INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Design and Evaluation of Communication Latency Hiding/Reduction Techniques for Message-Passing Environments

by

Ahmad Afsahi
B.Sc., Shiraz University, Iran, 1985
M.Sc., Sharif University of Technology, Iran, 1988

A Dissertation Submitted in Partial Fulfillment of the
Requirements of the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

We accept this dissertation as conforming
to the required standard

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

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

Dr. V. K. Bhargava, Departmental Member (Department of Electrical and Computer Engineering)

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

Dr. J. Duato, External Examiner (Department of Information Systems and Computer Architecture, Technical University of Valencia)

Supervisor: Dr. Nikitas J. Dimopoulos

Abstract

With the availability of fast microprocessors and small-scale multiprocessors, inter-node communication has become an increasingly important factor that limits the performance of parallel computers. Essentially, message-passing parallel computers require extremely short communication latency such that message transmissions have minimal impact on the overall computation time. This thesis concentrates on issues regarding hardware communication latency in single-hop reconfigurable networks, and software communication latency regardless of the type of network.

The first contribution of this thesis is the design and evaluation of two different categories of prediction techniques for message-passing systems. This thesis utilizes the communications locality property of message-passing parallel applications to devise a number of heuristics that can be used to predict the target of subsequent communication requests, and to predict the next consumable message at the receiving ends of communications.

Specifically, I propose two sets of predictors: *Cycle-based* predictors, which are purely dynamic predictors, and *Tag-based* predictors, which are static/dynamic predictors. The performance of the proposed predictors, specially Better-cycle2 and Tag-bettercycle2, are very well on the application benchmarks studied in this thesis. The proposed predictors could be easily implemented on the network interface due to their simple algorithms and low memory requirements.

As the second contribution of this thesis, I show that majority of reconfiguration delays in single-hop reconfigurable networks can be hidden by using one of the proposed high hit ratio predictors. The proposed predictors can be used in establishing a communication pathway between a source and a destination in such networks before this pathway is to be used.

This thesis' third contribution is the analysis of a broadcasting algorithm that utilizes latency hiding and reconfiguration in the network to speed the broadcasting operation. The analysis brings up closed formulations that yields the termination time of the algorithms.

The thesis' fourth contribution is a new total exchange algorithm in single-hop reconfigurable networks. I conjecture that this algorithm ensures a better termination time than what can be achieved by either of the direct, and standard exchange algorithms.

The fifth contribution of this thesis is the use and evaluation of the proposed predictors to predict the next consumable message at the receiving ends of communications. This thesis contributes by claiming that these message predictors can be efficiently used to drain the network and cache the incoming messages even if the corresponding receive calls have not been posted yet. This way, there is no need to copy the early arriving messages into a temporary buffer. The performance of the proposed predictors, Single-cycle, Tag-cycle2 and Tag-bettercycle2, on the parallel applications are quite promising and suggest that prediction has the potential to eliminate most of the remaining message copies.

Examiners:

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

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

Dr. V. K. Bhargava, Departmental Member (Department of Electrical and Computer Engineering)

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

Dr. J. Duato, External Examiner (Department of Information Systems and Computer Architecture, Technical University of Valencia)

Table of Contents

List of Figures

List of Tables

Trademarks

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Trademarks and registered trademarks used in this work, where the author was aware of them, are listed below. All other trademarks are the property of their respective owners.

IBM SP2	is a registered trademark of International Business Machines Corp.
IBM Deep Blue	is a registered trademark of International Business Machines Corp..
IBM P2SC CPU	is a registered trademark of International Business Machines Corp.
IBM Vulcan Switch	is a registered trademark of International Business Machines Corp.
Myrinet	is a registered trademark of Myricom.
ServerNet	is a registered trademark of Tandem Division of Compaq.
SGI Origin 2000	is a registered trademark of Silicon Graphics, Inc.
SGI Spider Switch	is a registered trademark of Silicon Graphics, Inc.
WaveStar LambdaRouter	is a registered trademark of Lucent Technology.

Glossary

AM	Active Messages
ASCI	Accelerated Strategic Computing Initiative program
BIP	Basic Interface for Parallelism
BT	Block Tridiagonal Application Benchmark
CA	Communication Assist
CDF	Cumulative Distribution Function
CGH	Computer Generated Holograms
CIC	Computing, Information and Communications Project
CG	Conjugate Gradient Application Benchmark
CLUMP	Cluster of Multiprocessors
COW	Cluster of Workstations
DM	Deformable Mirrors
DSM	Distributed Shared-Memory Multiprocessor
EP	Embarrassingly Parallel Application Benchmark
FIFO	First-in-first-out
FM	Fast Messages
FT	3-D Fast-Fourier Transform Application Benchmark
HPF	High Performance Fortran
IS	Integer Sort Application Benchmark
LAM/MPI	Local Area Multicomputer/Message Passing Interface
LAN	Local Area Networks
LAPI	Low-level Application Programmers Interface

LIFO	Last-in-first-out
LRU	Least Recently Used
LU	Lower-Upper Diagonal Application Benchmark
MG	Multigrid Application Benchmark
MIMD	Multiple Instructions Multiple Data
MPI	Message Passing Interface
MPICH	A Portable Implementation of MPI
MPP	Massively Parallel Processors systems
NI	Network Interface
NOW	Networks of Workstations
NPB	NAS Parallel Benchmarks
ORPC (k)	Optically Reconfigurable Parallel Computer
OPS	Optical Passive Stars
P2SC	Power2-Super Microprocessor
POPS	Partitioned Optical Passive Stars
PM	A High-Performance Communication Library
PSTSWM	Power Spectrum Transform Shallow Water Model
PVM	Parallel Virtual Machine
QCDMPI	Quantum Chromodynamics with Message passing Interface
RON (k, N)	Reconfigurable Optical Network.
RMA	Remote Memory Access
SAN	System Area Networks
SEED	Self Electro Optics Emitting Device

SHRIMP	Scalable High-Performance Really Inexpensive Multiprocessor
SP	Scalar Pentadiagonal Application Benchmark
SPMD	Single Program Multiple Data
TLB	Translation Lookaside Buffer
U-Net	A User-Level Network Interface Architecture
VCC	Virtual Circuit Caching
VCSEL	Vertical Cavity Surface Emitting Laser
VIA	Virtual Interface Architecture
VMMC-2	Virtual Memory-Mapped Communications

Acknowledgments

I would like to express my deepest appreciation to my supervisor, Dr. Nikitas J. Dimopoulos for his thoughtful suggestions that shaped and improved my ideas. I am very grateful to Nikitas for providing me with his valuable guidance, encouragement, support, criticism, patience, and kindness from the first day I came to Victoria.

I would like to thank the members of my dissertation committee. I wish to thank Dr. Kin F. Li, Dr. Vijay K. Bhargava, and Dr. D. Michael Miller for their support and suggestions. I am very grateful to Dr. José Duato for his kind acceptance to be the external examiner of this dissertation, and for his brilliant suggestions.

I am greatly indebted to my wife, Azita Gerami for her continuous support and encouragements. Without her understanding, I would not have finished my dissertation. I would like to express my gratitude to my parents who always encouraged me to pursue a Ph.D.

I want to thank all my friends and graduate fellows especially the fellow researchers at LAPIS including André Schoorl, Nicolaos P. Kourounakis, Shahadat Khan, Mohamed Watheq El-Kharashi, Stephen W. Neville, Rafael Parra Hernandez, Caedmon Somers, Jon Kanie, and Eric Laxdal who have made my stay so much fun.

I would like to thank the department's system and office staff for their continuous cooperation. I am thankful to Vicky Smith, Lynne Barrett, Maureen Denning, and Moneca Bracken.

Special thanks to Dr. Murray Campbell at the IBM T. J. Watson Research Center for his kind cooperation and help in accessing the IBM Deep Blue, and the staff of the computer center at the University of Victoria for the access to the University IBM SP2.

My dissertation research was supported by grants from the Natural Science and Engineering Research Council (NSERC) of Canada, and the University of Victoria.

I dedicate this dissertation to my wife, Azita Gerami,
and to my parents, Abbas Afsahi, and
Ghodsieh Sakouie for their support and
encouragement through the years

Chapter 1

Introduction

Research in the area of advanced computer architecture has been primarily focused on how to improve the performance of computers in order to solve computationally intensive problems [32, 62, 69]. Some of these problems are called *grand challenges*. A grand challenge is a fundamental problem in science or engineering that has a broad economic and/or scientific impact; coupled fields, geophysical, and astrophysical fluid dynamics (GAFD) turbulence, modeling the global climate system, formation of the large scale universe, global optimization algorithms for macromolecular modeling, petroleum exploration, aerodynamic simulations, ocean circulation, are just a few to mention.

The performance of processors is doubling each eighteen months [62]. However, there is always a demand for more computing power. To solve grand challenge problems, computer systems at the *teraflop* (10^{12} floating point operations per second) and *petaflop* (10^{15} floating point operations per second) performance levels are needed.

Processors are becoming very complex and only a few companies are designing new processors. Therefore, it is not cost-effective to build high performance computers just by using custom-design high performance processors. The trend is to design parallel computers using commodity processors to achieve teraflop and petaflop performance. For instance, two major projects to develop high performance supercomputers in the USA are: the federal program in *Computing, Information and Communications* (CIC) project at the national coordination office [98], and the Department of Energy *Accelerated Strategic Computing Initiative* (ASCI) program including Intel/Sandia Option Red, IBM/Lawrence Livermore National Laboratory Blue Pacific, and SGI/Los Alamos National Laboratory Blue Mountain [39].

This should not give us the wrong impression that such high performance computers, often called *Massively Parallel Processor* (MPP) systems, are only used for grand challenges and parallel scientific applications. Even for applications requiring lower computing power, parallel computing is a cost-effective solution. These days, many high performance parallel computing systems are being used in network and commercial applications such as data warehousing, internet servers, and digital libraries.

Parallel processing is at the heart of such powerful computers. Although parallelism appears at different levels for a single processor system, such as lookahead, pipelining, superscalarity, speculative execution, vectorization, interleaving, overlapping, multiplicity, time sharing, multitasking, multiprogramming, and multithreading, but it is the parallel processing and parallel computing among different processors which brings us such levels of performance.

Basically, a parallel computer is a "collection of processing elements that communicate and cooperate to solve large problems fast" [9]. In other words, a parallel computer, whether *message-passing* or *distributed shared-memory* (DSM), is a collection of complete computers, including processor and memory, that communicate through a general-purpose, high-performance, scalable interconnection network using a *communication assist* (CA) and/or a *network interface* (NI) [32], as shown in Figure 1.1.

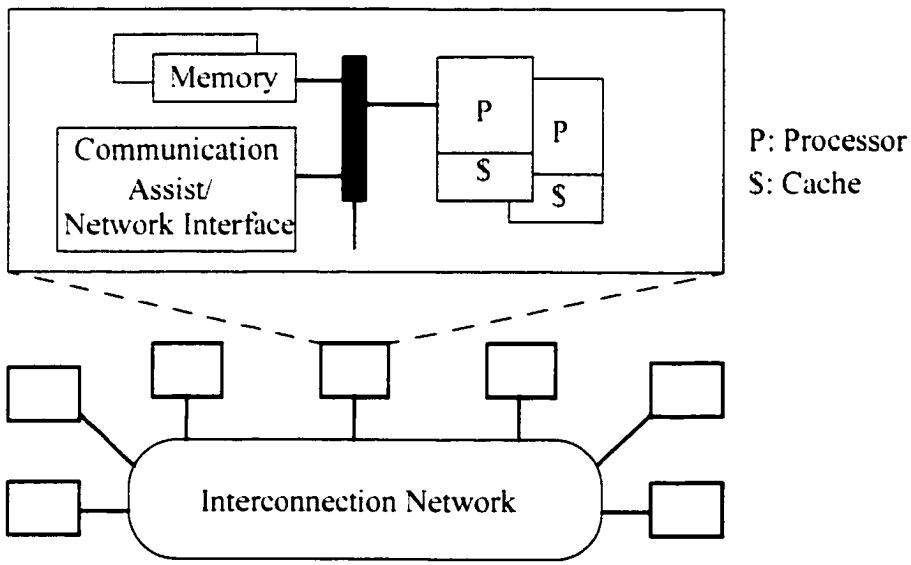


Figure 1.1: A generic parallel computer

Message-passing multicomputers, among all known parallel architectures, are the best to achieve such computing performance level. Message-passing multicomputers are characterized by the distribution of memory among a number of computing nodes that communicate with each other by exchanging messages through their interconnection networks. Each node has its own processor, local memory, and communication assist/network interface. All local memories are private and are accessible only by the local processors. The wide acceptance of message-passing multiprocessor systems has been proven by the introduction of *Message Passing Interface* (MPI) standard [92, 93]. Currently, in addition to vendor implementations of MPI on commercial machines, there are many freely available MPI implementations including MPICH [57] and LAM/MPI [78].

Recently, *Networks of Workstations* (NOW) [11], *Clusters of Workstations* (COW), and *Clusters of Multiprocessors* (CLUMP) [87], have been proposed to build inexpensive parallel computers, however, often at a lower performance level compared to MPP systems. The development of high-performance switches specially for building cost-effective interconnects known as *System Area Networks* (SAN) [23, 67, 113, 54] has motivated suitability of the networks of workstation/multiprocessors as an inexpensive high-performance computing platform. System area networks such as the Myricom Myrinet [23], the IBM Vulcan switch in the IBM SP2 machine [113], the Tandem ServerNet [67], and the Spider switch in SGI Origin 2000 machine [54], are a new generation of networks that falls between memory buses and commercial local area networks (LANs).

Parallel processing, whether MPP, DSM, NOW, COW, or CLUMP, puts tremendous pressure on the interconnection networks and the memory hierarchy subsystems. As the communication overhead is one of the most important factors affecting the performance of parallel computers [76, 69, 43], there has been a growing interest in the design of interconnection networks. In this respect, various types of interconnection networks, such as complete networks, hypercubes, meshes, rings, tori, irregular switch-based, stack-graphs, and hypermesh have been proposed and some of them have been implemented [46, 124, 108]. Meanwhile, many routing algorithms [47, 56, 12] have been proposed for such networks.

In parallel processing systems, the ability to efficiently communicate and share data between processors is very critical to obtaining high performance. In essence, parallel computers require extremely short communication latencies such that network transactions have minimal impact on the overall computation time. Communication hardware latency, communication software latency, and the user environment (multiprogramming, multiuser) are the major factors affecting the performance of parallel computer systems. This thesis concentrates on issues regarding hardware communication latency in electronic networks and reconfigurable optical networks, and software communication latency (regardless of the type of network).

In this thesis, I propose a number of techniques to achieve efficient communications in message-passing systems. This thesis makes five contributions:

- The first contribution of this thesis (Chapter 3) is the design and evaluation of two different categories of prediction techniques for message-passing systems. Specifically, I use these predictors to predict the target of communication messages in parallel applications.
- As the second contribution of this thesis (Chapter 4), I show that the majority of reconfiguration delays in reconfigurable networks can be hidden by using one of the high hit ratio proposed predictors in Chapter 3.
- The third contribution of this thesis (Chapter 5) is the analysis of a latency hiding broadcasting algorithm on single-hop reconfigurable networks under single-port and k-port modeling which brings up closed formulations that yield the termination time.
- As the fourth contribution of this thesis (Chapter 5), I propose a new total exchange algorithm in single-hop reconfigurable networks under single-port and k-port modeling.
- Finally, the fifth contribution (Chapter 6) is the use and evaluation of the proposed predictors in Chapter 3 to predict the next consumable message at the receiving ends of message-passing systems (regardless of the type of network). I argue that

these message predictors can be efficiently used to drain the network and cache the incoming messages even if the corresponding receive calls have not been posted yet.

Chapter 2 introduces the parallel applications used in this thesis. Chapter 7 concludes this dissertation and gives directions for future research. Appendix A describes how timing disturbances have been removed from the timing profiles of the parallel applications used in this thesis.

The rest of this chapter is organized as follows. In Section 1.1, I explain the communication locality in message-passing parallel applications and discuss different latency hiding techniques for parallel computer systems. In Section 1.2, I discuss the advantages of using prediction techniques at the send side of communications in the reconfigurable optical interconnection networks, and in the circuit switched and wormhole routing electronic interconnection networks. In Section 1.3, I describe the issues related to the messaging layer and software communication overhead in message-passing systems, and how prediction can help eliminate redundant message copying operations. I give an introduction to the issues regarding collective communications in Section 1.4. Finally, I summarize the contributions of this thesis in Section 1.5.

1.1 Communications Locality and Prediction Techniques

In this thesis, I am interested in the message-passing model of parallelism as message-passing parallel computers scale much better than the shared-memory parallel computers. Communication properties of message-passing parallel applications can be categorized by the *spatial*, *temporal*, and *volume* attributes of the communications [30, 75, 68]. The temporal attribute of communications in parallel applications characterizes the rate of message generation, and the rate of computations in the applications. The volume of communications is characterized by the number of messages, and the distribution of message sizes in the applications.

The Spatial attribute of communications in parallel applications is characterized by the distribution of message destinations. Point-to-point communication patterns may be repetitive in message-passing applications as most parallel algorithms consist of a number of computation and communication phases. Several researchers have worked to find or use the *communications locality* properties of parallel applications [30, 75, 68, 36, 37].

By *message destination communication locality*, I mean that if a certain source-destination pair has been used it will be re-used with high probability by a portion of code that is "near" the place that was used earlier, and that it will be re-used in the near future. By *message reception communication locality* I mean that if a certain message reception call has been used it will be re-used with high probability by a portion of code that is "near" the place that was used earlier, and that it will be re-used in the near future.

Traditionally, one approach to deal with communication latency is to *tolerate* the latency: that is, hide the latency from the processor's critical path by overlapping it with other high latency events, or hide it with computations. The processor is then free to do other useful tasks.

Three approaches can be used to tolerate latency in shared-memory and message-passing systems [32]. They are *proceeding past communication in the same thread*, *multithreading*, and *precommunication*. The first approach, proceeding past communication in the same thread in message-passing systems, is to make communication messages asynchronous and proceed past them either to other asynchronous communication messages, or to the computation in the same thread. This approach is usually used by the parallel algorithm designers. Some of the applications studied in this thesis use this type of latency tolerance by using nonblocking asynchronous MPI calls.

In multithreading, a thread issuing a communication operation suspends itself and lets another thread run. This approach is used for other threads too. It is hoped that when the first thread is rescheduled, its communication operations have concluded. Multithreading can be done in software or hardware. Software multithreading is very expensive. Some hardware multithreading research architectures for message-passing systems such as the J-Machine [35], and the M-Machine [52] have been reported.

In precommunication, communication operations are pulled up from the place that communications naturally occur in the program so that it is partially or entirely completed before data is needed. This can be done in software by inserting a *precommunication operation*, or in hardware, by *predicting* the subsequent communication operations and issue them early.

Precommunication is common in receiver-initiated communications (that is, in shared-memory systems) where communication commences when a data is needed such as a read operation. In *software-controlled prefetching*, the programmer or the compiler decides when and what to prefetch by analyzing the program and then inserting *prefetch* instructions before the actual data request in the program [95]. In *hardware-controlled prefetching*, dedicated hardware is used to predict the future accesses of sharing patterns and coherence activities by looking at their observed behavior [96, 77, 73, 133, 34, 107]. Thus, there is no need to add instructions to the program. These techniques assume that memory accesses and coherence activities in the near future will follow past patterns. Then, the hardware prefetches the data based on its prediction.

In sender-initiated systems (that is, in message-passing systems), it is usually difficult to do the communication operation earlier at the send sides and thus hide the latency. This is because message communication is naturally initiated to transfer the data when the data is produced. However, messages may arrive earlier at the receiver than it is needed which leads to a precommunication for the receiver side of communication.

As far as the author is aware, no precommunication technique has been proposed for message-passing systems. Predictions techniques can be used to predict the subsequent message destinations, and message reception calls in message-passing systems. This thesis, for the first time, proposes and evaluates two categories of pattern-based predictors, namely, *Cycle-based* predictors, and *Tag-based* predictors for message-passing systems. These predictors can be used dynamically (at the send side or receive side of communications) at the communication assist or network interface with or without the help of a programmer or the compiler.

1.2 Using the Proposed Predictors at the Send Side

In the following, I explain how message destination prediction can be helpful in hiding the reconfiguration delay in single-hop and multi-hop reconfigurable optical interconnection networks, and in hiding path setup time in circuit switched electronic networks. I also describe the benefit of message destination prediction techniques to reduce the latency of communications in current commercial wormhole routed networks.

The interconnection network plays a key role in the performance of message-passing parallel computers. A message is sent from a source to a destination through the interconnection network. High communication bandwidth and low communication latency are essential for efficient communication between a source and a destination. However, communication latency is the most important factor affecting the performance of message-passing parallel computers. In this thesis, I am interested in hiding and reducing the communication latency. Two categories of interconnection networks exist: electronic interconnetcion networks, and optical interconnection networks. I have developed prediction techniques that can be applied to both electronic and optical interconnection networks.

The proposed predictors can be used to set up the paths in advance in electronic networks using either circuit switching or *wave switching*. In circuit-switching, the routing header flit progresses through the message destination and reserves physical links. Wave switching is a hybrid switching technique for high performance routers in electronic interconnection networks. Wave switching combines wormhole switching and circuit switching in the same router architecture to reduce the fixed overhead of communication latency by exploiting communication locality. Hence, it is possible to hide the hardware communication latency using message destination predictions to pre-establish physical circuits in circuit switching and wave switching networks.

The predictors can even be useful to reduce communication latency in current commercial networks. For example, Myrinet networks [23] have a relatively long routing time compared with link transmission time. Predictors would allow sending the message header in advance for the predicted message destination. When data becomes available, they can

be directly transmitted through the network if the prediction was correct, thus reducing latency significantly. In case of mis-prediction, a message tail is forwarded to tear the path down. Obviously, null messages must be discarded at the destination.

Optics is ideally suited for implementing interconnection networks because of its superior characteristics over electronic interconnects such as higher bandwidth, greater number of fan-ins and fan-outs, higher interconnection densities, less signal crosstalk, freedom from planar constraint as it can easily exploit the third spatial dimension which dramatically increases the available communication bandwidth, lower signal and clock skew, lower power dissipation, inherent parallelism, immunity from electromagnetic interference and ground loops, and suitability for reconfigurable interconnects [100, 51, 74, 19, 50, 129, 82, 19].

Future massively parallel computers might benefit from using reconfigurable optical interconnection networks. Currently, there are some problems with the optical interconnect technology. Signal attenuation, optical element aligning, low conversion time between electronics to photonics and vice versa, and high reconfiguration delay are some disadvantages of optics which are mostly due to its relatively immature technology. However, this technology is maturing fast. As an example, *Lucent's WaveStar LambdaRouter* [86] relies on an array of hundreds of electrically configurable microscopic mirrors fabricated on a single substrate so that an individual wavelength can be passed to any of 256 input and output fibers.

As stated above, the reconfiguration delay in reconfigurable optical interconnection networks is currently very high. The proposed message destination predictors can be efficiently used to hide the reconfiguration delay in the single-hop and multi-hop reconfigurable optical interconnection networks concurrently to the computations [127, 84].

1.3 Redundant Message Copying in Software Messaging Layers

The communication software overhead currently dominates the communication time in cluster of workstations/multiprocessors. Crossing protection boundaries several times between the user space and the kernel space, passing several protocol layers, and involving a number of memory copying are three different sources of software communication cost.

Several researchers are working to minimize the cost of crossing protection boundaries, and using simple protocol layers by utilizing *user-level messaging* techniques such as *Active Messages* (AM) [125], *Fast Messages* (FM) [102], *Virtual Memory-Mapped Communications* (VMMC-2) [48], *U-Net* [126], *LAPI* [110], *Basic Interface for Parallelism* (BIP) [105], *Virtual Interface Architecture* (VIA) [49], and *PM* [121].

A significant portion of the software communication overhead belongs to a number of message copying operations. Ideally, message protocols should copy the message directly from the send buffer in its user space to the receive buffer in the destination without any intermediate buffering. However, applications at the send side do not know the final receive buffer addresses and, hence, the communication subsystems at the receiving end still copy messages at a temporary buffer.

Several research groups have tried to avoid memory copying [79, 14, 106, 119, 118]. They have been able to remove the extra memory copying operations between the application user buffer space and the network interface at the send side. However, they haven't been able to remove the memory copying at the receiver sides. They may achieve a zero-copy messaging at the receiver sides only when the receive call is already posted, a rendez-vous type communication is used for large messages, or the destination buffer address is already known by an extra communication (pre-communication). However, the predictors proposed in this dissertation can be efficiently used to predict the next message reception calls and thus move the corresponding incoming messages to a place near the CPU such as a staging cache.

1.4 Collective Communications

Communication operations may be either *point-to-point*, which involve a single source and a single destination, or *collective*, in which more than two processes participate. Collective communications are common basic patterns of interprocessor communication that are frequently used as building blocks in a variety of parallel algorithms. Proper implementation of these basic communication operations is a key to the performance of the par-

allel computers. Therefore, there has been a great deal of interest in their design and the study of their performance. Excellent surveys on collective communication algorithms can be found in [90, 53, 61].

Collective communication operations can be used for data movement, process control, or global operations. Data movement operations include, *broadcasting*, *multicasting*, *scattering*, *gathering*, *multinode broadcasting*, and *total exchange*. *Barrier synchronization*, is a type of process control. Global operations include *reduction*, and *scan*. The growing interest in collective communications is evident by their inclusion in the Message Passing Interface (MPI) [93, 92].

1.5 Thesis Contributions

In Chapter 2, I describe the applications used in this thesis along with the point-to-point communication primitives that they use. I explain the experimental methodology used to collect the communication traces of the applications.

In Chapter 3, I introduce a complete interconnection network using free-space reconfigurable optical interconnects for message-passing parallel machines. A computing node in this parallel machine configures its communication link(s) to reach to its destination node(s). Then it sends its message(s) over the established link(s).

I characterize some communication properties of the parallel applications by presenting their communication frequency and message destination distributions. I define the concept of communication locality in message-passing parallel applications, and caching in reconfigurable networks. I present evidence, using classical memory hierarchy heuristics, *LRU*, *LFU*, and *FIFO*, that there exists message destination communication locality in the message-passing parallel applications.

The first contribution of this thesis (Chapter 3) is the design and evaluation (in terms of hit-ratio) of two different categories of hardware/software communication latency hiding predictors for such reconfigurable message-passing environments. I have utilized the message destination locality property of message-passing parallel applications to devise a number of heuristics that can be used to *predict* the target of subsequent communication

calls. This technique, can be applied directly to reconfigurable interconnects to hide the communications latency by reconfiguring the communications network concurrently to the computation.

Specifically, I propose two sets of message destination predictors: *Cycle-based* predictors, which are purely dynamic predictors, and *Tag-based* predictors, which are static/dynamic predictors. In cycle-based predictors, *Single-cycle*, *Single-cycle2*, *Better-cycle*, and *Better-cycle2*, predictions are done dynamically at the network interface without any help from the programmer or compiler. In Tag-based predictors, *Tagging*, *Tag-cycle*, *Tag-cycle2*, *Tag-bettercycle*, and *Tag-bettercycle2*, predictions are done dynamically at the network interface as well, but they require an interface to pass some information from the program to the network interface. This can be done with the help of a programmer or the compiler through inserting instructions in the program such as *pre-connect (tag)* (or *pre-receive (tag)* as in Chapter 6). The performance of the proposed predictors, Better-cycle2 and Tag-bettercycle2, is very high and prove that they have the potential to hide the hardware communication latency in reconfigurable networks. The memory requirements of the predictors is very low. That makes them very attractive for the implementation on the communication assist or network interface.

