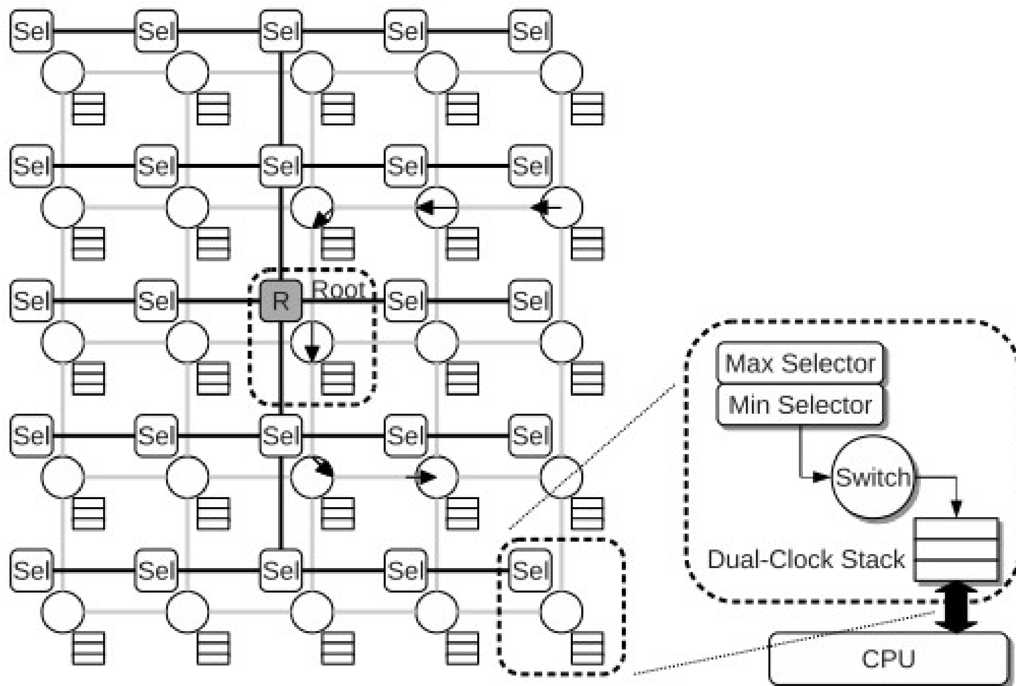


Figure 4: Illustration of IsoNet Selectors and Switches.

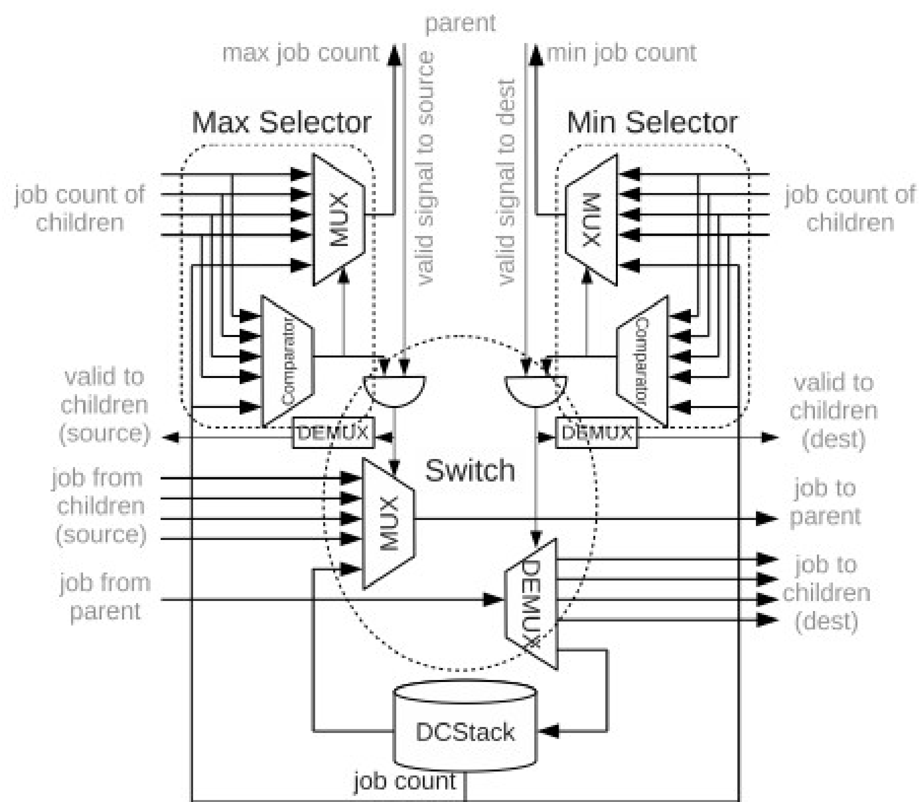


Figure 5: Block diagram of the architecture of one IsoNet node.

specifically, In software-based load-balancing, both scheduling and load-balancing are handled by the software. Consequently, the software overhead becomes significant as the number of CPUs grows. On the other hand, in the proposed load-balancing technique, the load-balancing is handled exclusively by the hardware. Therefore, the overhead of the software is substantially reduced. While the CPU cores are executing jobs, the load distribution and balancing engine (IsoNet) maintains balance by checking load imbalance among processing elements in every IsoNet clock cycle. Of course, the IsoNet clock need not be as fast as the CPU clock. In fact, it can be orders of magnitude slower.

As mentioned in Section 2.2, job-stealing is a popular approach to maintaining load balance in distributed queues. Carbon's hardware-based approach [11] also employs job-stealing. Typically, in conventional job-stealing implementations, a queue steals a job only when the queue itself is empty. However, as the number of processors grows, the time spent on *searching* for a job to steal also tends to grow, because each node in the system must be visited one-by-one until a job is found. In contrast, IsoNet maintains balance in a more *proactive* way: it constantly balances the job load through transfers between the queues, thereby avoiding the pathological situation where a local queue is totally empty while others are backlogged. In this manner, IsoNet minimizes the time wasted on *searching* for a job and greatly reduces the probability of conflicts, as will be demonstrated later on. This proactive initiative is precisely the reason why IsoNet checks for load imbalance in every IsoNet clock cycle; rather than wait until escalating load imbalance leads to empty queues that must steal jobs from other queues, IsoNet is constantly monitoring for imbalance and

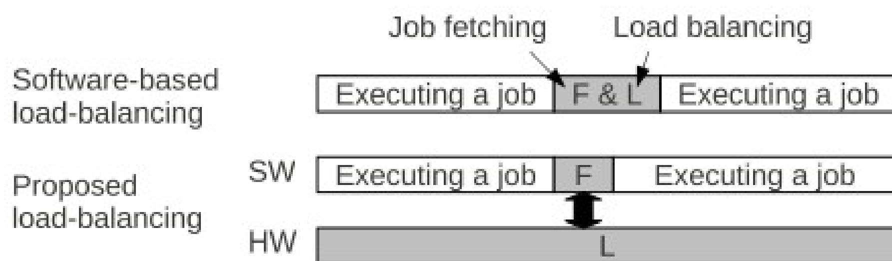


Figure 6: Load-balancing through interaction between hardware and software.

preemptively transfers jobs to maintain evenly occupied queues at all times. In this fashion, the latency of transferring jobs is hidden while the CPUs are executing jobs.

The Dual-Clock Stack (i.e., the job queue) provides several *job entries*. Each job entry comprises a 32-bit integer. From the viewpoint of the programmer, a job corresponds to arguments to a function call. Since the bit-width of each job entry directly affects the hardware cost, we decided - without loss of generality - to restrict each job entry to 32 bits. If the job cannot fit into this hardware structure, the job is stored in global memory and a pointer to its location is, instead, stored in the corresponding Dual-Clock Stack job entry.

2.3.2 The IsoNet Selector and Switch

The IsoNet load-balancing network is physically organized as a mesh, as shown by the light-colored lines in Figure 4. The mesh topology allows the IsoNet switches to transfer jobs between any two processing cores within the CMP.

However, the IsoNet Selectors are logically interconnected as a tree, as shown by the dark-colored lines in Figure 4. Each row of Selectors is connected horizontally in a line, and there is a single vertical connection along the column of Selectors that includes the root node (i.e., the load-balancing coordinator). The logic components comprising the root node are merely an adder and a comparator used to determine whether a job transfer is necessary or not. A job transfer is triggered only if the difference between the largest and smallest job counts is greater than one. The resulting tree is not a balanced tree, but it provides the shortest route from every node to the root node. Any node in IsoNet can be chosen to be the root node, but the most efficient assignment is one that places the root node as close to the middle of the die (assuming the processing cores are uniformly distributed across the die) as possible. As the coordinating entity, the root node ultimately decides the source and destination nodes of all job transfers, and notifies the affected nodes. The way load-balancing is attained is as follows:

During selection, each node sends its job count to its parent node. The parent node

compares the job counts of its children nodes and the node itself (through the use of the two Selectors). The parent node determines the node with the largest/smallest job count, it "remembers" which node is selected, and sends the job count of the selected node to its parent node until the root node is reached. The root node finally picks the source and destination nodes based on the largest and smallest job counts, respectively. Subsequently, the root node sends two valid control signals to the nodes which sent these minimum and maximum job counts; one valid signal is for the source and the other for the destination. These signals propagate outward from the root node to the selected source-destination pair. As each parent node receives the valid signal, they recursively forward it to the "remembered" child node and configure their Switch module to point in that direction. This process continues until the source and destination nodes are reached. In this fashion, the source and destination nodes are selected and - at the same time - a path from the source to the destination is formed on the IsoNet tree. An example of such a tree-based path from source to destination is illustrated in Figure 7. The number in the bottom right box of each node indicates the job count of that node, while the numbers in the top left boxes are the selected maximum and minimum job counts among that node and its children nodes. As can be seen in the figure, the Switches form a path from the node with the largest job count (node S) to the node with the smallest job count (node D). The path is along the Selector tree and passes through the root node. The root node initiates a job transfer only if the difference in the job counts between the source and destination is greater than one. This prevents unnecessary/oscillatory transfers.

Routing only on the tree introduces detouring paths instead of the shorter direct paths that can be obtained by a dimension-order routing (DOR) algorithm. Detouring paths incur higher power consumption. However, tree-based routing can be equipped with fault-tolerance mechanisms with minimal hardware cost, because it exploits the fault-free Selector tree that is formed by IsoNet's Reconfiguration Mode. The fault-tolerant features of IsoNet are described in detail in the following section.

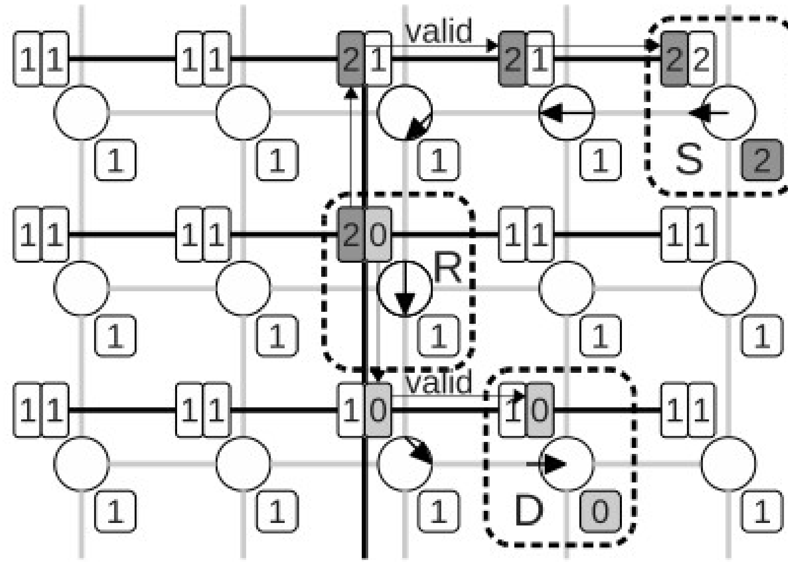


Figure 7: Forming a tree-based path between a Source node (S) and a Destination node (D), prior to a job transfer (R: Root node).

2.3.3 Single-Cycle Implementation

IsoNet was designed to complete any job transfer within a single IsoNet clock cycle. Single-cycle implementation can be achieved with minimal hardware cost, as will be demonstrated in this section.

In order to achieve such a single-cycle implementation, all IsoNet nodes must be implemented in purely combinational logic. This allows the load balancer to operate at the granularity of a single IsoNet clock cycle. Note again that an IsoNet clock cycle is much longer than a CPU clock cycle. In order to calculate the maximum feasible clock frequency of the IsoNet network, one needs to analyze the longest possible logic path (i.e., the critical path) of the load balancer. An analytical model of calculating the longest path is hereby presented.

It is shown in Figure 8 that the longest path of combinational logic that should be completed in a single IsoNet cycle. The IsoNet leaf nodes send their current job count to the parent nodes (the job counts come directly from registers). After a wire delay W_{count}, the information arrives at the parent node. The Selector in the parent node compares the counts and chooses the selected count (largest/smallest job count) after a gate delay $G_{selector}$.

If there are $N \times N$ nodes in the CMP, the counts will go through at most $N(= N/2 + N/2)$ nodes until they arrive at the root node (maximum Manhattan distance). After a root node delay of G_{root}, the valid signals are sent to the source and destination nodes. The valid signals go through the root node logic in the opposite direction of the tree from the root to the leaves. The wire delay from one node to the next node (i.e., between repeaters) is W_{valid}. The maximum number of nodes the valid signals have to traverse is, again, N. After a delay of G_{config}, each IsoNet Switch calculates its configuration signals, which are sent to the switch logic with a delay W_{config}. Then, the job to be transferred and the ready signals are exchanged between the source and the destination. The job and ready signals should go through $2N$ switches at most (longest Manhattan distance from the source to the root (N) and from the root to the destination (N) in an $N \times N$ mesh). The delay of a switch is G_{switch} and the wire delay from a switch to the next switch is W_{job}. Because the job and ready signals propagate simultaneously, W_{job} includes delays of both jobs and ready signals. Then, the total delay of the combinational logic path can be estimated by the following equation.

$$D_{total} = N(W_{count} + G_{selector} + W_{valid} + G_{root}) + G_{config} + W_{config} + 2N(W_{job} + G_{switch}) \quad (1)$$

We estimate the wire delays W_{count}, W_{valid}, and W_{job} by the Elmore delay model [25]. We take typical 65 nm technology parameters from [26]. They can be estimated by the following equation.

$$W_{count} = r_d(cL/N + z_d) + rL/N(0.5cL/N + z_d) = W_{valid} = W_{job} \quad (2)$$

Parameters r_d and z_d are technology parameters and their values are 250Ω and 50fF. c and r are unit-length capacitance and resistance which are also technology parameters and their value is 0.2pF/mm and 0.4kΩ, respectively. L/N is approximate length of wires

between adjacent nodes, where L is the length of a side of the core die. We assume a many core CMP with 1024 nodes, i.e., N is 32 and the core die size is 16 mm by 16 mm, i.e., L is 16 mm. Putting all these values into (2), W_{count}, W_{valid}, and W_{job} are 57.5ps.

It is assumed that the delay of all the gates and repeaters is 7.8ps, based on parameters from Toshibas 65 nm technology [27]. Assuming that their maximum logic depth is 4, then $G_{selector}$, G_{root}, G_{config}, and G_{switch} are 31.2ps. W_{config} is negligible because the switch signal generation block and the actual switch are placed adjacently. Putting everything into (1) gives a D_{total} is 11384.8ps. Thus, the maximum clock frequency required to accommodate single-cycle IsoNet operation in a CMP with 1024 cores is approximately 87.8MHz. This result is a testament to the light-weight nature of the proposed architecture: *even in an oversize chip with 1024 cores, single-cycle IsoNet operation is possible with a meager 87.8MHz*.

From equation (1), we can also infer that D_{total} is scalable in terms of the number of CPUs. After substituting equation (2) into (1), we can see that D_{total} is proportional to N. Note that the number of CPUs is N^2. Thus, D_{total} is related linearly to the number of CPUs, and, therefore, does not grow rapidly with the number of processing cores. This is a very important result, since it validates that *IsoNet is, indeed, a scalable solution*.

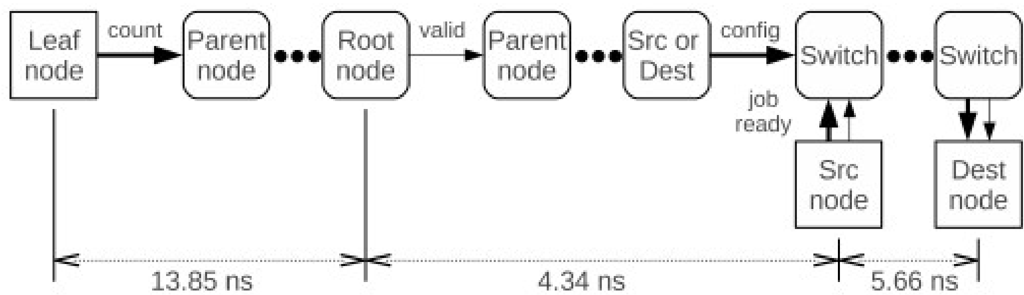


Figure 8: Longest combinational logic path within IsoNet. The delays shown in the figure are of the enhanced IsoNet implementation.

2.3.4 Scalability Enhancement: Enabling Multiple Job Transfers per Cycle

As will be demonstrated in Section 2.6, IsoNet works extremely well for *hundreds* of CMP cores. However, it starts to face scalability issues when the number of cores exceeds one thousand. The culprit is the limitation on the number of jobs transferred in each IsoNet cycle. In its original guise, IsoNet can transfer *only one job per IsoNet cycle*. As a result, the job transfer rate is upper-bounded because of this restriction. To address this issue and extend IsoNet's scalability to well over one thousand cores, this section presents techniques that enable *multiple job transfers per cycle*. We first introduce a local approach, whereby jobs are compared and transferred only between neighboring nodes (i.e., transfers occur only within a limited local vicinity). We then provide a further improvement, where the job transfer decision - which used to be taken solely by the root node - can now be taken by any intermediate node. Both techniques will be explained with the aid of the toy example in Figure 9.

One way to allow multiple job transfers per cycle is to distribute the decision-making process. All nodes can initiate job transfers, but only within their immediate neighborhood, i.e., their four adjacent neighbors in the mesh. Every node compares its job count to those of its four adjacent neighbors. If any of the four neighbors has a job count that is *smaller by more than one*, then the current node tosses one job to that neighbor. For example, node C in Figure 9 compares its job count with those of nodes A, B, D, and E. Since node A has a smaller - by more than one - job count than node C, a job is transferred from node C to node A. If a node's neighbors have *higher* job counts, the current node does nothing. In fact, this process is the reverse of job stealing: node A does *not* try to take a job from its neighbors. It simply waits until one of the neighbors with excess jobs (i.e., higher job count) initiates a job transfer. This is a very important rule: a node may only *toss* a job to any one of its four neighbors; it cannot steal a job from them. It is this attribute that ensures clash-free job transfers between overlapping neighborhoods. If every node follows this simple algorithm, local load-balancing can be effectively attained.

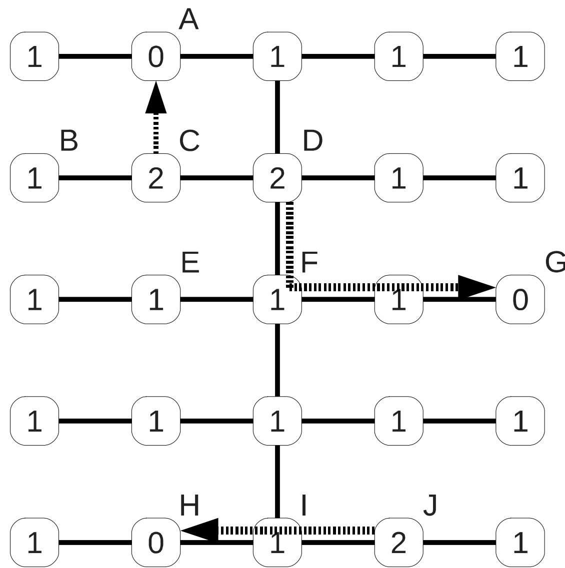


Figure 9: Illustration of multiple job transfers per IsoNet cycle.

Although this local approach allows multiple job transfers per cycle, it is not always successful in maintaining load balance. For example, node G in Figure 9 is idle, but its neighbors are unable to toss it a job, because they only have one job each. Even though node D has two jobs, it cannot transfer its extra job to node G, because job transfers occur only between adjacent nodes in the local approach. To enable a job transfer from node D to node G, IsoNet must be operating under the global approach described in Section 2.3.

Obviously, in order to improve on the limitations of the local balancing approach, we must borrow some of the benefits of the global approach and amalgamate them in a hybrid mechanism. In this fashion, IsoNet can transfer multiple jobs per cycle more efficiently. A fairly straight-forward way to achieve this is by allowing intermediate nodes on the Selector tree (see Figure 4) to make a source-destination transfer decision, before they propagate their information all the way to the root node. In other words, while collecting data from children nodes, any intermediate node can initiate a job transfer. For example, in Figure 9, intermediate node I can initiate a transfer between node J (source) and node H (destination), rather than sending the job counts upwards to the root of the tree. Similarly, node F can initiate a transfer from node D to node G in the same cycle. This approach

essentially *expands the scope* of the local approach beyond the four adjacent neighbors, towards a wider vicinity.

To allow intermediate nodes (i.e., non-root nodes) to make job source/destination decisions, the root node logic is placed in all IsoNet nodes. Remember that the root node logic is a comparator that compares the maximum and minimum job counts collected thus far to check if their difference is greater than one. If the difference is, indeed, greater than one, valid signals are sent to the appropriate children nodes. At the same time, to prevent these nodes from being selected again by the parent node (further up the tree), the intermediate node reports to its parent node that the maximum job count is zero and the minimum job count is the maximum possible number of entries in the job queue. This rule ensures the absence of overlapping transfer paths and guarantees that any one node can only be part of a single job transfer in any one cycle. Hence, multiple non-overlapping job transfers can occur in a single IsoNet cycle.

To implement the local balancing approach, a *Local Balancer* must be added to the IsoNet node logic, as shown at the bottom of Figure 10. The Local Balancer takes job counts from all four neighbors. A comparator in the Local Balancer compares the job counts of the four neighbors and selects the one with the smallest job count. If the smallest job count is smaller - by more than one - than that of its own job count, it notifies the Requester of its intention to toss a job to a neighbor.

The Requester issues a request to the affected neighbor, *unless* the current node is selected as a source or destination by the Selector tree (see Figure 4). In other words, the Selector tree has higher priority. If the neighbor replies with a grant, the Requester pops a job from the Dual-Clock Stack and transfers it to the neighbor.

The Arbiter in the Local Balancer makes a decision as to which request it should serve. If the current node is selected as a source or destination, no requests from neighbors can be served. Otherwise, a grant is given in a round-robin fashion.

As previously mentioned, in order to allow intermediate nodes to make job transfer

decisions, the root node logic must be included in all nodes.

2.4 Implementation

It is shown in Figure 8 that the longest path of combinational logic and its actual delay measured by implenetation. The IsoNet leaf nodes send their current job count to the parent nodes (the job counts come directly from registers). The Selector in the parent node compares the counts and chooses the selected count (largest/smallest job count). If there are $N \times N$ nodes in the CMP, the counts will go through at most N (= $N/2+N/2$) nodes until they arrive at the root node. In the case of the enhanced IsoNet design, it takes 13.85 ns to select the source and destination through the Selectors. The valid signals are sent to the source and destination nodes from the root node. The valid signals go through the root node logic in the opposite direction of the tree, i.e., from the root to the leaves. The maximum number of nodes the valid signals have to traverse is, again, N. This process takes 4.34 ns in the enhanced IsoNet. Each IsoNet Switch calculates its configuration signals, which are sent to the switch logic. Then, the job to be transferred and the ready signals are exchanged between the source and the destination. The job and ready signals should go through $2N$ switches at most. The job transfer takes at most 5.66 ns.

The Local Balancer of the enhanced IsoNet (Figure 10) is not shown in Figure 8, because it is not on the critical path. It can be executed concurrently with the logic shown in Figure 8. However, the critical path of the enhanced IsoNet is affected by the root node logic, which determines the source and destination nodes. To allow for multiple job transfers per cycle, the enhanced IsoNet architecture places the root node logic in every IsoNet node.

Both baseline and enhanced IsoNet architectures were fully implemented in HDL and subsequently passed through a detailed ASIC design flow using standard-cell libraries at the 45 nm technology node. The designs were synthesized, placed, and routed, before all measurements were taken. The die micrographs of the baseline IsoNet implementation are

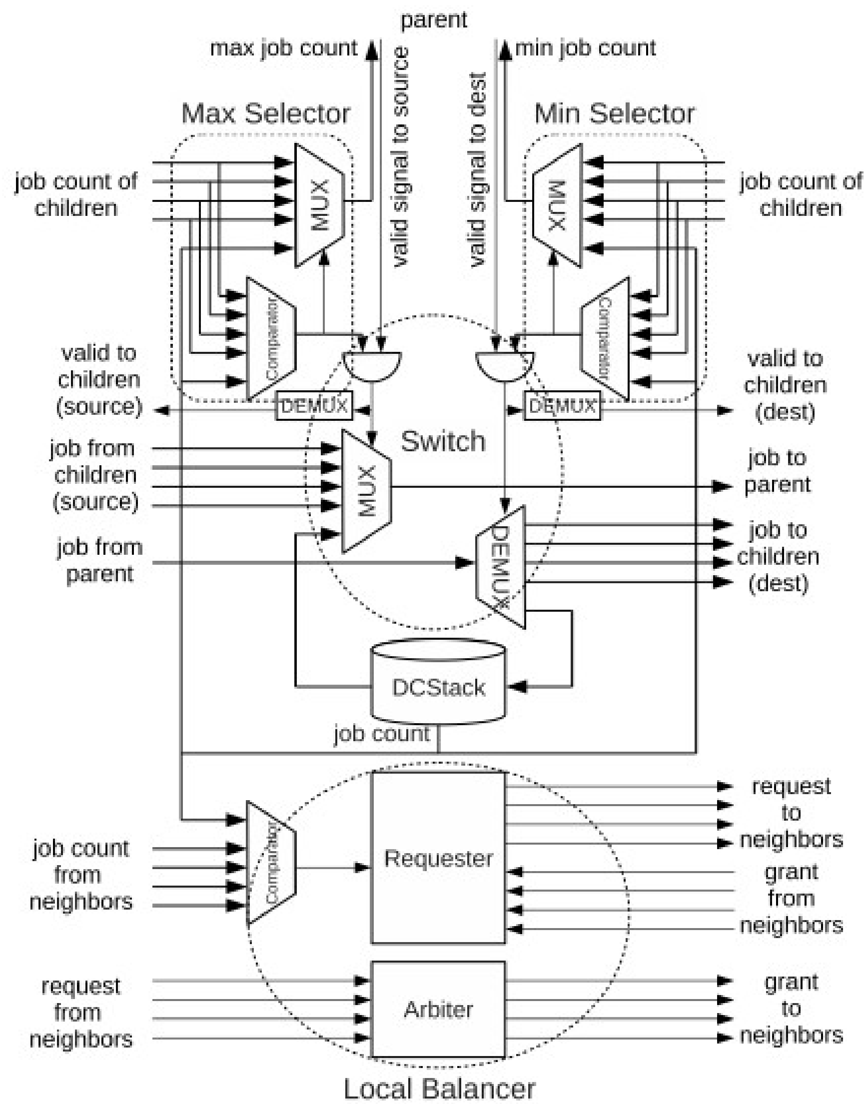


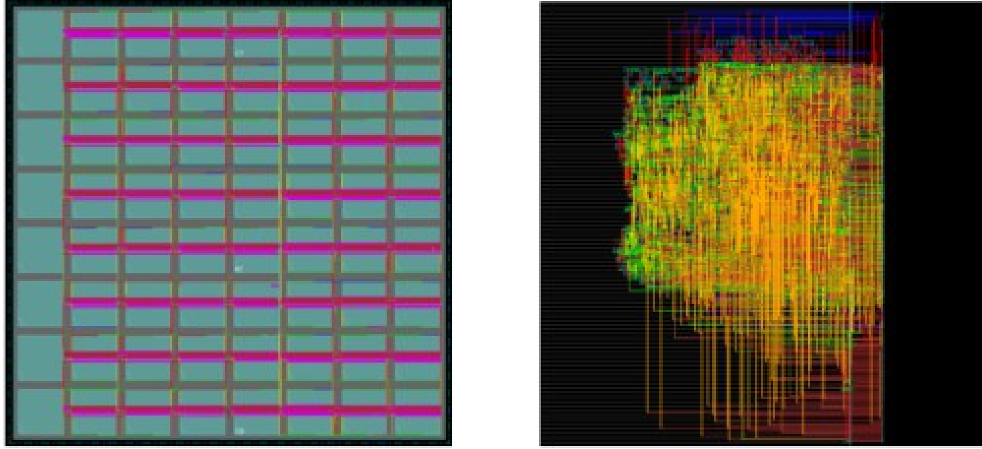
Figure 10: Overview of the enhanced IsoNet design that supports multiple job transfers per IsoNet cycle.

shown in Figure 11 (a 64-core CMP is assumed).

We assume that the size of each SRAM cell used in the Dual-Clock Stack is 0.346 mm^2 [28]. We size the Stack such that it contains 4 KB of SRAM memory. In addition, we assume that each core used in the design has a size of 500 μm × 500 μm, which is similar to the size of the ARM Cortex-A5 [29]. This is the type of lightweight core expected to be found in the many-core CMPs of the near future. Our implementation consists of 64 such processing cores arranged in an 8×8 grid, so we assume the die size to be 5 mm × 5 mm, which is a standard die cut. We first synthesize the design using Synopsys Design Compiler, and then place and route each individual core using Cadence Encounter. We use the same tool to perform timing optimization (buffer insertion) both at the single-core level, as well as the top level. Once the design is ready, we extract parasitic information into a standard parasitic exchange format (SPEF) file and use Synopsys PrimeTime to perform static timing analysis. In order to conduct power analysis, we perform gate-level Verilog simulations using Synopsys VCS, and extract the value change dump (VCD) file containing the switching information. This file is then fed into PrimeTime to obtain the power numbers.

