Architectural Enhancement for Message Passing Interconnects

by

Farshad Khun Jush

B.Sc., Shiraz University, Iran, 1991
M.Sc., Shiraz University, Iran, 1995

A Dissertation Submitted in Partial Fullfillment of the
Requirements for the Degree of

Doctor of Philosophy

in the Department of Electrical and Computer Engineering

©Farshad Khun Jush, 2008
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Architectural Enhancement for Message Passing Interconnects

by

Farshad Khun Jush

B.Sc., Shiraz University, Iran, 1991
M.Sc., Shiraz University, Iran, 1995

Supervisory Committee

Dr. Nikitas J. Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Kin F. Li, Department Member
(Department of Electrical and Computer Engineering)

Dr. Amirali Baniasadi, Department Member
(Department of Electrical and Computer Engineering)

Dr. D. Michael Miller, Outside Member
(Department of Computer Science)

Supervisory Committee

Dr. Nikitas J. Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Kin F. Li, Department Member
(Department of Electrical and Computer Engineering)

Dr. Amirali Baniasadi, Department Member
(Department of Electrical and Computer Engineering)

Dr. D. Michael Miller, Outside Member
(Department of Computer Science)

Abstract

Research in high-performance architecture has been focusing on achieving more computing power to solve computationally-intensive problems. Advancements in the processor industry are not applicable in applications that need several hundred or thousand-fold improvement in performance. The parallel architecture approach promises to provide more computing power and scalability. Cluster computing, consisting of low-cost and high-performance processors, has been an alternative to proprietary and expensive supercomputer platforms. As in any other parallel system, communication overhead (including hardware, software, and network) adversely affects the computation performance in a cluster environment. Therefore, decreasing this overhead is the main concern in such environments.

Communication overhead is the key obstacle to reaching hardware performance limits and is mostly associated with software overhead, a significant portion of which is attributed to message copying. Message copying is largely caused by a lack of knowledge of the next received message, which can be dealt with through speculation. To reduce this copying overhead and advance toward a finer granularity, architectural extensions comprised of a specialized *network cache* and instructions to manage the operations of these extensions were introduced. In order to investigate the effectiveness of the proposed architectural enhancement, a simulation environment was established by expanding an existing single-thread infrastructure to one that can run MPI applications. Then the proposed extensions were implemented, along with the MPI functions on top of the SimpleScalar infrastructure.

Further, two techniques were proposed in order to achieve zero-copy data transfer in message passing environments, two policies that determine when a message is to be bound and sent to the data cache. These policies are called Direct to Cache Transfer DTCT and lazy DTCT. The simulations showed that by using the proposed network extension along with the DTCT techniques fewer data cache misses were encountered as compared to when the DTCT techniques were not used. This involved a study of the possible overhead and cache pollution introduced by the operating system and the communications stack, as exemplified by Linux, TCP/IP and M-VIA. Then these effects on the proposed extensions were explored. Ultimately, this enabled a comparison of the performance achieved by applications running on a system incorporating the proposed extension with the performance of the same applications running on a standard system. The results showed that the proposed approach could improve the performance of MPI applications by 15 to 20%.

Moreover, data transfer mechanisms and the associated components in the CELL BE processor were studied. For this, two general data transfer methods were explored involving the PUT and GET functions, demonstrating that the SPE-initiated DMA

data transfers are faster than the corresponding PPE-initiated DMAs. The main components of each data transfer were also investigated. In the SPE-initiated GET function, the main component is data delivery. However, the PPE-initiated GET function shows a long DMA issue time as well as a lengthy gap in receiving successive messages. It was demonstrated that the main components of the SPE-initiated PUT function are data delivery and latency (that is, the time to receive the first byte), and the main components in the PPE-initiated PUT function are the DMA issue time and latency. Further, an investigation revealed that memory-management overhead is comparable to the data transfer time; therefore, this calls for techniques to hide the unavoidable overhead in order to reach high-throughput communication in MPI implementation in the Cell BE processor.

Table of Contents

Supervisory Committee ii

Abstract iii

Table of Contents vi

List of Tables x

List of Figures xii

List of Abbreviations xvi

Acknowledgment xviii

Dedication xx

1 Introduction 1

1.1 Parallel Computer Architectures 2

1.2 Dissertation Contributions 7

2 Background 10

2.1 I/O Data Transfer Mechanisms 11

2.1.1 Send and Receive Data Transfers 12

2.1.2 Memory Transactions in I/O Operations 14

2.2 Communication Subsystem Improvement Techniques 16

2.2.1 Massively Parallel Processors (MPPs) 16

2.2.2 Cluster Computing 18

3 Architectical Enhancement Techniques **30**
3.1 The Proposed Architectural Extension 31
3.1.1 Operation 33
3.1.2 Late Binding 34
3.1.3 Early Binding 36
3.1.4 Other Considerations 36
3.1.5 ISA Extensions 37
3.2 Network Cache Implementation 39
3.3 Data Transfer Policies 40
3.3.1 Direct-To-Cache Transfer Policy 40
3.3.2 Lazy Direct-To-Cache Transfer Policy 41
3.4 Summary 43

4 Experimental Methodology **44**
4.1 NAS Benchmarks 44
4.1.1 SP and BT Benchmarks 45
4.1.2 Conjugate Gradient (CG) 46
4.2 PSTSWM 46
4.3 QCDMPI 46
4.4 Simulation Methodology 47
4.4.1 Message Distribution and Data Collection 47
4.5 SimpleScalar Infrastructure 61
4.6 Implementation 62
4.6.1 Benchmark Programming Methodology 62
4.6.2 Time and Cycle Consistency on IBM RS/6000 SP and SimpleScalar 64
4.7 Summary 66

5 Characterizing the Caching Environment 67

5.1 Time and Cycle Consistency Using a Statistical Approach 68

5.2 Establishing the Message Traffic Intensity 72

5.2.1 The Impact of the Size of the Network Cache 76

5.3 Summary 78

6 Evaluating the Data Transfer Techniques 80

6.1 Wall-clock Time and Simulated-cycle Time Correspondence 82

6.2 Message Size Distributions 83

6.3 Optimum Data Cache Configuration 87

6.4 Data Cache Behavior 90

6.4.1 First Access Time Behavior 92

6.4.2 Last Access Time Behavior 94

6.5 Summary 97

7 Comparing Direct-to-Cache Transfer Policies to TCP/IP and M-VIA 99

7.1 TCP/IP and VIA Overview 100

7.1.1 TCP/IP Protocol 100

7.1.2 VIA Protocol 102

7.2 Experimental Methodology 103

7.2.1 Overhead Measurement during MPI_Receive Operation 103

7.2.2 Cache Behavior during and after MPI_Receive 106

7.3 Experimental Results 107

7.3.1 Message Access-Time Behavior 111

7.3.2 Speed Up 114

7.4 Summary 115

8 Hiding Data Delivery Latency on the Cell BE Processors 117

8.1 Motivation 118

8.2 The Cell Processor Architecture Overview 119

8.3 Methodology and Simulation Environment 121

8.3.1 First-Byte & Last-Byte Delivery Measurement 122

8.3.2 Synchronizing the PPE and the SPEs timers 123

8.3.3 Memory Management Overhead 125

8.3.4 Experimental Setup 127

8.4 Evaluation 127

8.4.1 DMA Transfer Time between the PPE and SPEs 128

8.4.2 DMA Latency among and inside the SPEs 135

8.4.3 Mailbox Communication 137

8.4.4 Address Translation Behavior 137

8.5 Summary 138

9 Conclusions and Future Work 141

9.1 Future Work 146

Bibliography 148

List of Tables

Table 4.1 Absolute and Relative Errors of results on IBM RS/6000 and on SimpleScalar . . . 65

Table 5.1 Simulator configuration . . . 71
Table 5.2 Comparison of different approaches . . . 72
Table 5.3 Dependence of the data cache response on the different transferring mechanism and traffic intensity . . . 75
Table 5.4 Network cache misses: messages arrive early (r= 24.62E07) . . . 77
Table 5.5 Network cache misses: (r= 24.639E07) . . . 77
Table 5.6 Network cache misses:(r= 24.89E07) . . . 78
Table 5.7 Network cache misses: messages are consumed immediately (r= 25.19E07) . . . 78

Table 6.1 Simulator configuration . . . 82
Table 6.2 Message size Distribution (Byte) in CG and PSTSWM Benchmarks for 64 Processors . . . 86

Table 7.1 Simulator configuration . . . 110

Table 8.1 Pseudo code for detecting first byte delivery on the PPE (PPE-initiated) . . . 124
Table 8.2 Pseudo code for detecting first byte delivery on the PPE (SPE-initiate) . . . 124
Table 8.3 Pseudo code for synchronizing the timers on the PPE and the SPEs . . . 125
Table 8.4 Pseudo code for the TLB miss penalty . . . 126

Table 8.5 Latency to transfer data to the same SPE using DMA and copy operations (μs) 135
Table 8.6 Mailbox communication behavior 137
Table 8.7 TLB miss and replacement policy (μs) 138
Table 8.8 Summary of data transfer latencies for blocking cases (μs) . . . 138

List of Figures

Figure 2.1 Copy operations in data transfer from a sender to a receiver . 12
Figure 2.2 A typical Network Interface Card Architecture 14

Figure 3.1 The *process* and *network* memory spaces and their relation through the network cache. 33
Figure 3.2 The network cache contents after the arrival of a message with Message ID equal to ID-N. 35
Figure 3.3 The network cache contents after the late binding. 35
Figure 3.4 The overall architecture of a network-extended node. 38
Figure 3.5 40

Figure 4.1 Message Size Distribution for CG Benchmark (Class W) . . . 50
Figure 4.2 Message Size Distribution for CG Benchmark (Class A) 51
Figure 4.3 Message Size Distribution for CG Benchmark (Class B) 52
Figure 4.4 Message Size Distribution for SP Benchmark (Class W) 53
Figure 4.5 Message Size Distribution for SP Benchmark (Class A) 54
Figure 4.6 Message Size Distribution for SP Benchmark (Class B) 55
Figure 4.7 Message Size Distribution for BT Benchmark (Class W) . . . 56
Figure 4.8 Message Size Distribution for BT Benchmark (Class A) 57
Figure 4.9 Message Size Distribution for BT Benchmark (Class B) 58
Figure 4.10 Message Size Distribution for QCDMPI Benchmark 59
Figure 4.11 Message Size Distribution for PSTSWM Benchmark 60
Figure 4.12 Process and Network Communication Method 62
Figure 4.13 63

Figure 5.1 Number of Cycles on SimpleScalar vs. time on IBM RS/6000 SP for the CG Benchmark (code was compiled with standard optimization) 69
Figure 5.2 Number of Cycles on SimpleScalar vs. time on IBM RS/6000 SP for the CG Benchmark (code was compiled under -O3 optimization) 69
Figure 5.3 Execution time vs Simulation-cycle/Wall-clock-unit-time *(DTCT)* CG Benchmark (Class A) 73

Figure 6.1 Execution time vs Simulation-cycle/Wall-clock-unit-time *(DTCT)* CG Benchmark (Class A) 84
Figure 6.2 Execution time vs Simulation-cycle/Wall-clock-unit-time *(Lazy DTCT)* CG Benchmark (Class A) 84
Figure 6.3 Execution time vs Simulation-cycle/Wall-clock-unit-time *(DTCT)* PSTSWM Benchmark 85
Figure 6.4 Execution time vs Simulation-cycle/Wall-clock-unit-time *(Lazy DTCT)* PSTSWM Benchmark 85
Figure 6.5 Data Cache Misses in DTCT Configuration in CG Benchmark (Class A) 88
Figure 6.6 Data Cache Performance in Lazy DTCT Configuration in CG Benchmark (Class A) 88
Figure 6.7 Data Cache Performance in DTCT Configuration in PSTSWM Benchmark 89
Figure 6.8 Data Cache Performance in Lazy DTCT Configuration in PSTSWM Benchmark 89
Figure 6.9 Network cache sensitivity to the size and associativity in PSTSWM Benchmark 90

Figure 6.10 Comparison of Data Cache Performance with different approaches for CG Benchmark (Class A) for different inter-arrival rates (Each bar represents the increments of the corresponding values with reference to those obtained by *LDTCT* per message) 93
Figure 6.11 Comparison of Data Cache Performance with different approaches for PSTSWM Benchmark for different inter-arrival rates (Each bar represents the increments of the corresponding values with reference to those obtained by *LDTCT* per message) 94
Figure 6.12 First access time behavior for different configurations in CG Benchmark (Class A) for different inter-arrival rates 95
Figure 6.13 First access time behavior for different configurations in PSTSWM Benchmark for different inter-arrival rates 95
Figure 6.14 Last access time behavior for different configurations in CG Benchmark for different inter-arrival rates 96

Figure 7.1 TCP/IP Communication Protocol 101
Figure 7.2 VIA Communication Protocol 101
Figure 7.3 A multi-computer configuration in Simics 104
Figure 7.4 TCP/IP and M-VIA overhead during MPI_Receive 105
Figure 7.5 Analysis of Read Miss Rate during MPI_Receive in PSTSWM benchmark for different payload sizes 108
Figure 7.6 Analysis of Miss Rate (R/W) during MPI_Receive in CG benchmark (Class A) for different payload sizes 108
Figure 7.7 Analysis of Read Miss after MPI_Receive in PSTSWM benchmark for different payload sizes 109
Figure 7.8 Analysis of R/W Miss after MPI_Receive in CG benchmark (Class A) for different payload sizes 109

Figure 7.9 Message First-Access-Time behavior of different policies 112
Figure 7.10 Message Last-Access-Time behavior of different policies 112
Figure 7.11 DTCT Policies speed up in Comparison to VIA 113
Figure 7.12 The effect of Data cache size increase (that is, 64KB) in VIA with respect to DTCT policies 113

Figure 8.1 An overview of the Cell processor 120
Figure 8.2 Components of data transfer 122
Figure 8.3 Total data transfer time per message using the GET function (SPE-initiated) 130
Figure 8.4 Accumulative data delivery components of the GET function (SPE-initiated) 130
Figure 8.5 Total data transfer time per message using the GET function (PPE-initiated) 131
Figure 8.6 Accumulative data delivery components of the GET function (PPE-initiated) 131
Figure 8.7 Total data transfer time per message using the PUT function (SPE-initiated) 133
Figure 8.8 Accumulative data delivery components of the PUT function (SPE-initiated) 133
Figure 8.9 Total data transfer time per message using the PUT function (PPE-initiated) 134
Figure 8.10 Accumulative data delivery components of the PUT function (PPE-initiated) 134
Figure 8.11 Latency to transfer data from one SPE to another SPE (GET) 136
Figure 8.12 Latency to transfer data from one SPE to another SPE (PUT) 136

List of Abbreviations

Active Message	AM
Asynchronous Memory Copy	AMC
Broadband Engine	BE
Block Tridiagonal Application Benchmark	BT
Computational Fluid Dynamic	CFD
Conjugate Gradient Application Benchmark	CG
Chip Multiprocessors	CMP
Cluster of Workstations	COWs
Direct Memory Access	DMA
Distributed Shared-Memory Multiprocessor	DSM
Direct-To-Cache-Transfer	DTCT
Embedded Transport Acceleration	ETA
First In First Out	FIFO
Fast Message	FM
3-D Fast-Fourier Transform Application Benchmark	FT
Grand Challenge Application	GCA
High-Performance Community	HPC
High Performance Fortran	HPF
Input/Output	I/O
I/O Acceleration Technology	IO/AT
Instruction Set Architecture	ISA
Low-level Application Programmers Interface	LAPI
Lazy Direct-To-Cache-Transfer	LDTCT
Memory Flow Controller	MFC
Memory Management Unit	MMU

Message Passing Interface	MPI
Massively Parallel Processor	MPP
Numerical Aerodynamic Simulation	NAS
Network Interface	NI
Network Interface Card	NIC
Network of Workstations	NOWs
NAS Parallel Benchmark	NPB
Non-Uniform Memory Access	NUMA
Parallel Spectral Transform Shallow Water Model	PSTSWM
Parallel Virtual Machine	PVM
Pure QCD Monte Carlo Simulation Code with MPI	QCDMPI
Remote Direct Memory Access	RDMA
Systems Area Network	SAN
Single Instruction Stream Multiple Data Stream	SIMD
Scalar Pentadiagonal Application Benchmark	SP
Synergistic Processing Element	SPE
Translation Lookaside Buffer	TLB
TCP/IP Offload Engine	TOE
User-managed TLB	UTLB
Virtual Interface	VI
Virtual Interface Architecture	VIA
Virtual Memory-Mapped Communication	VMMC

Acknowledgment

This work could not have been accomplished without the support of many.

First, my deep gratitude to my supervisor Professor Dr. Nikitas J. Dimopoulos whose guidance supported me through my Ph.D. studies. I have been extraordinarily fortunate in having his advice and access to his wide knowledge. His was a crucial contribution to my research and this dissertation. Professor Dimopoulos' intellectual maturity has nourished my own and I will benefit for a long time to come.

Words cannot express the magnitude of my appreciation to my wife, Mandana Saadat, for her commitment and her confidence in me. Her encouragement has sustained me in completing this work. I thank her and my dear sons, Bardia and Kasra, for their patience and most of all for their unconditional love. I also want to express my gratitude to my parents and parents-in-law for their supports over the years. Their belief in me and my abilities has allowed me to pursue my dreams.

Special thanks to the members of my Examiners Committee: Dr. Kin F. Li, Dr. Amirali Baniasadi, and Dr. D. Michael Miller. Their suggestions and comments have improved the quality of my research. I appreciate that in the midst of all their activities they agreed to be my Examiners. I am also grateful to Dr. Georgi N. Gaydadjiev for agreeing to be the External Examiner for this dissertation and his brilliant comments.

I want to acknowledge my friends and fellow researchers in the LAPIS lab, especially Rafael Parra-Hernandez, Nainesh Agarwal, Ehsan Atoofian, Maryam Mizani, Kaveh Jokar Deris, Kaveh Aasaraai, Solmaz Khezerlo, Erik Laxdal, Daniel C. Vanderster, Ambrose Chu, Scott Miller, Eugene Hyun, and Darshika Perera; I enjoyed working with you all.

I cannot end without thanking my instructors at Shiraz University for their contributions during my Bachelor and Master studies. I am specially indebted to

my Master's supervisor, Dr. Ahmad Towhidi, who is an oasis of ideas in computer science and engineering. I also extend my appreciation to Dr. Seradj-Dean Katebi, Dr. Hassan Eghbali Jahromi, Dr. Majid Azarakhsh, Dr. Mansour Zolghadri Jahromi, Dr. Gholamhossein Dastgheibifard, and Mr. Mohammadali Mobarhan.

My Ph.D. studies were supported in part by a scholarship from I.R. Iran's Ministry of Science, Research, and Technology. I am also grateful for financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Victoria through the Lansdowne Chair.

Farshad Khun Jush, Victoria, British Columbia, Canada

Dedication

In memory of my father, Hamdollah Khun Jush,
and my sister-in-law, Anahita Saadat

To my dear wife, Mandana Saadat,
my mother, Soghra Zalekian,
my father-in-law, Gholamhossein Saadat,
and my mother-in-law, Shahrbano Safazadeh

Chapter 1

Introduction

Research in high-performance architecture has been focusing on achieving more computing power to solve computationally-intensive problems, some examples of which are *Grand Challenge Applications* (GCAs). GCAs are fundamental problems in science and engineering with broad economic and scientific impact [2]. These applications need more computing power than a sequential computer can provide. Using faster processors employing parallelism at different levels (such as pipelining, superscalarity, multithreading, and vectorization) has been a viable way to speed up computation. With the advancements of single-core high-performance processors, recently a set of factors including poor performance/power efficiency and limited design scalability in monolithic designs has moved high-performance processor architectures toward designs that feature multiple processing cores on a single chip (also known as CMP). Examples of these processors are IBM Power5 [111], IBM Cell BE [70], Intel Montecito [90], Sun Niagara [82], AMD Opteron [69], and AMD Barcelona [8].

Although these approaches show significant improvement, they are applicable only to a limited extent. Moreover, commercial workloads, which are the largest and fastest growing applications in high-performance computing, increasingly need

computational power that cannot be provided by these approaches.

The memory system has a key role in reaching high-performance and scalable solutions for large-scale scientific applications. In fact, in single-core high-performance processors, increased processor clock speeds, along with new architectural solutions, have aggravated and widened the gap between processor and memory performance. The memory performance gap is exacerbated in CMP environments because the data shared among different cores may require traversing multiple cache hierarchies, which may cause the processor to be idle for several thousands of processor clock cycles. The existence of CMPs, along with the availability of high-bandwidth and low-latency networks, calls for new techniques to minimize and hide the message delivery latency in order to increase potential parallelism in parallel applications.

As stated, these approaches do not provide proper scalability and speed up where computational power and scalability are the key concerns (that is, GCAs). Another class of architectures that promises to provide more computing power and scalability is the parallel architecture approach. To overcome the computational power and scalability problems in high-performance processors, many different parallel computer systems that support intensive computation applications have emerged.

1.1 Parallel Computer Architectures

By definition, a parallel computer is a "collection of processing elements that communicate and cooperate to solve large problems fast" [22]. Although there are different taxonomies in categorizing parallel architectures, we can recognize two main categories: shared memory and distributed memory architectures.

In shared memory architectures, several processing elements are closely connected through placing all the memory into a single address space and supporting virtual address space across all the memory. In this approach, data is available to all

processes through simple *load* and *store* operations, which provide low-latency and high-bandwidth access to remote memory. The simplest physical connection is the common bus architecture. However, this technique suffers from scalability. Non-Uniform Memory Access (NUMA) architectures have been developed to alleviate this bottleneck by limiting the number of CPUs on one memory bus, and connecting the various nodes by means of a high-speed interconnect. This approach has its own drawbacks, including nonuniform access time to the shared memory distributed among the nodes.

In the distributed memory approach, a number of independent processing elements are connected through a network. The typical programming model consists of separate processes on each computer communicating by sending and/or receiving messages. These systems are the most common parallel computers because they are easy to assemble. IBM Blue Gene and IBM RS/6000 SP are two examples of such machines. There are two main categories in distributed memory architectures: Massively Parallel Processors (MPPs) and Cluster of Workstations (COWs). A MPP is a computer system with many independent arithmetic units or entire processing elements connected by an interconnection network and run in parallel. A Cluster of Workstations (COWs) is a number of workstations connected through a computer network.

To take advantage of higher parallelism in the applications, a highly scalable parallel platform is needed. MPPs provide scalability that can support up to hundreds or even thousands of processing elements. These processing elements are connected by a fast interconnection network through Network Interfaces, and they communicate by sending messages to each other. The processing elements consist of a main memory and at least one processor, and they run a separate copy of the operating system (OS) locally. The advantage of these systems is scalability for many coarse grained applications, as well as their favorable cost. This class of computers is specialized

and proprietary in such a way that even minor changes in hardware or software take a long time, which means a high time-to-market. Examples of this architecture are Intel Paragon, CM-5, IBM RS/6000 SP, IBM Blue Gene [13], and Earth Simulator [1].

The next category in distributed memory systems that has received more attention is cluster computing. This class of parallel architecture systems, which has many commonalities with MPPs, consists of a collection of commodity components, including PCs that are interconnected via various network technology. Utilizing commodity components results in better price/performance ratios. The downside of this approach is that they cannot easily be fine tuned. Examples include the Berkeley NOW [24], IBM SP2 [21], Barcelona JS20 Cluster [13], SGI Altrix XE x86-64 [11], and Solaris Cluster [12]. Cluster computing, consisting of low-cost and high-performance processors, has been used to prototype, debug, and run parallel applications since the early 1990s as an alternative to using proprietary and expensive supercomputer platforms. Other factors that made this transition possible were the standardization and support for many of the tools and utilities used in parallel applications. Message Passing Interface library (MPI) [5, 88], data parallel language HPF [3], and Parallel Virtual Machine (PVM) [112] are some examples of these standards which are ported to cluster environments. These give portability to the applications designed on clusters in such a way that they can be ported to the dedicated parallel platforms with little modification or effort.

There are several reasons why cluster computing (for example, Network of Workstations (NOWs) or Cluster of Workstations (COWs)) is preferred over expensive and proprietary parallel platforms [24, 30].

1. The emergence of high-performance processors together with decreasing economic cost are two of the main factors in the suitability of cluster computing. Processor performance has increased dramatically in the last decade due to

the fact that the number of transistors on integrated circuits doubles every 18 months [92].

2. As a result of new networking technologies and protocols, an increased communication bandwidth and a decreased communication latency have made cluster computing a feasible approach in parallel platforms. The cost-effective interconnects known as Systems Area Networks (SANs) [40, 66, 113, 26, 59, 9], which operate over shorter physical distances, have motivated the suitability of a network of workstations or multiprocessors as an inexpensive high-performance computing platform.

3. Workstation clusters are significantly less costly than specialized high-performance platforms.

4. The nodes in a cluster environment can be improved by adding more memory or processors, which means better scalability.

5. The development tools for workstations are more portable and mature than those in specialized parallel computing platforms.

As can be seen, COWs are a suitable alternative to high-performance computing platforms. However, communication overhead (including hardware, software, and network) adversely affects the computation performance in a cluster environment, as in any other message passing system. Therefore, decreasing this overhead is the main concern in such environments.

Generally, low-latency and high-bandwidth communication together with data sharing among processing elements are critical in obtaining high performance in these environments. The raw bandwidth of networks has increased significantly in the past few years and networking hardware supporting bandwidth in the order of gigabits

per second has become available. However, traditional networking architecture and protocols do not allow the applications to benefit from the available raw hardware interconnect performance. Latency is the main bottleneck in traditional networking architecture and protocols. The main components of latency are the speed limitations of transmission media and extra copies which are required to move the data to its final destinations. For example, the layered nature of the legacy networking software and the use of expensive system calls and the extra memory-to-memory copies required in these systems profoundly affects communication subsystem performance, as seen through the applications.

In fact, computer systems have been experiencing the latency problem due to memory or I/O operations (memory wall). Increased processor clock speeds, along with new architectural solutions, have aggravated and widened the gap between processor and memory performance. As a result of this performance gap, processors suffer several hundred cycles to access the data residing in the main memory. Besides the execution phase of an application, the communication component is also a limiting factor in reaching high performance in highly parallel and highly scalable applications. In the communication phase of an application, the processor needs to send (receive) the data to (from) the network. As mentioned, the layered nature of the legacy networking software, the use of expensive system calls, and the extra memory-to-memory copies make the processor wait for the completion of the communication component before resuming its execution. Therefore, it is necessary to tackle the latency problem, which is unavoidable, to leverage the potential of existing high-speed and low-latency networks, as well as fast processors.

Various mechanisms were proposed to achieve a true zero-copy data forwarding in message passing environments. These mechanisms have been proposed as processor extensions comprised of a specialized network cache as well as special instructions that help managing the operations of the cache and implementing late binding. This

architectural extension facilitates the placement of the arrived data in a cache, bound and ready to be consumed by the receiver. The structure of the extension is described in section 3.1.

1.2 Dissertation Contributions

This dissertation proposes and implements mechanisms to address the above-mentioned latency problem in high-bandwidth communication networks in message passing environments. These mechanisms have been proposed as a processor extension comprising a specialized network cache as well as special instructions that help managing the operations of the cache and implementing a late-binding mechanism. This extension facilitates the placement of the arrived data in a cache, bound and ready to be consumed by the receiver. This dissertation has five contributions:

- The main goal is to propose techniques to achieve zero-copy communication in message passing environments. A *network cache* has been proposed [18] in the processor architecture to transfer the received data near to where it will be consumed. The first contribution is the introduction of architectural extensions that facilitate the placement of the arrived message in message passing environments. The extensions are introduced in Chapter 3.

- As a second contribution, various data transfer techniques are proposed to achieve zero-copy data transfer in message passing environments. These are described in Chapter 3. Specifically, two different policies are introduced that determine when a message is to be bound and sent to the data cache. These policies are called Direct to Cache Transfer *DTCT* and *lazy DTCT*.

- The third contribution investigates behavior and the optimum configuration of the proposed architectural extensions (Chapters 5 and 6). Specifically, concerns

are investigated relating to the size and associativity of the network cache and their relationship to the message traffic. Additionally, the impact of the message traffic on the data cache is investigated. In Chapter 6, the caching environment is studied further and data transfer techniques are evaluated by different metrics to illustrate the effectiveness of the proposed extensions and data transfer techniques.

- The fourth contribution compares the benefits of the proposed extensions to two common communication protocols, TCP/IP and VIA. The comparison is presented in Chapter 7. For this part of the study, the Virtutec Simics environment [54], a full system simulator is employed. The receive operations overhead in TCP/IP and VIA implementation of MPICH [6] are evaluated and the cache behavior during these operations is explored. Subsequently, the achieved data are utilized within the SimpleScalar simulation environment of the extensions to obtain a realistic evaluation of the behavior of the proposed extensions in a system environment.

- As stated, the memory system has a key role in reaching high-performance and scalable solutions for large-scale scientific applications. The memory performance gap (also known as the memory wall) is exacerbated in CMP environments because the shared data among different cores may require traversing multiple memory hierarchies, which may cause the processor to be idle for several thousands of processor clock cycles. The existence of CMPs, along with the availability of high-bandwidth and low-latency networks, calls for new techniques to minimize and hide the message delivery latency to increase potential parallelism in parallel applications. As the final contribution, the data transfer mechanisms between the processing elements of the Cell BE are studied and their communication capabilities (in terms of latency and throughput) are

identified for a variety of communication patterns. The ultimate goal is to use this information, together with prediction techniques, to implement an efficient MPI environment on the Cell BE (Chapter 8). Two general data transfer methods are investigated, involving the *PUT* and *GET* functions. The *GET* function transfers data from the PPE to the SPEs. The *PUT* function transfers data from the SPEs to the PPE. The components contributing to the communication overhead in these data transfers are explored. These components include the DMA issue and set-up times, latency, data-delivery time, memory-management overhead, and synchronization between cores.

Chapter 2

Background

Parallel architectures promise to provide more computing power and scalability for computation-intensive applications. Low-latency and high-bandwidth communication, together with the sharing of data among processing elements are critical to obtaining high performance in these environments. The raw bandwidth of networks has increased significantly in the past few years and networking hardware supporting bandwidth in the order of gigabits per second has become available. However, the communication latency of traditional networking architecture and protocols does not allow applications to benefit from the potential performance of the raw hardware interconnect.

Several factors can be enumerated as the source of the bandwidth/latency imbalance, both in hardware and software. In the hardware category, the first observation comes from Moore's law [92], which predicts a doubling in the number of transistors per chip every 18 months due to advancements in the scaling of semiconductor devices. In fact, bandwidth benefits from this law more than latency [100] because of an increase in the number of transistors and consequently more pins that can operate in parallel. However, latency does not keep pace with the improvement in bandwidth. The smaller feature size results in larger transistor

counts; in turn, we expect bigger chips in new generations of semiconductor devices. Given a bigger chip, the time needed to traverse that chip becomes longer, which results in longer latency. Another possibility is that improving bandwidth might hurt latency. For example, increasing bandwidth in a memory module can be achieved by adding several memory modules, but complex address decoder circuitry may increase latency.

As well as the above-mentioned observations concerning hardware, software makes the situation worse. Applications issue a system call when they need to send (receive) a message to (from) a network in a cluster system. The layered nature of the legacy networking software and the use of expensive system calls and extra memory-to-memory copies, all of which are necessary in these systems, profoundly affect the communication subsystem performance as seen in the running applications.

Although there have been introduced several high-bandwidth and low-latency communication networks, which are usually used in cluster environments, applications generally cannot benefit from these networks because of the high processing overhead attributed to communication software, including network interface control, flow control, buffer management, memory copying, polling, and interrupt handling.

In the following, an overview of I/O processing is provided, including data transfers and memory operations that affect performance.

2.1 I/O Data Transfer Mechanisms

The following subsections discuss the necessary steps and copy operations involved in sending and receiving data from a network. Then, a qualitative description of memory transactions from a cache perspective will be explained.

2.1.1 Send and Receive Data Transfers

In traditional software messaging layers there are usually four message copying operations from the send buffer to the receive buffer, as shown in Figure 2.1. These occur during data transfer from the sender to a system buffer, and from the system to the Network Interface (NI) buffer. Crossing the network, at the receive side, the arrived message is copied from the NI to the system buffer, and also from the system to the receive buffer when the receive call is posted. Below is a further elaboration of the send and receive mechanisms.

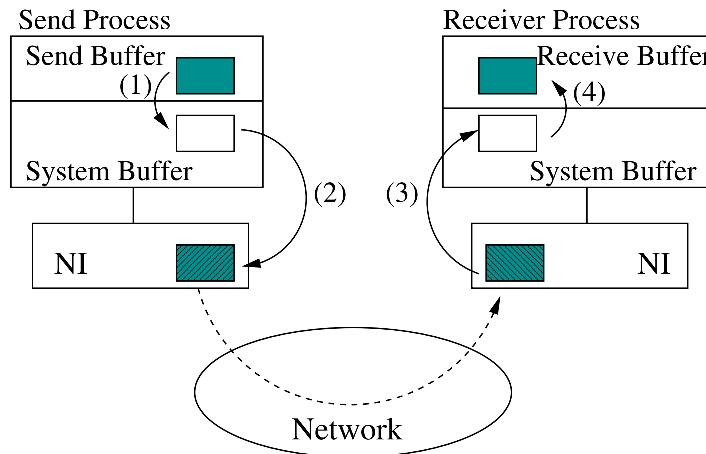

Figure 2.1: Copy operations in data transfer from a sender to a receiver

Different methods can be employed to move a message from a source location to the network. These techniques involve Direct Memory Access (DMA) transfer or regular copy operations. Due to their better performance, contemporary processors leverage DMA techniques to transfer data between a processor's memory system and a network interface card (NIC). With this technique, a DMA engine transfers data from the memory to the NIC, or in the reverse direction, without the involvement of the processor, except for the DMA initiations and terminations.

A node initiates a `send` transfer by issuing a send command. Two actions are involved in sending a message. First, user data is copied into a staging buffer, which is the DMA buffer queue in the system area. Then, a descriptor is written into the NIC control registers area and a flag is set to inform the NIC of the new request. The descriptor includes the destination address, the start address of the message in the DMA buffer area, and the size of the message. The NIC repeatedly polls the status of the ready flag for the existence of a new message to transfer. The NIC starts transferring the message to its own memory area using DMA when it detects the ready flag is set. Finally, the message is injected into the network by the NIC.

To receive a message from the network, the NIC receives the message destined for this node into its memory buffer. The NIC also contains a queue of received descriptors that locate the free buffers in the DMA staging-buffer area. The NIC uses this information to start a DMA operation to transfer the received message into the staging buffers. The arrival of a message can be recognized by the receiving host in two different mechanisms: polling and interrupt. With polling, the NIC sets a flag, which is repeatedly polled by the host, to indicate the arrival of a message. This needs the full attention of the host's processor, which is not an efficient technique. With interrupt, the NIC informs the processor of the arrival of a new message through raising an interrupt. After recognizing the arrival, the receiving host's CPU starts copying data from the DMA staging buffer to the user buffer. Note that DMA can be employed for this transfer as well. Figure 2.2 shows the detail of these transfers at the sender and receiver hosts.

Several copy operations are involved in sending and receiving data from networks and contribute to high-latency I/O transfers. As well as these operations, memory transactions affect the data transfer performance. This problem is discussed in the following subsection.

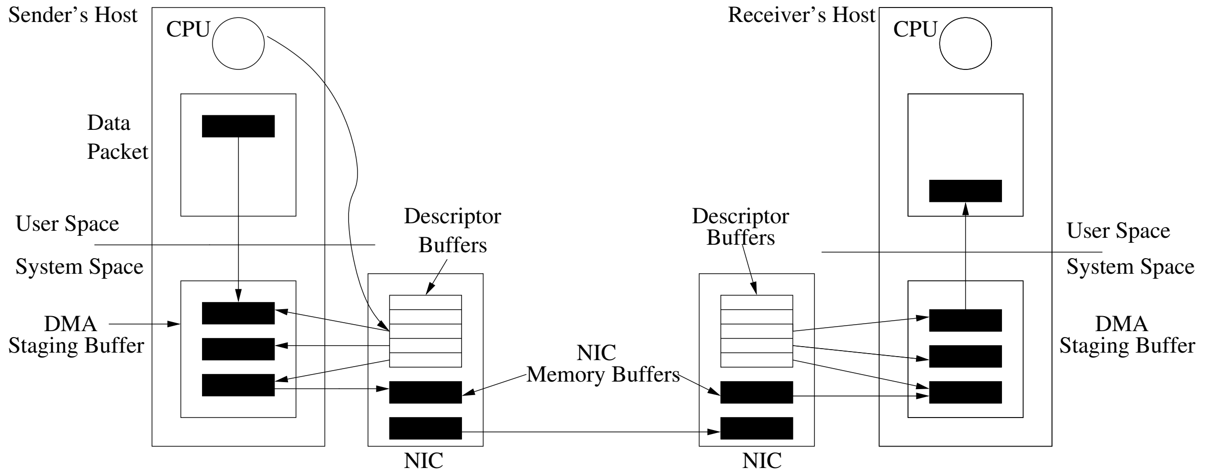

Figure 2.2: A typical Network Interface Card Architecture

2.1.2 Memory Transactions in I/O Operations

I/O operations, as mentioned above, involve memory transactions to send or receive data from a network. These operations affect the memory hierarchy of the processor and may adversely impact the performance. The following model is used to show the memory transactions involved in I/O operations and how they may affect performance.