In order to efficiently use the proposed predictors in Chapter 3 to hide the hardware latency of the reconfigurable interconnects, enough lead time should exist such that the reconfiguration of the interconnect be completed before the communication request arrives. In Chapter 4, I present the pure execution times of the computation phases of the parallel applications on the IBM Deep Blue machine at the IBM T. J. Watson Research Center using its high-performance switch and under the user space mode.

As the second contribution of this thesis, Chapter 4 states that by comparing the inter-send computation times of these parallel benchmarks with some specific reconfiguration times, most of the time, we are able to fully utilize these computation times for the concurrent reconfiguration of the interconnect when we know, in advance, the next target using one of the proposed high hit ratio target prediction algorithms introduced in Chapter 3. I present the performance enhancements of the proposed predictors on the application

benchmarks for the total reconfiguration time. Finally, I show that by applying the predictors at the send sides, applications at the receiver sides would also benefit as messages arrive earlier than before.

As the third contribution of this thesis (Chapter 5), I present and analyze a broadcasting algorithm that utilizes latency hiding and reconfiguration in the network to speed the broadcasting operation under single-port and k-port modeling. In this algorithm, the reconfiguration phase of some of the nodes is overlapped with the message transmission phase of the other nodes which ultimately reduces the broadcasting time. The analysis brings up closed formulation that yields the termination time of the algorithm.

The fourth contribution of this thesis (Chapter 5) is a *combined total exchange algorithm* based on a combination of the *direct* [109, 120], and *standard exchange* [71, 24] algorithms. This ensures a better termination time than that which can be achieved by either of the two algorithms. Also, known algorithms [20, 40] for scattering and all-to-all broadcasting have been adapted to the network.

In Chapter 6, I present the frequency and distributions of receive communication calls in the applications. I present evidence that there exists message reception communications locality in the message-passing parallel applications. As I stated earlier, the communication subsystems at the receiving end still copy early arriving messages unnecessarily at a temporary buffer. As far as the author is aware, no prediction techniques have been proposed to remove this unnecessary message copying.

I use the proposed predictors introduced in Chapter 3 to predict the next consumable message, and to thus establish the existence of message reception communications locality. As the fifth contribution of this thesis, Chapter 6 argues that these message predictors can be efficiently used to drain the network and cache the incoming messages even if the corresponding receive calls have not been posted yet. This way, there is no need to unnecessarily copy the early arriving messages into a temporary buffer.

The performance of the proposed predictors, Single-cycle, Tag-cycle2 and Tag-bettercycle2, in terms of hit ratio, on the parallel applications are quite promising and suggest that prediction has the potential to eliminate most of the remaining message copies.

Moreover, the memory requirements of these predictors is very low making them easy to implement. Finally, I discuss ways in which these predictions could be used to drastically reduce the latency due to message copying.

In Chapter 7, I conclude this thesis and give some directions for future research.

Chapter 2

Application Benchmarks and Experimental Methodology

In Section 2.1, I describe the applications used in this thesis. I explain the various point-to-point message-passing primitives of the applications in Section 2.2. I discuss the experimental methodology in Section 2.3.

2.1 Parallel Benchmarks

This thesis (except Chapter 5) studies the computation and communication characteristics of actual parallel applications. For these studies, I have used some well-known parallel benchmarks form the *NAS parallel benchmarks* (NPB) suite [13], the *Parallel Spectral Transform Shallow Water Model* (PSTSWM) parallel application [125], and the pure *Quantum Chromo Dynamics Monte Carlo Simulation Code* with *MPI* (QCDMPI) parallel application [65]. Although the results presented in this thesis are for the above parallel applications, these applications have been widely used as benchmarks representing the computations in scientific and engineering parallel applications.

I used the MPI [92] implementation of the NAS benchmarks, version 2.3, the PSTSWM, version 6.2, and the QCDMPI, version 1.4, and run them on several IBM SP2 machines. I chose the IBM SP2 as it is a message-passing parallel machine so that the chosen parallel applications are mapped directly on it. I used different system sizes and problem sizes of the applications in this study. NPB 2.3 comes with five problem sizes for each benchmark: small class "S", workstation class "W", large class "A" and larger classes "B" and "C". Due to access limitations in the use of the IBM Deep Blue machine at the IBM T. J. Watson Research Center, and space limitations in using the University of Victoria IBM SP2, I was able to experiment with only the "W" and "A" classes and the results included in this thesis represent theses classes.

2.1.1 NPB: NAS Parallel Benchmarks Suite

The NAS Parallel Benchmarks (NPB) [13] have been developed at the NASA Ames Research Center to study the performance of massively parallel processor systems and networks of workstations. The NAS Parallel Benchmarks are a set of eight benchmark problems, each of which focuses on some important aspect of highly parallel supercomputing for aerophysics applications. The NPB are a set of implementations of the NAS Parallel Benchmarks based on Fortran 77 and the MPI message-passing interface standard, and are not tied to any specific system.

The NPB consists of five "kernels", and three "simulated computational fluid dynamic (CFD) applications". The three simulated CFD application benchmarks, *lower-upper diagonal* (LU), *scalar pentadiagonal* (SP), and *block tridiagonal* (BT) are intended to accurately represent the principal computational and data movement requirements of modern CFD applications. The kernels, *conjugate gradient* (CG), *multigrid* (MG), *embarrassingly parallel* (EP), *3-D fast-Fourier transform* (FT), and *integer sort* (IS) are relatively compact problems, each of which emphasizes a particular type of numerical computation. I am interested in the point-to-point patterns of the LU, BT, and SP applications, and CG and MG kernels. EP, FT, and IS kernels are not suitable for this study. EP and FT use only collective communication operations while each node in the IS kernel always communicates with a specific node.

2.1.1.1 CG

The *conjugate gradient* kernel, CG, tests the performance of the system for unstructured grid computations which by their nature require irregular long distance communications which is a challenge for all kinds of parallel computers. Essentially, it requires computing a sparse matrix-vector product. The inverse power method is used to find an estimate of the largest eigenvalue of a symmetric positive-definite sparse matrix with a random pattern of non-zeros. This code requires a power-of-two number of processors.

2.1.1.2 MG

The second kernel benchmark is a simplified *multigrid kernel*, MG, which solves a 3-D poisson PDE. Four iterations of the V-cycle multigrid algorithm are used to obtain an approximate solution u to the discrete Poisson problem $\nabla^2 u = v$ on a $256 \times 256 \times 256$ grid with periodic boundary conditions. This code is a good test of both short and long distance highly structured communication. This code requires a power-of-two number of processors. The partitioning of the grid onto processors occurs such that the grid is successively halved, starting with the z dimension, then the y dimension and then the x dimension, and repeating until all power-of-two processors are assigned.

2.1.1.3 LU

The *lower-upper diagonal* benchmark, LU, employs a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse block 5×5 lower and upper triangular system. A 2-D partitioning of the grid onto processors occurs by halving the grid repeatedly in the first two dimensions, alternately x and then y, until all power-of-two processors are assigned, resulting in vertical pencil-like grid partitions on the individual processors. The ordering of point based operations constituting the SSOR procedure proceeds on diagonals which progressively sweep from one corner on a given z plane to the opposite corner of the same z plane, thereupon proceeding to the next z plane. Communication of partition boundary data occurs after completion of computation on all diagonals that contact an adjacent partition. LU is very sensitive to the small-message communication performance of an MPI implementation. It is the only benchmark in the NPB 2.3 suite that sends large numbers of very small (40 byte) messages.

2.1.1.4 BT and SP

The BT and SP algorithms have a similar structure: each solves three sets of uncoupled systems of equations, first in the x, then in the y, and finally in the z direction. In the *block tridiagonal* benchmark, BT, multiple independent systems of non-diagonally dominant, block tridiagonal equations with a 5×5 block size are solved. In the *scalar pentadiagonal* benchmark, SP, multiple independent systems of non-diagonally dominant, scalar pen-

tadiagonal equations with a 5×5 block size are solved. Both BT and SP codes require a square number of processors. These codes have been written so that if a given parallel platform only permits a power-of-two number of processors to be assigned to a job, then unneeded processors are deemed inactive and are ignored during computation, but are counted when determining Mflop/s rates.

2.1.2 PSTSWM

The *Parallel Spectral Transform Shallow Water Model* (PSTSWM) application [125], was developed by Worley at Oak Ridge National Laboratory and Foster at Argonne National Laboratory. 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. PSTSWM was developed to evaluate parallel algorithms for the spectral transform method as it is used in global atmospheric circulation models. 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. PSTSWM is written in Fortran 77 with VMS extensions and a small number of C preprocessor directives. I used the MPI implementation of the PSTSWM with the default input sizes.

2.1.3 QCDMPI

Pure Quantum Chromo Dynamics Monte Carlo Simulation Code with MPI (QCDMPI) [65], written by Hioki at Tezukayama University, is a pure Quantum Chromo Dynamics simulation code with MPI calls. It is a powerful tool to analyze the non-perturbative aspects of QCD. This program can be applied to any dimensional QCD such as the 3-dimensional QCD in which the color and/or quark confinement mechanism are obtained. QCDMPI runs on any number of processors and also any dimensional partitioning of the system can be applied.

2.2 Applications' Communication Primitives

As stated earlier, I am only interested in the patterns of the point-to-point communications between pair-wise nodes in the above applications as discussed in Chapter 3, Chapter 4, and Chapter 6 of this thesis. Efficient algorithms for collective communications are presented in Chapter 5. These applications use synchronous and asynchronous MPI send and receive primitives [92]. I briefly explain these communication primitives here.

An MPI program consists of autonomous processes, executing their own code, in an *multiple instructions multiple data* (MIMD) style. Note that all parallel applications studied in this thesis use an *single program multiple data* (SPMD) style. Processes are identified according to their relative rank in a group, that is, consecutive integers in the range 0 to *groupsize* - 1. If the group consists of all processes then the processes are ranked from 0 to N - 1 where N is the total number of processes in the application.

The processes communicate via calls to MPI communication primitives. The basic point-to-point communication operations are *send* and *receive*. There are two general point-to-point communication operations in MPI: *blocking* and *nonblocking*. Blocking send or receive calls will not return until the parameters of the calls can be safely modified. That is, in the case of a send call, the *message envelop* has been created and the message has been sent out or has been buffered into a system buffer. For the case of a receive call, it means that the message has been received into the receive buffer. Note that the message envelop consists of a fixed number of fields (*source, dest, tag, comm*) and it is used to distinguish messages and selectively receive them. Nonblocking communication operations just post or start the operation. Thus the application programmer must explicitly complete the communication call later at some point in the program using one of the various function calls in MPI such as *MPI_Wait* or *MPI_Waitall*.

There are four communication modes in MPI: *standard, buffered, synchronous*, and *ready*. These correspond to four different types of send operations. In the synchronous mode send call, the call will not finish until a matching receive call has been issued and has begun reception of the message. In the buffered mode send call, the send call is local (in contrary to other communication modes where the send calls are nonlocal) and is not

waiting for the receive call to be posted. Actually, it buffers data when the receive call is not posted. In the ready mode send call, the receive call must have been posted earlier. In the standard mode, it is up to the system to buffer the data or send it as in synchronous mode. Note that the standard mode is the only mode for the receive calls.

2.2.1 MPI_Send

MPI_Send (buf, count, datatype, dest, tag, comm) [92] is a standard blocking send call which is a combination of buffered and synchronous mode and is dependent on the implementation. When the call finishes, the send buffer can be used. In the buffered mode, data is written from the send buffer to the system buffer and the call returns. In the synchronous mode, the call waits for the receiver to be posted and then returns. The LU, MG, CG, and PSTSWM applications use this type of send call.

2.2.2 MPI_Isend

MPI_Isend (buf, count, datatype, dest, tag, comm, request) [92] is a standard non-blocking send call. It returns immediately. Therefore, the send buffer cannot be reused. It can be implemented in the buffered or synchronous mode. It needs another call, *MPI_Wait* or *MPI_Waitall*, to complete the call. These completion calls are explained later in Section 2.2.6 and Section 2.2.7, respectively. BT and SP use this type of send call.

2.2.3 MPI_Sendrecv_replace

MPI_Sendrecv_replace (buf, count, datatype, dest, sendtag, source, recvtag, comm, status) [92] combines in one call the sending of a message and receiving another message in the same buffer. QCDMPI uses this type of communication call.

2.2.4 MPI_Recv

MPI_Recv (buf, count, datatype, source, tag, comm, status) [92] is a standard blocking receive call. When it returns, the data is available at the destination buffer. LU and PSTSWM use this type of receive call.

2.2.5 MPI_Irecv

MPI_Irecv (buf, count, datatype, source, tag, comm, request) [92] is a standard non-blocking receive call. It immediately posts the call and returns. Hence, data is not available at the time of return. It needs another completion call such as *MPI_Wait* or *MPI_Waitall* to complete this call. All applications except QCDMPI use this type of receive call.

2.2.6 MPI_Wait

A call to *MPI_Wait (request, status)* [92] returns when the operation identified by *request* is complete. For *MPI_Isend* operation, when *MPI_Wait* returns the send buffer can be reused. For *MPI_Recv* operation, the completion of the *MPI_Wait* call notifies the availability of the data at the receive buffer. BT, LU, MG, CG, PSTSWM applications all use this type of completion call.

2.2.7 MPI_Waitall

MPI_Waitall (count, array_of_requests, array_of_statuses) [92] waits for the completion of all nonblocking calls associated with the active handles in the list. BT and SP use this type of completion call.

2.3 Experimental Methodology

I executed the applications on the 12-node IBM SP2 machine at the University of Victoria for gathering their communication traces, and on the 30-node IBM Deep Blue at the IBM T. J. Watson Research Center for collecting their timing profiles. I wrote my own profiling codes using the wrapper facility of the MPI to gather the communication traces, and the timing profiles of these applications. I did this by inserting monitor operations in the profiling MPI library for the communication related activities. These operations include arithmetic operations for the calculation of the desired characteristics. It is worth mentioning that gathering communication traces does not affect the communication patterns of these applications. However, it affects the temporal properties of these applications. In Appendix A, I explain the approach used to remove the timing disturbances from the timing profiles of the applications.

Chapter 3

Design and Evaluation of Latency Hiding/Reduction Message Destination Predictors

Interconnection networks and their services such as message delivery and flow control are a major source of communication hardware latency in parallel computer systems. In Section 3.1, I briefly describe message-passing computers and message switching layers. Then, as a specific circuit switched interconnection network, I introduce a *reconfigurable optical network*, *RON* (k, N), for message-passing parallel computers. The advantages of such reconfigurable optical interconnects are their high bandwidth and their ability to provide versatile application-dependent network reconfigurations.

I characterize some communication properties of the parallel application benchmarks by presenting their communication frequency and message destination distributions in Section 3.2. I define the concept of *communication locality* in message-passing parallel applications, and *caching* in reconfigurable networks in Section 3.3. I present evidence that there exists message destination communication locality in the message-passing parallel applications in Section 3.3.1. Using classical replacement heuristics, *LRU*, *LFU*, and *FIFO*, I show that message destinations display a form of locality.

I have utilized the message destination locality property of message-passing parallel applications to devise a number of heuristics that can be used to *predict* the target of subsequent communication requests. Thus, in Section 3.4, I contribute by proposing and evaluating (in terms of hit ratio) two different categories of hardware/software *communication latency hiding predictors* for message-passing environments. By utilizing such predictors, the hardware communication latency in reconfigurable interconnects can be effectively hidden by reconfiguring the communication network concurrent to the computation. I

compare the performance and storage requirements of the proposed predictors in Section 3.5. In Section 3.6, I elaborate on how these predictors can be used and integrated into the network interfaces. Finally, I summarize this chapter in Section 3.7.

3.1 Introduction

Message-passing multicomputers are composed of a number of computing modules that communicate with each other by exchanging messages through their interconnection networks. Each computing module has its own processors, local memory, and communication assist/network interface. All local memories are private and are accessible only by the local processors. Communication hardware latency, communication software latency, and the user environment (multiprogramming, multiuser) are the major factors affecting the performance of message-passing parallel computer systems.

Interconnection networks, and their services such as message delivery and flow control are a major source of communication hardware latency. Essentially, an interconnection network is characterized by its *topology*, *switching strategy*, *flow control mechanism*, and *routing algorithm*. The topology is the physical structure of the network. The interconnection network [46] might be a shared-medium network (such as Ethernet, Token Ring), a direct network (such as mesh, torus), an indirect network (multistage interconnection network such as IBM SP [112], or irregular such as Myrinet [23]), or a hybrid network (such as hypermesh) [117].

The routing algorithm determines which routes messages should follow through the network to reach their destinations. There are many different routing algorithms with different guarantees and performance such as Duato's adaptive routing [47], Glass and NI's turn-model routing [56], and up*-down* routing [12].

The flow control mechanism determines when the message, or packet, or portion of a message should move along its route. Packets or flits may be blocked, buffered, discarded or detoured to an alternate route based on the flow control mechanism.

3.1.1 Message Switching Layers

The switching strategy determines how a message moves along its routes. There are many switching strategies. *Circuit switching, packet switching, virtual cut-through,* and *wormhole switching* are the basic switching strategies [46]. In packet switching, messages are divided into fixed-size packets. Each packet is routed individually from source to destination and has to be buffered in each intermediate node. It is also called *store-and-forward switching*. In virtual cut-through switching, the entire packet does not need to be buffered in the nodes. The packet header can be examined and after the routing decision is made and the output channel is free the header and the following data can be immediately transmitted. In wormhole switching, the packet is broken up into flits. Wormhole switching pipelines the flits through the network just like the virtual cut-through switching strategy but it has reduced buffer requirements.

In circuit switching, a physical path is reserved from a source to a destination before the actual message transmission takes place. The routing header is injected into the network. It reserves physical links as it is transmitted through intermediate nodes. A complete path is set up when the routing header reaches the destination. Then an acknowledgment is transmitted back to the source. Then, the message contents can be sent along the reserved channels. The disadvantage is that during message transmission other messages may be blocked. The advantage is the minimum message transfer latency as the physical path is already established.

In Chapter 3 through Chapter 5 of this thesis, I am interested in the circuit switching strategy. As I explain later in Section 3.3, message destinations in message-passing parallel applications display a form of locality. Thus, it is possible to use this communication locality to pre-establish the physical links and thus hide the path setup time. This applies both to the electronic circuit switched interconnection networks, and to the reconfigurable optical interconnection networks. However, as I describe in Section 3.4, the prediction techniques that I propose in this chapter would also reduce the communication time in wormhole routed networks. In the next section, I consider a circuit switched reconfigurable optical interconnection network as an specific case.

3.1.2 Reconfigurable Optical Networks

Several topological properties, such as *degree*, *average distance*, and *diameter*, can be used to evaluate and compare different interconnection networks. Most of these properties can be derived from the underlying graph of an interconnection network, where processors and communication links are mapped onto the vertices (nodes) and edges (links) of the graph, respectively.

A *Graph* consists of a set of vertices, V, interconnected by a set of edges, E, symbolized as $G = (V,E)$ [122]. The number of vertices and edges in a graph is $N = |V|$, and $|E|$ respectively. An edge $e \in E$ connects vertices u and v, written as $e = uv$, and is said to be *incident* with u and v. A vertex v has *degree* d_v if it is incident with exactly d_v edges. In a *regular* graph G, all vertices have the same degree, equal to d_G. A *path* from v_1 to v_k is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that for every $1 \leq i < k$, the edge $v_i v_{i+1}$ is in E. The *distance* between u and v, $dist(u,v)$, is the minimum length of a path between u and v. The *eccentricity* of u is $e(u) = dist(u,v)$, where v is a vertex such that $dist(u,v) = MAX_{w \in V} dist(u,w)$. The maximum eccentricity among all vertices is the *diameter* of the graph.

I am interested in having a complete interconnection network, where any computing node can communicate with any other node in a single-hop. Complete interconnection networks can be modeled by a complete graph, K_N. A complete graph is a regular graph where all N vertices are linked together and the diameter is one. Each vertex has degree d_G equal to $N - 1$, and the number of edges, $|E|$, is $N(N-1)/2$ far too high to be of practical interest when N is large. These limitations prevent implementing complete networks using metal-based interconnections as there is a fixed physical link between any two nodes.

Optics is ideally suited for implementing interconnection networks because of its superior characteristics over electronics [100, 51, 74], such as higher interconnection density, higher bandwidth, suitability for reconfigurable interconnects, greater fan-in and fan-out, lower error rate, freedom from planar constraints (light beams can easily cross each other), immunity from electromagnetic field and ground loops, lower signal crosstalk,

Several research groups in academia and industry are working on different aspects of utilizing optical interconnects in massively parallel processing systems including works on the feasibility study and technology related problems of optical interconnects, architectures for optically interconnected computer systems, and communications and algorithmic issues for such parallel systems [82, 19].

One of the main features of an optical interconnect is its capability to *reconfigure*. This is very suitable for the construction of 3-D VLSI computers [89]. By *interconnect reconfiguration*, I simply mean the ability to change the interconnect dynamically upon demand. In essence, the advantages of reconfigurable optical interconnects are due to their ability to provide versatile application-dependent network configurations. *Free-space optical interconnects* are a class of optical interconnects that can support network reconfiguration.

Free-space optical interconnects use free-space (vacuum, air or glass) for optical signal propagation. In free-space optical interconnects, optical signals can propagate very close to each other and pass each other without interaction. It can easily exploit the third spatial dimension which dramatically increases the available communication bandwidth. Free-space reconfigurable optical interconnects result in much denser interconnection networks than metal-based and guided-wave interconnections [28, 83], and have the potential to solve the problems associated with implementing complete networks due to their ability to reconfigure.

I introduce an abstract model [1] for a complete interconnection network using free-space reconfigurable optical interconnects for massively parallel computers, and discuss its characteristics.

Definition A *reconfigurable optical network*, *RON* (k, N), consists of N computing nodes with their own local memory. A node is capable of connecting directly to any other node. A node can establish k simultaneous connections. These connections are established dynamically by reconfiguring the optical interconnect. The links remain established until they are explicitly destroyed.

Messages are sent using *circuit switching*. That is, a connection must be established between the source and destination pair before the message is sent. Each node has the ability to simultaneously send and receive k messages on its k links (the *k-port* model), or exactly one message on one of its links (the *single-port* model). Full-duplex communication where a node can send and receive messages at the same time is supported. A simplified block diagram of the network is shown in Figure 3.1 where each node uses only one of its links.

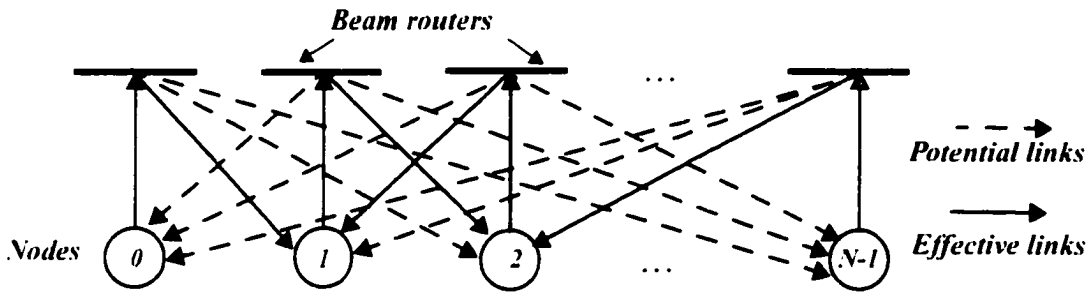


Figure 3.1: *RON (k, N)*, a massively parallel computer interconnected by a complete free-space optical interconnection network

Various implementation technologies exist to embody the above abstract model. Such technologies include *vertical-cavity surface-emitting lasers* (VCSELs) for photon generation, *self-electro-optic effect devices* (SEEDs) for modulation, frequency hoping for coding, wavelength tuning for transmitters and receivers, *computer generated holograms* (CGH), and *deformable mirrors* (DM) for switching and optical beam routing. The switching in the case of CGH can be achieved by recording the desired source-destination communication patterns. As stated in Chapter 1, deformable mirrors, such as *Lucent's WaveStar LambdaRouter* [86], are also reaching maturity. Optical beam routing in a free-space optical interconnection network often employs other external optical elements such as mirrors, prisms, lenses.

Each node has a fixed number of tunable transmitters for sending optical beams toward its beam router, such as a computer generated hologram or a deformable mirror, to be redirected to the receivers of the other nodes. Also, each node has a large number of fixed receivers at its input ports. Some of these input ports may be used only for collective communications operations while others may be used for pair-wise communications.

Path setup phase can be done by sending an encoded light beam to the beam router to reprogram the computer generated hologram, or to deform the mirror such that the actual message can be delivered to the destination(s) directly. It can be done in two different ways. First, the router (CGH or DM) upon receiving the message (which includes the payload) stores the message in a buffer and then configures its output links so that it can forward the message to the destination node(s). This approach needs a buffer for the entire message at each beam router which is of high cost. It also involves an extra copy. The better approach is to send an optical beam having only the destination address to the beam router for the path setup phase. Then, after some time, to be called *reconfiguration delay*, the second beam containing the actual message can be sent through the configured router to its destination.

Collision can happen at the receiving nodes considering the fact that several beams may arrive at a destination node at the same time. Hence, a destination node may not be able to complete the path setup phase, or accept the message. However, I assume that due to the availability of a large number of fixed receivers at the destinations, connections are established immediately after some time (reconfiguration delay).

I assume an unbounded number of available wavelengths for the system. However, in case of a limited number of available wavelengths, one can utilize spread-spectrum techniques where each transmitter sends its information changing the wavelength in a pseudo-random fashion. The receiver can reconstruct the transmitted message if it is aware of the pseudo-random code used for encoding the sequence of wavelengths used during the transmission.

I am not interested in the technology itself, and implementation concerns are outside the scope of this dissertation. Instead, I am particularly interested in the abstract model of this network. I shall assume that one or more of the technologies outlined above will be used to implement the proposed interconnect. Under such an implementation, the various overheads associated with the reconfiguration of the network (such as beam steering, setting up the computer-generated holograms, tuning the transmitters, or sending the frequency code in a frequency hoping implementation etc.) are lumped together as the reconfiguration delay d. I assume that the reconfiguration delay, d, most of the time is constant but occasionally may be unbounded due to hot spots in applications.

3.1.2.1 Communication Modeling

An important concern is to model the communication time T required to send a message from one node to another. I use the communication modeling of Hockney [66]. Hockney's model characterizes the communication time for a point-to-point communication operation as: $T = t_s + \frac{l_m}{r_\infty}$, where t_s is the start-up time which is equal to the time needed to send a zero byte message, and includes the time required to prepare the message, such as adding a header, and a trailer. l_m is the length of message to be transmitted, and r_∞ is the *asymptotic bandwidth* in Mbytes per second and is the maximum bandwidth achievable when the message length approaches infinity. The communication time can be written as: $T = t_s + l_m \tau$ where τ is the per unit transmission time and is equal to the reciprocal of r_∞. For the *RON* (k, N), I amend the model by explicitly including the reconfiguration delay d that is necessary for a node to configure a link that would connect directly to its target node(s). The transmission time then becomes $T = d + t_s + l_m \tau$.

The time on the fly, $l_m \tau$, for small messages is negligible compared to the setup time, t_s, and the reconfiguration delay, d. In the current generation of parallel computer systems, the setup time, t_s, is several tens of microseconds [43]. Several researchers are working to minimize the setup time by using user-level messaging techniques such as *Active Messages* (AM) [125] and *Fast Messages* (FM) [102]. In Chapter 6, I discuss issues regarding

the software overhead component of the communication latency. I utilize the prediction techniques proposed in this chapter to reduce the communication latency by avoiding unnecessary memory copying operations at the receiver side of communications.

In this chapter, I am particularly interested in the techniques that hide the reconfiguration delay, d. For this, and for the first time as far the author is aware, I propose and evaluate different communication latency hiding predictors at the send side of communications in message-passing systems using reconfigurable networks so that the reconfiguration delay can be hidden. In essence, by utilizing such predictors, the hardware communication latency in reconfigurable interconnects can be effectively hidden by reconfiguring the communication networks concurrent to the computations.