Figure 11(a) shows the top-level placement and routing result, which consists of all the buffers added during timing optimization, as well as the routing of the buses between the cores. The placement and routing result of one node is shown in Figure 11(b). Using this very-large-scale integration (VLSI) physical implementation, the area, power consumption, and maximum clock frequency of IsoNet were measured. The results are summarized in Table 1.

The area of one *baseline* IsoNet node was found to measure 70 μm × 70 μm, with a gate count of 3,500 gates and 20,620 transistors. Since the *enhanced* IsoNet has more logic, its gate count and transistor count increases by 6.83% and 9.11%, respectively, as compared to the baseline. However, given the extremely lightweight nature of the baseline, the additional cost of the enhanced IsoNet barely affects the *overall* IsoNet overhead, as



(a) Top-level routing result of the 64-core system.

(b) The routing within a single IsoNet node.

Figure 11: Die micrographs of the resulting VLSI implementation of the baseline IsoNet architecture, assuming a 64-core CMP (8×8 mesh).

Table 1: Summary of measurements from VLSI implementation.

Category	Item	Baseline IsoNet	Enhanced IsoNet
Hardware cost per node	Gate count	3,500 gates	3,739 gates
	Transistor count	20,620	22,499
Wiring overhead per node	Intra-node	5.234×10^4 μm	8.747×10^4 μm
	Inter-node	6.048×10^4 μm	1.612×10^5 μm
Power consumption per node	Initialization phase	80.718 μW	84.547 μW
	Distribution phase	85.813 μW	93.500 μW
	Maintenance phase	80.984 μW	86.891 μW
Clock	Critical path delay	21.14 ns	23.85 ns
	Maximum clock freq.	47.30 MHz	41.93 MHz

compared to the entire CMP infrastructure. To put these numbers in perspective, we assess the enhanced IsoNet's feasibility using a recently revealed research chip by Intel as an example. Said chip incorporates 48 x86-compatible cores [1] on a single chip die. Each core tile is implemented with 48M transistors - in 45 nm technology - and operates at 1 GHz. The chip consumes 125 W at 50°C. To equip such a system with an IsoNet network, it would require 22,499 transistors per node (excluding the Dual-Clock Stack), which corresponds to merely 0.047 % of the overall transistor budget of each tile. Taking into account the SRAM requirement of the Dual-Clock Stack, 4K bytes correspond to roughly 196,608 transistors, assuming a typical 6-transistor bit cell. A count of 196,608 transistors per node translates to an overhead of 0.410 %. In *total*, the area overhead of the enhanced IsoNet would only be 0.456 %.

The power consumption was measured for three distinct operating phases. At the beginning of execution, an application pushes all its jobs into a single queue, in order to be processed (this is known as the initialization phase). Upon finishing initialization, IsoNet starts distributing jobs to other queues until all the job queues in the system have the same number of jobs (distribution phase). The distribution phase occurs simultaneously with job processing. As jobs are being processed, they are randomly popped from queues. To maintain load balance, IsoNet transfers jobs among nodes in an effort to keep the queues as evenly balanced as possible (maintenance phase). The power consumed in the aforementioned three phases was measured as 80.718 μW, 85.813 μW, and 80.984 μW per node, respectively. In the case of the enhanced IsoNet implementation, these numbers are 84.547 μW, 93.500 μW, and 86.891 μW, respectively. Again, to put this in perspective, we use the same 48-core Intel CMP example [1]: 48 nodes would translate to 4.058 mW, 4.488 mW, and 4.171 mW, for each of the three operating phases of the *enhanced* IsoNet. This corresponds to less than 0.0036% of the total chip power budget (remember, the entire chip consumes 125 W [1]). The above numbers exclude the power of the SRAM-based Dual-Clock Stack. In order to account for this power as well, we employed CACTI [30].

A 4 KB dual-ported SRAM-based memory implemented in 45 nm technology consumes 7.308 mW at a clock frequency of 1 GHz. Although the Dual-Clock Stack in IsoNet need not operate at this high CMP clock frequency, we still pessimistically assume this power consumption, which corresponds to only 0.281% of the CMP's power budget. In *total*, the power consumption overhead of the entire enhanced IsoNet (including the Dual-Clock Stack memory) would be 0.285%.

The wiring overhead of the two designs is reflected in the *total wire length* numbers shown in Table 1. The total wire length per node is 1.128×10^5 μm for the baseline IsoNet and 2.486×10^5 μm for the enhanced IsoNet. More specifically, the *intra-node* and *inter-node* wire lengths of the baseline IsoNet are 5.234×10^4 μm and 6.048×10^4 μm, respectively. Those of the enhanced IsoNet are 8.747×10^4 μm and 1.612×10^5 μm, respectively. Even though this metric may not yield great insight when viewed in isolation, it can be used in conjunction with the *power consumption*. The power consumption figures shown in Table 1 include the power of wires. Hence, we can infer from the power results that the overhead of wires would not be prohibitive.

Since, a 48-core CMP may not be so representative of the *many-core* microprocessors of the future (with hundreds of cores), we use another example to examine the feasibility of IsoNet. The NVIDIA GTX 570 GPU constitutes one of the closest incarnations to a real-life many-core chip, since it comprises 480 lightweight CUDA cores implemented with 3B transistors. It consumes approximately 219 W [2]. If the enhanced IsoNet were to be retrofitted to the NVIDIA GTX 570 GPU, the area overhead (including SRAM) would be 3.506%, while the power overhead would be 1.623%. Table 2 summarizes the overhead of both baseline and enhanced IsoNet, as compared to Intel's SCC [1] and NVIDIA's GTX 570 [2].

To estimate the maximum feasible operating clock frequency for a single-cycle implementation, the delay of the critical path was assessed. The critical path delay of the

baseline IsoNet was found to be 21.14 ns, which dictates that IsoNet can operate at a maximum clock frequency of 47.30 MHz. Similarly, the critical path delay of the enhanced IsoNet is 23.85 ns, which corresponds to a 41.93 MHz maximum clock frequency.

2.5 Supporting Fault-Tolerance

IsoNet can support fault-tolerance through two distinct mechanisms: the Transparent Mode and the Reconfiguration Mode. The responsibility of the Transparent Mode is to provide seamless load-balancing operation in the presence of faulty processing cores (i.e., faulty CPUs), while the Reconfiguration Mode restructures the IsoNet network fabric whenever there are faulty IsoNet nodes (i.e., faults within the load-balancing network itself). If any fault within IsoNet is detected, the entire load-balancing network switches to the Reconfiguration Mode, during which the topology of the Selector tree is reconfigured accordingly and/or root node duties are transferred to another node (if the problem is within the current root node). Subsequently, a fault-tolerant routing algorithm is used to bypass the faulty IsoNet node when a job is transferred through the IsoNet switches. A tree-based routing algorithm exploits the tree topology configured by the Reconfiguration Mode. Since this topology avoids any faulty node, the routing algorithm does not need to worry about faults.

2.5.1 Transparent Mode

IsoNet allows for the "hiding" of faulty processing elements from the load-balancing mechanism. Nodes that are hidden will simply never be chosen as source or destination for a

Table 2: Overhead of baseline and enhanced IsoNet over Intel's SCC [1] and NVIDIA's GTX 570 [2].

Chip	Item	Entire	Baseline IsoNet		Enhanced IsoNet	
SCC 48 cores	(a)	2.304 B	10.427 M	0.453%	10.517 M	0.456%
	(b)	125 W	0.3549 W	0.284%	0.3553 W	0.285%
GTX 570 480 cores	(a)	3B	104.27 M	3.476%	105.17 M	3.506%
	(b)	219 W	3.549 W	1.621%	3.553 W	1.623%

(a) Transistor count (including the Stack SRAM)
(b) Power consumption of the distribution phase (including the Stack SRAM)

job transfer. They are, therefore, discarded from the load-balancing algorithm, even though their corresponding IsoNet node still relays information that needs to pass through it in order to reach a parent or child node of the IsoNet tree. The IsoNet Selector of the afflicted node passes the minimum or maximum job counts down/up the tree, but it excludes itself from the comparator operation. Once a CPU is designated as faulty, then the corresponding IsoNet node enters Transparent Mode. CPU fault detection is beyond the scope of this thesis.

2.5.2 Reconfiguration Mode

Upon detection of a faulty IsoNet node, the load-balancing operation of that cycle is discarded and the entire IsoNet switches into the Reconfiguration Mode. The Reconfiguration Mode lasts one clock cycle and returns to normal mode the following cycle. The goal of the Reconfiguration Mode is twofold: (1) to reconfigure the Selector tree in such a way as to bypass the faulty nodes, and (2) to transfer the duties of the root node to another node (if the current root node fails).

Each IsoNet node is assumed to have four incoming control signals, one from each of the four cardinal directions (i.e., one from each neighboring node). Each signal indicates whether the corresponding neighbor is faulty or not. Similarly, each node has four outgoing control signals to notify its neighbors of its own health status. IsoNet node and link failures are indistinguishable; the outgoing control signals simply indicate a fault in the corresponding direction (i.e., the fault could be in either the adjacent inter-node link or in the adjacent IsoNet node). In addition, there is one more global control signal that is broadcast to all IsoNet nodes when there is some fault in the load-balancing network during that cycle. It is this signal that triggers the switch to Reconfiguration Mode. As previously stated, fault detection is an orthogonal problem and beyond the scope of this thesis; in this work, it is assumed that a fault detection mechanism is already present in the system.

The Reconfiguration Mode of operation is designed to use existing IsoNet hardware,

without incurring undue overhead. In fact, it is extremely efficient: it utilizes the comparators used to determine the source and destination nodes during normal load-balancing operation. Once IsoNet enters Reconfiguration Mode, it uses the same components for a different purpose. When reconfiguring the Selector tree (see Section 2.3.2 and Figure 4) - in case of an IsoNet node failure - the Reconfiguration Mode reuses the Min Selector to select a new minimum-distance path to the root node that bypasses the faulty node. Hence, in the Reconfiguration Mode, the inputs to the Min Selector are no longer job counts, but, instead, they become distances to the root node. The whole process unfolds as an outgoing, expanding wave starting from the root node and propagating to the leaf nodes. Assuming first that the root node is not the faulty node, a zero-distance is sent out from the root node to all of its adjacent nodes through the IsoNet mesh network. The Min Selector of each node receives distance values from its four neighboring nodes and outputs the minimum distance among them. Based on the output, the Selector module (i.e., both Min and Max Selectors) reconfigures the Selector tree topology by connecting itself to the adjacent node whose distance is the minimum. If any neighboring node has a fault, its distance is set to the maximum distance count so that it is never selected. Then, the Min Selector sends the minimum distance plus one to its neighboring nodes. The distances are eventually propagated outward from the root node to all the nodes and the topology is reconfigured so that every node can reach the root node by the shortest path while bypassing the faulty node.

The Reconfiguration Mode can also handle the situation of a faulty root node. This is done by having a pre-assigned root node candidate, which acts as a backup/reserve root node. The root node candidate is always a node adjacent to the actual root node, so that the candidate can directly detect a root node failure (through the health status control signals). When a fault in the root node is detected and the system switches to Reconfiguration Mode, the root node candidate undertakes root node duties and sends the aforementioned zero-distance to its adjacent nodes. It is assumed that the probability of both the root node and the candidate exhibiting failures in the same clock cycle simultaneously is near zero.

In order to handle faults within the root node, all IsoNet nodes have root-node logic (as described in Section 2.3.2), so that they can undertake root-node duties at any given time (i.e., every node has root-node functionality, but only one is designated as root at any instance).

Once a candidate becomes the new root node, a new candidate needs to be chosen from its neighbors. The candidate selection process is performed by the Max Selector of the new root node. The inputs to the Max Selector now become the maximum distances from the leaf node to the root node through that branch. The adjacent node with the maximum such distance value is selected as the new root node candidate. The maximum distance values are calculated in a process that unfolds in the opposite direction to the one described above for the Min Selectors, i.e., it can be viewed as a collapsing wave propagating inwards from the leaf nodes to the new root node: as soon as each node is reconfigured, the output of its Min Selector is provided as input to the Max Selector. Once the values propagate from the leaf nodes to the root node, the latter performs the final distance comparison and selects one of its neighboring nodes as the new root node candidate.

Figure 12 illustrates with an example the operation of the Reconfiguration Mode. When there is a faulty node, its connecting links (dashed lines) are missing, as shown in the figure. The top left boxes correspond to the Max and Min Selectors, and the numbers in the boxes are now the maximum distance from the leaf node to the root node through that branch and the minimum distance from that node to the root node. The node immediately to the right of the faulty node needs to find an alternative relaying node. It compares the minimum distances of nodes above and below, and chooses the node below since its minimum distance is smaller (2 versus 4). The root node candidate is selected among nodes adjacent to the root node. Both the node above the root node and the node below can be root node candidates because their maximum distance to the leaf node is the maximum among all adjacent nodes (with a value of 4). Either one is, therefore, selected arbitrarily.

The Reconfiguration Mode also enables IsoNet to support multiple applications running

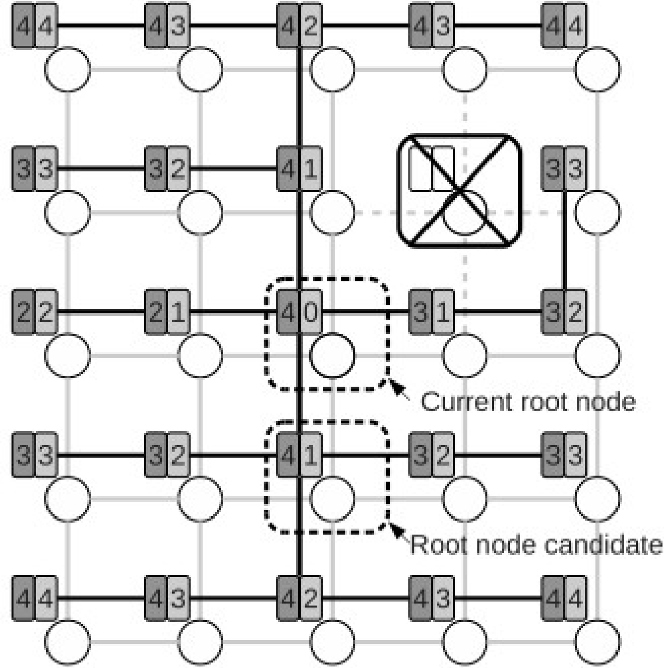


Figure 12: Reconfiguring the topology and selecting a new root node candidate.

at the same time, by reconfiguring itself so as to connect only a group of cores. Thus, multiple applications can be supported by grouping cores into individual logical trees (one tree for each core group), based on which applications are running. Note that one constraint of core grouping is that the cores within a group must be contiguous. The enhancement of core grouping and multiple concurrent applications is left for future work.

2.6 Evaluation

2.6.1 Simulation Setup

The evaluation framework employed in this work is double-faceted: for CMPs up to 64 cores, we use a full-system, execution-driven environment, while for many-core systems with 128 to 1024 cores, we use a trace-driven, cycle-accurate simulator.

More specifically, we simulate CMPs with processing core counts ranging from 4 to 64 using Wind River's Simics [31] full-system simulator. All processing cores are x86-based, and the CMPs run Fedora Linux v.12 (with Linux kernel v.2.6.33). The detailed simulation parameters are given in Table 3. The IsoNet clock frequency is assumed to be *32 times*

slower than the system clock frequency, so a single job transfer may be triggered every 32 system cycles.

In total, we compared four job queue management techniques: the software-based techniques of Job-Stealing (see Section 2.2) as a baseline, the hardware-based dynamic load-balancing technique of Carbon [11], and the two incarnations (baseline and enhanced) of the proposed IsoNet mechanism (both hardware-based). The hardware-based techniques of Carbon and IsoNet were implemented in device modeling language (DML), Simics's own hardware description language, and incorporated within the Simics CMP models. A comparison with ADM [15] was not deemed to be essential, because ADM has been shown to exhibit comparable performance with Carbon and only outperforms Carbon for a certain application, where a customized load-balancing scheme works much more efficiently.

One crippling limitation of existing full-system, execution-driven simulators is that they become prohibitively slow as the number of simulated processing cores increases beyond one hundred. Therefore, it is practically impossible to simulate such systems with Simics. Since *IsoNet is targeting many-core CMPs with tens, or hundreds, of cores*, we resorted to using our own trace-driven cycle accurate simulator to compare Carbon and IsoNet for core counts greater than 64. This simulator implements cycle-accurate models of both Carbon and IsoNet, but takes input from a workload generator that closely mimics real processors. The access pattern of the workload generator is synthesized based on a very detailed profile obtained by the Simics-based full-system simulator.

Table 3: Simulated system parameters

Parameter	Value
Processors	1 GHz x86 (4-64 Pentium 4)
Main memory	2GB SDRAM
OS	Linux Fedora 12 (Kernel 2.6.33)
L1 cache	16KB, 2-way, 64B line
L2 cache (shared)	512KB, 8-way, 128B line
L1 hit	3 cycles
L2 hit	12 cycles
Main memory access	218 cycles

Our simulations use benchmarks from the key emerging application domain of RMS [11]. The chosen applications are dominated by fine-grained parallelism and are deemed suitable for evaluating load-balancing schemes. We evaluate *two types* of fine-grained parallel applications: loop-level and task-level parallel benchmarks. Task-level parallel benchmarks exhibit a more complex behavior and are characterized by different job-conflict attributes than loop-level parallel benchmarks. This is because their *execution pattern* is not as regular as in loop-level benchmarks. A loop-level parallel benchmark usually spawns all the jobs at the beginning of execution and executes the same code repeatedly with different, but similarly-sized data. Thus, the execution time of each job does not vary by much. In contrast, the execution time of each job in a task-level parallel benchmark is often dependent on the size of the provided data set. Moreover, a job may generate more jobs, if necessary. Some applications spawn jobs gradually, while others do so immediately in bursts.

The loop-level parallel benchmarks used are Gauss-Seidel (GS), Dense Matrix-Matrix Multiplication (MMM), Scaled Vector Addition (SVA), Dense Matrix Vector Multiplication (MVM), and Sparse Matrix Vector Multiplication (SMVM). Since not enough information is given in [11] to implement *all* the task-level benchmarks used in said paper, we implemented only a subset: Binomial Tree (BT), Forward Solve (FS), and Backward Solve (BS) (as used in [11]). Furthermore, we added Quick Sort (QS) [32] and Octree Partitioning [19] as additional task-level parallel benchmarks. The detailed profiles of the various benchmarks are given in Table 4. We keep the job sizes similar to those used in [11] for fairness in the comparisons.

Loop-level parallel benchmarks have a main loop that consists of independent iterations. In our experiments, each iteration is mapped to a job. Because the independent iterations can be executed independently on different CPUs, they scale well as long as the number of jobs is more than the number of CPUs.

Two specific task-level parallel benchmarks - namely, FS and BS - exhibit similar behavior to loop-level parallel benchmarks. While, in general, loop-level parallel benchmarks

Table 4: Profile of RMS benchmarks

Benchmark	Job size (number of instructions)			Number of jobs
	Min	Max	Average	
GS	23535	78419	26506	65536
MMM	5127	80838	6473	65536
SVA	3335	71452	4103	65536
MVM	1927	77634	3261	65536
SMVM	1830	667637	2782	65536
FS	33	272278	110914	65536
BS	110574	340441	110912	65536
OP	5539	29871616	35783	66360
QS	88	51882	730	65484
BT	117	198632	559	65790

execute the same code on different data - which is compatible with the single-instruction multiple-data (SIMD) paradigm - FS and BS may execute different code depending on the job. However, since FS and BS also have a loop that consists of independent iterations, they exhibit similar scalability with the loop-level parallel benchmarks.

OP, QS, and BT spawn new jobs at run-time. Specifically, OP and QS begin with a single job and proceeds to gradually spawn new jobs. The number of new jobs spawned is dependent on the input data. However, since the newly created jobs are independent of each other, they can be executed independently. At the beginning of execution, the number of jobs is small, which limits scalability, but the number soon exceeds the number of CPUs. BT works in the reverse manner, as compared to OP and QS. It begins with a large number of jobs and the number gradually decreases. BT's jobs are also independent and the benchmark scales well, as long as the number of initial jobs is large enough.

We do not consider any applications that heavily rely on shared variables and synchronization primitives, because such applications cannot scale with the number of cores, even if a scalable load-balancing technique is provided. Experiments with applications that make frequent use of shared variables and synchronization primitives indicate that IsoNet also faces scalability issues as the number of cores exceeds 32, although it still exhibits much better scalability than software-based techniques. Since our proposed load-balancing

technique targets tens, or hundreds, of cores, we only focus our evaluation on *scalable* applications that do not heavily depend on shared variables and synchronization primitives, in order to isolate the true potential of efficient load-balancing.

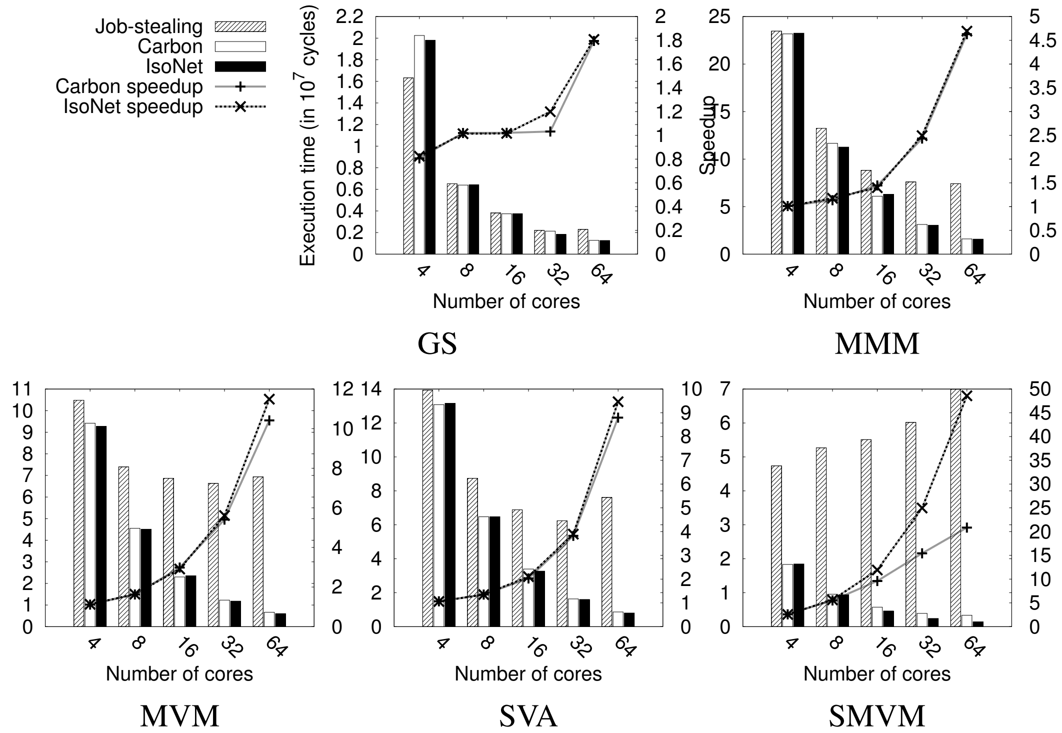
2.6.2 Performance Analysis

The slightly more complex *enhanced* implementation of IsoNet need not be used with relatively small-scale CMPs (i.e., below 64 cores), since the simpler, *baseline* version can perform equally well (it is not a bottleneck at such low core counts). However, for many-core CMPs with *hundreds of cores*, the *enhanced* IsoNet offers markedly improved scalability, as will be demonstrated here.

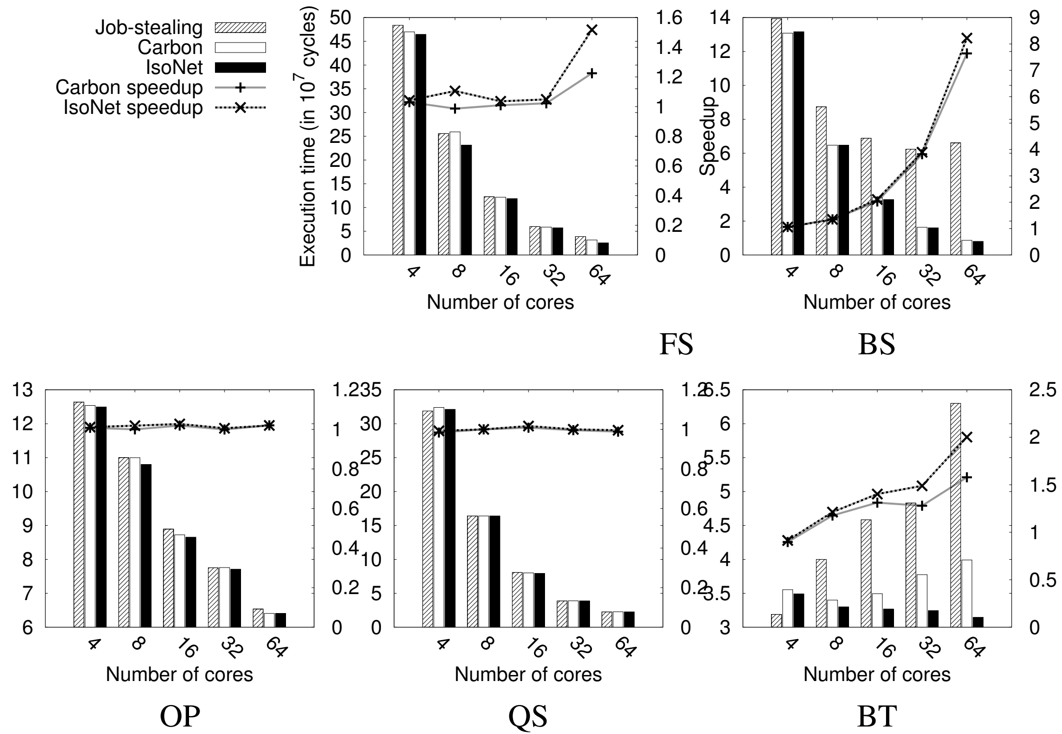
For loop-level parallel applications, the full-system simulations confirm that IsoNet outperforms Job-Stealing, exhibits comparable performance with Carbon up to 32 cores, and is slightly better than Carbon with 64 cores, as shown in Figure 13(a). The height of the bars indicates the total execution time and the numbers below the x-axis of each histogram refer to the number of cores.

Hardware-based job queues provide no significant benefit when the average job size is large - as in Gauss-Seidel (GS) - because the job queue is not accessed frequently. A somewhat odd behavior is observed when the number of cores is 4. Carbon and IsoNet exhibit poorer performance than software-based Job-Stealing. This is attributed to the overhead of accessing the hardware. In GS, this overhead is not compensated by the performance improvement of the hardware-based job queue. However, this is just an implementation artifact. If new instructions are added to the instruction set of the processor - as done in Carbon [11] - the overhead can be eliminated. Conversely, if we assume a simpler OS, like real-time operating systems, the overhead can also be significantly reduced.

The benefits of hardware-based job queues become evident when the job size is small (all other loop-level benchmark applications), where a significant speedup is observed, especially with a large number of cores. In addition, we can observe that performance improvement tends to increase as the job size becomes smaller. Note that the graphs are



(a) Loop-level parallel benchmarks



(b) Task-level parallel benchmarks

Figure 13: *Full-system* simulation results for (a) loop-level and (b) task-level parallel benchmarks.

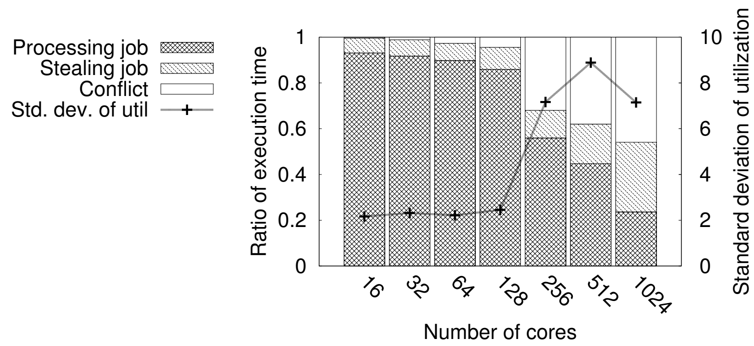
ordered by average job size. Carbon and IsoNet exhibit similar performance improvement in systems of up to 32 cores. However, in SVA and MVM, where the job size is even smaller, IsoNet starts outperforming Carbon with 64 cores.

In the case of very small job sizes - as in Sparse Matrix Vector Multiplication (SMVM) - IsoNet outperforms Carbon even when the number of cores is 16 and 32. This is because of two distinct features of IsoNet; one is conflict-free job queue management, and the other is IsoNet's proactive approach, whereby a job is delivered before a local queue drains.

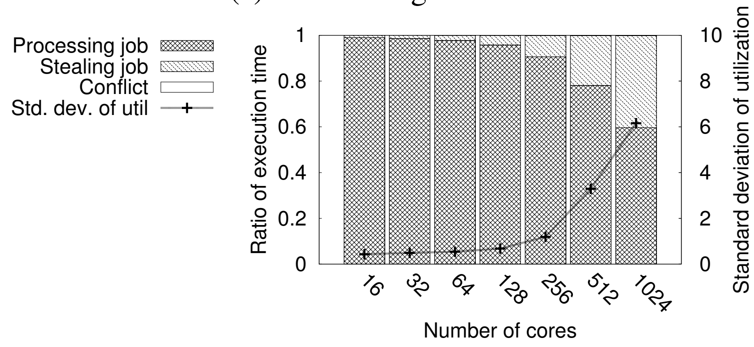
A deeper insight into the workings of all evaluated techniques is provided in Figure 14, which presents profiling results from one of the *loop-level* parallel benchmarks; namely, Gauss-Seidel (GS). The bottom part of each bar corresponds to the pure processing time spent on the jobs, the middle part represents the time spent on stealing a job from a job queue, and the top part corresponds to the waiting time due to conflicts. The *curves* in Figure 14 depict the *standard deviation of system utilization*.