As explained, several data structures, including descriptors, status flags, and staging buffers are necessary for I/O operations. These structures are accessible by the processor and the NIC; therefore, a coherency mechanism is necessary through the system interconnect [45]. Today's processors often employ snoopy protocols for cache coherency purposes. This technique usually exploits a memory bus coupled to other components, while other configurations (such as a tree configuration) might also be employed. In the bus configuration, each memory transaction is visible to the bus, and the bus takes appropriate actions according to the snoopy protocol.

Because of the increasing gap between the CPU and the performance of the memory subsystem, the memory transactions, which are the results of I/O operations as well as the CPU computations, affect the performance of running applications. The following elaborates the unavoidable memory transactions in I/O operations.

Upon arrival, the data is transferred to its target buffer through *write* operations while the memory controller invalidates the corresponding cache line(s) from the processor's cache. These memory transactions affect subsequent access. For example, the invalidated cache lines are usually accessed again by the processor; thus, a cache miss is the outcome of this access. Such incurred cache misses, which are translated into several hundred clock cycles, have a profound impact on performance. In general, the number of cache misses to access a payload of N cache lines is equal to 2N. Specifically, N cache misses occur during the transfer of the arrived data into the target buffer in the processor's memory, and the same number of cache misses occur in the first access of the data by the running application if the already cached data is evicted from the data cache due to replacements.

When sending a message, the data is transferred to the staging buffer from the processor's cache when the data exists there. If the requested data does not reside in the processor's cache, the data will be retrieved from the processor's memory.

Other serious concerns that need attention are the computational and memory requirements of deep-layered communication protocols such as TCP/IP. For example, several copy operations are required to pass data through different layers of communication protocols.

In summary, I/O operations directly involve memory subsystems through accessing destination or source buffers, or indirectly by manipulating message descriptors. In addition, the cache subsystem, which is affected by the execution of communication protocols, experiences misses that further affect the working set of the running application. In turn, the gap between processor and memory performance makes

the cache misses expensive. To minimize such cache misses and the memory copying problem, solutions are proposed in this dissertation that minimize interference in the caching subsystems and ensure data availability to the consuming process or thread.

To overcome the aforementioned bottlenecks, others have proposed various approaches to enhance some aspects of communication performance. These include improving network performance, providing more efficient communication software layers, designing dedicated message processors, providing bulk data transfers, off-loading or on-loading communication protocol processing, and integrating NIC and the processor on the same chip. These techniques are surveyed in the following section. Then, our proposed techniques are explained.

2.2 Communication Subsystem Improvement Techniques

The improvement of various aspects of communication performance has been an active research area in parallel computer architecture for more than two decades. First, improvement techniques were employed in Massively Parallel Processors (MPPs), which have a large number of processors jointed by an interconnection network. Then, some were extended and employed in cluster environments. In the following subsection, these techniques are discussed in detail.

2.2.1 Massively Parallel Processors (MPPs)

The main objectives of the proposed techniques in MPPs are to improve the latency of data transfer and to leverage the proprietary high-bandwidth interconnection networks. These techniques cover a variety of methods that include integrating message transactions into the memory controller or the cache controller, incorporating messaging deep into the processor, integrating the network interface on the memory bus, providing dedicated message processors, and supporting bulk transfers.

There have been several attempts to overcome communication latency in such architectures. The Flash and CrayT3D [85, 25] projects are two that endeavored to integrate messaging systems into the memory controller. The Flash multiprocessor exploited a custom node controller called MAGIC to handle all communication within and among nodes using a programmable protocol processor. The CrayT3D also provided a block-transfer engine that supports memory-to-memory data transfers using DMA.

Another approach integrates message processing into the cache controller [19, 58, 87]. The Alewife MIT [19] employs one of these techniques. On each node, a communication and memory management unit handles memory requests from the host processor and determines the source of the requested memory. This unit is responsible for a cache-coherence mechanism and synthesizes messages when data needs to be received from a remote node. DASH [87] is another technique in this category; it is comprised of a prefetching mechanism and mechanism for depositing data directly into another processor's cache. The KSR1 [58] project also provides a shared address space through a cache-only memory.

Another way to improve the communication subsystem in MPPs is to integrate the messaging system deep into the processor [44, 46, 97, 99]. Some techniques incorporate multiple contexts and integrate a cache and a communication controller. For example, the goal in Avalanche [44] is to change the memory architecture of the processor to include a multi-level context-sensitive cache that is tightly coupled to the communication fabric. The proposed technique addresses the placement of the arrived data into the different levels of memory hierarchy. The J-machine project at MIT [46] also supports multiple communication paradigms within a single machine. It, too, has a special message-driven processor, which handles message passing among different nodes and performs memory management functions. Another scheme involves utilizing dedicated message processors to handle message passing

communications with the network [34, 102, 107]. These techniques incorporate a processor core in their network interface that is dedicated to sending and receiving data from the network.

2.2.2 Cluster Computing

Cluster computing consisting of low-cost and high-performance processors has been used to prototype, debug, and run parallel applications since the early 1990s, as an alternative to using proprietary and expensive supercomputer platforms such as MPPs. The emergence of high-performance processors at a decreasing cost is a significant factor in the suitability of cluster computing. As explained earlier, processor performance has increased dramatically in the last decade because the number of transistors on integrated circuits doubles every 18 months [92].

However, communication overhead, including hardware, software, and network, adversely affects the computation performance in a cluster environment, as in any other message passing system. Modern switched networks called System Area Networks (SANs) (for examples, Myricom Myrinet, IBM Vulcan, Infiniband, Tandem Servernet, Spider, Cray Gigaring, Quadrics) [40,113,23,9,26,66,59,65], which provide high-bandwidth and low-latency communication, have been designed to alleviate the bandwidth and latency in legacy networks that are used in cluster environments. Even with these low-latency and high-bandwidth SANs, users see little difference to traditional local area networks because of the high processing overhead attributed to communication software, including network interface control, flow control, buffer management, memory copying, polling, and interrupt handling [94].

Several researchers have investigated communication methods to reduce the latency caused by OSs through employing direct user-level access to the network interface hardware (that is, user-level messaging) in cluster environments. Typically, in

these methods the OS maps device memory and/or device registers in the user address space. Therefore, the application can initiate sending and receiving information from the network interface by simple store and load operations respectively. This approach is used by AM, U-Net, U-Net/MM, VMMC2, BIP, and VIA [53, 52, 114, 47, 103, 48], which employ much simpler communication protocols as compared to legacy protocols such as TCP/IP. Data transfer mechanisms and message copying, control transfer mechanisms, address translation mechanisms, protection and reliability concerns are key factors in the performance of the user-level communication systems.

When the OS is removed from the critical path, message copying becomes a significant portion of the software communication overhead. As explained earlier, with traditional software messaging layers there are usually four message-copying operations from the send buffer to the receive buffer, as shown in Figure 2.1.

Memory speed and bus bandwidth make the situation worse, since these have failed to keep pace with the networks' bandwidth and performance and it seems likely the gap will widen in the future [94, 62]. Therefore, many systems have sought methods to deliver a message to a destination by avoiding redundant copies at the application interface or network interface. Examples include U-Net/MM, VMMC, VMMC2, Fast socket, AM, and MPILAPI [114, 39, 47, 108, 53, 33, 48].

The following subsections discuss in more detail software and hardware techniques that attempt to alleviate the communication latency problem in cluster architectures.

Software Techniques

As stated above, high-performance computing necessitates efficient communication across the interconnect. Modern SANs such as Myrinet [40], Quadrics [9], and Infiniband [26] provide high bandwidth and low latency, and use several user-level

messaging techniques to achieve this efficiency. For this dissertation a number of these techniques have been surveyed and the pros and cons of each explored.

Active Message (AM) was one of the first techniques to reduce communication latency, and the main idea is that each message provides the address of a handler to the receiver in order to get the message out of the network. A key design concern involved how to overlap the computation and communication in order to minimize communication overhead in message passing environments. With AM, the handler is responsible for getting messages out of the network and integrating them into the computation running at the processing node. As a result of integrating the communication and computation, the communication overhead could be reduced. Although AMs performed well on massively parallel machines, the immediate response on arrival became more and more difficult on processors with deep pipelines used in cluster environments. The processor needs to dispatch the handler for running and this could be done in two ways, interrupt or polling. Both cause extra operations in the processor resulting in raising either the overhead (in the case of interrupt due to context switching) or the latency (in the case of polling).

Fast Messages (FM) solved the problem by replacing implicit dispatching with explicit poll operation and buffering. Fast Messages can postpone the execution of the handler and reduce the overhead of the messaging system by changing the frequency of polling based on application requirements. As a result of buffering incoming messages, FMs incur extra copies while transferring a message to its final target.

The next user-level messaging system, U-Net, differs significantly from AM and FM. It provides an interface to the network that is closer to the functionality typically found in LAN interface hardware. It does not allocate buffers, perform implicit message buffering, or dispatch messages. The key idea in designing U-Net was to remove the OS kernel from the critical path of sending and receiving messages. U-Net reduced software overhead by shrinking the processing operations required to

send and receive messages and by providing access to the lowest layer of the network.

With the U-Net approach, a physically contiguous buffer region is allocated to each application and is pinned down in the physical memory. The received message is copied into the buffer that has already been allocated upon its arrival through the OS involvement, and a message descriptor is pushed onto the corresponding receive queue. The receive model, supported by U-Net, is either a polling or an event-driven mechanism. In other words, the process can either periodically check the status of the receive queue or receive an interrupt upon the arrival of the next message. This approach suffers from duplication at the receiver and from the large memory size necessary for the communication; this results in decreasing the degree of scalability. Another shortcoming is a reliability concern. Because of its reliance on the underlying network there is a chance of either losing a message or receiving a duplicate message.

U-Net/MM addressed the large memory-size shortcoming in U-Net by adding a translation mechanism. U-Net/MM adds a Translation Look-aside Buffer (*TLB*) into the network interface to solve the problem of fixed buffer regions. In this approach, the size of the network memory is a function of the TLB size. Although zero-copy receives are made applicable by assigning a pool of application-specific buffers to receive buffers, the true zero-copy approach is not always applicable because the application has no control on the received message's destination buffers in the pool.

The next approach was VMMC, developed at Princeton University, to eliminate the copy at the receiving end. VMMC provides a primitive such as `put`, which transfers data directly between the sender and receiver's virtual address under a two-phase rendezvous protocol. In this approach, a mapping called *import-export* exists between sender and receiver applications. The receive buffer and a set of permissions that control the access rights for the buffer are exported from the receiving side to the sending side. Then, the data can be sent to an exported receive buffer by a sender application after importing the buffer. The basic data transfer operations in VMMC

is an explicit request to transfer data from the sender's virtual memory to a previously imported receive buffer. The synchronization between sender and receiver contributes to more network traffic and latency, particularly for short messages. Another concern with this method is the requirement of pinning down communication buffers.

As mentioned earlier, the VMMC model does not support zero-copy protocols because the sender does not have prior knowledge of the receiver's address unless the receiver sends a message to the sender. The other problem with the VMMC approach is the address translation. The NI keeps the mapping for the receive buffers virtual addresses and in the case of a miss it interrupts the host CPU for the translation, which causes a big overhead.

VMMC2 was implemented to address VMMC's shortcomings. The new model introduced three features: User-managed TLB (UTLB) for address translation, transfer redirection, and reliable communication. These features are discussed next.

The VMMC2 transfers the data using a firmware that uses the DMA engine. It sends the pages directly from the host memory to the NI outgoing buffers, and the data in the NI's outgoing buffer is sent to the network at the same time. In addition, the transfer redirection module determines the destination address of the arrived messages using VMMC2 firmware. The user-level send and receive instructions use a virtual address for the send or receive buffer area, and user buffers must be pinned during data transfer operations. However, the NI interface needs the physical address to transfer data using the DMA operation. VMMC2 resolves this problem by introducing the UTLB. The UTLB contains a per process array that holds the physical address of pages belonging to the process virtual memory pages that are pinned in the host's physical memory. VMMC2 also deals with transient network failures and provides applications with reliable communication through implementing retransmission and sender-side buffering on the network interface.

Another technique was MPI-LAPI, an implementation of the Message Passing

Interface standard (MPI) on top of LAPI for IBM RS/6000 SP [33]. LAPI is a low-level user space one-sided communication. With this technique one process specifies all communication parameters, and synchronization is accomplished explicitly to ensure the correctness of communication using the API library for the IBM SP series [14, 110]. It uses eager protocol for short messages and a two-phase rendezvous protocol for long messages, which adds to the network traffic and latency. In this implementation, if a matching receive is not found on the receiver, the message is copied into the early arrival buffer and hence a one-copy transfer is accomplished. For short messages it can achieve good performance in cases where the matching receive has already been issued.

The diversity of approaches and lack of consensus had stalled progress in refining research results into commercial products until Intel, Compaq, and Microsoft corporations jointly proposed an architecture specification called *Virtual Interface* (VI). VI defines a set of functions, data structures, and the associated semantics for moving data in and out of a process memory [48]. It provides a low-latency and high-bandwidth communication between applications on two nodes in a cluster-computing environment. It combines the basic operations of U-Net, adds the remote memory transfer of VMMC, and uses VMMC2's UTLB. It provides remote DMA to transfer data directly from the sender memory to the destination memory. However, it is still necessary to have coordination between sender and receiver. The other concern is how to correctly handle transmission errors. For example, the interface must verify packet checksums before any data can be stored in the memory to ensure the correctness of the message. As the checksum is usually placed at the end of the packet, the interface must buffer the entire packet before transferring it into main memory.

Remote Direct Memory Access (RDMA) [10] has been proposed to overcome the latency of traditional send/receive-based communication protocols (for example, sockets over TCP/IP). This technique allows nodes to communicate without involving

the remote node's CPU. This entails two basic operations: RDMA-write and RDMA-read. RDMA-write transfers data to a remote node's memory and RDMA-read receives data from a remote node's memory. In these operations, the initiator node posts a descriptor, which includes the addresses of local and remote buffers, to the NIC, and the NIC handles the actual data transfer asynchronously. On the remote side, it is the NIC's responsibility to send/receive the data from/to the host memory without CPU involvement. RDMA has potential benefits to leverage. First, as the host processors are not involved in data transfer, the applications can benefit from this computational capability. Second, RDMA eliminates expensive context switches and copy operations in data transfer, which results in better utilization of memory bandwidth and the CPU. Recently, InfiniBand [26], Myrinet [40], and Quadrics [9] have introduced RDMA in LAN environments.

A major concern with all the above page re-mapping techniques is their poor performance for short messages, which is important for current cluster environments. The target buffer's address is another bottleneck in these techniques.

Prediction, discussed below, is another technique that has been used to reduce memory latency in Distributed Shared Memory (DSM) as well as in high-performance processors.

Prediction Techniques

In all the previous techniques, the bottleneck at the receiver is the address of the target buffer. If the receive instruction has not been posted at the receive side, the destination address is unknown upon the arrival of the message; therefore, a message has to be stored in an intermediate area when it arrives. Subsequently, the message is transferred to its final target address once the corresponding receive is posted, which means at least one-copy latency for the arriving message. Even then, cache misses

insert further delays in data use and in the progression of the computation. Prediction techniques have been proposed to tackle this problem, not only in DSM but also in high-performance processors .

Technology improvements and new techniques cause a huge gap between memory access and processor speeds; therefore, memory latency becomes one of the main bottlenecks in these environments. To alleviate this effect, prediction and multithreading are heavily used in DSM and high-performance architectures [49,80,93,95,89,15].

Afsahi and Dimopoulos have proposed heuristic techniques to predict message destinations in message-passing systems [16]. By using these techniques, one can determine the destination node that will receive the next message. One can also predict which of the received messages will be consumed next, but the target buffer still cannot be determined until the corresponding receive is posted. This can be alleviated if an efficient late-binding mechanism exists that can bind the received data to its intended destination without resorting to copying. In message passing environments, several messages may arrive and wait to be bound. Predicting which of the waiting messages will be consumed next allows one to ensure that this message is cached at the consuming process and is readied for binding.

Hardware Techniques

In addition to the above methods that attempt to alleviate the software communication bottleneck at the NIC to CPU connection, there are others that try to solve the bottleneck problem at the hardware level in TCP/IP environments. In fact, although 10-Gigabit Ethernet networks can be plugged into high-end servers through 10GBE adapters, keeping up with the available bandwidth on the links is challenging [55]. Some studies [57, 105] show that performance losses are mainly attributed to the combination of various overheads in interactions between the CPU, memory systems

and the NIC.

TCP/IP Offload Engine (TOE) [104, 68] is a technique that tries to tackle the TCP/IP performance bottlenecks. Network speeds have increased tremendously over the last decade; therefore, the CPU ends up dedicating more cycles to network traffic handling and memory copying than to the applications it is running. In this approach, TOEs offload TCP/IP processing from the main processor and execute it on a specialized processor residing in the NIC. The main motivation for this approach is to increase the network delivery throughput while reducing the CPU utilization. This method can improve the network throughput and the utilization for bulk transfers. However, it has its own disadvantages: lack of flexibility, scalability, and extensibility [91]. In addition, it needs to store and forward the arrived data to its final destination, which contributes to a long latency in applications having numerous short messages. Moreover, TOE engines are attached to I/O subsystems; therefore, arrived data has to cross the I/O buses, which are slow in comparison with the main processor. Thus, the key to improve the bandwidth in transferring data involves improving the host's ability to move data, decreasing the amount of data that needs to be moved, or decreasing the number of times that the data is required to traverse the memory bus [101, 109].

Mogul [91] points out that TOE's time has come. While technical problems still persist in TOE, this method is necessary if one is to increase network throughput. Such high throughput requirements are present in graphical systems, storage, and so on. To reach high data rates, these systems employ special-purpose network interfaces in which high bandwidth and high reliability, rather than flexibility, are their main goals. Using TOE with the availability of much cheaper Gbit/s (or faster) Ethernet is not justifiable or cost-effective. However, the incurred data copy costs in the traditional network protocol stacks prevent the exploitation of the networks' raw performance. Therefore, the copy problem in these protocols should

be attacked, as explained above. As a matter of fact, another study points out that "many copy avoidance techniques for network I/O are not applicable or may even backfire if applied to file I/O" [41]. This study suggests using Remote Direct Memory Access (RDMA) as a likely solution for copy-avoidance techniques. In spite of these arguments against TOE, Infiniband and Myrinet exploit this technique in their new NIC architectures [32]. They have also proposed Socket Direct Protocols (SDP) [61, 31] for two purposes: first, to provide a smooth transition in deploying existing socket-based applications on clusters; second, to use the offloaded stack for protocol processing.

The alternative approach to offloading is called onloading [105], and is provided by Intel for TCP/IP platforms; this technique is also called I/O Acceleration Technology (IO/AT) [86]. This approach leverages multi-core and multi-threaded features in state-of-the-art processors. In contrast to offloading, onloading runs the TCP/IP communication protocol on a separate thread or core in a multi-core or multi-threaded architecture. It uses the flexibility, scalability, and extensibility features of high-performance processors. This approach includes several techniques to reduce latency in high-performance servers. To address system overheads, the Embedded Transport Acceleration (ETA) technique [106] is proposed; it tries to reduce the operating system overhead by dedicating one or more cores for TCP/IP processing.

Further, Intel proposed techniques to reduce the memory copying overhead: lightweight thread, direct cache access, and asynchronous memory copies. Lightweight threading hides the memory-access latency through the employment of several lightweight threads executed in a single OS thread. Each is responsible for processing packets and switches to a different thread when it incurs a long-latency operation such as a cache miss. Another proposed technique called Direct Cache Access (DCA) [67] attempts to bring data closer to the CPU. Asynchronous memory copy (AMC) is another technique to resolve the memory latency problem. It allows copy operations

to take place asynchronously with respect to the CPU.

Another approach to deal with packet delivery in high-bandwidth TCP/IP networks has been proposed by Binkert et al. [36, 38, 37]. This takes advantage of integrating NIC with the host CPU to provide increased flexibility for kernel-based performance optimization through exposing send and receive buffers directly to kernel software running on the host CPU. This approach tries to avoid long-latency copy operations from the NIC buffers to the kernel space because of the ever-increasing disparity between processor and memory performance. Integration of NIC on CPU is not a new technique and was employed in J-Machine [46], M-Machine [56], *T [96], and IBM's BlueGene/L [60]. Similar to this approach, Mukherjee [94] called for tighter coupling between CPUs and network interfaces. He also proposed placing the NIC in the coherent memory domain.

This chapter has surveyed different techniques that have tried to improve communication system performance. At the send side, as discussed above, user-level messaging layers use programmed I/O or DMA to avoid system buffer copying. Some network interfaces also permit writing directly into the network. In contrast to the send side, bypassing the system buffer copying at the receiving side may not be achievable. The bottleneck at the receiver in all these techniques is the address of the target buffers, which is not known until the receive is posted. If the receive instruction has not been posted at the receive side, the destination address is unknown upon the arrival of a message; therefore, the arrived message has to be stored in an intermediate area upon its arrival. The message is transferred to its final target address once the corresponding receive is posted, which means at least one-copy latency for the arriving message. Even then, cache misses insert further delays in data use and in the progression of computations.

As stated, remote direct memory access (RDMA) was introduced to improve data movement at the receiver, but this requires the modification of applications in order

to be RDMA-aware. Additionally, the overhead of this technique for short messages adversely affects the performance of the underlying communication system.

To further improve the latency of the communication system and to overcome the receiving-side bottleneck in the aforementioned approaches, we have proposed architectural extensions along with data transfer techniques to facilitate the placement of the message [71, 72, 73, 74, 75, 76]. These proposed techniques place data into the lowest and fastest level of the memory hierarchy (that is, the data cache), which enables the processor to access it without incurring latency and as early as possible, and binds the data through an efficient binding mechanism. Previous work by Afsahi et al. [17] in predicting message consumption patterns allows the received messages to be managed more efficiently. Message prediction could be used in the architectural solutions to manage the necessary replacements in the proposed components in order to reduce their size.

Our mechanism has been proposed as a processor extension comprised of a specialized network cache and special instructions to help managing the operations of the cache and implementing the late binding. These architectural extensions will facilitate the placement of the message data in a cache, and have it bound and ready to be consumed by the receiver. This mechanism focuses primarily on short messages by transferring the arrived messages into a receiver's cache (network cache) ready to be accessed at a minimum cost. The long messages are addressed by the late-binding optimization.

Chapter 3

Architectural Enhancement Techniques

Computer systems have been experiencing latency problems due to memory or I/O operations (also known as memory wall). Increased processor clock speeds along with new architectural solutions for processing have aggravated and widened the gap between processor and memory performance. As a result of this performance gap, processors expend several hundred cycles in order to access the data residing in the main memory.

Ideally, if the data destined to be consumed by a process or thread has been cached, the process or thread will encounter minimum delay in accessing the data. To achieve this goal, two problems need to be addressed. First, it has to be determined from the network's already arrived messages which is the next to be consumed. That message could then be located in a cache attached to the processor that will consume it. Second, if the receive instruction has not been posted at the receive side, the destination address is unknown upon the arrival of the corresponding message; therefore, the arrived message has to be stored in an intermediate area upon its arrival. This problem can be alleviated if an efficient late-binding mechanism exists that can bind the received data to its intended destination without resorting to copying.

To address the first issue, heuristic techniques have been proposed by Afsahi [16]

to predict message destinations in message passing systems. With these prediction techniques it is possible to determine the destination node that will receive the next message. It is also possible to predict which of the received messages will be consumed next, but the target buffer still cannot be determined until the corresponding receive is posted. As explained above, this problem can be alleviated if an efficient late-binding mechanism exists. The following proposal attacks the late-binding problem.

We have implemented different mechanisms to achieve true zero-copy data forwarding in message passing environments. These mechanisms have been proposed as processor extensions that comprise a specialized network cache as well as special instructions that help managing the operations of the cache and implementing the late binding technique. This architectural extension facilitates the placement of the arrived data in a cache, and have it bound and ready to be consumed by the receiver.

The structure of the extension is described in the following section along with techniques to achieve zero-copy data messaging by leveraging the proposed extensions.

3.1 The Proposed Architectural Extension

The main goal of this dissertation has been to propose techniques to achieve zero-copy communication in message passing environments. A network cache has been proposed [18] in the processor architecture to transfer the received data near the processor where it will be consumed. The predictors can be incorporated in the replacement policies of caches and pages that hold the received messages.

It has been shown that it is possible to predict the sequence of message reception calls [16]; therefore, using the message consumption pattern one can move the next-to-be-used message near the place where it will be consumed. This is most efficiently achieved if the message is moved to the lowest and fastest level of the processor's memory hierarchy (that is, the processor's cache). Predictors can reside beside the

network interface and monitor the message reception patterns to predict the next message to be consumed. The network interface uses these predictions to place early-arriving messages into the cache. However, these messages cannot be bound to the consuming process/thread until the appropriate receive message is issued. As a result, the resolution of the target address imposes serialization and cannot be hidden.

Ideally, if the data destined to be consumed by a process or thread has been cached, that process or thread will encounter minimum delay in accessing this data. Our aim has been to introduce architectural extensions that will facilitate the placement of the message data into a cache, bound and ready to be consumed by the thread or process. This is accomplished through the introduction of a special network cache and extensions to the ISA. It is presumed that a message includes, in addition to the payload, a message ID that may be part of or indeed the entire MPI message envelope. The message ID is used to identify the message and bind it to its target address. For this purpose, two memory spaces can be considered:

- *Network memory space*: This is where network buffers are allocated and where received messages are held waiting to be bound to the process address space.

- *Process memory space*: This is the process memory address space where process objects stay, including bound messages. To facilitate late binding, a separate network cache is proposed that can distinguish the two memory spaces, as shown in Figure 3.1.

The following subsection elaborates the proposed data structures, required operations, and various data transfer mechanisms to achieve zero-copy data transfer in message passing environments.

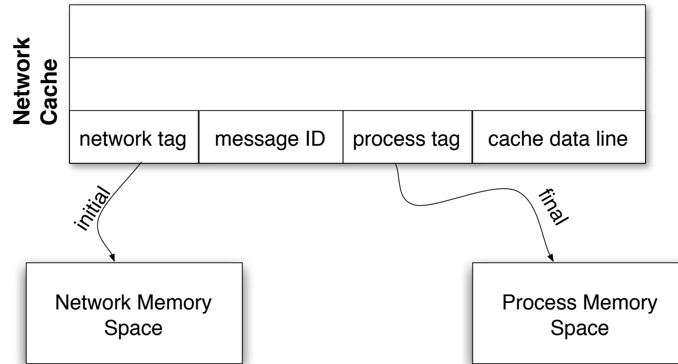

Figure 3.1: The *process* and *network* memory spaces and their relation through the network cache.

3.1.1 Operation

First, we deal with the messages of a length identical to that of a cache line in message passing environments; for messages that are shorter than the cache line, the approach is identical. More complex cases, such as the messages spanning several cache lines, are elaborated upon in section 3.1.4.

The proposed network cache includes three separate tags. The *network tag* is associated with the Network Memory Space, while the *process tag* is associated with the Process Memory Space. A separate *message ID tag* holds the message ID. All three tags can be searched associatively, as depicted in Figure 3.1. For this purpose, we have used three different caches more than likely with various configurations for the network, message and process caches in order to be searched associatively.

Upon its arrival, a message is transferred into the network cache. The *network tag* is set to the address of the buffer in network memory space that is allocated to the message and the *message ID tag* is set accordingly. The message lives in the network cache and migrates to the Network Memory Space according to a cache replacement policy that will utilize the prediction heuristics discussed in [17], to

replace the message that is least likely to be consumed next.

The *message ID* and the *network tag* of a replaced message are held by a link translation buffer so that the message can be located and reloaded into the cache quickly if necessary.

As mentioned earlier, the resolution of the target address imposes serialization and cannot be hidden when a receive call is issued after the arrival of the data; therefore, binding techniques will be proposed to overcome the serialization imposed by the resolution of the target address.

3.1.2 Late Binding

A *receive call* targeting a message on the network cache will invalidate the *message ID* and *network tags* and will set the *process tag* to point to the address of the object destined to receive the message in *Process Memory Space*. The buffer in *Network Memory Space* is now released and can be garbage collected. Similarly, the link translation buffer entry is invalidated, concluding the late binding of the message to its target object. From this point onward, the cache line is associated with the *Process Memory Space*. Upon cache replacement, the message is written back to its targeted object in the *Process Memory Space* and eventually to the data cache. Both the data cache and the network cache are searched for cached objects, and the aforementioned binding process ensures that the object is not replicated in the network and data caches.

Figures 3.2 and 3.3 illustrate the late binding for an arrived message with the message ID equal to ID-N. In this case, a memory buffer, with a return address equal to P, is allocated for the message in the *Network Memory Space* after it arrives. Then, the message tag of a selected entry in the network cache is set to the address of the allocated buffer (that is, P), and the message ID is set to that of the arrived message,

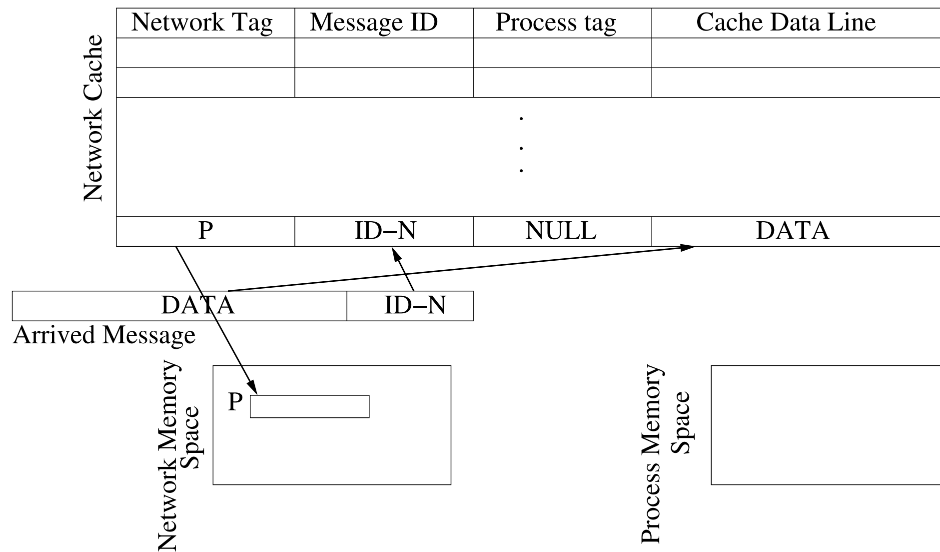

Figure 3.2: The network cache contents after the arrival of a message with Message ID equal to ID-N.

ID-N. A receive call targeting this message will invalidate the message ID as well as network tags and will set the process tag to the address of the object destined to receive the message in the *Process Memory Space*, Q in this case.

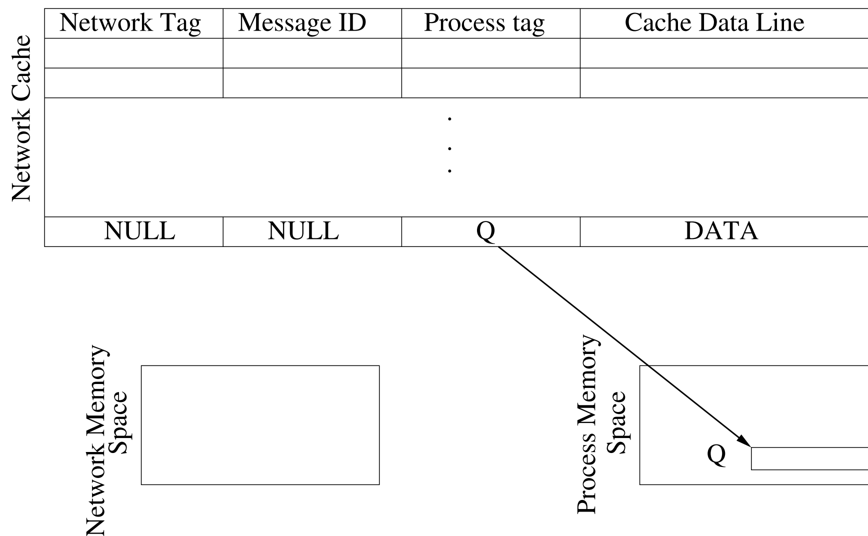

Figure 3.3: The network cache contents after the late binding.

3.1.3 Early Binding

While late binding binds an early-arriving message to a subsequent receive, an earlier posted receive could prepare and reserve a cache line in the network cache and set the corresponding process tag and message ID tag in anticipation of a subsequent message. When the anticipated message arrives, it is placed directly into the reserved network cache line and immediately becomes available to the process through the network cache.

It is also possible to reserve a cache line in the data cache and set the corresponding tag in anticipation of a subsequent message. When the anticipated message arrives, it is placed directly into the reserved data cache line and immediately becomes available to the process. This approach was selected in our implementation.

3.1.4 Other Considerations

The network cache mechanism can be extended to accommodate larger messages by dividing them into blocks and annotating them. Alternatives can be considered for dealing with larger messages which span several cache lines. First, we can consider the larger messages as single unit of data. To handle these the network cache structure should keep track of different cache lines belonging to early-arrived messages. This can be done by annotating each cache line to point to its next and previous lines (for example, using a structure like a doubly-linked list of pointers). This tracking mechanism is necessary in the case of a replacement in any cache line belonging to a message residing in the network cache. In this situation, the entire message needs to be transferred to the network memory space and the corresponding cache lines must become available for the incoming messages to follow.

The second method treats messages as different cache lines in the network cache since the final accesses to the message elements are through units of standard data

formats (for example, integer, float, double). We are able to treat different lines of a message as different short messages, and different cache lines can migrate to the network memory without transferring the entire message into the network memory space. However, we need to keep track of lines evicted from the network cache. This can be accomplished using a network TLB which incorporates the message ID, both process, and network virtual addresses.

Several factors should be explored to discover the appropriate approach. Timing issues related to evicting data from the network cache into the network memory space, transferring data from the network memory space and network cache into the data cache, and the complexity of the tracking mechanism are some examples of concerns that need more investigation.

Also, very large messages can be handled in a similar manner by introducing special *network TLBs* which incorporate the message ID and the process and network virtual addresses. In this case, the early arrived data is transferred into the network memory space.

3.1.5 ISA Extensions

In a general architecture, one may implement a network processor by including a few specialized network-specific instructions that facilitate the implementation of the late-binding operations described earlier. More precisely these are:

- *network_load*
- *network_store*
- *remap*

The *network_load* and *network_store* instructions are identical to the standard load and store instructions, with the exception that they cause the network cache to

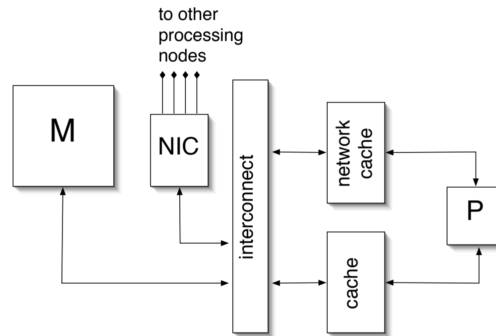

Figure 3.4: The overall architecture of a network-extended node.

be searched according to the network tag. No other cache is searched. Regular load and store instructions target both the normal data cache and the network cache, and the network cache is searched according to the *process tag*.

The *remap message_id,new_process_tag* instruction binds the cache line identified by the *message_id* to the *new_process_tag*. The *message_id* and *new_process_tag* are in registers. The *new_process_tag* corresponds to the virtual address of the object that is to receive the message.

The message ID is used as a key to search the network cache for the presence of the identified message. Upon a cache hit, the physical address corresponding to the *new_process_tag* is written into the cache. Upon a cache miss, the link translation buffer is queried based on the message ID to yield the network tag and bring the corresponding cache line into the cache. This may be done as part of the remap instruction, or it may be emulated in software depending upon its complexity and the achieved miss rate for the remap operations. The network tag is reset and one of the processor status bits is set/reset to indicate the success or failure of the instruction.

These instructions can be added as new ones for the ISA architecture of the processor. The compiler also has to support these new instructions.