3.2 Communication Frequency and Message Destination Distribution

Several researchers have investigated the communication behavior of parallel applications [30, 75, 68, 72, 37]. Chodnekar and his colleagues [30] have developed a traffic characterization methodology for parallel applications. They have considered the inter-arrival time distribution of messages (send calls), spatial message distribution, and the message volume in message-passing and shared-memory applications. Kim and Lilja [75] examined the communication patterns of message-passing parallel scientific programs in terms of message size, message destination, and generation distributions for the send time, receive time, and computation time. Hsu and Banerjee [68] analyzed the communication characteristics of parallel CAD applications on a hypercube. Karlsson and Brorsson [72] have compared the communication properties of parallel applications in message-passing applications using MPI, and shared memory applications using TreadMarks [10]. de Lahaut and Germain [37] have shown that in scientific applications written in High Performance Fortran (HPF) [85] a large part of communications can be known from the analysis of the code. This is called *static communications*, communications that can be known at compile-time, in contrast to *dynamic communications* where communications can be determined only at run-time.

Essentially, communication properties of parallel applications can be categorized by the *spatial*, *temporal*, and *volume* attributes of the communications [30, 75, 68]. The temporal attribute of communications in parallel applications characterizes the rate of message generations, and the rate of computations. I present the cumulative distribution function of the inter-send computation times of the applications studied in this thesis in Chapter 4.

The volume of communications is characterized by the number of messages, and the distribution of message sizes in the applications. In this chapter, I am particularly interested in the number of messages . In Chapter 4, I show the distribution of message sizes in the parallel applications.

One of the communication volume characteristics of parallel applications is the frequency of send messages. I use a number of parallel benchmarks, as introduced in Chapter 2, and extract their communication traces. The processes in these applications use blocking and nonblocking standard MPI send primitives, namely *MPI_Send*, *MPI_Isend*, and *MPI_Sendrecv_replace* [92]. Figure 3.2 illustrates the number of send communication calls per process in the applications under different system sizes. I executed all applications once for each different system size and counted the number of send calls for each process of the applications. Hence, in Figure 3.2, by average, minimum, and maximum, I mean the average, minimum, and maximum number of send calls taken over all processes of each application. It is evident that processes in the BT, SP, CG, and QCDMPI applications have the same number of send communication calls for each different system size. This is also true for LU, MG, and PSTSWM when the number of processes is four, four and eight, and a power of two, respectively.

The Spatial attribute of communications in parallel applications is characterized by the distribution of message destinations. It is commonly assumed that the message destinations are evenly distributed among all of the processes although an individual process may not see a uniform message destination distribution [75, 30].

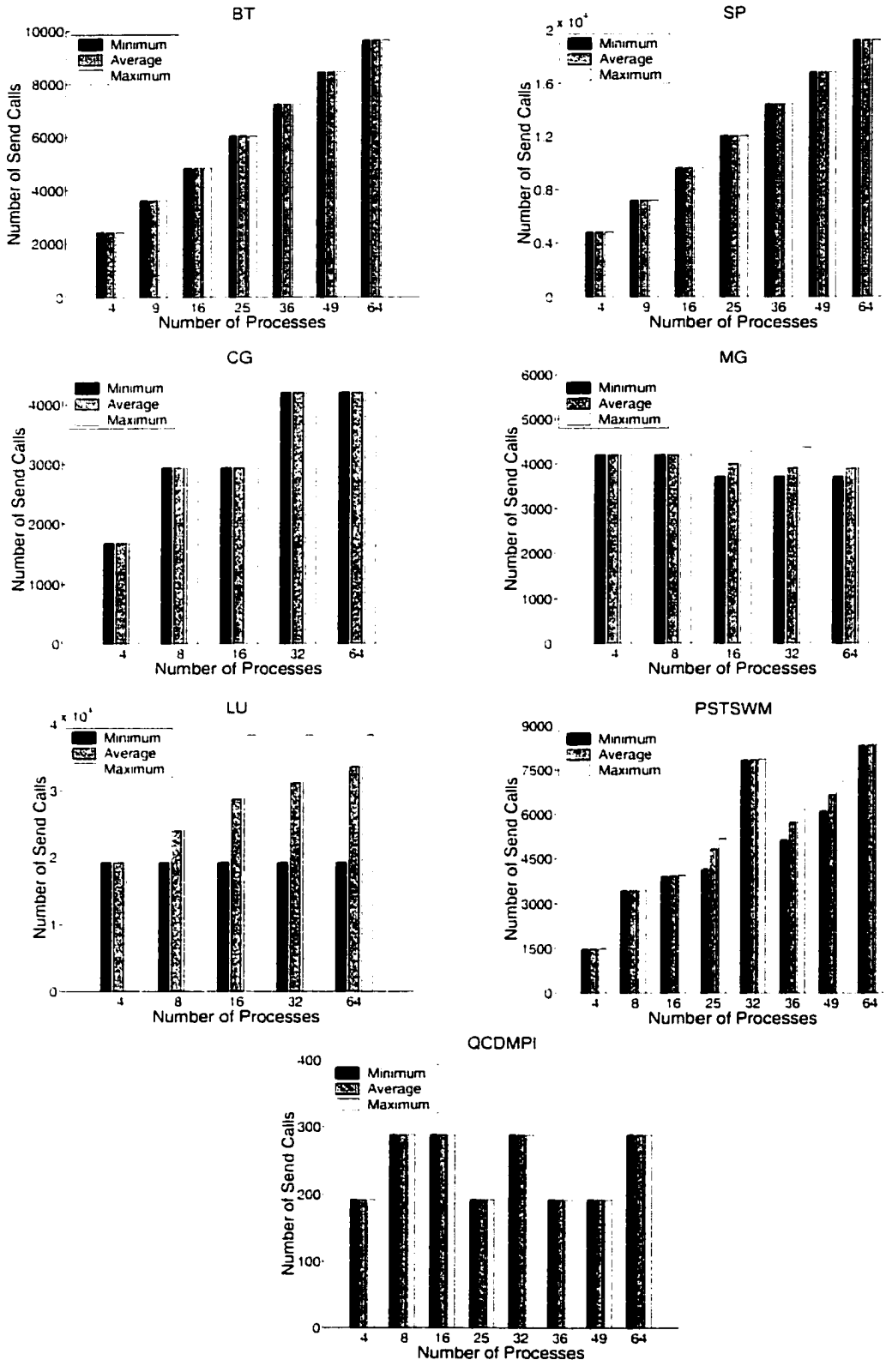


Figure 3.2: Number of send calls per process in the applications under different system sizes

In MPI, the send operation (*MPI_Send*, *MPI_Isend*, and *MPI_Sendrecv_replace* communication calls in the parallel applications studied in this thesis), associates an *envelope* with a message. Messages in addition to the data part carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which is collectively called the *message envelope*. These fields are the source process of a message, *source*, the destination process of a message, *dest*, the message tag, *tag*, and the message communicator, *comm*. The message source is implicitly determined by the identity of the message sender and need not be explicitly carried by messages. The other fields are specified by arguments in the send operation. The destination process is specified by the *dest* argument. The integer-valued message tag is specified by the *tag* argument. This integer can be used by the program to distinguish different types of messages. A communicator specifies the communication context for a communication operation. It also specifies the set of processes that share this communication context. Each communication context provides a separate communication universe. Messages are always received within the context they were sent, and messages sent in different contexts do not interfere. The BT, SP, and PSTSWM applications use a number of different communicators including the predefined communicator, *MPI_COMM_WORLD*, provided by MPI while other parallel applications, CG, MG, LU, and QCDMPI use only the predefined communicator.

As stated above, a message envelop consists of *source*, *dest*, *tag*, and *comm*. The *source* and *tag* of a message envelop do not affect the link establishment phase for a message transmission to a destination process. Thus, I assigned a different identifier, called *unique message destination identifier*, for each <*dest*, *comm*> tuple found in the communication traces of the applications. For simpilicity, from now on, I use the term "message destination" instead of unique message destination identifier. Figure 3.3, shows the minimum, average, and maximum number of message destinations per process in the applications under different system sizes. It is evident that processes in all applications communicate with only a favorite subset of all other processes. Note that processes in the BT, and SP applications, in contrast to the other applications, have the same number of message destinations under different system sizes (except when N is four). This is also

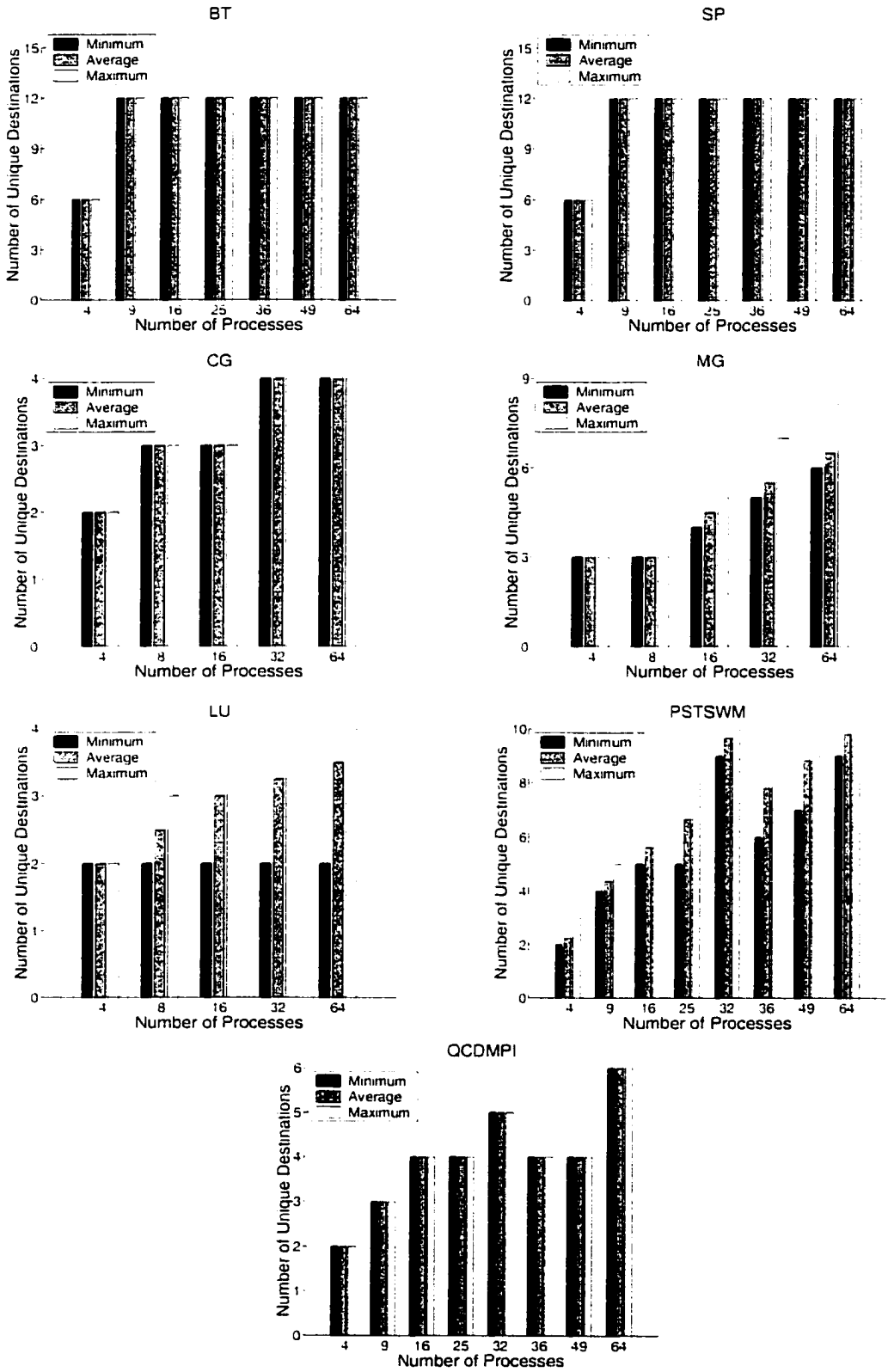


Figure 3.3: Number of message destinations per process in the applications under different system sizes

true for CG when the number of processes is 8 and 16, and for MG when it is 4 and 8. Meanwhile, in all applications, except BT and SP, the number of message destinations increases when the number of processes increases (note the exception cases in PSTSWM and QCDMPI when the number of processes increases from 32 to 36).

Figure 3.4, illustrates the distribution of message destinations in the applications when the number of processes is 64. The BT, SP, CG, PSTSWM, and QCDMPI applications verify the assumption that the message destinations are uniformly distributed among all of the processes. MG shows an almost uniform message destination. However, LU presents three different peaks for message destinations.

Figure 3.5, shows the distribution of message destinations for one of the processes, process zero, of the applications when the number of processes is 64. I choose process zero because it is a favorite destination of all processes and is usually responsible for distributing data and verifying the results of the computation. It is clear that this process tends to communicate with only a favorite subset of all other processes in the applications. I have found similar results for all other processes in each application as it can be seen in Figure 3.4.

3.3 Communication Locality and Caching

I define the terms *message destination communication locality*, and *caching* in conjunction with this work as follows. By message destination communication locality I mean that if a certain source-destination pair has been used it will be re-used with high probability by a portion of code that is "near" the place that was used earlier, and that it will be re-used in the near future. If communication locality exists in parallel applications, then it is possible to *cache* the configuration that a previous communication request has made and reuse it at a later stage. Caching in the context of this discussion will mean that when a communication channel is established it will remain established until it is explicitly destroyed. As already mentioned, in the context of free-space optical interconnect maintaining an established communication channel does not interfere with communications that are in progress in other parts of the network.

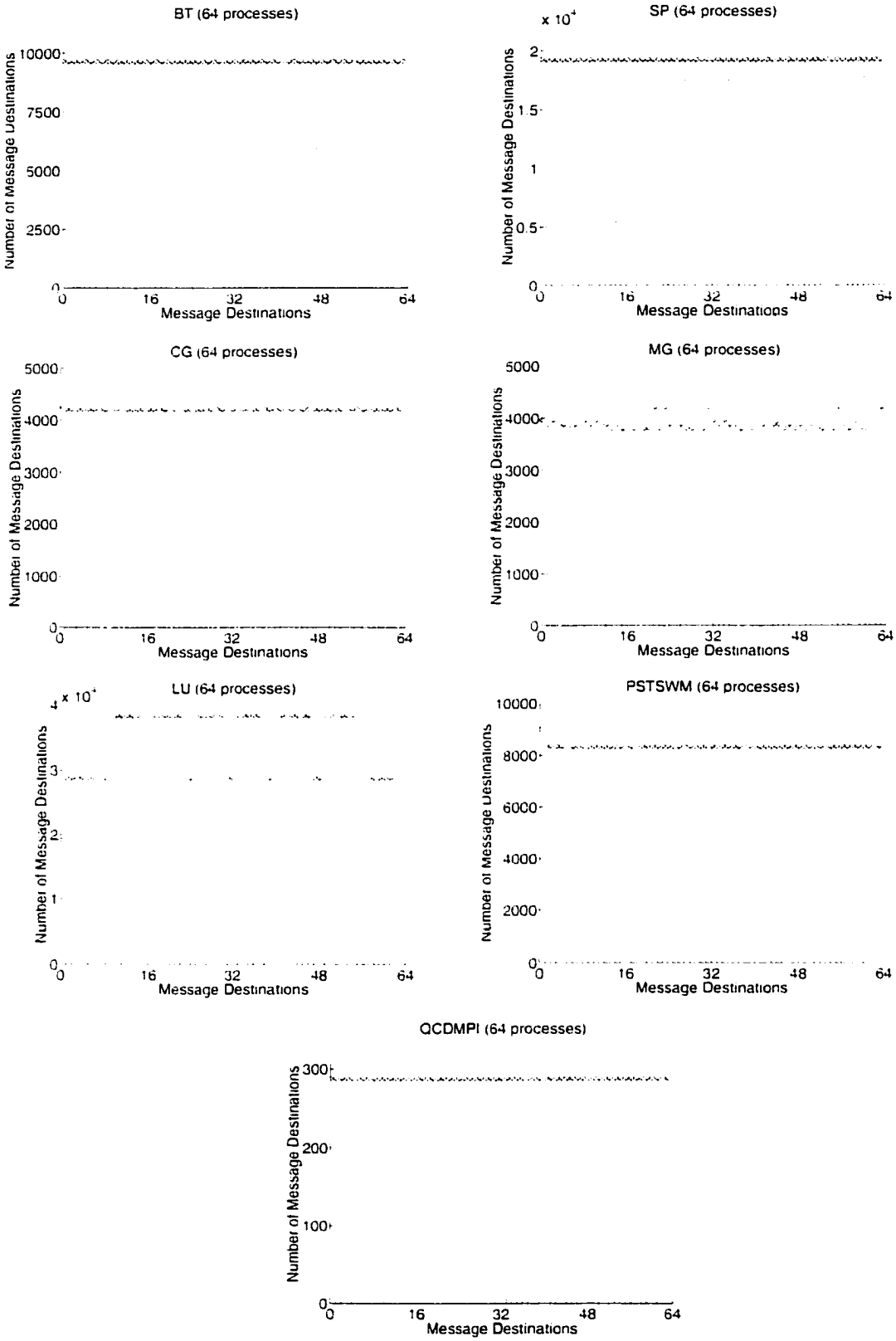


Figure 3.4: Distribution of message destinations in the applications when $N = 64$

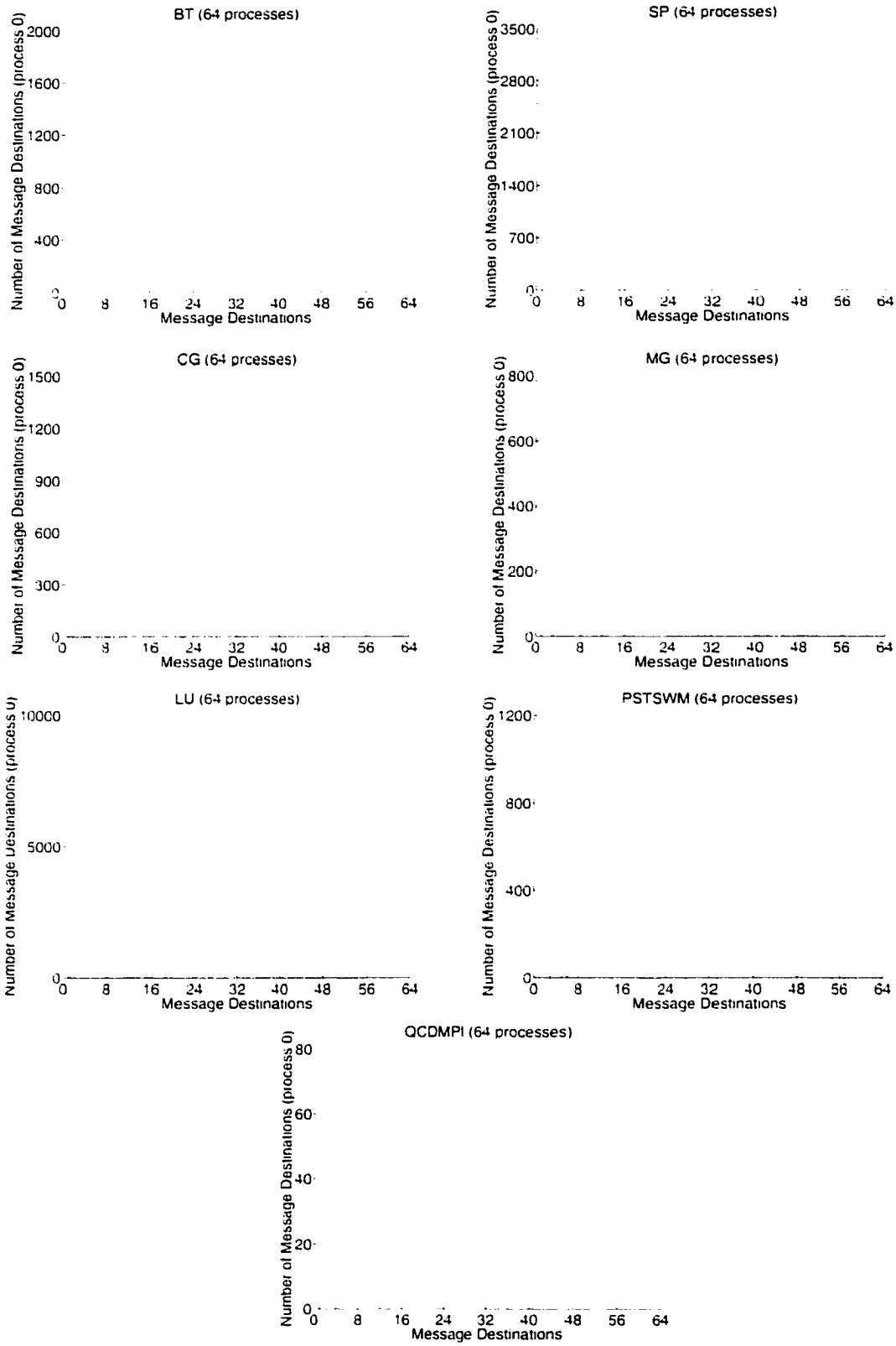


Figure 3.5: Distribution of message destinations in the applications for process zero, when $N = 64$

In the message-passing programming paradigm, many parallel algorithms are built from loops consisting of computation and communication phases. Therefore, communication patterns may be repetitive. This has motivated researchers to find the *communication locality* properties of parallel applications [75, 68]. Kim and Lilja [75] have recently shown that there is a locality in message destination, message sizes, and consecutive runs of send and receive primitives in parallel algorithms. They have proposed and expanded the concept of memory access locality based on the *Least Recently Used* (LRU) [68] stack model to determine these localities.

In the following subsection, I expand on the work by Kim and Lilja [75] by utilizing the FIFO and LFU heuristics on the applications to see the existence of message destination communication locality or repetitive message destinations. I use the term *hit ratio* to establish and compare the performance of these heuristics. If the next message destination is already in the set of message destinations maintained by the LRU, LFU, and FIFO heuristics, I count a *hit*, otherwise, I count a *miss*. It is clear that the hit ratio is equal to the number of hits divided by the total number of hits and misses.

3.3.1 The LRU, FIFO and LFU Heuristics

The *Least Recently Used* (LRU), *First-In-First-Out* (FIFO), and *Least Frequently Used* (LFU) heuristics, all maintain a set of k (k is the *window size*) message destinations. If the next message destination is not in the set, then it replaces one of the destinations in the set according to which of the LRU, FIFO or LFU strategies is adopted. The window size, k, corresponds to the number of input, output ports used in *RON* (k, N). Figure 3.6 shows the results of the LRU, FIFO, and LFU heuristics on the applications when the number of processes is 64. Figure 3.7, Figure 3.8 and Figure 3.9 illustrate the size scalibiltiy of the these heuristics on the applications. It is clear that the hit ratios in all applications approach 1 as the window size increases. The performance of the FIFO algorithm is almost the same as the LRU for all benchmarks. However, the LFU algorithm has a better performance than the LRU and FIFO heuristics, the exception is for the LU benchmark, when $k = 2$ and $N = 16$, 32, and 64.

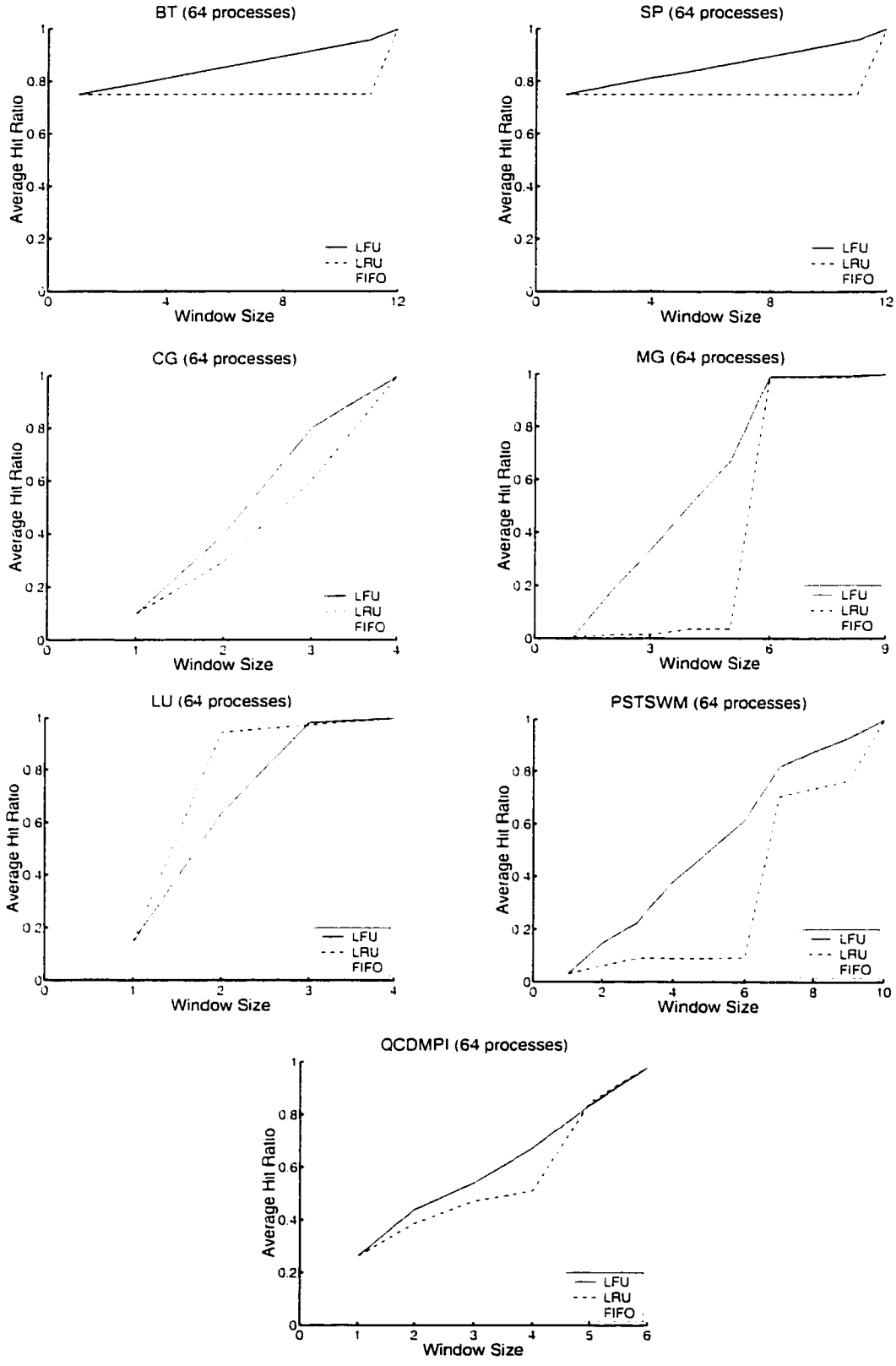


Figure 3.6: Comparison of the LRU, FIFO, and LFU heuristics when $N = 64$

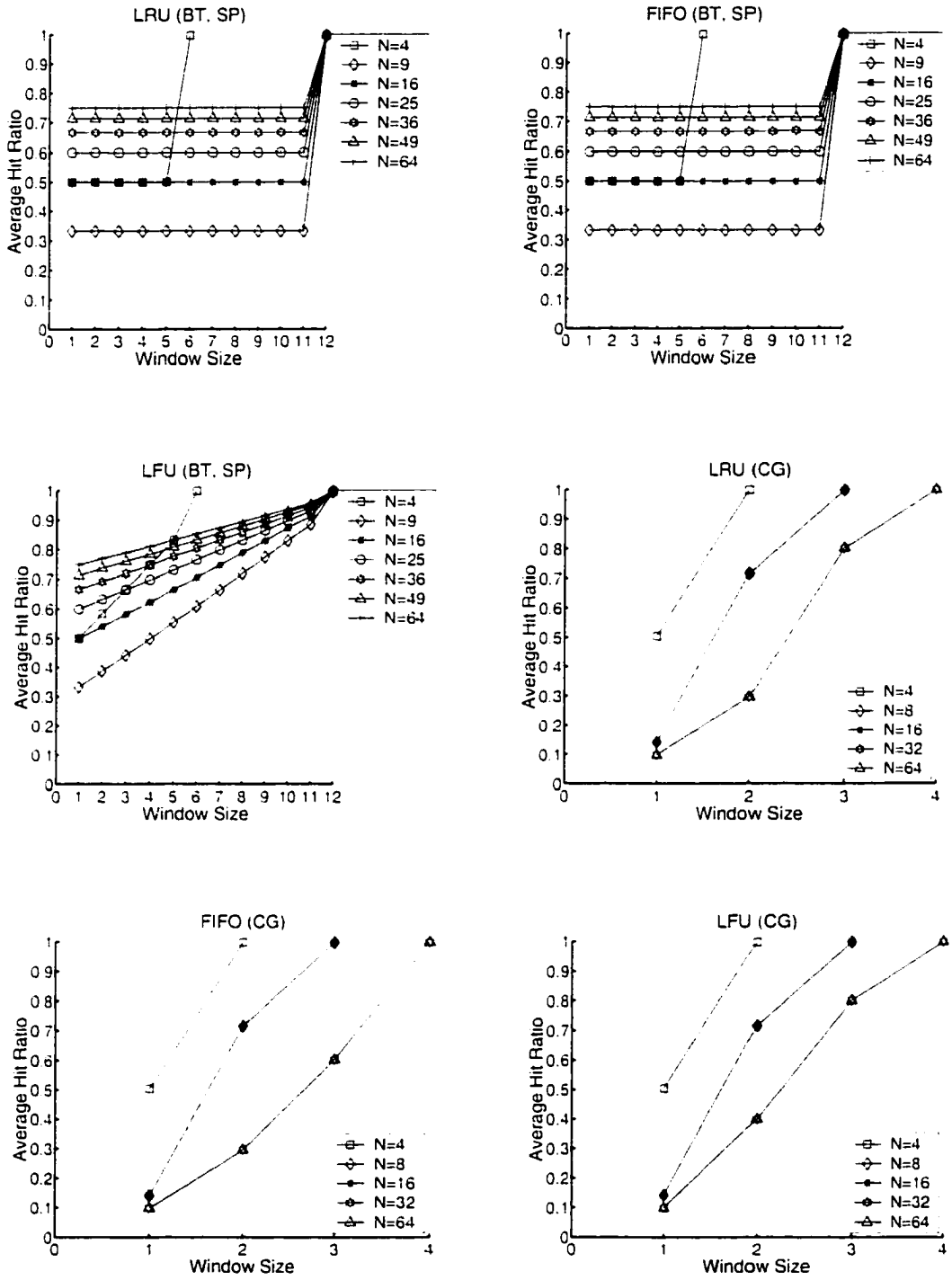


Figure 3.7: Effects of the scalibilty of the LRU, FIFO, and LFU heuristics on the BT, SP and CG applications