It is interesting to note that in Figure 14(b), Carbon reduces execution time by *apparently* removing conflicts. However, conflicts are not actually removed, but, instead, *hidden* from software. A Carbon LTU prefetches a job while a thread is processing another job. Because this profile illustrates the software's perspective, conflicts are barely seen in the graph. However, the LTU cannot always succeed in prefetching a job on time, due to conflicts. In many cases, it takes too long for the LTU to prefetch a job, so the thread finishes processing a job *before* the LTU is done with prefetching. In theses cases, the thread needs to fetch itself a job from GTU. This wasted time in Carbon is accounted for in the "Stealing job" segments of Figure 14. IsoNet in fact reduces the time spent on stealing a job even further, by delivering a job *before* the local queue is empty. Since Carbon employs job-stealing, it steals a job only when the queue becomes empty. The processor, therefore, stalls while waiting for a job.

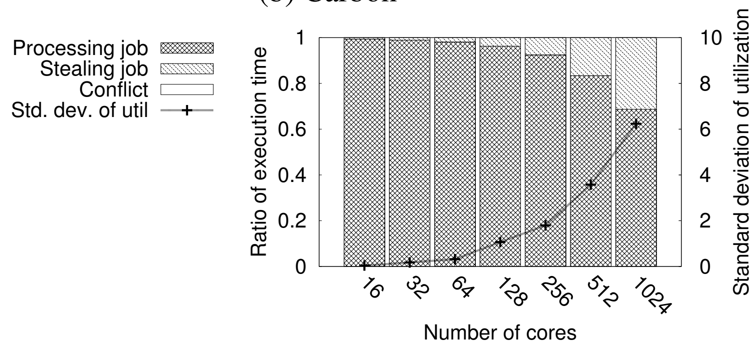
To isolate the benefits of the *enhanced* IsoNet implementation, we must refer to the *standard deviation* of the system utilization (i.e., the *curves* in Figure 14). Here, *utilization*



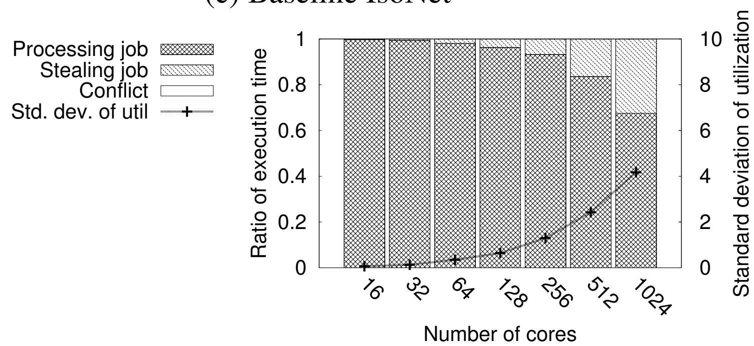
(a) Job-stealing



(b) Carbon



(c) Baseline IsoNet



(d) Enhanced IsoNet

Figure 14: Profile of the execution time of Gauss-Seidel (GS).

is defined as the ratio of time spent on processing jobs over the total execution time. Utilization is measured for each CMP core. The “Processing job” bars in Figure 14 correspond to the *average* utilization. Comparing Figures 14(c) and (d), one can see that the standard deviation of the utilization is substantially reduced with the enhanced IsoNet, while the average utilization is similar. This implies that cores are utilized more evenly. Since the total execution time is bounded by the longest execution time among the individual cores, reducing the standard deviation tends to reduce this critical, performance-determining longest execution time.

Turning now to task-level parallel benchmarks, it is shown in Figure 13(b) that all applications except Binomial Tree (BT) exhibit similar performance with all load-balancing techniques. This behavior is due to the fact that most of these benchmarks do *not* suffer from conflicts in CMPs with small numbers of cores. However, in applications with small job sizes that tend to suffer from conflicts - such as BT - IsoNet offers enormous benefits, as shown at the bottom of Figure 13(b). In spite of a similar job size as Quick Sort (QS), BT shows worse scalability with job-stealing and Carbon. This is because BT spawns jobs immediately after initialization, while QS spawns jobs gradually over time. Therefore, BT suffers from many more conflicts and, thus, benefits more from IsoNet.

In summary, for CMPs with up to 64 cores, the *baseline* IsoNet outperforms - on average - the job-stealing technique by 28.08% (up to 97.94%) and Carbon by 5.28% (up to 57.09%), respectively. For many-core CMPs with 128 to 1024 cores, the *enhanced* IsoNet outperforms - on average - the *baseline* IsoNet by 4.82% (up to 37.39%) and Carbon by 36.13% (up to 69.68%), respectively.

We conclude this subsection by investigating the sensitivity of system performance to the application’s job size. It is shown in Figure 15 how the results of a loop-level parallel benchmark (SVA) and a task-level parallel benchmark (OP) change according to the job size. We observe that, indeed, the loop-level parallel benchmarks are sensitive to the job

size. However, task-level parallel benchmarks appear (for the most part) to be almost insensitive to the job size. The behavior of SVA, as shown in Figure 15(a), indicates that - for loop-level parallel benchmarks - a hardware-based job queue offers the most benefit with smaller job sizes. In sharp contrast, one can hardly observe any difference as the job size of the task-level parallel benchmark OP is varied, as illustrated in Figure 15(b). This behavior is attributed to the fact that OP has an irregular computation kernel and is affected by other factors, such as the pattern of job spawning.

2.6.3 Beyond One Hundred Cores: Towards Many-Core CMPs

The chance of conflicts tends to increase as the number of cores increases. Even though IsoNet exhibits similar, or slightly better, performance than Carbon up to 64 cores, it is expected to significantly outperform Carbon with more than 64 cores. As previously mentioned, since it is prohibitively slow to simulate many-core systems with Simics, we performed experiments with our own cycle-accurate, trace-driven simulator for CMPs with 128 to 1024 cores.

Figure 16 shows the execution time of loop-level parallel and task-level parallel benchmarks up to 1024 cores. These graphs illustrate the system execution time normalized to the execution time under Carbon. Although Carbon and IsoNet exhibit similar performance up to 64 cores for GS and MMM, IsoNet starts to significantly outperform Carbon as the number of cores exceeds 128. As for the other benchmarks, it is clearly evident that IsoNet outperforms Carbon with increasing numbers of cores. This result demonstrates that IsoNet scales substantially better than Carbon in the many-core regime (beyond one hundred cores).

The graphs of Figure 16 also compare the scalability of the *baseline* IsoNet and the *enhanced* IsoNet. As expected, their performance is similar when the number of cores is less than one hundred. However, in systems with larger core counts, the enhanced IsoNet shows significantly better scalability. This is directly attributable to the presence of multiple job transfers per IsoNet cycle. In the case of Sparse Matrix Vector Multiplication (SMVM) at

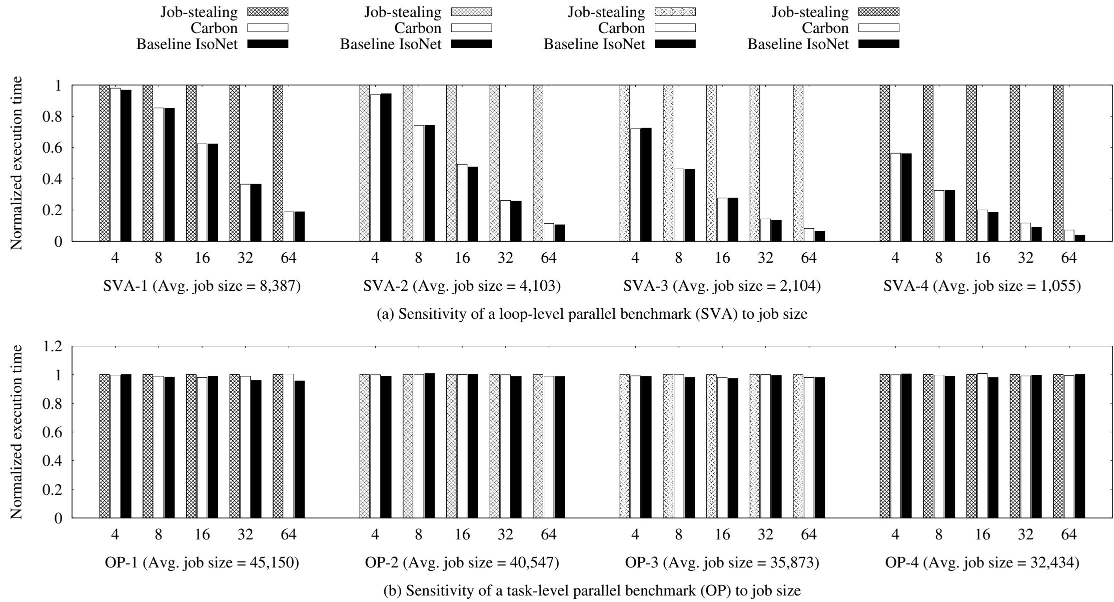


Figure 15: Sensitivity analysis of the two benchmark types (loop-level/task-level parallel) on the average job size. The numbers below the x axes of the graphs (4, 8, ..., 64) refer to the number of processing cores.

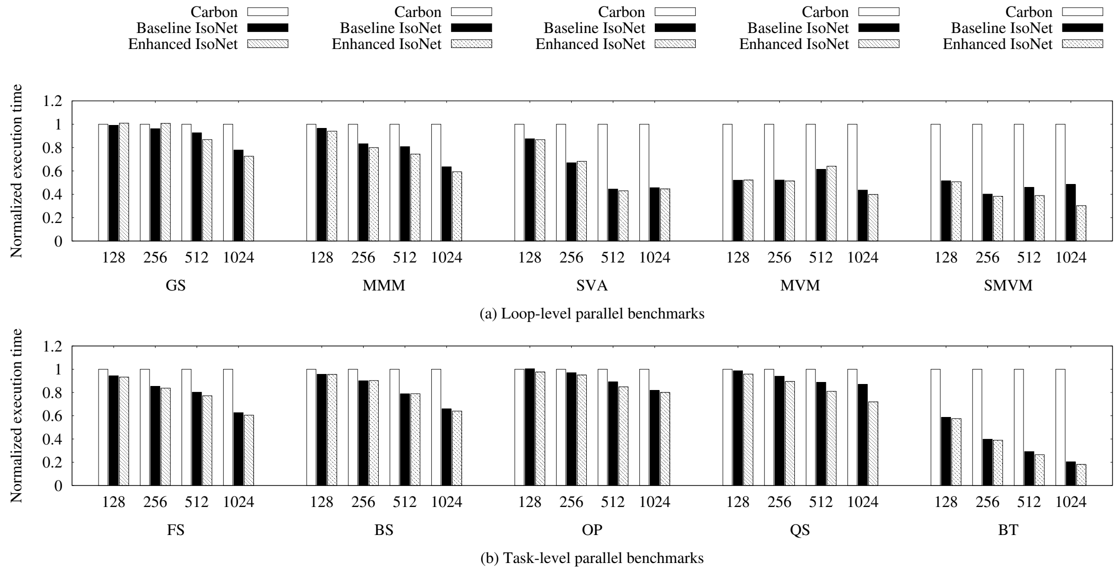


Figure 16: *Trace-driven* simulation results for (a) loop-level and (b) task-level parallel benchmarks. The numbers below the x axes of the graphs (128, 256, ..., 1024) refer to the number of processing cores.

1024 cores, the enhanced IsoNet design reduces execution time by around 70%, compared to Carbon, and by around 37%, compared to the baseline IsoNet.

However, the enhanced IsoNet sometimes exhibits *worse* performance than the baseline. Remember that when local job transfers occur, the involved nodes are not seen by the parent nodes further up the tree. Even though there might be a node with a *globally minimum* job count, it may not be able to receive a job, because the node is hidden by a local, non-optimal job transfer. On the contrary, in the baseline IsoNet, the node with the globally minimum number of jobs will always get a new job. If this node becomes idle (i.e., empty job queue), the total execution time may be affected. Although this happens rarely, it still causes the enhanced IsoNet to perform worse than the baseline design in some degenerate cases.

2.7 Conclusion

The last few years have witnessed the emergence of a powerful new thrust in the domain of microprocessor design: CMPs are steadily becoming a mainstay in modern computer architecture. As the number of processing cores in these multicore chips rapidly increases toward uncharted territories, it becomes imperative to ensure that the overwhelming abundance of hardware resources is *efficiently utilized*. Scalable and proficient parallelism in the many-core era requires effective distribution of the processing load across the entire chip.

This chapter addresses the vital need for efficient chip-wide load distribution by proposing IsoNet, a novel hardware-based, conflict-free, dynamic load distributor and balancer. Comprising a micro-network of lightweight load-balancing nodes (one for each CPU core), IsoNet dynamically re-distributes jobs between the processing elements at run-time, in order to maintain load balance throughout the CMP. The jobs themselves are maintained in distributed, hardware-based task queues, which are administered by the IsoNet engine.

In addition to a baseline version of IsoNet that can transfer a single pair of jobs in every clock cycle, an enhanced implementation is also presented, which can transfer *multiple jobs*

per cycle. This augmentation further improves the scalability of the proposed mechanism up to 1024 CPU cores. At the cost of a very modest area overhead over the baseline IsoNet, the enhanced IsoNet can further reduce execution time by up to 37% in many-core microprocessors with high core counts.

More importantly, IsoNet provides comprehensive fault-tolerance support through two special operation modes: the Transparent Mode handles CPU faults, while the Reconfiguration Mode deals with intra-IsoNet faults by dynamically reconfiguring the load distribution micro-network.

Detailed evaluation in a full-system simulation environment with real application workloads demonstrates that IsoNet significantly outperforms existing software-based load balancing techniques. Furthermore, IsoNet is shown to outperform Carbon [11], a hardware-based state-of-the-art task scheduler, by up to 70% (36% on average) in many-core CMPs with 128 to 1024 cores. More importantly, unlike the other techniques assessed in this work, IsoNet is shown to sustain *performance scalability* to more than *one thousand CPU cores*.

Finally, the IsoNet architecture is *fully implemented in 45 nm VLSI technology*. The design is synthesized, placed, and routed using a *commercial-grade ASIC design flow*. Subsequent timing, area, and power analysis indicates that IsoNet incurs near-negligible overhead. This attribute is partly due to the fact that IsoNet can efficiently function at a substantially lower operating frequency than the CPU.

CHAPTER 3

SHARDED ROUTER: A NOVEL ON-CHIP ROUTER ARCHITECTURE EMPLOYING BANDWIDTH SHARDING AND STEALING

Rapidly diminishing technology feature sizes have enabled massive transistor integration densities. Today's micro-processors comprise more than a billion on-chip transistors [2], and this explosive trend does not seem to be abating. The endless abundance of computational resources, along with diminishing returns from instruction-level parallelism (ILP), have led computer designers to explore the multicore archetype. This paradigm shift has signaled the genesis of CMP, which incorporates several processing cores onto a single die, and targets a different form of software parallelism; namely, thread-level parallelism (TLP). Current prevailing conditions indicate that the number of processing elements on a single chip will continue to rise dramatically in the foreseeable future. Inevitably, such growth puts undue strain on the on-chip interconnection backbone, which is now tasked with the mission-critical role of effectively sustaining the rising communication demands of the CMP.

NoC are widely viewed as the de facto communication medium of future CMPs [33], due to their inherent scalability attributes and their modular nature. Much like their macro-network brethren, packet-based *on-chip* networks scale very efficiently with network size. Technology downscaling also enables increases in the NoC's physical channel bit-width (i.e., the inter-router link/bus width). Inter-router links consist of a number of parallel wires, with each wire transferring a single bit of information. Wider buses (i.e., with more wires) facilitate massively parallel inter-router data transfers. Existing state-of-the-art NoC designs [34, 35, 36] already assume 128-bit links, while 256- and 512-bit channel widths have also been evaluated [37, 38]. In fact, 512-bit channel widths are presently being realized for the external memory channels of AMD's and NVIDIA's graphics chipsets [39].

Intel's Sandy Bridge micro-architecture (Core i7) employs 256-bit wide on-chip communication channels [38], while the Intel Single-Chip Cloud Computer [40] utilizes 144-bit wide channels. Tilera's NoC [41] employs 160-bit wide physical channels (in five independent 32-bit sub-networks).

However, from an architectural viewpoint, the wider physical channel size is not efficiently exploited, because the packet size (a packet is typically composed of a number of flow-control units, called "flits") is usually not a multiple of the channel width. This nuance is of utmost importance, and it has been largely ignored so far. In order to effectively utilize all bandwidth afforded by the parallel inter-router links, the flits must be able to make full use of the parallel wires comprising the physical channel.

In this thesis, we advocate *fine-grained slicing* of the physical channel, so that the channel bandwidth can be fully utilized. However, despite a boost in channel utilization, bandwidth slicing is also known to incur non-negligible latency overhead, due to increased serialization. In other words, a packet must now be decomposed into *more* flits, because the channel is *logically* narrower. This deficiency is precisely the fundamental driver of this work. The ultimate goal is to eliminate the increase in zero-load latency incurred by channel slicing, while, at the same time, maximizing the physical channel utilization. Toward this end, we hereby propose a novel NoC router micro-architecture that employs bandwidth "sharding" (a term borrowed from the database community), i.e., partitioning of the channel resources. The *Sharded Router* also benefits from a bandwidth-stealing technique, which allows flits to exploit idle bandwidth in the other slices. Thus, multiple flits can be transferred at the same time so as to maximize the channel utilization. The arsenal of mechanisms provided by the Sharded Router architecture can lower the zero-load latency to the same levels as in a conventional router, while throughput is substantially improved through the full exploitation of all available bandwidth resources.

The new design is thoroughly evaluated using both synthetic traffic patterns (to stress the network to its limits) and a comprehensive, full-system evaluation framework running

real multithreaded applications. Our results clearly demonstrate the efficacy of the Sharded Router; average network latency is reduced by up to 19% (13% on average), and the execution time of the various PARSEC benchmark applications [5] decreases by up to 43% (21% on average). Finally, hardware synthesis analysis using Synopsys Design Compiler verifies the modest area overhead (around 10%) of the Sharded Router over a conventional NoC router implementation.

The rest of this chapter is organized as follows: Section 3.1 presents a preliminary research on the flit size. Section 3.2 provides a more detailed motivation for the new router design, and presents a high-level conceptual description of the concept advocated in this work. Section 3.3 discusses related work in the area of channel/bandwidth slicing, while Section 3.4 introduces the Sharded Router architecture and its various techniques and mechanisms. Section 3.5 describes the employed evaluation framework and presents the simulation results and analysis. Finally, Section 3.1.7 concludes this chapter.

3.1 Preliminary Research on Optimal Flit Size

A "packet" is a meaningful unit of the upper-layer protocol, e.g., the cache-coherence protocol, while a "flit" is the smallest unit of flow control maintained by the NoC. A packet consists of a number of flits. If the packet size is larger than one flit, then the packet is split into multiple flits. In off-chip communication systems, the flit is split once more into *phits*. However, in the context of on-chip communication, the terms *flit* and *phit* typically have the same meaning and are of equal size. The flit size usually matches the physical channel width. If a network consists of multiple - physically separated - sub-networks, one sub-network uses only part of the channel and its flit size is matched to the size of the sub-channel.

Recent studies on NoC design usually assume a physical channel width of 128 bits [34, 35, 36], but 256 and 512 bits have also been evaluated [37, 38]. In some commercial

products, the channel width ranges from 144 bits to 256 bits (144 bits in Intel's Single-Chip Cloud Computer [40], 160 bits in Tilera's chips [41], and 256 bits in Intel's Sandy Bridge microprocessor [38]).

The wide range of flit sizes inspires us to address an obvious (yet unclear) question: what is the optimal flit size? In embedded systems, there has been extensive research in design space exploration that customizes/optimizes design parameters to given applications [42, 43, 44, 45, 46]. However, in *general-purpose* computing, it is very difficult to pinpoint specific numbers. A general rule is to maximize the performance at reasonable hardware cost. Regarding the flit size, in particular, it is still difficult to answer the aforementioned question, because the flit size is related to various aspects of a system, such as the physical implementation of global wires, the cost and performance of routers, and the workload characteristics. To the best of our knowledge, there has been no prior discussion on determining the appropriate flit size for general-purpose micro-processors.

This section aims to draw a meaningful conclusion by answering the following questions that cover all key aspects pertaining to flit size in NoCs:

- Can we afford wide flits as technology scales? (Section 3.1.2)
- Is the cost of wide-flit routers justifiable? (Section 3.5.3)
- How much do wide flits contribute to overall performance? (Section 3.1.4)
- Do memory-intensive workloads need wide flits? (Section 3.1.5)
- Do we need wider flits as the number of processing elements increases? (Section 3.1.6)

3.1.1 Preamble

Since we cannot cover all variety of architectures in this study, we have to assume a well-established and widely used NoC setting, which is the conventional wormhole router. Figure 17 shows the router architecture assumed in this chapter.

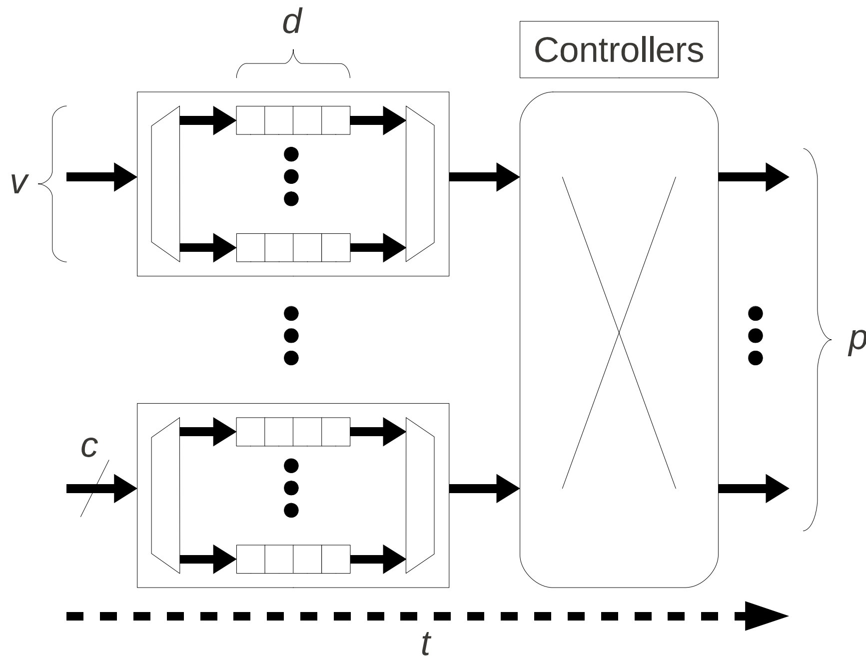


Figure 17: The assumed NoC router architecture and its salient parameters [*v*: number of virtual channels per port, *d*: buffer depth, *c*: physical channel width in bits, *p*: number of ports, *t*: number of pipeline stages].

The main duty of the router is to forward an incoming flit to one of several output ports. The output ports are connected to adjacent routers through physical links, whose width is *c* bits, and one output port is connected to a network interface controller. There are *p* input ports and *p* output ports in a router. Each input port has *v* buffers in parallel, which corresponds to *v* virtual channels. The depth of one buffer is *d* flits. It takes *t* cycles from flit arrival to departure (excluding contention).

Alternatively, there are routers that do not employ a switch, such as ring-based routers [47, 48, 49, 50, 51] and rotary routers [52, 53], but they are not considered here, since they are more specialized and not as widely used as the generic design assumed in this work.

To deliver a message, the router augments additional bits (overhead) to the packets, which specify the destination of the packet and include implementation-dependent control fields, e.g., packet type and virtual channel ID. As previously mentioned, if the packet is larger than the flit size, it is split into multiple flits. Figure 18 illustrates how a packet is

handled within a router. The header overhead is h bits and the payload size is l bits. The total number of bits in a packet is $h + l$ bits. If this size is larger than the flit size, f, the packet is split into N flits. If $h + l$ is not a multiple of f, the last flit is not fully utilized.

The flit size f may or may not be identical to the physical channel width c. Unless otherwise specified, we will assume a single physical network, where $f = c$.

To quantify an optimal/ideal flit width, a series of experiments are conducted. We employ Simics/GEMS [31, 6], a cycle-accurate full-system simulator, for the experiments. The parameters used for the experiments are shown in Table 5. The default values shown in the second column are used throughout, unless otherwise specified. The number of virtual channels (VCs) per port is three, because the MOESI-directory cache coherence protocol requires at least 3 VCs to avoid protocol-level deadlocks [54].

3.1.2 Global Wires

As technology advances, feature sizes shrink well into the nanometer regime. If we keep the same flit size, the area overhead of the global wires, which connect routers, decreases. However, if the power consumption is also taken into consideration, the result is quite the opposite.

Table 6 shows projected technology parameters. The values for 65 nm and 45 nm technologies are derived from ITRS 2009 [3], while those for 32 nm and 22 nm are from ITRS 2011 [4]. The chip size remains the same, regardless of technology scaling, but the number of transistors in a chip increases as technology advances. The wiring pitch

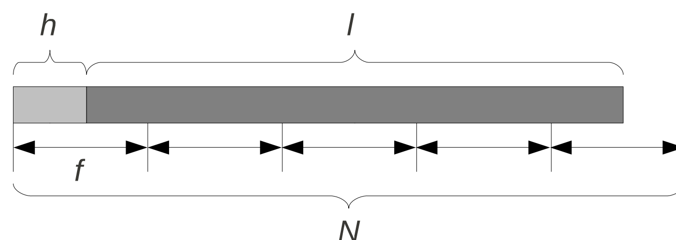


Figure 18: Splitting a packet into flits [*h*: header overhead, *l*: payload size, *f*: flit size, *N*: number of flits].

Table 5: System parameters

Parameter	Default value
Processor	x86 Pentium 4
Number of processors	64
Operating system	Linux Fedora
L1 cache size	32 KB
L1 cache number of ways	4
L1 cache line size	64 B
L2 cache (shared)	16 MB, 16-way, 128-B line
MSHR size	32 for I- and 32 for D-cache
Main memory	2 GB SDRAM
Cache coherence protocol	MOESI-directory
Benchmark	PARSEC
Topology	2D mesh
Number of virtual channels (v)	3
Buffer depth (d)	8 flits per virtual channel
Number of pipeline stages (t)	4
Number of ports (p)	5
Header overhead (h)	16 bits

of global wires also shrinks as the feature size shrinks. The "power index" parameter refers to the power consumption per GHz per area of wires [4]. It is the average of the power consumption of local, intermediate, and global wires [4]. The power index increases, because the coupling capacity increases as the feature size decreases. As for the total chip power, it decreases, because the supply voltage decreases.

We can compute the power consumption of the global wires (γ) by multiplying the power index (y) by the area of global wires (β). The area of global wires (β) is computed as the product of the total wire length (α), wiring pitch (x), and the number of wires (flit size). The wiring pitch of the global wires is given in the table. It is assumed that the same number of wires is used across the different technologies. To estimate the total wire length, we assume that the number of nodes in the network increases at the same rate as the number of transistors, since the chip size remains the same, regardless of the technology used. The normalized total wire length (α) is computed to be proportional to the square root of the number of transistors (w). The normalized wire area (β) is the normalized total wiring

Table 6: Projection of the power consumption of global wires. [3, 4]

Item	Unit	Value			
Technology	nm	65	45	32	22
Chip size	mm^2	260	260	260	260
Transistors (w)	MTRs	1106	2212	4424	8848
Global wiring pitch (x)	nm	290	205	140	100
Power index (y)	$W/GHz \cdot cm^2$	1.6	1.8	2.2	2.7
Total chip power (z)	W	198	146	158	143
Supply voltage	V	1.10	0.95	0.87	0.80
Normalized total wire length (α)[1]		1.00	1.41	2.00	2.83
Normalized wire area (β)[2]		1.00	0.99	0.97	0.97
Normalized wire power (γ)[3]		1.00	1.12	1.33	1.65
Normalized power portion (δ)[4]		1.00	1.53	1.66	2.28

[1] $\alpha \propto \sqrt{w}$
[2] $\beta \propto \alpha \times x \times$ number of bits (flit size)
[3] $\gamma \propto \beta \times y \times$ clock frequency
[4] $\delta \propto \gamma / z$

length (α) times the global wiring pitch (x) times the number of wires (flit size). Since the number of wires is assumed to be the same, the normalized wire area is proportional to the product of the total wire length and the global wiring pitch. Multiplying the normalized wire area (β) by the power index (y) gives us the power consumption of the wires per GHz. Assuming the clock frequency is the same, we can regard it as the normalized wire power (γ). The power portion (δ) is the normalized wire power (γ) over the total chip power (z).

As a conclusion, we can see that the power portion of the global wires (δ) increases as technology scales. This means that if we want to keep the flit size the same, we need to increase the power budget for the global wires. Therefore, **technology scaling does *not* allow for a direct widening of the flits**.

3.1.3 Cost of Router

It is well-known that the flit buffers and the crossbar switch are the two major components that determine the area and power consumption of a router [47]. Both the area cost and power consumption of the buffers increase linearly with the physical channel width [55].

Those of the crossbar increase quadratically with the physical channel width [55]. The following equations summarize the relationship between the cost of the buffer (3) and the crossbar switch (4). C_{buffer} refers to either the area cost, or the power consumption of the buffer, and C_{switch} refers to the crossbar switch.

$$C_{buffer} \propto c \times v \times d \tag{3}$$

$$C_{switch} \propto c^2 \times p^2 \tag{4}$$

From these equations we can expect that the cost of the router increases at a greater-than-linear rate. Figure 19 puts the area cost into perspective. A detailed breakdown of the area cost of a router is reported in [56]. The buffer accounts for 37.58%, the switch for 53.13%, and the allocators (arbiters) for 9.28% of the area of a 128-bit router [56]. If we double the flit size, the area of the router increases by 2.97 times. If the flit size becomes four times larger, the area of the router is 10.10 times larger than before.

The conclusion of this section is that the cost of a router increases sharply with the flit size, because the cost of the crossbar switch increases quadratically. If the performance improvement does not compensate for the increase in the cost, widening of the flit size is hard to justify.

3.1.4 Latency

If we ignore traffic congestion, the latency of a packet can be estimated by the following equation [57] (the congestion will be considered separately in Section 3.1.6). Parameter H denotes the hop count. Note that even if a packet spans multiple flits, the latency is not a multiple of the number of flits, since the router is pipelined.

$$L_{packet} = (t + 1) \times H + t_s \times (N - 1) \tag{5}$$

Parameter t_s refers to the delay of the switch, which is equal to one cycle in this analysis.