The overall architecture of a network-extended node is shown in Figure 3.4.

3.2 Network Cache Implementation

The network cache mechanism presented in section 3.1 (and in Figure 3.1) calls for searches based on a network tag, as long as the received message has not been bound to the receiving process. During binding the message is identified through the *message ID* while after the message is bound it is identified through its *process tag*. These identifiers (that is, the *network tag*, *message tag* and *process tag*) need to be searched. The organization of the network cache has not been specified, though the implication being that it is fully associative. However, a fully associative organization is not practical. A set-associative organization can also be introduced as follows [74].

The network cache is divided into three sections: the *network*, the *message*, and the *process*. The *network* section stores message payloads indexed by the *network tag* that links the payload to the buffer in the network space where the received message is located. The *message section* stores pointers to the message payload that resides in the *network section* while the *message ID* is used as a tag. Similarly, the *process section* also stores a pointer to the message payload, while the process address (which is determined after a receive call is issued) forms the tag that identifies the said pointer. The size and associativity of each section are determined independently and are set so that a minimum number of replacements (because of conflicts between any of the three tags) will result. The size and associativity of each section are studied in section 5.2.1. Figure 3.5 shows the set-associative implementation of the network cache.

In the following sections, the proposed data transfer techniques that leverage the above-introduced data structures are illustrated.

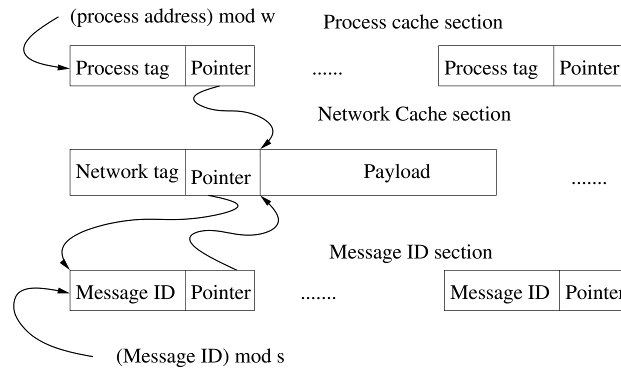

Figure 3.5: Set-associative implementation of the network cache

3.3 Data Transfer Policies

This section elaborates on handling arrived messages using the proposed extension. Specifically, two different policies are introduced that determine when a message is to be bound and sent to the data cache. These policies are called Direct to Cache Transfer *DTCT* and *lazy DTCT*.

3.3.1 Direct-To-Cache Transfer Policy

With this technique, while unbound, the message resides in the network cache. As soon as the corresponding receive is posted, the message is transferred to the data cache and bound. The binding process using this DTCT technique is explained in detail below.

Upon its arrival a message is transferred into the network cache and at the same time a buffer is allocated for it in the network memory space. The network tag is set to the allocated buffer address, and the message ID is set accordingly in the message ID section. Henceforth, the message lives in the network cache and is transferred to the data cache upon receiving the corresponding receive call as explained below.

When a receive call is issued by the running application, the message ID section is searched for the message. If an entry exists in this section, it indicates that the corresponding message has already arrived and is located in the network cache, or resides in the network memory section in the case of a replacement in the network cache. After the message is retrieved from either one, it is transferred to the data cache. Therefore, there are two kinds of data transfers. The first transfers data from the network cache to the data cache, which can be accomplished efficiently. In the second case, where the arrived message resides in the network memory, data is transferred from the network memory to the data cache, which takes more time in comparison. After this transfer, the message is accessible through the processor without latency or delay.

In general, processors that do not employ our extension experience a cache miss in their first access to the arrived data. As a result, the memory management system needs to access the memory to obtain and transfer the data into the data cache. High-latency memory accesses are expensive and affect processor performance adversely. Our technique avoids this high-latency access time by transferring the arrived data to its final destination.

Because transferring data into the data cache might evict elements currently in use, we must investigate the effect of the DTCT approach on the caching system of the processor.

3.3.2 Lazy Direct-To-Cache Transfer Policy

Similar to the DTCT technique, the lazy DTCT (LDTCT) caches arriving messages into the network cache and at the same time allocates a buffer in the network memory space. The network tag is set to the allocated buffer address, and the message ID is set accordingly in the message ID section. However, under LDTCT, the bound

message remains in the network cache. The LDTCT technique has the advantage over the DTCT of not incurring the cost of copying the message to the data cache. However, this technique has the potential to be slower in cases where the cost of copying is less than that of accessing the network cache, which involves indirection. The efficacy of these two techniques is studied in Chapter 6. In the remainder of this section, the binding process using this LDTCT technique is presented in more detail.

When a receive call is issued by the running application the message ID section is searched for the message. If an entry exists inside this section, indicating a corresponding message has already arrived, it is located in either the network cache or the network memory section in the case of a replacement in the network cache. Up to this point the LDTCT is the same as DTCT approach. The following paragraphs point out the differences between the two.

Where the arrived message resides in the network cache, in contrast to the DTCT technique, it remains there, and its entry in the message cache section will be invalidated. The binding of this message to the process address space is accomplished by adding an entry into the process section of the network cache. Afterward, the processor will access the data through the process cache. In other words, the processor searches the data cache and the process cache simultaneously for any data access. Where the replacement of the bound message resides in the network cache, the data will be transferred into the data cache and will be accessed by the processor through that cache onward.

However, if the arrived data has been transferred to the network memory due to a replacement in the network cache, the data will be retrieved from the network memory space and sent to the data cache. This case is similar to the DTCT technique.

3.4 Summary

As can be observed, the proposed DTCT and LDTCT techniques avoid a high-latency access time by transferring the arrived data to the data cache. Processors experience a cache miss in the first access to the arrived data that has been transferred to the main memory. Our techniques alleviate this high-latency access by providing a late-binding mechanism and transferring the arrived data to the data cache before the consuming thread or process needs to access it.

While the late-binding mechanism binds an early-arriving message to a subsequent receive, an earlier-posted receive could prepare and reserve a cache line in the network cache and set the corresponding *process tag* and *message ID tag* (pre-binding) in anticipation of a subsequent message. When that message arrives, it is placed directly into the reserved cache line and immediately becomes available to the process.

As a future work, the *network cache* mechanism can be extended to accommodate larger messages by dividing them into blocks. In this case, an annotating mechanism can be added to the network cache to keep track of messages that span more than one cache line. In addition, very large messages can be handled in a similar manner by introducing special *network TLBs* which incorporate the *message ID* and process as well as network virtual addresses. For messages that are less than a cache line, the approach is identical to that of a cache line.

This chapter introduced architectural extensions comprised of a specialized network cache, data transfer mechanisms, and instructions to manage the operations of this extension. In the next chapter, experimental methodology is established to investigate the effectiveness of the proposed architectural extension in handling the arrived data in message passing environments.

Chapter 4

Experimental Methodology

As the proposed extension targets message passing environments, some standard message passing benchmarks are required to investigate its effectiveness. For this some standard benchmarks were employed, including *NAS Parallel Benchmark* (NPB) suite version (2.4) [28], the *Parallel Spectral Transform Shallow Water Model* (PSTSWM) [115], and the *pure QCD monte carlo simulation code with MPI* (QCDMPI) [63]. These benchmarks have been widely used in the High-Performance Community (HPC) as the de-facto standards for MPI benchmarks representing the computations in scientific and engineering parallel applications. These suites include different sets of problems, and some (for example, LU) have become part of the SPEC MPI2007 suite. Below is a brief description of the functionality of, and justification for, the benchmarks used.

4.1 NAS Benchmarks

The Numerical Aerodynamic Simulation (NAS) program, which is a collection of benchmarks, has been designed to measure the performance of highly parallel computers. The benchmarks, which are derived from computational fluid dynamic

codes (CFD), have been accepted as a standard to compare the performance of parallel architectures, including massively parallel processors and clusters of workstations. This suite consists of eight benchmarks to solve different applications of aerodynamics area, including five kernels and three computational fluid dynamic applications.

The NAS parallel benchmarks consist of six different problem sizes based on the size and number of messages involved in solving the application. The NPB includes short class "S", workstation class "W", large class "A", larger class "B", class "C", and a new class "D".

The NPBs have different implementations. We have employed the MPI [5] implementation of the NPBs since the MPI library is the de-facto standard for the implementation of parallel algorithms in message passing environments. The language of these benchmarks is Fortran 77.

We have used a subset of NPB suite version (2.4) benchmarks, including block tridiagonal (BT), scalar pentadiagonal (SP), and conjugate gradient (CG), which have an abundance of point-to-point communications. The multigrid (MG) and lower-upper diagonal (LU) benchmarks have not been used because they employ collective communication (for example, MPI_ANY_SOURCE) in some of their receive calls.

The benchmarks are briefly explained in the following subsections.

4.1.1 SP and BT Benchmarks

The SP and BT benchmarks have similar structures: each solves three sets of uncoupled systems of equations. These systems are scalar pentadiagonal in the SP code, and the block tridiagonal with 5x5 blocks in the BT code [29]. The SP and BP implementations use a multi-partition scheme to solve these systems. In this approach, each processor is responsible for several disjoint sub-blocks of the grid. Both the SP and BT codes require a square number of processors.

4.1.2 Conjugate Gradient (CG)

The CG benchmark solves an unstructured sparse linear system using the conjugate gradient method. It employs the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzero values. The inverse power method involves solving a linear system of equations $\mathrm{A}z=x$ using the conjugate gradient method [28]. This benchmark requires a power-of-two number of processors.

4.2 PSTSWM

PSTSWM is a message-passing benchmark code and parallel algorithm testbed that solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method; it was developed to evaluate parallel algorithms for this method. Multiple parallel algorithms are embedded in the code and can be selected at run-time, as can the problem size, number of processors, and data decomposition. It is written in FORTRAN 77 and a small number of C preprocessor directives. From among this benchmark's various implementations, we used the MPI version.

4.3 QCDMPI

QCDMPI is a pure QCD (Quantum Chromo Dynamics) Monte Carlo simulation code with MPI [64]. Lattice QCD and its Monte Carlo simulation is a powerful tool to analyze the non-perturbative aspects of QCD. In order to obtain reliable and stable results from Lattice QCD, an enormous amount of computational power is required. QCDMPI allows the simulation to be performed in parallel while keeping the portability through the MPI interface.

4.4 Simulation Methodology

To investigate the effectiveness of the proposed architectural extension, our simulation environment was established by expanding an existing single-thread infrastructure to one that can run MPI applications. For this, we had to implement the proposed extensions and the MPI functions on top of an existing infrastructure. This was accomplished through several steps, including an examination of the distribution of messages and the collection of payloads to use in the simulator. These steps are elaborated in the following section.

4.4.1 Message Distribution and Data Collection

The key decision in a cache design is the cache line and the associativity of the cache. Therefore, the first concern involves the distribution of message sizes as a factor in the selection of these hardware parameters. For this reason, we ran the benchmarks for various numbers of processors, from 4 to 64.

As stated above, the proposed extension targets point-to-point communications in message passing environments; therefore, a subset of NAS benchmarks was used, including block tridiagonal (BT), Scalar pentadiagonal (SP), and conjugate gradient (CG), which have an abundance of point-to-point communications. We analyzed the message distributions for different classes (that is, W, A, B) of the above-mentioned benchmarks. The results show that the message size distributions are the same for various NPB classes. Another key observation is the scalability of the message size as the number of processors increases. In other words, as the applications are distributed to larger and larger systems, their granularities become smaller and the number of short messages increases. Thus, these proposed techniques become more relevant.

Given that all the classes of benchmarks have similar message size distributions and given that class B is computationally intensive in the extreme, which makes it

difficult to run under simulation, it was decided to use the large class (A) from the NAS benchmarks for our experiments. Class A includes enough messages to provide a rich communications environment in which to investigate the proposed techniques, and Class A is a good representative of large-scale cluster applications.

In addition to the subset of the NAS benchmarks, the PSTSWM and QCDMPI benchmarks were also employed. These were executed on the University of Victoria's IBM RS/6000 SP [21] and exploited the instrumentation facility of the MPI library for two purposes:

1. to explore the message size distribution;
2. to collect the payload to use in the architecture simulator in order to run the NAS benchmarks there.

Figures 4.1 through 4.11 show the message-size distributions for different NPB classes. As can be seen from these figures, the number of messages scales based on the benchmark classes; however, the relation between message sizes remains almost identical.

The above benchmarks were executed, and the required data were collected in several files for use in the simulation infrastructure. All traces of message passing functions used in the benchmarks were collected (including MPI_Recv, MPI_Send, and others) to verify the benchmarks message size distributions. Moreover, the payload data in the message passing functions were collected in order to be able to run the parallel benchmarks under simulation. This approach is described in the following sections.

As pointed out, the benchmarks were run to discover the message-size distributions, the concern being the distribution of message sizes as a factor in selecting the hardware parameters. The key decision in a cache design is the cache line

and the associativity of the cache. The benchmarks were run for various numbers of processors, from 4 to 64. The distribution of message sizes shows the SP and PSTSWM use more distinct sizes in their communication calls than CG or BT. As can be observed from the results, some benchmarks such as CG and BT have less variation in message size than SP and PSTSWM.

The communication behavior of the CG benchmark reveals that half of the messages are either 8 or 16 bytes long. As a result, in our first implementation we used this benchmark. By considering the CG message distribution, we explored different configurations of the network cache (that is, total size and associativity) with 16-byte cache lines. Since this benchmark uses a multitude of short messages, it was a good way to begin to test this infrastructure design and the implementation of the proposed architecture.

After exploring the effectiveness of the proposed extension in the CG benchmark, we employed the PSTSWM benchmark, which has an abundance of short messages (8 bytes), in order to investigate the extension performance in various situations with different configurations.

The next section explains the infrastructure employed to investigate the effectiveness of the proposed approach.

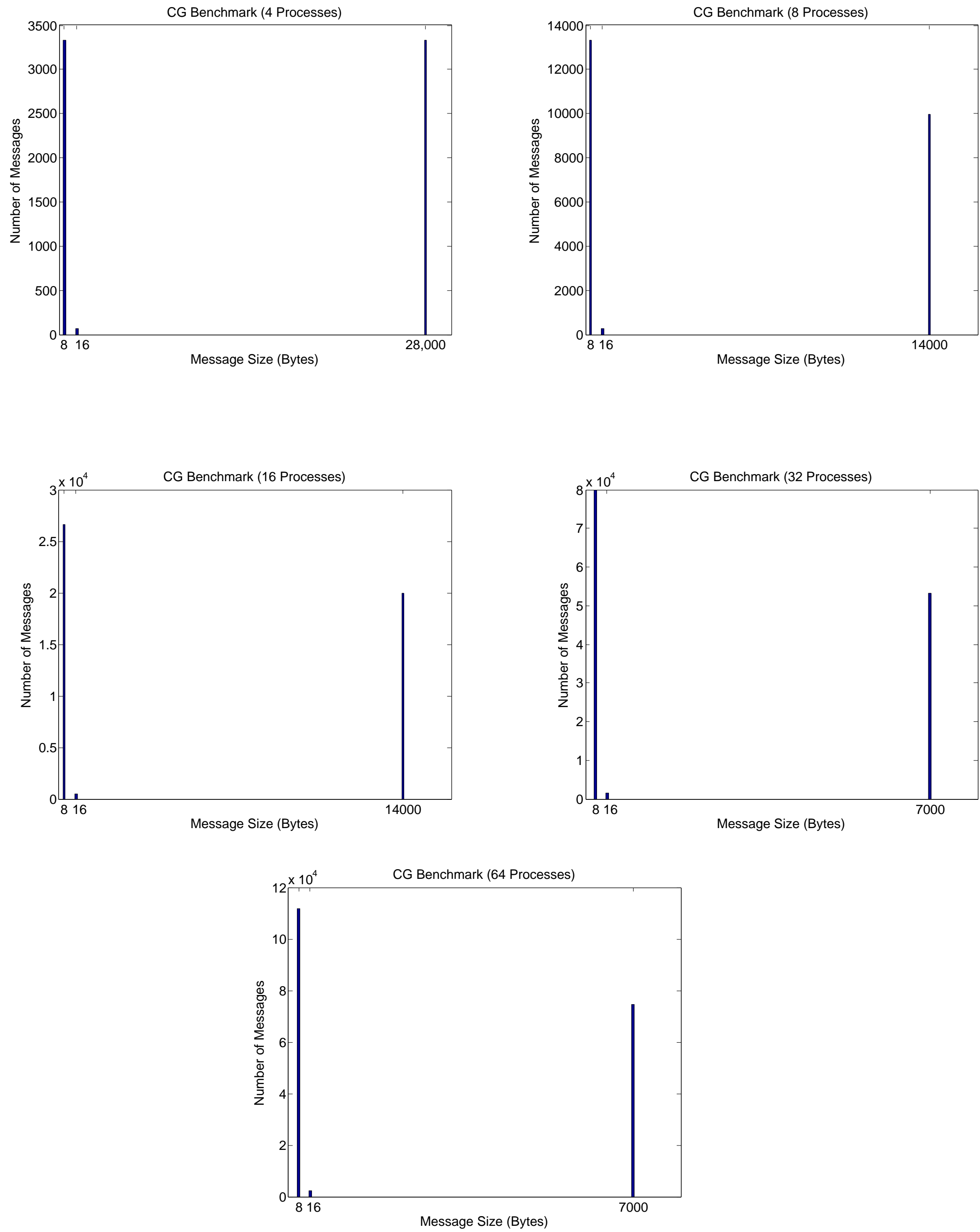

Figure 4.1: Message Size Distribution for CG Benchmark (Class W)

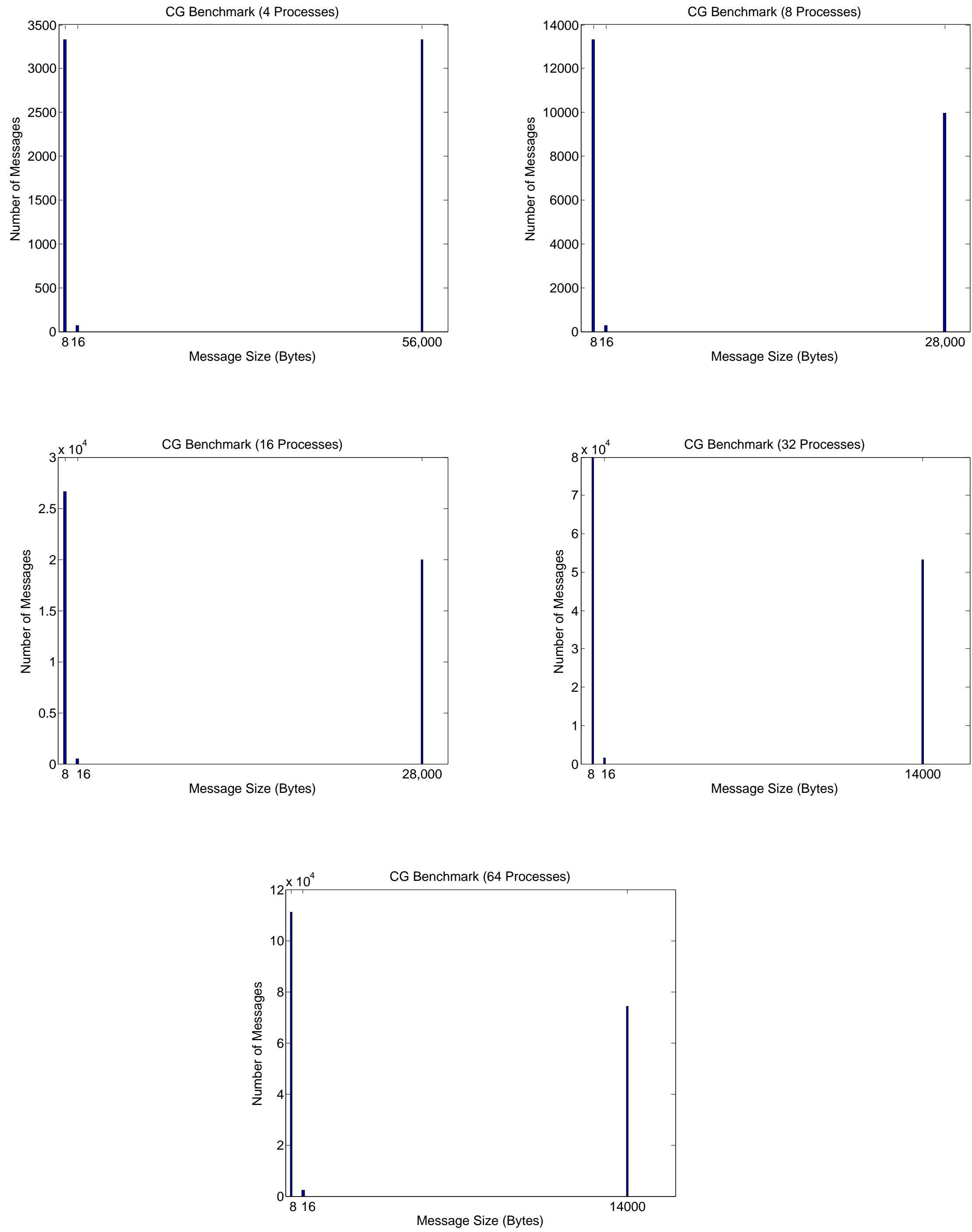

Figure 4.2: Message Size Distribution for CG Benchmark (Class A)

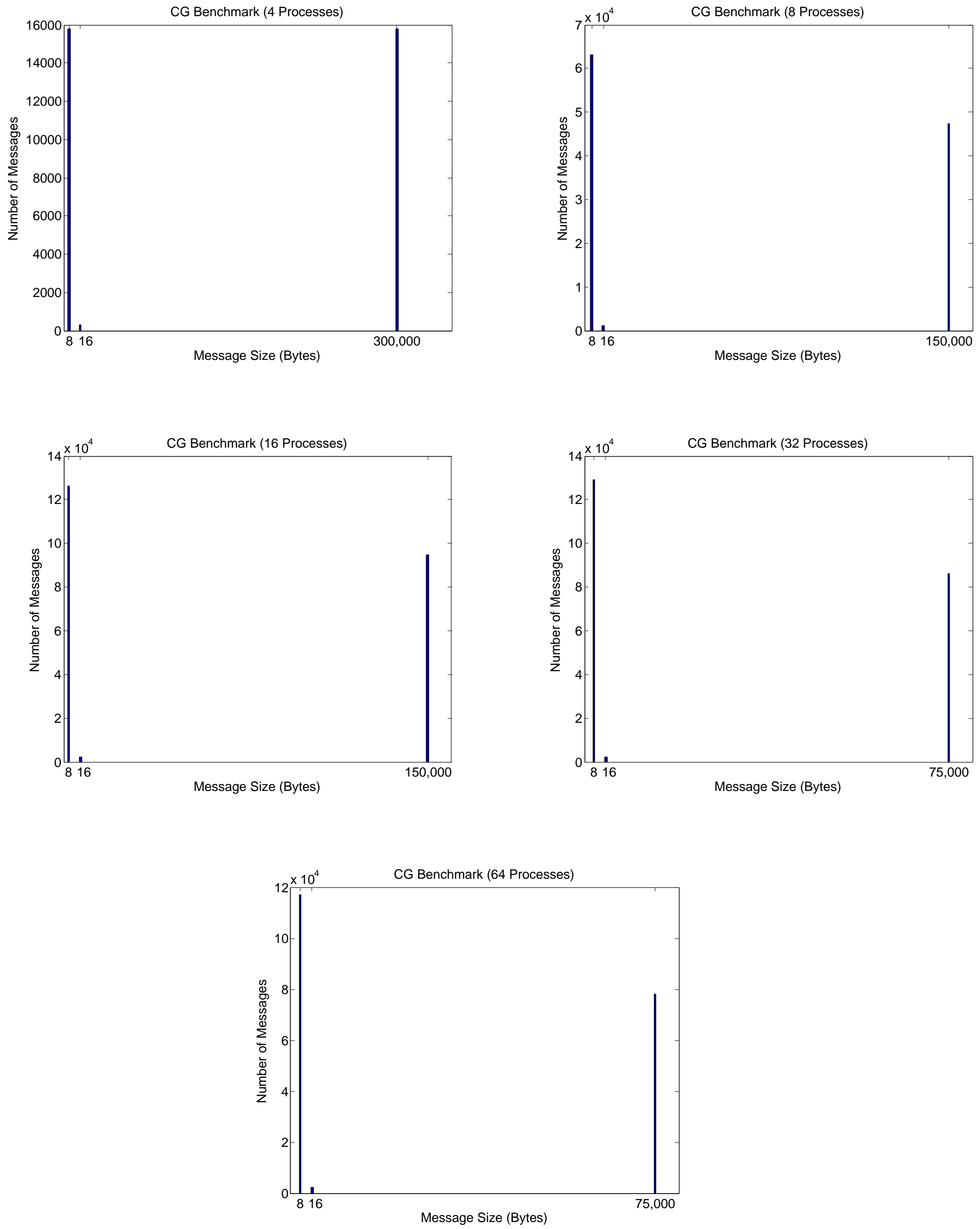

Figure 4.3: Message Size Distribution for CG Benchmark (Class B)

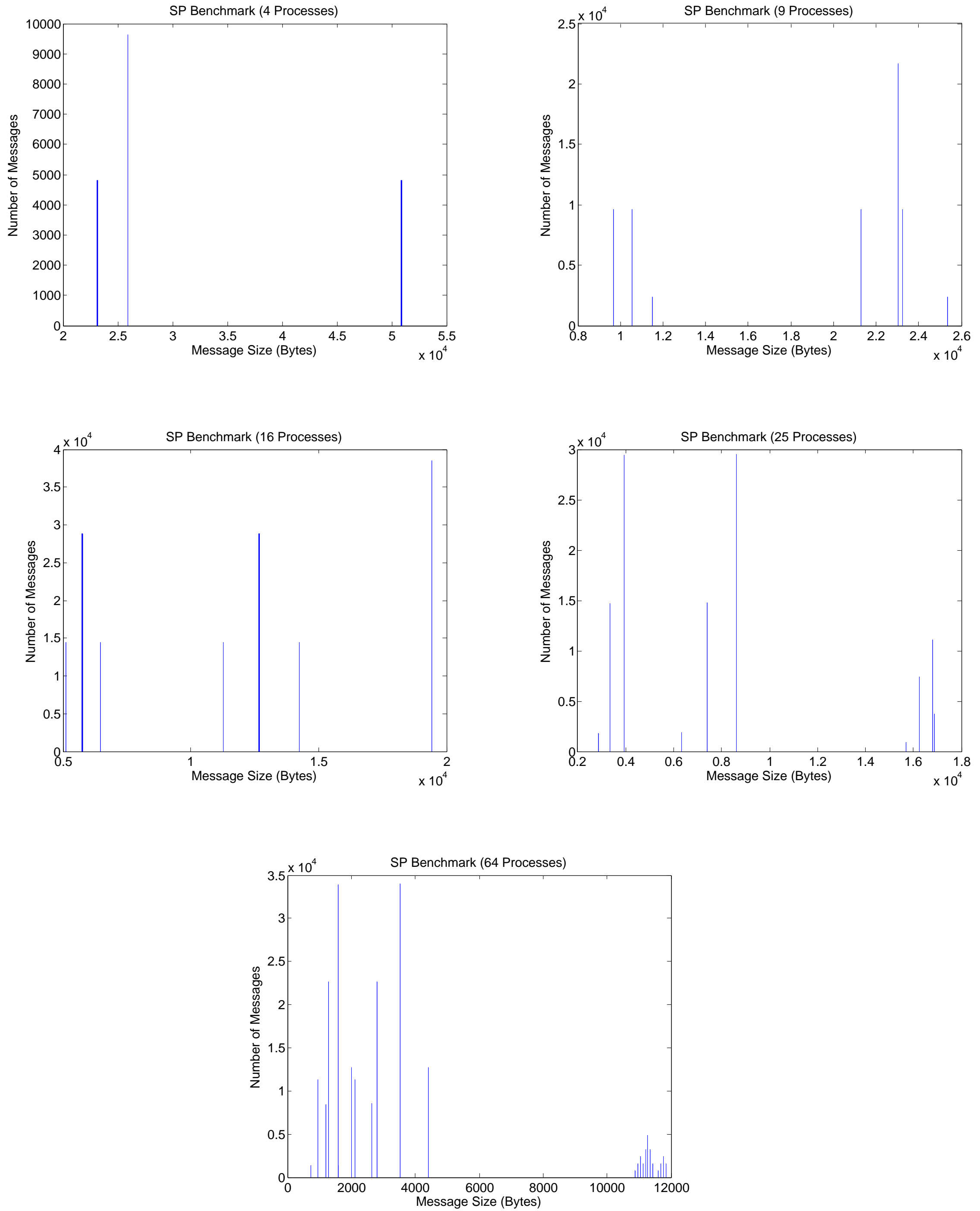

Figure 4.4: Message Size Distribution for SP Benchmark (Class W)

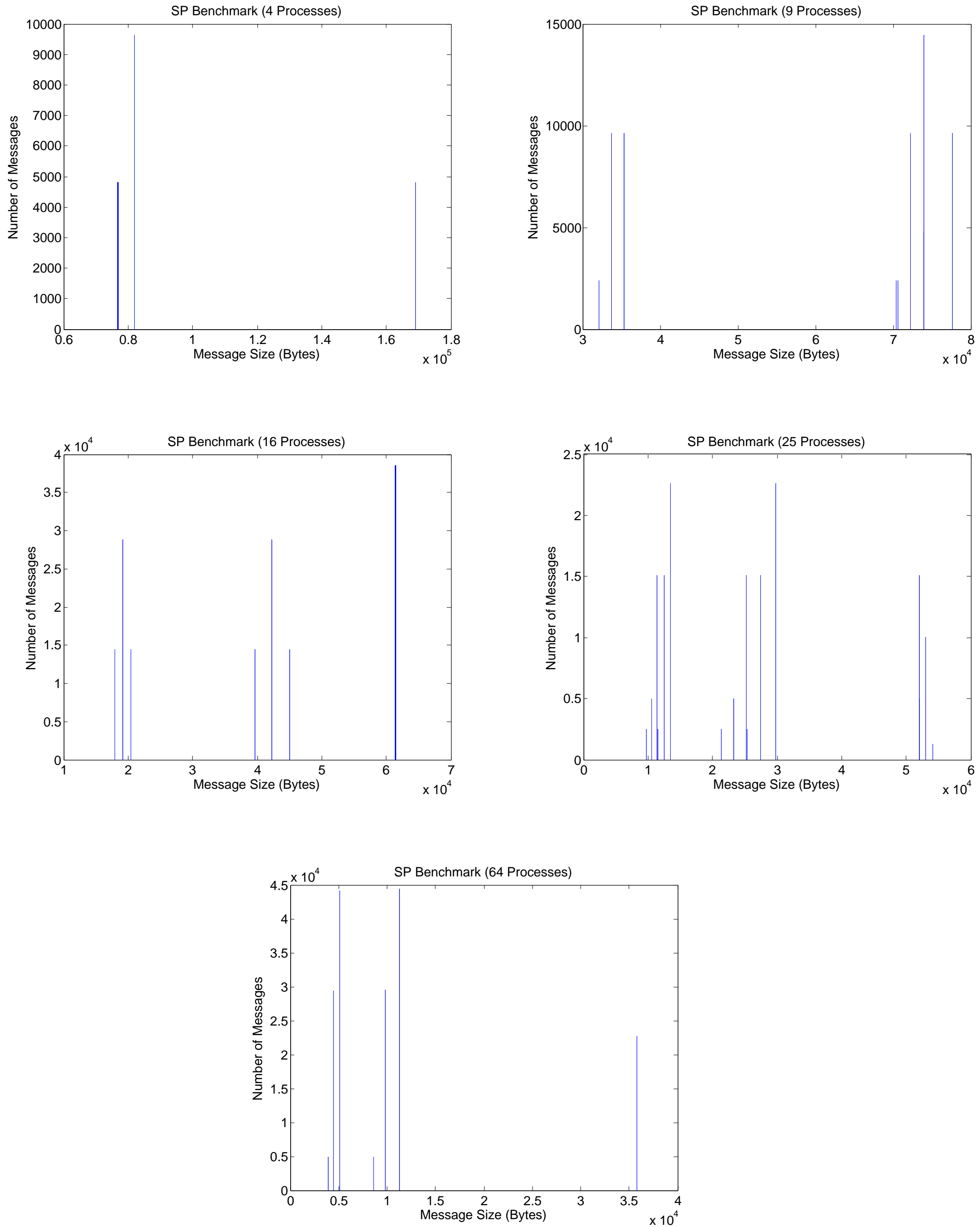

Figure 4.5: Message Size Distribution for SP Benchmark (Class A)

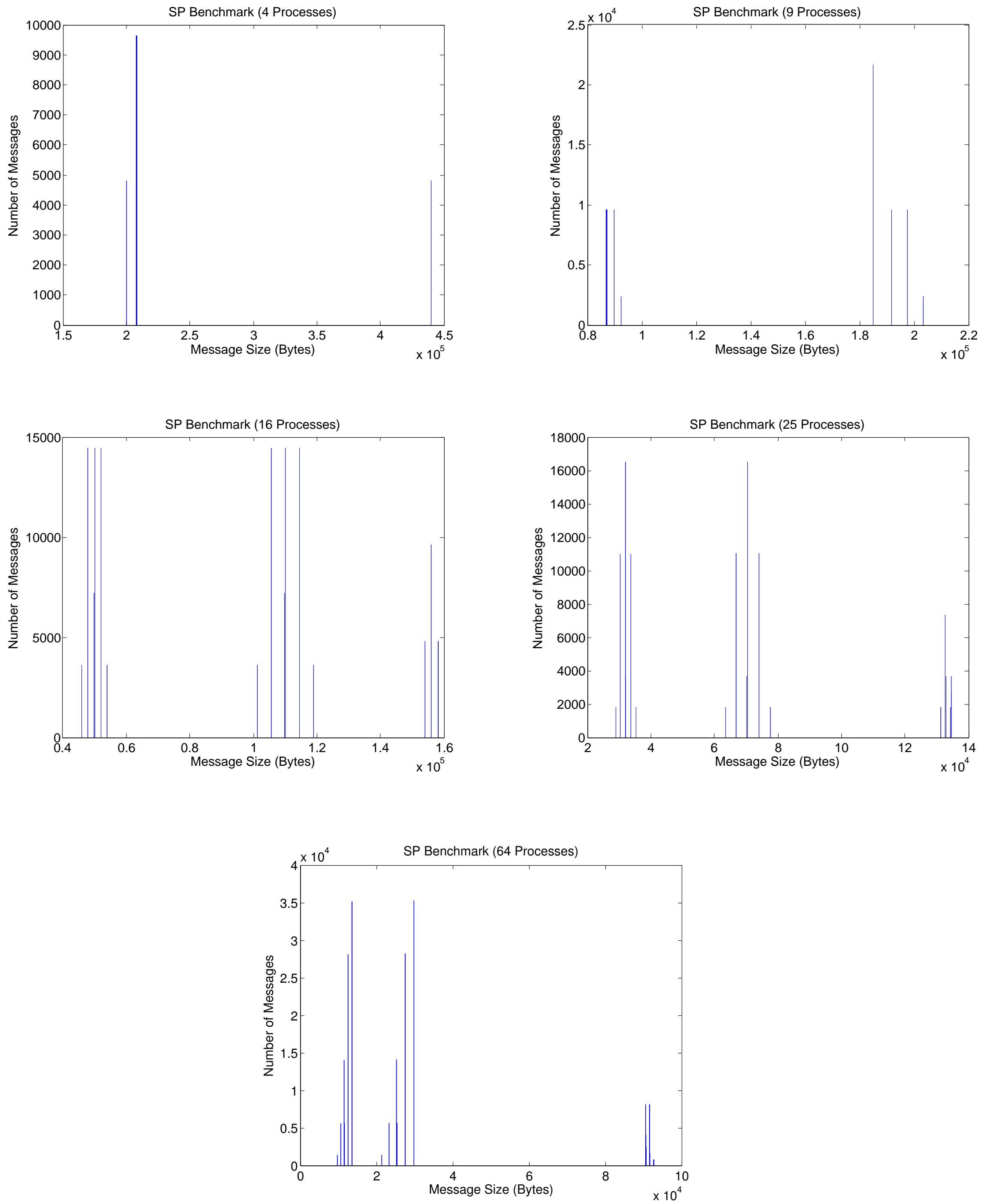

Figure 4.6: Message Size Distribution for SP Benchmark (Class B)