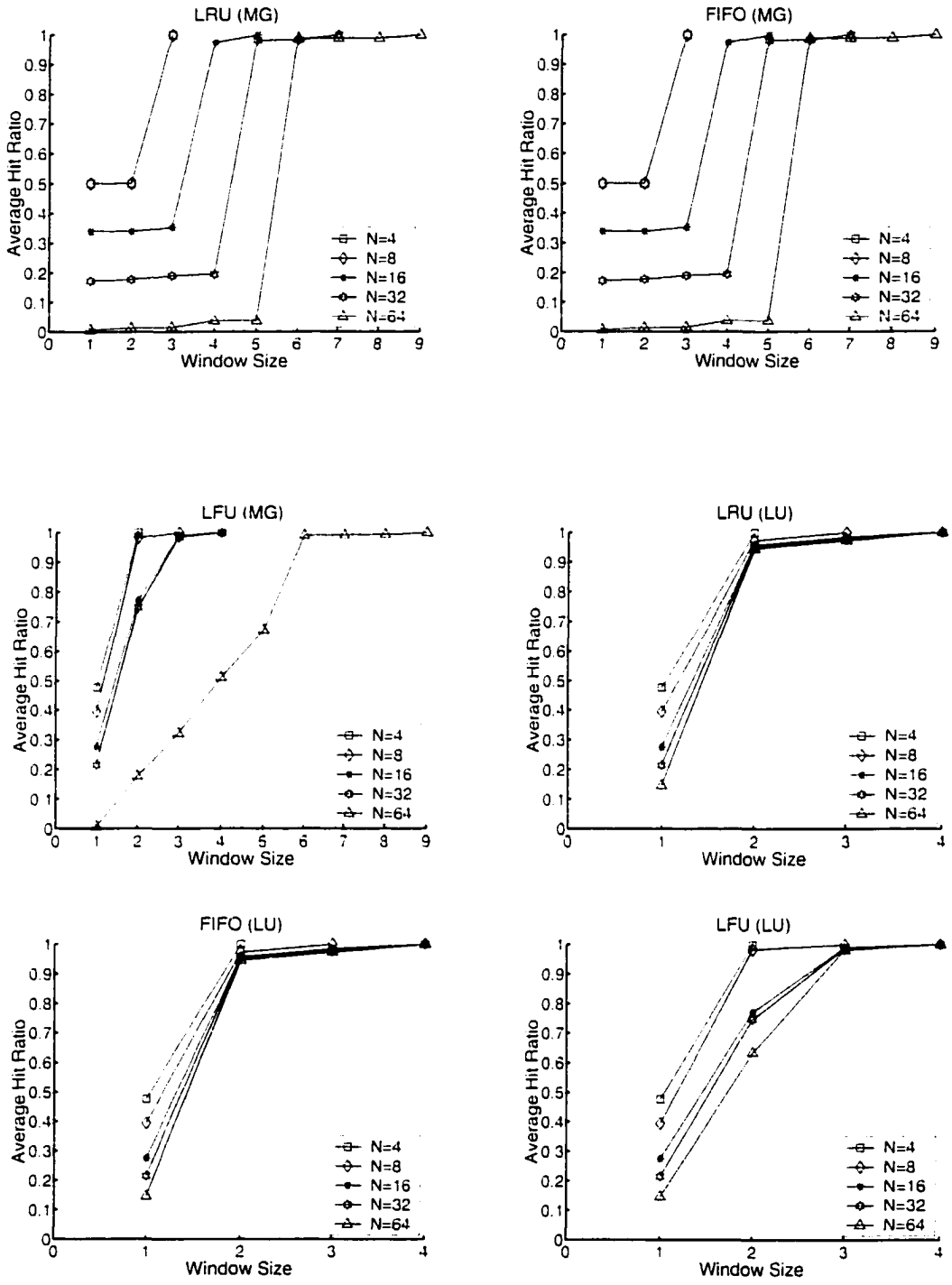


Figure 3.8: Effects of the scalibilty of the LRU, FIFO, and LFU heuristics on the MG and LU applications

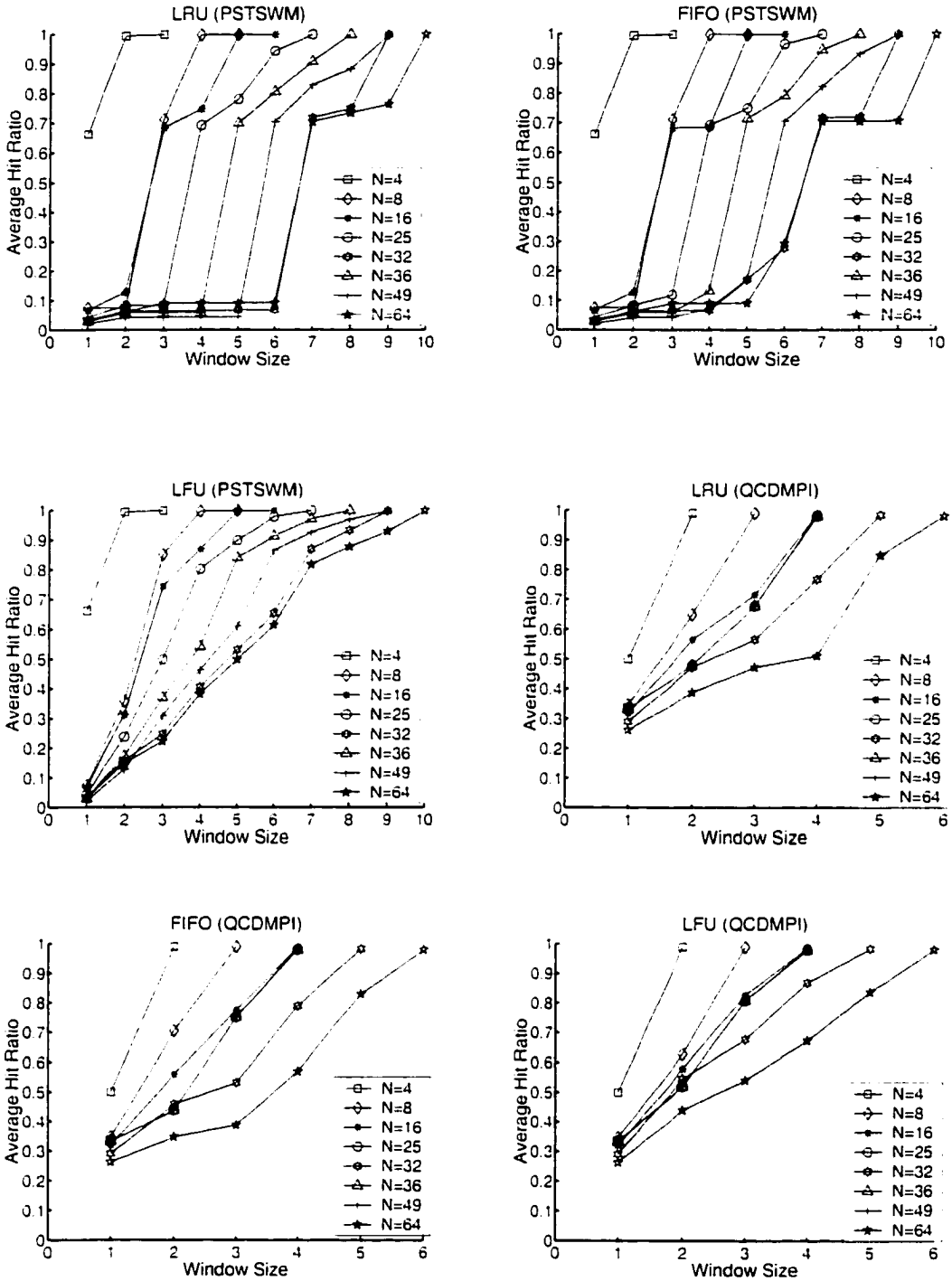


Figure 3.9: Effects of the scalibilty of the LRU, FIFO, and LFU heuristics on the PSTSWM and QCDMPI applications

Basically, the LRU, FIFO and LFU heuristics do not predict exactly the next message destination but show the probability that the next message destination is in the message destination set of the LRU, FIFO and LFU heuristics, respectively. For instance, the PSTSWM application shows nearly 70% hit ratio for a window size of seven under the LRU heuristic when the number of processes is 64. This means that 70% of the time one of the seven most recent message destination will be used in the next message. The LRU, FIFO, and the LFU heuristics perform better when k is sufficiently large. However, this adds to the hardware complexity as k links should be setup and remain active before the next message is ready to be sent.

I am interested in having predictors that can predict the next message destination with a high probability, and work under single-port modeling to minimize the cost of hardware implementation. In the following section, I propose a number of novel message destination predictors.

3.4 Message Destination Predictors

As noted earlier, a node sends a message to another node by first establishing a link to the target (hence the reconfiguration delay d) and then sending the actual message over the established link. It is obvious that if the link is already in place, then the configuration phase does not enter the picture with a commensurate saving in the message transmission time. I would like to establish efficient algorithms where the link establishment costs are minimized. The stated objective can be accomplished, if the target of the communication operation can be *predicted* before the message itself is available. In this way, the communication pathway can be established and be ready to be used as soon as the message to be sent becomes available.

There are several ways of accomplishing this. If the communication operation is regular and known, then it is possible that one can determine the destinations and the instances that these shall be used. I have developed such algorithms for broadcasting/multibroadcasting [1] and discuss them in Chapter 5. However, if the algorithm is not known, as is usually the case for point-to-point communications, the approach mentioned above cannot be used.

Prediction techniques have been proposed in the past to predict the future accesses of sharing patterns and coherence activities in distributed shared memory (DSM) by looking at their observed behavior [96, 77, 73, 133, 34, 107]. These techniques assume that memory accesses and coherence activities in the near future will follow past patterns. Sakr and his colleagues have used time series and neural networks for the prediction of the next memory sharing requests [107]. Dahlgren and his colleagues devised hardware regular stride techniques to prefetch several blocks ahead of the current data block [34]. More elaborate hardware-based irregular stride prefetching approaches have been proposed by Zhang and Torrellas [133]. Kaxiras and Goodman have recently proposed an instruction-based approach which maintains the history of load and store instructions in relation to cache misses and predicting their future behavior [73]. This is in contrast to address-based techniques that keep data-access history for the predictions. Mukherjee and Hill proposed a general pattern-based predictor, *cosmos*, to learn and predict the coherence activity for a memory block in a DSM [96]. Cosmos makes a prediction in two steps. First, it uses a cache block address to index into a message history table to obtain the <processor and message-type> tuples of the last few coherence messages received for that cache block. Then it uses these <processor, message-type> tuples to index a pattern history table to obtain a <processor, message-type> tuple prediction. In a recent paper, Lai and Falsafi proposed a new class of pattern-based predictors, *memory sharing predictors*, to eliminate the coherence overhead on a remote access latency by just predicting the memory request messages, those primary messages that invoke a sequence of protocol actions [77]. It improves prediction accuracy over cosmos by eliminating the acknowledgments messages from the pattern tables. It also reduces memory overhead and perturbation in the tables due to message re-ordering. Both works in [96, 77] are adaptations of Yeh and Patt's two-level *PAp* branch predictor [131]. *PAp* is a two-level adaptive branch predictor based on the past behavior of the same branch.

In software-controlled prefetching, the programmer or compiler decides when and what to prefetch by analyzing the code and inserting *prefetch* instructions. Mowry and Gupta [95] have used software-controlled prefetching, and multithreading to hide and reduce the latency in shared memory multiprocessors.

As stated above, many prediction techniques have been proposed to reduce or hide the latency of a remote memory access in shared memory systems. However, to the best of my knowledge, no prediction technique has been proposed to predict the next message destination for message-passing systems to hide the latency of reconfiguration delay in reconfigurable networks.

I explore the effect that a number of heuristics have in predicting the target of a communication request. The set of predictors proposed in this section [2, 3] predict the message destination of a subsequent communication request based on a past history of communication patterns on a per source process basis. These predictors can be used dynamically at the communication assist or network interface with or without the help of the programmer or a compiler.

Actually, I propose two sets of predictors in this thesis: *Cycle-based* predictors, which are pure dynamic predictors, and *Tag-based* predictors, which are static/dynamic predictors. In Cycle-based predictors, *Single-cycle*, *Single-cycle2*, *Better-cycle*, and *Better-cycle2*, predictions are done dynamically at the network interface without any help from the programmer or compiler. In Tag-based predictors, *Tagging*, *Tag-cycle*, *Tag-cycle2*, *Tag-bettercycle*, and *Tag-bettercycle2*, predictions are done dynamically at the network interface as well, but they require some information to be passed from the program to the network interface. This can be done with the help of the programmer and/or the compiler through inserting instructions such as *pre-connect (tag)* in the program. The Tag-based predictors can be pure dynamic predictors if another level of prediction is done on the tag themselves at the network interface. This way, there is no need for the program to pass pre-connect (tag) information to the network interface. I leave this approach for the future research.

It is worth mentioning that these predictors can be used in any circuit-switched networks including the works proposed in [36, 132]. Dao and his colleagues [36] exploit the communication locality to improve the performance of parallel computers using *wave switching*, a hybrid switching technique for high performance routers in electronic interconnection networks. Wave switching combines wormhole switching and circuit switch-

ing in the same router architecture to reduce the fixed overhead of communication latency by exploiting communication locality. Thus, it is possible to reduce latency for communications that display locality and use pre-established physical circuits. Yuan and others [132] use the communication locality in circuit-switched time-multiplexed optical interconnection networks. They rely upon existing techniques for identifying communication patterns such that their compiled communication algorithms compute the minimal multiplexing degree required for establishing all-optical paths from sources to destinations in such networks.

The predictors can even be useful in reducing the latency in current commercial networks. For example, Myrinet networks [23] have a relatively long routing time compared with link transmission time. Predictors would allow sending the routing header in advance for the predicted message destination. When the message becomes available, it can be directly transmitted through the network if the prediction was correct, thus reducing latency significantly. In case of a mis-prediction, a message tail is forwarded to tear the path down. Obviously, null messages must be discarded at the destination.

As in the LRU, LFU, and FIFO heuristics, I use the *hit ratio* to establish and compare the performance of these predictors. As a hit ratio, I define the percentage of times that the predicted message destination was correct out of all communication requests. The hit ratios presented for the performance of the predictors are either the minimum, the average, or the maximum of the hit ratios taken over all nodes of each application.

3.4.1 The Single-cycle Predictor

The *Single-cycle* predictor is based on the fact that if a group of message destinations are requested repeatedly in a cyclical fashion, then a single port can accommodate these requests by ensuring that the connection to the subsequent message destination in the cycle can be established as soon as the current request terminates. This predictor implements a simple cycle discovery algorithm. Starting with a *cycle-head* message destination (this is the first message destination that is requested at start-up, or the one that causes a miss), I log the sequence of requests until the cycle-head is requested again. This stored sequence constitutes a cycle, and can be used to predict the subsequent requests. If the pre-

dicted message destination coincides with the subsequent requested message destination, then I record a hit. Otherwise, I record a miss and the cycle formation stage commences with the cycle-head being the message destination that caused the miss.

Figure 3.10 illustrates an example for the operation of the Single-cycle predictor. The top trace represents the sequence of requested message destinations, while the bottom trace represents the predicted message destinations according to the Single-cycle predictor. The arrows with the cross represent misses, while the ones with the circle represent hits. The "dash" in place of a predicted message destination indicates that a cycle is being formed, and therefore no predicted message destination is offered (note that this is also added to the misses).

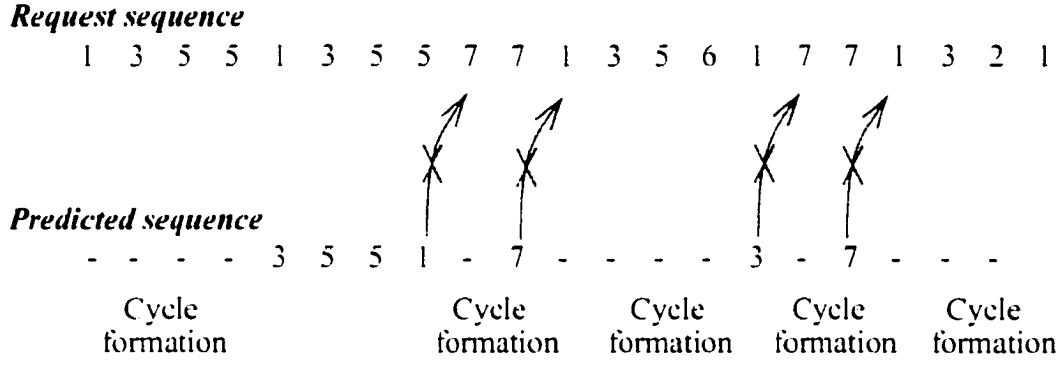


Figure 3.10: Operation of the Single-cycle predictor on a sample request sequence

Figure 3.11, shows the behavior of this algorithm. The performance of the Single-cycle predictor is very good on the CG, LU, MG (except when N = 4, 8), BT and SP (except when N = 4). The Single-cycle predictor behaves poorly on the PSTSWM (except when N = 36, 49) and QCDMPI applications.

The performance of the Single-cycle predictor is much better than the LRU, FIFO and LFU heuristics under the single-port modeling for the LU and CG benchmarks, for the MG, PSTSWM applications (except when N = 4, 8), and for BT and SP (except when N = 4). However, the performance for QCDMPI is almost the same. Note that I compare the performance of the predictors with the LRU, LFU, and FIFO heuristics under single-port modeling for the same optical interconnect implementation cost although the proposed

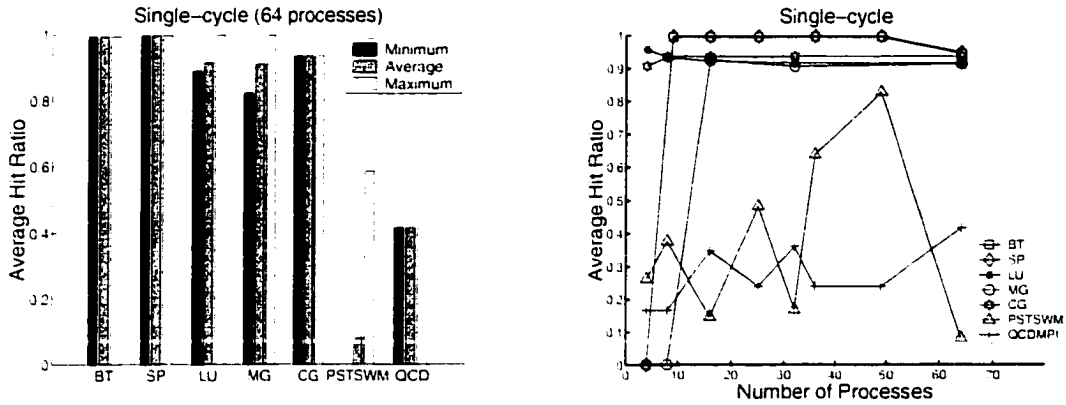


Figure 3.11: Effect of the Single-cycle predictor on the applications

predictors have higher memory requirements (refer to Section 3.5.1). Figure 3.12 compares the performance of the Single-cycle predictor with the LRU, LFU, and FIFO under single-port modeling when $N = 64$.

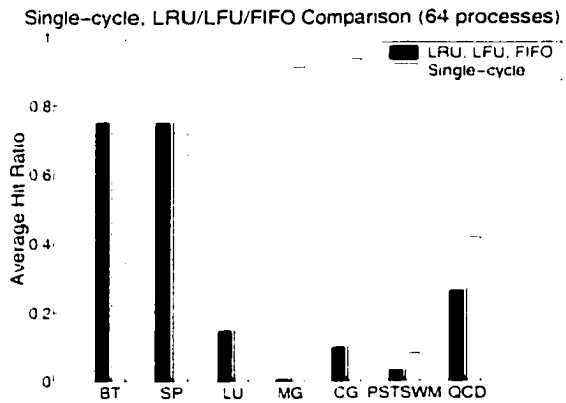


Figure 3.12: Comparison of the performance of the Single-cycle predictor with the LRU, LFU, and FIFO heuristics on the applications under single-port modeling when $N = 64$

3.4.2 The Single-cycle2 Predictor

In the communication traces of some of the applications, there exist cycles of length one (such as the one composed of the requested message destination 7 in Figure 3.10). For these situations, there will always be two misses until the predictor determines that there is a cycle of length one. The *Single-cycle2* predictor is identical to the single-cycle predictor with the addition that during cycle formation, the previously requested message destination is offered as the predicted message destination. If a miss occurs during cycle forma-

tion. the formation phase continues until a cycle is formed. Then and only then misses cause a new cycle formation phase to begin. I applied the Single-cycle2 predictor to the request sequence of the previous example as shown in Figure 3.13. As was expected, the Single-cycle2 predictor reacts better to cycles of length one.

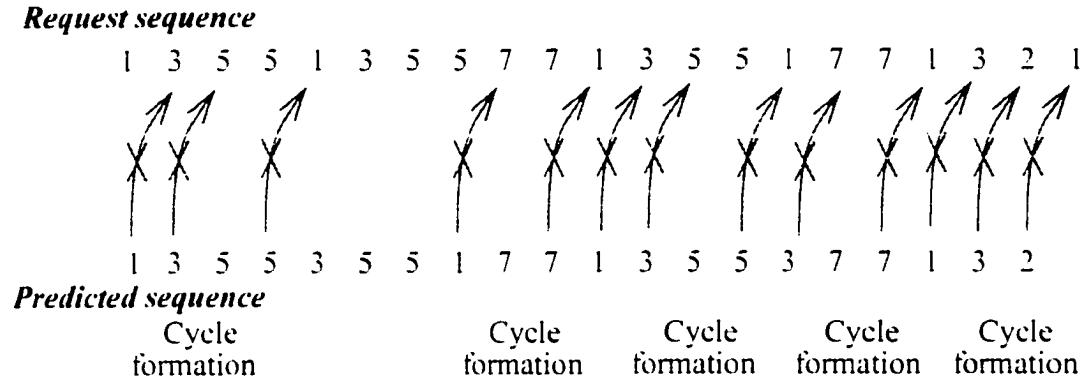


Figure 3.13: Operation of the Single-cycle2 predictor on the sample request sequence

Figure 3.14 illustrates the performance of the Single-cycle2 predictor. This predictor has a better performance than the single-cycle algorithm.

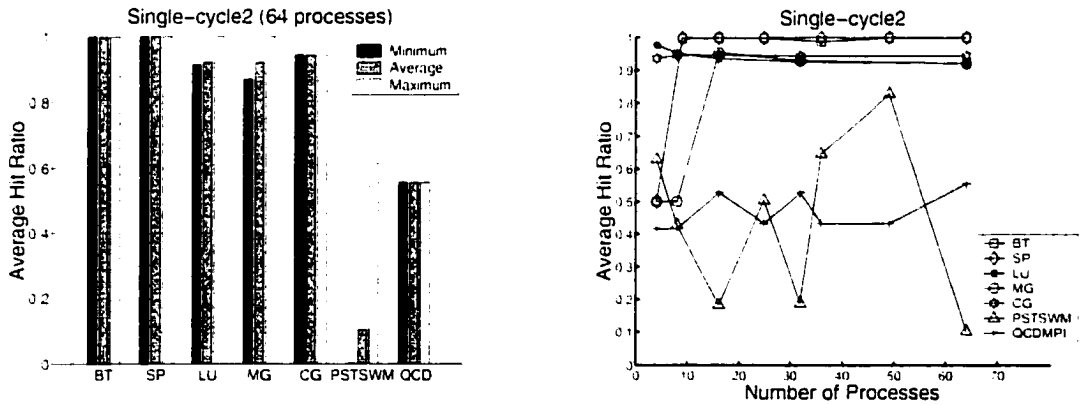


Figure 3.14: Effect of the Single-cycle2 predictor on the applications

3.4.3 The Better-cycle and Better-cycle2 Predictors

In the Single-cycle and Single-cycle2 algorithms, as soon as a message destination breaks a cycle I discard the cycle and start forming a new cycle with this message destination as the new cycle-head. Then I just rely upon the new cycle to predict the next message destination. The Single-cycle and Single-cycle2 predictors could achieve a better performance if the previous cycle information was not discarded as new cycle is formed.