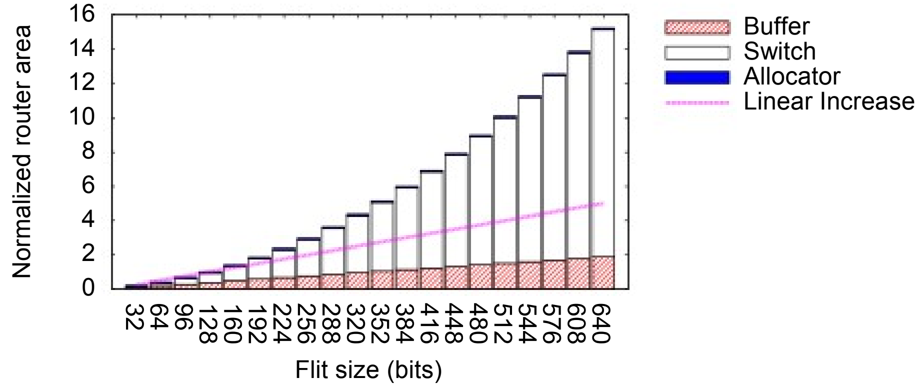


Figure 19: The increasing cost of a router with increasing flit size (width). The reference line indicates a linear increase, whereby the cost increases at the same rate as the flit size.

Parameter N can be re-written in terms of flit size f as follows.

$$N = \left\lceil \frac{h+l}{f} \right\rceil \tag{6}$$

From these equations, we can infer the following: for large networks, the term $(t+1) \times H$ would dominate the latency. Cost budget permitting, devoting resources to reduce the number of pipeline stages (t) by using pre-configuration [35, 58, 36], or prediction [59, 60, 61], would be more cost-effective than widening the flit size. We can also consider reducing the hop count (H) by employing alternative topologies [62].

The network traffic usually consists of packets of different sizes. Let us denote l_s to be the size of the shortest packet and l_l of the longest one. When we increase the flit size (f), it is expected that the performance will improve until f reaches $l_s + h$. The improvement slows down after $l_s + h$ until $l_l + h$, and there is no more improvement after $l_l + h$.

Let us put this into perspective by using the default values given in Table 5. The number of processors is assumed to be 64 (8 × 8 mesh). Then, the average hop count (H) is approximately 8 (= (8 + 8)/2). The number of pipeline stages (t) is 4 (typical) and the header overhead (h) is 16 bits. In the MOESI-directory protocol, there are two types of packets: control packets of length 64 bits (l_s) and data packets of length 576 bits (l_l).

Figure 30(a) shows the speedup results. Since the profiling information of the applications of the PARSEC benchmark suite [5] in Table 7 shows that the control packets account for 70% of network traffic, the "Mix" curve in Figure 30 is the weighted average of 70% the latency of control packets and 30% the latency of data packets. Again, a reference line is added that indicates linear increase.

The performance improvement of "Mix" from 32 bits to 64, 64 to 96, and 96 to 128 is 7.83%, 3.83%, and 1.46%, respectively. After 128 bits, the performance improvement is less than 1%. As expected, the performance improvement is relatively large until the flit size is less than around 80 bits ($l_s + h = 64 + 16$). However, the performance improvement is far less than the linear increase. If we reduce the network size, the latency is more sensitive than in a larger network. Figure 30(b) shows the results of a 4 × 4 network. The speedup is still far less than linear. For the weighted average ("Mix"), we can hardly see any performance improvement beyond 96 bits.

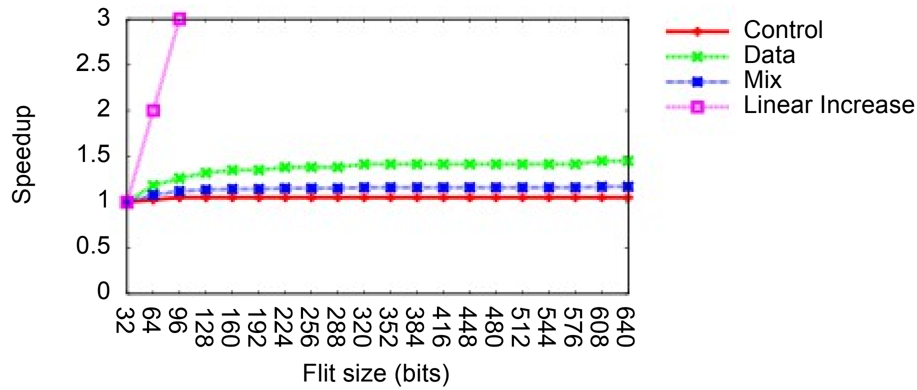
The conclusion of this section is that the performance improvement achieved by widening the flit size saturates beyond a certain point. The suggested rule of thumb is that the flit size should be matched to the shortest packet size ($f = l_s + h$).

3.1.5 Workload Characteristics

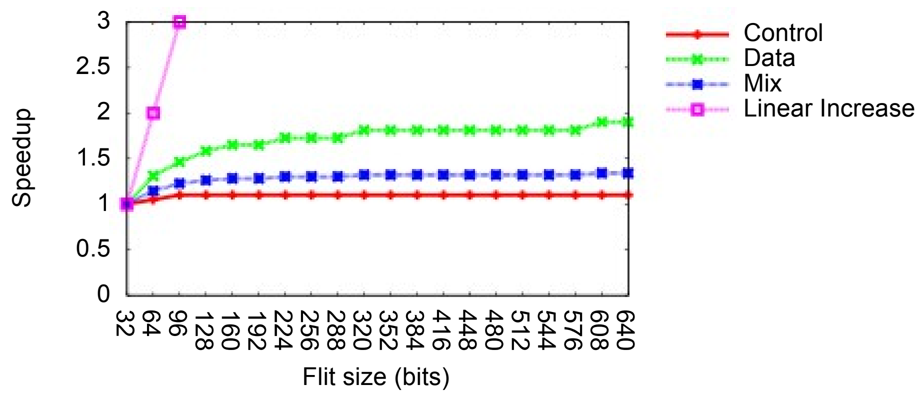
The analysis of the previous section does not consider traffic congestion. In other words, it is valid only at low injection rates. Indeed, it has been reported that the injection rates of real workloads cannot be high, because of the *self-throttling* effect [63].

The cache controller injects packets into the network when the cache is accessed. Upon a cache miss, the processor should be stalled (sooner or later). Even though the processor may need to issue more memory accesses, it cannot do so until pending cache lines are filled. Therefore, the injection rate cannot be high, even for memory-intensive workloads.

Our experimental results confirm this argument. Table 7 summarizes the characteristics of the applications of the PARSEC benchmark suite [5]. PARSEC benchmarks do not target a certain application domain; they represent a variety of diverse application domains [5].



(a) 8 × 8 network



(b) 4 × 4 network

Figure 20: Overall speedup with increasing flit size (width).

Table 7: Profile of the applications in the PARSEC benchmark suite [5].

Application	Cache misses /Kcycle/node	Injected packets /Kcycle/node	Percentage of control packets
blackscholes	0.41	2.21	73.46%
bodytrack	0.67	3.56	75.53%
ferret	0.26	1.43	71.60%
fluidanimate	0.24	1.35	71.13%
freqmine	0.28	1.48	72.27%
streamcluster	0.48	2.42	72.10%
swaptions	0.38	2.04	72.85%
vips	0.23	1.27	70.64%
x264	0.28	1.54	71.26%

From the perspective of the network, the injection rate is only affected by the cache misses per cycle. The second column of Table 7 shows the number of cache misses per 1,000 cycles per node. We can see that the cache miss rate is co-related with the injection rate. The third column shows the injection rate in terms of packets/1,000-cycles/node. The highest injection rate is only 3.56 packets/1,000-cycles/node (0.00356 packets/cycle/node), which is far less than the typical saturation point of a NoC. The last column shows the percentage of the control packets among all network traffic. The percentage does not vary much with the application.

The conclusion of this section is that we can keep the rule of thumb of Section 3.1.4, at least up to 64 cores, because the injection rate is very low in real workloads.

3.1.6 Throughput

If the number of cores increases to the tens or hundreds, the network can saturate even at low injection rates. To accommodate a large number of cores, a high-throughput network is necessary.

One way to increase the throughput of a network is to increase the flit size. Again, widening the flit size is not a cost-effective way to increase the throughput, because of *fragmentation*. The discrepancy between the packet size and the flit size limits utilization.

The utilization (U) of the physical channel is estimated by equation (7) below. Parameter l denotes the payload size, N is the number of flits computed by equation (6), and f is the flit size. Parameter U indicates how many bits are actually used for delivering the payload among all transmitted bits:

$$U = \frac{l}{N \times f} \tag{7}$$

To put this into perspective, we analyze the utilization of the control and data packets of the MOESI-directory cache coherence protocol.

We have drawn the rule of thumb of Section 3.1.4 from the fact that the latency improvement saturates when the flit size (f) exceeds the smallest packet size ($l_s + h$). When also considering the utilization, we arrive at the same conclusion. Figure 21 shows that the overall utilization ("Mix") gradually decreases (as the flit size increases) when the flit size (f) exceeds the smallest packet size ($l_s + h$). How fast it decreases depends on how much the smallest packet type is accounted for among all network traffic. Regardless, the concluding remark is that the overall utilization decreases with increasing flit size when f exceeds $l_s + h$, because of fragmentation.

The network throughput can be enhanced in different ways. Deepening the buffers, reducing the number of pipeline stages, and adding more virtual channels per port all contribute to the throughput. However, it is true that the performance improvement achieved through such techniques also saturates beyond a point.

If the budget allows, or if the only remaining way to improve throughput is to widen the physical channel width (c), we may consider widening the flit size (f). An alternative way to exploit the wide physical channel is to employ separate networks, with each one using only a part of the physical channel. Figure 22 compares the throughput of (1) one physical network with wide flits ($c = f$), and (2) two physically separated networks with narrow flits ($c = 2 \times f$). The baseline is one network with 80-bit flits. According to the profile of Table 7, 70% control packets - whose size is 64 bits - and 30% data packets - whose size is 576 bits - are injected. The traffic pattern is uniform random. The flit size is set to 80 bits by our rule of thumb ($l_s + h = 64 + 16 = 80$). When the flit size is doubled (one 160-bit network), we can see that the throughput improves. However, it is clearly evident that the physically separated networks (two 80-bit networks) offer better throughput than the monolithic network. In the physically separated networks, one network carries only control packets and the other network carries data packets. Even though the two networks are not evenly utilized, they offer better throughput than the monolithic network. Recall that the router cost of one 160-bit network is approximately three times larger than that of

a baseline 80-bit network, whereas the cost of two physically separated 80-bit networks is two times larger.

The conclusion of this section is that the widening of the flit size is not a cost-effective way to enhance throughput, because of fragmentation. If a wide physical channel is available, it is better to employ physically separated networks than to widen the flit size.

3.1.7 Conclusion

The answers to the key questions posed in the introduction of this section have been answered. Even though technology scales persistently, the number of global wires cannot grow as rapidly. The cost of a router increases sharply with increasing flit size, because the overhead of the crossbar switch increases quadratically. The performance improvement achieved by widening the flit size does not outweigh the increase in the cost. Increasing the flit size until the size of the smallest packet type is reached improves performance, but the performance improvement saturates as the flit size exceeds the smallest packet size. At least up to 64 cores, one need not increase the flit size to support high injection rates, because the injection rate of real applications is very low, due to the self-throttling effect. To enhance throughput, we may consider widening the physical channel width. However, employing physically separated networks to utilize this extra width is more cost-effective than widening the flit size of a monolithic network.

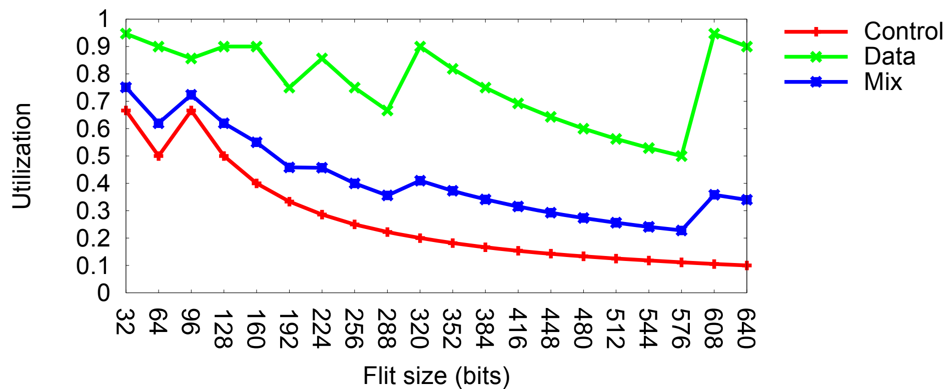


Figure 21: Physical channel utilization with increasing flit size (width).

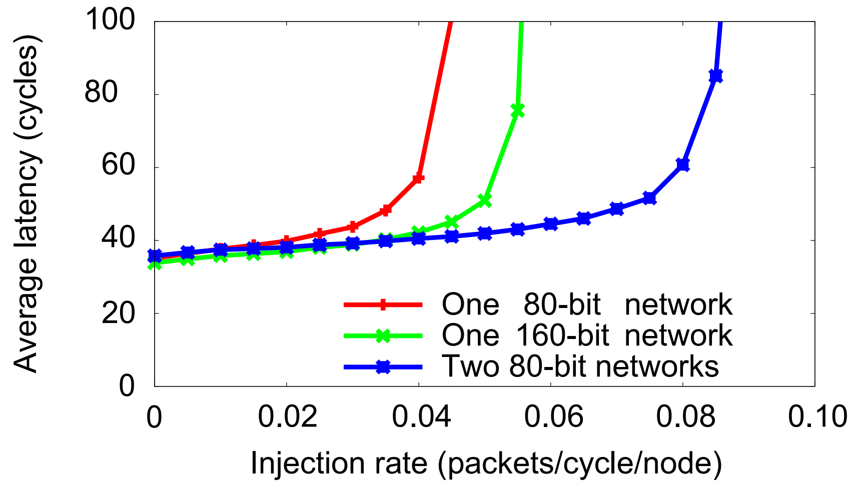


Figure 22: Throughput comparison of one physical network with wide flits vs. two physically separated networks with narrow flits.

The final conclusion of this section is that a wide monolithic network is not efficent and physically separated networks is a more cost-effective way to exploit the wide physical channel. The work in this chapter addresses the challenge of employing physically separated networks.

3.2 Motivation for Channel/Bandwidth Slicing and the Concept of Router Sharding

As previously mentioned, diminutive technology feature sizes enable tighter on-chip integration. In addition to more computational resources, this downscaling also enables wider parallel links, which can exploit more bit-level parallelism. Given the trends in the interconnection networks of existing multicore designs, channel widths of 256-to-512 bits are certainly not unreasonable.

The important question, however, is whether this massive bandwidth can be fully exploited. The answer is somewhat disheartening: given the current architectural practices, the increase in bandwidth capacity due to wider channels may remain largely untapped. The main cause for this inefficiency is the mismatch between the typical packet size in a CMP and the actual physical channel size. In general, the packet size is not a multiple of

the flit size.

Most modern CMPs rely on a cache coherence protocol to support the well-established and ubiquitous shared-memory programming model. The traffic in the NoC of the microprocessor is predominantly generated by this coherence protocol. In general, the NoC is responsible for the transfer of last-level cache (LLC) data (if the LLC is shared, which is a popular choice), in-bound and out-bound off-chip main memory traffic, and any cache coherence messages. Figure 23 abstractly depicts the sizes of two main packet types generated by the cache coherence protocol. The *size*, *field*, and the *number of different types* of packets are highly dependent on the implementation and the specifics of the coherence protocol. This figure, in particular, shows the MOESI-CMP-directory implementation of the GEMS simulator [6]. A more detailed message classification can be found in [64]. The payload of a control packet comprises the *control* and *physical address* fields. The *control* field may include the message class, or a dirty bit, depending on the implementation, but it typically consists of only a few bits. The dominant part of the control packet is the *address* field, whose size is 64 bits in 64-bit CPUs. Similarly, the dominant parts of the payload of a *data packet* are the *address* and *cache block* fields, whose sizes are 64 and 512 bits (64-B cache lines are typical in modern commercial CPUs), respectively.

For clarity and convenience, the payload size of control packets will henceforth be considered to be 64 bits, while the size of data packets will be assumed as 576 bits for the rest of this chapter. In other words, the *control* field of both packets (see Figure 23) will be ignored, because its size is small compared to the other fields, and it varies with the

Figure 23: Abstract visualization of the size of the two main packet types generated by the MOESI-CMP-directory implementation of the GEMS simulator [6]. In general, the types of messages traversing the NoC of a CMP are dependent on the employed cache coherence protocol.

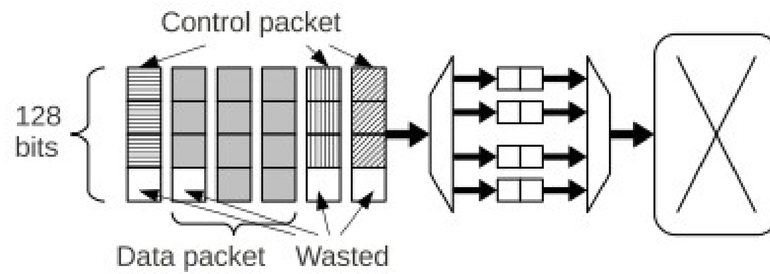
implementation and protocol.

Obviously, 64 and 576 are not multiples of 128, which is a characteristic NoC physical channel bit-width. Hence, if the two main packet types are carried in 128-bit flits, 64 bits are wasted per packet ($576 = 128 \times 4 + 64$). The work in [37] exploits the fact that the flit size (128 bits) is twice the size of the control packet (64 bits) and tries to accommodate two control packets within a single flit.

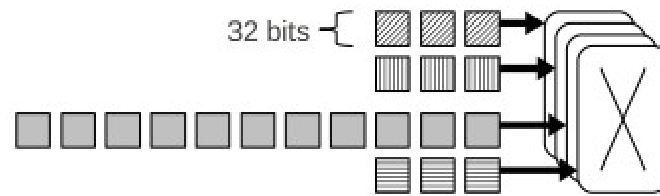
If the flit size increases while not being able to avoid wasted space, a significant portion of the available bandwidth will also be wasted. The sizes of the various fields within the packets are not likely to increase with the physical channel width. For example, the size of the address field is 64 bits, which is the address bit-width of the processor. The address bit-width is not expected to increase beyond 64 bits in the foreseeable future. The size of the cache block is the size of a cacheline. It is well-known that a larger cacheline does not always yield better performance, because spatial locality is naturally limited.

Figure 24 presents a conceptual illustration of the NoC physical channel utilization assuming different router micro-architectural approaches. More specifically, Figure 24(a) shows an example scenario when using a conventional NoC router. Of the 128 bits in each flit, 64 bits are used for the payload, and a part of the remaining 64 bits is used for the header. For example, if 32 bits are used in the header, then 32 bits are wasted per packet.

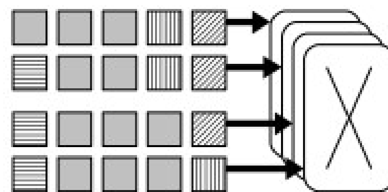
To fully utilize the available bandwidth of the physical channel, we advocate the notion of physical channel *slicing*. For example, we may split the 128-bit physical channel into 4 *independent* 32-bit physical channels, as shown in Figure 24(b). This means that there are 4 *independent routers* in each node, and each router utilizes a 32-bit physical channel. This router architecture is referred to as the *Slice Router* [65]. Since 64 and 576 are multiples of 32 bits, the Slice Router fully utilizes the physical channel. However, since the channel width is reduced, the size of each flit is reduced accordingly. For instance, when the channel width is 128 bits, one 64-bit control packet can be sent within a single flit. However, when the channel width is 32 bits, the 64-bit control packet must be split into 2 flits, and one



(a) Conventional router approach.



(b) The proposed approach of channel slicing.



(c) A naive approach that mixes different packet types.

Figure 24: Conceptual view of the NoC physical channel utilization assuming various router micro-architectural approaches.

additional flit must be added to serve as the header flit. Thus, 3 flits are required to send a 64-bit control packet over a 32-bit physical channel. Similarly, 19 flits are required to transmit a 576-bit data packet when the channel width is 32 bits, whereas it only takes 5 flits when the channel width is 128 bits. Therefore, the packet latency increases when the channel is sliced into smaller chunks. To overcome this potentially show-stopping longer latency, this thesis proposes the use of *fine-grained bandwidth partitioning* (aka "sharding") and a brace of micro-architectural techniques: *bandwidth-* and *buffer-stealing*. These mechanisms work in unison within the *Sharded Router* in order to fully utilize the available bandwidth without adversely affecting packet latency. The details of the proposed new architecture will be presented in Section 3.4.

Figure 24(c) illustrates a naive approach that can also fully utilize the physical channel without increasing the packet latency. In this approach, a packet is broken into 32-bit pieces and transmitted over any available channel. Since the physically separated routers work independently, flits may be ejected out-of-order. Therefore, there should be additional buffers to collect all the pieces for correct re-assembly. This requirement incurs significant overhead and may incur additional delay. Moreover, since buffer space is not infinite, the collection buffer may fill up and cause deadlocks. To avoid deadlocks, a complicated flow control must also be implemented. On the contrary, the bandwidth-stealing technique introduced in this paper does not suffer from these problems, while it also reduces the packet latency to levels similar to the ones observed in conventional routers.

It is worth noting at this point the significance of avoiding protocol-level deadlocks within the NoC of a CMP. Protocol-level deadlocks occur when a node's buffer becomes full, while the node is waiting for a certain type of message that is not presently in its buffer. The most popular method to avoid such protocol-level deadlocks is to employ *virtual channels* within the NoC, in order to separate the different *message classes* (i.e., types). Hence, any proposed router architecture intended to be used in a large-scale CMP must necessarily employ a mechanism to isolate the various message classes and ensure the avoidance of

protocol-level deadlocks.

3.3 Related Work

There is a vast body of literature devoted to NoC router architectural techniques and augmentations. In this section, the focus will be on mechanisms that are related to channel/bandwidth slicing and network segregation/decomposition. This domain is deemed the most relevant to our work on the Sharded Router.

The authors of [66, 67] explored various configurations of physically separated networks. These separated networks work independently with no interactions between themselves. When separated networks are employed, one of the networks is selected whenever a packet is to be injected. The selection process considers the current load balance among the networks, because any one of them may become a bottleneck (since the networks work completely independently). Kumar et al. [68] proposed a *virtual concentration* scheme that allows a packet to be transferred to any network, regardless of the network used during injection. However, the virtual concentration mechanism cannot reduce the longer packet latency incurred by the narrower channels. Instead, the bandwidth-stealing technique employed by the proposed Sharded Router reduces the latency by exploiting idle bandwidth in the other networks.

Spatial division multiplexing [69, 70, 71] techniques divide the physical channel into sub-channels, and manage these sub-channels as circuit-switched networks in order to provide throughput guarantees. The widths of the assumed sub-channels are usually very narrow, i.e., just a few bits. Therefore, the latency of a single packet is substantially increased, because the packet is sent bit-by-bit in a serial manner. The main purpose of spatial division multiplexing is to guarantee the throughput performance, while sacrificing the latency performance. Link division multiplexing [72] and lane division multiplexing [73] work in a similar vein. These techniques are only suitable for specific applications that value throughput much more than latency.

Channel slicing is employed by asynchronous NoC routers [74]. In order to enable asynchronous hand-shaking among the routers, a completion-detection circuit is required, which lies on the design's critical path. The latency overhead of the detection circuit increases with the channel width. Thus, the overhead is reduced by splitting the physical channel.

The work in [65] demonstrated the benefits of physical channel slicing with regard to fault tolerance. Sliced router designs are shown to be more resilient to faults. The fault-tolerant attributes of [65] are easily applicable to the Sharded Router as well, due to the same underlying concept of slicing.

There have been approaches that adopt a separate narrow channel to support the wider main network. In these approaches, the separate physical channel is used only for a dedicated function. For example, the designs in [35, 58, 36] utilize the extra channel as part of a pre-configuration network, while the architecture in [36] employs another network solely for negative acknowledgements.

Finally, researchers have tried to enhance network utilization by handling control packets differently [37, 67, 64]. As previously mentioned, the authors of [37] fuse two short control packets into one wide flit, so that they can be transmitted in one cycle. Balfour et al. [67] demonstrate that a physically separated network for control packets enhances both the area-delay and area-power products. The authors of [64] improve power efficiency by carrying the control and data packets on different interconnect wires.

3.4 The Sharded Router Architecture - A Sliced NoC Design Employing Bandwidth- and Buffer-Stealing

3.4.1 The Baseline NoC Router

Before proceeding with the details of the Sharded Router, we briefly describe the basic attributes of a conventional baseline NoC router design. This description will aid the reader's comprehension of the Sharded design, since the latter will be juxtaposed to the generic NoC archetype.

The baseline router shown in Figure 25 has 5 input/output ports. One port is for the network interface controller (NIC), which is the gateway to the local processing element. Packets are injected into (and ejected from) this port. The remaining four ports correspond to each of the four cardinal directions in a 2D mesh. Each port has 4 VCs and each VC has a 4-flit deep FIFO buffer. The physical channel width (i.e., *phit* size, which is usually equal to the *flit* size) is 128 bits. The generic design is assumed to be a canonical 3-stage router, as found in the literature [34, 35, 58, 47]. The three pipeline stages correspond to (1) buffer write and route computation, (2) virtual channel and (speculative) switch allocation/arbitration, and (3) switch/crossbar traversal.

The grey box marked with the number ‘1’ in Figure 25 is the main crossbar switch that interconnects the input and output ports. The DEMUX ‘2’ and MUX ‘3’ are used to select a VC within a port. The DEMUX ‘4’ and MUX ‘5’ are used to select a specific flit slot within each VC buffer. FIFO order in the VCs is maintained through the pointer logic controlling ‘4’ and ‘5.’

We further assume that the router is used to handle the shared-LLC traffic of a CMP, while fully conforming to the employed cache coherence protocol. More specifically, the MOESI-CMP-directory implementation of GEMS [6] is used for cache coherence. As mentioned in Section 3.2, the packets can be classified into *control* and *data* packets, whose sizes are 64 bits and 576 bits, respectively. The specific cache coherence protocol requires *at least* 3 virtual channels to avoid protocol-level deadlocks. However, we assume the presence of 4 virtual channels throughout this paper, which is more intuitive and practical from a hardware implementation perspective (power-of-2). When a packet is injected into the network, the appropriate VC is allocated according to the packet’s message class. As the number of available VCs increase, more VCs are dedicated to each message class (as defined by the cache coherence protocol). All VCs within one message class are treated identically, i.e., a packet belonging to one particular message class may freely go into any one of the VCs dedicated to that class. These VCs are typically allocated in a round-robin

fashion.

It should be noted that the parameters and attributes described here are chosen without loss of generality. In other words, the Sharded Router architecture to be described in the following subsection can be modified and applied to any cache coherence protocol and can be compared to any generic NoC implementation. The parameters have been made specific in order to enhance understanding.

3.4.2 The Micro-architecture of the Sharded Router

Figure 26 shows a high-level conceptual block diagram of the proposed Sharded Router's micro-architecture. The notion of "sharding" refers to the fact that the conventional design is partitioned (sliced) into 4 *independent* sub-networks, called *slices*. Rather than a wide 128-bit physical channel, each of the four sliced networks has a narrow 32-bit channel (the total aggregate width between the four slices is still 128 bits). Each slice may have only one 16-flit deep FIFO buffer (i.e., each slice corresponds to one VC of the conventional router design), or - in the general case - each slice may have multiple VCs. Note that a *flit* in the Sharded Router is only 32 bits in size, rather than 128, and it goes through the same pipeline stages as in the conventional NoC router.

The Sharded Router architecture employs *four* main crossbar switches, marked as '1' in Figure 26; one crossbar is used for each of the four slices. The main crossbar switches are used to direct the flits to their output ports. The bit-width of each switch is 32 bits (i.e., much narrower than the 128-bit crossbar of the baseline router). The DEMUXes '2' and MUXes '3' in Figure 26 are used to select a specific VC within a port *of a single slice*. The figure depicts an implementation with 2 VCs per slice (i.e., 8 VCs in total), but if there is only one VC per slice, then components '2' and '3' are not necessary. There are 4 DEMUXes and 4 MUXes, because up to 4 flits may be selected in the same clock cycle when using the *bandwidth-stealing* technique, which will be described in the following subsection. For the same reason, four DEMUXes '4' and four MUXes '5' are necessary to select individual flits within a VC FIFO buffer. The DEMUXes '6' and MUXes '7' are

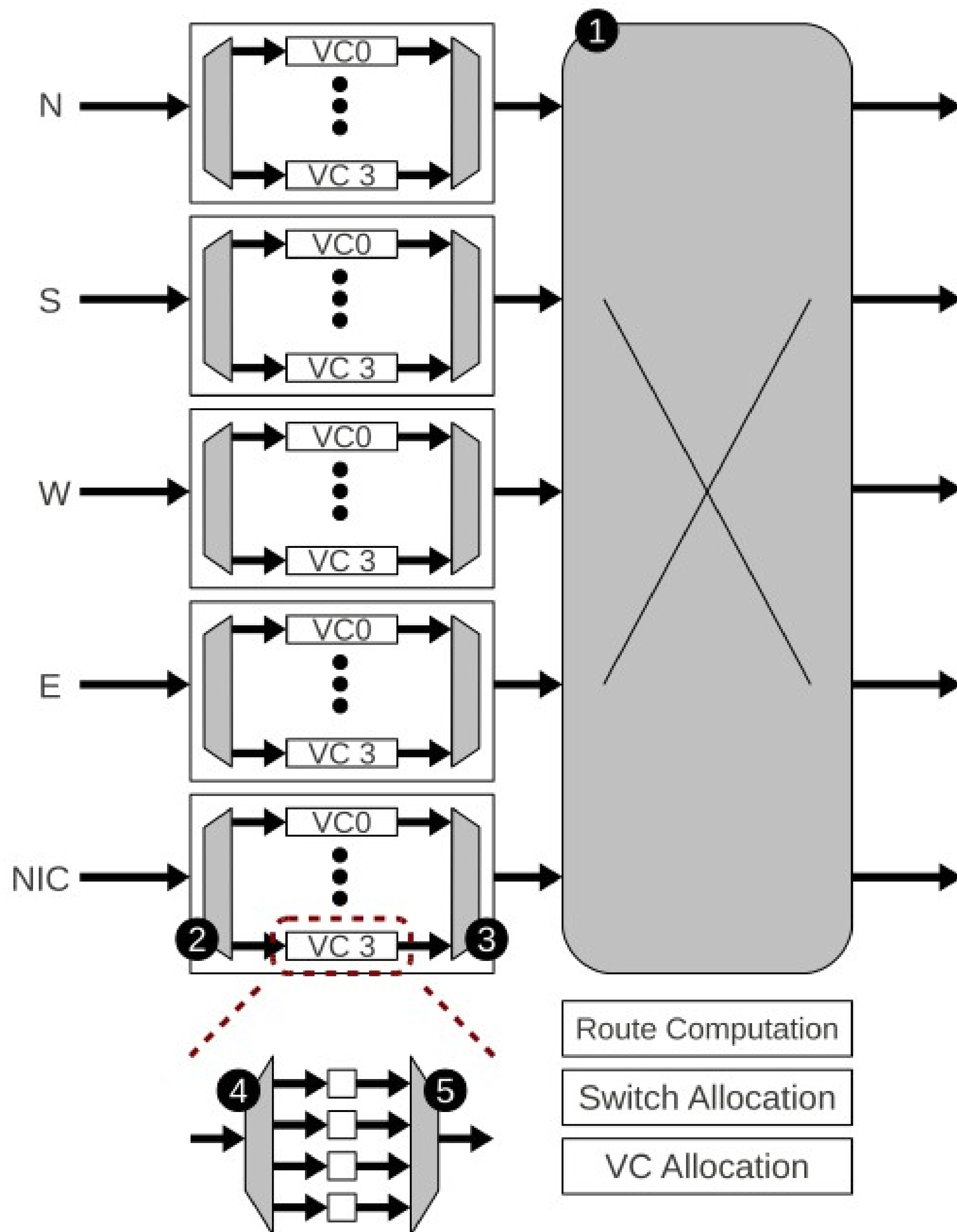


Figure 25: A conceptual overview of the baseline router's micro-architecture. This is a typical input-buffered NoC router design, where the Virtual Channel (VC) buffers employ a *parallel* (rather than serial) FIFO implementation. The FIFO order is maintained by the pointer logic controlling the input DEMUX and output MUX ('4' and '5' in diagram above).