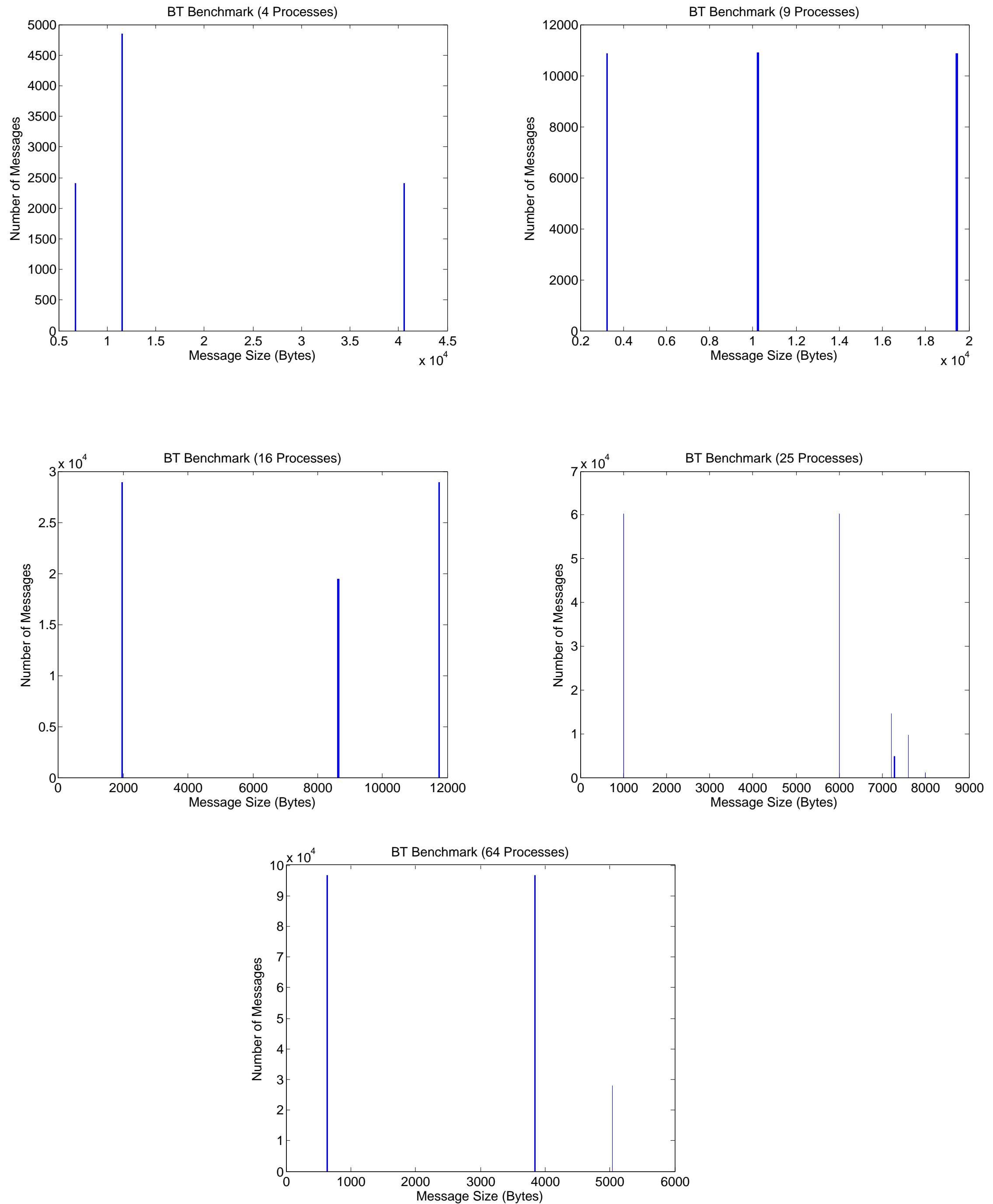

Figure 4.7: Message Size Distribution for BT Benchmark (Class W)

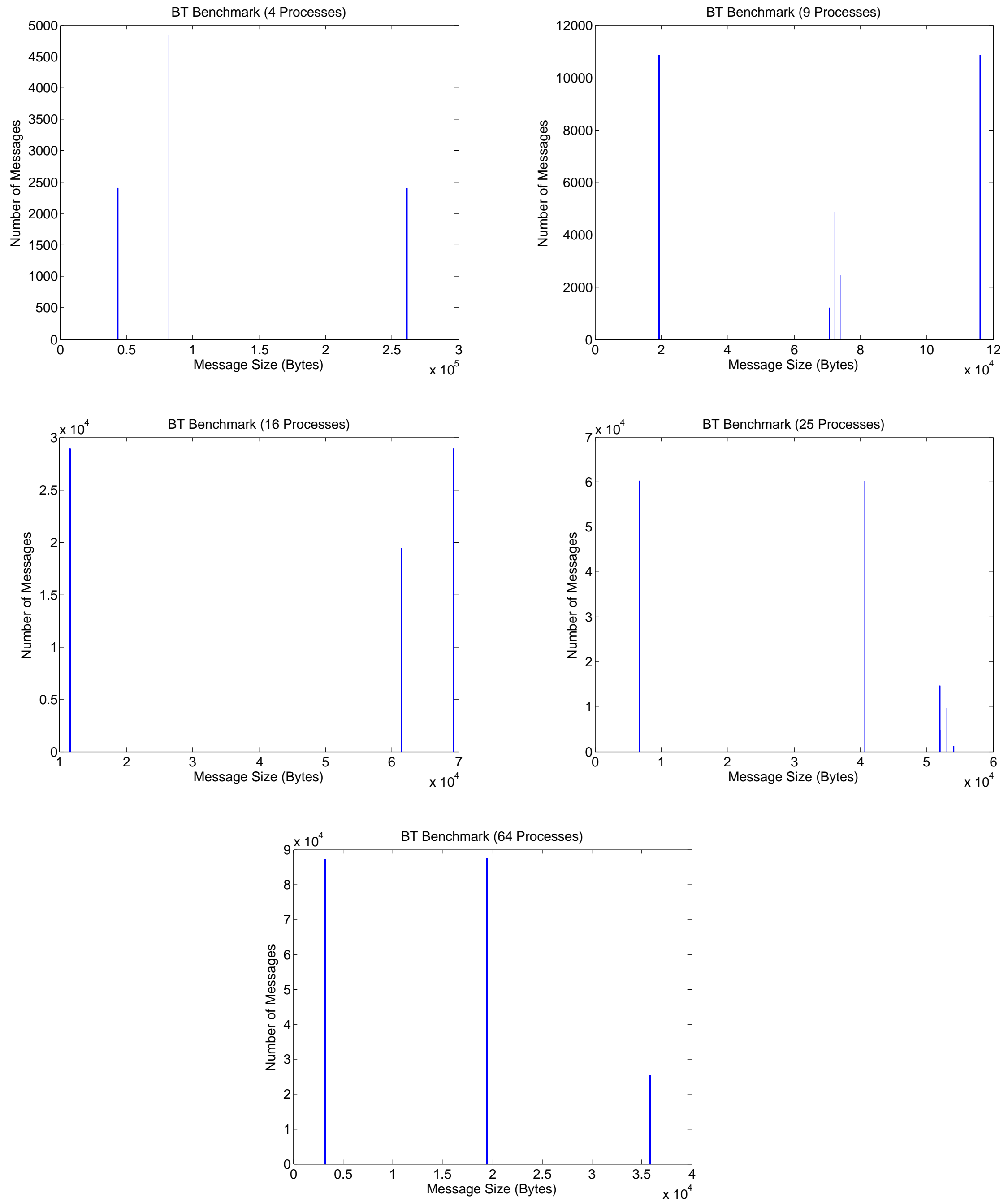

Figure 4.8: Message Size Distribution for BT Benchmark (Class A)

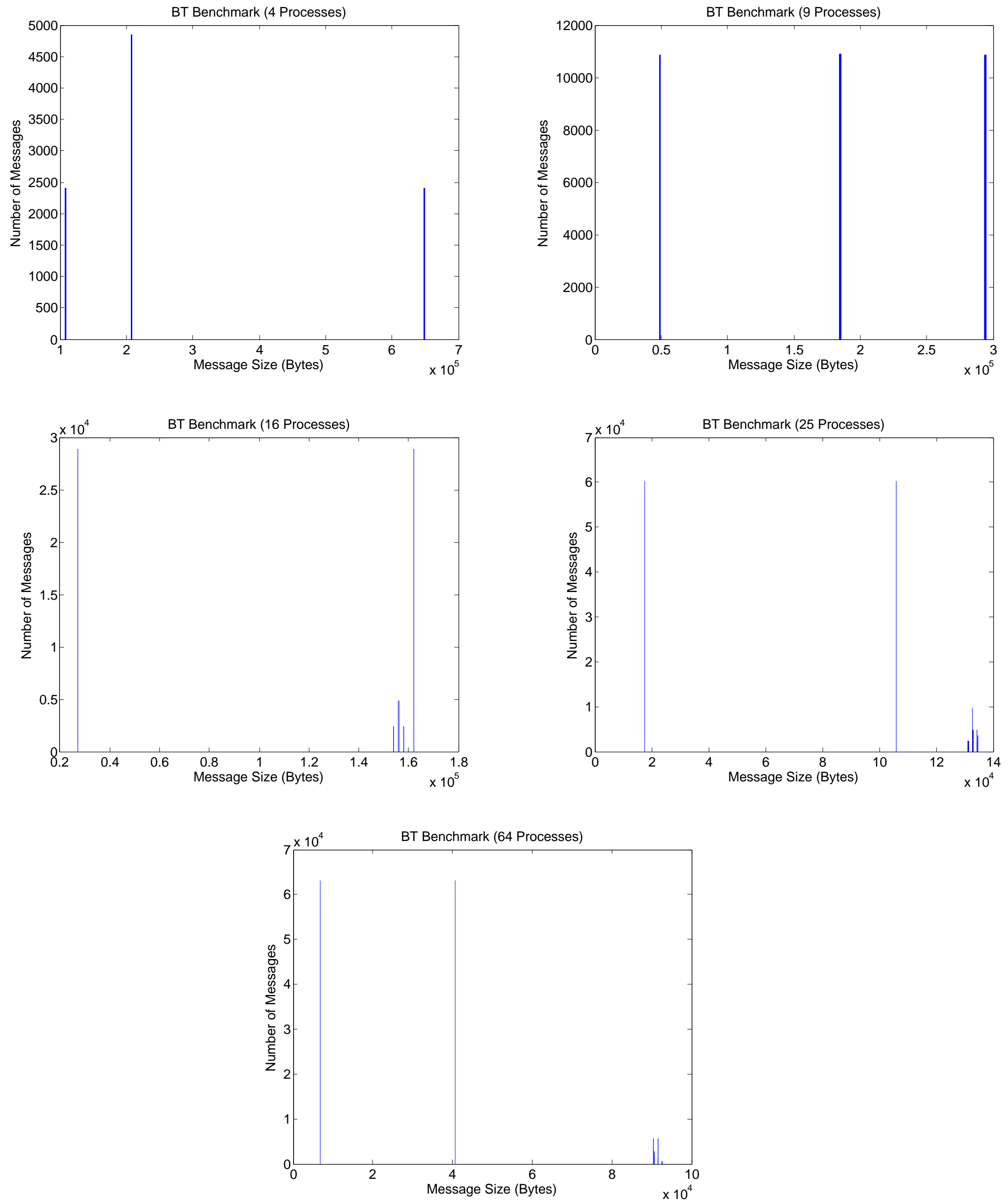

Figure 4.9: Message Size Distribution for BT Benchmark (Class B)

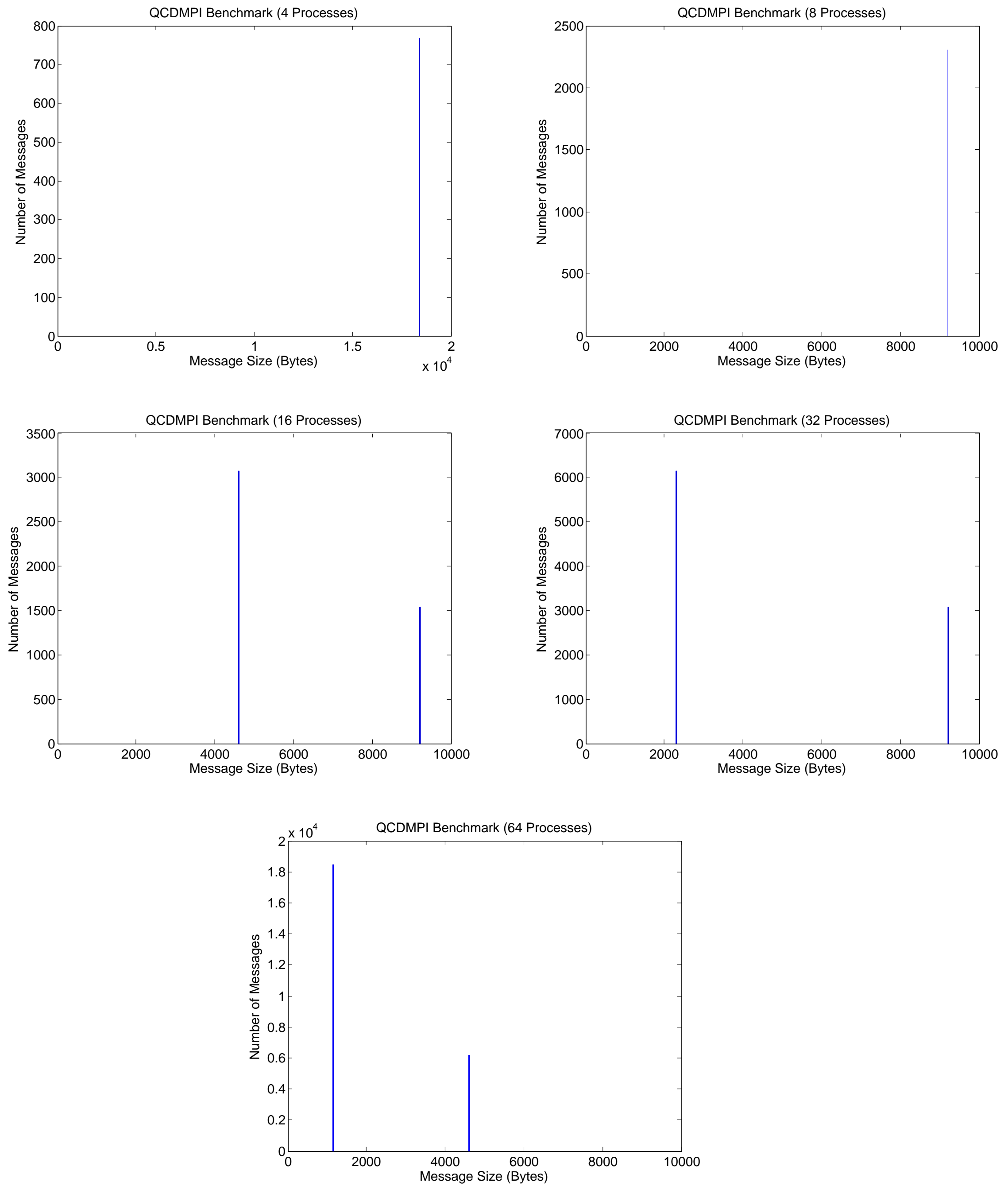

Figure 4.10: Message Size Distribution for QCDMPI Benchmark

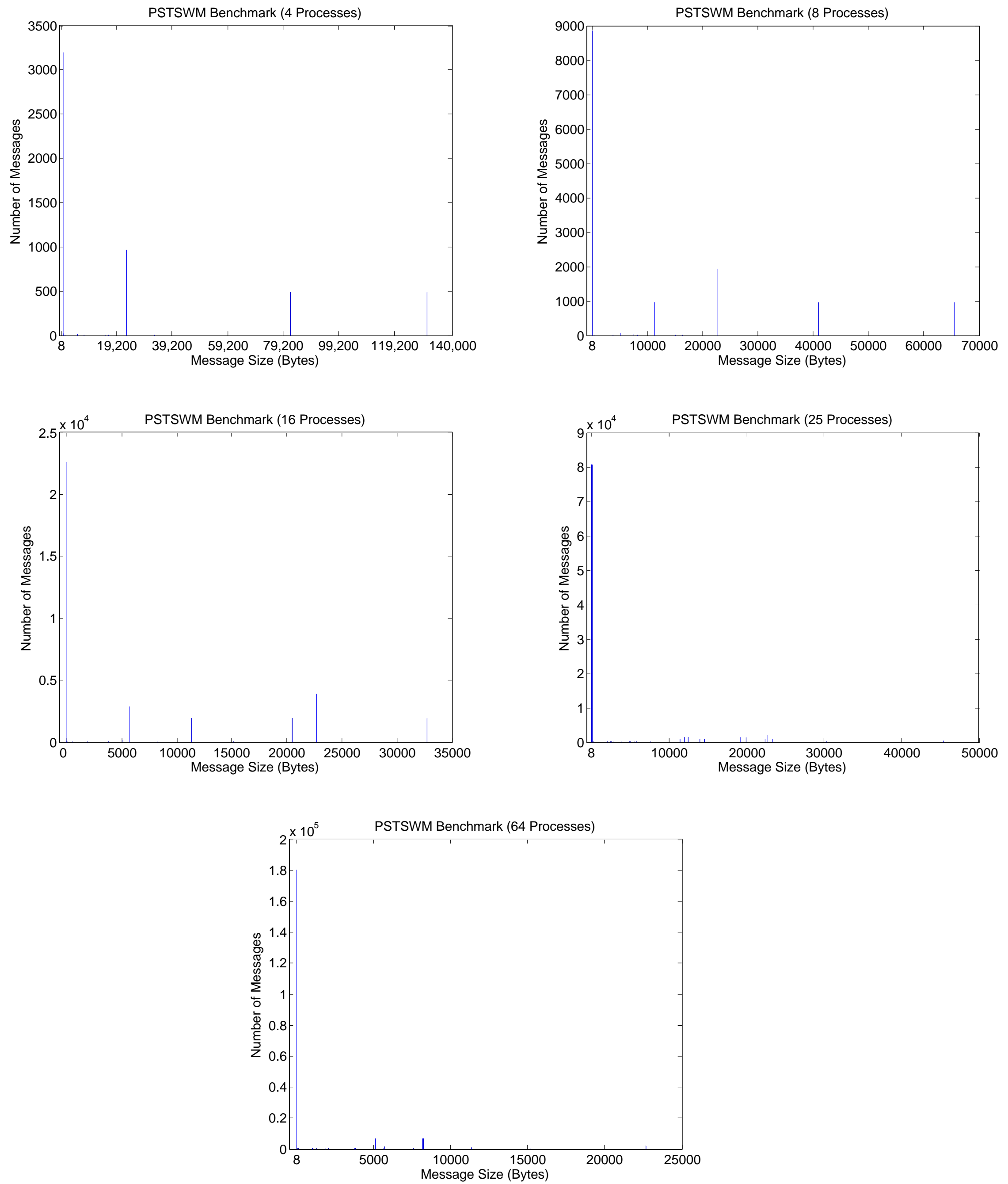

Figure 4.11: Message Size Distribution for PSTSWM Benchmark

4.5 SimpleScalar Infrastructure

The SimpleScalar infrastructure was selected to model and verify the proposed architecture. The SimpleScalar toolset provides an infrastructure for simulation and architectural modeling. The toolset can model a variety of platforms ranging from simple unpipelined processors to detailed dynamically-scheduled microarchitectures with multiple-level memory hierarchies [27]. For users with more individual needs, SimpleScalar offers a documented and well-structured design, which simplifies extending the toolset to accomplish most architectural modeling tasks.

This infrastructure provides a cycle-accurate simulation, which allows for a precise notion of time in running parallel benchmarks. In addition, because of its extensibility, we were able to modify its cache data structure to implement the network cache.

A network processor [79] approach was used in supporting network features by adding new instructions to the ISA. SimpleScalar also provides the ability to add new instructions without changing the compiler codes. The decoding section of the processor pipeline was changed by adding codes to detect the added MPI instructions and to take appropriate actions.

The original implementation of SimpleScalar is a single-threaded simulator that executes only one thread. Since the simulator is required to test and examine the performance of a processor that communicates through message passing with other processes, a connection to the network should be simulated. Therefore, the environment to test the proposed architecture needs to have at least two different threads working simultaneously, one for simulating the network component responsible for communicating messages with other processes in the network and another for running the processor that executes the application as shown in Figure 4.12. The following steps were taken to make this approach possible.

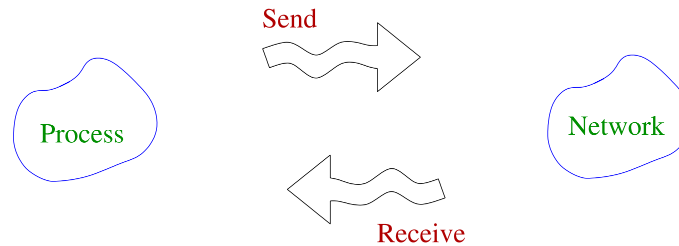

Figure 4.12: Process and Network Communication Method

4.6 Implementation

As stated previously, the SimpleScalar infrastructure was selected to model and verify the proposed architecture due to its cycle accurate simulation. Because of the large computational costs involved when one chooses to simulate a complete cluster environment, we elected to simulate only one processor while the network traffic was to be provided by a separate entity. Thus the simulation environment incorporates two threads working simultaneously.

4.6.1 Benchmark Programming Methodology

In this study, communication traces of BT, CG, and SP from the NAS suite were obtained, along with PSTSWM and QCDMPI. These benchmarks are organized in a master/slave configuration. A general model of such a configuration is shown in Figure 4.13.

In the first stage of investigation, the following steps were taken. First, the benchmarks were run on the University of Victoria's 128 processor IBM RS/6000 SP and time and payload traces were collected for the MPI calls and the sent and received messages. Then the master process was executed alone while it was supplied with the network traffic as recorded when the complete benchmark was run on the

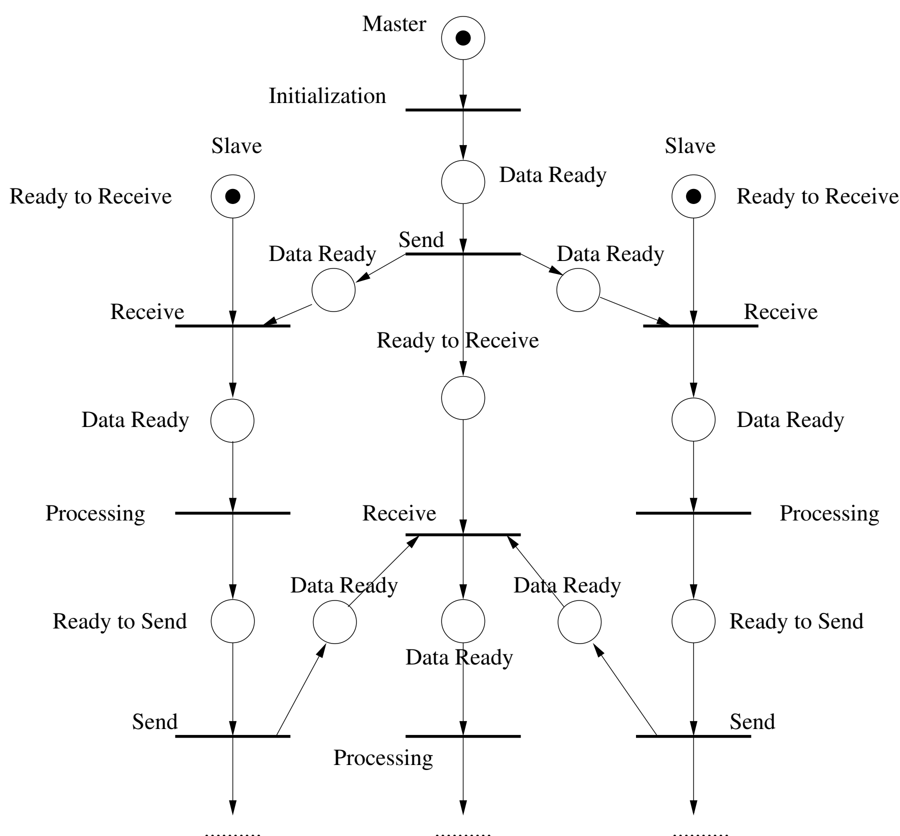

Figure 4.13: Petri net model for a typical Master/Slave organization

IBM facility. The master processes ran both natively on the IBM facility and under simulation on the SimpleScalar environment. This was done to validate our approach and simulation environment.

Having collected the required messages, their endianism was converted to ensure architecture compatibility. The necessary MPI library functions were implemented in SimpleScalar ensuring they obtained the necessary data from the collected message traces. The sim-fast simulator from the SimpleScalar simulation suite was then selected for this part of study. This high-speed simulator was chosen because the main objective at this stage was the correctness and functionality of the simulator. Finally, we collected the computational results of the master process and compared

them across the various platforms (that is, the results when the full benchmark was run on the IBM facility, the results when the master process was run on the IBM facility, and under simulation).

Table 4.1 shows that the results of running benchmarks on SimpleScalar are almost the same as those on the IBM RS/6000 SP facility. These results are relative to those obtained by executing the full benchmark on the IBM facility and pertain to executing the master process on SimpleScalar. In particular, the computational results of the CG benchmark diverge after the second iteration out of 15 (that is, class A). This discrepancy arises from the data values used in the CG benchmark and the different implementations of functional units on two different architectures. In this benchmark the data are too small and the main operations are divisions, which are imprecise. Nevertheless, the number of messages received before the results begin to diverge, is significant (around 1130) and this allowed us to test the *Network Processor* extensions.

As stated, the master processor is executed on SimpleScalar as the representative of the benchmark. The main reason for this selection is the fact that the master process sends and receives most messages and has an intensive communication environment. Therefore, it provides the most challenging communications environment rather than the slave processes which are mostly responsible for the computation section of a parallel application.

4.6.2 Time and Cycle Consistency on IBM RS/6000 SP and SimpleScalar

An important issue faced when developing the simulation methodology was to ensure timing accuracy. The traces obtained from the IBM facility expressed time as wall clock, while SimpleScalar understands time as an ordered sequence of events and

Table 4.1: Absolute and Relative Errors of results on IBM RS/6000 and on SimpleScalar

Benchmark	Absolute Err	Relative Err
CG before divergence	8.02E-13	4.68E-14
CG after divergence	2.86E+00	0.16E+00
BT	2.10E-16	4.95E-17
SP	1.21E-14	1.11E-10
PSTSWM	4.22E-16	1.79E-19
QCDMPI	0.00E+00	0.00E+00

expresses it in terms of cycles. Since the PowerPC architecture used by the IBM facility is different from the MIPS/DLX architecture used by the SimpleScalar, we needed to find a correspondence between the wall clock unit and the simulation cycle. Two methods were employed to accomplish this. The first was a statistical method based on the comparison of the number of simulation cycles and the time elapsed on the IBM RS/6000 SP facility at several fragments of code [74].

The second method was based on simulating the benchmark completely with the messaging environment and varying the relation between one simulation cycle and one unit of wall clock time, effectively increasing or decreasing the average message inter-arrival time. We then choose the correspondence between wall clock time and simulated cycle time so that the messaging environment delivered the messages (relative to the computation stage) at the same points under simulation and under the original runs on the cluster [71, 72]. These techniques are elaborated in Chapter 5.

4.7 Summary

This chapter has explained the simulation environment, including benchmarks, simulator infrastructure, and timing concerns. Preparation of this infrastructure was the key to investigating the effectiveness of the proposed techniques. It was illustrated that the established infrastructure can accurately simulate the aforementioned benchmarks; thus, enabling the design of the extensions and experiments to examine their behavior. In subsequent chapters, this infrastructure is employed to investigate the performance of the *Network Cache* and associated transfer techniques. For this, their effects are first explored using the normal operation of the processor's cache system. Then, the optimum configurations of the proposed extensions are examined.

Chapter 5

Characterizing the Caching Environment

This chapter focuses on concerns related to the caching environment. Specifically, problems related to the size and associativity of the network cache and their relationship to message traffic were investigated. Additionally, the impact of message traffic on the data cache was investigated.

The proposed extensions and associated data transfer mechanisms as described in Chapter 3 were implemented and then explored as to whether our techniques interfered with the normal operations of the data cache. As a preliminary study, the CG benchmark was selected to explore the efficacy of this approach. Subsequently, investigations were extended by employing another standard benchmark (PSTSWM). As stated, these benchmarks have a multitude of short messages that can effectively test the benefits of the proposed extensions.

An important concern while developing the simulation methodology was to ensure timing accuracy. The traces obtained from the IBM facility express time as wall clock, while SimpleScalar understands time as an ordered sequence of events and expresses it in terms of cycles. Moreover, because the PowerPC architecture used by the IBM facility is different from the MIPS/DLX architecture used by SimpleScalar we needed to find a correspondence between the wall clock unit and the simulation cycle. Two

methods were employed to accomplish this. The first was a statistical method based on the comparison of the number of simulation cycles and the elapsed time on the IBM RS/6000 SP facility at several fragments of code [74, 73]. The second was based on simulating the benchmark completely with the messaging environment [71, 72, 75, 76]. The details of these methods are discussed in subsequent sections.

5.1 Time and Cycle Consistency Using a Statistical Approach

To find a correspondence between the wall clock unit and the simulation cycle we first employed a statistical approach. This was accomplished by comparing the number of simulation cycles and the elapsed time for several pieces of code of varying length and devoid of system or MPI calls. The selected sections were parts of the main loop, which is executed 15 times in the CG benchmark Class A. The collected values belong to different iterations of the main loop, which receives different sets of data in each iteration, at three locations.

We used the achieved values in order to establish a relation between simulation cycles and wall clock unit on IBM RS/6000 SP. We achieved this using a statistical approach. For this purpose, we established a statistical relation between simulation cycles and wall clock unit on IBM RS/6000 SP as illustrated in Figures 5.1 and 5.2. As can be seen, this relation is a linear one, and it holds whether the compilation is optimized or not.

As a result of establishing this relation we were able to calculate the cycles the data needs to be available in order to run the application, based on the send and receive time collected from running the benchmarks on the IBM RS/6000 SP. In this case the simulated messaging system simply injects the appropriate message and

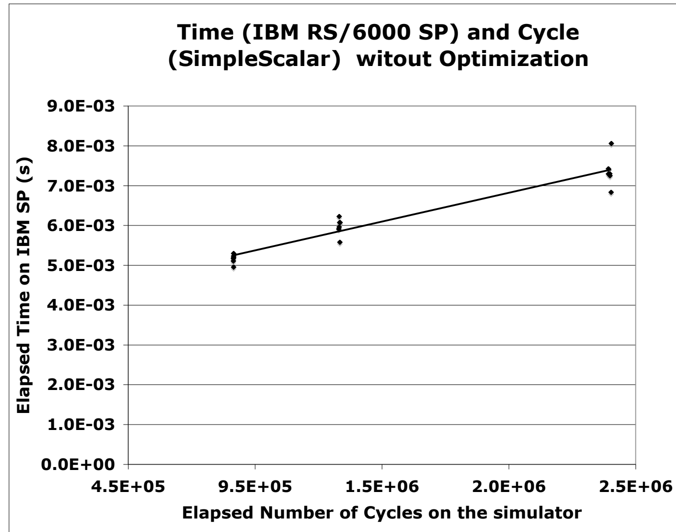

Figure 5.1: Number of Cycles on SimpleScalar vs. time on IBM RS/6000 SP for the CG Benchmark (code was compiled with standard optimization)

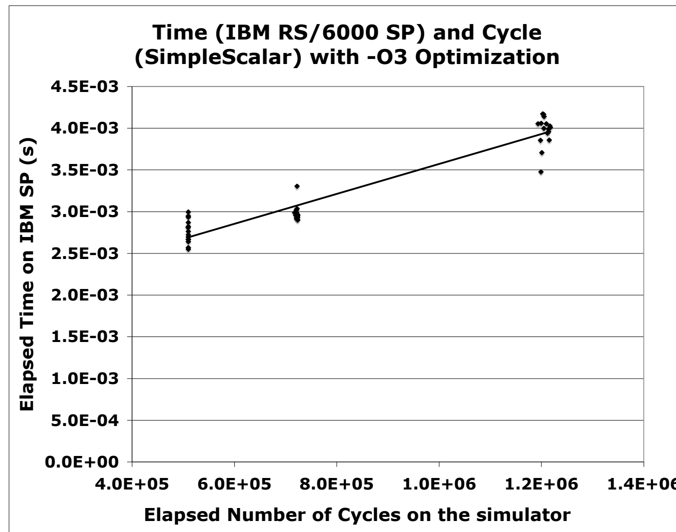

Figure 5.2: Number of Cycles on SimpleScalar vs. time on IBM RS/6000 SP for the CG Benchmark (code was compiled under -O3 optimization)

payload into the proposed network cache at the cycle that corresponds to the wall clock time of the message as measured at the IBM RS/6000 SP system.

In order to check the impact of the proposed extension on the data cache, different policies were considered in transferring data from the NI into the memory. These configurations can be categorized as below:

- The processor is responsible for transferring the data from the network interface memory to process memory.
- The processor uses DMA to get data from the NI into a buffer and then transfers it to its final destination in process memory.
- The processor uses DMA to transfer data directly into the process memory.
- The processor uses the proposed extension to transfer data into the data cache (short messages) or the processor memory (long messages).

For the experiments reported in this section we chose the size and associativity as reported in Table 5.1. It should be pointed out that the main focus here is to examine the impact of the proposed extension on the caching system, and the following parameters selected for the different hardware components (data cache, network cache, and so on) are not optimal. The following section explores the optimal configuration for the network cache.

The main question needing to be answered here was whether the binding process would cause excessive replacements in the data cache. As in the DTCT approach, the data is transferred into the data cache upon binding; however, it is more likely to interfere with the normal operation of the processor's data cache. As a result, here, the investigation focused on the DTCT approach.

In moving the message payload from the network cache to the data cache we ensured the consistency of the cached lines with the data in memory. A potential

Table 5.1: Simulator configuration

IL1 Cache	16K 4-way set-assoc., 32B blocks
DL1 Cache	16K 4-way set-assoc., 16B blocks
Network Cache	8K 8-way set-assoc., 16B blocks
Message Cache	4K 4-way set-assoc.
Process Cache	4k 4-way set-assoc.

consistency problem arises from the fact that the message payload may be bound at an arbitrary process address that may not coincide with block boundaries determined by the length of the data cache line.

The proposed mechanism was applied only to short messages (less than or equal to 16 bytes). Longer messages were handled in the traditional way; they were copied to their final process buffer destination upon issuing the corresponding receive call. A preliminary hardware implementation of the network cache design discussed here was presented in [20].

The results of the simulations are listed in Table 5.2 for both the 4-processor and 64-processor cases. The first entry in each case is the value of the referred quantity (cache misses) when we use the proposed extension, while the remaining entries are the increments of the corresponding values with reference to those obtained by the proposed extension. A positive number signifies that the corresponding quantity is larger than that obtained when using the proposed extension.

It can be inferred from these results that the number of cache accesses and cache misses are smallest when we use the extension to transfer and bind the received messages directly into the processor's data cache. This is true for both the 4-processor and 64-processor environments. The 64-processor environment has a more intensive communication pattern, with more messages exchanged between processors

and less computation time between communication events. These results lead us to believe that it is possible to place a message directly into the data cache during the receive operation so that it will be used by the consuming process as soon as it is needed. The results presented show that the proposed extension has fewer cache misses, replacements, invalidations, and writebacks, meaning a better cache behavior in comparison with other approaches. The achieved cache statistics also imply that placing the message into the data cache during the receive operation does not interfere adversely with the caching environment for the remainder of the computation. In addition, it is observed that most of the bound messages persist in the cache long enough to be used by the consuming process since most of the arrived messages were accessed without a cache miss the first time.

Table 5.2: Comparison of different approaches

Approach.nP	Data Cache Accesses	Cache misses	Writebacks	Replacements
Our Extension.4P	330153832	70009051	3457029	70008891
All DMA.4P *	0	239	239	0
DMA and Processor.4P *	239	478	239	0
All Processor.4P *	2428	1471	1070	1376
Our Extension.64P	202538696	23822152	13356760	23821640
All DMA.64P *	771	2133	2133	2537
DMA and Processor.64P *	3311	4673	2133	2537
All Processor.64P *	8098	9590	1971	3532

*Values are incremental with respect to the corresponding values in the first row.

5.2 Establishing the Message Traffic Intensity

The first approach, as explained, explored a statistical method to find a correspondence between the number of simulation cycles and the elapsed time on the IBM RS/6000 SP facility. The second method established a message traffic intensity by varying the relation between one simulation cycle and one unit of wall-clock

time, effectively enabling an increase or decrease in the average message inter-arrival time [72, 75]. The main objective of establishing this new message traffic intensity was to simulate the designed extension in various message inter-arrival times. Using this approach, we could explore how the messages-arrival times impact on the caching structure.