In the *Better-cycle* predictor, each cycle-head has its own cycle. For this, I keep the last cycle associated with each cycle-head encountered in the communication pattern of each process. This means that when a cycle breaks I keep this cycle in memory for the corresponding cycle-head for later references. When a cycle breaks, if I haven't already seen the new cycle-head then I form a cycle for it, otherwise I predict the next message destination based on the member of the cycle associated with this cycle-head that I have from the past in memory. If the predicted message destination coincides with the subsequent requested message destination, then I record a hit. If not, then I record a miss and revise the cycle for this cycle-head. The state diagram of this predictor is shown in Figure 3.15.

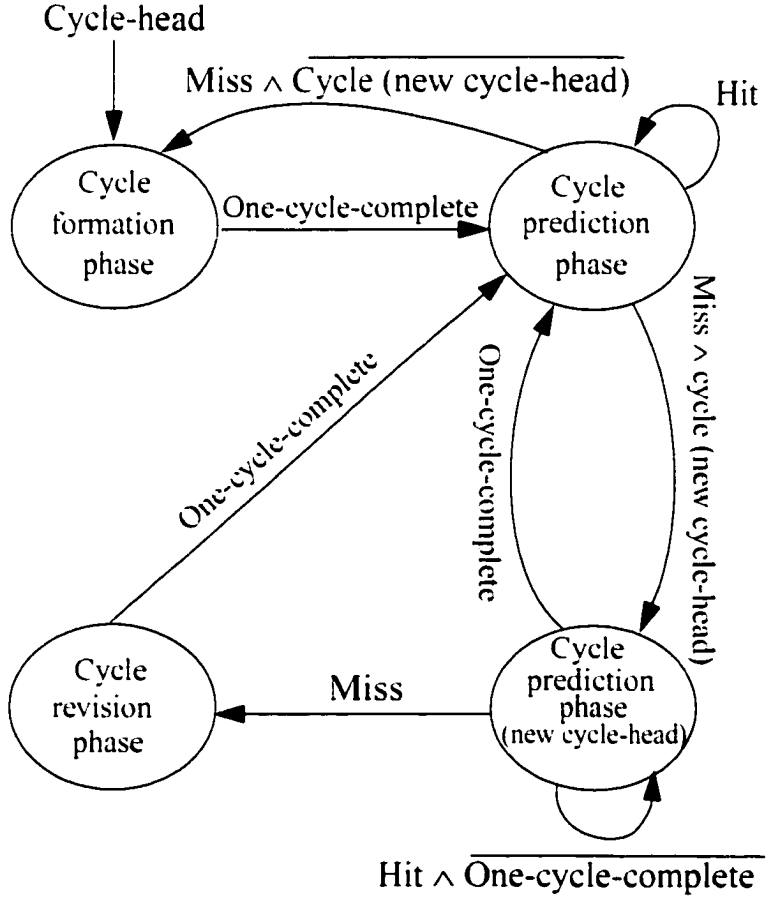


Figure 3.15: State diagram of the Better-cycle predictor

The top left state is the "cycle formation phase" initiated with a cycle-head. This is the same as the cycle formation phase in the Single-cycle predictor. Upon a cycle completion, I enter the "cycle prediction phase". In case of a mis-prediction in the "cycle prediction phase", I move back to the "cycle formation phase" if the new cycle-head has not been visited so far (that is, there is no cycle associated with this new cycle-head in the memory). Otherwise, I move forward to the "cycle prediction phase for the new cycle-head". I move back to "cycle prediction phase" after one complete cycle to continue the predictions for this new cycle-head. In case of a mis-prediction during the first cycle of predictions in the "cycle prediction phase for the new cycle-head", I move to the "cycle-revision phase" to revise the cycle for this new cycle-head. It is clear that after the revision phase, I move to the "cycle prediction phase" for the next cycles of predictions.

Figure 3.16 illustrates the operation of the Better-cycle predictor on the sample request sequence. It is clear that the first cycle associated with cycle-head 1 consists of message destinations 1, 3, 5, and 6. However, in the fourth appearance of this cycle-head a revised cycle forms which contains message destinations 1, 3, and 2.

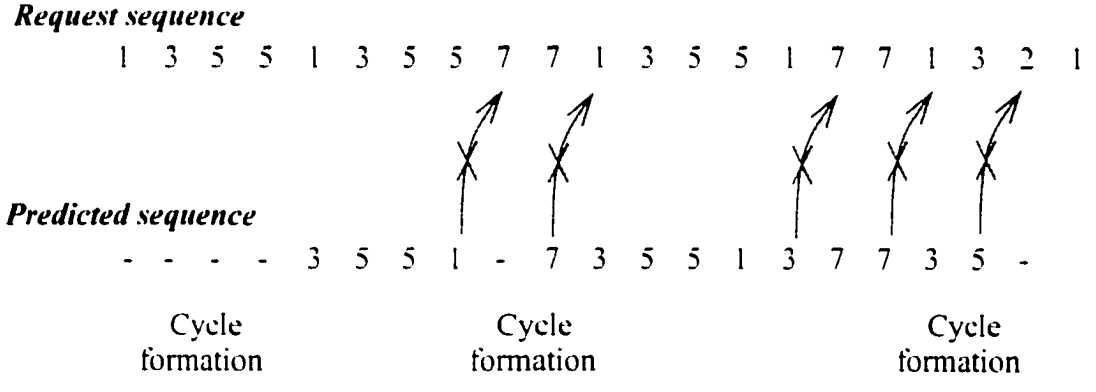


Figure 3.16: Operation of the Better-cycle predictor on the sample request sequence

The performance of the Better-cycle predictor on the benchmarks is shown in Figure 3.17. It is evident that its performance is exceptionally better for all benchmarks compared to the Single-cycle predictor except for the QCDMPI benchmark when N = 25, 32, 36 and 49.

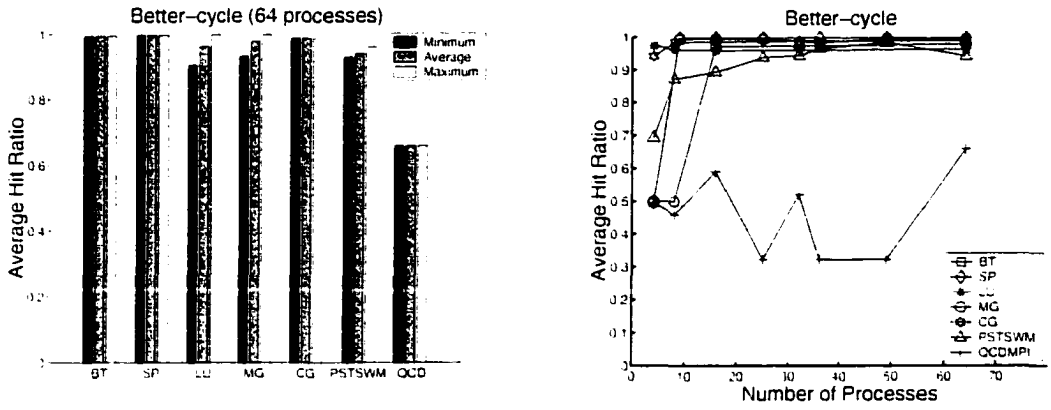


Figure 3.17: Effect of the Better-cycle predictor on the applications

The *Better-cycle2* predictor is identical to the Better-cycle predictor with the addition that during cycle formation and cycle revision phases the previously requested message destination is offered as the predicted message destination. Figure 3.18 illustrates the operation of the Better-cycle2 predictor on the same sample request sequence.

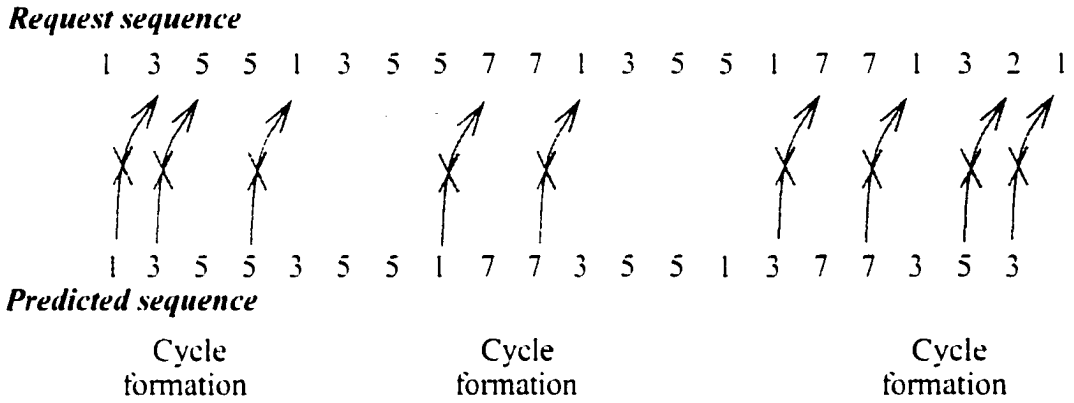


Figure 3.18: Operation of the Better-cycle2 predictor on the sample request sequence

The Better-cycle2 predictor has a better performance than the Single-cycle, Single-cycle2, and the Better-cycle predictor for the QCDMPI benchmark. The performance of this predictor is shown in Figure 3.19. It is worth mentioning that I found that the applications have a very small number of cycle-heads (at most 9) under the Better-cycle and Better-cycle2 predictors and different system sizes. Section 3.5.1 discusses the memory requirement of all predictors proposed in this thesis.

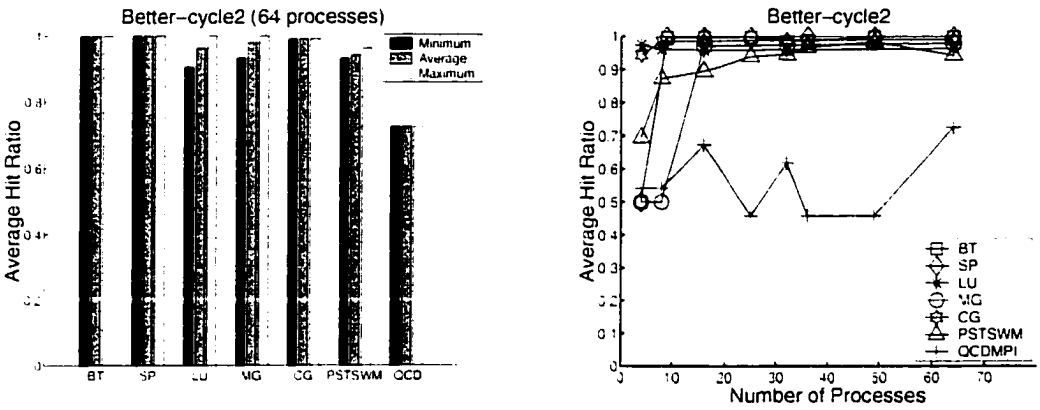


Figure 3.19: Effect of the Better-cycle2 predictor on the applications

3.4.4 The Tagging Predictor

The *Tagging* predictor assumes a static communication environment in the sense that a particular communication request (send) in a section of code, will be to the same message destination with a large probability. Therefore, as the execution trace nears the section of code in question, it can cause the communication subsystem to establish the connection to the target node before the actual communications request is issued. This can be implemented with the help of the compiler or by the programmer through a *pre-connect (tag)* operation which will force the communication system to establish the communication connection before the actual communication request is issued. As noted earlier, for this predictor and other Tag-based predictors, I can avoid the help from the compiler or the programmer by predicting the tag itself at the network interface. This way, there is no need for the program to pass pre-connect (tag) information to the network interface. However, the performance of these *2-level Tag-based* prediction techniques has not been evaluated yet.

I attach a different *tag* (this is different than the tag in an MPI communication call; it may be a unique identifier or the program counter at the address of the communication call) to each of the communication requests found in the applications. This tag is passed to the communication subsystem by the pre-connect (tag) operation. To this tag and at the communication assist, I assign the requested message destination the first time a link is

established. A hit is recorded if in subsequent encounters of the tag, the requested message destination is the same as the one already associated with the tag. Otherwise, a miss is recorded and the tag is assigned the newly requested message destination.

The performance of the Tagging predictor is presented in Figure 3.20. As can be seen, the Tagging predictor results in excellent performance (hit ratios in the upper 90%) for all the application benchmarks except the CG, PSTSWM, and QCDMPI. The reason is that these benchmarks include send operations with message destinations calculated based on loop variables. Thus, the same section of code cycles through a number of different message destinations. As we have seen earlier, the Better-cycle and Better-cycle2 predictors are excellent in discovering such cyclic occurrences for the CG and PSTSWM benchmarks. Meanwhile, the Better-cycle2 predictor has better performance for the QCDMPI benchmark compared to the Tagging predictor.

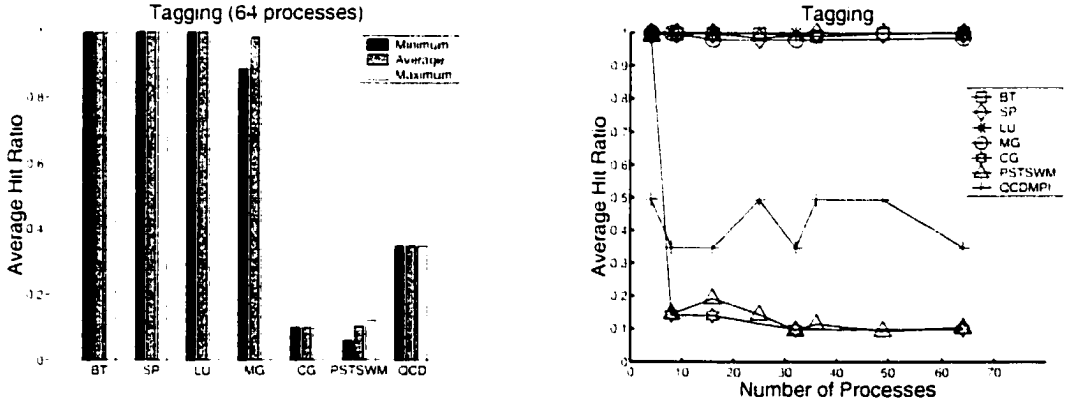


Figure 3.20: Effects of the Tagging predictor on the applications

3.4.5 The Tag-cycle and Tag-cycle2 Predictors

The Tagging predictor does not have a good performance on the CG, PSTSWM, and the QCDMPI benchmarks while the Single-cycle and Single-cycle2 predictors showed good results for the CG benchmark. I combine the Tagging algorithm with the Single-cycle algorithm and call it the *Tag-cycle* algorithm.

In the Tag-cycle predictor, I attach a different tag to each of the communication requests found in the application benchmarks and do a Single-cycle discovery algorithm on each tag. To this tag and at the communication assist, I assign the requested message destination, to be called *tagcycle-head* message destination (this is the first message destination that is requested at this tag, or the one that causes a miss). I log the sequence of the requests at this tag until the tagcycle-head is requested again. This stored sequence constitutes a cycle at each tag, and can be used to predict the subsequent requests. A hit is recorded if in subsequent encounter of the tag, the requested message destination is the same as the predicted one in the cycle. If not, then I record a miss and the cycle formation stage begins with the tagcycle-head being the message destination that caused the miss. The Tag-cycle predictor performs exceptionally well across all the benchmarks except for the QCDMPI benchmark as shown in Figure 3.21.

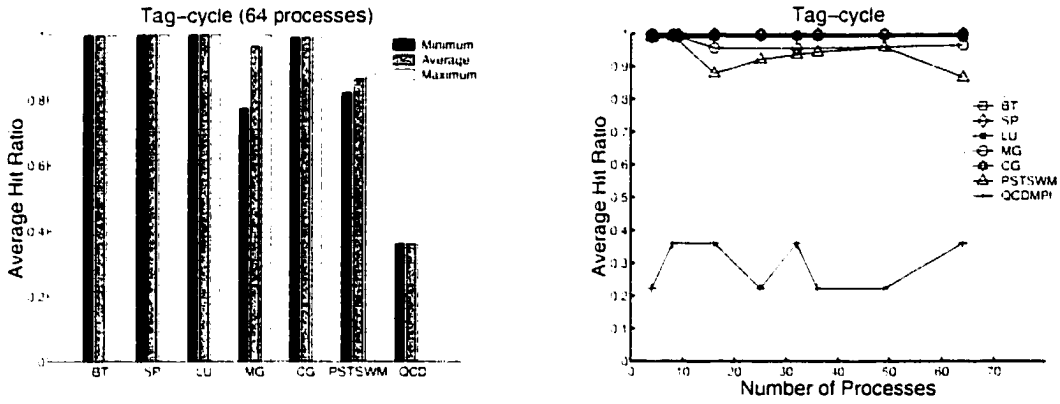


Figure 3.21: Effects of the Tag-cycle predictor on the applications

The *Tag-cycle2* predictor is identical to the Tag-cycle predictor with the addition that during cycle formation, similar to the Single-cycle2 predictor, the previously requested message destination is offered as the predicted one. The performance of the Tag-cycle2 predictor, as shown in Figure 3.22, is better than the Tagging and Tag-cycle predictors for all benchmarks.

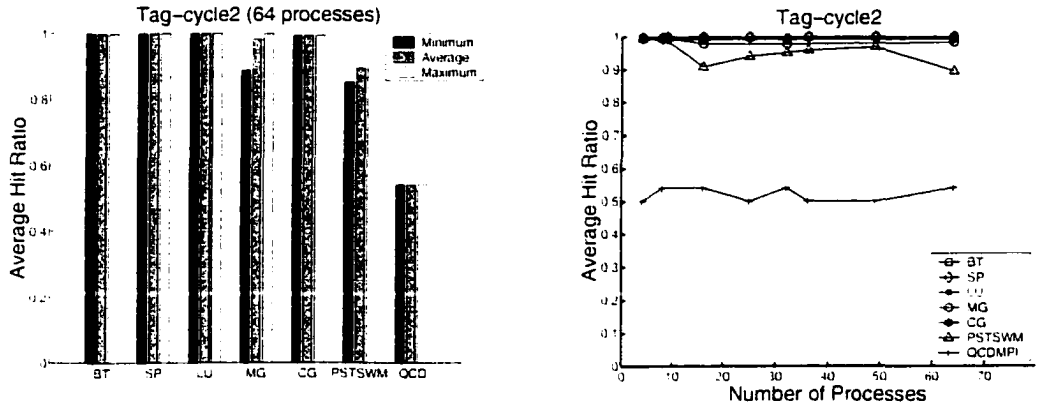


Figure 3.22: Effects of the Tag-cycle2 predictor on the applications

3.4.6 The Tag-bettercycle and Tag-bettercycle2 Predictors

The Better-cycle and Better-cycle2 algorithms have better performance on the parallel applications than the Single-cycle and Single-cycle2 algorithms. Therefore, I combine the Better-cycle and Better-cycle2 algorithms with the Tagging algorithm to get better performance than the Tag-cycle and Tag-cycle2 algorithms. I call these *Tag-bettercycle* and *Tag-bettercycle2* predictors. The performance of these two predictors are shown in Figure 3.23, and Figure 3.24.

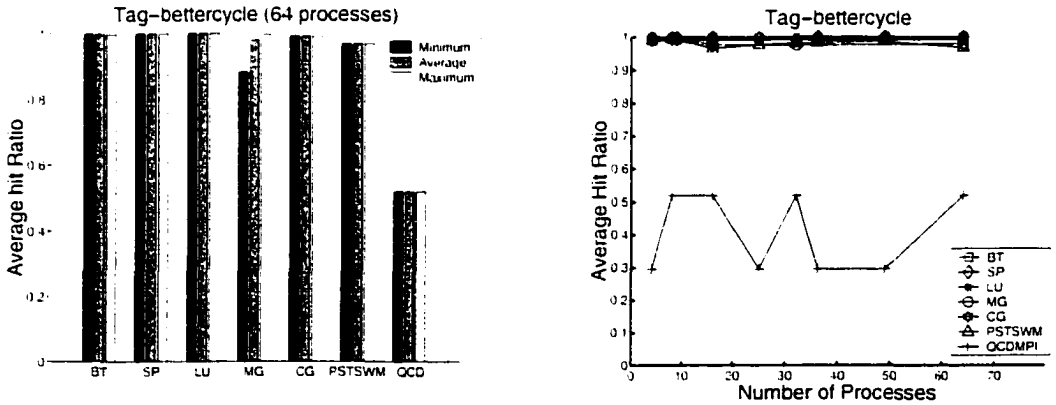


Figure 3.23: Effects of the Tag-bettercycle predictor on the applications

In Tag-bettercycle predictor, I attach a different tag to each of the communication requests found in the benchmarks and do a Better-cycle discovery algorithm on each tag. To this tag and at the communication assist, I assign the requested target node, to be called

tagbettercycle-head node. The Tag-bettercycle2 predictor is identical to the Tag-bettercycle predictor with the addition that during cycle formation, similar to the Better-cycle2 predictor, the previously requested message destination is offered as the predicted message destination. The performance of Tag-bettercycle for the QCDMPI benchmark is better than the Tag-cycle algorithm, but not better than the Tag-cycle2 predictor. However, the Tag-bettercycle2 predictor is superior to all other predictors for all parallel benchmarks. Moreover, I found that the applications have very small number of tagbettercycle-heads (at most 3) under the Tag-bettercycle and Tag-bettercycle2 predictors and different system sizes.

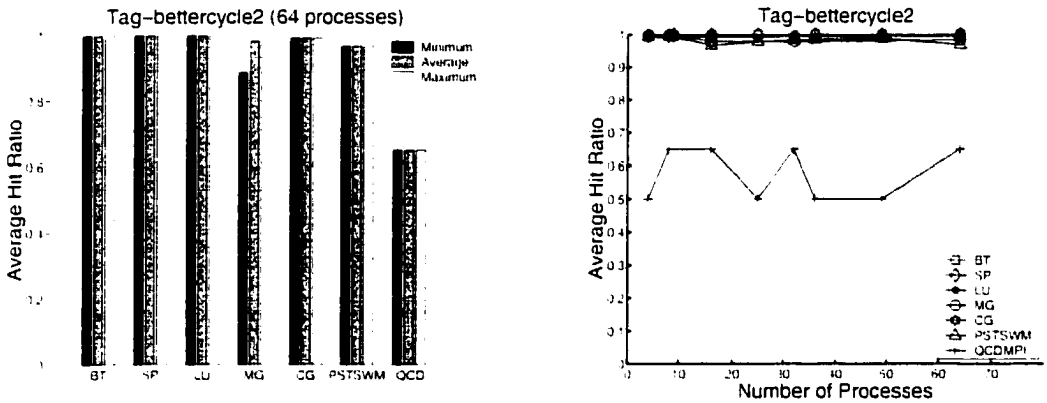


Figure 3.24: Effects of the Tag-bettercycle2 predictor on the applications

3.5 Predictors' Comparison

Figure 3.25, presents a comparison of the performance of the predictors presented in this chapter when the number of processors is 64, 32 and 36, and 16, respectively. It is evident that the Tag-bettercycle2 predictor has the best overall performance for all applications (except for QCDMPI when the number of processes is 16, and 64 where Better-cycle2 has a better performance) and its hit ratio is consistently very high. It is also clear that under single-port modeling, the proposed predictors outperform the classical LRU, LFU, FIFO heuristics.

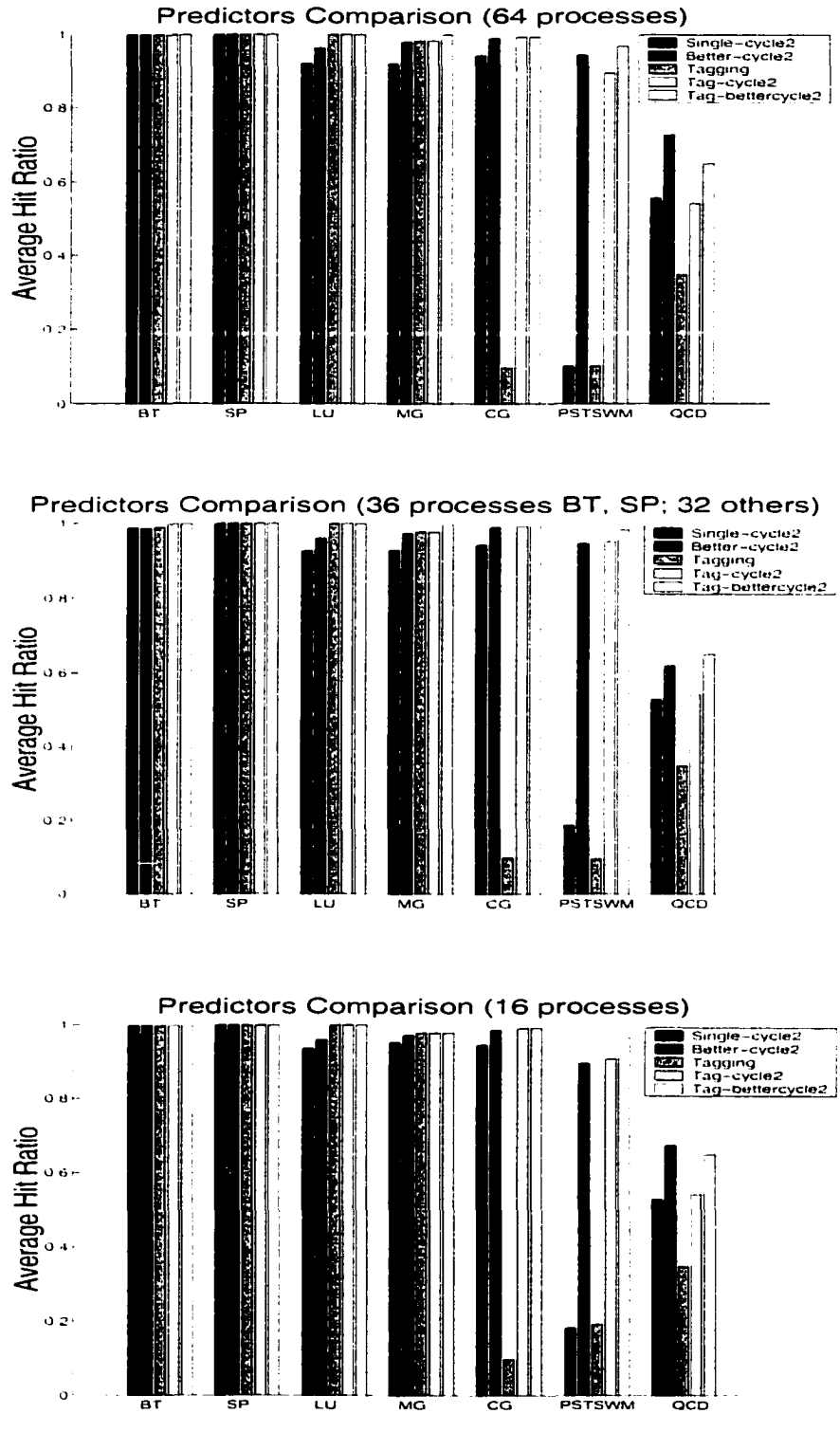


Figure 3.25: Comparison of the performance of the predictors proposed in this chapter when number of processes is 64, 32 (36 for BT and SP), and 16

3.5.1 Predictor's Memory Requirements

Table 3.1 compares the maximum memory requirement of the proposed message destination predictors on the application benchmarks when the number of processors is 64. I have found that the memory requirement of the predictors decrease gradually when the number of processes decreases. The numbers in the table are the multiplication factor for the amount of storage needed to maintain the message destination and its communicator. Having 64 processes in this case study and at most 4 different communicators in the applications, one needs to have only one byte of storage per each message destination and its communicator.