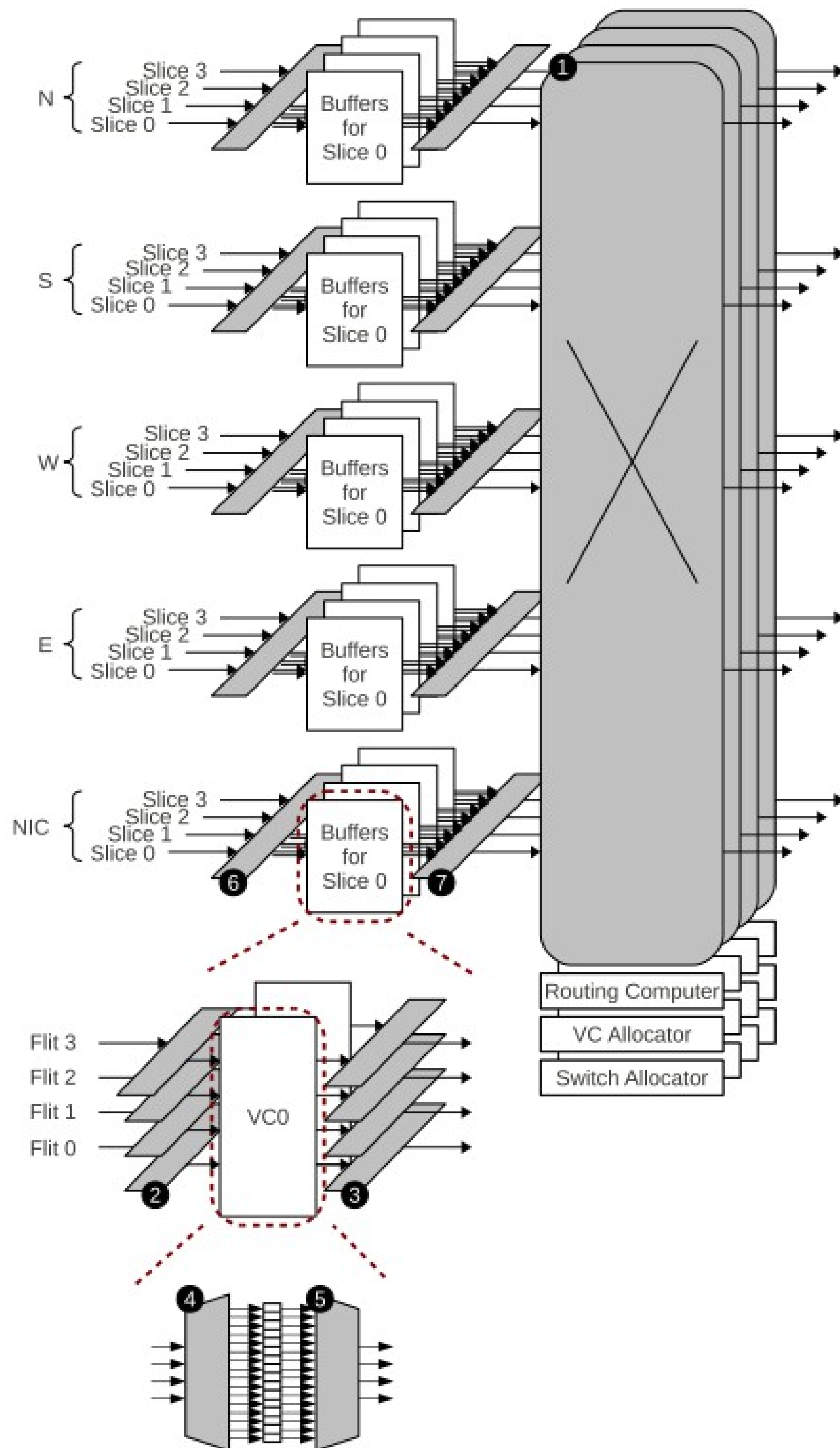


Figure 26: A conceptual overview of the Sharded Router's micro-architecture. The proposed design has 4 physically separated networks (called "slices") and each network has a physical channel width of 32 bits. In this case, each slice has two Virtual Channel (VC) FIFO buffers.

used to select flits between slices. They enable flits to temporarily get transferred to another slice when performing *bandwidth* and *buffer stealing*.

Even though the Sharded Router appears - at first sight - to be significantly more complicated than the conventional design, its area overhead is, in fact, a modest 10.55% over the baseline, as will be described in Section 3.5.3. The reason why the overhead is contained to within reasonable levels is because the underlying architecture relies heavily on the *partitioning of existing resources*. The aggregate amount of hardware remains largely the same. The baseline router's constituent modules are simply "sharded" into 4 narrower, leaner independent slices.

The proposed router has four slices, because of our original assumption that the cache coherence protocol requires the network to have four virtual networks to avoid protocol-level deadlocks (see Section 3.4.1). Similarly, the 128-bit physical channel width of the baseline router is divided by 4, and 32 bits of channel width are assigned to each slice. The slices are assigned according to the packet types supported by the cache coherence protocol, in the same way VCs are assigned in the baseline router. For example, request packets can be assigned to Slice 0, while response packets are assigned to Slice 1. The NIC injects packets to one of the slices, according to the packet type. Hence, each *slice* of the Sharded Router undertakes the duties of one *virtual channel* of the conventional router. In this fashion, each slice (or group of slices) corresponds to one message class of the cache coherence protocol.

In the case of the baseline router, a 64-bit control packet fits within a single 128-bit flit, i.e., one flit can accommodate both the header and the payload. On the contrary, in the proposed Sharded Router, a 64-bit control packet requires three 32-bit flits; one for the header and two for the payload. Similarly, 19 flits are required to send a data packet in the Sharded Router, as opposed to 5 flits required in the conventional router. As previously mentioned, this increase in flits will incur additional packet delay. However, through the intelligent use of two novel mechanisms, the Sharded Router eliminates this issue. These

mechanisms are presented in the following two subsections.

3.4.3 The Bandwidth-Stealing Mechanism

Despite the independence in the operation of the four slices of the Sharded Router, it turns out that it is beneficial to allow packets in one slice to utilize the crossbar switch of other slices. This activity is the central theme of the bandwidth-stealing mechanism employed in this work. In essence, bandwidth-stealing allows flits to utilize the physical channel(s) of other slices, when the other slices are idle. Figure 27 illustrates the concept of bandwidth-stealing. In this example, Slice 0 has three flits in its FIFO buffer (indicated by grey squares). Slice 1 has no flits, while Slices 2 and 3 each have one flit in their respective buffers. Since Slice 1 has no flits to be transferred, Slice 0 can "steal" its physical channel to send additional flits. Slice 2 has one flit in its buffer, but it cannot transfer it because its destination buffer is full (in the adjacent router). Thus, Slice 0 can also "steal" the physical channel of Slice 2. Since the channel of Slice 3 is in use (blue arrow), Slice 0 cannot "steal" it. Thus, by "stealing" the physical channel bandwidth of Slices 1 and 2, Slice 0 can transfer 3 flits to the neighboring router simultaneously. In order to support bandwidth-stealing, the FIFOs should be capable of reading and writing multiple flits in the same clock cycle.

Figure 28 illustrates the datapath of flits in more detail. This particular example illustrates a case where three flits depart simultaneously (i.e., in the same clock cycle) from VC0 of a slice to go to the downstream router. This feat is achieved by stealing bandwidth from other idle slices. The MUXes '5' select three flits from VC0. The MUX '3' selects

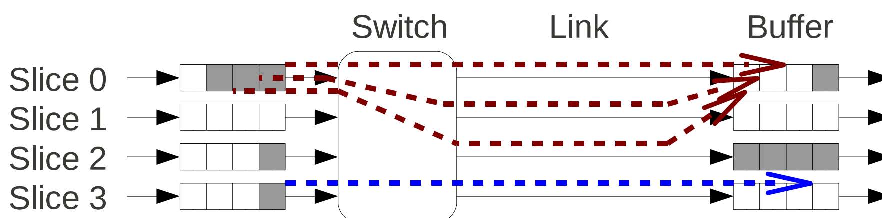


Figure 27: An example illustration of the Sharded Router's bandwidth-stealing mechanism. Flits residing in Slice 0 may "steal" the physical channel bandwidth of idle Slices 1 and 2, thus fully utilizing the available physical links.

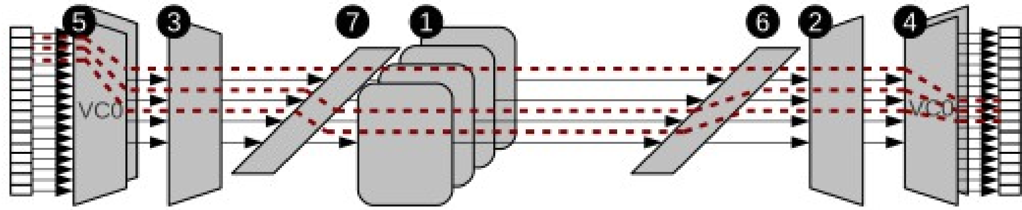


Figure 28: The datapath of flits stealing bandwidth from other (idle) slices. In this example, three flits depart VC0 of a particular slice, in the same clock cycle, by stealing bandwidth from two other slices. The flits are re-directed to their original VC and slice upon arrival at the downstream router.

VC0 among all VCs. The MUXes '7' direct two of the flits to the crossbars of *other* slices (i.e., they facilitate temporary transfer of flits between slices). Three crossbar switches '1' are activated to transfer the three flits in the same clock cycle. All flits are subsequently directed to the original slice by the DEMUXes '6.' Finally, the DEMUXes '2' and '4' guide the flits all the way to VC0 of the downstream router.

To avoid buffer overflows in the downstream routers, bandwidth-stealing is allowed only if there is enough space in the destination buffer. In the example of Figure 27, Slice 0 is allowed to transfer 3 flits, because there are 3 empty slots in the destination buffer. Since bandwidth-stealing does not reserve any resource (it merely uses resources, if they are available), it does not induce any blocking or network deadlocks.

The order of flits - which cannot be violated under the popular wormhole-switching technique employed in the majority of existing NoCs - is preserved by following the order of the slice numbering scheme. In the example of Figure 27, the first flit in the queue (buffer) must be transferred through the slice with the lowest number (Slice 0). The second flit should take the slice with the next-higher number (Slice 1), and, similarly, the last flit should use Slice 2. As an additional example, let us suppose that Slice 2 is allowed to send 3 flits through Slices 0, 2, and 3, using bandwidth-stealing. The first flit should go through Slice 0, the second flit through Slice 2, and the third flit through Slice 3. Thus, concurrent flit transfers maintain flit order by observing the slice numbering order.

Through the use of bandwidth-stealing, the per-packet latency can be substantially reduced. In the best case, the latency can be as low as in the baseline router. This enables the Sharded Router to achieve similar latencies as a baseline router, while offering significantly higher throughput.

3.4.4 Replacing Virtual Channels with a Buffer-Stealing Technique

Since each slice of the Sharded Router has only one FIFO buffer, in-flight packets may suffer from head-of-line (HoL) blocking, when the flits at the head of the buffer are temporarily blocked. Such HoL blocking is generally avoided using VCs. However, since the individual slices of the Sharded Router may deliberately be kept simple and lightweight, VCs may not be employed (this is an implementation option). Therefore, the proposed design resorts to the use of another novel technique, called *buffer-stealing*, to mitigate HoL blocking issues. Buffer stealing avoids HoL blocking without the use of VCs. This mechanism builds extensively on resources used by the bandwidth-stealing mechanism of Section 3.4.3 and uses existing data paths. Hence, buffer-stealing incurs minimal extra overhead.

In fact, the basic principle of the buffer-stealing mechanism is similar to the concept of bandwidth-stealing. When the physical channel of a slice is blocked, buffer-stealing allows the borrowing of the buffer of another slice (if it is available) in order to bypass HoL blocking. Of course, the danger when using other buffers is the occurrence of protocol-level deadlocks. To prevent such deadlocks, buffer-stealing is allowed only if it is *guaranteed* to be safe. The downstream router determines whether buffer-stealing is safe and informs the upstream router. This safety information is sent in addition to the regular buffer credits.

Figure 29 illustrates the buffer-stealing technique through a simple example. Suppose that the flits in Slice 1 of Router 0 (designated with the letter ‘B’) are destined for Router 2 through Router 1. However, their intermediate destination buffer in Slice 1 of Router 1 is full, because it is occupied by flits of a different packet, designated with the letter ‘A.’ The latter are destined for Router 3, but their respective destination buffer in Slice 1 of Router 3 is also full. In this pathological situation, no flit can move, because their destination buffers

are occupied. However, if the ‘B’ flits know in advance that they can make a short detour in Router 1, they can avoid the HoL blocking by “stealing” an idle buffer in another slice of the neighboring Router 1. For instance, the ‘B’ flits of Router 0 may steal the buffer of Slice 2 in Router 1 to bypass the HoL blocking of the ‘A’ flits, and subsequently return to their original slice in the next router (Router 2).

To prevent a protocol-level deadlock, Router 1 in Figure 29 is responsible to report on buffer-stealing *safety* to upstream Router 0. In general, every downstream router should report the safety of *every slice* to its upstream neighbors. The policy chosen to guarantee safety is very conservative, but simple to implement. If *all* destinations other than the blocked destination (i.e., the destination of the blocked flits causing the HoL blocking) are available, *and* the buffer of the slice-under-test is *empty*, then buffer-stealing is deemed to be safe. This pessimistic scenario is chosen so as to limit the bit-width of the safety information to one per slice, in order to minimize the overhead. In the example of Figure 29, the blocked ‘A’ flits in Router 1 wish to be transferred to Router 3. If all destinations other than Router 3 are available (based on incoming credit information from the downstream routers), then Slices 0, 2, and 3 of Router 1 are considered safe for buffer-stealing, since they all have *empty buffers*. Hence, no protocol-level deadlock can occur as a result of buffer-stealing.

Upon receiving this safety information, Router 0 decides whether or not to steal a buffer from another (safe) slice of downstream Router 1. It will steal a buffer if the subsequent destination of the ‘B’ flits (i.e., after Router 1) is different from the destination of the blocked ‘A’ flits in Router 1. Since the ‘B’ flits are destined for Router 2 - after traversing Router 1 - while the destination of the blocked ‘A’ flits is Router 3, then buffer-stealing will enable the ‘B’ flits to bypass the HoL blocking in Router 1. Thus, Router 0 may freely choose any of the *safe* slices in Router 1 for buffer-stealing. Since the *safe* signals from Router 1 guarantee that all next-hop destinations other than Router 3 are available, then the “borrowed” buffer in Router 1 is guaranteed not to be blocked. Forward progress to

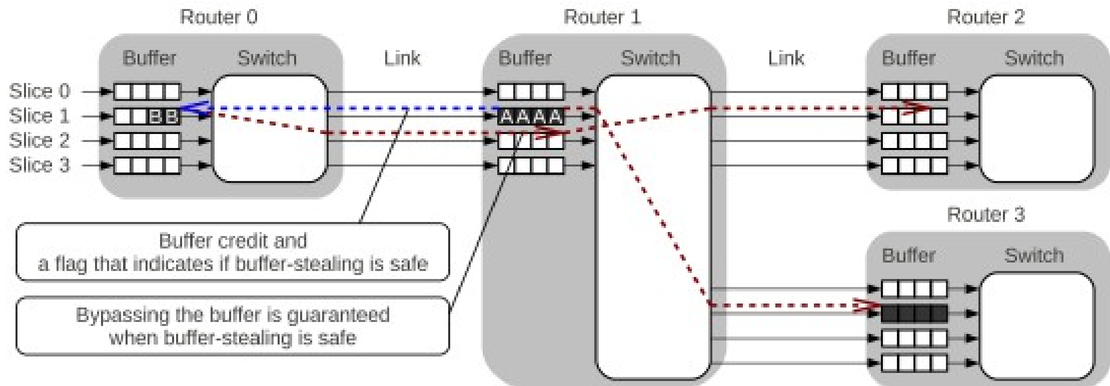


Figure 29: An example illustration of the Sharded Router's buffer-stealing technique. The 'B' flits in Router 0 can temporarily "steal" the buffer of Slice 2 in Router 1 to bypass the HoL blocking incurred by the 'A' flits. The 'B' flits can then return to their original slice (Slice 1) in downstream Router 2.

Router 2 is, therefore, also guaranteed by extension, which is what ensures the absence of protocol-level deadlocks.

In order to implement the buffer-stealing mechanism, each router must be able to perform *next-hop routing*, which is a well-known technique [75]. In other words, the router must be able to compute a packet's output destination in the downstream router (i.e., the output direction after the packet reaches the next router). Moreover, each router must be able to remember the next-hop output direction of the previous packet. For example, in Figure 29, Router 0 is expected to remember the output destination of the 'A' flits (i.e., Router 3), even though the 'A' flits have already left Router 0 and now reside in the downstream Router 1.

3.5 Experimental Evaluation

3.5.1 Simulation Framework

Our evaluation approach is double-faceted; it utilizes (a) synthetic traffic patterns, and (b) real application workloads running in an execution-driven, full-system simulation environment. We employ Wind River's Simics [31], extended with the Wisconsin Multifacet GEMS simulator [6] and GARNET [76], a cycle-accurate NoC simulator. Without loss of generality, all simulations assume deterministic XY routing.

Synthetic traffic patterns are initially used in order to stress the evaluated designs and isolate their inherent network attributes. For synthetic simulations, GARNET is utilized in a "network-only" mode, with Simics and GEMS detached. Uniform random traffic and hotspot traffic are then injected into the network. The GARNET simulator cycle-accurately models the micro-architecture of the routers. The two main designs under investigation in this paper (baseline and Sharded Router) were implemented within GARNET.

To assess the impact of the proposed router on overall system performance, we then simulate a 64-core tiled CMP system (in an 8×8 mesh) within the aforementioned full-system simulation framework. The simulation parameters are given in Table 8. The executed applications are part of the PARSEC benchmark suite [5]. PARSEC is a benchmark suite that contains multithreaded workloads from various emerging applications. All applications use 128 threads. The *MOESI-CMP-directory* cache coherence protocol is used for these experiments. It requires at least three *virtual networks* (i.e., at least three VCs) to prevent protocol-level deadlocks. As previously mentioned, our designs use four VCs for practical convenience (powers of two yield easier hardware implementations).

Table 9 summarizes the parameters of the NoC routers. The "Baseline" design refers to a conventional router implementation, whereas the "Proposed" design refers to the Sharded Router. For fair comparison, **we set the total channel width and the total buffer size**

Table 8: Simulated system parameters.

Processors	64 x86 Pentium 4 cores
Operating system	Linux Fedora 12 (Kernel 2.6.33)
L1 cache	32 KB, 4-way, 64-B cacheline
L2 cache (shared)	16 MB, 16-way, 128-B cacheline
Main memory	2 GB SDRAM
L1 hit	3 cycles
L2 hit	6 cycles
Directory latency	80 cycles
Memory access latency	300 cycles on average

per port to be the same between the two designs under test. Additionally, we compare configurations with more VCs. $Baseline_2$ and $Baseline_4$ have 8 and 16 VCs per port, respectively, whereas $Proposed_2$ and $Proposed_4$ have 2 and 4 VCs per *physical network*, respectively, which amounts to the same total of 8 and 16 VCs *per port*, respectively. The subscripts '2' and '4' indicate the number of VCs per virtual network, i.e., $Baseline_2$ has 2 VCs in each of its 4 virtual networks (required by the cache coherence protocol), i.e., a total of 8 VCs per port. $Baseline_2$ has the same amount of buffers as $Proposed_2$, while $Baseline_4$ has the same amount of buffers as $Proposed_4$. When the number of VCs is more than 1 per physical network, the buffer-stealing technique is disabled, since the VCs mitigate the HoL blocking issue (see Section 3.4.4).

3.5.2 Performance Evaluation

We first demonstrate the results with synthetic traffic patterns to assess the network performance of all designs under evaluation. For all simulations, we use *uniform random* and *hotspot* traffic patterns in an 8×8 mesh. Under hotspot traffic, 20% of nodes receive twice as many packets as the other nodes. The average network latency was measured after a warm-up period of 1,000 injected packets and while the network was at steady-state. After the analysis under synthetic traffic, we conclude this subsection with a detailed assessment using real applications running in our full-system simulation framework.

Table 9: Summary of the main parameters of the NoC routers. "Baseline" refers to a conventional NoC router implementation, whereas "Proposed" refers to the Sharded Router.

	Baseline	Proposed
Channel width per physical network (w)	128 bits	32 bits
Number of physical networks (x)	1	4
Virtual channels per physical network (y)	4	1
Buffer depth (z)	4	16
Total channel width per port ($w \times x$)	128 bits	128 bits
Total buffer size per port ($w \times x \times y \times z$)	2,048 bits	2,048 bits

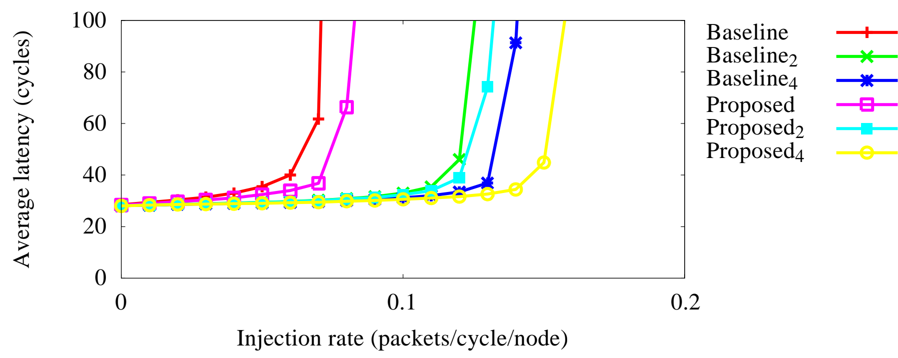
3.5.2.1 Evaluation Using Synthetic Workloads

Figure 30(a) compares performance under uniform random traffic. Because of bandwidth-stealing, the zero-load latency of the proposed Sharded Router is dramatically reduced and, in fact, becomes near-identical with that of the baseline router. When comparing Baseline vs. Proposed, Baseline_2 vs. Proposed_2, and Baseline_4 vs. Proposed_4, one can clearly observe that the proposed router exhibits better throughput than the baseline conventional design. This is due to the fact that the physical channel utilization is optimized through slicing. Furthermore, the buffer-stealing mechanism improves the throughput even more. Similar trends are observed under hotspot traffic, as shown in Figure 30(b).

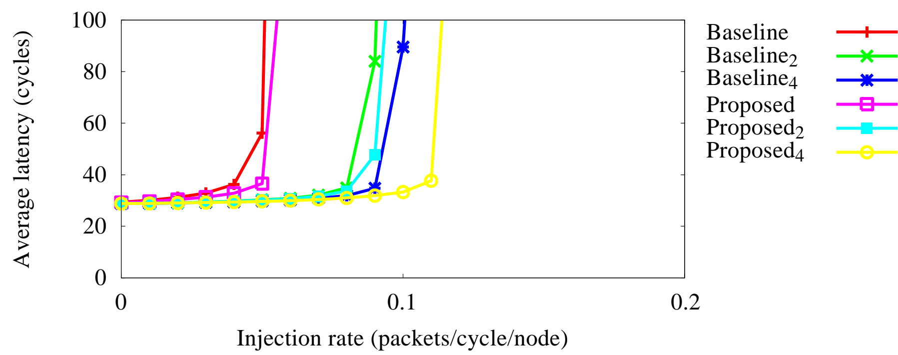
Note that the performance gap between the baseline and the proposed routers increases with the number of VCs. This is because a higher number of VCs translates into more optimized utilization of the available channel bandwidth.

The area cost of the proposed router is about 10%. Details of the cost estimation are presented in Section 3.5.3. It is true that one may consider devoting an additional 10% overhead to the baseline, instead of employing the proposed router. For example, the resulting baseline architecture may have deeper buffers, or an additional VC per input port. Figure 31 evaluates this scenario. The Proposed router has 4 VCs and each VC buffer is 16-flit deep. As given in Table 9, the corresponding Baseline router has 4 VCs and 4-deep buffers. The Baseline is enlarged by increasing the buffer depth and the number of VCs. Baseline_b has 4 VCs but each VC buffer is now 5-flit deep. Baseline_v has 5 VCs and 4-deep buffers. In a similar vein, Baseline_{4b} has 16 VCs and 5-deep buffers, while Baseline_{4v} has 20 VCs and 4-deep buffers.

Figure 31(a) compares the proposed router against the above-mentioned enlarged baselines. The Proposed router outperforms Baseline_b and exhibits similar performance with Baseline_v. Note that this is the worst-case scenario for the proposed router. If the number of VCs grows, the performance gap between the proposed router and the baseline router



(a) Uniform random traffic.



(b) Hotspot traffic.

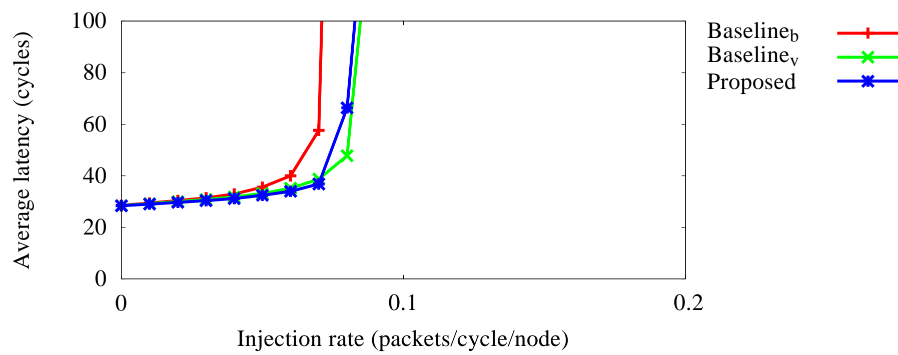
Figure 30: Performance comparison under two synthetic traffic patterns.

also grows. It is shown in Figure 31(b) that the Proposed_4 router substantially outperforms Baseline_{4b}, as well as Baseline_{4v}. These experiments confirm our claim that the proposed router exploits the physical channel more effectively than (even larger) conventional routers.

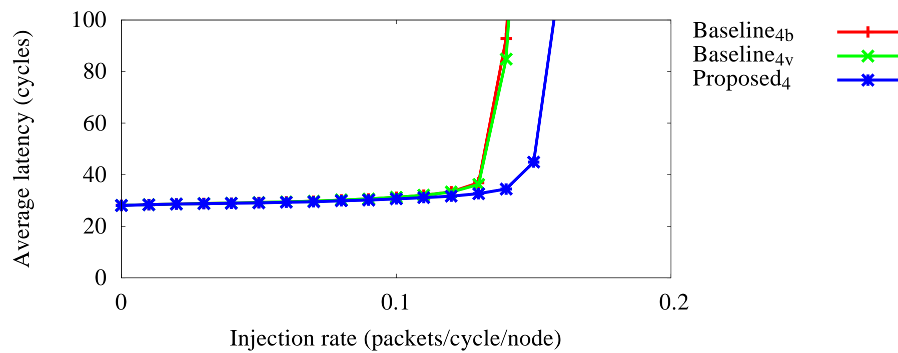
The strengths of the Sharded Router become more pronounced as the physical channel width grows. Figure 32 compares performance with (a) a 256-bit channel, and (b) a 512-bit channel. In the case of the 256-bit channel, the proposed Sharded Router has 4 separate 64-bit channels, while in the 512-bit case, it has 4 separate 128-bit channels. The throughput of the proposed router is improved substantially when compared to the baseline, because more bandwidth is wasted in conventional routers as the channel width increases. The performance of Proposed_2 is almost identical with that of Baseline_4, which has a buffer twice as large. Conversely, the Sharded Router design can maintain the same throughput as Baseline_4, but with half the buffer space.