Figure 5.3 depicts the termination time of the simulation (expressed as the number of simulation cycles) versus the number of simulation-cycles per wall-clock-unit time. We denote the number of cycles per wall-clock-unit time as r. As can be seen, the termination time increases gradually as the number of cycles per wall-clock-unit time increases, exhibiting a knee when the cycle per unit time is such that the average computational speed of the simulated system matches with that of the PowerPC.

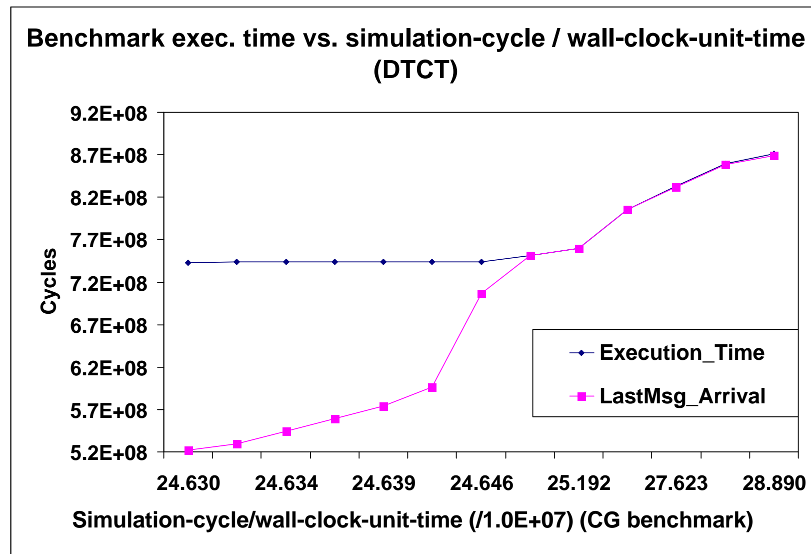

Figure 5.3: Execution time vs Simulation-cycle/Wall-clock-unit-time *(DTCT)* CG Benchmark (Class A)

As stated, the main objective of the current approach was to simulate the designed extension in various message inter-arrival times. Using this approach we were able to explore how the messages' arrival times impact on the caching structure. In subsequent simulations the obtained value of cycles per-unit-time were used to establish the timing of the message sequence. We also ran the simulations assuming slightly increased/decreased cycles per unit-time value to explore messaging environments with different inter-arrival times. Doing so effected whether the messages arrived earlier or later with regard to their respected receive requests and enabled an exploration of the impact on the caching structure.

The impact of our method on the data cache was compared with the impact of the methods listed below.

- The processor is responsible for transferring the data from the network interface memory to the process memory.

- The processor uses DMA to get data from the NI into a buffer and then transfers it to its final destination in process memory.

- The processor uses DMA to transfer data directly into the process memory.

- The processor uses the proposed extension to transfer data into the data cache (short messages) or the processor memory (long messages).

The results of the simulations are presented in Table 5.3. The results of three experiments are listed. The first assumed a cycle/time ratio of 25.19E07 corresponding to the knee of the curve of Figure 5.3 depicting the situation where both the simulated and the PowerPC processors are roughly equivalent. The second environment set the cycle/unit time ratio corresponding with the situation where messages arrive earlier than the intended time, while the last experiment set the

cycle/unit time ratio corresponding to the situation where messages arrive later than their intended time.

The first entry in each case is the value of the referred quantity (cache misses) when using the proposed extension, while the remaining entries are the increments of the corresponding values with reference to the values obtained by the proposed extension. A positive number signifies that the corresponding quantity is larger than that obtained when using the extension.

Table 5.3: Dependence of the data cache response on the different transferring mechanism and traffic intensity

Approach.64P	Data Cache Accesses	Cache misses	Replacements	Writebacks	Invalidation
(cycle/unit-time = 24.89E07)					
Our Extension	110802247	5905966	5914321	1316548	428
All DMA*	103342	16612	69730	32308	3764
DMA and Processor*	102619	16866	69752	32473	3764
All Processor*	101799	16849	66491	30194	1706
(cycle/unit-time = 25.19E07)					
Our Extension	110991056	5915987	5911328	1258070	563
All DMA*	82077	15610	70029	38156	3629
DMA and Processor*	82331	15864	70029	38321	3629
All Processor*	77753	15699	65197	38025	1571
(cycle/unit-time = 26.69E07)					
Our Extension	110890870	5900534	5896230	1316532	432
All DMA*	99896	16883	70467	31250	3760
DMA and Processor*	99856	17410	71539	32474	3760
All Processor*	96279	17645	66707	32179	1702

*Values are incremental with respect to the corresponding values in the first row.

As can be inferred from these results, regardless of the inter-arrival time of messages the number of cache accesses and cache misses are smallest when we use the proposed extension to transfer and bind the received messages directly into the processor's data cache. The 64-processor environment has a more intensive

communication pattern with more messages exchanged between processors and a lower computation time between communication events. The results presented show that the proposed extension has fewer cache misses, replacements, invalidations, and writebacks, meaning a better cache behavior in comparison to other approaches. Therefore, and similar to the previous experiment (statistical approach), these results lead us to believe that it is possible to place a message directly into the data cache during the receive operation so that it will be used by the consuming process as soon as it is needed. The achieved results also imply that placing the message into the data cache during the receive operation does not interfere adversely with the caching environment for the remainder of the computation.

5.2.1 The Impact of the Size of the Network Cache

In this section the impact of the size of the network cache and its associativity is studied. We simulated utilizing arrival ratios $r=$ 24.62E07, 24.639E07, 24.89E07, 25.19E07, and 27.62E07 to represent message sequences that arrive at their intended time ($r=$ 25.19E07), as well as earlier ($r=$ 24.62E07, 24.639E07, 24.89E07) and later ($r=$ 27.62E07). Different network cache sizes and associativity were employed, while the *message ID* cache was kept large (8 KB 16-way set-associative) so as not to interfere with the behavior of the network cache. In other words, the size of the *message ID* section is selected so that no replacement occurs in this cache section.

Given that the message distribution in the sets of the network cache depends on the allocated buffer, we studied the impact of the message buffer selection policies. For these experiments we used message buffers that were allocated on 16B boundaries within buffer pools of varying sizes. Once the buffer pool was exhausted another pool of a similar size was allocated. Once a message was bound its corresponding buffer was freed and became available for subsequent allocations.

Table 5.4: Network cache misses: messages arrive early (r= 24.62E07)

All short messages	2544	2544	2544	2544	2544	2544	2544	2544
Early arrived Messages	2541	2541	2541	2541	2541	2541	2541	2541
	Netcache Configuration[1]							
	16:2	16.4	32:2	32:4	64:2	64:4	128:2	128:4
Memory buffer (16B elements)	Number of Replacements							
4KB buffer	2232	2088	2064	1859	1807	1391	1281	272
256B buffer	2232	2088	2064	1859	1807	1391	1281	272

The result of these simulations are depicted in Tables 5.4 through 5.7. For the r = 27.62E07 case in which all the messages arrive late, the results showed that all the entries of the network-cache performance table were zero. This behavior was due to the late arrival of the messages; therefore, there is no need to cache the arrived message into the network cache. The number of network cache replacements is presented in terms of the network cache configuration and the buffer pool size. As can be seen, a rather small network cache with wide associativity achieves zero replacements, while the impact of the size of the message pool is insignificant.

Table 5.5: Network cache misses: (r= 24.639E07)

All short msgs.	2544	2544	2544	2544	2544	2544	2544	2544
Early arrived Msgs	2065	2065	2065	2065	2065	2065	2065	2065
	Netcache Configuration							
	2:1	2:2	4:2	4:4	8:2	8:4	16:2	16:4
Memory buf. (16B elements)	Number of Replacements							
4KB buffer	1796	941	224	108	44	0	0	0
256B buffer	1796	941	224	108	44	0	0	0

[1]The configuration is expressed as number of sets: associativity: Thus 32:4 means a cache of 32 sets each having 4 lines of 16B for a total size of 32 * 4 * 16 = 2KB.

Table 5.6: Network cache misses:(r= 24.89E07)

All short msgs.	2544	2544	2544	2544	2544	2544	2544	2544
Early arrived Msgs	1567	1567	1567	1567	1567	1567	1567	1567
	Netcache Configuration							
	2:1	2:2	4:2	4:4	8:2	8:4	16:2	16:4
Memory buf. (16B elements)	Number of Replacements							
4KB buffer	11	0	0	0	0	0	0	0
256B buffer	11	0	0	0	0	0	0	0

Table 5.7: Network cache misses: messages are consumed immediately (r= 25.19E07)

All short msgs.	2544	2544	2544	2544	2544	2544	2544	2544
Early arrived Msgs	1058	1058	1058	1058	1058	1058	1058	1058
	Netcache Configuration							
	2:1	2:2	4:2	4:4	8:2	8:4	16:2	16:4
Memory buf. (16B elements)	Number of Replacements							
4KB buffer	9	0	0	0	0	0	0	0
256B buffer	9	0	0	0	0	0	0	0

5.3 Summary

This chapter presented the design of the proposed network extension, the primary purpose of which was to ensure a self-consistent design and to verify the effectiveness of the extension in handling short messages with different arrival-time combinations. It has been shown that received messages can be bound and transferred into the data cache and this does not affect the data-caching characteristics of the application regardless of the message arrival times. This is important since it allows us to proceed with the implementation of the proposed late binding mechanism; it promises to bind and transfer messages into the data cache without the intervention of expensive memory-copy operations. Moreover, it has been shown that a small network cache is sufficient to overcome expensive replacements.

The following chapter focuses on obtaining timing performance as well as the

study of two late-binding policies: DTCT and LDTCT. In addition, the introduction of another benchmark is planned in order to obtain a more complete picture of the performance of the proposed schemes.

Chapter 6

Evaluating the Data Transfer Techniques

This chapter is a further study of the caching environment and an evaluation of data transfer techniques through measuring different metrics to illustrate the effectiveness of the proposed extensions. For this, we considered several performance metrics that measure the benefits of the techniques employed. These metrics include the number of cache accesses, replacements, writebacks, invalidations, and misses in the processor's data cache. We also demonstrate how the proposed data transfer techniques provide significant reductions in the access latency for I/O intensive environments without polluting the data cache, such as message passing configurations and SMPs.

Specifically, Direct to Cache Transfer *DTCT* and *lazy DTCT (LDTCT)* data transfer techniques are evaluated. As mentioned in Chapter 3, in these techniques, a message is transferred into the network cache upon its arrival and at the same time a buffer is allocated for it in the network memory. The network tag is set to the allocated buffer address, and the message ID is set accordingly in the message ID section. Henceforth, the message lives in the network cache and is transferred to the data cache upon receiving a corresponding receive call as explained below.

The main difference between the *DTCT* and *LDTCT* techniques occurs when a receive call is issued by the running application. With the *DTCT* technique, when

that receive call is issued, if the corresponding message has already arrived, the message is retrieved from either the network cache or the network memory. Then, it is transferred into the data cache. Subsequently, the processor accesses the arrived message through the data cache.

In contrast, with the *LDTCT*, if the arrived message resides in the network cache it will remain there and its entry in the message cache section will be invalidated. The binding of this message to the process address space is accomplished by adding an entry into the process section of the network cache. Subsequently, the processor will access the data through the process cache. In other words, the processor searches the data cache and the process cache simultaneously for every data access. However, if the arrived message has been evicted from the network cache due to a replacement, it will be retrieved from the network memory and transferred into the data cache upon binding.

As stated, the *LDTCT* technique has the advantage over the DTCT of not incurring the cost of copying the message to the data cache. However, this technique has the potential to be slower in cases where the cost of copying is less than that of accessing the network cache, which involves indirection. On the other hand, the *DTCT* method transfers data from the network cache to the data cache, which can be accomplished efficiently, although this could result in a replacement in the data cache. These issues are investigated in the following sections.

For this part of the investigation, an additional benchmark (PSTSWM) has been employed to examine the implemented extension in various situations. We try to find a final configuration that includes a data cache, network cache, data transfer mechanism, and other related components. As a new benchmark has been introduced, we need to establish the experimental methodology taking into consideration the available benchmarks. The methodology and results obtained are presented in the following sections.

6.1 Wall-clock Time and Simulated-cycle Time Correspondence

As mentioned, timing accuracy was an important issue. Here, we simulated the benchmarks completely with the messaging environment and varied the relationship between one simulation cycle and one unit of wall-clock time, effectively increasing or decreasing the average message inter-arrival time. Then a correspondence between wall-clock time and simulated-cycle time was selected so that the messaging environment delivers the messages (relative to the computation stage) at the same point under simulation and the original runs on the cluster.

To establish the message traffic intensity in benchmarks CG and PSTSWM the simulator parameters were selected as shown in Table 6.1.

Table 6.1: Simulator configuration

DL1 (Data Cache L1) Size	32KB (256:4, 32B blk.)
DL1 Access Latency	2 Cycles
L2 Cache Access Latency	15 Cycles
IL1 (Instruction Cache) Size	16KB (512:1, 32B blk.)
IL1 Access Latency	2 Cycles
Memory Access Latency	200 Cycles <first_chunk>
Memory Access Latency	4 Cycles <inter_chunk>
Memory Access Bus Width	16B
Network Section	4KB (64:4, 16B blk.)
Network Section Latency	2 Cycles
Message Section	4KB (64:16, 4B blk.)
Message Section Latency	2 Cycles

In the subsequent experiments the network cache was kept large enough (4 KB 64:4:16) so that no replacement occurred in this cache section because of one more benchmark (PSTSWM) being introduced at this point of the study.

Figures 6.1 through 6.4 depict the termination time of the simulation (expressed as the number of simulation cycles) versus the number of simulation cycles per wall-clock-unit time for CG and PSTSWM benchmarks. They also show the distance between the arrival of the last message and the termination time of the benchmark. The number of cycles per wall-clock-unit time is represented as r. As can be observed, the termination time increases gradually as the number of cycles per wall-clock-unit time increases, exhibiting a knee when the cycle per unit time is such that there is a match between the average computational speed of the simulated system and the PowerPC.

In subsequent simulations, this value of cycles per unit time is used to select the optimum data cache size and to establish the timing of the message sequence. The simulations are also run using slightly increased/decreased cycles per unit time value to explore messaging environments with different inter-arrival times. By doing so, we can examine whether the messages arrive earlier or later with regard to their respected receive requests, and explore the impact on the caching structure.

6.2 Message Size Distributions

These experiments involved the CG Class A and PSTSWM benchmarks, and ran 1M instructions first in fast-forward mode and then to the completion of each benchmark in performance simulation mode on sim-outorder. The reason for this was that the first message arrived after the fast-forward time and 100% of messages arrived before the simulation terminated. It should be noted that this work is concerned with short

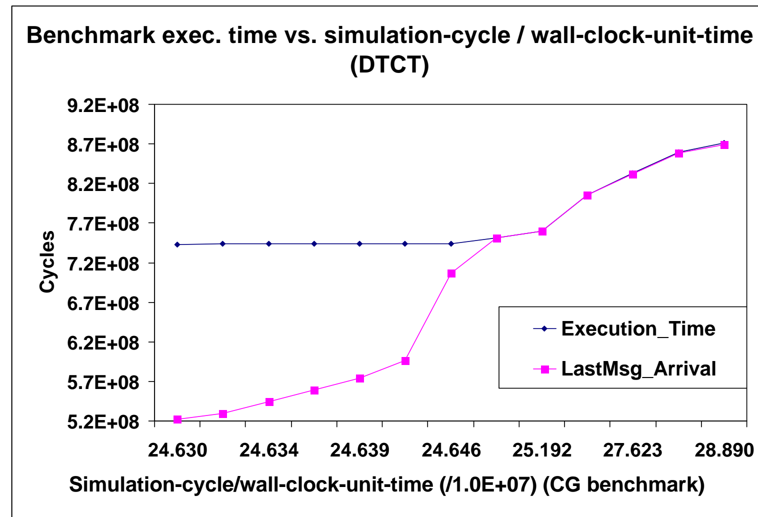

Figure 6.1: Execution time vs Simulation-cycle/Wall-clock-unit-time *(DTCT)* CG Benchmark (Class A)

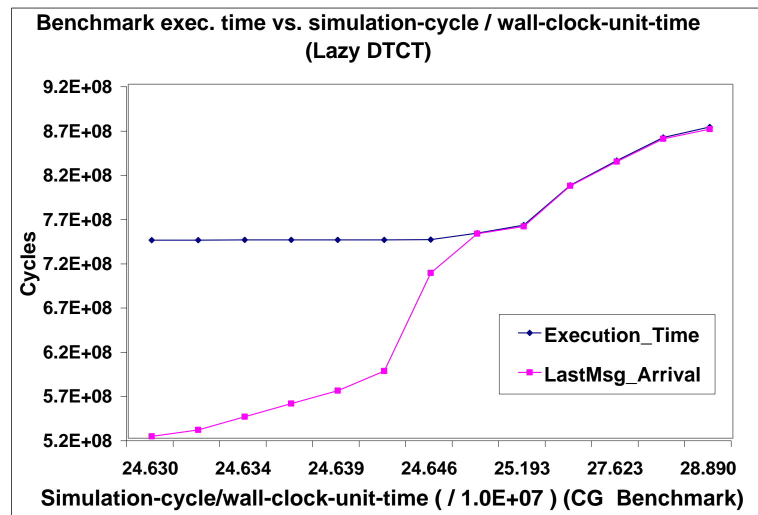

Figure 6.2: Execution time vs Simulation-cycle/Wall-clock-unit-time *(Lazy DTCT)* CG Benchmark (Class A)

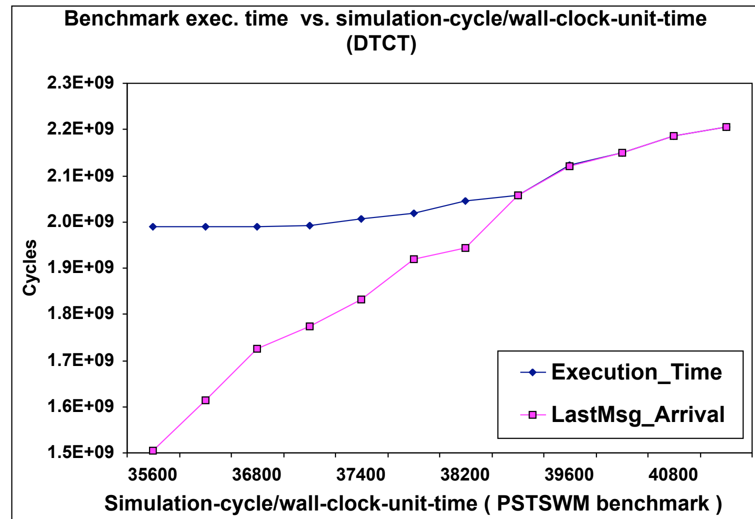

Figure 6.3: Execution time vs Simulation-cycle/Wall-clock-unit-time *(DTCT)* PSTSWM Benchmark

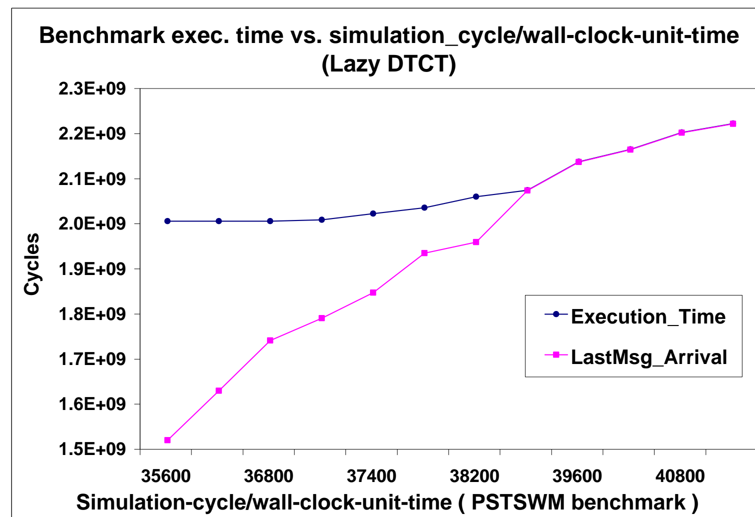

Figure 6.4: Execution time vs Simulation-cycle/Wall-clock-unit-time *(Lazy DTCT)* PSTSWM Benchmark

messages (that is, messages of length identical or less than that of a network cache line).

The total number of messages received by the master process for the CG, including short (less than or equal to 16B) and long messages (greater than 16B), for that period is 4208 for the 64-processor configuration. For the PSTSWM, the total number of messages received by the master process, including short and long messages, is 10463 for the 64-processor configuration. The distribution of messages in the benchmarks is shown in Table 6.2. This table shows the total number of messages received by all processes when we ran the benchmark on 64 processors.

Table 6.2: Message size Distribution (Byte) in CG and PSTSWM Benchmarks for 64 Processors

Benchmark	size$\leq$16B	16B$<$size$<$64KB	size$>$64KB
PSTSWM	370101	41293	0
CG Class W	113992	74504	0
CG Class A	113992	74504	0
CG Class B	119067	78062	0

In previous experiments the impact of the proposed extension on the processor's caching system for the CG benchmark was shown. The results confirmed that the extension has no adverse effect on the processor's caching system. Here, we investigate further the problems related to the caching systems, that is, different binding policies and the overhead incurred in using the network extension in both benchmarks. Additionally, we need to establish whether the proposed network extension speeds up delivery of the payload to the consuming process as compared to a processor that does not employ the extension. If the processor employing the network cache extension exhibits a faster access to the required data, it will result in a reduction in

the communication sections. The shorter communication section would need shorter computation sections to be hidden effectively thus allowing a finer granularity and an improved parallelism. This is where major gains in performance are achievable.

In the following sections we study the impact of the proposed network cache and determine the minimum size, organization and binding policy that produce enhanced performance. Section 6.3 investigates the optimum data cache configuration using the observed traffic intensity.

Section 6.4 examines the impact of the proposed binding methods $DTCT$ and *lazy* $DTCT$. The access time behavior for different methods is investigated in section 6.4.1.

6.3 Optimum Data Cache Configuration

Before evaluating the extension behavior and performance, it was decided to find the optimum data cache size for the above-mentioned inter-arrival time in order to minimize the data cache's effect on the proposed extension and to achieve a realistic behavior in the following experiments. Therefore, the solution for the optimum data cache size was used for the rest of the experiments. We ran the simulations assuming slightly increased/decreased cycles per unit-time-value to explore data cache behavior with different inter-arrival times. The results of these experiments are shown in Figures 6.5 to 6.8 for the $DTCT$ and *lazy* $DTCT$ respectively. It is clear that the data cache behaviors for both configurations are almost the same.

As shown in these figures, the optimum data cache size for this configuration can be selected as a 32KB cache (256 sets, 4-way, 32B cache lines). This cache size was used in the following experiments.

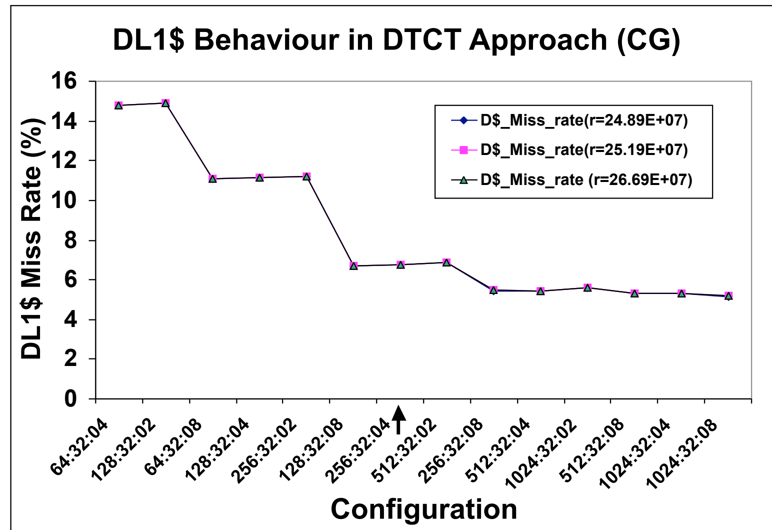

Figure 6.5: Data Cache Misses in DTCT Configuration in CG Benchmark (Class A)

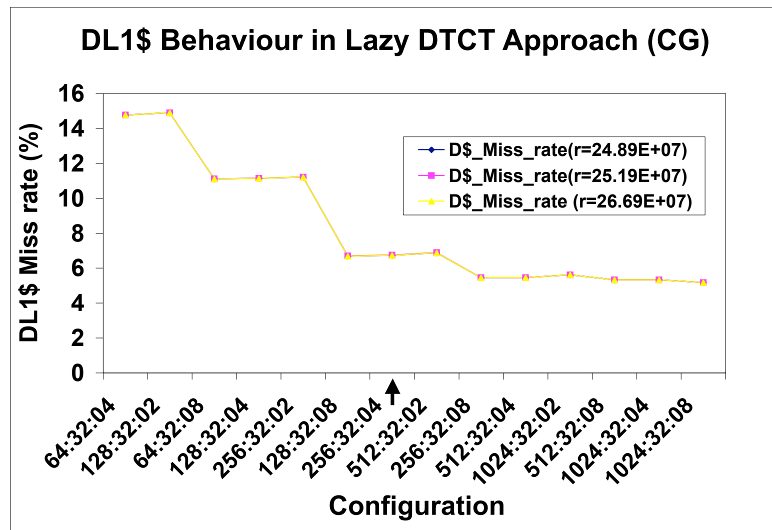

Figure 6.6: Data Cache Performance in Lazy DTCT Configuration in CG Benchmark (Class A)

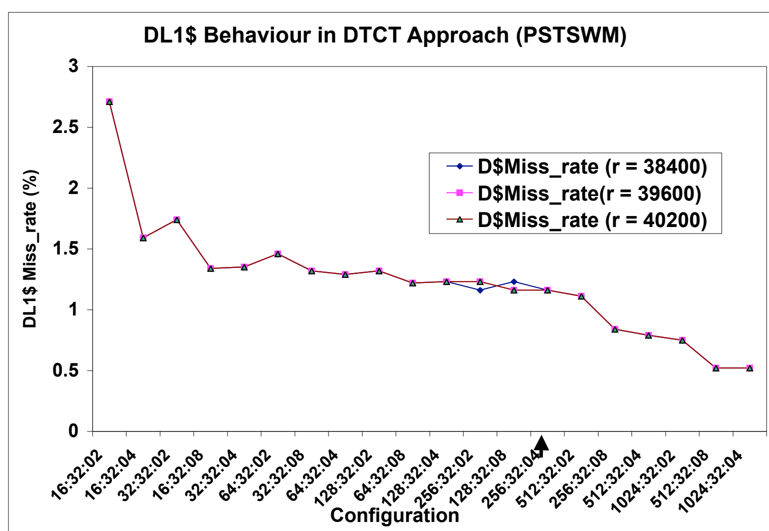

Figure 6.7: Data Cache Performance in DTCT Configuration in PSTSWM Benchmark

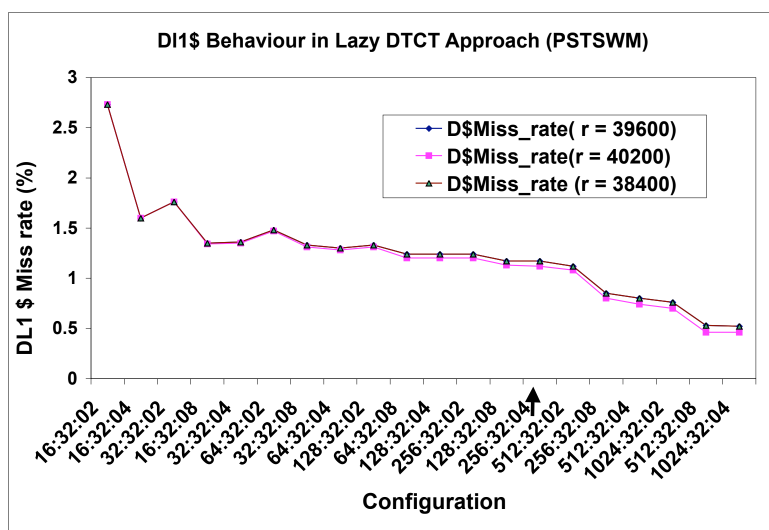

Figure 6.8: Data Cache Performance in Lazy DTCT Configuration in PSTSWM Benchmark

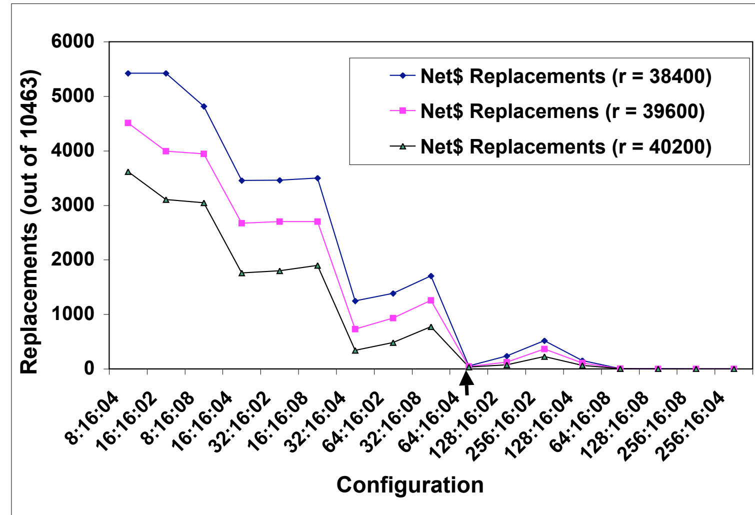

Figure 6.9: Network cache sensitivity to the size and associativity in PSTSWM Benchmark

6.4 Data Cache Behavior

Previously we studied the impact of varying the size and associativity of the network cache [71], which proved that a small network cache suffices to accommodate all the messages. Therefore, the size of the *network* cache was customized to 4KB (64:4 with 16B blocks) onward. The network cache behavior in the PSTSWM benchmark [75,76] was also investigated. Figure 6.9 confirms that a relatively small network cache is sufficient for the proposed extension.

One question we wanted to explore was whether the proposed binding methods $DTCT$ and $LDTCT$ would cause excessive replacements in the data cache. For this the performance metrics of the caching system were considered. These metrics include (i) the number of accesses, (ii) replacements, (iii) writebacks, (iv) invalidations, and (v) misses in the processor's data cache.

In order to compare the impact of the proposed methods on the data cache to

those of various hardware assists that may be employed to facilitate the transfer and binding of a message in a classical communications architecture, the following environments were simulated.

- The processor is responsible for transferring the data from the network interface memory to the process memory. This policy is referred as *Processver*.
- The processor uses DMA to get the data from the NI into a buffer and then transfers that data to its final destination in the process memory. This policy is referred to as *Dmaver1*.
- The processor uses DMA to transfer the data directly into the process memory. This policy is called *Dmaver2*.
- The processor uses the proposed extension to transfer data into the data cache (short messages) or the processor memory (long messages); this is the *DTCT* approach.
- The bound short message is left in the network cache; however, long messages are transferred into the process memory; this is the *LDTCT* approach.

The results of the simulations are presented in Figures 6.10 and 6.11. In this set of experiments, we assumed cycle/time ratios of 24.98E+07, 25.19E+07, and 26.69E+07 corresponding to the locations in the curve of Figure 6.2, depicting situations where both the simulated and the PowerPC processors are roughly equivalent for the CG benchmark. For the PSTSWM benchmark, we also considered equivalent situations for both environments (simulator and the PowerPC processor) which occur for r = 38400, r = 39600, and r = 40200.

The size of the network cache was chosen to be large enough so that the messages would remain there until they were bound and consumed; we therefore measured the

performance characteristics of the data cache (the number of accesses, misses and so on) and normalized the number of messages in order to compare across the different benchmarks.

Each bar represents the increments of the corresponding values with reference to those obtained by $LDTCT$ per message. For this the cache statistics that were collected during the complete execution of the program are used and divided by the number of messages in each benchmark for the various approaches. A positive number signifies that the corresponding quantity is larger than that obtained using $LDTCT$.

As can be seen from these results, the number of cache accesses and misses are smallest when using $LDTCT$. These results lead us to believe that it is possible to keep a message in the network cache during the receive operation so that it can be used by the consuming process as soon as it is needed. Keeping the message in the network cache during this operation does not interfere adversely with the caching environment for the remainder of the computation. In addition, most of the bound messages persist in the network cache long enough to be used by the consuming process.

Note that the $LDTCT$ technique has a better data cache performance than the $DTCT$ method; however, the scale of the figures does not allow us to see this behavior clearly.

6.4.1 First Access Time Behavior

This section compares the *first message access time* characteristics of the above-mentioned policies. This is defined as the time needed between the issue of a receive call and the first time the message is accessed by the computation. This includes the miss penalty and hit access times inevitable in accessing the data.

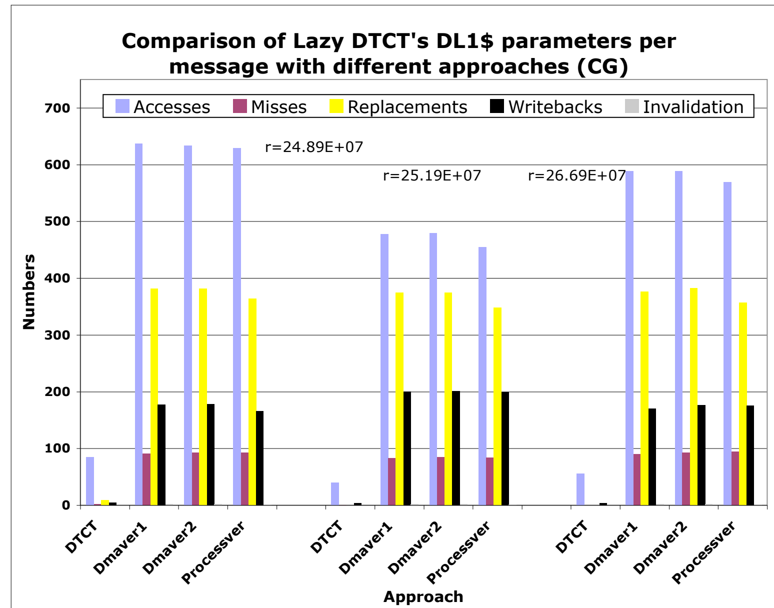

Figure 6.10: Comparison of Data Cache Performance with different approaches for CG Benchmark (Class A) for different inter-arrival rates (Each bar represents the increments of the corresponding values with reference to those obtained by $LDTCT$ per message)

Figures 6.12 and 6.13 show the results of these experiments. They confirm that the $LDTCT$ policy accesses the data the first time much faster than other policies and that the data can be consumed sooner by the processor. This ensures that $LDTCT$ in high-bandwidth SANs can provide data to a CPU in such a way that it can be accessed quickly, which in turn results in reducing the message delivery latency and consequently increases the potential parallelism in parallel applications. In addition, this method does not degrade the performance because it does not pollute the data cache, as discussed in section 6.4.1. This point is investigated further in the following section.