Table 3.1: Memory requirements (in bytes) of the predictors when $N = 64$

	BT	SP	CG	MG	LU	QCD	PSTSWM
Single-cycle(2)	49	49	9	7	4	8	33
Better-cycle(2)	49	49	18	28	12	32	297
Tagging	12	12	10	12	10	2	8
Tag-cycle(2)	24	24	40	24	20	10	48
Tag-bettercycle(2)	24	24	40	36	20	30	48

It is quite clear that the memory requirements of the predictors is very low. That makes them very attractive for implementation on the communication assist or network interface. Comparatively, the Better-cycle, and Tag-bettercycle predictors have a little higher memory requirements than the other predictors. Although, the classical LRU, LFU, and FIFO heuristics need less memory, as stated earlier, the beauty of the proposed predictors lies in the fact that they operate under single-port modeling. That is, only one communication channel is available at any time, and this is reconfigured on demand. This brings the cost of optical interconnect implementation to the minimum. The storage requirement of the predictors have been found using the following formulae:

$$Mem_{Single-cycle(2)} = \text{Maximum cycle length} \quad (3.1)$$

$$Mem_{Better-cycle(2)} = Mem_{Single-cycle(2)} \times \text{Maximum number of cycle-heads} \quad (3.2)$$

$$Mem_{Tagging} = \text{Maximum number of tags} \quad (3.3)$$

$$Mem_{Tag-cycle(2)} = Mem_{Tagging} \times \text{Maximum cycle length of each tags} \quad (3.4)$$

$$Mem_{Tag-bettercycle(2)} = Mem_{Tag-cycle(2)} \times \text{Maximum number of tagbettercycle-heads} \quad (3.5)$$

3.6 Using Message Predictors

In this section, I briefly discuss how a message destination predictor can be used and integrated into the network interface. Predictors would reside beside the communication assist or network interface and accelerate the reconfiguration phase of the interconnect. They monitor the message destination patterns of their host node and make a prediction according to their prediction algorithms. Then, the network interface uses the predictions to establish the links to its final message destinations.

As stated above, the predictors would execute on the communication assist of each node of the parallel machine, and predict the message destinations for communications originating at the node on which they reside based on the past history of communications. In Cycle-based predictors (Single-cycle, Single-cycle2, Better-cycle, and Better-cycle2), predictors do not need any help from the compiler or programmer. However, as stated earlier, in Tag-based predictors (Tagging, Tag-cycle, Tag-cycle2, Tag-bettercycle, and Tag-bettercycle2), predictors require an interface to pass some information from the program to the network interface. With a simple help from the programmer or compiler, this can be

done through inserting *pre-connect (tag)* instructions in the program well above each specific send communication operation but evidently after the previous send communication operation.

Determining when to perform the path setup action (reconfiguration phase) is quite simple. Basically, predictors should map the prediction into the path setup action when the previous communication has terminated. Thus, as soon as the previous message transmission is complete, the communication assist reconfigures the link to the next message destination. It is clear that upon a mis-prediction, the on-going reconfiguration which is not correct and may or may not be completed by the time of the mis-prediction due to a shorter inter-send computation time (to be discussed in Chapter 4) immediately stops and a new reconfiguration takes place.

3.7 Summary

Interconnection networks are still a source of bottleneck for high performance communications in massively parallel environments. In this chapter, I introduced a reconfigurable interconnection network that could alleviate the communication problems in such environments.

In order to benefit from such interconnects effectively, reconfiguration delay should be hidden. For this, I analyzed the communication properties of some parallel applications in terms of communication frequency and message destination distributions. Using classical memory hierarchy heuristics, I found that message destinations display a form of locality.

Having message destination locality in parallel applications, I proposed a number of predictors that can be used to accurately predict the message destination of the subsequent communication request. The proposed predictors would execute on the communication assist of each node of the parallel machine. The performance of the proposed predictors, especially Better-cycle2 and Tag-bettercycle2, are very good and they could effectively hide the hardware communication latency by reconfiguring the communications network concurrently to the computation.

For these predictors to be used efficiently, I shall argue, in Chapter 4, that at least in the application benchmarks studied, there is enough computation preceding a communication request such that the predictors could effectively hide the reconfiguration cost [4.3].

Chapter 4

Reconfiguration Time Enhancements Using Predictors

To reconfigure the optical interconnect concurrently to the computation, or to spectulatively setup the path in electronic interconnects, two conditions are necessary: (1) An accurate prediction of the destination; (2) Enough lead time so that the reconfiguration of the interconnect (or the path setup phase) be completed before the communication request arrives.

In Chapter 3, I utilized the message destination locality property of parallel applications to devise a number of heuristics that can be used to "predict" the target of subsequent communication requests. This technique, can be applied directly to reconfigurable interconnects to hide the communications latency by reconfiguring the communication network concurrently to the computation.

I present the pure execution times of the computation phases of the parallel benchmarks on the IBM Deep Blue machine at the IBM T. J. Watson Research Center using its high-performance switch under the user space mode. This chapter contributes by arguing that by comparing the inter-communication computation times of these parallel benchmarks with some specific reconfiguration times, most of the time, we are able to fully utilize these computation times for the concurrent reconfiguration of the interconnect when we know, in advance, the next target using one of the proposed high hit-ratio target prediction algorithms introduced in Chapter 3.

In this chapter, I first show the distribution of message sizes of the applications in Section 4.1. In Section 4.2, the pure inter-send computation times of the parallel applications on an IBM SP2 machine is presented. I present the performance enhancements of the proposed predictors on the application benchmarks for the total reconfiguration time in Section 4.3. In Section 4.4, I discuss how the predictors at the send side affect the receive side of communications. Finally, I conclude this chapter in Section 4.5.

4.1 Distribution of Message Sizes

The volume of communications is characterized by the number of messages, and the distribution of message sizes in the applications. I presented the number of messages in Chapter 3. In this chapter, I am particularly interested in the distribution of message sizes in the applications. In Section 4.3, I use the size of messages in the applications to calculate the message transfer delay time. Figure 4.1 through Figure 4.4 illustrate the distribution of message sizes of all applications under different systems sizes. The MG, PSTSWM, SP, and BT applications use more distinct message sizes in their communication calls than the other applications. The CG, LU, and QCDMPI use a few number of distinct message sizes.

4.2 Inter-send Computation Times

In Section 4.3, I shall examine the effectiveness of the proposed predictors. I shall quantify the ability of the proposed predictors in hiding the reconfiguration delays. For this, I need to know the pure computation times between any two send communication operations.

I did experiments on a fast machine to establish the inter-send computation times and the effects of the heuristics on the total reconfiguration delay. I used the IBM SP2 Deep Blue machine at IBM T. J. Watson Research Center, a 30 node machine with 160 MHZ P2SC thin nodes, 256MB RAM and a second generation high performance switch and ran the suite of applications, one process on each node under the user space mode, when I was the only user of this machine. This avoided any task switching that might have affected my measurements. My measurements determined a lower bound on the *inter-send* computation times (i.e. the time devoted to computation between any two send communication call).

I excluded all timing overheads in the profiling codes to compute the execution times of the computation and communication phases of the parallel application benchmarks. The inter-send computation measurements excluded any overhead associated with any other

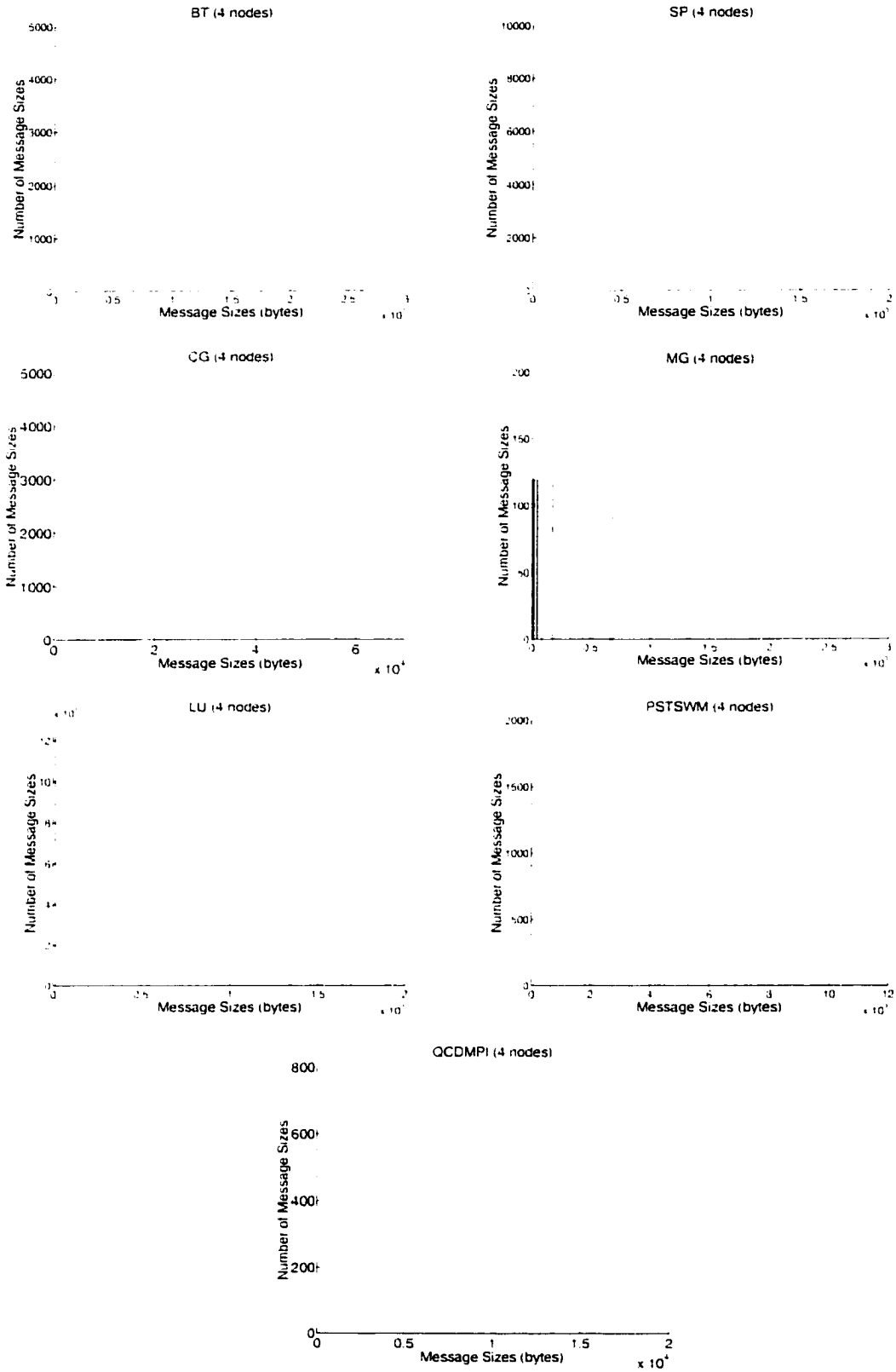


Figure 4.1: Distribution of message sizes of the applications when $N = 4$

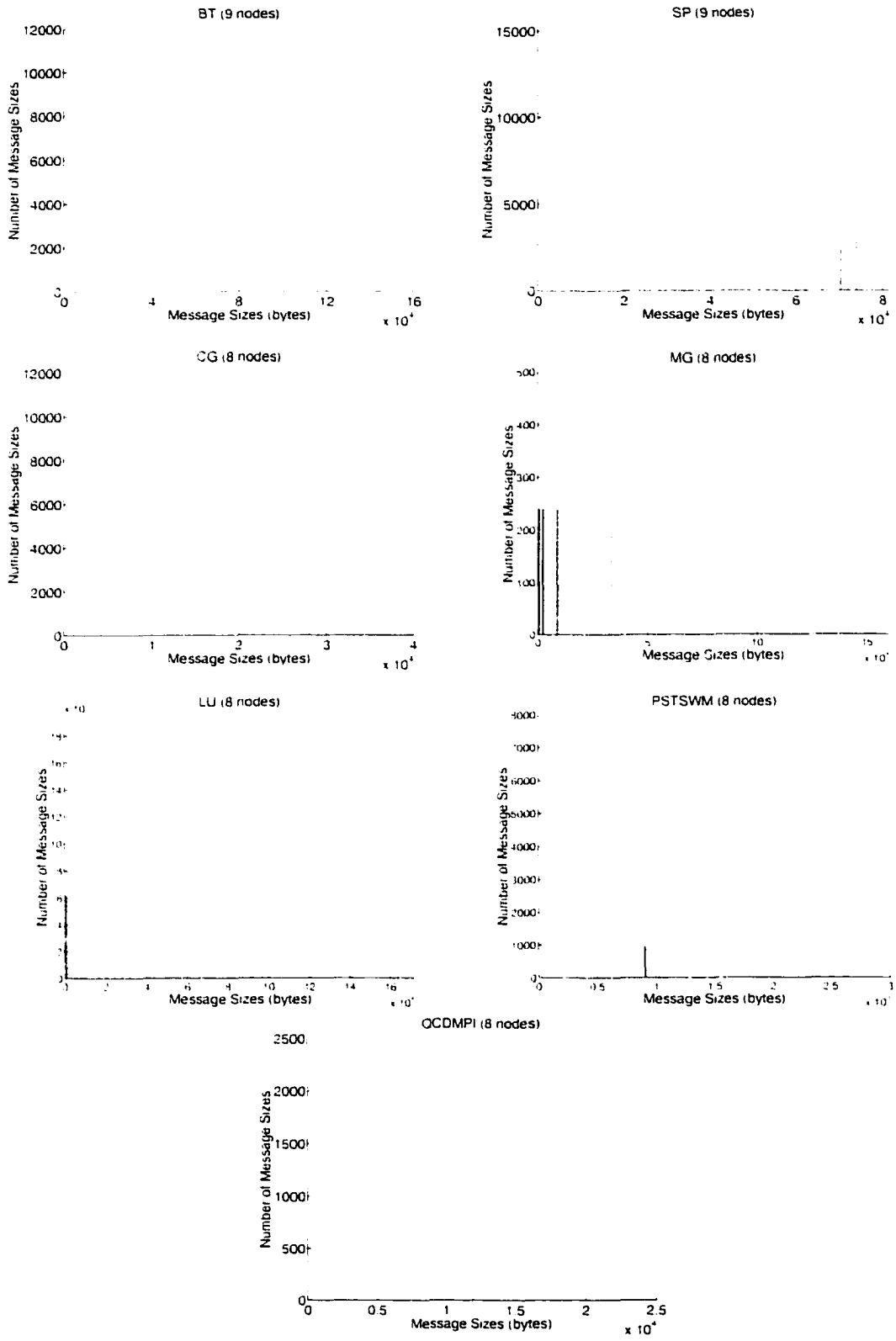


Figure 4.2: Distribution of message sizes of the applications when $N = 9$ for BT and SP, and 8 for CG, MG, LU, PSTSWM, and QCDMPI

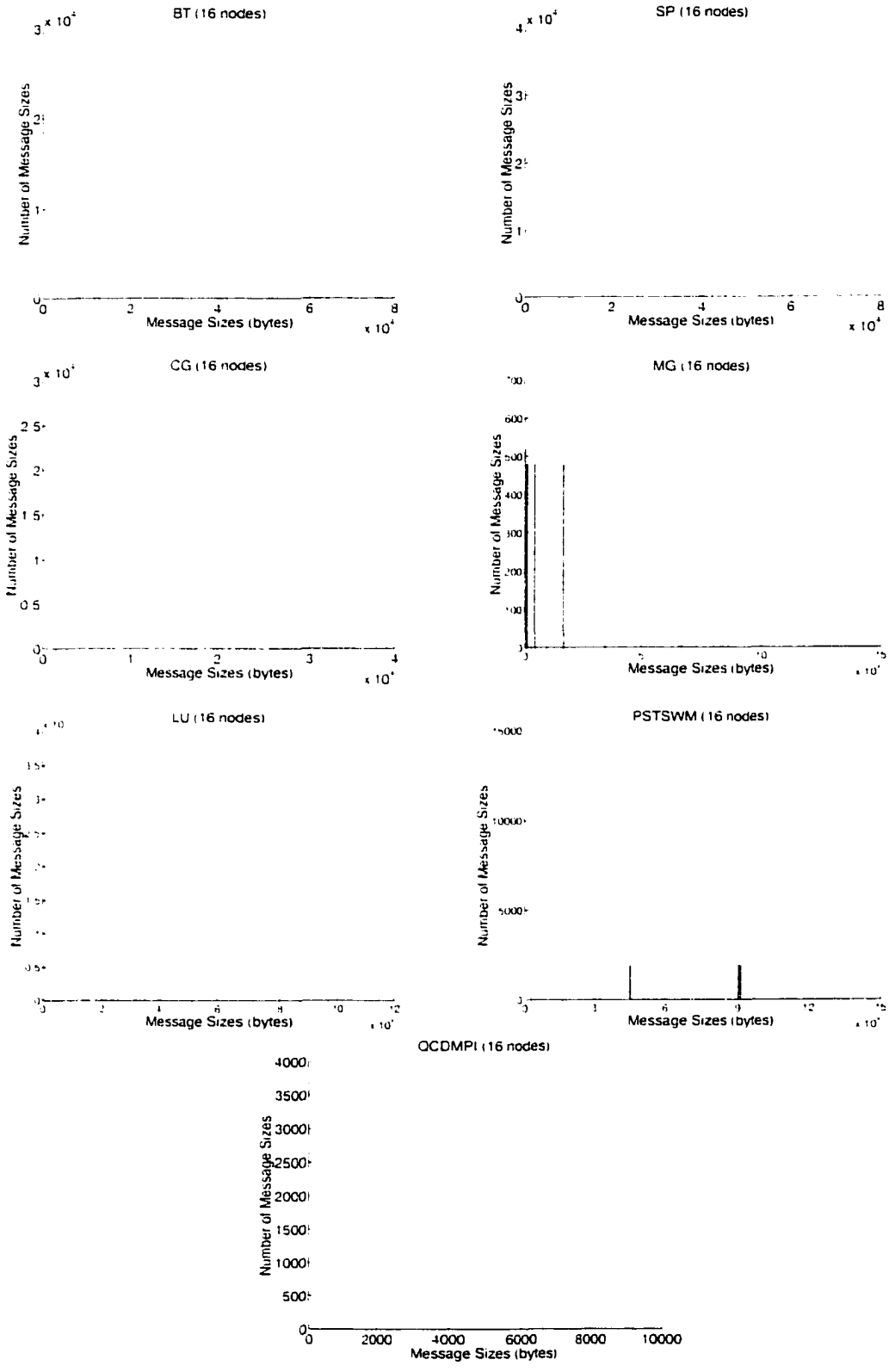


Figure 4.3: Distribution of message sizes of the applications when $N = 16$

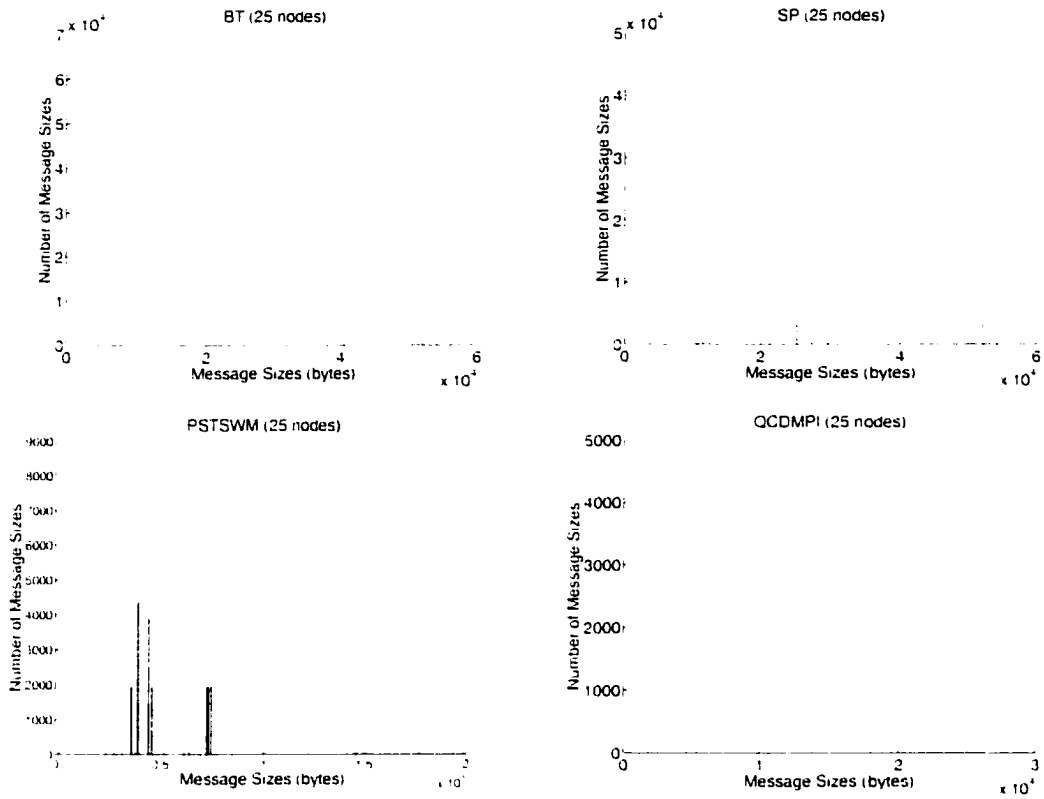


Figure 4.4: Distribution of message sizes of the BT, SP, PSTSWM, and QCDMPI applications when $N = 25$

communication primitives (e.g. receive communication call, collective communications). Thus it can be considered as a lower bound on the pure computation time. In Appendix A, I explain how the pure inter-send computation times have been computed.

The temporal attribute of inter-send computations in parallel applications characterizes the rate of computations. The inter-arrival times of the computation time can be used to obtain the *cumulative distribution function* (CDF) of the computation times. The CDF of the computation times can then be used for curve fitting to generate the inter-arrival times of computation times for simulation purposes. Figure 4.5 presents the cumulative distribution function of the inter-send computation times for node zero of the applications (16 nodes for CG, MG, and LU; 25 nodes for BT, SP, PSTSWM, and QCDMPI). Note that I have found similar cumulative distribution function plots for other system sizes.

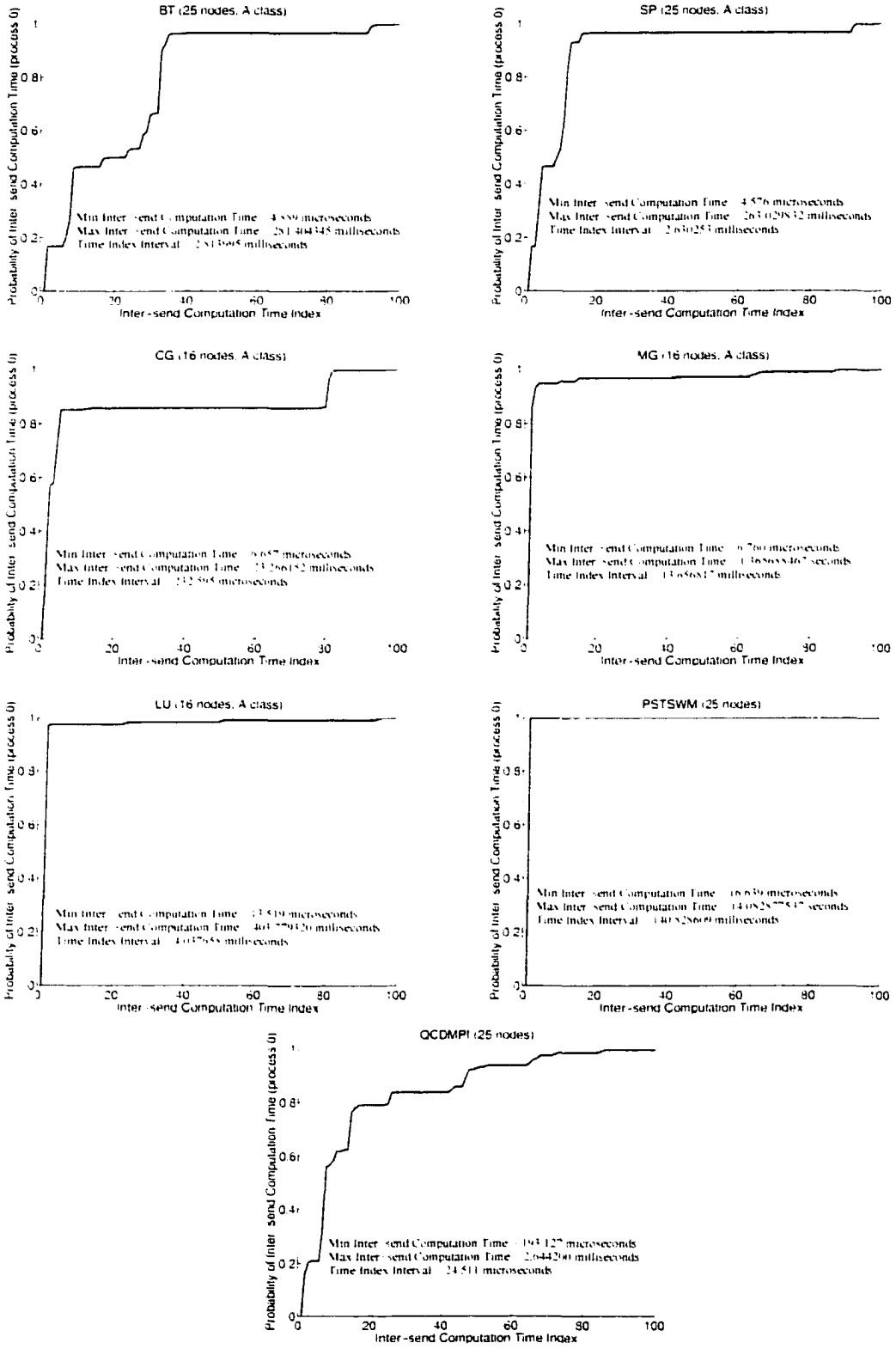


Figure 4.5: Cumulative distribution function of the inter-send computation times for node zero of the application benchmarks when the number of processors is 16 for CG, MG, and LU, and 25 for BT, SP, QCDMPI, and PSTSWM.

Table 4.1 shows the minimum pure inter-send computation times of the applications under different system sizes. Note that LU, MG, and CG run only on a power-of-two number of processors. The inter-send computation times for the CG (4 nodes) and QCDMPI application benchmarks are quite large while all other applications have a minimum of less than 23 microseconds pure computation times.

Table 4.1: Minimum inter-send computation times (microseconds) in NAS Parallel Benchmarks, PSTSWM, and QCDMPI when $N = 4, 8, 9, 16$, and 25

	4 nodes	**8 nodes (9 for BT, SP)**	**16 nodes**	**25 nodes**
BT (W)	4.161	4.161	4.161	4.161
BT (A)	4.576	4.472	4.472	4.889
SP (W)	4.161	4.161	4.161	4.161
SP (A)	4.784	4.472	4.472	4.576
LU (W)	9.568	8.216	8.112	---
LU (A)	22.568	12.688	13.519	---
MG (W)	6.344	5.720	5.928	---
MG (A)	7.592	7.176	6.760	---
CG (W)	407.99	6.864	7.384	---
CG (A)	829.92	7.176	6.657	---
PSTSWM	7.176	6.240	6.032	16.639
QCDMPI	1392.352	695.344	353.080	193.127

IBM Deep Blue uses a state-of-the-art high performance CPU, Power2-Super (P2SC) microprocessor, in its nodes. The nodes are interconnected via an adapter to a high performance, multistage, packet-switched network for interprocessor communications. I am interested in having a rough comparison between the pure inter-send computation times of the applications running on such powerful machines and the current state-of-the-art reconfiguration delay associated with optical interconnects. Researchers in optical engineering are using different approaches to design reconfigurable interconnects [103, 81]. In [103], the authors report a 25 microseconds reconfiguration delay for their experimental recon-

figurable interconnects. Based on these reports, I compare the pure computation times of the application benchmarks with 25 microseconds reconfiguration time, and with reconfiguration times of 10, 5, and 1 microseconds as a measure of future advancements in the area of reconfigurable interconnects. Figure 4.6 presents the distribution of the inter-send computation times on different applications when the computation times are more than 5, 10, 25 microseconds and the number of processors is 4, 8 or 9, 16, and 25.