The enhanced throughput maintained by the proposed router architecture is attributed to the much improved utilization of the channel bandwidth. Figure 33 compares the channel utilization of the baseline and proposed router designs. The utilization is measured in terms of flits/cycle/channel. The physical channel width is 128 bits and uniform random traffic is used. In the baseline router, when a flit is transferred over a channel, the bandwidth of the channel is not always fully utilized. For example, the size of a control packet is 64 bits. Assuming 32 bits are used for the header information, the remaining 32 bits are not utilized when employing a 128-bit physical channel ($128-64-32 = 32$ non-utilized bits). Therefore, the *effective* utilization (marked as "Effective" in Figure 33) of the baseline router is lower than the nominal utilization ("Baseline"). Compared to the *effective* (i.e., real) utilization of the baseline router, the Sharded Router offers higher utilization at higher injection rates, which results in higher overall throughput.

Figure 34 shows the performance contributions of the *bandwidth-stealing* and *buffer-stealing* techniques. The physical channel width is 128 bits and uniform random traffic is

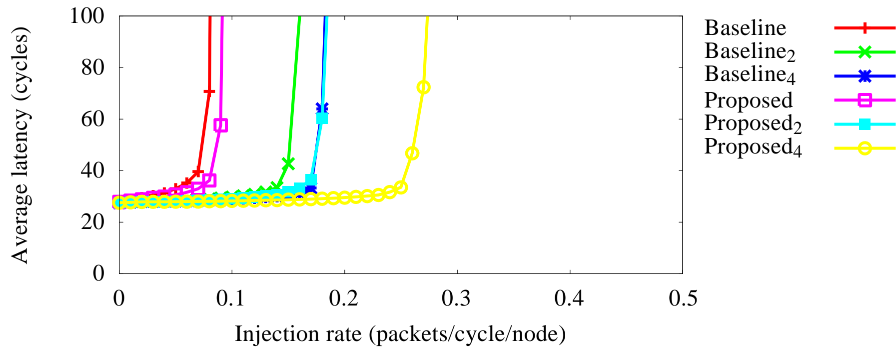


(a) Enlarged baselines derived from Baseline

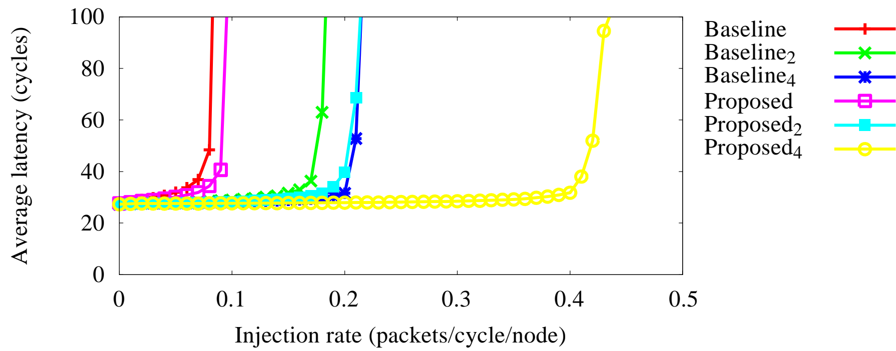


(b) Enlarged baselines derived from Baseline$_4$

Figure 31: Performance comparison with *enlarged* baselines having deeper buffers (Baseline$_b$ and Baseline$_{4b}$) and more VCs (Baseline$_v$ and Baseline$_{4v}$).



(a) 256-bit physical channel.



(b) 512-bit physical channel.

Figure 32: Performance comparison with wider physical channel widths.

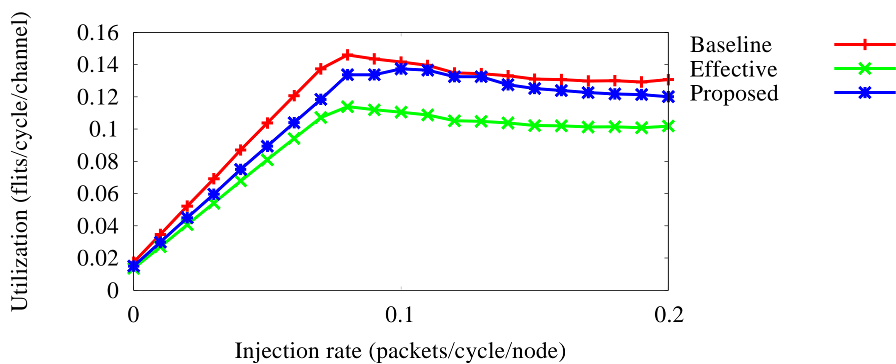


Figure 33: Physical channel utilization. The "Effective" utilization curve is the *real* utilization of the baseline router design, when the non-utilized bits within a flit are accounted for in the calculations.

used. When the 128-bit physical channel is sliced into four 32-bit channels *without* employing any stealing techniques, the zero-load latency becomes much longer than the baseline router. This barebones scenario is indicated by the “Sliced” curve in Figure 34. However, after employing the bandwidth-stealing technique, the zero-load latency decreases markedly and the throughput also improves, as shown in the graph. The throughput is further improved when buffer-stealing is also employed. Hence, the two stealing mechanisms are instrumental in optimizing the operational efficacy and efficiency of the Sharded Router.

3.5.2.2 Evaluation Using Real Application Workloads

It is true that the on-chip LLC network traffic of multithreaded applications in current CMPs is quite low and does not really stress the NoC routers [63, 77]. This is the reason why researchers often employ *multi-programmed* workloads [78], or *server-consolidation* workloads [79], to elevate the traffic within the NoC. However, the multithreaded applications of the near future are expected to utilize more and more of the available hardware resources. Obviously, as the number of on-chip cores increase to the many-core realm (i.e., tens, or even hundreds, of processing elements), the demand for network throughput will explode. Moreover, as reported in [80], the number of external memory controllers is also likely to increase, in order to accommodate the insatiable demands for off-chip memory. The stress on the NoC will inevitably increase, since the on-chip network will have to distribute the increased memory traffic. It is, therefore, imperative to develop high-throughput and high-performance router designs. The new capabilities of such designs can also be used to provide extra services to the CMP. For example, higher-throughput routers can leverage memory prefetching techniques [81, 82] much more aggressively, thus benefiting the entire system.

In order to authentically capture the expected increase in a CMP’s on-chip traffic in the near future, we employ a full-system simulator running real multithreaded workloads, and we inject a small amount of *additional* dummy traffic, similar to the methodology in [83]. This additional traffic is uniform-randomly injected *alongside the real application traffic*

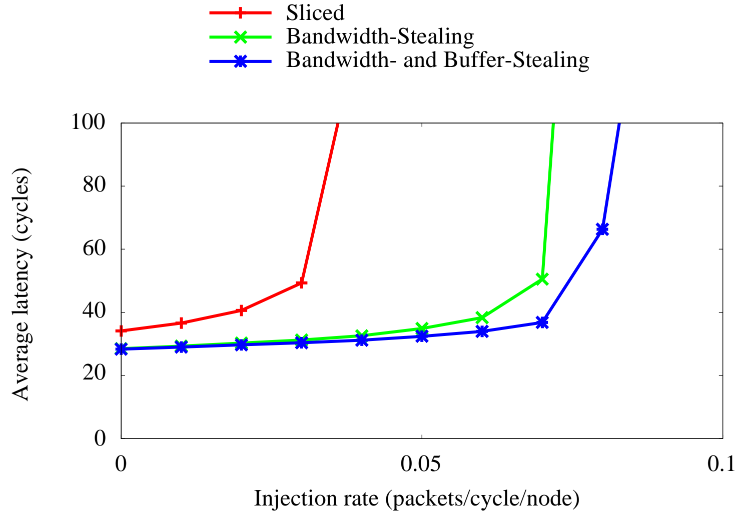


Figure 34: The performance contributions of the two stealing techniques employed in the Sharded Router architecture. The "Sliced" curve refers to a barebones sliced (sharded) router with no stealing mechanisms.

at a rate of 0.02 packets/cycle/node. To isolate the effects on the real application traffic, he dummy traffic is *not* included in the assessment statistics. Thus, the reported packet latencies and application performance indicators are derived only from the real application traffic traversing the network.

The multithreaded applications used are part of the PARSEC benchmark suite [5], and they run in the full-system Simics/GEMS/GARNET simulation framework described in Section 3.5.1. The NoC parameters are as shown in Table 9. Figure 35 summarizes the results for eight benchmark applications. Specifically, Figure 35(a) shows the *average network latency* when using the two designs under evaluation (baseline and the proposed Sharded Router). The average network latency is reduced by 6.83% to 18.86% (13.49% on average). As a result of this decrease in packet latency, the *execution time* of the applications is also reduced, as depicted in Figure 35(b). The execution time is normalized to the times achieved when using the baseline router design. The Sharded Router helps reduce the execution time by 4.10% to 43.14% (21.39% on average). Obviously, the Sharded Router architecture yields noteworthy performance improvements under real application workloads. More importantly, the new router design has been shown to perform extremely

well even under very high traffic injection rates. Thus, the attained performance boost is only expected to grow with increasing on-chip traffic demands.

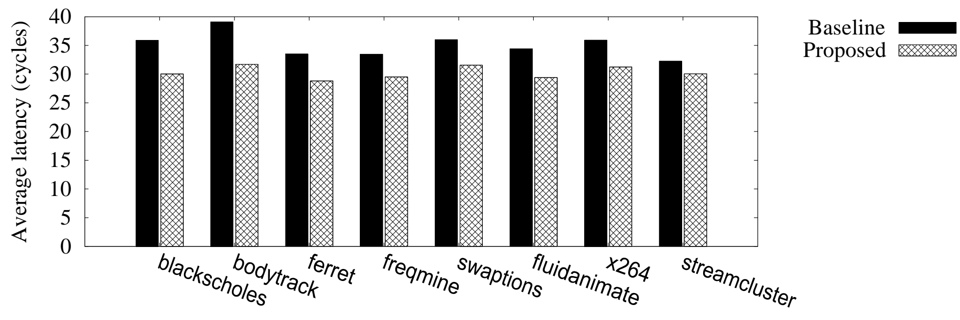
It is clearly shown in Figure 36 that the performance gain by using the proposed router grows with increasing traffic demand. The benchmark used in this experiment is `blackscholes`, but the same trend has been observed in all benchmarks. When we increase the injection rate of the *dummy traffic* (which is injected alongside the real application traffic), we can see that the average latency of the baseline router increases sharply, whereas that of the proposed router remains the same. Specifically, when the injection rate of the dummy traffic is 0.05 packets/cycle/node, the average latency is reduced by 78.68% when using the proposed router.

3.5.3 Hardware Cost Analysis

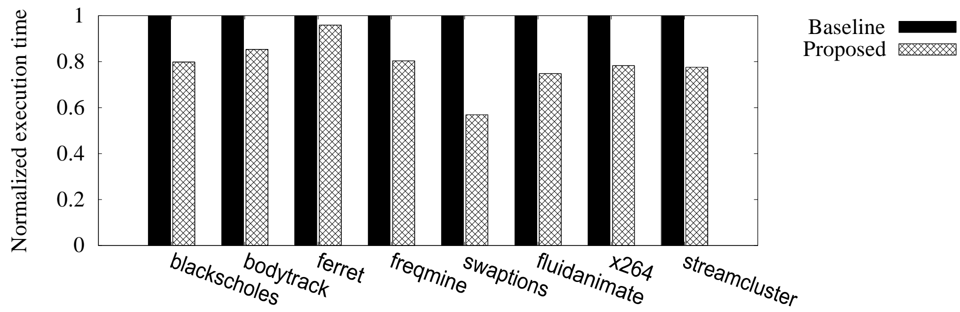
The most area-dominant components in a router are the buffers and the wide MUXes and DEMUXes (in addition to the crossbar switch) [47]. As shown in Table 9, the size of the buffers is the same in the proposed Sharded Router as it is in the conventional design. However, the additional MUXes and DEMUXes required by the Sharded Router incur some overhead. Moreover, the more elaborate control logic - which facilitates fine-grained sharding - also increases the overhead. Regardless, the total area overhead of the Sharded Router is limited to a modest 10.55%, as will be demonstrated shortly.

Since the area overhead of the crossbar switch may vary with the actual circuit implementation, we begin this subsection by providing a more generalized high-level analysis of the hardware cost of the crossbars. This analysis aims to help the reader appreciate the nuances of the Sharded Router's micro-architecture. The area overhead of the crossbar switches is estimated as $O(p^2w^2)$, where p denotes the number of input/output ports and w denotes the bit-width of the data path [47]. More specifically, we use the following equation to estimate the area overhead of a crossbar switch component:

$$a = w^2 \times i \times o \times c \tag{8}$$



(a) Average network latency.



(b) Normalized execution time.

Figure 35: Performance evaluation using a full-system, execution-driven simulation framework running real multithreaded applications from the PARSEC benchmark suite [5] on a 64-core CMP.

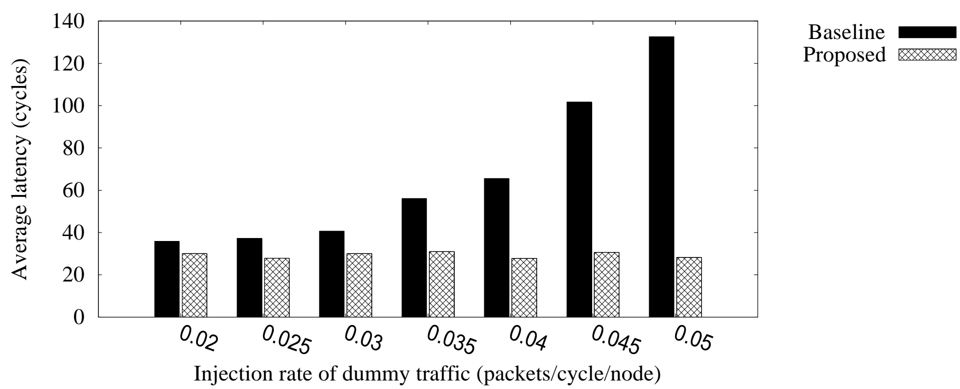


Figure 36: Sensitivity analysis on the injection rate of the *additional dummy traffic* injected alongside the real application traffic of the multithreaded workload. The multithreaded benchmark used here is `blackscholes`.

In the above equation, *w* is the bit-width, *i* is the number of input ports, *o* is the number of output ports, and *c* is the number of copies (instances) the switch is used in the design.

Based on this equation, the unshaded (top) part of Table 10 compares the area overhead of the crossbar switches and the MUXes/DEMUXes of the "Baseline$_2$" and "Proposed$_2$" router designs. Note that the *Component* numbers in the left-most column of the table refer to the hardware components of Figure 26. We compare these designs, in particular, instead of "Baseline" and "Proposed," because the DEMUXes '2' and MUXes '3' in Figure 26 are not required in the simple "Proposed" configuration (i.e., when only one VC is present in each slice). Hence, had we compared the "Proposed" configuration, the hardware cost of the Sharded Router would have been underestimated. Instead, by assessing the "Proposed$_2$" configuration, we accurately account for *all* additional hardware components.

As can be seen in Table 10, the hardware cost of component '1' - which is the main crossbar switch - is reduced by slicing the physical channels. The cost of components '2,' '3,' '4,' and '5' is the same between the two designs, because the Sharded Router merely splits the modules into multiple smaller pieces. The additional cost from components '6' and '7' is somewhat outweighed by the reduced overhead of component '1.' In total, the hardware area cost of the crossbar switches and MUXes/DEMUXes increases by an estimated 5% in the case of the proposed Sharded Router.

After analyzing *analytically* the hardware cost of the crossbars and MUXes and DEMUXes, we proceed with **the actual gate-count results of the *entire* router designs**. Both routers under investigation were fully implemented in synthesizable Verilog HDL and synthesized using Synopsys Design Compiler. The reported gate counts for the *complete* router implementations are shown in the shaded (bottom) part of Table 10. The hardware area overhead of the entire Sharded Router in terms of gate count is approximately 10.55%, which is a modest cost compared to the enormous performance benefits demonstrated in Section 3.5.2. The critical path delay of the proposed router is longer, but the increment is not significant. Specifically, the proposed router's critical path delay is reported as 1.46

Table 10: Hardware cost comparison between the $Baseline_2$ and $Proposed_2$ designs.

Component (see Figure 26)	$Baseline_2$					$Proposed_2$				
	w	i	o	c	a	w	i	o	c	a
1	128	5	5	1	409,600	32	5	5	4	102,400
2	128	1	8	5	655,360	32	4	8	20	655,360
3	128	8	1	5	655,360	32	8	4	20	655,360
4	128	1	4	40	2,621,440	32	4	16	40	2,621,440
5	128	4	1	40	2,621,440	32	16	4	40	2,621,440
6						32	4	16	5	327,680
7						32	16	4	5	327,680
Total for Crossbar Switches and MUXes/DEMUXes	Sum of a's				6,963,200	Sum of a's				7,311,360
	Percentage				100.00%	Percentage				105.00%
Entire router	Total gate count				114,901	Total gate count				127,028
	Percentage				100.00%	Percentage				110.55%
Critical path delay	Critical path delay				1.36 ns	Critical path delay				1.46 ns
	Percentage				100.00%	Percentage				107.35%
Power consumption	Power consumption				289.24 mW	Power consumption				249.36 mW
	Percentage				100.00%	Percentage				86.21%

ns, which is 7.35% longer than that of the baseline. Thus, it is important to evaluate performance while accounting for this drop in maximum operating frequency (as a result of the longer critical path). Figure 37 compares the performance in terms of *time*, instead of *cycles*, when considering the difference in the maximum clock frequency. The period of one clock cycle in the baseline router is 1.36 ns (as per the synthesis results of Table 10), while that of the proposed router is 1.46 ns. Obviously, even if we take the critical path delay into consideration for a performance comparison, we can see that the proposed router still offers significant performance improvement over the baseline.

The power consumption of the proposed router is reduced by 13.79% compared with the baseline. The baseline router wastes power by using the wide buffer entries (flits are much wider in the baseline router), even if the entire flit width is not fully utilized, whereas the proposed router uses narrower buffer entries that are better utilized. Since the sliced buffer entries are much narrower than the wide entries of the baseline, the proposed router allows for finer granularity in the utilisation of buffer space, which is known to be one of the primary power consumers in on-chip routers.

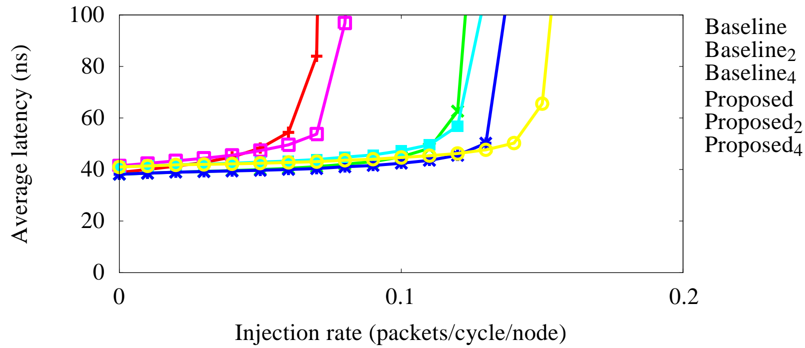


Figure 37: Performance comparison in terms of *time* (instead of *cycles*), in order to account for the longer critical path in the proposed router. One clock cycle in the baseline router is 1.36 ns, while that of the proposed router is 1.46 ns (as per the hardware synthesis results of Table 10).

3.6 Conclusion

In addition to enabling massive transistor integration densities, technology downscaling has also facilitated the widening of on-chip communication links. The inter-router physical links in modern NoC-based multicore micro-processors range in width from 128 to 256 bits (in each direction), while even wider parallel links are being investigated. However, this increase in bit-level parallelism is not yielding proportional improvements in network performance, because the extra link bandwidth is not fully utilized. The typical packet size is not always a multiple of the channel width, thus wasting valuable channel resources.

This chapter addresses the problematic facet of under-utilized wide parallel NoC links by proposing a novel router micro-architecture that relies on bandwidth slicing. The *Sharded Router* employs fine-grained bandwidth sharding (i.e., partitioning) to decompose the NoC into multiple narrower independent networks. Furthermore, the proposed new router design relies on two optimization techniques to further boost performance and throughput. The *bandwidth-stealing* mechanism lowers the zero-load latency of the individual sub-networks, by utilizing idle link bandwidth in the other sub-networks. Thus, link utilization is maximized. The complementary *buffer-stealing* technique avoids HoL blocking when there is only one virtual channel per physical network.

Detailed experiments using both synthetic traffic traces and real multithreaded application workloads running in an execution-driven, full-system simulation framework corroborate the efficacy and efficiency of the Sharded Router. Specifically, the proposed design reduces the average network latency of real benchmark applications by up to 19% and their execution time by up to 43%. More importantly, the Sharded Router's throughput benefits seem to increase as the physical channel width increases. Finally, hardware synthesis analysis using a commercial-grade tool indicates that the hardware overhead of the new router architecture is contained to approximately 10% over a conventional design.

CHAPTER 4

A PROGRAMMABLE PROCESSING ARRAY ARCHITECTURE SUPPORTING DYNAMIC TASK SCHEDULING AND MODULE-LEVEL PREFETCHING

The widespread adoption of MPPAs as general-purpose hardware accelerators faces several challenges. One of them is the *expressiveness* of the execution model. Both GPUs and the latest accelerated processing units (APU, a term coined by AMD for their CPU/GPU Fusion line of products) currently employ the venerable SIMD model. While this is a powerful model, it is suitable only for certain applications with regular computational kernels, such as graphics applications. Moreover, within the context of parallel programming, *debugging* is an often forgotten challenge that is very important in real-world applications. Finally, from the hardware perspective, the *memory hierarchy* is one of the most challenging design decisions. For relatively small numbers of cores, the cache is adequate, but for large numbers of cores, the cache coherence protocol becomes a bottleneck, as it does *not* scale well [84].

This chapter aims to address all three of the aforementioned challenges that impede the consolidation of MPPAs as the de facto processing archetype of the future. We hereby propose a hardware architecture for MPPAs, which supports an event-driven execution model. The combination of said event-driven execution model and appropriate support from the hardware architecture enables us to overcome these challenges.

The key contributions of the proposed architecture are *dynamic task scheduling* and *module-level prefetching*. While previous architectures supporting a similar execution model determine task mapping at compile-time, our proposed architecture allows *run-time dynamic scheduling*. This attribute allows for better expressiveness. At the same time, the execution model imposes sufficient limitations on the semantics for a better debugging environment. In order to overcome the run-time overhead incurred by the dynamic task

scheduling, we employ *module-level prefetching*, which also hides the memory access latency. By exploiting the fact that the execution model forces the input data of a module to be explicit, the hardware can prefetch instructions and data while other modules are running. Since prefetching is performed at the module level, it works accurately regardless of any data dependencies and branches. Finally, the proposed execution model does not assume a global shared memory, thus eliminating the need for a cache coherence protocol and offering markedly better scalability.

Extensive simulations using a cycle-level simulator of the proposed architecture running real application benchmarks demonstrate the capabilities and effectiveness of the new processing paradigm. Our results are extremely promising and clearly highlight the vast potential of such architectures.

The rest of this chapter is organized as follows: Section 4.1 discusses and analyzes prior related work. Section 4.2 gives a motivational example that is also used as an illustrative example throughout the chapter. Section 4.3 defines the utilized execution model. The proposed hardware MPPA architecture and its architectural support for the execution model are introduced in Sections 4.4 and 4.5, respectively. Section 4.6 presents the employed evaluation framework, the various experiments, and accompanying analysis. Finally, Section 4.7 concludes the chapter.

4.1 Related Work

Figure 38 shows a high-level overview of a typical microprocessor architecture employing MPPA as a programmable hardware accelerator. The assumption is that the main CPUs and the MPPA are integrated on the same die (akin to the latest trends in the industry). The MPPA comprises a multitude of small cores and supporting logic. The latter includes the interconnection network, a memory sub-system, and hardware support for the programming model.

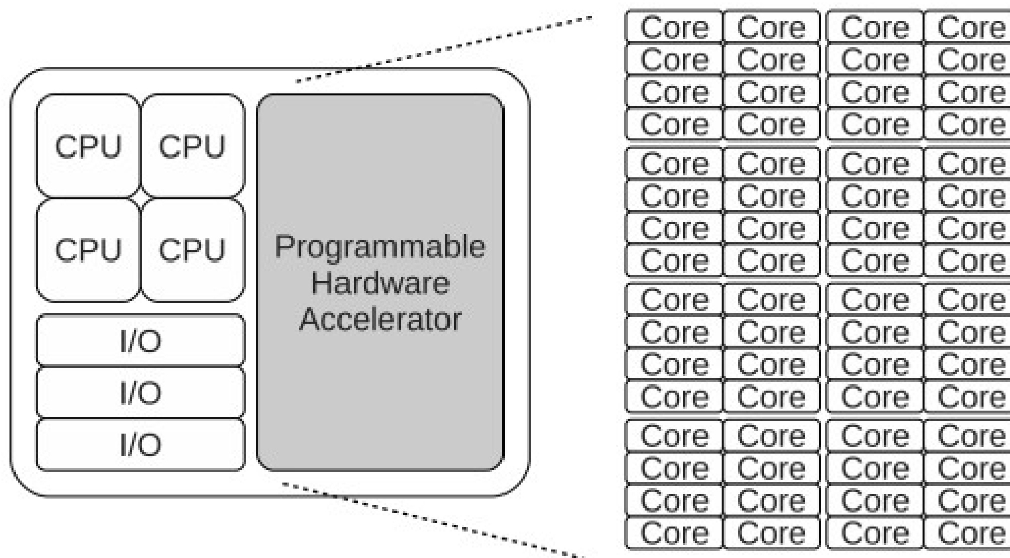


Figure 38: A high-level overview of a processor architecture employing a Massively Parallel Processing Array (MPPA) as a programmable hardware accelerator.

Coarse-grain reconfigurable architectures (CGRA) [85] share a similar concept. However, the basic building block of CGRAs is an arithmetic and logic unit (ALU), while that of MPPAs is a whole CPU core. The target architecture of CGRAs consists of ALUs and a reconfigurable interconnection infrastructure. The designer can modify the functionality of the system by reconfiguring the interconnections among the various ALUs. Since CGRAs have ALUs - not generic processors - as their primary primitive, they are only amenable to the implementation of data-path-dominated algorithms, not control-oriented algorithms.

A similar architecture can also be found in the Cell microprocessor [86] architecture. It consists of a power processor element (PPE) as a main CPU and eight synergistic processing elements (SPEs) acting as an accelerator. If the number of SPEs were tens, or hundreds, we could classify this architecture as MPPA.

As previously mentioned, commercialized MPPAs - including GPGPUs [87] and AMD's APU [8] - adopt the SIMD model. In academia, the stream processor [88] is a well-known SIMD-type processor. The SIMD model is effective for applications with regular computational kernels, whereby the same kernel is replicated on a number of cores. All the cores execute the same job, with different data. However, as modern algorithms are getting more

complex and irregular (in order to accommodate more functionality), the need for a more flexible programming model is growing.

Tilera [89] and Rigel [84] support a standard multithreading programming model, thus providing the programmer with the maximum (known) flexibility. This programming model can be applied to any kind of parallel algorithm. However, the same model also imposes significant burden on the process of debugging, and the hardware itself. Maximum flexibility makes debugging very difficult, because there are too many possible causes for unexpected behavior. Without careful synchronization and protection, the program is likely to be unreliable and unpredictable. As for the hardware, a cache coherence protocol must be implemented, in order to support the shared memory assumption of the multithreading programming model. Tilera [89] employs a dynamic distributed cache (DDC), but its scalability is still not proven for the 1000-core systems similar to Rigel [84]. Rigel implements a 1000-core accelerator, but the coherency of its caches is maintained by software.

The Ambric architecture [90] is the MPPA implementation that is most relevant to our work, because it adopts a similar execution model, i.e., a Kahn process network (KPN) with bounded queues. Mapping tasks on processing elements is determined at compile-time. At run-time, it does not allow *dynamic task scheduling*. This restriction limits the expressiveness of its execution model, because new tasks cannot be instantiated and their interconnection cannot be modified at run-time. Moreover, if there are dependencies among tasks, there may exist idle processing elements that are waiting for results from other tasks. Instead, dynamic task scheduling offers better expressiveness and yields higher utilization.

Our dynamic task scheduling policy follows a simple first-come, first-serve algorithm. However, task scheduling should consider resource constraints, communication cost, and performance issues, among others. There has been significant prior work in this domain, especially aimed at multi-processor systems-on-chip (MPSoC) [91, 92]. We believe that the specific algorithm employed by the dynamic task scheduler is orthogonal to this work, so we leave this analysis for future work.

It is true that dynamic task scheduling incurs run-time overhead. If the size of tasks is small and the number of processing elements is large, the overhead can be excessive [93]. To overcome this overhead, and to hide memory access latency, we adopt *prefetching* in this thesis. For GPGPUs, an inter-thread prefetching technique has been proposed [94]. The authors exploit the common memory accesses among threads and devise a throttling mechanism to avoid performance degradation from mis-predictions. Our approach is to exploit the execution model itself and take advantage of the fact that all input data should be explicitly declared for every task. While a task is running, a hardware prefetcher prefetches all the data for the next task. Since input data is explicitly associated with the task, prefetching is always accurate.

The *execution model* can be derived from various programming models. Our event-driven execution model can support various models of computation, including KPN, synchronous data-flow graphs, finite state machines, etc. StreamIt [95] adopts the data-flow model, which can also be supported by our event-driven execution model. Previous work on programming models [96] is complementary to our work.

4.2 Motivational Example

The *quicksort algorithm* [32] is used throughout this chapter as an illustrative example. Figure 39 illustrates the parallelism exhibited in the quicksort algorithm. Given an array of values to be sorted, a pivot is selected, which is usually the first element in the array. The array is partitioned so that the left side of the pivot contains smaller elements than the pivot, while the right side contains larger elements than the pivot. Subsequently, the same partitioning is done recursively and independently on each side.

Once the partitioning of a segment finishes, its sub-segments can commence partitioning. However, the partitioning of the individual segments of the array can be done independently and simultaneously. Given a large array size, the quicksort algorithm exhibits abundant parallelism, as illustrated in Figure 39.