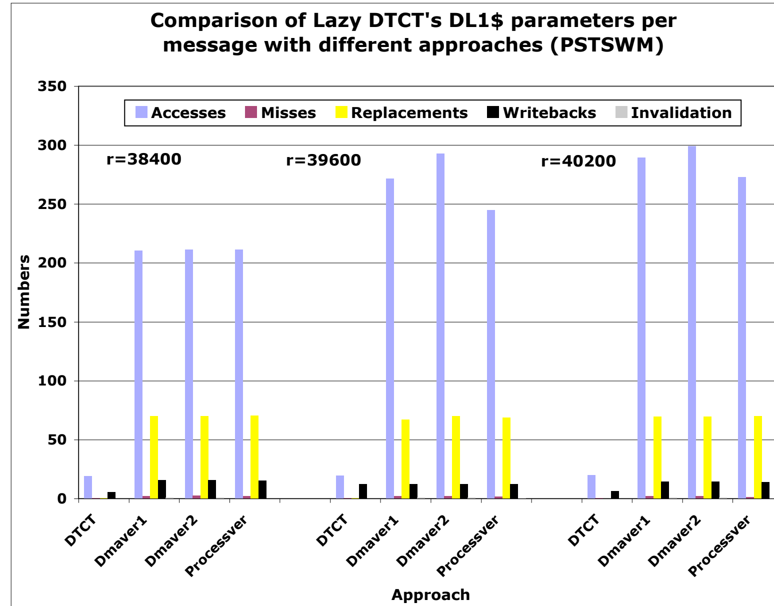

Figure 6.11: Comparison of Data Cache Performance with different approaches for PSTSWM Benchmark for different inter-arrival rates (Each bar represents the increments of the corresponding values with reference to those obtained by *LDTCT* per message)

6.4.2 Last Access Time Behavior

As mentioned above, a short message *first access time* implies that the arrived message can be accessed quickly. However, the proposed mechanisms might affect the processor's caching system, which would result in poor application performance due to more delays in accessing the subsequent messages. To investigate this problem we measured the *message last access time* characteristics of the above-mentioned policies, defining this as the time needed between the issue of a receive call and when the message is actually being accessed by the computation for the last time, representing how much faster the application can access the arrived messages for the

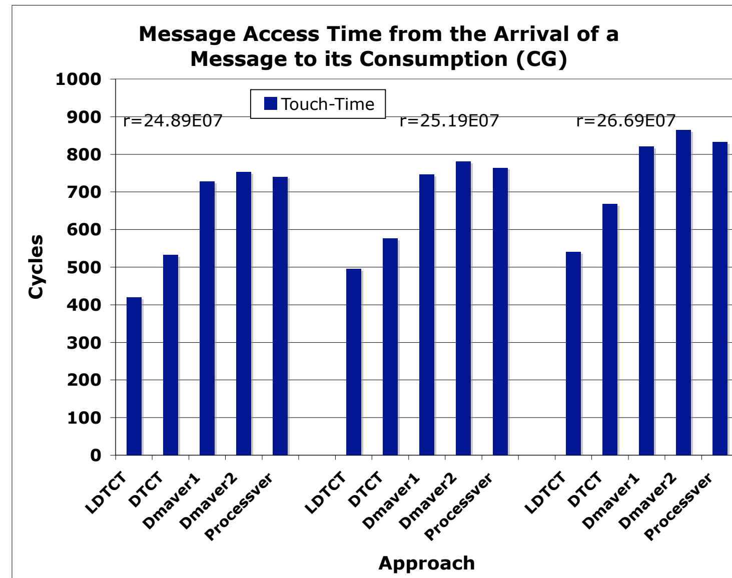

Figure 6.12: First access time behavior for different configurations in CG Benchmark (Class A) for different inter-arrival rates

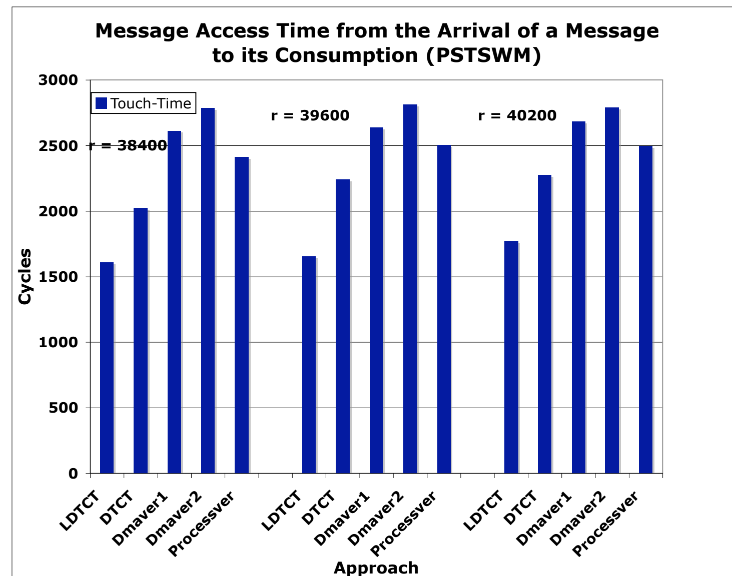

Figure 6.13: First access time behavior for different configurations in PSTSWM Benchmark for different inter-arrival rates

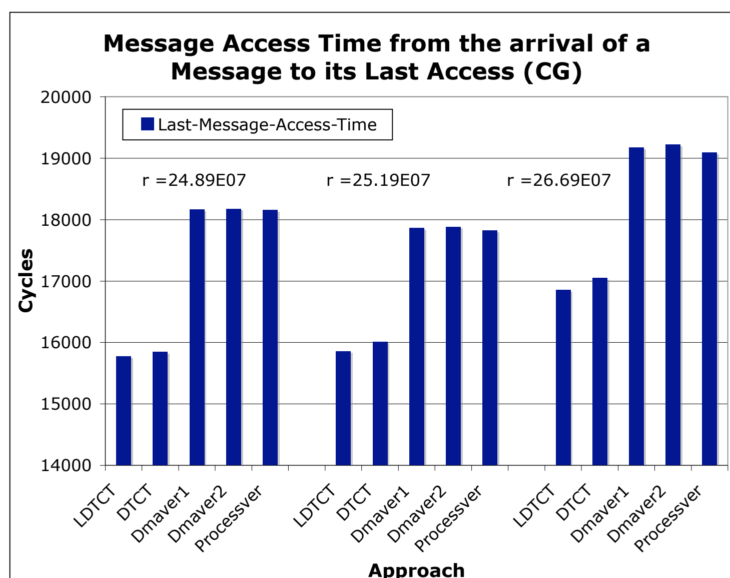

Figure 6.14: Last access time behavior for different configurations in CG Benchmark for different inter-arrival rates

last time. The faster the arrived message is accessed, the better the performance.

Figure 6.14 shows the result of the conducted experiments for the CG benchmark. Since the PSTSWM benchmark accesses short messages only once, the *last message access time* values are the same as those shown in Figure 6.13.

These results confirm that the *lazy DTCT* policy accesses the data for the last time much faster than other policies and thus the data can be consumed sooner by the processor. This indicates that the proposed extension improves the communication by enabling the delivery of the message payload to the consuming process earlier than is otherwise possible.

Note that these results were achieved in a simulation environment (SimpleScalar) that was tailored in order to run message passing applications. Specifically, only the master process was simulated and the MPI traffic was delivered at set points in time

(corresponding to the points in time observed during the runs of the same application on the IBM RS 6000/SP system).

Thus, although our methods improved the times at which the delivered data became available, these improvements were not accumulative in the sense that they did not affect the times at which the MPI traffic was delivered. We expect therefore that the last access time to be further improved in a MPI environment that incorporates our extensions. Taking these into consideration, message passing applications in a real environment can benefit from the *DTCT* techniques severalfold more than the results shown in this section.

6.5 Summary

The simulations show that by using the proposed network extension along with the *lazy DTCT*, less data cache misses were encountered as compared to when *lazy DTCT* was not used. It was also demonstrated that *lazy DTCT* provides a significant reduction in access latency for the arrived messages in I/O intensive environments such as message passing configurations and SMPs without polluting the data cache.

The question of performance has been answered, albeit indirectly. One can reasonably expect that an enhancement would ensure that the termination time of a given code is advanced, thus ensuring a faster execution. This is not easily demonstrated in a parallel environment. The issue here is that the benchmark codes have been finely tuned so that the computation sections hide the communication thus ensuring optimum execution times for a given architecture. Therefore, an improved communication environment would not impact the execution time of the application. Simply put, the computation sections will remain identical and dominate the run time, while the (shorter) communication sections will still be effectively hidden by the computation sections.

Yet, the improvement lies in the fact that shorter communication sections need shorter computation sections in order to be hidden effectively and allowing a finer granularity and an improved parallelism. This is where major gains in performance are achievable. This work has established that the proposed extensions improve the communication by enabling the delivery of the message payload to the consuming process earlier than is otherwise possible. Thus, it is a thesis that the proposed extensions result in clusters with efficient receive-side communications.

It should be pointed out that these results ensure the benefit of this policy per message base conservatively, in the sense that a real message passing environment can benefit from the proposed techniques more than the results shown in this chapter. In fact, the improvement of the access latency affects the rest of computations as well as communication in a real parallel environment. In other words, the improvement of the access time of each message accumulatively improves the performance of running applications much more than that we observed. Taking this into consideration, message passing applications in a real environment can benefit from the *DTCT* techniques severalfold more than the results shown in this section.

This chapter has presented the impact of the *DTCT* and *lazy DTCT* approaches on the message access time, and the investigation of the impact of the messaging on the data cache, as well as the required size and organization of the data and network caches. The following chapter illustrates the effectiveness of the proposed Direct-to-Cache transfer policies compared to well-established communication protocols such as TCP/IP and VIA.

Chapter 7

Comparing Direct-to-Cache Transfer Policies to TCP/IP and M-VIA

This chapter presents a further evaluation of the proposed data transfer techniques with standard communication protocols in cluster environments. The benefits of the network cache environment are compared with two common communication protocols, TCP/IP and VIA [48]. Additional metrics that better characterize the behavior of an application are also presented, as well as the messaging environment under these protocols.

For this to be accomplished there was a need to study the possible overhead and cache pollution introduced through the operating system and the communications stack, as exemplified by Linux, TCP/IP and M-VIA. This overhead was introduced into the simulation environment to observe its effects on the proposed extensions. It was also possible to compare the performance achieved by an application running on a system incorporating the proposed extensions with the performance of the same application running on a standard system.

The Virtutec Simics environment [54], a full system simulator including processor cores, peripheral devices, memories, and network connections was employed. We

evaluated the receive operations overhead in TCP/IP and VIA implementation of MPICH [6] and explored the cache behavior during those operations. MVICH [7] is used as the VIA implementation of MPICH. The system simulation environment described above has been used to obtain cache and network traffic parameters in a complete system environment. Subsequently, the achieved data have been utilized within the SimpleScalar simulation environment of the extensions to obtain a realistic evaluation of their behavior in a system environment.

In the following sections, the TCP/IP and VIA protocols are described briefly. Then, the simulation environments, the assumption, and the achieved results are presented.

7.1 TCP/IP and VIA Overview

7.1.1 TCP/IP Protocol

The TCP/IP protocol uses *sockets* as the user interface to send and receive data to and from the NIC. To send a message, the sender copies the data into a transmit buffer and pushes a descriptor into the corresponding send queue of the socket. The sender then transmits a request to the NIC, which transfers the data already copied into the socket's buffers into the NIC's transmit FIFO queue, and finally into the network, as shown in Figure 7.1.

At the receiver, the arrived data is transferred into the NIC's receive FIFO queue. It then transfers that data into the socket buffer in the kernel space using DMA. After the arrival of the data packet, the NIC notifies the processor of the arrival of new data. The operating system copies the newly arrived data into the user buffer.

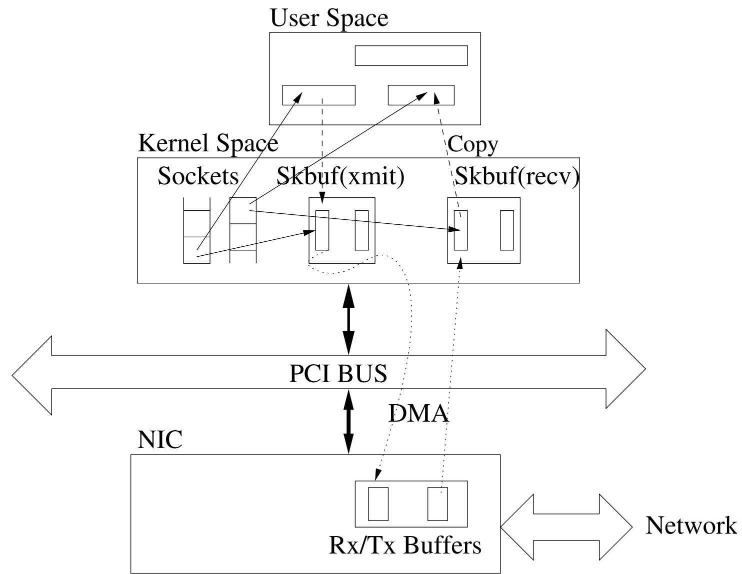

Figure 7.1: TCP/IP Communication Protocol

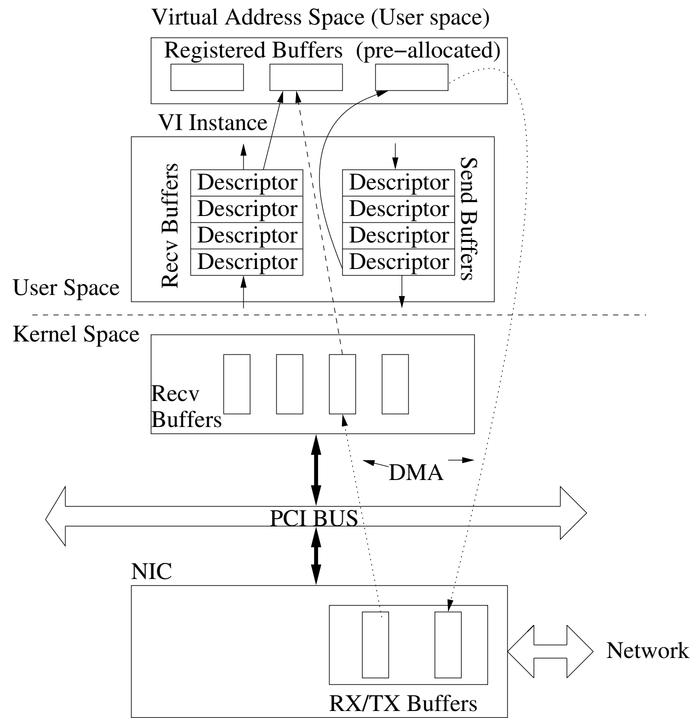

Figure 7.2: VIA Communication Protocol

7.1.2 VIA Protocol

M-VIA (ver1.2) [4] was used, which is a modular implementation of the Virtual Interface Architecture (VIA) Specification. Each VI instance, which is a communication endpoint, includes send and receive queues as shown in Figure 7.2. VIA provides two types of data transfer: (i) a traditional send/receive messaging method and (ii) Remote Direct Memory Access (RDMA). This work is concerned with the message passing environment of VIA, that is, the traditional send/receive messaging method. With the send/receive method the receiver has to post the corresponding receive descriptor before the arrival of a message. In the case of receiving a packet before posting the corresponding receive descriptor, while the receiver discards the arrived message, the discarded packet has to be resent by the sender.

When sending a message, the sender application creates a descriptor, including the virtual address and the length of the send buffer, in the registered memory region [4]. The descriptor is posted and the NIC is informed through the doorbell mechanism, which is used to notify the VI NIC that work has been placed in a work queue. The NIC obtains the address and length of the user buffer from the descriptor and transfers the data from the user buffer to the NIC via DMA. Thereafter, the NIC injects the data into the network.

At the receiver side, the application creates a receiver descriptor in the registered memory, including the virtual address of the receive buffer. The descriptor is posted and the NIC is informed through the doorbell mechanism. Having received the information from the network, the NIC stores the incoming message in staging buffers and the message header is checked to find the VI ID at the receiving side. The NIC then transfers the message from the staging memory to the user space using DMA.

As described, to transfer the received data to its final destination, legacy protocols such as TCP/IP have a layered and deep-stack architecture that introduces extra

overhead. However, user-level communication protocols such as VIA have access to the user's application memory and employ smaller stack architecture, which results in less overhead. It should be noted that VIA still transfers the received data into the main memory; therefore, the processor experiences a data cache miss, at least in its first access to the received data. In contrast, the proposed extensions transfer the received data into a network cache or data cache that can be accessed quickly.

7.2 Experimental Methodology

As explained earlier, the aim of this part of the study is to evaluate the effectiveness of the proposed Direct-to-Cache transfer policies compared to well-established communication protocols such as TCP/IP and VIA. In order to establish a sound comparison and to consider the overhead that an operating system inevitably introduces, we proceeded in two phases. The first involved the study of the communication and memory requirements of MPI applications over established protocols TCP/IP and VIA. The second used the parameters established in phase one to study the effect of the proposed extensions.

It was necessary to use this phased approach because it was not possible to establish a complete environment (simulation or otherwise) that incorporates the proposed extensions, networking stack, operating system, and application environment.

7.2.1 Overhead Measurement during MPI_Receive Operation

To evaluate the overhead incurred by TCP/IP and VIA protocols and to explore the data cache behavior during receive operations, Virtutec Simics was used, which simulates network nodes and all their software, from network device drivers to operating systems, as well as communication stacks and application programs.

The enterprise-multi configuration of the predefined and preconfigured platforms were used. This setting is based on the enterprise system configuration, which simulates x86-p4 processors running RedHat Linux 7.3 with kernel version 2.4.18-3[1].

In the enterprise-multi configuration, two x86-p4 based PCs are connected through an Ethernet link, as shown in Figure 7.3. The network interface model has been changed to DEC21140A, a Fast Ethernet NIC because of its ability to transfer data to and from the host memory via DMA; furthermore, its driver exists in the communication protocol's implementation of M-VIA. A 32KB ICache and a 512KB L2 unified cache were also added to each processor model. The size of L1 DCache was varied for the simulation experimentations.

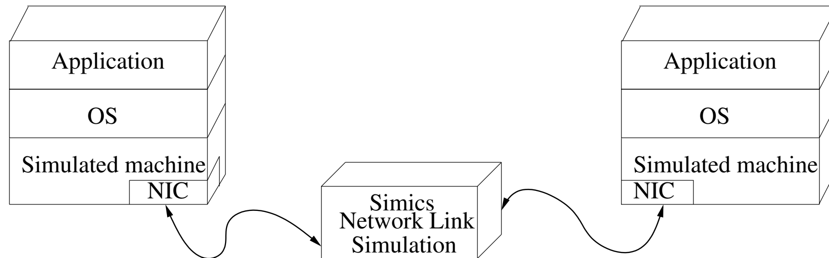

Figure 7.3: A multi-computer configuration in Simics

To investigate the overhead incurred by the TCP/IP protocol, we instrumented the Linux kernel version 2.4.18-3 to collect the necessary information without interfering with the normal execution of the simulated processors. For this, the *magic* instruction facility provided in Simics was used. This instruction was added at

[1]Kernel version 2.4.18-3 is not the latest version of Linux distributions. Newer kernel implementations such as 2.6 have better stack management in the TCP/IP protocol. However, the last distribution of M-VIA supports kernel version 2.4.18-3; therefore, it was necessary to use it as a matter of support by the available VIA implementation. The same statement holds for DEC21140A, which is a 100Mbps in comparison to available Gbps NICs.

appropriate locations in the TCP/IP and M-VIA protocols inside the kernel as well as the M-VIA implementation.

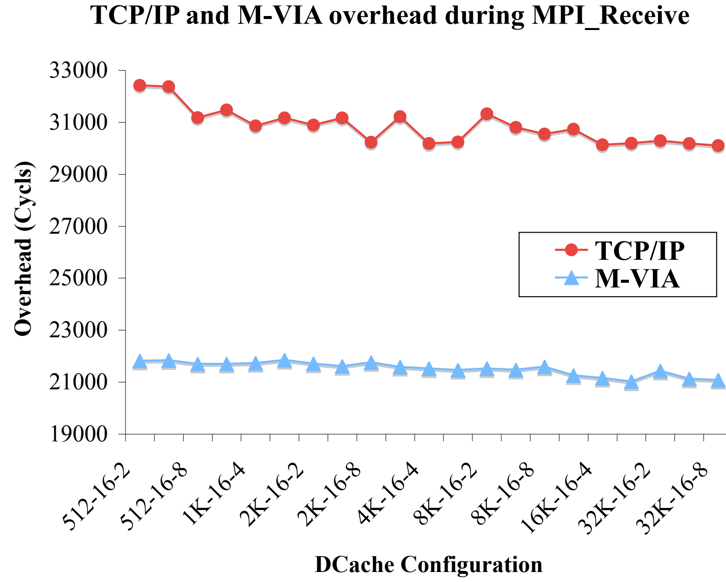

Figure 7.4: TCP/IP and M-VIA overhead during MPI_Receive

Having prepared the environment for evaluating the receive function's overhead, we ran several ping-pong micro-benchmarks and used the K-best measure algorithm [42] to find the overhead incurred in each protocol. Figure 7.4 shows the overhead of the MPI_Receive function in a message passing environment in the TCP/IP and M-VIA implementations. Because of the concern regarding short messages, this statistic represents each protocol's overhead and consists of context switching, memory copying operations, and communication processing in order to receive a one byte message in different cache configurations[2].

[2]In Figures 7.4 to 7.8, the notation s-l-i is used to denote the organization of the cache. s represents the total number of sets, l represents the length of the cache line in bytes and i represents the interleaving factor. Thus 32-16-4 means a cache of 32 sets each having lines of 16B for a total size of 32 * 16 * 4 = 2KB.

As shown in Figure 7.4, during receive operations the overhead in TCP/IP is almost 1.5 times more than in VIA, which is consistent with the fact that the TCP/IP protocol has a deeper protocol stack. For the remainder, the performance of the extensions is compared to the more efficient VIA protocol.

7.2.2 Cache Behavior during and after MPI_Receive

The cache state is an important factor in analyzing the protocols under discussion. Since the communication protocols execute on the same processor during receive operations, the caching system is affected by activities such as context switching and the execution of communication protocols themselves.

Moreover, a micro-architectural simulation environment (SimpleScalar) has to take into account all the above execution details in order to evaluate the proposed architectures accurately. This interference can affect not only the execution of communication protocols but also the behavior of applications after returning from the MPI_Receive function. For example, if the communication protocol pollutes the data cache due to its large working set and memory footprint during the receive function, the useful and valid data, which are needed after returning from that function, could be evicted from the data cache.

We ran the CG benchmark from the NAS parallel benchmark suite and the PSTSWM benchmark on the Simics simulator and explored their cache behavior. The miss rates of the data cache were measured during and after the receive operation by executing the above-mentioned MPI applications in different cache configurations. These miss rates are used in the simulation environment that implements the proposed extension (SimpleScalar).

As stated, the communication protocol for this part of the study was M-VIA. Figures 7.5 to 7.8 illustrate the data cache miss rates versus different cache

configurations in CG and PSTSWM benchmarks for different payload sizes (from less than or equal to 16B to 64KB). As depicted, the data cache miss rate during the receive function in CG is less than that in PSTSWM. This phenomenon is explained by considering the working set of each benchmark. Because the CG benchmark's working set is smaller in comparison to that of the PSTSWM, the data cache miss rates in CG are considerably less than those in PSTSWM. Also, the cache miss rates show similar behavior (diminishing at about 32KB). This behavior is observed for both benchmarks during and after a MPI_Receive. This cache configuration (32KB - 256 sets, 8-way, 16B cache lines) is used in the subsequent experiments.

7.3 Experimental Results

After preparing the environment and importing the information collected from the Virtutec Simics into the SimpleScalar simulator, the proposed *Network Processor* extension was implemented and the effectiveness of this extension was tested in handling short messages. At this stage, it was important to explore the complete impact imposed by the OS and networking stacks.

As explained, the miss rates of the data cache were measured during and after the receive operation by executing the MPI applications in different cache configurations. These miss rates were used in the simulation environment in order to randomly invalidate cache lines to match the observed behavior under M-VIA. The results obtained are the average values of executing the benchmarks twenty times. In addition, the kernel overhead that M-VIA has incorporated was taken into consideration. The simulator parameters are shown in Table 7.1.

Here, the focus was on the effectiveness of the extension compared to VIA specifically in order to establish how the proposed network extension speeds up the delivery of the payload to the consuming process as compared to a processor that

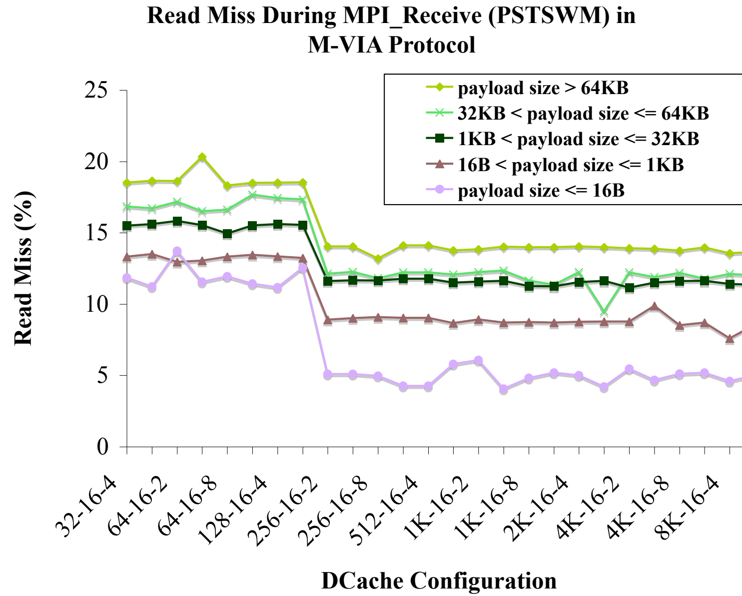

Figure 7.5: Analysis of Read Miss Rate during MPI_Receive in PSTSWM benchmark for different payload sizes

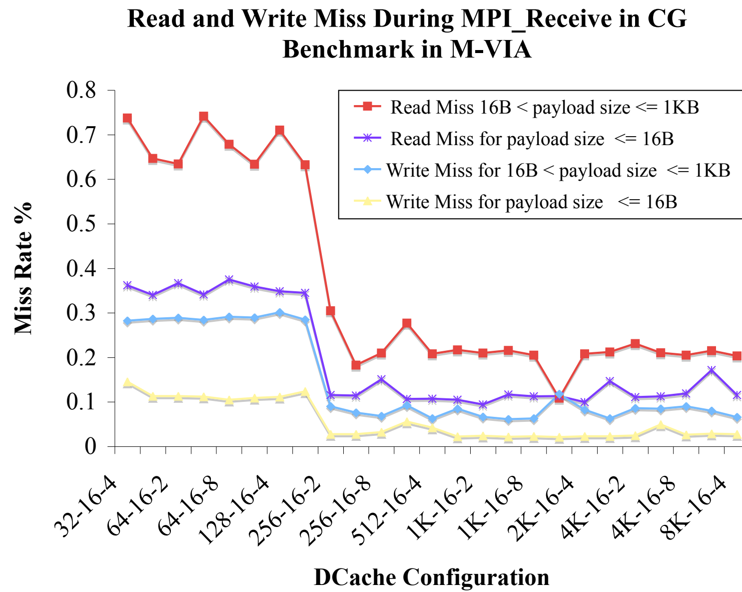

Figure 7.6: Analysis of Miss Rate (R/W) during MPI_Receive in CG benchmark (Class A) for different payload sizes

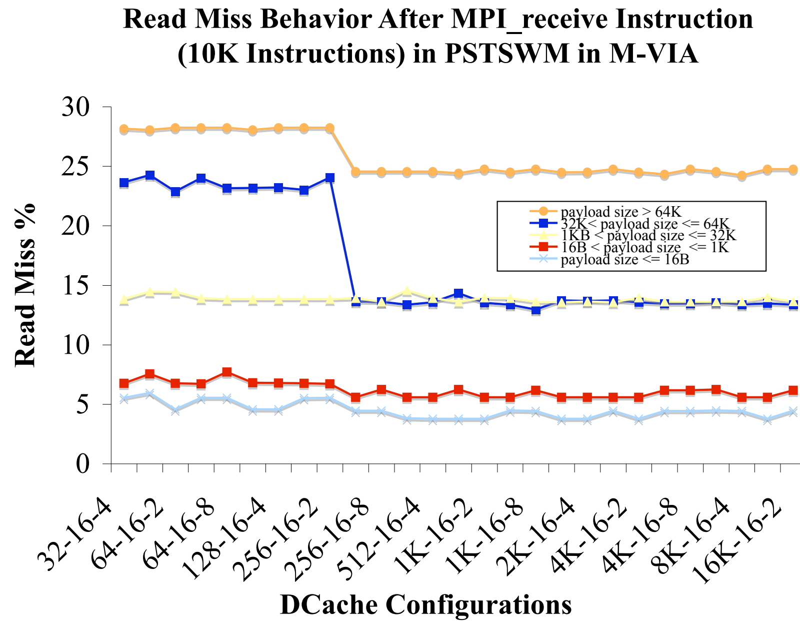

Figure 7.7: Analysis of Read Miss after MPI_Receive in PSTSWM benchmark for different payload sizes

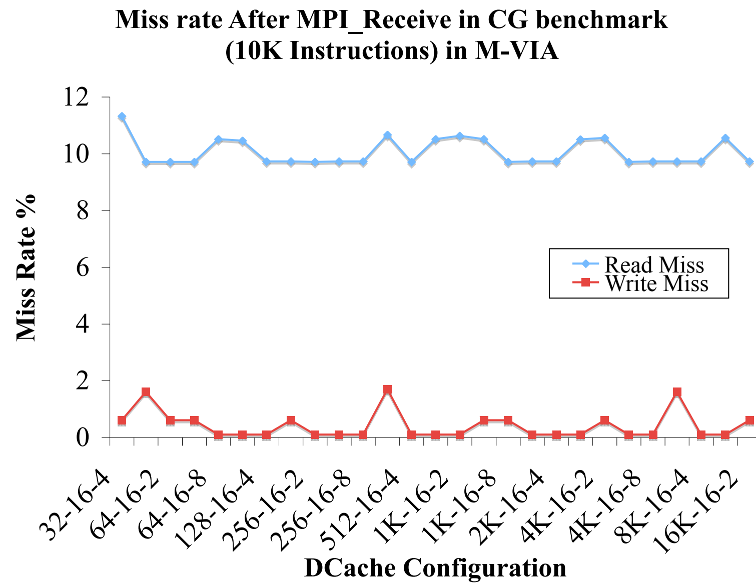

Figure 7.8: Analysis of R/W Miss after MPI_Receive in CG benchmark (Class A) for different payload sizes

Table 7.1: Simulator configuration

Parameter	Value
DL1 (Data Cache L1) Size	32KB (256:8, 16B blk.)
DL1 Access Latency	2 Cycles
L2 Cache Access Latency	15 Cycles
IL1 (Instruction Cache) Size	16KB (512:1, 32B blk.)
IL1 Access Latency	2 Cycles
Memory Access Latency	200 Cycles <first_chunk>
Memory Access Latency	4 Cycles <inter_chunk>
Memory Access Bus Width	16B
Network Section	2KB (64:2, 16 B blk.)
Network Section Latency	2 Cycles
Message Section	256B (32:8, 1B blk.)
Message Section Latency	2 Cycles
Process Section	256B (64:4, 1B blk.)
Process Section Latency	2 Cycles

does not employ the extension and uses standard VIA protocol. In order to compare the impact of our methods on the access time, and eventually on the execution time of MPI applications, the following environments were simulated.

- The NIC uses DMA to transfer the data into the receive buffers in the kernel space. Then the processor transfers the message to its final destination using the copy operation. This mechanism is the same as the one used in the VIA protocol. This policy is called *VIA*.

- The processor uses the proposed extension to transfer data into the data cache (short messages) or the processor memory (long messages), which is the *DTCT* policy.

- The bound message (short message) is left in the network cache or is transferred into the process memory (long messages), which is the *LDTCT* policy.

In subsequent sections, the impact of the proposed network cache is studied and its speed up in comparison to M-VIA and TCP/IP protocols is determined.

7.3.1 Message Access-Time Behavior

In this section, the message access time characteristics of the above-mentioned policies are compared. *First-access-time* is defined as the time needed between the issue of a receive call and the first access of the message by the computation. This includes both the miss penalty and the hit access times that are necessary to access the data. *Message last-access-time* defines the time needed between the issue of a receive call and the last access of the message by the computation before the arrival of the next message. *Message last-access-time* includes both the miss penalty and the hit access times that are inevitable in accessing the data.

Figures 7.9 and 7.10 show the results of the experiments. These confirm that the *DTCT* policies access the data for the first and last times faster than M-VIA, and the data can be consumed sooner by the consuming processor. This ensures that *DTCT* policies in high-bandwidth SANs can provide data to a CPU in such a way that it can be consumed quickly, which in turn reduces the message-delivery latency, and consequently increases the potential parallelism in parallel applications.

It is also clear that the *LDTCT* slightly outperforms *DTCT* in terms of accessing and consuming the data. The reason for this behavior is the latency of sending data from the network cache to the data cache in the DTCT approach. Moreover, it is possible that in the *DTCT* approach the received data evicted from the data cache means longer access time in subsequent accesses. It should be noted that in the *LDTCT* approach data is accessed by an indirection through the process cache; however, the number of accesses to the arrived messages in MPI benchmarks usually is so few that it will not result in many penalties for this indirection.

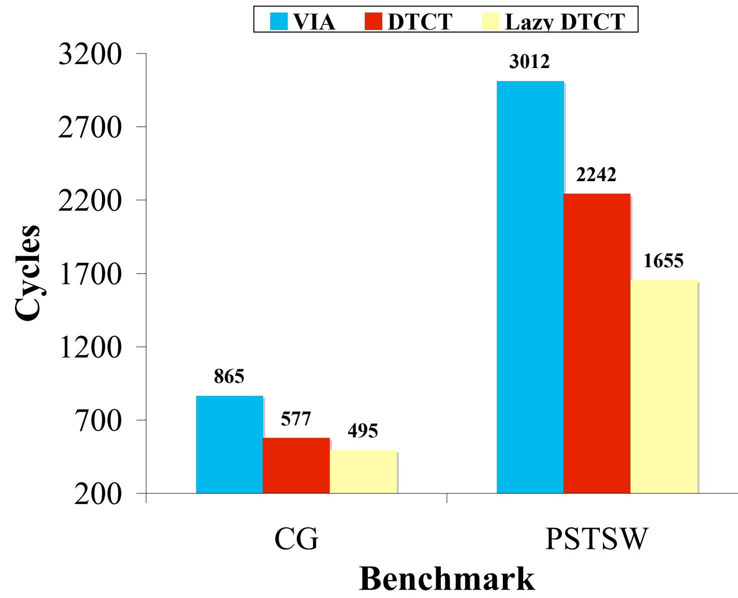

Figure 7.9: Message First-Access-Time behavior of different policies

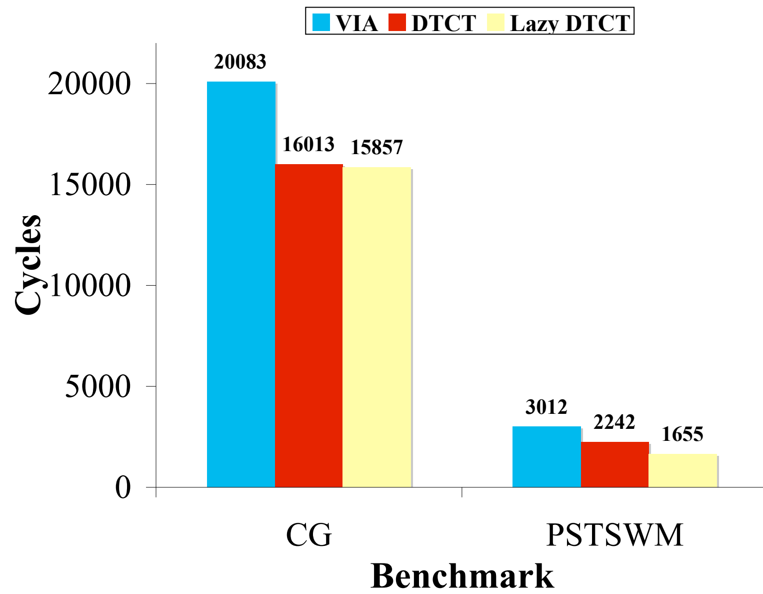

Figure 7.10: Message Last-Access-Time behavior of different policies

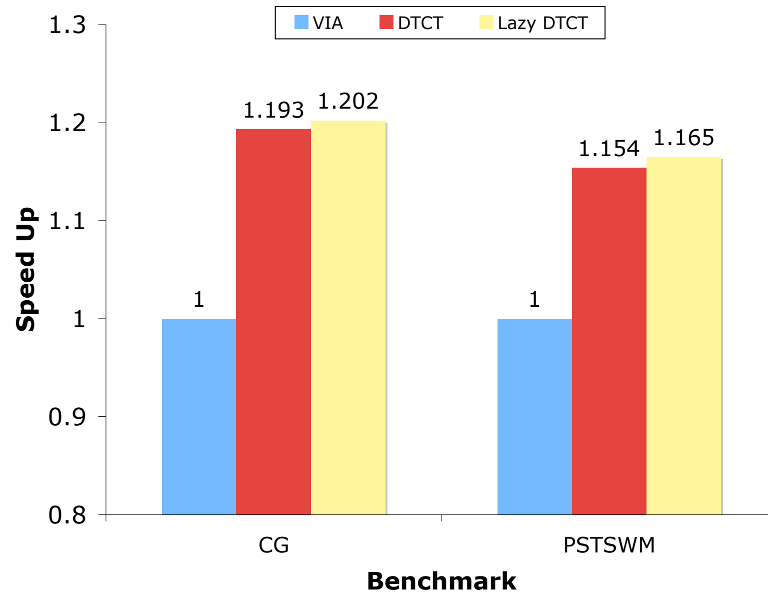

Figure 7.11: DTCT Policies speed up in Comparison to VIA

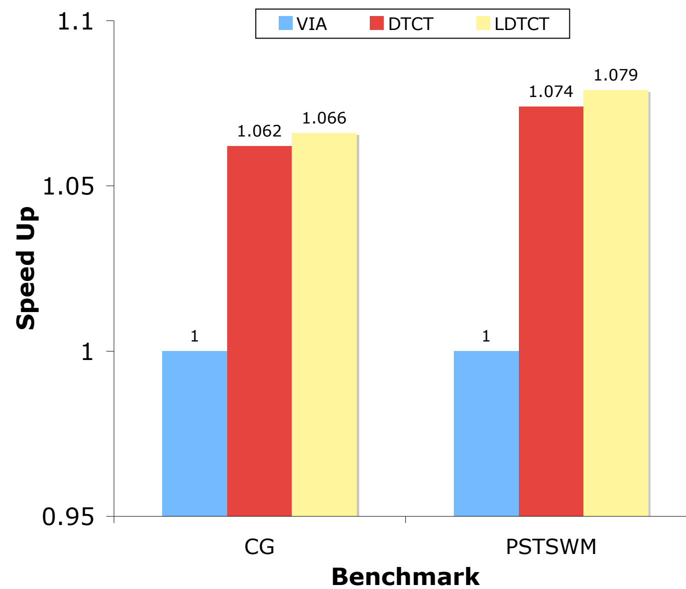

Figure 7.12: The effect of Data cache size increase (that is, 64KB) in VIA with respect to DTCT policies

7.3.2 Speed Up

In this section the execution time and speed up of *DTCT* policies were compared to M-VIA. *Speed up* defines the ratio of the execution time of the M-VIA implementation to those using *DTCT* policies.