Examining the distribution of the inter-send times, revealed that they are quite widely distributed. All applications have nearly 100% inter-send computation times that are greater than 5 microseconds. For the BT, SP, LU, MG, and CG (except 4 nodes) application benchmarks, between 60% to 80% of the computation times are above 25 microseconds. The PSTSWM and QCDMPI application benchmarks have nearly 100% inter-send computation times that are greater than 25 microseconds. It is evident that the majority of the reconfigurations can proceed in parallel with the computation and be readied before the end of the computation. For the cases where the computation time is not sufficiently long to completely hide the reconfiguration it effectively reduces the reconfiguration cost by the corresponding length of time.

4.3 Total Reconfiguration Time Enhancement

I assume a multicomputer with nodes similar to the thin nodes of an IBM SP2 system but with a reconfigurable optical interconnect which has a reconfiguration delay d (d = 1, 5, 10, 25 microseconds). It is interesting to see the effectiveness of the proposed predictors on such a multicomputer system. Specifically, I shall quantify the ability of the proposed predictors in hiding the reconfiguration delays. For the calculations used to quantify the reconfiguration hiding capabilities of the predictors, I use the lower bound of the inter-send computation times.

Figure 4.7 illustrates different scenarios for message transmission in the multicomputer with the reconfigurable optical interconnect. Note that as soon as a send call is issued, the message can be sent to the destination if the link is already established. Reconfiguration is started as soon as the message is delivered to the destination. Thus, the *message_transfer_delay* (the delay associated with the transfer of a message) reduces the

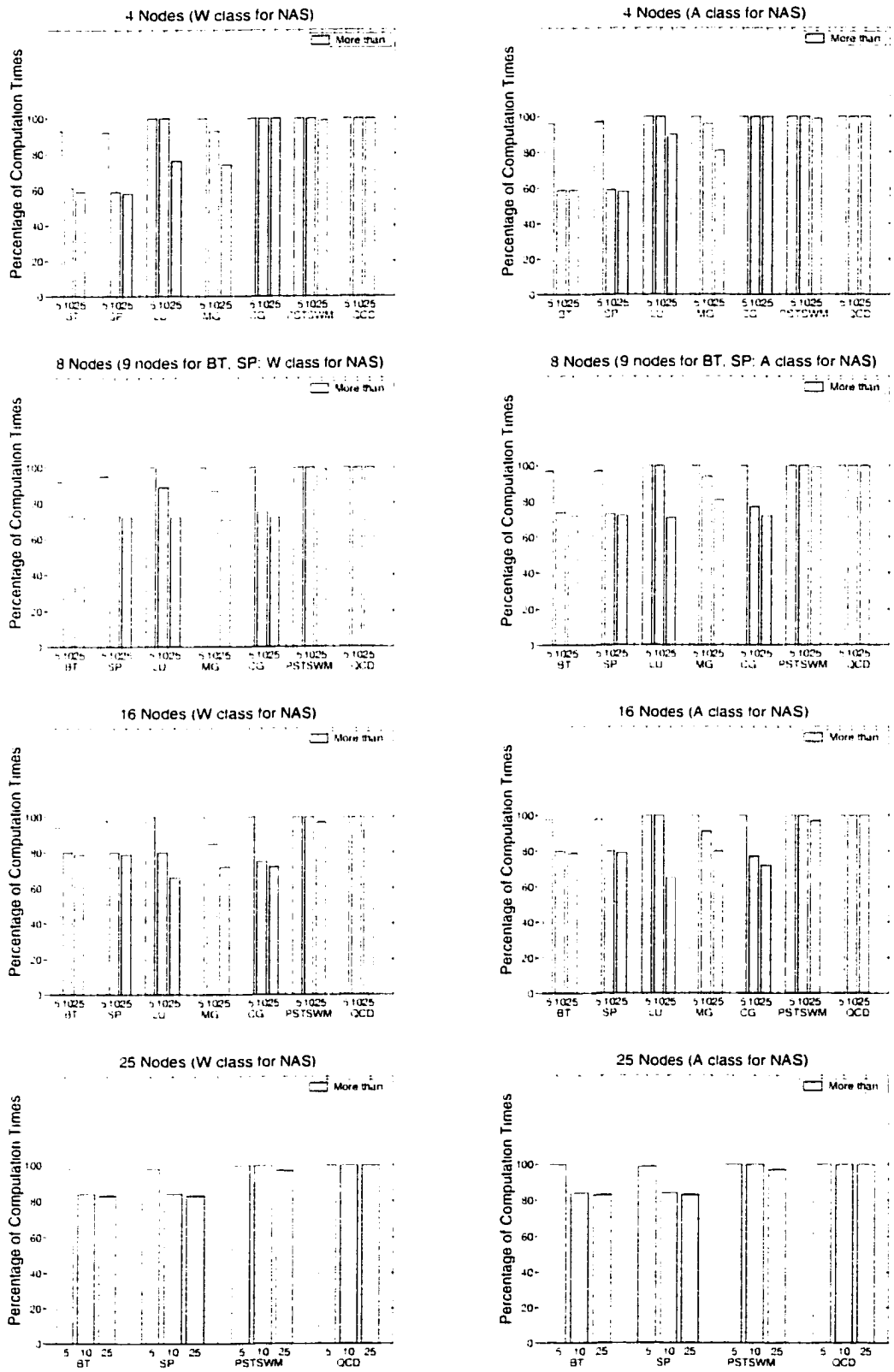


Figure 4.6: Percentage of the inter-send computation times for different benchmarks that are more than 5, 10, and 25 microseconds when $N = 4$, 8 or 9, 16, and 25.

amount of time available before the next send call is issued. For this, I subtract the *message_transfer_delay* (for the specific message size) from the corresponding *inter-send time* and call the remaining time, the *available_time*. This allows me to compute the lower bound of the times that can be hidden. For each *message_transfer_delay* calculation, I use the corresponding message size and a one Gigabyte per second communication channel.

If the *available_time* is greater than zero as in Figure 4.7(a) (that is the *message_transfer_delay* is less than the corresponding *inter-send time*), and it is more than the *reconfiguration_delay* then a correct prediction would help completely hide the *reconfiguration_delay*. If the *available_time* is greater than zero as in Figure 4.7(b) but it is less than the *reconfiguration_delay* then part of the *reconfiguration_delay* equal to the *available_time* can be hidden. However, if the *available_time* is less than zero as in Figure 4.7(c) (that is the *message_transfer_delay* is greater than the corresponding *inter-send time*), then prediction would not help.

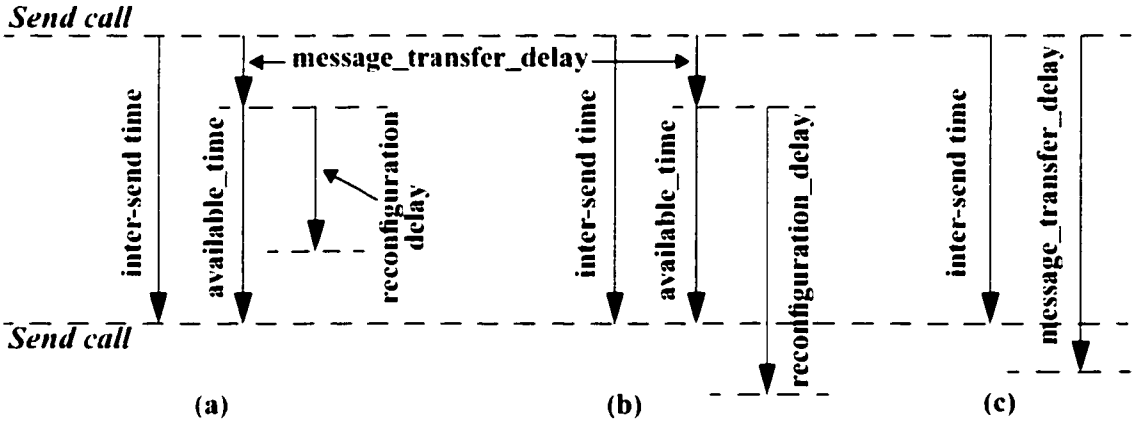


Figure 4.7: Different scenarios for message transmission in a multicomputer with a reconfigurable optical interconnect (a) when the message_transfer_delay is less than the inter_send time, and the available time is larger than the reconfiguration_delay (b) when the message_transfer_delay is less than the inter_send time, and the available time is less than the reconfiguration_delay (c) when the message_transfer_delay is larger than the inter-send time

The algorithm used to obtain the time spent in reconfiguring the interconnect with and without applying the predictors is given by the following pseudocode. The *total_original_reconfiguration* is the sum of the reconfiguration delays encountered in the applications' run-time. The *total_new_reconfiguration* is the sum of the reconfiguration

delays encountered in the applications' run-time when predictions are used to hide them with the inter-send computation times. The *reconfiguration-ratio* is the ratio of *total_new_reconfiguration* over *total_original_reconfiguration*. It is clear that the less this ratio, the better is the predictor's capability to hide the reconfiguration delay.

```
total_new_reconfiguration = 0.0;
total_original_reconfiguration = 0.0;
for each inter_send_computation {
    available_time = inter_send_computation - message_transfer_delay;
    if (available_time < 0) {
        total_new_reconfiguration += reconfiguration_delay;
        total_original_reconfiguration += reconfiguration_delay;
    }
    else {
        if (hit) then
            if (available_time < reconfiguration_delay) then
                total_new_reconfiguration += reconfiguration_delay - available_time;
            else;
        else total_new_reconfiguration += reconfiguration_delay;
        total_original_reconfiguration += reconfiguration_delay;
    }
}
reconfiguration-ratio = total_new_reconfiguration / total_original_reconfiguration
```

Figure 4.8 through Figure 4.11 illustrate the *reconfiguration-ratio*, the average ratio of the total new reconfiguration delay (after applying predictions) over the total original reconfiguration delay for each application benchmark under two different CPU speeds and four different reconfiguration delays. I present the results for two different CPU speeds: one for the current P2SC thin nodes, and one for a 10 times faster CPU as a measure of future CPUs. The results are shown for the best predictors, Better-cycle2 and Tag-bettercycle2. In these figures, shorter bars are better. For the sake of completeness, I have included the results for LRU, LFU, and FIFO heuristics under single-port modeling (recall

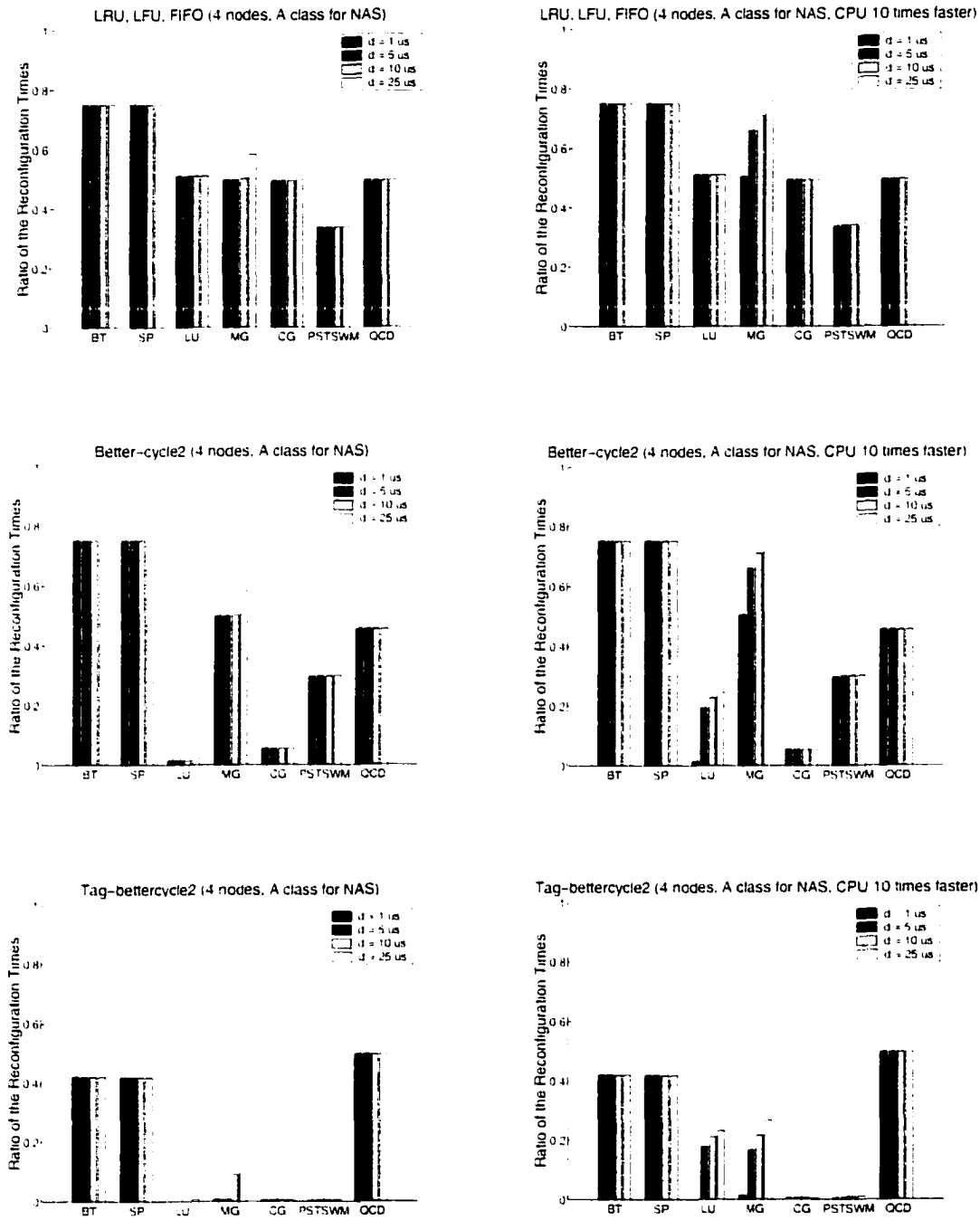


Figure 4.8: Average ratio of the total reconfiguration time after hiding over the total original reconfiguration time for different benchmarks with the current generation and a 10 times faster CPU when d = 1, 5, 10, and 25 microseconds; A class for NPB, 4 nodes (shorter bars are better)

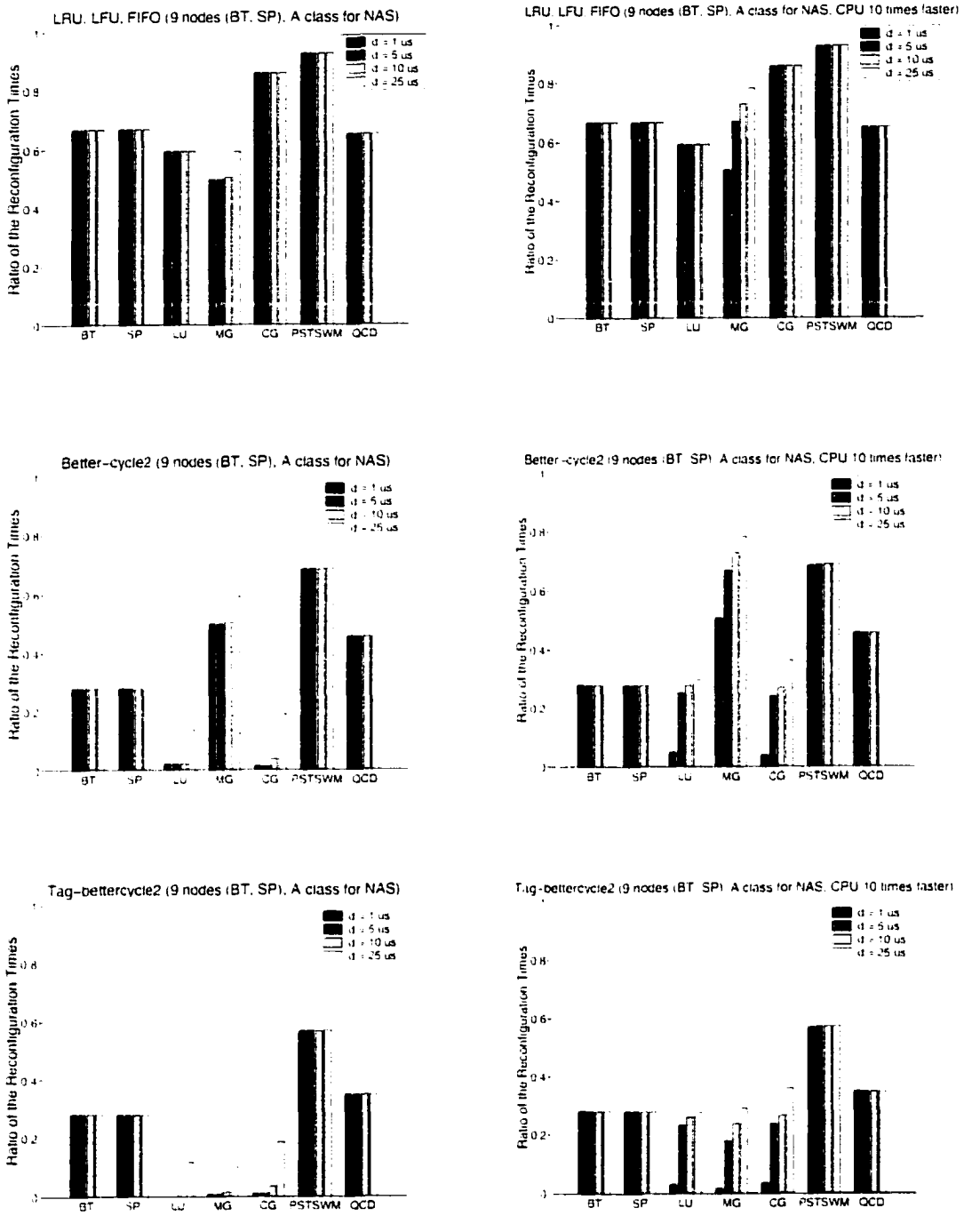


Figure 4.9: Average ratio of the total reconfiguration time after hiding over the total original reconfiguration time for different benchmarks with the current generation and a 10 times faster CPU when d = 1, 5, 10, and 25 microseconds; A class for NPB, 9 nodes for BT and SP, 8 nodes for other applications (shorter bars are better)

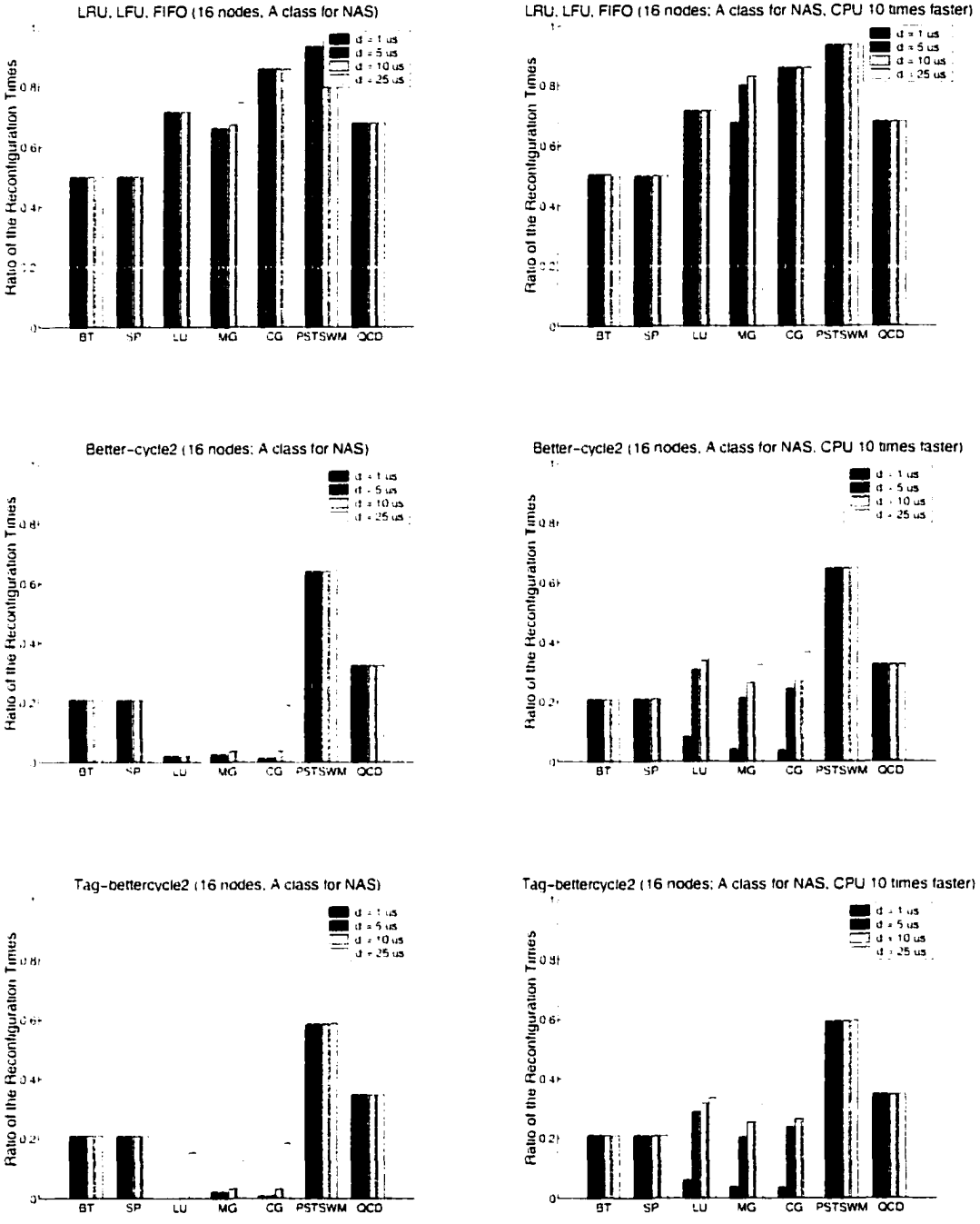


Figure 4.10: Average ratio of the total reconfiguration time after hiding over the total original reconfiguration time for different benchmarks with the current generation and a 10 times faster CPU when d = 1, 5, 10, and 25 microseconds: A class for NPB, 16nodes (shorter bars are better)

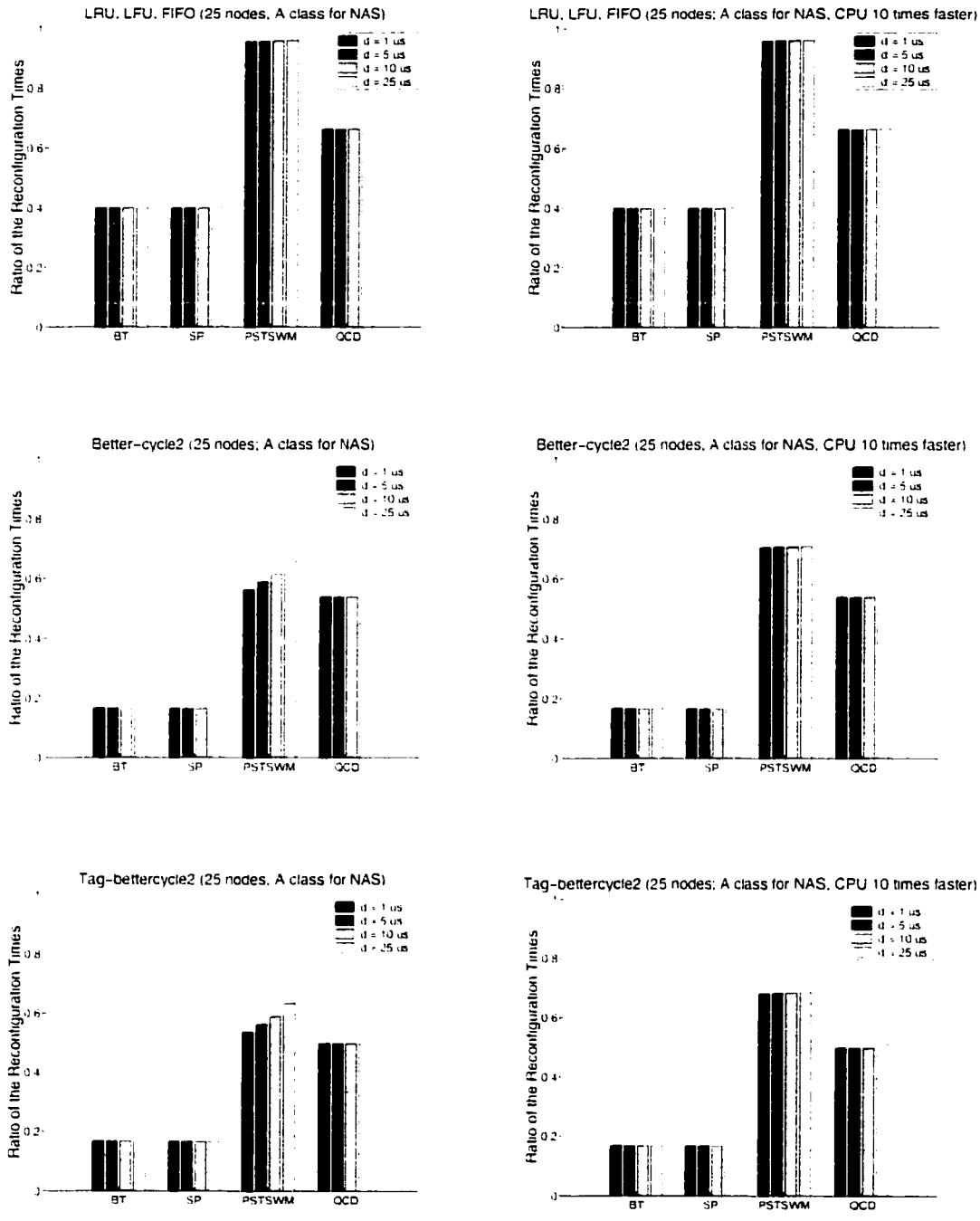


Figure 4.11: Average ratio of the total reconfiguration time after hiding over the total original reconfiguration time for different benchmarks with the current generation and a 10 times faster CPU when d = 1, 5, 10, and 25 microseconds. A class for NPB, 25 nodes (shorter bars are better)

that the LRU, LFU, and FIFO heuristics under single-port modeling predict the next destination to be the same as the previous message destination). It is clear that the Bettercycle2, and Tag-bettercycle2 predictors outperform the LRU/LFU/FIFOF heuristics. The Tag-bettercycle2 predictor improves the total reconfiguration delay better than the Bettercycle2 predictor, especially when the number of processors is 4, or 9. Under the Tag-bettercycle2 predictor, the majority of reconfiguration delays in the CG, MG, and LU benchmarks can be hidden. Meanwhile, the reconfiguration-ratio for BT and SP decreases from 0.4 to 0.18 when the number of nodes increases from 4 to 25. The QCDMPI has a reconfiguration-ratio between 0.3 and 0.5. However, the PSTSWM application shows a consistent reconfiguration-ratio of near 0.6 (except when $N = 4$). It is also evident that the ratios increase with a faster CPU for the same reconfiguration delay. However, the reconfiguration delay time may also decrease in the future. In this respect, it is informative to compare the bar graphs under different reconfiguration delays and processor speeds. From the plots for BT, SP, QCDMPI, and PSTSWM, it seems that the reconfiguration delay is not a factor. It means that either the inter-send computation times are so short that they cannot hide the reconfiguration delays or they are long enough that they can hide large reconfiguration delays.

In general, the results are consistent with the fact that we can hide most of the reconfiguration delays using one of the proposed high hit-ratio predictors. Figure 4.12 shows a summary of the average ratio of the total new reconfiguration delay over the total original reconfiguration delay with the current generation and a 10 times faster CPU when applying the Tag-bettercycle2 predictor on the benchmarks for $d = 25$ microseconds, A class for NPB, and under different system sizes.