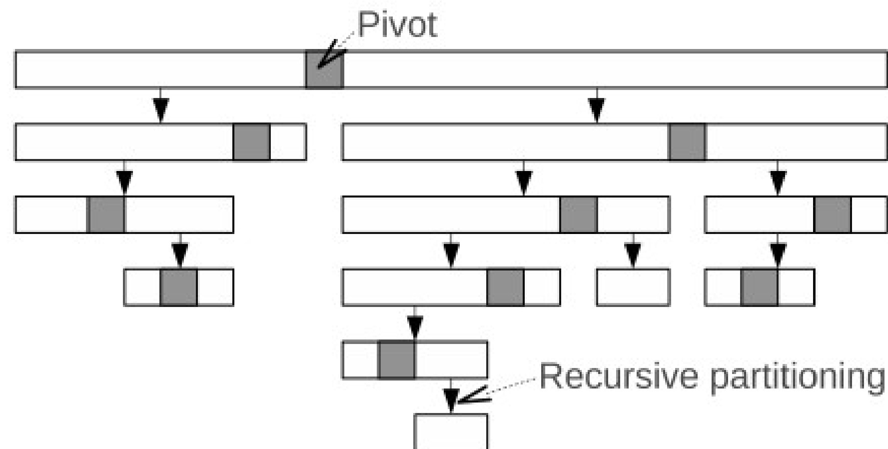


Figure 39: Illustration of the parallelism exhibited in the quicksort algorithm.

Although partitioning can be done independently, it does not mean that all the partitioning processes take exactly the same code path. Depending on the elements in the array, each partitioning may take a different path of the code. Moreover, one does not know at compile-time how many times recursive partitioning is needed. This information is dependent on the input data and is determined at run-time. If a multithreading programming model is used, we can create new threads for the partitioning of the sub-segments. In contrast, GPGPU does not allow spawning of new threads at run-time. In such a case, we may employ *job queueing* instead of spawning new threads [93]. Using this approach, the threads are created at initialization. Every thread fetches a job from a centralized (or distributed) job queue(s). If there is no job in the queue, some threads may become idle. When a new job is created, it is pushed into the queue, thus obviating the need to create a new thread.

Algorithmic nuances render the SIMD implementation of quicksort inefficient. Figure 40 shows the execution time of quicksort when varying the number of threads. The execution time is measured on NVIDIA's Quadro NVS 295 GPU, which has 8 CUDA cores per multiprocessor. In a GPU context, a *multiprocessor* consists of multiple CUDA cores and a memory that is shared by the cores within the multiprocessor. A thread block is mapped to a multiprocessor and threads in the block are executed by the cores in the

mapped multiprocessor. In this experiment, the number of blocks is fixed to one and only the number of threads per block is varied. Hence, in this setting, all the threads are executed in a single multiprocessor. Assuming all the cores within the multiprocessor are fully utilized, one would expect the execution time to decrease proportionally to the number of threads at least up to 8 threads, since there are 8 cores in a multiprocessor. Instead, Figure 40 shows a different result for quicksort (*QS on GPGPU* curve).

As seen in Figure 40, the execution time of quicksort does not benefit from an increasing number of threads (*QS on GPGPU* curve). In contrast, the execution time of vector addition does (*VA on GPGPU* curve). Vector addition adds two vectors by adding each corresponding element in the vectors. This algorithm is a typical example that is suitable for the SIMD model. It is obvious that the performance of vector addition is improved, even with a small number of threads.

To provide another reference for comparison, we conducted the same experiments on a multicore CPU machine, which employs the multithreading execution model. The execution time on the multicore machine is measured using the Simics full-system simulator [31]. Eight x86 processors are assumed, and the operating system on the simulated machine is Fedora 12 (Linux kernel 2.6.33). It is evident that the execution time of quicksort (*QS on multicore* curve) scales well up to 8 threads. However, as discussed in Section 4.1, the multithreading model faces scalability issues because of the cache coherence protocol. We may not be able to sustain any performance improvement beyond tens of processors.

Note that this experiment demonstrates an inefficiency of the SIMD model, *not* of the GPGPU paradigm. When the number of blocks and the number of threads increase far beyond 8, the performance of quicksort may benefit from various aspects of GPGPU support, including multiple number of multiprocessors, warp scheduling, thread multiplexing to hide memory latency, and so on. Regardless, what this experiment shows is that the hardware resources are not fully utilized, because of the limitations of the SIMD model.

One more alternative method to implement the quicksort algorithm is by using KPN, as

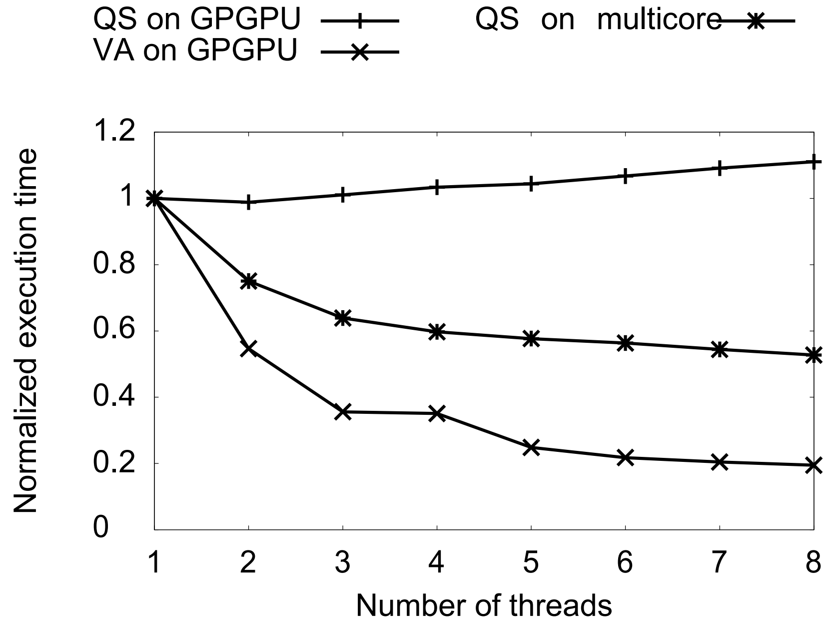


Figure 40: Inefficiency of the SIMD model for applications with irregular computation kernels.

in Ambric [90]. Unfortunately, it is very hard to express the dynamic nature of quicksort by using KPN, because we do not know how many times we need the recursive partitioning process, while the task mapping and the interconnections are required to be determined at compile-time.

It should be noted that it is always possible to tailor a specific algorithm for a particular model, just like GPU-Quicksort [97] does. However, what is discussed in this chapter is a *general* way to implement algorithms.

4.3 The Execution Model of the Proposed MPPA Architecture

This section provides the definition and details of the *execution model* of our proposed architecture.

4.3.1 Specification

The execution model consists of a set of modules M, a set of signals S, and a net list N. A module $m \in M$ is defined as a tuple of behavior b, an input port list P_i, an output port list

P_o, a sensitivity list C, and a prefetch list F.

$$m = (b, P_i, P_o, C, F) \in M \tag{9}$$

b is a set of instructions that specifies the behavior of the module. In fact, b can be viewed as a program including computation, memory access, function calls, etc. C and F are a subset of P_i ($C \subseteq P_i$, $F \subseteq P_i$). C indicates when this module should be executed, and F determines the prerequisite data before running this module. The *internal state* of a module can be represented by a feedback signal from the output to the input of the same module.

N defines the connectivity of ports and signals. Each signal $s \in S$ should have a corresponding unique entry $n(s)$ in N. $n(s)$ is defined as a tuple of a driver port d and a set of sink ports K.

$$n(s) = (d, K) \in N \tag{10}$$

$n(s)$ indicates a signal s is connected to ports d and $\forall k \in K$. Data is written only through port d and broadcast to all the ports in K.

4.3.2 Semantics

A module is triggered when any signal connected to the ports in C changes. To execute the module, the instructions (b) and signals connected to ports in F should be prefetched. Once they are ready, a module is executed. The execution of a module is atomic, i.e., a module cannot be stopped until it finishes its execution. The atomic execution semantics eliminate the need for explicit synchronization primitives, such as locks and barriers. Instead, the communication channel serves as the synchronization primitive [9].

Function calls and memory accesses are strictly limited to within a module. b can access only functions within its own module boundaries (i.e., b) and signals connected to P_i and P_o. There is no global shared memory. These features ensure *encapsulation*. As highlighted in [90], encapsulation facilitates efficient *debugging*, since it limits the possible causes of errors within its own code body (i.e., b) and input signals.

The communication semantics follow non-blocking writes and blocking reads. In practice, since the depth of FIFOs cannot be infinite, the write may be blocked when the FIFO is full. There should only be one driver for a channel, but multiple sinkers can read data from the channel. Written data is broadcast to all sinkers connected to that signal.

Since the hardware architecture supports *dynamic task scheduling*, the descriptions of M, S, and N are allowed to be reconfigured at run-time. This feature offers more expressiveness to the execution model, and it improves the utilization of hardware resources.

A module can be instantiated and destroyed at run-time. The sensitivity list (C) and the prefetch list (F) can also be modified at run-time. Moreover, signals may be instantiated and destroyed at run-time, which implies that the addition and removal of nets from N are allowed. Finally, the model also allows the modification of d and K in $n(s)$.

4.3.3 Using the Event-Driven Execution Model

This subsection illustrates how the quicksort algorithm may be specified with the event-driven execution model. The algorithm would consist of two modules: `partition` and `collection`, as illustrated in Figure 41. The `partition` module partitions the given array into two sub-arrays. It partitions one of these again and passes the other to a new module. It instantiates another `partition` module and a signal, it connects the signal to the new module, and it sends the sub-array to the new module. If partitioning is finished, the final output is sent to the `collection` module that collects all the sorted segments of the array. The `collection` module generates the final output when all the segments are collected.

The `partition` module has one input port (P_i) and two output ports (P_o). The input port is included both in the sensitivity list (C) and the prefetch list (P), which means that whenever the signal connected to the input port changes, the `partition` module is triggered, and - before the module is executed - the input signal is prefetched.

The input port consists of *start* and *end* positions, as well as the actual array of elements to be sorted. The *start* and *end* positions are necessary to inform the `collection` module

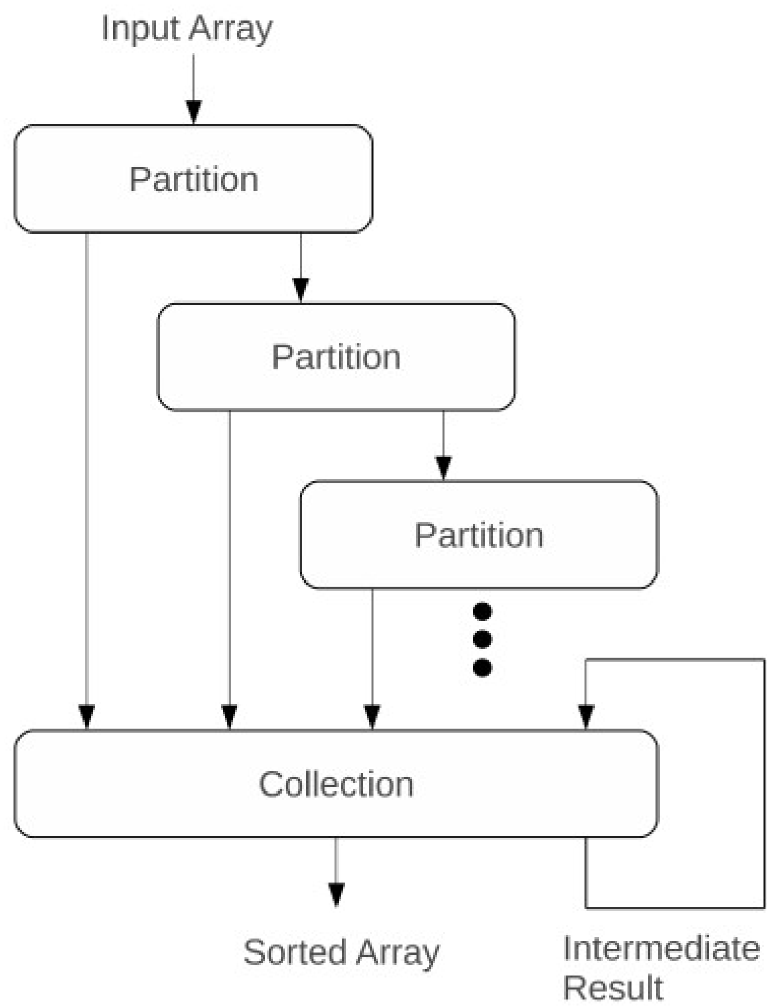


Figure 41: Module diagram of the quicksort algorithm, as specified using the proposed event-driven execution model.

which segment of the input array is to be sorted. The *start* and *end* positions do not need to be sent through separate ports, because the semantics of the port comprise a stream of bytes with variable length. As long as the sender and the receiver agree, any aggregated type of data can be carried through the port.

The input ports of the `collection` module are ports for the `partition` modules to send their outputs, and an extra port for the intermediate result. The two output ports are for the final output and for the intermediate result. The intermediate result stores the collected sorted segments so far. It can be considered as the state of the `collection` module. This input port should be included in the prefetch list, but not in the sensitivity list, since the intermediate result needs to be prefetched before the module is executed, but its change does not need to trigger the module.

4.4 The Hardware Architecture

Our proposed MPPA architecture consists of several identical *tiles*, as shown in Figure 42. A conventional NoC is used to interconnect the nodes (core tiles). Although core tiles are identical, we designate one of them as the *execution engine* (denoted as '*E*' in Figure 42). The execution engine is implemented in software, which runs on the μCPU of the particular core tile. The execution engine is placed in the middle of the MPPA, so as to minimize the average distance to/from the other nodes. The execution engine consists of a *scheduler*, *signal storage*, and *interconnect directory*. All data managed by the execution engine is stored in the device memory. The scratch-pad memory of the execution engine node is used as a software-managed cache memory. Recall that the execution model consists of modules (*M*), signals (*S*), and a net list (*N*). The scheduler manages and schedules the states of modules. The signal storage stores signal values and the locations of signals (if the signals are fetched by nodes). The interconnect directory keeps track of the connections of ports and signals.

The host CPU interface facilitates interaction with the system's main CPU(s). From

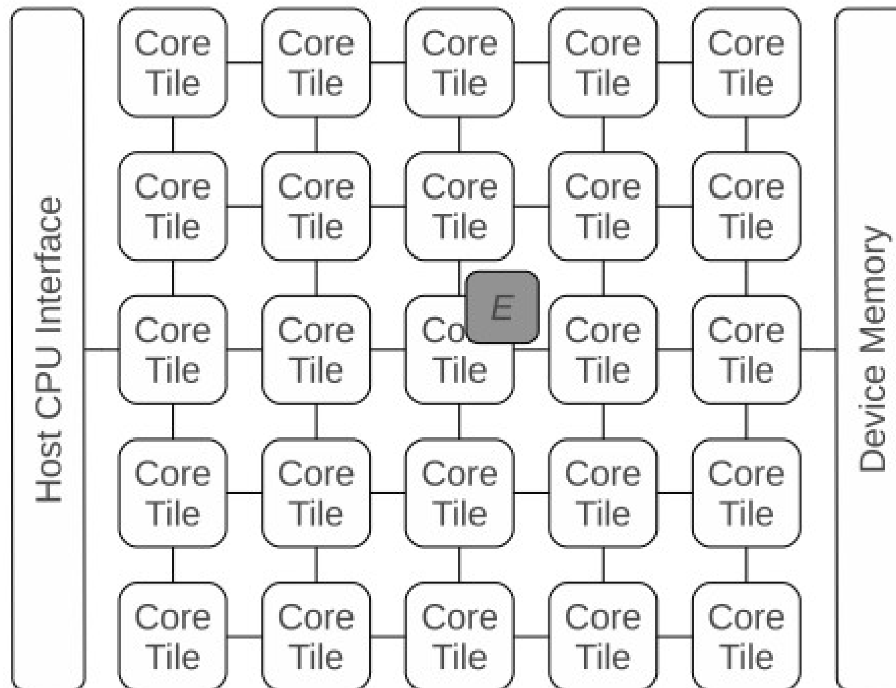


Figure 42: The proposed MPPA microarchitecture consists of several identical *tiles* interconnected using an on-chip interconnection network.

the viewpoint of the execution model, a host CPU is treated as a module. The core tile connected to the host CPU interface is dedicated to handling the interactions with the host CPU(s).

The MPPA also makes use of *device memories*. The device memories have larger capacity - but longer access latency - than the scratch-pad memory of the core tile shown in Figure 43. Only execution engines can *directly* access the device memories. Other nodes are required to place a request to the execution engine. The device memory is separated into multiple banks for concurrent accesses by various execution engines. This segregation aims to eliminate conflicts on the device memory by accesses from different execution engines.

A detailed block diagram of one core tile is depicted in Figure 43. A core tile consists of a scratch-pad memory, a context manager, input/output queues, a message queue, a prefetcher, a message handler, and a network interface. Only nodes serving as execution engines have their memory, message queue, and network interface enabled.

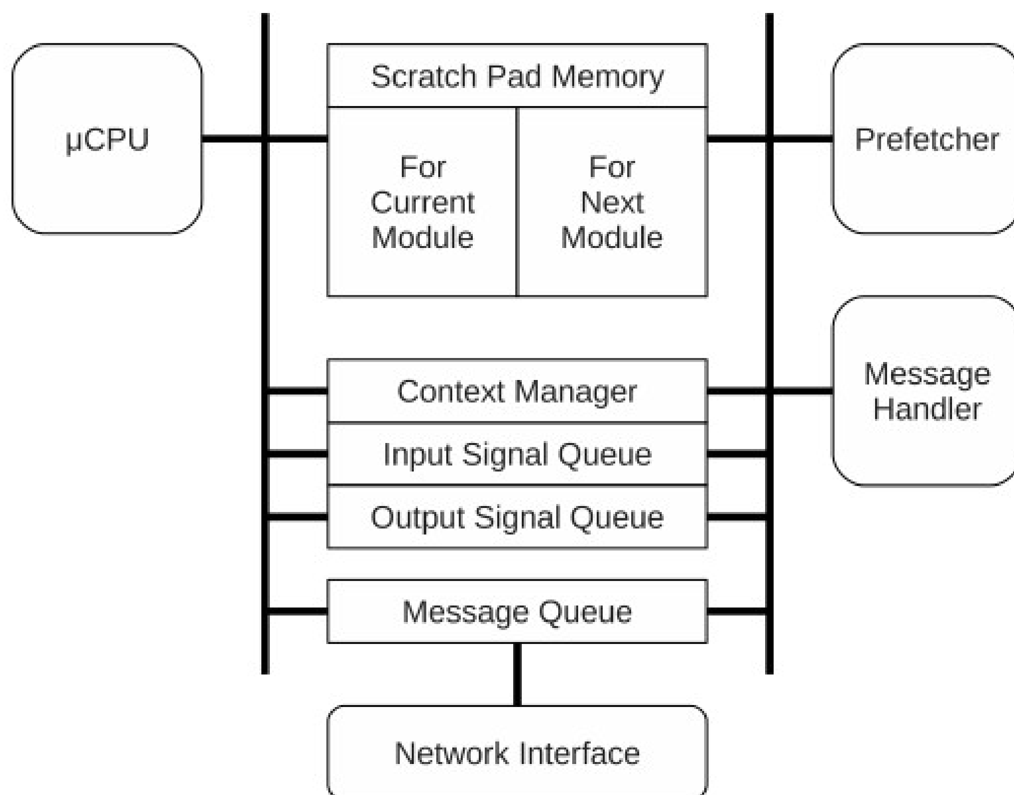


Figure 43: Block diagram of a *single* core tile of the many-core MPPA architecture shown in Figure 42.

The scratch-pad memory is, essentially, a double buffer. Half of the buffer is dedicated to the current module and the other half is reserved for the next module. While the μCPU accesses the current module's half, the prefetcher prefetches code and variables to the next module's half. The two buffer halves switch their roles upon receiving a control signal from the context manager.

The context manager is accessed when the current module completes its execution. If there is no other available module to run, the context manager disables the μCPU. Otherwise, it sends control signals to the memory and peripherals to switch to the next module, and then it restarts the μCPU so as to run the next module.

The input queue retains input signals for the current module and the next module. The input signals for the next module are prefetched by the prefetcher. Input signals for the current module are discarded when control signals from the context manager indicate a context switch.

The output queue stores the output signals. When an output signal is updated, a control message is sent to the interconnect directory to trigger those modules whose sensitivity list includes the updated signal. The actual data is kept in the output queue until the context manager triggers a context switch. When context switching is triggered, the output queue flushes the output signals to the signal storage.

The message queue is used to send and receive control messages. Although a complete list of control messages is not given in this thesis (see next section), it is assumed that all control messages are defined by the system. Signals are carried within control messages.

The prefetcher is responsible for prefetching all the necessary inputs and instructions. When a control signal arrives from the context manager, the prefetcher commences operation.

The message handler is a counterpart to the prefetcher. Some input signals of a particular module may be stored in *other nodes*, instead of the signal storage. In such a case, the signal storage forwards a request message to those nodes. When such requests arrive at

the requested nodes, the message handler reads the requested signal from the output signal queue and forwards it to the requester.

Finally, the network interface is a typical NoC router/switch. It supports multiple outstanding requests for the concurrent prefetching of multiple input signals.

4.5 Architectural Support for the Execution Model

This section explains how the hardware architecture supports the execution model.

4.5.1 Execution Engine

The heart of the architectural support is the execution engine. While the hardware facilitates communication, most of its functionality is implemented in software running on the μCPU. Implementation in software gives us flexibility in the number and location of execution engines, which will be demonstrated shortly.

One possible way to visualize our MPPA is to regard the execution engine as an event-driven simulation kernel and the specification of an algorithm as HDL. The execution model described in Section 4.3.1 is, essentially, an extension of HDL. The execution engine executes the specification in a similar way as an event-driven simulation kernel.

The execution engine interacts with modules running on other μCPUs through messages. Table 11 summarizes the various messages. Note that this table only shows the portion of the supported message set that is needed to understand the rest of this chapter.

For example, any module can instantiate another module by sending a request message `REQ_INST_MODULE` to the scheduler. Recall that the execution engine consists of a scheduler, signal storage, and interconnect directory. After the scheduler instantiates a new module, it sends a response message `RES_INST_MODULE` to the requester. Similarly, a signal can be instantiated by exchanging `REQ_INST_SIGNAL` and `RES_INST_SIGNAL` with the signal storage. A module is allowed to change its own or other modules' sensitivity list and prefetch list by sending corresponding messages. The remaining messages will be explained in the following subsections.

Table 11: Message types supported by the proposed MPPA architecture

Category	Type	From	To	Payload
Instantiation	REQ_INST_MODULE	Module	Scheduler	Arguments for the constructor
	RES_INST_MODULE	Scheduler	Module	Module instance ID
	REQ_INST_SIGNAL	Module	Signal storage	None
	RES_INST_SIGNAL	Signal storage	Module	Signal instance ID
Reconfiguration	ADD_SENSITIVITY	Module	Interconnect directory	Module instance ID, port ID
	REMOVE_SENSITIVITY	Module	Interconnect directory	Module instance ID, port ID
	ADD_PREFETCH	Module	Scheduler	Module instance ID, port ID
	REMOVE_PREFETCH	Module	Scheduler	Module instance ID, port ID
Prefetching	REQ_FETCH_MODULE	Prefetcher	Scheduler	None
	RES_FETCH_MODULE	Scheduler	Prefetcher	List of input ports to be prefetched
	MODULE_INSTANCE	Scheduler	Prefetcher	Module instance
	REQ_SIGNAL	Prefetcher	Interconnect directory	Port ID, destination node
	RES_SIGNAL	Signal storage or node	Prefetcher	Signal data
Execution	NOTIFY_SIGNAL_UPDATE	Module	Interconnect directory	Signal instance ID
	TRIGGER_MODULE	Interconnect directory	Scheduler	List of modules

The scheduler keeps track of the state of modules and their location. The states of a module can be `wait`, `ready`, and `run`. There are three queues, and modules are stored in a corresponding queue according to their state. Initially, the state of a module is `wait`. When a signal connected to the port in the sensitivity list changes, the module is triggered and its state is changed to `ready`. Once the module is fetched by a node, its state becomes `run` until it finishes. Unless another signal triggers this module again, its state returns to wait. In addition, the scheduler stores instances of modules in the device memory. When a module is fetched by a node, it reads its instance from the memory and sends it to the node.

The signal storage stores values of signals in the device memory. Sometimes, the latest value resides in the output queue of a node. When an output of a module is updated, its new value is stored in the output queue of that node. The signal storage and the scheduler are notified of the fact that the output has been updated. The signal storage invalidates its copy and keeps track of the signal's location. The scheduler triggers the modules (i.e., it moves modules from the wait queue to the ready queue) whose sensitivity lists include that signal.

The interconnect directory keeps track of the connectivity of signals and ports. A module accesses its input and output through ports. It is unaware of which signal is connected to its ports. To access a signal, the module sends a request to the interconnect directory in

order to find which signal is connected to the port. Then, the interconnect directory forwards the request to the signal storage, and the signal storage responds to the module. The interconnect directory also keeps track of the sensitivity list. If a signal is updated, the list of its associated modules is sent to the scheduler.

4.5.2 Module-Level Prefetching

Dynamic scheduling incurs run-time overhead. The preferching mechanism is employed to hide the overhead, as well as the memory access latency. Hiding memory access latency is not demonstrated in detail in this thesis.

The execution model enables accurate prefetching by forcing a module to only access the code within its boundaries and to only access its explicitly associated inputs and outputs.

Figure 44 shows a sequence diagram of the prefetching process. While a module is executed within a μCPU, the prefetcher prefetches *instructions* and *data* for the next module. The fetching of *instructions* involves the scheduling process within the scheduler and memory accesses to the device memory. The fetching of *data* involves accessing of the signal storage and memory accesses to the device memory. Therefore, prefetching hides both the overhead of the execution engines and the access latency to the device memory.

As soon as a module starts running on a μCPU, the prefetcher starts prefetching the next module to run. It gets a module instance ID to run by exchanging `REQ_FETCH_MODULE` and `RES_FETCH_MODULE` messages with the scheduler. If the module to run is not the same module currently running, the scheduler provides the prefetcher with the code of a module via a `MODULE_INSTANCE` message, after reading it from the device memory. Otherwise, the prefetcher keeps the current module and fetches only input signals. The prefetcher may get `none`, which indicates that no module is ready to run. Subsequently, the node goes into a *sleep mode* as soon as the current module finishes, unless it receives a module to run from the scheduler by another `RES_FETCH_MODULE`.

`RES_FETCH_MODULE` contains the list of input ports to be prefetched for the module. The prefetcher sends request messages `REQ_SIGNAL` to the interconnect directory. The

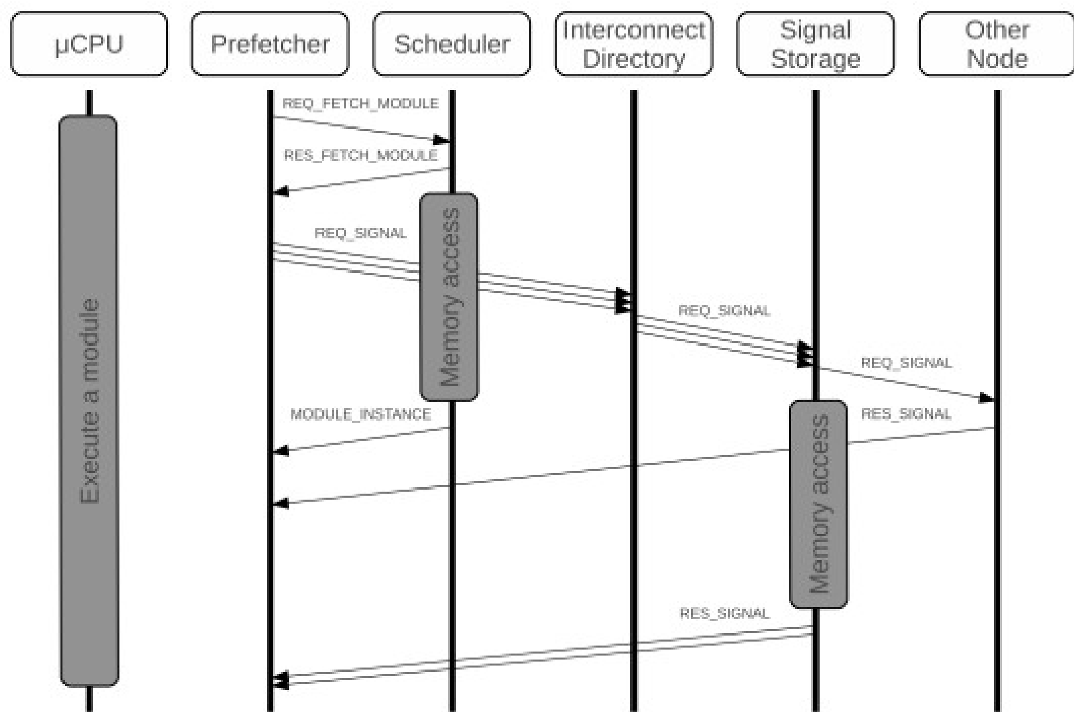


Figure 44: Sequence diagram of the prefetching process of the proposed MPPA architecture. Notice how prefetching can hide both the overhead of the execution engine and the access latency to the device memory.

interconnect directory fills the signal ID field of the message by looking up its port-to-signal mapping table and forwards the message to the signal storage. The signal storage returns the signals through a RES_SIGNAL message. If other nodes hold the requested signals, the request messages are forwarded to them.

If the execution of the module takes longer than prefetching, the latter can hide the memory access latency, as well as the scheduling overhead. If there is only one task per μCPU, instructions do not need to be fetched again. Inputs need to be fetched only when they are changed, just as the semantics of the programming model dictates. Even though a cache may be used, this cache latency cannot be hidden. If an input is changed, the corresponding cache line would be invalidated by the coherence protocol. The cache line would then be re-fetched. The benefit of the proposed method over a cache is scalability. To the best of our knowledge, there is no cache coherence protocol that can scale efficiently to more than one hundred cores. In our proposed method, the programming model eliminates the need for a coherence protocol.

4.5.3 An Event-Driven Execution Example

Figure 45 illustrates how the proposed MPPA executes an event-driven model of our quick-sort algorithm example. The figure shows three core tiles and the execution engine. There are six instances of the partition module (P0-P5). P0, P1, and P2 are running on the μCPU and P3, P4, and P5 are prefetched and waiting for execution. One instance of the collection module is in the wait queue (COL).