Figure 7.11 shows the result of the experiment: *DTCT* and *lazy DTCT (LDTCT)* both outperform M-VIA by 15 to 20%. Therefore, the proposed extensions would result in clusters with efficient receive-side communications, thus allowing finer granularity and improved parallelism.

It was investigated as to whether the achieved speed up was the result of having two cache spaces (data and network caches) for *DTCT* approaches or not. For this, we used a data cache twice as large as the baseline cache (32KB) for the VIA approach in order to show the effect of having more cache spaces in comparison to *DTCT* and *LDTCT*, which have more cache spaces as the result of adding a network cache. As shown in Figure 7.12, both the *lazy DTCT* and *DTCT* still outperform VIA, although VIA has more data cache in comparison to the *DTCT* and *lazy DTCT* approaches.

We also investigated the instruction cache behavior and the possibility of achieving better performance by having more instruction cache space. For this we measured the hit ratio of the instruction cache for the VIA approach. The result revealed that the hit ratio for the existing instruction cache is equal to 99.2%. Therefore, the performance improvement cannot be significant as the misses in the instruction cache are mostly due to the cold miss for the first access to each instruction, which can not be avoided. In addition, parallel applications involve data parallel schemes that have regular instruction patterns because the main structures in these applications are for loop structures. In fact, the existing instruction cache enables the accommodation of loop structures in the above-mentioned benchmarks for the VIA implementation; therefore, increasing the instruction cache does not improve

the performance of the applications.

7.4 Summary

This chapter has presented the results of the evaluation of a network processor extension specifically designed to decrease the message reception latency in a MPI environment. The goal has been to compare the DTCT approaches to TCP/IP and VIA protocols.

The simulations have shown that by using the proposed network extension together with DTCT approaches, the execution of MPI applications is improved by 15 to 20%.

The observed *speed up* is attributed to improved communication efficiency afforded by the proposed techniques. It is well known [43] that the CG (class A) and PSTSWM benchmarks show poor scalability beyond 16 processors signifying that, on average, the communication portion becomes larger than that of the computation. At these scales the computation portion is not long enough to completely hide the communication and thus inefficiencies and poor scalability result. Since the proposed extensions produce more efficient communications, this environment can hide the communications portion more efficiently and thus yield better *speed up* at finer granularities.

It was also demonstrated that $DTCT$ methods provide significant reductions in the access latency for the arrived messages in I/O intensive environments (such as message passing configurations and SMPs) without polluting the data cache because the arrived data is kept inside a separate cache.

It should be noted that assumptions have been made in regard to the overhead needed for implementing the $DTCT$ techniques. As stated in Chapter 3, the proposed processor extension is comprised of a few special network-specific instructions that

help managing the operations of the cache and implementing the late binding. These could be added as new instructions to the ISA of the processor. Therefore, an efficient implementation of these network-specific instructions can reduce the involvement of the communications protocol to move the data between the processor's data cache and the NIC.

The next chapter studies the data transfer mechanisms on a multi-core processor, the Cell Broadband Engine (Cell BE). Different data transfer mechanisms and their various components are investigated to gain more insight into data transfer in the Cell BE. Using these measurements, preliminary techniques are proposed to facilitate the receiving and binding of the arrived data in the Cell processor environment. A cluster system is envisioned comprised of several cell processors, each supporting several computation threads.

Chapter 8

Hiding Data Delivery Latency on the Cell BE Processors

As explained in the previous chapters, the main contributors to message-delivery latency in message passing environments are the copying operations needed to transfer and bind a received message to the consuming process or thread. A significant portion of the software communication overhead is attributed to message copying. Therefore, if the necessary data could be placed into the fastest level of the memory hierarchy closest to the processor where the consuming process or thread resides and before it is needed, latency will be minimized.

Recently, a set of factors that include poor performance/power efficiency and limited design scalability in monolithic designs has moved high-performance processor architectures toward designs that feature multiple processing cores on a single chip (CMP).

The advent of the Cell Processor [70] with its multitude of synergistic processing elements (SPEs) and their associated fast memories provides an opportunity for an advanced computational environment for demanding MPI applications. The challenge is to ensure data availability to the computations running on the SPEs. This data

may exist locally, on another SPE, on the PPE, or globally on another Cell processor.

As discussed in Section 3, Afsahi et al. [17] have developed methods of predicting the consumption of received data in a parallel environment. We have developed methods of placing the data into the memory hierarchy of the processor that eventually consumes it [76, 75, 72].

The present chapter focuses on studying the data transfer mechanisms between the processing elements of the Cell BE (the PPE and the SPEs) and identifying their communication capabilities in terms of latency and throughput for a variety of communication patterns. The ultimate goal is to use this information together with Afsahi's prediction techniques to implement an efficient MPI environment.

8.1 Motivation

The Cell processor has shown potential for high-performance computing that uses MPI as the de-facto programming model. Therefore, it is vital to implement MPI efficiently on the Cell BE processor in order to leverage its tremendous computational power. For this purpose, there have been feasibility studies of MPI implementations on the Cell BE [83, 84]. In these studies, a minimal set of a synchronous mode MPI on the Cell BE has been implemented, and the results show the potential of the Cell BE to run MPI applications efficiently.

Also, a programming model, MPI microtask, has been proposed [98] by IBM. In this model, which is based on standard MPI, programmers do not need to manage the SPEs' local stores as long as they divide their application into partitions (microtasks) that fit into the local stores. As well as the studies on MPI implementations, there have been several compiler proposals that aim at automatically generating parallel codes for the Cell BE processor [51, 35, 50].

Another study [81] has characterized the DMA transfers in the Cell BE processor when the DMA operations are issued on the SPEs. In our study, we have repeated the DMA characterization as discussed in [81], but have also investigated different components of DMA operations and additional modes of data transfers, including PPE initiated, mailbox communication, and memory management overheads.

We quantified the latency of the DMA technique and measured its different components to gain more insight into data transfer in the Cell BE [77, 78]. Using these measurements, preliminary techniques were proposed to facilitate the receiving and binding of the arrived data in the Cell processor environment.

The portion of the results focusing on SPE-issued DMA operations agree with the ones presented in [81]. However, the results on the remaining modes of communication and the memory management behavior are, to the best of our knowledge, new.

8.2 The Cell Processor Architecture Overview

The Cell BE was designed as a joint venture by Sony, Toshiba, and IBM to be the processor for Sony's Playstation gaming system [70]. It consists of a high performance PowerPC core called PPE that controls eight simple in-order SIMD cores called synergistic processing elements (SPEs). An overview of the Cell processor is shown in Figure 8.1.

The PPE, the eight SPEs, the memory controller, and I/O controller are connected through four data rings known as EIB. Each unit can transfer data via EIB at a rate of eight bytes/cycle. In the Cell processor both types of cores share access to a common address space that includes main memory, address ranges corresponding to each SPE's local store, control registers, and I/O devices.

The PPE is a dual-threaded 64-bit PowerPC with a vector multimedia extension (VMX). It runs the operating system and manages the SPEs and includes a

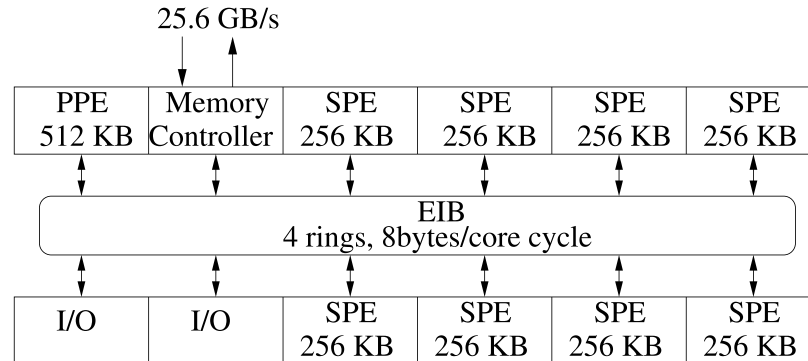

Figure 8.1: An overview of the Cell processor

conventional cache hierarchy with 32KB first-level instruction and data caches, and a 512KB second-level cache. As a standard 64-bit PowerPC processor, the PPE can run existing 64-bit PowerPC binaries.

Each SPE contains a synergistic processing unit (SPU) and a memory flow controller (MFC). The SPU is an in-order SIMD architecture capable of operating on 128-bit vectors and is optimized for computation-intensive applications. The SPEs have their own local storage (256 KB) from which they fetch instructions and read and write data. They have a simple in-order implementation to save power and area. Data and instructions are transferred between the PPE and SPEs using a DMA controller that is part of the MFC. The MFC includes the DMA controller and a memory management unit (MMU). The DMA controller unit consists of two DMA queues for transferring data between the SPEs and between the SPE and the PPE, one for SPE-initiated DMA and another for PPE-initiated DMA commands.

The MFC has a MMU responsible for address translation and protection using segment and page tables. A DMA transaction between a SPE and the PPE involves a data transfer between a local store address on the SPE and an effective address on the system memory. For this transfer the MFC obtains the system's memory effective address through a virtual address translation using its page table. The first DMA

transfer between the SPE and the PPE causes a page table fault in the MFC. The DMA engine also transfers data directly between the local stores of two SPEs and inside its SPE.

8.3 Methodology and Simulation Environment

The focus of this section is the study and quantification of the latency and throughput of different data-transfer techniques.

The first phase involves the study of the communication overhead to transfer data (send or receive) between the PPE and the SPEs. There are two general data transfer methods involving the PUT and GET functions. The GET function transfers data from the PPE to the SPEs. The PUT function transfers data from the SPEs to the PPE. These functions can be initiated from either the PPE or the SPE side and employ different mechanisms to set up and transfer data between the cores.

These different data transfer mechanisms result in different characteristics in terms of setup, latency, and throughput of the data transfer channels. To gain more insight we explored different components contributing to the communication overhead in these techniques. These included the DMA issue and set-up times, latency, data-delivery time, memory-management overhead, and synchronization between cores. The total data transfer time can be expressed as a function of several components, as shown in Figure 8.2, which include:

- First byte arrival time (Latency): a lower bound on the delay incurred in transferring a message containing a byte from its source core to its destination core.
- Data Delivery: defined as the time the processor needs to transfer the arrived message to its final destination.

- DMA overhead (DMA set up time): defined as the time to enqueue a DMA transfer.
- Gap: the minimum time interval between consecutive message arrivals in a non-blocking transfer.

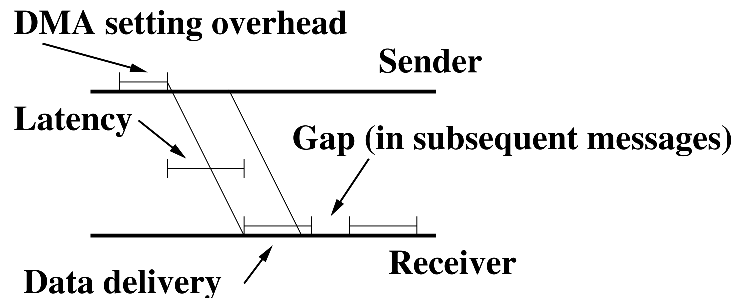

Figure 8.2: Components of data transfer

To quantify these components, several micro-benchmarks were written. The DMA issue time is established easily by measuring the time elapsed between issuing a DMA function (PUT or GET) and a return from this function. The onboard timer is used to measure the elapsed time.

The data delivery time is established by determining the arrival times of the first and last elements of a message by continuously monitoring the destination buffer. Determining the remainder of these components (the first byte delivery and the MMU overhead) is more complex. The methods are discussed in the subsequent sections.

8.3.1 First-Byte & Last-Byte Delivery Measurement

This section explains the methodology used in evaluating the latency quantity of first-byte delivery. First-byte delivery is defined as the time elapsed from issuing a DMA command to receiving the first byte of the message. This approach detects the arrival of the first byte by checking the contents of the destination buffer.

Since PUT and GET exhibit long issue times on the PPE, we elected to implement a two-threaded benchmark where the first thread implements the data transfer and the second (in a tight loop) determines the time of first-byte-arrival. The pseudo code depicted in Table 8.1 shows the implementation when the initiator and the target of the data transfer are on the PPE. It should be noted that the processor switches between two threads, which takes some time for these context-switches. Thus, if the context-switch overhead is comparable to the measured components it will affect the correctness of this measurement. This is discussed further in section 8.4.1.

Table 8.2 shows the pseudo code of the micro-benchmark implementation when the initiator resides on the SPE while the target of the transfer is the PPE. Similar benchmarks were developed which utilize the PUT and GET functions in the opposite directions to those depicted in Tables 8.1 and 8.2.

The micro-benchmark depicted in Table 8.2 determines times at two different entities (the PPE and the SPE). The associated timers need to be synchronized. The synchronization mechanism is discussed in the following subsections. The pseudo code is only shown for the first-byte arrival time on the PPE in the SPE-initiated GET function. Similar benchmarks were developed to detect the arrival of the last byte.

8.3.2 Synchronizing the PPE and the SPEs timers

As the associated timers of the SPEs and the PPE are located on each core and run independently, they need to be synchronized; this is crucial for accurate measurements. The main concern that needs be taken into account is the accuracy of this synchronization. To reach this objective, mailbox communications are selected that directly access the SPEs' problem state. This approach is chosen because of

Table 8.1: Pseudo code for detecting first byte delivery on the PPE (PPE-initiated)

The PPE's Code	Thread's code (on the PPE)
	do{
`mutex_lock( )`	`mutex_lock( )`
`ChkStatus = 1`	`Status = ChkStatus`
`mutex_unlock( )`	`mutex_unlock( )`
`spe_mfcio_put ( )`	}**while**`(Status == 0)`
do{	
`mutex_lock( )`	`do {`
`Status = ChkStatus`	`}while(!Buffer[0])`
`mutex_unlock( )`	`BaseTime = GetTimer()`
}**while**`(Status == 1)`	`mutex_lock( )`
	`ChkStatus = 0`
	`mutex_unlock( )`

Table 8.2: Pseudo code for detecting first byte delivery on the PPE (SPE-initiate)

The PPE's Code	The SPE's Code
`Synch. Part on the PPE`	
`Finding times relation`	`Synch. Part on the SPE`
`do {`	`StartOfDMA = GetTimer()`
`}while (!Buffer[0])`	`mfc_put(parameters)`
`BaseTime = GetTimer()`	`Wait for the ack.`
`Rcv. StartOfDMA`	`Send StartOfDMA to PPE`
`Find arrival Latency`	

its short round-trip latency between the PPE and the SPEs, which is from 10 to 14 cycles. It enabled us to find the times on each core with high accuracy and the least delay. The code in Table 8.3 is part of the synchronization approach; the data is written into the mailbox using the *_spe_in_mbox_write()* function, which directly accesses the SPEs' problem state. The start time on the SPE is obtained from the time we receive data in the mailbox on the SPE and the round-trip time, which is evaluated by a ping-pong method.

Table 8.3: Pseudo code for synchronizing the timers on the PPE and the SPEs

The PPE's Code	The SPE's Code
`time0 = GetTime()`	
`_spe_in_mbox_write()`	`spu_read_in_mbox()`
`SpeT=_spe_out_mbox_read()`	`T = GetTime()`
`time1=GetTime()`	`spu_write_out_mbox(T)`
`R-T=time1 - time0`	
`StartSpeT=SpeT+R-T/2`	

8.3.3 Memory Management Overhead

Memory management overhead plays a significant role in the latency when a transfer from a new source to a new target destination is required.

First, we examine the miss penalty of the Translation Look-aside Buffer (TLB) on the SPEs. In each data transfer between the PPE and the SPE the effective address of the memory location on the PPE is obtained through the TLB in the MFC, which is a 256-entry table, and each entry has the mapping address of a 4K page in the main memory of the PPE. If there is no entry in the TLB for the address of the referenced memory location, this will result in a TLB miss. Then the memory management

system will find the corresponding effective address for the memory location and will fill out the corresponding entry in the TLB for future access.

To measure the *TLB miss penalty*, the micro-benchmark uses the fact that the first transfer causes a cold miss in the TLB, while the second transfer of the same buffer will result in a hit. Therefore, the difference between these two cases is equal to the memory-management overhead.

Table 8.4: Pseudo code for the TLB miss penalty

The SPE's Code

```
StartOfDMA1 = GetTimer()
mfc_get(parameters)
Wait for the completion
EndOfDMA1 = GetTimer()
StartOfDMA2 = GetTimer()
mfc_get(parameters)
Wait for the completion
EndOfDMA2 = GetTimer()
TLB-Miss-Penalty = (EndofDMA1 - StartOfDMA1)
                 - (EndOfDMA2 - StartofDMA2)
```

Another quantity measured was the replacement penalty because of a conflict in the TLB. For this a micro-benchmark was written to generate addresses that result in the same entry in the TLB table. As stated, the first access to the TLB table results in a miss, while the following accesses with the same address result in a hit. However, accessing the TLB with different addresses that refer to the same entry in the table causes a replacement. Therefore, in the case of conflicts in the TLB table the difference between the times of a TLB hit and the following replacement is equal

to the replacement penalty. This approach is almost the same as the code shown in Table 8.4

8.3.4 Experimental Setup

The aforementioned methods were used to measure the different components of various data transfer techniques. The experiments were conducted on a 3.2 GHz PS3 running Linux Kernel 2.6.16. For all experiments we used C language intrinsics with the libspe2.1 library. The SPU decrementer register was used to measure the elapsed time in the SPEs, which ticks every 12.5ns. The time-base register was used in the PPE for measuring elapsed times; it has the same tick rate as the SPU decrementer. All experiments were run 20000 times and the minimum results were obtained using the k-best algorithm.

The following section describes the techniques and experimental results.

8.4 Evaluation

Having prepared the environment, we ran the micro-benchmarks and explored the incurred latency and overhead during data transfers.

While in section 8.3 the methods of achieving synchronization and determining elapsed times were discussed. Here, the main objective is to determine the communication components that play a role in delivering a message. The following mechanisms were considered:

- Transferring data between the PPE and the SPE using the PUT and GET functions (SPE-initiated and PPE-initiated).
- Transferring data between the SPEs.
- Using Mailbox for data transfer.

These mechanisms have different characteristics in terms of latency, data-delivery, DMA setting overhead, and receiving rates. In some cases, in spite of having a high-bandwidth communication path, other factors impose restrictions on reaching high throughput in transferring data. These situations are explored in the following sections.

8.4.1 DMA Transfer Time between the PPE and SPEs

This section presents the results for transferring data using DMA between the PPE and SPEs. First, the micro-benchmarks are run in a blocking mode; that is, we need to wait for the termination of each data transfer and receive an acknowledgement from the receiver. Subsequently, the non-blocking mode of transfer is considered. In this mode, several DMAs are initiated without waiting for the arrival of individual acknowledgements. We call this a *batched mode*. For example, if we issue *n DMA* transfers before waiting for the acknowledgement, we will have a *batch-n* data transfer.

GET Function Characteristics

The different components of data transfers were measured using the GET function, initiated on the PPE and the SPEs. These measurements were obtained for the blocking and non-blocking methods. Figures 8.3 and 8.5 show the total data transfer time *per message* using the GET function between the PPE and SPEs. As can be observed, the latency of the SPE-initiated GET function is much less than that of the PPE-initiated function. In order to investigate this behavior further the components of these data transfers were quantified.

Figures 8.4 and 8.6 show the different components of data transfer. To recognize dominant factors in each of these figures the total time of data transfer for all messages

is illustrated. For example, a batch-2 data transfer shows the elapsed time from issuing the first message to the arrival of the last byte of the second message.

Figures 8.4 and 8.6 include the overhead of setting up a DMA transfer in the GET function, first message arrival (latency), data delivery, and the gap between successive messages.

Note that Figures 8.5 and 8.6 depict different time intervals. While Figure 8.5 shows the total transfer time per message as measured at the PPE, Figure 8.6 shows the components of the transfer as observed at the SPE. These components exclude the acknowledgement received by the PPE after the completion of the data delivery of the last message at the SPE, and reflect the transfer time taken by all n messages in a *batch-n*. The difference in these two observations yields the acknowledgement time at 5μs.

As can be observed from the figures, the main component of the SPE-initiated GET function is data delivery. In other words, the DMA setup time on the SPE, which is accomplished by enquing the request through channels, is fast enough so that there is no bottleneck in this data delivery. However, the PPE-initiated GET function shows long DMA-issue time, as well as a long gap in receiving successive messages.

PUT Function Characteristics

This section investigates the various components of data transfers using the PUT function initiated on the PPE, as well as the SPEs. Figures 8.7 and 8.9 show the total data transfer time per message between the PPE and the SPEs using the PUT function. As can be observed, the latency of the SPE-initiated PUT function is much less than the PPE-initiated function. As in the previous section, the components of these data transfers are also quantified.

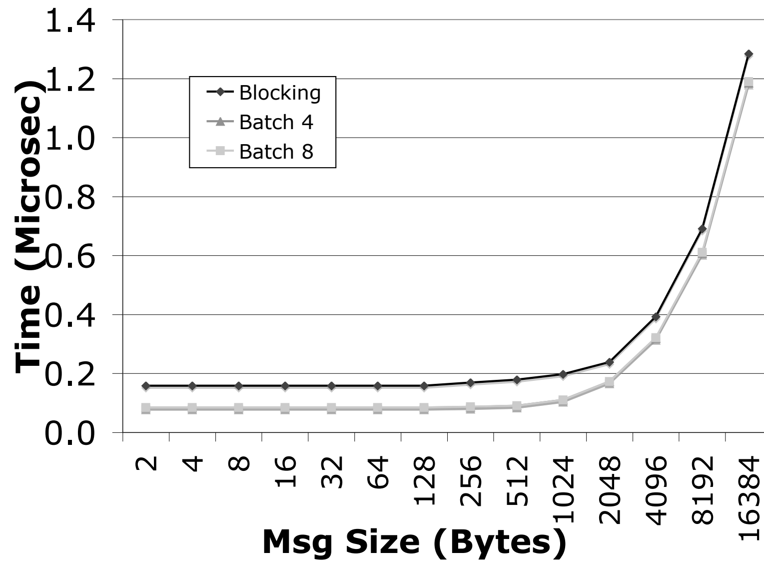

Figure 8.3: Total data transfer time per message using the GET function (SPE-initiated)

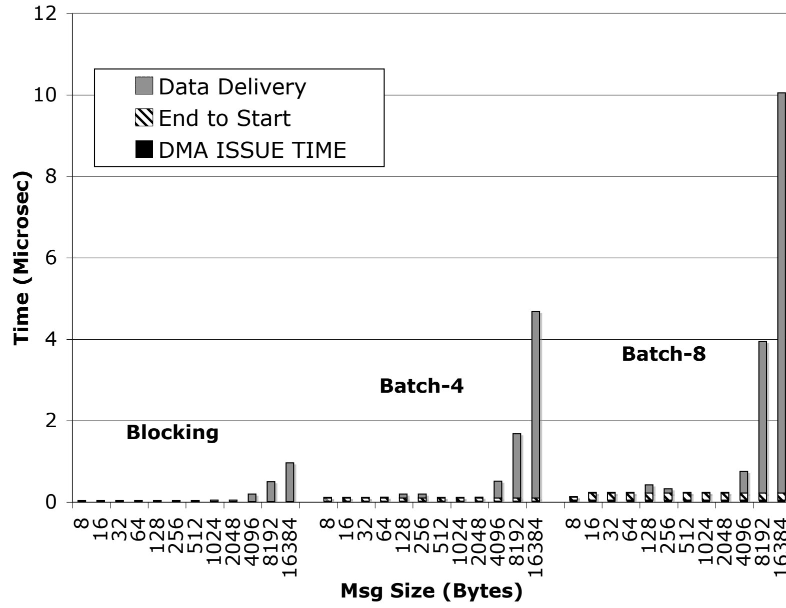

Figure 8.4: Accumulative data delivery components of the GET function (SPE-initiated)

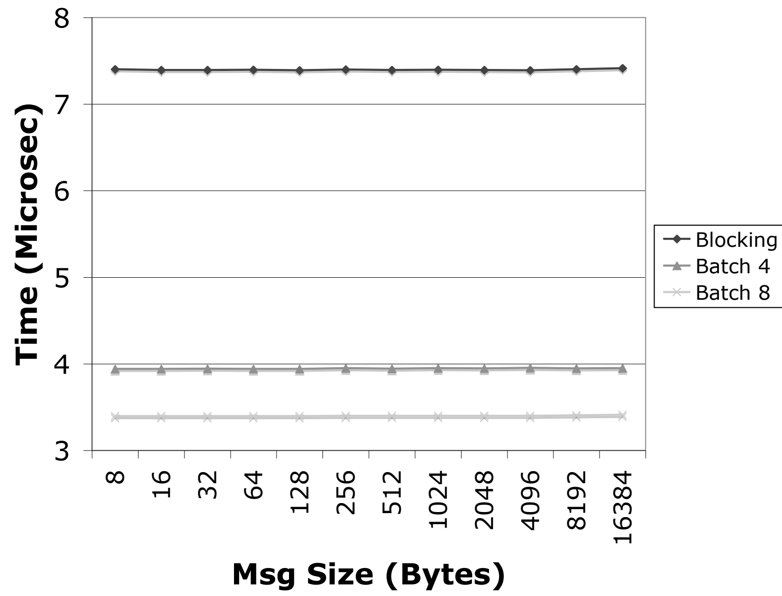

Figure 8.5: Total data transfer time per message using the GET function (PPE-initiated)

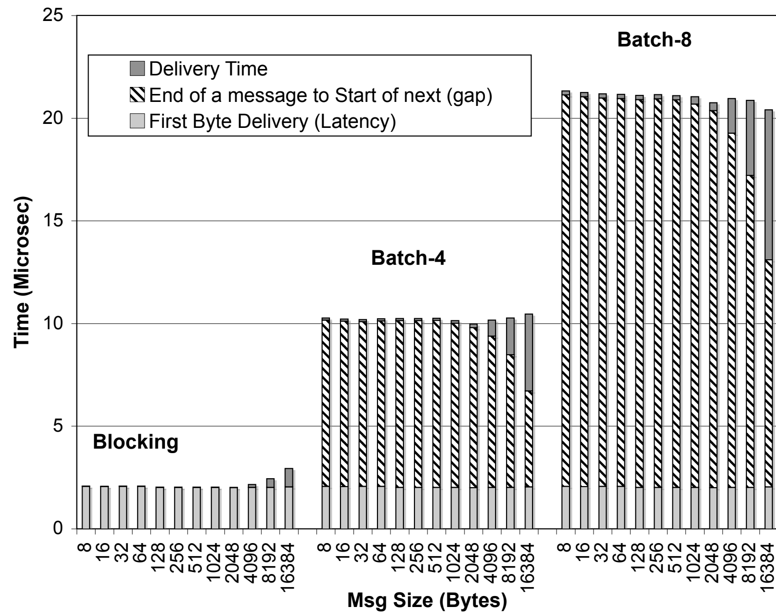

Figure 8.6: Accumulative data delivery components of the GET function (PPE-initiated)

Figures 8.8 and 8.10 include the overhead of setting up a DMA transfer in the PUT function, first message arrival time (latency), data delivery, and the gap between successive messages. As with the GET function, we show the total time of data transfer for all messages. To find the latency of data transfer per message we need to divide the total time by the number of sent messages. For example, in a batch-8 transfer we should divide the total time by 8.

It should be pointed out that the results in Figures 8.9 and 8.10 were achieved using different methods. Figure 8.9 depicts the latency of the PUT function by measuring the elapsed time from issuing it to receiving the corresponding acknowledgement on the PPE. However, Figure 8.10 represents the different components of the data transfer using the PUT function through a different thread on the PPE. By observing the figures, if the acknowledgement is received after the arrival of the first message and before the completion of the whole data transfer, it will be possible to describe the existing discrepancies. We speculate that the indicated discrepancy is related to the arrival of the acknowledgement before the completion of the data transfer on the PPE. This depends upon the implementation of the data transfer libraries on the Cell BE processor.

As mentioned, a two-threaded benchmark was implemented where the first thread accomplishes the data transfer and the second (in a tight loop) determines the time of first-byte-arrival. A potential problem with this approach is the context-switch overhead between the two threads that might affect the accuracy of the measurement. Therefore, the achieved results were compared using two approaches: the regular single-threaded blocking method and the two-threaded method. If the context-switching overhead is comparable to the other components, we should expect a significant discrepancy between the results obtained from these two methods. It is clear that the results achieved for the blocking situations (see Figures 8.9 and 8.10) are almost identical, which indicates the overhead of the associated context-switch

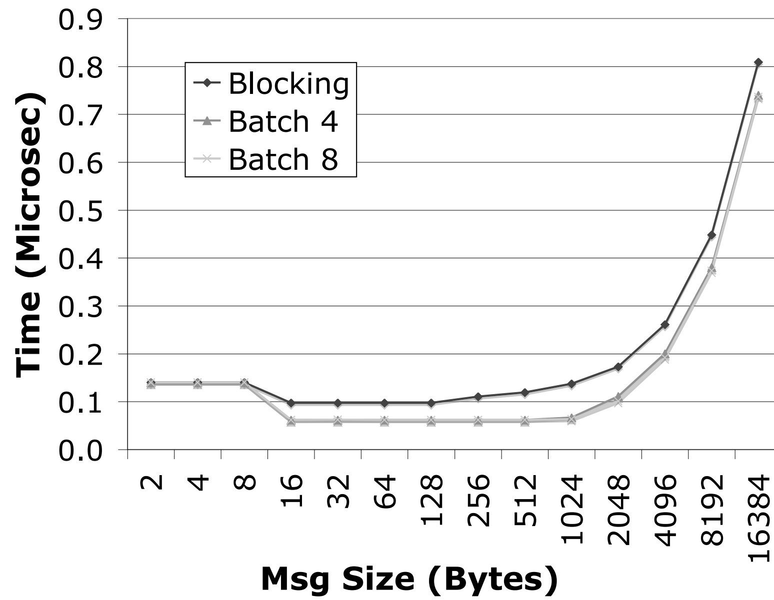

Figure 8.7: Total data transfer time per message using the PUT function (SPE-initiated)

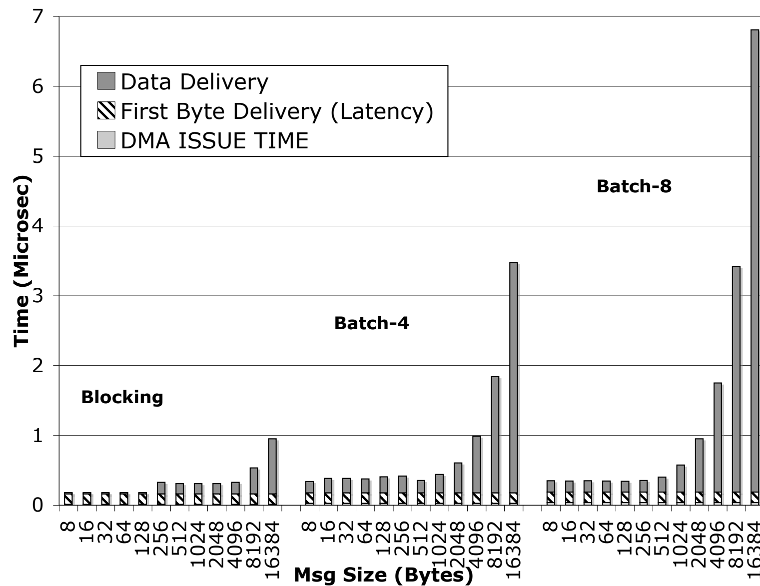

Figure 8.8: Accumulative data delivery components of the PUT function (SPE-initiated)

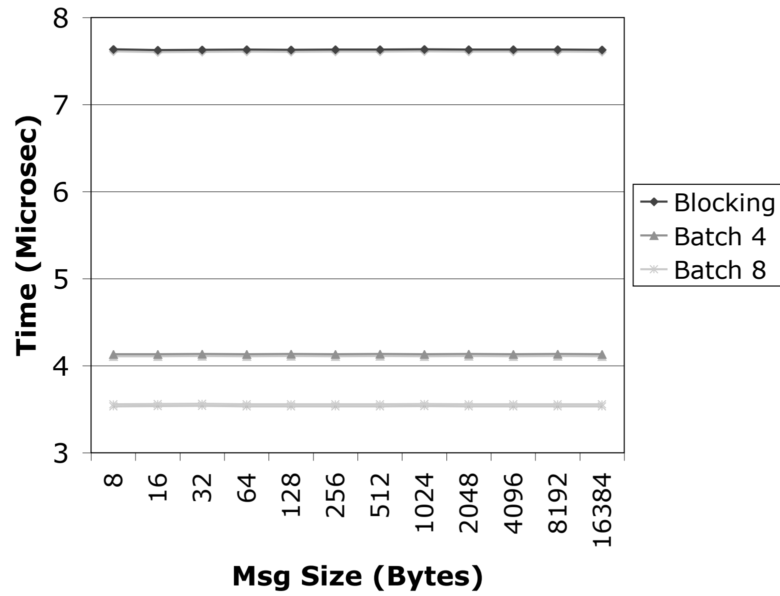

Figure 8.9: Total data transfer time per message using the PUT function (PPE-initiated)

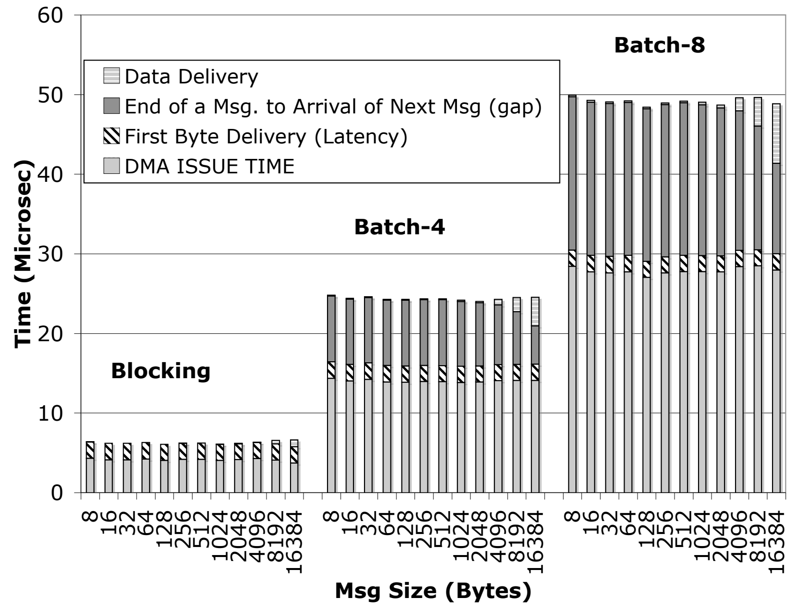

Figure 8.10: Accumulative data delivery components of the PUT function (PPE-initiated)

between the two threads in the two-threaded approach is negligible in comparison to the other data transfer components.

It is clear that the main components of the SPE-initiated PUT function are data delivery and latency (the time to receive the first byte). There are two main concerns in regard to the PPE-initiated PUT function: the DMA issue time and the latency. As a result of these observations, some latency hiding techniques need to be considered for PUT functions on the PPE side.

8.4.2 DMA Latency among and inside the SPEs

This section measures the latency of transferring data using DMAs between different SPEs. Synchronization through mailboxes among different SPEs is used to start the data transfer. The local store area address of destination to source is also transferred through DMA. PUT and GET operations are used for this purpose. Note that the results depicted in Figures 8.11 and 8.12 match the results in [81]. We summarize these results with other techniques in Table 8.8.