Suppose that P0 generates an output. The output is stored in the output queue and the fact that the output signal has been updated is signified via NOTIFY_SIGNAL_UPDATE (1). The output is actually written to a port. Which signal is connected to that port is determined at run-time and managed by the interconnect directory. NOTIFY_SIGNAL_UPDATE is sent to the interconnect directory, which looks up the connected signal, augments the signal ID, and forwards the message to the signal storage (2). NOTIFY_SIGNAL_UPDATE indicates only that the signal is updated and the actual data is still stored in the output queue. The

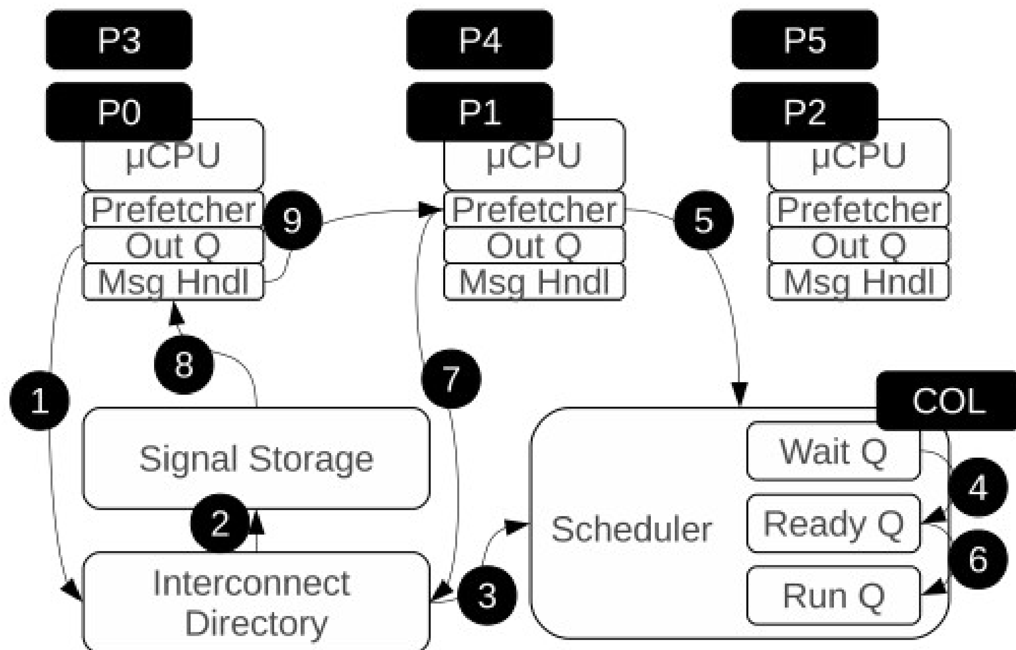


Figure 45: Illustrative example of an event-driven execution of the quicksort algorithm.

signal storage changes the location of the signal to point to the first core tile. The interconnect directory also keeps track of the sensitivity list. It looks up which module should be triggered by the updated signal and sends `TRIGGER_MODULE` to the scheduler (3). The scheduler moves the module (in this example, `COL`) from the wait queue to the ready queue.

Right after this, suppose that `P1` finishes. Since `P4` has been prefetched, the second core tile immediately switches to run `P4`. At the same time, the prefetcher starts prefetching the next module. It requests the next module from the scheduler by sending `REQ_FETCH_MODULE` (5). The scheduler looks up the ready queue to check if there is any available module. In this example, `COL` is in the ready queue. The scheduler moves `COL` to the run queue (6) and sends the module and the list of its associated input ports to the requester via `REQ_FETCH_MODULE`. Then, the prefetcher starts prefetching the input signals. It sends `REQ_SIGNAL` to the interconnect directory, where its connected signals can be looked up (7). The interconnect directory augments the signal ID and forwards the message to the signal storage. The signal storage looks up the entry of the signal and finds that its data is

stored in the first core tile. The signal storage forwards REQ_SIGNAL to the first core tile (8). The message handler in the first core tile sends the data to the second core tile via RES_SIGNAL (9). Although it is not shown in this example, the output queue of the second core tile flushes the signals associated with P1 while the prefetcher is working. As long as the prefetching process explained in this paragraph finishes before P4 finishes, the second core tile can continue to work on COL without any delay, as soon as P4 finishes.

4.6 Experimental Evaluation

To evaluate the proposed MPPA architecture, we employ a detailed, cycle-level simulator to model the entire MPPA and associated devices. Table 12 summarizes the simulated architectural parameters. In terms of benchmark applications, we use the task-level parallel benchmarks of the recognition, mining and synthesis (RMS) benchmark suite [11]. Specifically, the applications are Forward Solve (FS), Backward Solve (BS), Cholesky Factorization (CF), Canny Edge Detection (CED), and Binomial Tree (BT). In addition, we use Octree Partitioning (OP) [19] and Quick Sort (QS) [32]. Table 13 shows the module execution times for all simulated benchmarks. Note that these execution times do not account for the memory access latency. If the prefetching finishes in time, the memory access latency can be hidden, as previously explained.

The chosen applications have abundant parallelism, which makes them suitable for MPPA. However, they exhibit heavy *dependencies among tasks*, which are not efficiently supported by existing MPPAs, like GPUs (as used for GPGPU), because the latter adopt a

Table 12: Simulated system parameters

Parameter	Value
Number of Core Tiles	32
Memory access time	1 cycle for scratch-pad memory 100 cycles for device memories
Memory size	8 KB scratch-pad memory per core 32 MB device memory
Communication delay	4 cycles per hop

Table 13: Module execution times for the benchmark applications used

Benchmark	Execution time in cycles		
	Min	Max	Average
Forward Solve (FS)	26	646	336.00
Backward Solve (BS)	42	569	305.50
Cholesky Factorization (CF)	151	11800	789.35
Canny Edge Detection (CED)	330	5011	669.68
Binomial Tree (BT)	117	4506	462.71
Octree Partitioning (OP)	1441	6679	2678.70
Quick Sort (QS)	88	47027	683.70

SIMD-based programming model whose efficiency is maximized only when *all cores run the same code*. Moreover, all chosen applications are dominated by *short tasks*, whereby the execution time of each task is very short and the overhead of dynamic scheduling becomes quite significant [11]. This attribute will help us evaluate the efficiency of our proposed prefetching mechanism. Since the chosen benchmarks are dominated by short tasks, their memory requirement is at most 4 KB. Hence, an 8 KB scratch-pad memory is enough for double-buffering purposes. In the case of the execution engine, the full size of the scratch-pad memory can be utilized as a cache.

The applications are efficiently implemented in the proposed MPPA, because its execution model allows dynamic instantiation of modules and run-time reconfiguration of their interconnections. More importantly, the prefetching mechanism hides the run-time overhead of dynamic task scheduling, as will be demonstrated shortly.

Figure 46 shows the average access time of the scheduler (denoted by the line graph and the right y-axis), normalized to the average execution time of the modules. A normalized access time below 1 indicates that the access time is completely hidden by the prefetching mechanism, because the accessing of the scheduler is complete before the module execution finishes. Since the memory requirements of the chosen benchmarks are not particularly high, most device memory access time is hidden by the execution engine's cache in this experiment. Remember, the scratch-pad memory in the *execution engine* core tile is used as a cache for the device memory. The access time of the scheduler shown in Figure 46 is, in

fact, dominated by the queueing latency.

By juxtaposing Table 13 and Figure 46, we can observe that the shorter the average execution time of an application is, the longer the scheduler access time becomes. Shorter execution times lead to more frequent accesses to the scheduler, which result in longer queuing delays [93]. This is the reason why short-task dominated benchmarks suffer from excessive overhead incurred by dynamic scheduling.

However, prefetching may be used to alleviate the issue. Prefetching hides the dynamic scheduling overhead by fetching modules *simultaneously* with the execution of other modules. As a result, the prefetching mechanism improves the utilization of core tiles, as demonstrated by the dark-colored bars in Figure 46 (the bars refer to the left y-axis). Since the average execution times of FS, BS, and BT are very short, even prefetching cannot hide the entire scheduler access time. However, prefetching still improves the utilization substantially.

The execution engine may need to be split up to support a growing number of core tiles. The optimal number of execution engines is dependent on the characteristics of the applications. Determining such an optimal number is left as future work. To evaluate the impact of the number of execution engines, we performed an experiment whereby the execution engine is split into a *distinct* scheduler, signal storage, and interconnect directory, with each component assigned to a separate core tile. Thus, *three* tiles are dedicated to the execution engine in total, while the others are used solely for compute purposes. Figure 47 compares this setup to the conventional case, where only one core tile is devoted to the execution engine. Although dedicating three core tiles to the execution engine always exhibits better *utilization* than dedicating one tile, the latter offers better *performance* up to 48 cores, because there are more *processing* tiles (as opposed to execution-engine tiles). This is the reason why the experiments of Figure 46 assume a single-tile execution engine. When the number of core tiles becomes larger than 48, the higher utilization of the system with a three-tile execution engine starts to compensate for the smaller number of working

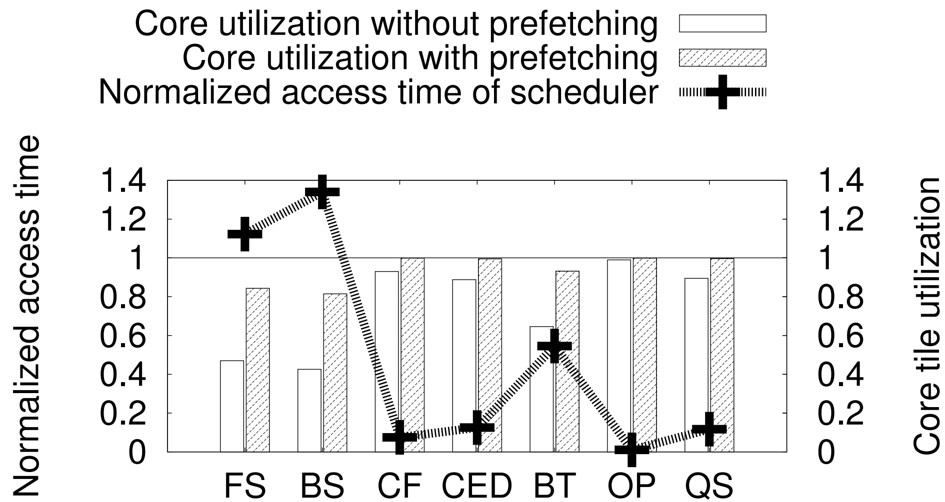


Figure 46: Average access times of the scheduler (normalized to the average execution time of the modules), and average utilization of the processing elements (i.e., the core tiles).

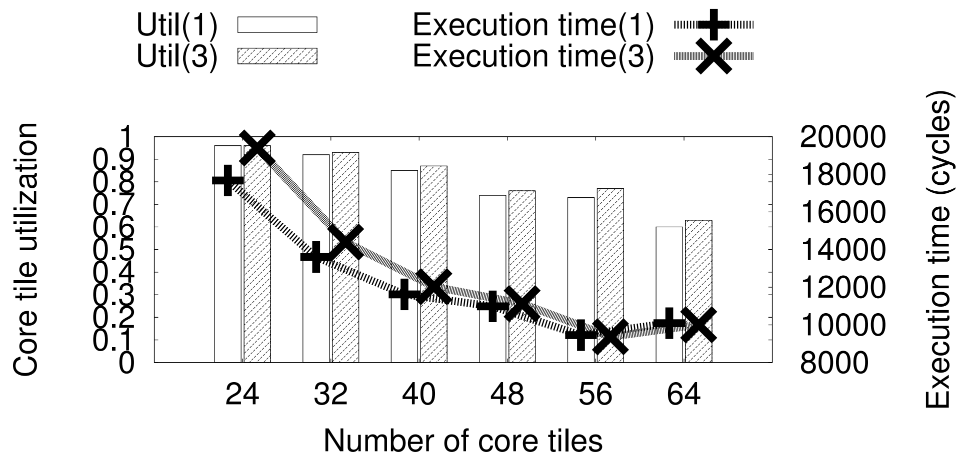


Figure 47: The impact on performance of the number of core tiles designated to serve as part of the *execution engine*. "Util(*k*)" and "Execution time(*k*)" denote the tile utilization and the total execution time, respectively, when the number of core tiles devoted to the execution engine is *k*. The benchmark used is CED.

core tiles. Of course, dedicating three tiles to the execution engine of a 56-tile system only yields a 0.89% improvement in execution time. However, this is expected to increase as the number of core tiles increases. Overall, this experiment demonstrates that the proposed MPPA architecture scales well up to 56 cores, even for an application with heavy data dependencies, such as CED. This is because the overhead of dynamic scheduling is hidden by the prefetching mechanism. As a point of reference, when the entire CED benchmark is executed on a *single core* (i.e., serially), its execution time is 395,791 cycles.

The experiments of this section demonstrate the improvements obtained through the use of prefetching. It is demonstrated that the proposed MPPA scales well up to 56 cores, even for applications dominated by short tasks, where the overhead of the dynamic scheduling could be excessive. Further studies on the splitting of the execution engines would enable even larger-scale MPPAs with tens, or hundreds, of cores.

4.7 Conclusion

The last few years have witnessed the emergence of the powerful computational paradigm of MPPA, employed as general-purpose hardware accelerators. GPUs constitute a prime example of this concept, as manifested by the increasing popularity of GPGPU. However, the widespread adoption of MPPAs as general-purpose hardware accelerators faces three fundamental challenges: the expressiveness of the programming model, the debugging capabilities, and the memory hierarchy.

This thesis proposes an MPPA hardware architecture that effectively addresses these issues through the intelligent interplay between the *execution model* and the *hardware architecture*. The presented design employs an *event-driven* execution model that facilitates efficient debugging. Our execution model offers better expressiveness than existing GPGPU practices by allowing hardware-supported run-time reconfigurability and *dynamic task scheduling*, which greatly improves the utilization of processing elements. The execution model also ensures *encapsulation* of the modules. All the accesses to data and function

calls are limited within the module and no global shared memory is assumed. Encapsulation facilitates debugging by limiting possible causes of erroneous behavior, while the absence of a shared memory *eliminates the need for a cache coherence protocol*. Finally, the explicit declaration of all input signals enables accurate *module-level prefetching*, which is demonstrated - through simulation experiments - to hide the access latency to both the device memory and the execution engine's scheduler.

CHAPTER 5
CONCLUSION

Dwindling technology feature sizes have helped materialize the billion-transistor microprocessor. Unprecedented integration densities have enabled the transition to the CMP paradigm. As the number of processing cores increase to the many-core realm, many new challenges arise that must be addressed. In this thesis three important challenges are studied.

IsoNet is proposed as a hardware-based load-balancing engine. It employs a lightweight micro-network working indepedently from the existing infrastructure. In addition, comprehensive fault-tolerance support is provided. Compared with the state-of-the-art hardware-based load-balancing technique, it improves the system performance by up to 70% (36% on average) with 128 to 1024 cores. It is fully implemented in 45 nm standard CMOS technology. Subsequent analysis confirms that IsoNet incurs negligible overhead.

On-chip communication architecture is also becoming very important because the many-core systems tend to be communication-centric. One of key design parameters of the on-chip router is its flit size. Our preliminary study on the flit size indicates that a wide flit is not an efficent choice. Instead, a physically separated network offers better efficiency. However, due to discrepancy between the flit size and the packet size, the network is not fully utilized. To address this problem, Sharded Router is proposed in this thesis. It reduces the average execution time of PARSEC benchmarks by up to 43% at 10% hardware area overhead.

Finally, the event-driven execution model is proposed in this thesis. The execution model facilitates debugging by enforcing encapsulation while offering better expressiveness than popular parallel programming models such as OpenMP and CUDA. From the hardware perspective, it eliminates the need for costly cache coherence protocol. The event-driven execution model is implemented in collaboration of hardware peripherals and

the execution engine running on one of MPPA tiles. The execution engine implements dynamic scheduling of modules. The dynamic scheduling offers flexibility in scheduling modules. The overhead involved in dynamic scheduling is addressed by module-level prefetching. They contribute to high utilization of MPPA cores.

The event-driven execution model is at its early stage. The MPPA architecture has many opportunities exploiting the execution model. The limited memory bandwidth is one of key challenges that many-core architectures should solve. The event-driven execution model can enable locality-aware scheduling that exploits existing data in the on-chip memory. These are left as our future work.

REFERENCES

[1] Intel, “The single-chip cloud computer.” http://techresearch.intel.com.

[2] nVIDIA, “Product specification of the GeForce GTX 570.” http://www.nvidia.com/object/product-geforce-gtx-570-us.html.

[3] “International technology roadmap for semiconductors 2009.” http://public.itrs.net.

[4] “International technology roadmap for semiconductors 2011.” http://public.itrs.net.

[5] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[6] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, “Multifacet’s general execution-driven multiprocessor simulator (gems) toolset,” *SIGARCH Computer Architecture News*, vol. 33, p. 2005, 2005.

[7] Intel, “Product specification of Intel Core i5-2540M Processor,” 2011. http://www.intel.com/SandyBridge.

[8] N. Brookwood, “AMD Fusion family of APUs: enabling a superior, immersive, PC experience,” 2010.

[9] M. Butts, “Synchronization through communication in a massively parallel processor array,” *IEEE Micro*, vol. 27, pp. 32-40, 2007.

[10] D. Cederman and P. Tsigas, “On dynamic load balancing on graphic processors,” in *Proceedings of the ACM Symposium on Graphics Hardware*, pp. 57-64, 2008.

[11] S. Kumar, C. Hughes, and A. Nguyen, “Carbon: Architectural support for fine-grained parallelism on chip multiprocessors,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pp. 162-173, 2007.

[12] J. Giacomoni, T. Moseley, and M. Vachharajani, “FastForward for efficient pipeline parallelism a cache-optimized concurrent lock-free queue,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 43-52, 2008.

[13] W. N. S. III, D. Lea, and M. L. Scott, “Scalable synchronous queues,” *Communications of ACM*, vol. 52, no. 5, pp. 100-111, 2009.

[14] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, “Using elimination to implement scalable and lock-free FIFO queues,” in *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 253-262, 2005.

[15] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 311-322, 2010.

[16] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10-16, 2005.

[17] P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Technology@Intel Magazine*, 2005.

[18] B. Saha and et al., "Enabling scalability and performance in a large scale CMP environment," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 73-86, 2007.

[19] L. Soares, C. Menier, B. Raffin, and J. L. Roch, "Work stealing for time-constrained octree exploration: Application to real-time 3d modeling," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, 2007.

[20] E. Berger and J. C. Browne, "Scalable load distribution and load balancing for dynamic parallel programs," in *Proceedings of the International Workshop on Cluster-Based Computing*, 1999.

[21] J. Agron and et al., "Run-time services for hybrid CPU/FPGA systems on chip," in *Proceedings of the IEEE International Real-Time Systems Symposium*, pp. 3-12, 2006.

[22] P. Kuacharoen, M. A. Shalan, and V. J. M. III, "A configurable hardware scheduler for real-time systems," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2003.

[23] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 140-151, 2009.

[24] C. J. Myers, *Asynchronous Circuit Design*. Wiley, 2001.

[25] W. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of Applied Physics*, pp. 55-63, 1948.

[26] S. K. Lim, *Practical problems in VLSI physical design automation*. Springer, 2008.

[27] TC320. http://www.semicon.toshiba.co.jp.

[28] U. Bhattacharya, Y. Wang, F. Hamzaoglu, Y. Ng, L. Wei, Z. Chen, J. Rohlman, I. Young, and K. Zhang, "45nm SRAM technology development and technology lead vehicle," *Intel Technology Journal*, 2008.

[29] ARM, "Cortex-A5 specification." http://www.arm.com/.

[30] S. Wilton and N. Jouppi, "An enhanced access and cycle time model for on-chip caches," 1994. Technical Report 93/5, WRL.

[31] Wind River Systems. http://www.windriver.com/.

[32] C. A. R. Hoare, "Algorithm 64: Quicksort," *Communication ACM*, vol. 4, no. 7, p. 321, 1961.

[33] A. Chien, "Keynote: NoC's at the center of chip architecture," 2009. International Symposium on Networks-on-Chip.

[34] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, "Express virtual channels: towards the ideal interconnection fabric," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[35] A. Kumary, P. Kunduz, A. Singhx, L.-S. Pehy, and N. Jhay, "A 4.6Tbits/s 3.6GHz single-cycle NoC router with a novel switch allocator in 65nm CMOS," in *Proceedings of the 25th International Conference on Computer Design*, pp. 63 -70, 2007.

[36] M. Hayenga, N. E. Jerger, and M. Lipasti, "SCARAB: a single cycle adaptive routing and bufferless network," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[37] R. Das, S. Eachempati, A. Mishra, V. Narayanan, and C. Das, "Design and evaluation of a hierarchical on-chip interconnect for next-generation CMPs," in *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pp. 175 -186, 2009.

[38] C. Fallin, X. Yu, G. Nazario, and O. Mutlu, "A high-performance hierarchical ring on-chip interconnect with low-cost routers," tech. rep., Computer Architecture Lab (CALCM), Carnegie Mellon University, 2011.

[39] N. Novakovic, "New-generation GPU memory bandwidth increases: more for compute than for graphics?," 2011.

[40] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart, "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173 -183, 2011.

[41] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15 -31, 2007.

[42] C.-L. Chou and R. Marculescu, "User-centric design space exploration for heterogeneous network-on-chip platforms," in *Proceedings of Design, Automation Test in Europe Conference Exhibition*, pp. 15 -20, april 2009.

[43] R. K. Jena, M. Aqel, and P. Mahanti, "Network-on-chip design space exploration: A PSO based integrated approach," *European Journal of Scientific Research*, vol. 64, no. 1, pp. 5 -18, 2011.

[44] L. Indrusiak, L. Ost, L. Moller, F. Moraes, and M. Glesner, "Applying UML interactions and actor-oriented simulationto the design space exploration of network-on-chip interconnects," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pp. 491 -494, april 2008.

[45] D. Matos, G. Palermo, V. Zaccaria, C. Reinbrecht, A. Susin, C. Silvano, and L. Carro, "Floorplanning-aware design space exploration for application-specific hierarchical networks on-chip," in *Proceedings of the 4th International Workshop on Network on Chip Architectures*, pp. 31-36, 2011.

[46] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, pp. 23:1-23:20, 2008.

[47] J. Kim, "Low-cost router microarchitecture for on-chip networks," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 255 -266, 2009.

[48] M. Holliday and M. Stumm, "Performance evaluation of hierarchical ring-based shared memory multiprocessors," *IEEE Transactions on Computers*, vol. 43, pp. 52-67, 1994.

[49] F. Sibai, "Adapting the hyper-ring interconnect for many-core processors," in *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications*, pp. 649 -654, 2008.

[50] J.-H. Chuang and W.-C. Chao, "Torus with slotted rings architecture for a cache-coherent multiprocessor," in *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, pp. 76-81, 1994.

[51] S. Bourduas and Z. Zilic, "A hybrid ring/mesh interconnect for network-on-chip using hierarchical rings for global routing," in *Proceedings of the First International Symposium on Networks-on-Chip*, pp. 195-204, 2007.

[52] P. Abad, V. Puente, and J.-A. Gregorio, "MRR: Enabling fully adaptive multicast routing for CMP interconnection networks," in *Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, pp. 355 -366, 2009.

[53] P. Abad, V. Puente, J. A. Gregorio, and P. Prieto, "Rotary router: an efficient architecture for cmp interconnection networks," in *Proceedings of the 34th International Symposium on Computer Architecture*, pp. 116-125, 2007.

[54] S. Volos, C. Seiculescu, B. Grot, N. Khosro Pour, B. Falsafi, and D. M. G., "CCNoC: Specializing on-chip interconnects for energy efficiency in cache-coherent servers," in *Proceedings of the 6th ACM/IEEE International Symposium on Networks-on-Chip*, 2012.

[55] A. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A power-area simulator for interconnection networks," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 20, no. 1, pp. 191 -196, 2012.

[56] D. Park, S. Eachempati, R. Das, A. Mishra, Y. Xie, N. Vijaykrishnan, and C. Das, "MIRA: A multi-layered on-chip interconnect router architecture," in *Proceedings of the 35th International Symposium on Computer Architecture*, pp. 251 -261, 2008.

[57] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection networks*. Margan Kaufmann, 2003.

[58] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti, "Circuit-switched coherence," in *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pp. 193-202, 2008.

[59] G. Michelogiannakis, D. Pnevmatikatos, and M. Katevenis, "Approaching ideal NoC latency with pre-configured routes," in *Proceedings of the First International Symposium on Networks-on-Chip*, 2007.

[60] H. Matsutani, M. Koibuchi, H. Amano, and T. Yoshinaga, "Prediction router: Yet another low latency on-chip router architecture," in *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 367 -378, 2009.

[61] J. Kim, C. Nicopoulos, and D. Park, "A gracefully degrading and energy-efficient modular router architecture for on-chip networks," *SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 4-15, 2006.

[62] B. Grot, J. Hestness, S. Keckler, and O. Mutlu, "Express cube topologies for on-chip interconnects," in *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pp. 163 -174, 2009.

[63] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 196 -207, 2009.

[64] A. Flores, J. Aragon, and M. Acacio, "Heterogeneous interconnects for energy-efficient message management in cmps," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 16 -28, 2010.

[65] Y. Chen, L. Xie, J. Li, and Z. Lu, "Slice router: For fine-granularity fault-tolerant networks-on-chip," in *Proceedings of the International Conference on Multimedia Technology*, pp. 3230 -3233, 2011.

[66] Y. J. Yoon, N. Concer, M. Petracca, and L. Carloni, "Virtual channels vs. multiple physical networks: A comparative analysis," in *Proceedings of the 47th ACM/IEEE Design Automation Conference*, pp. 162 -165, 2010.

[67] J. Balfour and W. J. Dally, "Design tradeoffs for tiled CMP on-chip networks," in *Proceedings of the 20th Annual International Conference on Supercomputing*, pp. 187-198, 2006.

[68] P. Kumar, Y. Pan, J. Kim, G. Memik, and A. Choudhary, "Exploring concentration and channel slicing in on-chip network router," in *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pp. 276 -285, 2009.

[69] A. Leroy, D. Milojevic, D. Verkest, F. Robert, and F. Catthoor, "Concepts and implementation of spatial division multiplexing for guaranteed throughput in networks-on-chip," *IEEE Transactions on Computers*, vol. 57, no. 9, pp. 1182 -1195, 2008.

[70] P. Marchal, D. Verkest, A. Shickova, F. Catthoor, F. Robert, and A. Leroy, "Spatial division multiplexing: a novel approach for guaranteed throughput on NoCs," in *Proceedings of the Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp. 81 -86, 2005.

[71] Z. Yang, A. Kumar, and Y. Ha, "An area-efficient dynamically reconfigurable spatial division multiplexing network-on-chip with static throughput guarantee," in *Proceedings of the International Conference on Field-Programmable Technology*, pp. 389 -392, 2010.

[72] A. Morgenshtein, A. Kolodny, and R. Ginosar, "Link division multiplexing (LDM) for network-on-chip links," in *Proceedings of the IEEE 24th Convention of Electrical and Electronics Engineers in Israel*, pp. 245 -249, 2006.

[73] P. Wolkotte, G. Smit, G. Rauwerda, and L. Smit, "An energy-efficient reconfigurable circuit-switched network-on-chip," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[74] W. Song and D. Edwards, "Building asynchronous routers with independent sub-channels," in *Proceedings of the International Symposium on System-on-Chip*, pp. 048 -051, 2009.

[75] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[76] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 33-42, 2009.

[77] R. Hesse, J. Nicholls, and N. Jerger, "Fine-grained bandwidth adaptivity in networks-on-chip using bidirectional channels," in *Proceedings of the Sixth IEEE/ACM International Symposium on Networks on Chip*, pp. 132 -141, 2012.

[78] R. Das, O. Mutlu, T. Moscibroda, and C. Das, "Application-aware prioritization mechanisms for on-chip networks," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 280 -291, 2009.

[79] N. E. Jerger, D. Vantrease, and M. Lipasti, "An evaluation of server consolidation workloads for multi-core designs," in *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, pp. 47-56, 2007.

[80] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, "Achieving predictable performance through better memory controller placement in many-core CMPs," in *Proceedings of the 36th International Symposium on Computer Architecture*, pp. 451-461, 2009.

[81] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121 -133, 1999.

[82] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer cache assisted prefetching," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 62-73, 2002.

[83] B. Grot, S. W. Keckler, and O. Mutlu, "Preemptive virtual clock: a flexible, efficient, and cost-effective QOS scheme for networks-on-chip," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 268-279, 2009.

[84] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 140-151, ACM, 2009.

[85] M. Sima, M. McGuire, and J. Lamoureux, "Coarse-grain reconfigurable architectures - taxonomy -," in *Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 975-978, 2009.

[86] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, pp. 589-604, 2005.

[87] nVIDIA, *CUDA Programming Guide*, 2008.

[88] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *IEEE Computer*, pp. 54-62, 2003.

[89] Tilera, "TILE-Gx processor family overview," 2010.

[90] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *Proceedings of the 15th Annual IEEE Symposium on Field-ProgrammableCustom Computing Machines*, pp. 55-64, 2007.

[91] E. de Souza Carvalho, N. Calazans, and F. Moraes, "Dynamic task mapping for MP-SoCs," *IEEE Design Test of Computers*, vol. 27, no. 5, pp. 26 -35, 2010.

[92] A. Shabbir, A. Kumar, B. Mesman, and H. Corporaal, "Distributed resource management for concurrent execution of multimedia applications on MPSoC platforms," in *Proceedings of the International Symposium on Systems, Architectures, MOdeling and Simulation*, 2011.

[93] J. Lee, C. Nicopoulos, Y. Lee, H. G. Lee, and J. Kim, "Hardware-based job queue management for manycore architectures and OpenMP environments," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2011.

[94] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU applications," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 213 -224, 2010.

[95] W. Thies, M. Karczmarek, M. I. Gordon, D. Z. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe, "Streamit: A compiler for streaming applications," Technical Report MIT/LCS Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, MA, 2001.

[96] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek, "A retargetable parallel-programming framework for MPSoC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, pp. 39:1-39:18, 2008.

[97] D. Cederman and P. Tsigas, "GPU-Quicksort: A practical quicksort algorithm for graphics processors," *J. Exp. Algorithmics*, vol. 14, pp. 4:1.4-4:1.24, 2010.