The latency of data transfer inside each SPE was also investigated using DMA transfer and copy operations (processor load and store instructions). This measurement, shown in Table 8.5, will help us choose the most efficient technique to transfer and bind the arrived message to its destination thread on the SPEs.

Table 8.5: Latency to transfer data to the same SPE using DMA and copy operations (μs)

Method	16B	1KB	4KB	8KB	16KB
Using DMA	0.070	0.075	0.10	0.23	0.74
Using Copy	0.0022	0.091	0.28	1.1	4.2

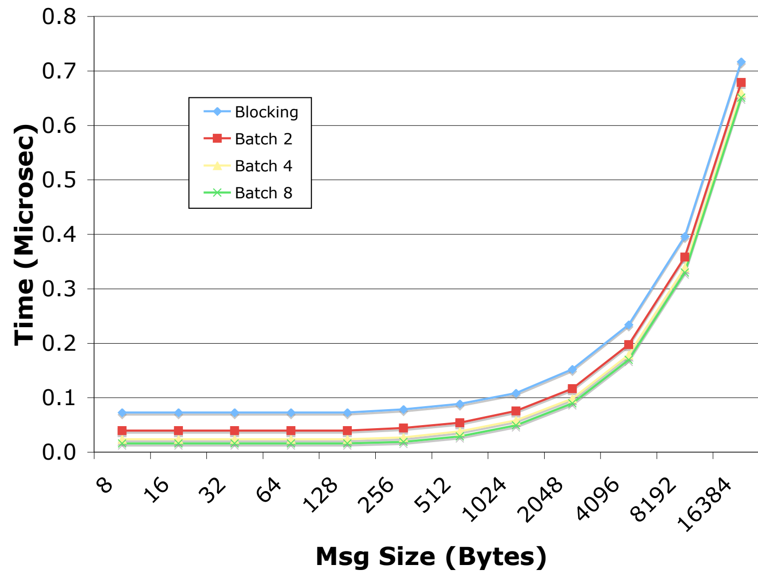

Figure 8.11: Latency to transfer data from one SPE to another SPE (GET)

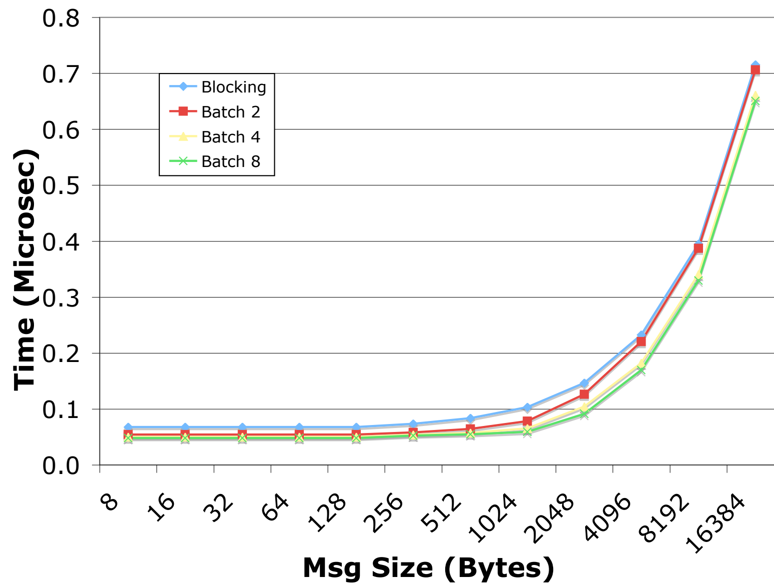

Figure 8.12: Latency to transfer data from one SPE to another SPE (PUT)

8.4.3 Mailbox Communication

Mailbox Communication is another mechanism used to transfer data (32 bits) between the PPE and the SPEs in the Cell Processor. For this purpose the latency of the mailbox communication was measured in two different situations: using library functions and directly accessing the SPE's problem state.

Table 8.6 shows the latency of the mailbox communication in the above-mentioned situations. As can be inferred from the figure, mailbox communication through directly accessing the SPE's problem state is one order of magnitude faster than using the library function. However, the former method is an unprotected access which needs more caution.

Table 8.6: Mailbox communication behavior

Method	Latency (μs)
Using Library Functions	5.69
Using Direct Access to SPE Problem State	0.279

8.4.4 Address Translation Behavior

This section examines the address translation behavior in data transfer between the PPE and the SPEs.

First explored was the miss penalty of Translation Look-aside Buffer (TLB) on the SPEs. Second, the replacement penalty was measured due to a conflict in the TLB table. Table 8.7 shows the results of these experiments for different message sizes.

Table 8.7: TLB miss and replacement policy (μs)

	16B	256B	1KB	4KB	8KB	16KB
TLB Miss Penalty	0.15	0.15	0.15	0.13	0.29	0.42
Repl. Penalty	36.5	37.0	36.5	36.8	69.4	146.0

8.5 Summary

This chapter presents the latency and memory management overhead incurred during the transfer of data between the PPE and the SPEs, as well as between the SPEs in the Cell processor. A summary of the achieved results is provided in Table 8.8. In consideration of this summary different situations were distinguished in order to transfer data among cores in the Cell processor.

Table 8.8: Summary of data transfer latencies for blocking cases (μs)

Method	16B	1KB	4KB	8KB	16KB
PPE-initiated GET	7.43	7.43	7.43	7.43	7.47
SPE-initiated GET	0.15	0.19	0.39	0.69	1.28
PPE-initiated PUT	7.6	7.6	7.6	7.6	7.6
SPE-initiated PUT	0.10	0.15	0.28	0.49	0.89
SPEtoSPE (GET))	0.072	0.21	0.23	0.39	0.71
SPEtoSPE (PUT)	0.067	0.10	0.23	0.39	0.71
Same SPE (DMA)	0.070	0.075	0.10	0.23	0.74
Same SPE (Copy)	0.0022	0.091	0.28	1.1	4.2

To send data from the PPE to the SPEs, two cases can be distinguished: long and short messages. For long messages, the SPE-initiated GET function has the

least overhead in transferring data from the SPE to the PPE. The address of the destination buffer is usually sent through the parameters during the spawning of the SPE thread; therefore, the SPE does not need to communicate with the PPE to obtain the destination buffer. However, it is likely that the destination buffer changes during execution time. In this case, mailbox communication can be employed, which takes 0.27μs, to receive the destination buffer instead of issuing a long latency GET function on the PPE. For short messages (less than or equal to 4B), mailbox communication is an extremely efficient method of transferring data from the PPE to the SPE.

To send data from the SPEs to the PPE we again distinguish between long and short messages. For long messages, as observed in Table 8.8, the SPE-initiated PUT function has the least overhead provided that the SPE is responsible for the data transfer and knows the target buffer. However, if the PPE is responsible for the data transfer there are two possible ways for this to occur. First, we might employ a mailbox communication to send the target address to the SPE to initiate the transfer. Second, if we predict the pattern of data consumption on the PPE we can issue the data transfer ahead of time to tackle the PUT function's bottleneck, which is the DMA issue time, as shown in Figure 8.10.

In the case of transferring data from a memory location on the PPE (or even from another SPE) to a SPE for the first time, the results show that the TLB miss penalty is from 0.15 μs to 0.4 μs, which is comparable to the achieved data transfer times. This high overhead is attributed to memory management operations in the PPE in order to allocate the required memory and to set appropriate tables for address mapping on the SPEs. The TLB latency therefore needs to be hidden. This can be achieved by pre-allocating TLB entries and by using prediction [17] to manage this allocation.

As stated, the results achieved use two different timers (the PPE and the SPE) when the initiator and the receiver are located on different cores. In this case we

had to synchronize these timers and used mailbox communication through the SPEs' problem state for this purpose. The measured round-trip time for this communication is from 10 to 14 timer cycles (0.120 μs to 0.175 μs), which is quite short in comparison to our measurements. Therefore, we can be confident that the results have a high degree of accuracy.

Chapter 9

Conclusions and Future Work

As is known, low-latency and high-bandwidth communication, in conjunction with data sharing among processing elements, is critical in obtaining high performance in cluster environments that use message passing for their communication. The raw bandwidth of networks has increased significantly in the past few years and networking hardware supporting bandwidth on the order of gigabits per second has become available. However, traditional networking architecture and protocols do not allow the applications to benefit from the available raw hardware interconnect performance. Latency is the main bottleneck in traditional networking architecture and protocols. The main components of latency are the speed limitations of transmission media and extra copies that are required to move the data to its final destination. The layered nature of the legacy networking software, the use of expensive system calls, and the extra memory-to-memory copies make the processor wait for the completion of the communication component before resuming its execution. Therefore, it has been necessary to tackle the unavoidable latency problem in order to leverage the potential of existing high-speed and low-latency networks, as well as fast processors.

We dealt with the messages of a length identical to that of a cache line in message passing environments. Two factors can be enumerated as the source of this decision.

First, as data size in messages becomes small the overhead of the data transfer is dominated by communication overhead to receive (send) the data from (to) the network. Second, as the applications are distributed to larger and larger systems, their granularities become smaller and the number of short messages increases. Thus, improvement of I/O communications for short messages has a drastic impact on the performance of applications.

The first contribution in this dissertation was to propose mechanisms to achieve true zero-copy data forwarding in message passing environments in order to hide the unavoidable latency problem. These mechanisms were proposed as processor extensions comprised of a specialized network cache and special instructions that help managing the operations of the cache and implementing the late binding. These extensions were to facilitate the placement of the arrived data in a cache, have it bound and ready to be consumed by the receiver.

To investigate the effectiveness of the proposed architectural enhancement a simulation environment was established by expanding an existing single-thread infrastructure to one that can run MPI applications. The proposed extensions were implemented along with the MPI functions on top of the SimpleScalar infrastructure, which can model a variety of platforms ranging from simple unpipelined processors to detailed dynamically-scheduled microarchitectures with multiple-level memory hierarchies.

This was accomplished in several steps, including examining the distribution of messages and collecting payloads to use in the simulator. Since the simulator required testing and an examination of the performance of a processor that communicates through message passing with other processes, a connection to the network was simulated. This was achieved through creating two different threads working simultaneously, one for simulating the network unit responsible for communicating messages with other processes in the network and the other for running the processor

that executes the application. We demonstrated that the established infrastructure could accurately simulate the used benchmarks. Thus, we were able to design the proposed extensions and conduct experiments in order to examine their behavior.

The effectiveness of the proposed extensions was verified in handling short messages with different arrival-time combinations in different situations. The *DTCT* data transfer mechanisms were compared to three classic data transfer mechanisms used to transfer the arrived data from the NIC to the buffer destination. Under the *DTCT* approach the data is transferred into the data cache upon binding. The classic techniques consider whether the processor should be responsible for transferring the data from the network interface memory to the process memory or whether the processor should use DMA to transfer data from the network interface to its final destination. It was shown that the received messages can be bound and transferred into the data cache and this does not affect the data caching characteristics of the application regardless of the message arrival times.

Next, the *lazy DTCT* data transfer mechanism was evaluated. With this technique, a message remains in the network cache (even after it is bound) and is transferred to the data cache only if it is bound and scheduled to be evicted from the network cache in the case of a replacement. The simulations showed that by using the proposed network extension along with the *lazy DTCT* we encountered fewer data cache misses compared to when the *lazy DTCT* was not used. We also demonstrated that the *lazy DTCT* provides a significant reduction in the access latency for the arrived messages in I/O intensive environments. The question of performance was answered, albeit indirectly. One would expect that an enhancement would ensure that the termination time of a given code is advanced, thus accomplishing a faster execution. This is not easily demonstrated in a parallel environment. The issue here is that the benchmark codes have been finely tuned so that the computation sections hide the communication thus ensuring optimum execution times for a given

architecture. Therefore, an improved communication environment would not impact the execution time of the application. Simply put, the computation sections will remain identical and dominate the run time, while the (shorter) communication sections will still be effectively hidden by the computation sections. The improvement lies in the fact that shorter communication sections need shorter computation sections in order to be hidden effectively thus allowing a finer granularity and an improved parallelism. This is where major gains in performance are achievable.

It should be noted that these results ensured the benefit of this policy per message base conservatively, in the sense that a real message passing environment can benefit from the proposed techniques more than the results shown in this dissertation. In a real message passing environment usually the last access times spill over several communication cycles and several messages are in use at some points. In addition, the improvement of the access latency affects the rest of computations as well as communication in a real parallel environment. Therefore, we expect to benefit from the improvements in the messages' last access time in the next arriving messages through receiving them earlier. However, constraints existed in the simulation environment do not allow us to benefit from these improvements because the simulation environment works under fixed message arrival times, which are statically obtained before running the applications. Therefore, these improvements do not affect the arrival time of the rest of messages. In other words, in a real message passing environment the improvement of the access time of each message accumulatively improves the performance of running applications much more than that we observed. Taking these into consideration, message passing applications in a real environment can benefit from the *DTCT* techniques severalfold more than the results shown in this section.

A further contribution involved comparing the execution time and speed up of *DTCT* approaches to M-VIA and TCP/IP. For this, we employed the Virtutec Simics

environment; a full system simulator, including processor cores; peripheral devices; memories; and network connections. We studied the possible overhead and cache pollution introduced through the operating system and the communications stack as exemplified by Linux, TCP/IP and M-VIA. Subsequently, the achieved data was utilized within the SimpleScalar simulation environment of our proposed extensions to obtain a realistic evaluation of the behavior of those extensions in a system environment. As explained in Chapter 7, the overhead in TCP/IP is almost 1.5 times more than that in VIA during receive operations, which is consistent with the fact that the TCP/IP protocol has a deeper protocol stack. Therefore, the performance of the extensions was compared to the more efficient VIA protocol. For this purpose, we compared the VIA approach to the proposed *DTCT* data transfer mechanisms showing that the *DTCT* and *lazy DTCT* approaches outperform M-VIA by 15 to 20%, respectively.

A final contribution in this dissertation involved an extended characterization of data transfers in the CELL BE processor. As discussed in Chapter 8, recently a set of factors, including poor performance/power efficiency and limited scalability in monolithic designs, has moved high-performance processor architectures toward designs that feature multiple processing cores on a single chip (CMP). The Cell processor, with its multitude of synergistic processing elements (SPEs) and their associated fast memories, has shown potential for use in high-performance computing that employs MPI as the de-facto programming model. Therefore, it is vital to implement MPI efficiently on the Cell BE processor to leverage its tremendous computational power. For this, an understanding of the data transfer mechanisms and associated components is necessary for an efficient implementation. This required studying the data transfer mechanisms between the processing elements of the Cell BE (the PPE and the SPEs) and identifying their communication capabilities in terms of latency and throughput for a variety of communication patterns. The ultimate goal

was to use this information together with Afsahi's prediction techniques to implement an efficient MPI environment.

Two general data transfer methods were studied, involving the *PUT* and *GET* functions. These functions can be initiated from either the PPE or the SPE side and employ different mechanisms to set up and transfer data between the cores. We showed that the SPE-initiated DMAs are faster than the corresponding PPE-initiated DMAs. We also investigated the main components of each data transfer.

In the SPE-initiated GET function, the main component is data delivery. However, the PPE-initiated GET function exhibits long DMA-issue time as well as a long gap in receiving successive messages. Our work has revealed that the main components of the SPE-initiated PUT function are data delivery and latency (the time to receive the first byte). However, there are two main concerns that need to be dealt with in the PPE-initiated PUT function: the DMA issue time and the latency.

We have also investigated whether the memory-management overhead is comparable to the data transfer time. The results achieved call for techniques to hide the unavoidable overhead in order to reach high-throughput communication in MPI implementation in the Cell BE.

9.1 Future Work

This dissertation has dealt with messages of a length identical to that of a cache line in message passing environments. Future research should extend the network cache mechanism to accommodate larger messages by dividing them into blocks. An annotating mechanism can be added to the network cache to keep track of messages that span more than one cache line. Whether the message that is kept inside the network cache should be treated as a whole message or as separate cache lines would need to be addressed. This mechanism and the associated parameters would need to

be investigated for the optimum configuration. In addition, very large messages could be handled in a similar manner by introducing special *network TLBs* that incorporate the *message ID* as well as process and network virtual addresses.

Another promising research direction involves applying the proposed techniques to multi-core processors. Since multi-core processors have emerged as the mainstream in processor architecture design, and taking into consideration the multitude of cores in contemporary processors, one approach to speed up data delivery in message passing environments could involve using a core as a communication helper. The proposed techniques in this dissertation could be extended to be used in multi-core processors. The L1 cache of the helper core corresponds to the proposed network cache in our techniques. Moreover, these proposed techniques, including message prediction, *DTCT*, and *lazy DTCT* can be tailored according to the requirements of modern processor architectures.

As stated in Chapter 8, the Cell BE processor has potential for use in high-performance computing that uses MPI as the de-facto programming model. Therefore, it is vital to implement MPI efficiently on the Cell BE processor. The challenge is to ensure data availability to the computations running on the Cell's cores. This data may exist locally on another SPE, on the PPE, or globally on another Cell BE processor in a cell blade. The implementation of primitive operations in the MPI environment is crucial, and this in turn requires understanding the data transfer mechanisms on the Cell BE processor. Some of these primitive operations include send, receive, wait, and collective operations.

It is worthwhile to further study the performance of collective operations in the Cell BE processor in single-port and multi-port modes. This research will involve both the SPEs and the PPE cores for collective communication operations and will encompass the design and implementation of *Broadcast* and *Total-Exchange* collective operations and the investigation of their latencies and associated components.

Bibliography

[1] "The earth simulator center." [Online]. Available: http://www.jamstec.go.jp/esc/

[2] "Grand Challing Applications." [Online]. Available: http://www.mcs.anl.gov/Projects/grand-challenges/

[3] "High Performance Fortran Homepage." [Online]. Available: http://www.vcpc.univie.ac.at/information/mirror/HPFF/

[4] "M-VIA: Virtual Interface Architecture for Linux." [Online]. Available: http://www.nserc.gov/research/FTG/via/

[5] "Message Passing Interface Forum: MPI 2.1." [Online]. Available: http://www.mpi-forum.org/

[6] "MPICH-A Portable Implementation of MPI." [Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/

[7] "MVICH: MPI for Virtual Interface Architecture." [Online]. Available: http://www.nersc.gov/research/FTG/mvich/index.html

[8] "Quad-Core AMD Opteron." [Online]. Available: http://multicore.amd.com/us-en/AMD-Multi-Core.aspx

[9] "Quadrics Interconnect Homepage." [Online]. Available: http://www.quadrics.com/

[10] "RDMA Consortium. Architectural Specifications for RDMA over TCP/IP." [Online]. Available: http://www.rdmaconsortium.org/

[11] "Silicon graphics inc." [Online]. Available: http://www.sgi.com

[12] "Sun microsystems." [Online]. Available: http://www.sun.com

[13] "Top 500 Supercomuter site." [Online]. Available: http://www.top500.org/

[14] "PSSP Command and Technical Reference–LAPI Chapter," IBM, Tech. Rep., 1997.

[15] M. Acacio, J. Gonzlez, J. Garca, and J. Duato, "Owner Prediction for Accelerating Cache-to-Cache Transfers in a cc-NUMA Architecture," in *Proceedings of the 2002 International Conference for High-Performance Computing, Networking Storage and Analysis, SC02*, Baltimore, USA, 2002.

[16] A. Afsahi, "Design and Evaluation of Communication Latency Hiding/Reduction Techniques for Message Passing Environments," Ph.D. dissertation, Department of ECE, University of Victoria, 2000.

[17] A. Afsahi and N. J. Dimopoulos, "Efficient Communication Using Message Prediction for Cluster of Multiprocessors," in *Proceedings of the 4th Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, CANPC'00, Lecture Notes in Computer Science, No. 1797*, January 2000, pp. 162–178.

[18] A. Afsahi and N. J. Dimopoulos, "Architectural Extension to Support Efficient Communication Using Message Prediction," in *Proceedings of the 16th Annual*

International Symposium on High Performance Computing Systems and Applications, June 2002, pp. 18–25.

[19] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, 1995, pp. 2–13.

[20] N. Agarwal and N. J. Dimopoulos, "Using CoDeL to Rapidly Prototype Network Processor Extensions," in *Proceedings of Computer Systems: Architectures, Modeling, and Simulation: 3rd and 4th International Workshops (SAMOS 2004)*, July 2004, pp. 333–342.

[21] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "SP2 System Architecture," *IBM Journal of Research and Development*, vol. 34, no. 2, pp. 152–184, 1995.

[22] G. Almasi and Gottlieb, *Highly Parallel Computing*, 1st ed. Benjamin/Cummings, 1989.

[23] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, "High Speed Switching for Local Area Networks," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System*, 1992, pp. 98–108.

[24] T. Anderson, D. Culler, and D. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, February 1995.

[25] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick, "Empirical Evaluation of the CRAY-T3D: A Compiler Perspective," in *Proceedings of*

the 22nd International Symposium on Computer Architecture (ISCA-22), S. Margherita Ligure, Italy, 1995, pp. 320–331.

[26] I. T. Association, "InfiniBand Architecture Specification." [Online]. Available: http://www.infinibandta.org

[27] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, February 2002.

[28] D. Bailey, T. Harris, W. Sahpir, and R. van der Wijngaart, "The NAS Parallel Benchmarks 2.0: Report NAS-95-020," Nasa Ames Research Center, Tech. Rep. NAS-95-020, 1995.

[29] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1991. [Online]. Available: citeseer.ist.psu.edu/bailey95nas.html

[30] M. Baker and R. Buyya, "Cluster Computing: The Commodity Supercomputing," *Journal of Software-Practice and Experience*, Jan. 1998.

[31] P. Balaji, S. Bhagvat, H.-W. Jin, and D. Panda, "Asynchronous Zero-copy Communication for Synchronous Sockets in the Sockets Direct Protocol (sdp) over InfiniBand," in *Proceedings of the Workshop on Communication Architecture for Clusters (CAC 06), held in conjunction with IPDPS 2006*, Rhodes Island, Greece, APRIL 2006.

[32] P. Balaji, W.-C. Feng, and D. Panda, "Bridging the Ethernet-Ethernot Performance Gap," *IEEE Micro*, vol. 26, no. 3, pp. 24–40, March 2006.

[33] M. Banikazemi, R. K. Govindaraju, R. Blackmore, and D. K. Panda, "MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems," *IEEE Transactions Parallel Distributed Systems*, vol. 12, no. 10, pp. 1081–1093, October 2001.

[34] E. Barton, J. Crownie, and M. McLaren, "Message Passing on the Meiko CS-2," *In Parallel Computing*, vol. 20, pp. 497–507, April 1994.

[35] P. Bellens, J. Perez, R. Badia, and J. Labarta, "CellSs: A Programming Model for the Cell BE Architecture," in *Proceedings of the 2006 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC06*, Tampa, FL, USA, 2006, pp. 86–96.

[36] N. Binkert, R. Dreslinski, E. Hallnor, L. Hsu, S. Raasch, A. Schultz, and S. Reinhardt, "The Performance Potential of an Integrated Network Interface," in *Proceedings of the 1st ANCHOR Workshop, ANCHOR2004, in conjunction with ISCA-31*, June 2004.

[37] N. Binkert, A. Saidi, and S. Reinhardt, "Integrated Network Interfaces for High-Bandwidth TCP/IP," in *Proceedings of the 2006 Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, October 2005, pp. 315–324.

[38] N. Binkert, L. Hsu, A. Saidi, R. Dreslinski, A. Schultz, and S. Reinhardt, "Performance Analysis of System Overheads in TCP/IP Workloads," in *Proceedings of the 2005 Conference on Parallel Architectures and Compilation Techniques (PACT 2005)*, September 2005.

[39] M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Lai, and M. R. Mesarina, "Virtual-memory-mapped Network Interfaces," *IEEE Micro*, pp. 21–28, February 1995.

[40] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. S, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, Februray 1995.

[41] J. Brustoloni, "Interoperation of Copy Avoidance in Network and File I/O," in *Proceedings of INFOCOM '99*, New York, USA, March 1999, pp. 534–542.

[42] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 1st ed. Prentice Hall, 2003.

[43] F. Cappelo and D. Etiemble, "MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks," in *Proceedings of the 2000 International Conference for High-Performance Computing, Networking Storage and Analysis, SC00*, 2000.

[44] J. B. Carter, A. Davis, R. Kuramkote, C.-C. Kuo, L. B. Stoller, and M. Swanson, "Avalanche: A Communication and Memory Architecture for Scalable Parallel Computing," in *Proceedings of the 5th Workshop on Scalable Shared Memory Multiprocessors*, 1995.

[45] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, 1st ed. Morgan Kaufmann, 1998.

[46] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler, "The Message-driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," *IEEE Micro*, vol. 38, no. 8, pp. 23–38, April 1992.

[47] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li, "VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication," in *Proceedings of the Hot Interconnect 97*, 1997.

[48] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture," *IEEE Micro*, pp. 66–76, March-April 1998.

[49] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen, "Simultaneous Multithreading: A Platform for Next-generation Processors," *IEEE Computer*, vol. 17, no. 5, pp. 12–18, September/October 1997.

[50] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo, "Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture," *IBM System Journal*, vol. 45, no. 1, pp. 59–84, 2006.

[51] A. Eichenberger, K. O'Brien, W. Peng, T. Chen, P. Oden, D. Prener, J. Shepherd, S. Byoungro, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing Compiler for a Cell Processor," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005)*, 2005, pp. 161–172.

[52] T. V. Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[53] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schause, "Active Messages: A Mechanism for Integrated Communication and Computation,"

in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA-19)*, May 1992, pp. 256–265.

[54] J. Engblom, D. Kagedal, A. Moestedt, and J. Runeson, “Developing Embedded Networked Products using the Simics Full-System Simulator,” in *Proceedings of the PIMRC 2005*, Berlin, Germany, September 2005.

[55] W.-C. Feng, J. Hurwitz, H. Newman, S. Ravot, R. Cottrell, O. Martin, F. Coccetti, J. Cheng, X. Wei, and S. Low, “Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters, and Grids: A Case Study,” in *Proceedings of the 2003 International Conference for High-Performance Computing, Networking Storage and Analysis, SC2003*, November 2003.

[56] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, “The M-machine Multicomputer, Tech. Rep. AIM-1532, 30, 1995. [Online]. Available: citeseer.ist.psu.edu/article/fillo95mmachine.html

[57] A. Foong, T. Huff, H. Hum, J. patwardhan, and G. Regnier, “TCP Performance Re-visited,” in *Proceedings of 2003 IEEE International Symposium Performance Analysis of Systems and Software*, March 2003.

[58] S. Frank, H. Burkhardt, and J. Rothnie, “The KSR1: Bridging the Gap Between Shared Memory and MPPs,” in *Proceedings of Compcon93*, February 1993, pp. 285–294.

[59] M. Galles, “Spider: A High-Speed Network Interconnect,” *IEEE Micro*, vol. 17, no. 2, January/Februray 1997.

[60] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, “Overview

of the Blue Gene/L System Architecture," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 195–212, 2005.

[61] D. Goldenberg, R. R. M. Kagan, and M. Tsirkin, "Zero Copy Sockets Direct Protocol over InfiniBand–Preliminary Implementation and Performance Analysis," in *Proceedings of the 13th Symposium on High Performance Interconnects*, 2005, pp. 128–137.

[62] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2003.

[63] S. Hioki, "Construction of Staples in Lattice Gauge Theory on a Parallel Computer," *Parallel Computing*, vol. 22, no. 10, pp. 1335–1344, October 1996.

[64] S. Hioki, "QCDMPI – pure QCD Monte Carlo Simulation code with MPI," *Nuclear Physics B - Proceedings Supplements*, vol. 63, pp. 1000–1002(3), April 1998.

[65] R. W. Horst and D. Garcia, "The SCX Channel: A New, Supercomputer-class System Interconnect," in *Proceedings of the Hot Interconnects III*, 1995.

[66] R. W. Horst and D. Garcia, "ServerNet SAN I/O Architecture," in *Proceedings of the Hot Interconnects V*, 1997.

[67] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, Madison, USA, June 2005, pp. 50–59.

[68] K. Kant, "TCP Offload Performance for Front-end Servers," in *Proceedings of 2003 Global Telecommunications Conference, GLOBECOM '03*, December 2003, pp. 3242–3247.

[69] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron Processor for Multiprocessor Servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, 2003.

[70] J. Khale, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal Research and Development*, vol. 49, no. 4/5, pp. 589–604, 2005.

[71] F. Khunjush and N. J. Dimopoulos, "Hiding Message Delivery and Reducing Memory Access Latency by providing Direct-to-Cache Transfer during Receive Operations in a Message Passing Environment," in *Proceedings of the 6th International Workshop on MEmory performance: DEaling with Applications , systems and architecture (MEDEA 05), in conjunction with PACT05*, St. Louis, USA, September 2005.

[72] F. Khunjush and N. J. Dimopoulos, "Hiding Message Delivery and Reducing Memory Access Latency by providing Direct-to-Cache Transfer during Receive Operations in a Message Passing Environment," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 1, pp. 41–48, March 2006.

[73] F. Khunjush and N. J. Dimopoulos, "Architectural Enhancements for Message Passing Interconnects," in *Poster Abstracts. Advanced Computer Architecture and Compilation for Embedded Systems*, July 2005, pp. 231–235.

[74] F. Khunjush and N. J. Dimopoulos, "Evaluation of Direct-To-Cache Transfer during Receive Operations in a Message Passing Environment," in *Proceedings of the 2nd International Workshop on Advanced Networking and Communications Hardware, ANCHOR2005, in conjunction with ISCA-32*, Madison, USA, June 2005, pp. 41–48.

[75] F. Khunjush and N. J. Dimopoulos, "Lazy Direct-to-Cache Transfer during Receive Operations in a Message Passing Environment," in *Proceedings of the 3rd ACM International Conference on Computing Frontiers, CF'06*, Ischia, Italy, May 2006, pp. 331–340.

[76] F. Khunjush and N. J. Dimopoulos, "Comparing Direct-to-Cache Transfer policies to TCP/IP and M-VIA during Receive Operations in MPI Environments," in *Proceedings of the 5th International Symposium on Parallel and Distributed Processing and Applications (ISPA 2007), Lectur Notes in Computer Science (4742)*, Niagra Falls, Canada, August 2007, pp. 208–222.

[77] F. Khunjush and N. J. Dimopoulos, "Using the Cell Processor as a Network assist to minimize latency," in *Proceedings of IEEE 2007 Canadian Conference on Electrical and Computer Engineering*, Vancouver, Canada, April 2007, pp. 936–939.

[78] F. Khunjush and N. J. Dimopoulos, "Extended Characterization of DMA Transfers on the Cell BE processor," in *Proceedings of the 13th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-08), held in conjunction with IPDPS 2008*, Maimi, FL, USA, April 2008.

[79] F. Khunjush, M. W. El-Kharashi, K. F. Li, and N. J. Dimopoulos, "Network Processor Design: Issues and Challenges," in *Proceedings of 2003 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Aug. 2003, pp. 164–168.

[80] J. Kim and D. J. Lilja, "Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs," in *Proceedings of*

the Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, February 1998, pp. 206–216.

[81] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006.

[82] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multi-threaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[83] M. Krishna, A. Kumar, N. Jayam, G. Senthilkumar, P. K. Baruah, R. Sharma, S. Kapoor, and A. Srinivasan, "A Synchronous Mode MPI Implementation on the Cell BE Architecture," in *Proceedings of the 5th International Symposium on Parallel and Distributed Processing and Applications (ISPA 2007)*, Niagra Falls, Canada, August 2007, pp. 982–991.

[84] A. Kumar, N. Jayam, A. Srinivasan, G. Senthilkumar, P. K. Baruah, S. Kapoor, M. Krishna, and R. Sarma, "Feasibility Study of MPI Implementation on the Heterogenous Multi-Core Cell BE Architecture," in *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA07)*, San Diego, California, USA, June 2007, pp. 55–56.

[85] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. H. A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA-21)*, Chicago, IL, USA, April 1994, pp. 302–313.

[86] K. Lauritzen, T. Sawicki, T. Stachura, and C. Wilson, "Intel I/O Acceleration Technology Improves Network Performance, Reliability and

Efficiently." [Online]. Available: http://www.intel.com/technology/magazine/communications/Intel-IOAT-0305.pdf

[87] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "The DASH Prototype: Implementation and Performance," in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA-19)*, Gold Coast, Australia, May 1992, pp. 92–103.

[88] E. Lusk, "Programming with MPI on Clusters," in *Proceedings of International Conference on Cluster Computing*, October 2001, pp. 360–362.

[89] M. Martin, P. Harper, D. Sorin, M. Hill, and D. Wood, "Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-30)*, June 2003.

[90] C. McNairy and R. Bhatia, "Montecito: a Dual-core, Dual-thread Itanium Processor," *IEEE Micro*, vol. 25, no. 2, pp. 10–20, 2005.

[91] J. Mogul, "TCP Offload is a Dumb Idea Whose Time Has Come," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, USA, MAY 2003, pp. 25–30.

[92] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, April 1965.

[93] T. Mowry and A. Gupta, "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 87–106, 1991.

[94] S. S. Mukherjee and M. D. Hill, "A Survey of User-level Network Interfaces for Systems Area Networks," Computer Science Dept., Univ. of Wisconsin-Madison, Tech. Rep. Tech. Rep. 1340, 1997.

[95] S. S. Mukherjee and M. D. Hill, "Using Prediction to Accelerate Coherence Protocols," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, 1998.

[96] R. S. Nikhil, G. M. Papadopoulos, and Arvind, "*T: A Multithreaded Massively Parallel Architecture," in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA-19)*, 1992, pp. 156–167.

[97] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The J-machine Multicomputer: An Architectural Evaluation," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA-20)*, 1993, pp. 224–235.

[98] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "MPI Microtask for Programming the Cell Broadband Engine Processor," *IBM System Journal*, vol. 45, no. 1, pp. 85–102, 2006.

[99] G. M. Papadopoulos and D. E. Culler, "Monsoon: An Explicit Token Store Architecture," in *Proceedings of the 17th International Symposium on Computer Architecture (ISCA-17)*, May 1990, pp. 82–91.

[100] D. A. Patterson, "Latency Lags Bandwith," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, October 2004.

[101] I. Philp and Y.-L. Liong, "The Scheduled Transfer (ST) Protocol," in *Proceedings of the 3rd International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC'99)*, January 1999, pp. 108–121.

[102] R. Pierce and G. Regnier, "The Paragon Implementation of the NX Message Passing Interface," in *Proceedings of the Scalable High-Performance Computing Conference*, May 1994, pp. 184–190.

[103] L. Prylli and B. Tourancheau, "BIP: A New Protocol Designed for High Performance Networking on Myrinet," in *Proceedings of the PC-NOW98: International Workshop on Personal Computer based Network Of Workstations, in Conjunction with PPS/SPDP '98*, 1998.

[104] M. Rangarajan and A. Bohra, "TCP servers: Offloading TCP Processing in Internet Servers. Design, Implementation and Performance," Rutgers University, Tech. Rep., 2002. [Online]. Available: citeseer.ist.psu.edu/article/rangarajan02tcp.html

[105] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong, "TCP Onloading for Data Center Servers," *IEEE Computer*, vol. 37, no. 11, pp. 48–58, Nov. 2004.

[106] G. Regnier, D. Minturn, G. McAlpine, V. Saletore, and A. Foong, "ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine," in *Proceedings of the 11th Symposium on High Performance Interconnects*, August 2003, pp. 76–82.

[107] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-Level Shared Memory," in *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA-21)*, 1994, pp. 325–337.

[108] S. H. Rodrigues, T. E. Anderson, and D. E. Culler, "High-Performance Local Area Communication with Fast Sockets," in *Proceedings of USENIX 1997 Annual Technical Conference*, January 1997.

[109] A. Romanow and S. Bailey, "An Overview of RDMA over IP," in *Proceedings of the 1st International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2003)*, Feburary 2003.

[110] G. Shah and C. Bender, "Performance and Experience with LAPI– A New High-Performance Communication Library for the IBM RS/6000 SP," in *Proceedings of International Parallel Processing Symposium*, 1998.

[111] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "POWER5 System Microarchitecture," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 505–521, 2005.

[112] P. L. Springer, "PVM Support for Clusters," in *Proceedings of International Conference on Cluster Computing*, October 2001, pp. 183–186.

[113] C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varke, "The SP2 High-Performance Switch," *IBM Systems Journal*, vol. 34, no. 2, pp. 185–204, 1995.

[114] M. Welsh, A. Basu, X. W. Huang, and T. V. Eicke, "Memory Management for User-Level Network Interfaces," *IEEE Micro*, pp. 77–82, March/April 1998.

[115] P. H. Worley and I. T. Foster, "Parallel Spectral Transform Shallow Water Model: A Runtime-tunable Parallel Benchmark Code," in *Proceedings of the Scalable High Performance Computing Conference*, September 1994, pp. 207–214.