4.4 Predictors' Effect on the Receive Side

It is interesting to discover the effect of applying the heuristics at the send side of communications on the receiving sides and hence on the total execution time. Using one of the high hit-ratio predictors reduces the total reconfiguration delay. When this happens at the sender sides, most of the time the messages are delivered sooner at the receiver sides. If

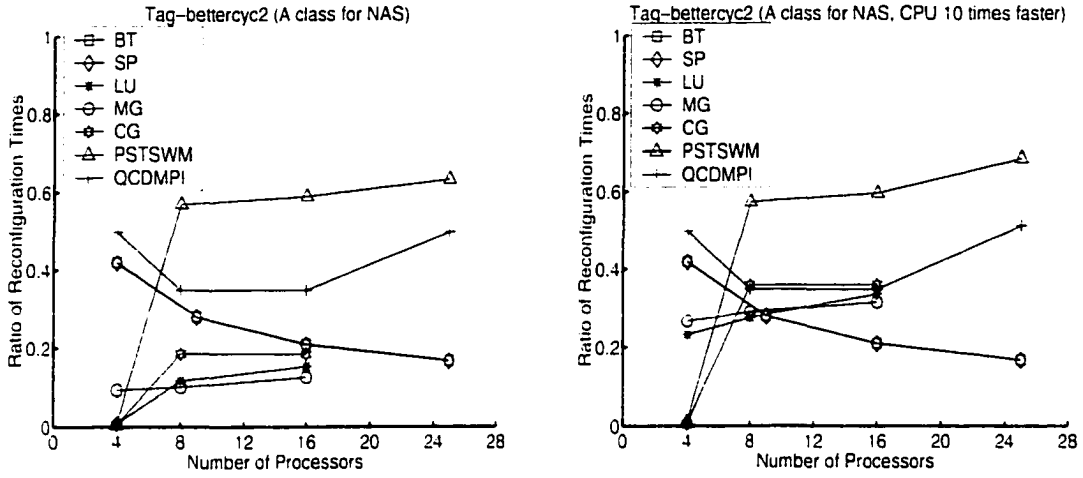


Figure 4.12: Summary of the average ratio of the total reconfiguration time after hiding over the total original reconfiguration time with the current generation and a 10 times faster CPU when applying the Tag-bettercycle2 predictor on the benchmarks with $d = 25$ microseconds, A class for NPB, and under different system sizes

the receive calls have been issued after the message has arrived, there would be no gain. However, if they are issued earlier, then there would be performance enhancement on the receiving side and therefore on the whole execution time. This is shown in the Figure 4.13.

I have used the following strategy for discovering the number of times that the receive calls are issued earlier than their corresponding send calls. I synchronized the timing traces of each node of these applications. I have considered the times just before the send and receive calls are issued. In case of blocking and non-blocking send calls, the time just before the calls (*MPI_Send* and *MPI_Isend*) have been taken into account. That is the time that the message is ready to be sent over. For the blocking receive call (*MPI_Recv*), I did the same. That is the time that the receiver is ready to get the message. However, for the non-blocking receive call (*MPI_Irecv*), I consider the time when the wait call (*MPI_Wait*) is issued for the corresponding receive call (*MPI_Irecv*). This gives us the worst case scenario for the number of times the receive calls are issued before their corresponding send calls.

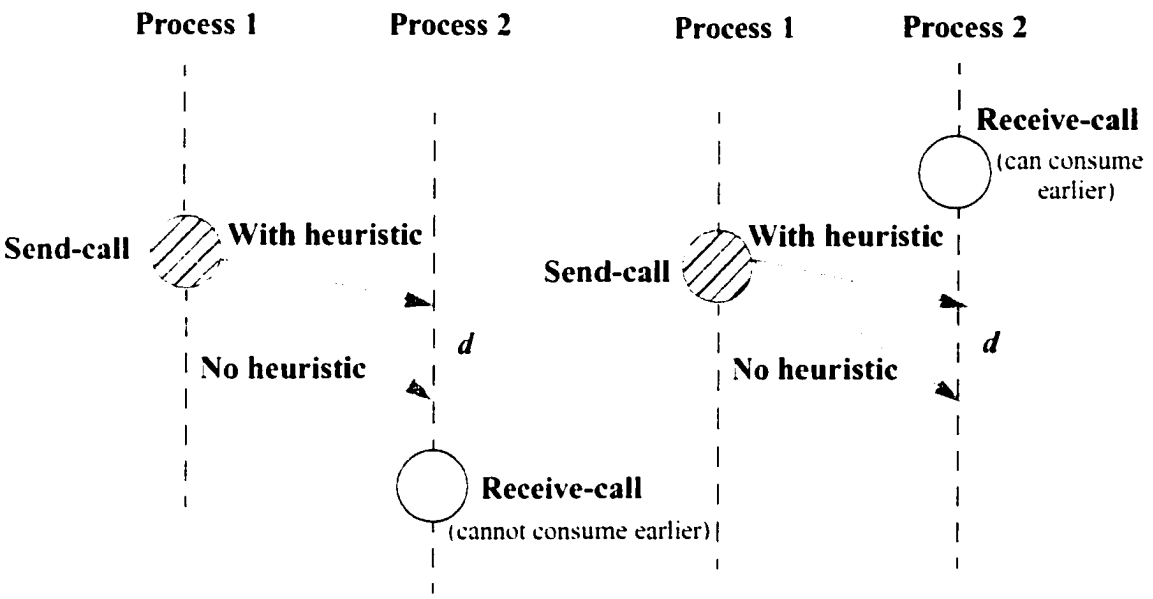


Figure 4.13: Heuristics effects on the receiving side

I present the average percentage of the times that the receive calls are issued earlier than their corresponding send calls for the CG, SP, and PSTSWM benchmarks in Figure 4.14. The results are true for d = 1, 5, 10, and 25 microseconds. LU and MG benchmarks are using *MPI_ANY_SOURCE* [92] for some of their receive calls and hence one cannot identify the sources of messages to compare with. What I have calculated is a lower bound of the improvement. A trace-driven simulator should be written for the exact calculation of the improvement.

4.5 Summary

In order to efficiently use the proposed predictors in Chapter 3 to hide the hardware latency of the reconfigurable interconnects, enough lead time should exist such that the reconfiguration of the interconnect be completed before the communication request arrives. For this, I presented the distribution of execution times of the computation phases of the parallel application benchmarks on an IBM SP2 machine. The results showed that most of the time, we are able to fully utilize these computation times for the concurrent reconfiguration of the interconnect when we know, in advance, the next target using one of the proposed high hit-ratio target prediction algorithms.

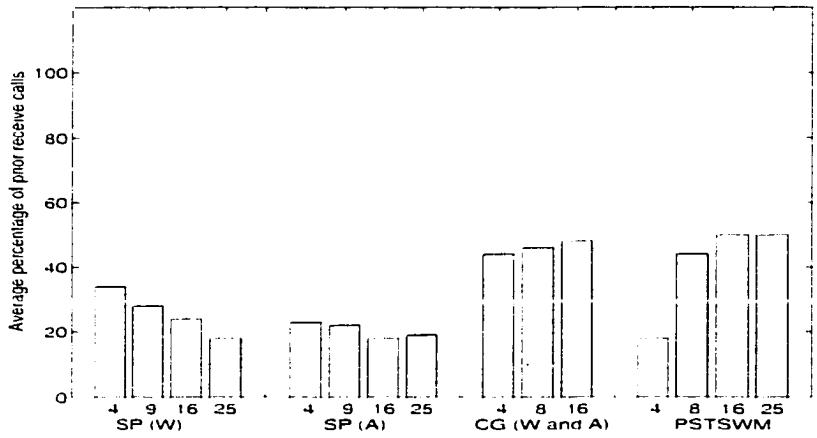


Figure 4.14: Average percentage of the times the receive calls are issued before the corresponding send calls

I also presented the performance enhancements of the best predictors, Better-cycle2, and Tag-bettercycle2, on the application benchmarks for the total reconfiguration time.

Finally, I considered the effects that using message destination predictors have on the receiving sides of communications. I showed that up to 50% of the time applications might benefit from the situations where they post early receive calls. However, A trace-driven simulator should be written for the calculation of the improvement.

I did not evaluate the application speedup when using the predictors on the applications. Rough estimates point to minimal speedup gains. This is because the parallel applications studied are very coarse-grained and hence the $\frac{communication}{computation}$ ratio is small. Table 4.2 shows the communication to computation ratios for the applications under different system sizes. These applications have been written to avoid a lot of communications between pair-wise nodes mostly because of the high communication latency in the current generation of parallel systems [43], and partly because of the algorithms, themselves. As shown in Table 4.2, the communication to computation ratio is increasing when the num-

ber of nodes increases. This means that we might have better speedup for these applications for larger system sizes. However, the inter-send computation times may decrease and thus reconfiguration delays cannot be hidden.

Table 4.2: Communication to computation ratio of the applications

	4 nodes	**8 nodes (9 for BT, SP)**	**16 nodes**	**25 nodes**
BT (W)	0.015	0.098	0.210	0.260
BT (A)	0.003	0.037	0.061	0.099
SP (W)	0.015	0.074	0.167	0.280
SP (A)	0.009	0.034	0.053	0.115
LU (W)	0.033	0.072	0.143	---
LU (A)	0.012	0.033	0.126	---
MG (W)	0.096	0.088	0.171	---
MG (A)	0.009	0.013	0.028	---
CG (W)	0.105	0.189	0.772	---
CG (A)	0.052	0.089	0.264	---
PSTSWM	0.055	0.114	0.277	0.5
QCDMPI	0.082	0.79	0.333	4.42

In this chapter, and Chapter 3 of this dissertation, I am particularly interested in the point-to-point communications in parallel applications. In Chapter 5, I discuss efficient collective communication algorithms for such reconfigurable interconnects.

Chapter 5

Collective Communications on a Reconfigurable Interconnection Network

Collective communications are basic patterns of interprocessor communication that are frequently used as building blocks in a variety of parallel algorithms. Proper implementation of collective communication algorithms is a key to the overall performance of parallel computers.

Free-space optical interconnection is used to fashion a reconfigurable network. Since network reconfiguration is expensive compared to message transmission in such networks, *latency hiding techniques* can be used to increase the performance of collective communications operations.

I present and analyze a broadcasting/multi-broadcasting algorithm [20] that utilizes latency hiding and reconfiguration in the network, $RON(k, N)$, to speed these operations. As the first contribution of this chapter, the analysis of the broadcasting algorithm includes a closed formulation that yields the termination time. Secondly, I contribute by proposing a *combined total exchange algorithm* based on a combination of the *direct* [109, 120], and *standard exchange* [71, 24] algorithms. This ensures a better termination time than what can be achieved by either of the two algorithms. Meanwhile, known algorithms for scattering and all-to-all broadcasting from the literature [40, 21] have been adapted to the network.

5.1 Introduction

Communication operations may be either *point-to-point*, as discussed so far, or *collective*, in which more than two processes participate. The study of classical algorithms brings up some generic communication patterns, collective communications, that appear very often in parallel algorithms [70, 76]. Collective communications are common basic

patterns of interprocessor communication that are frequently used as building blocks in a variety of parallel algorithms. Proper implementation of these basic communication operations on various parallel architectures is a key to the efficient execution of the parallel algorithms that use them, and hence, on the overall performance of the parallel computers.

Whether communication operations are programmed by the user (low-level routines), contained in a library such as MPI [92, 93], and *Parallel Virtual Machine* (PVM) [115], or generated by a compiler to translate high-level data parallel language such as *High Performance Fortran* (HPF) [85], their latency directly affects the total computation time of the parallel application. The growing interest in collective communication operations is evident by their inclusion in the MPI.

Collective communication operations can be used for data movement, process synchronization, or global operations, as shown in Figure 5.1. Data movement operations include, *broadcasting, multi-broadcasting, muticasting, scattering, gathering, multinode broadcasting,* and *total exchange*. In broadcasting, a node sends its unique message to all other nodes. Broadcasting is used in a variety of linear algebra algorithms [76], such as matrix-vector multiplication, matrix-matrix multiplication, LU-factorization, and Householder transformations. It is also used in database queries and transitive closure algorithms. In multi-broadcasting, a node broadcasts a number of messages to all other nodes. In multicasting, a special case of broadcasting, a node sends its unique message to a subset of all the other nodes. In scattering, a node sends a different message to all other nodes. It is basically used for distribution of data among the processors. Gathering is the exact reverse of scattering. That is, a node receives a different message from all other nodes. I will not discuss it here as a separate operation. In multinode broadcasting, all nodes send their unique messages to all other nodes. In total exchange, all nodes send their different messages to all other nodes. *Personalized communications* (scattering, gathering, and total exchange) are used, for instance, in transposing a matrix, and the conversion between different data structures, or in neural network simulations. It is worth mentioning that the terminology is not yet standard. For example, broadcasting is referred as *one-to-all,*

multinode broadcasting is referred as *all-to-all* or *gossiping*, scattering is referred as *personalized one-to-all*, and total exchange is referred as *multi-scattering* or *personalized all-to-all*.

Barrier synchronization, is a type of process synchronization. It defines a logical point in the control flow of an algorithm at which all members of the group must arrive before any of the processes in the subset is allowed to proceed further. Therefore, one of the processes plays the role of a barrier process. This process gathers messages of all other processes, and then broadcasts a message to them indicating that they can continue.

Global operations include *reduction*, and *scan*. In reduction, an operation such as *sum*, *max*, *min*, is applied across data items received from each member of the group. In an *N/1 reduction* operation, the resultant data resides at the root node. Therefore, it contains a gathering operation. In an *N/N reduction* operation, every node or process involved in the operation obtains a copy of the reduced data. Hence, it is a combination of gathering and broadcasting. In scan operation, given processes $p_0, p_1, \ldots, p_n$, and data items $d_0, d_1, \ldots, d_n$, an operation $\otimes$ is applied such that the result $d_0 \otimes d_1 \otimes \ldots \otimes d_i$ is available at the process p_i.

Collective operations have been usually proposed and designed for systems that support only point-to-point, or *unicast*, communication in hardware. In these environments, collective operations are implemented by sending multiple unicast messages. Such implementations are called *unicast-based*. An alternative approach is to provide more direct support for collective communication in the hardware. Two main approaches have been studied. The first approach uses a network other than the primary data network to implement collective communications [80]. In the second approach, the data network is enhanced to better support some collective communications. To improve collective communication performance and reduce software overhead, two such enhancements to routers have been proposed: *message replication* and *intermediate reception*. Message replication refers to the ability to duplicate incoming messages onto more than one outgoing chan-

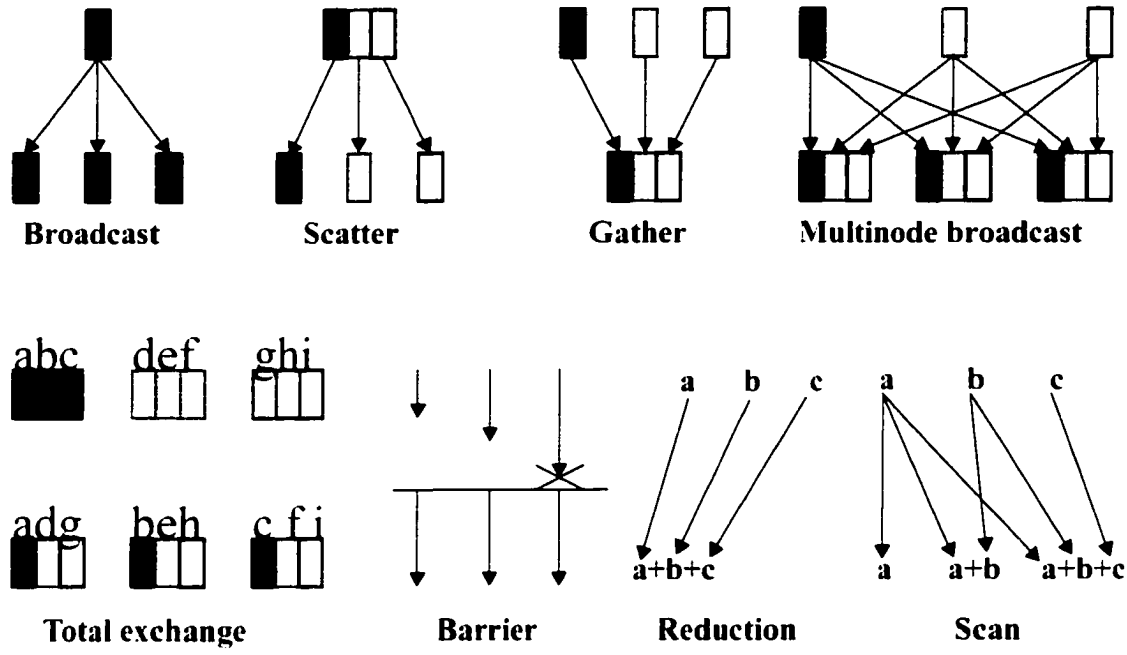


Figure 5.1: Some collective communication operations

nels, while intermediate reception is the ability to simultaneously deliver an incoming message to the local processor, and to an outgoing channel. Ni has proposed how scalable parallel computers should support efficient hardware multicast [99].

Numerous works have been reported on collective communications. Excellent surveys on collective communication algorithms in *store-and-forward* systems can be found in [53]. Another survey of broadcasting and multinode broadcasting in store-and-forward systems can be found in [61]. Dimakopoulos and Dimopoulos have shown how total exchange can be done in cayley graphs [41]. They have also presented collective communication algorithms on binary fat trees [42]. McKinley and his colleagues have surveyed collective communications on hypercubes, meshes, and tori in *wormhole-routed* networks [90]. Recently, Banikazemi and others, have proposed efficient broadcasting and multicasting algorithms using communication capabilities of heterogeneous networks of workstations [15]. In the context of optical interconnection networks, Berthome and Ferreira [20, 21] have presented broadcasting and multicasting algorithms for networks using *optical passive stars* (OPS). Comparative Study of one-to-many wavelength division multi-

plexing (WDM) lightwave interconnection networks, based on hypergragh theory [18], have been studied by Bourdin and his colleagues [25]. Gravenstreter and Melhem have presented some communication algorithms in *partitioned optical passive stars* (POPS) networks [59].

In this chapter, I present and analyze some collective communication algorithms for the reconfigurable network, *RON (k, N)*, defined in Chapter 3. In Section 5.2, I describe the communication modeling. I present and analyze broadcasting [20], and multi-broadcasting algorithms that utilize the reconfiguration capabilities of the network in Section 5.3. Later on in Section 5.5 and Section 5.6, known algorithms from literature for scattering and multinode broadcasting [20, 40] are adapted to the network. Then, I propose a new algorithm for total exchange operation, to be called *combined total exchange algorithm*, in Section 5.7. Finally, I summarize this chapter in Section 5.8.

5.2 Communication Modeling for Broadcasting/Multi-broadcasting

As discussed in Chapter 3, I use a modified Hockney's communication model [66]. I modify the Hockney's model into two models. In this section, I define the first model as used for hiding the reconfiguration delays in broadcasting and multi-broadcasting algorithms. In Section 5.4, I define the second model for other collective communication algorithms. The second model supports combining messages into a single message as used in scattering, multinode broadcasting, and total exchange algorithms, to be discussed later. Note that these algorithms are efficient but they do not hide the reconfiguration delay in the network.

The communication time to send a unit length message, l_m, from one node to another in the network is equal to $T = d + t_s + l_m\tau$. I incorporate both t_s and $l_m\tau$ into a single message delay $t_m = t_s + l_m\tau$. Thus, a unit length message transmission takes $T = d + t_m$. For the remaining of the discussion, and without loss of generality, I shall assume that $t_m = 1$ for a message of fixed length used in broadcasting/multi-broadcasting.

Culler and his colleagues have proposed the *LogP* model [33] which uses another terminology for communication modeling. LogP models sequences of point-to-point communications of short messages. L is the network hardware latency for one-word message transfer. O is the combined overhead in processing the message at the sender (o_s) and receiver (o_r). P is the number of processors. The gap, g, is the minimum time interval between two consecutive message transmission from a processor. Alexandrov and others have proposed the *LogGP* model [8] which incorporates long messages into the LogP model. The Gap per byte for long messages, G, is defined as the time per byte for a long message. Bar-Noy and Kipnis have developed the *postal model* [16], a special case of LogP model, where g is one. However, they don't consider the parameters o and G.

A node in LogP, LogGP, and postal models can send another message immediately g time after the previous message has been sent without waiting for the previous message to be delivered at the destination. These models are more suitable for the current state-of-the-art wormhole-routed networks where messages can be pipelined through the network. However, a node in my communication modeling can send another message only after its previous message has been delivered and its link has been reconfigured (if needed). This is because my model is a *telephone-like model* based on the circuit-switching technique which is suitable for reconfigurable optical networks.

The model that I have used is slightly different from the model that is offered in [20, 21, 40]. The difference lies in the fact that in the network, *RON (k, N)*, only the sender is allowed to reconfigure, and hence the delay penalties occur there. The receiver, in contrast to the models in [21, 40], and in [20] is entirely passive.

I use the notations B_m, MB_m, S_m, G_m, TE_m, for broadcasting time, multi-broadcasting time, scattering time, multinode broadcasting time, and total exchange time, respectively. I derive time complexities of collective communication algorithms in the network, *RON (k, N)*, under the model m, where $m \in \{F1, Fk\}$. *F1* stands for full-duplex, single-port communication. While, *Fk* stands for full-duplex, k-port communication.

5.3 Broadcasting and Multi-broadcasting

In this section, I shall concentrate in techniques that could effectively hide the reconfiguration delay d in the network. By reconfiguration latency hiding, I mean the process in which while some nodes are in their reconfiguration phase, other nodes are in their message transmission phase. Hence, the reconfiguration phase is overlapped with the message transmission phase which ultimately reduces the broadcasting and multi-broadcasting times.

5.3.1 Broadcasting

In broadcasting, a node, assuming node n_0 without loss of generality, sends its unique message to all other nodes. I assume an unbounded number of available wavelengths for the system. As noted earlier in Chapter 3, techniques such as spread-spectrum can be used in case of limited number of available wavelengths. In the following, I first discuss the broadcasting algorithm under k-port modeling, and then present the results for the single-port modeling.

***K-port*:** The naive algorithm is to let the broadcasting node n_0 inform k new nodes at a step. Clearly, it takes $(d+1)\left\lceil \frac{N-1}{k} \right\rceil$ time units. In a more efficient algorithm, BI_{Fk}, node n_0 sends the message to k other nodes and these k nodes, upon receiving the message, send it to k other nodes each, which are distinct from the nodes that have received the message thus far. Continuing this way, the algorithm will terminate after $\lceil \log_k(N(k-1)+1) \rceil - 1$ steps, while in terms of elapsed time, the algorithm will take $(d+1)(\lceil \log_k(N(k-1)+1) \rceil - 1)$ time units.

Obviously, one can do better than this if one allows the nodes that have already been informed, to re-send the same message to a different group of nodes. Thus, starting with node n_0, it sends the message to k nodes. At the end of this step, $k+1$ nodes possess the message which they now send to k other nodes each. Proceeding this way, this algorithm, $B2_{Fk}$, will terminate after $\lceil \log_{k-1} N \rceil$ steps and will require $(d+1)\lceil \log_{k-1} N \rceil$ time units.

The above algorithms, $B1_{Fk}$ and $B2_{Fk}$, are logarithmic in time, but they suffer because of the large reconfiguration delay, d, that each node incurs. I am interested in devising algorithms that will overcome the existence of the large reconfiguration delays by essentially hiding it. The algorithm $B1_{Fk}$ can be improved if the configuration of all the links forming the tree proceed in parallel. Hence, in this new algorithm, $B3_{Fk}$, the broadcasting message would reach the leaves of the tree in time $d + \lceil \log_k(N(k-1)+1) \rceil - 1$.

The algorithm $B2_{Fk}$ can be improved if the configurations can take place concurrent to the message transmissions. I adopt a greedy algorithm, $B4_{Fk}$, where a node reconfigures its links to reach k children which lead to a *pre-configured* tree of an appropriate $O(\log_k N)$ depth. As soon as the broadcasting node has finished sending its message, it reconfigures its links to reach another predefined tree. It is understood that while this node is reconfiguring (this takes d steps time units), nodes that have already been configured and are in possession of the message send it to k neighbors each. This process repeats at each node every time it sends the message. Potentially, the message, starting at node n_x will reach $1 + k + k^2 + \ldots + k^d = \frac{k^{d+1} - 1}{k - 1}$ nodes before node n_x be able to reconfigure. Figure 5.2 depicts the $B4_{Fk}$ algorithm for a 2-port network with 41 nodes and a reconfiguration delay of 1. This algorithm is optimal since a node after sending/receiving the message immediately reconfigures to send the message to a new node. This algorithm is similar to the broadcasting algorithm by Berthome and Ferreira for their loosely-coupled *optically reconfigurable parallel computer, ORPC (k)*, using optical passive stars (OPS) [20].

It is clear that either this broadcasting network is a dedicated network, or there exists a global control where nodes understand that a broadcasting is going to take place and hence they reconfigure their links correspondingly. In the latter case, an early reconfiguration delay should be added to the broadcasting time.

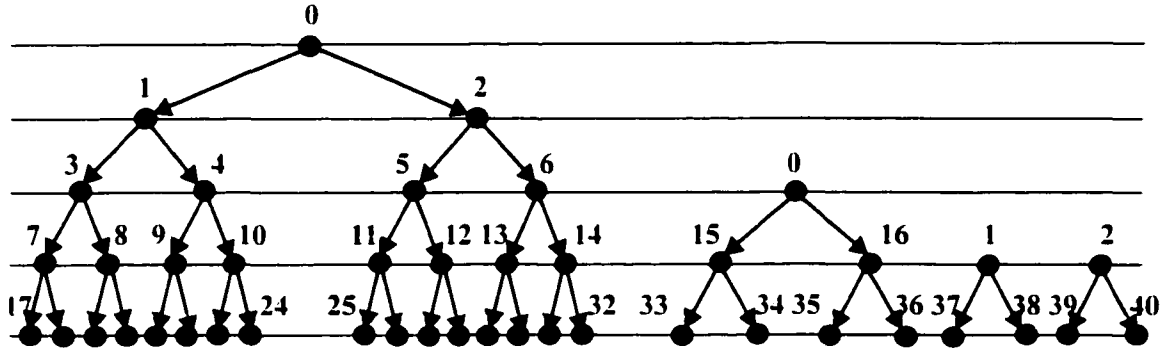


Figure 5.2: Latency hiding broadcasting algorithm for *RON (k, N)*, $N = 41$, $k = 2$, $d = 1$

5.3.1.1 Analysis of the Greedy Algorithm

Before presenting the analysis of the greedy algorithm, it is worth noting that it can be shown that the total number of nodes, $N(S)$, informed up to step S follows the recurrence relations:

$$N(S) = \begin{cases} 1 & for \quad S = 0 \\ kN(S-1) + 1 & for \; S \le d+1 \\ kN(S-1) + N(S-d-1) & for \; S > d+1 \end{cases} \tag{5.1}$$

It can also be shown that the number of nodes, $r(S)$, that receive the message at each step, S, follows the recurrence relations:

$$r(S) = \begin{cases} kN(S-1) & for \; S \le d+1 \\ kN(S-1) + N(S-d-1) & for \; S > d+1 \end{cases} \tag{5.2}$$

These recurrence relations are a kind of generalization of the Fibonacci functions defined by Bar-Noy and Kipnis for the postal model [16], and are similar to the recurrence relations of the broadcasting algorithms by Berthome and Ferreira [20]. The above relations and those in [16, 20] cannot be solved for a general d. They should be computed step

by step or be given in a table in order to find the termination time of the algorithms. However, as will be shown in the following, the analysis of the broadcasting algorithm includes a closed formulation that yields the termination time.

I present another approach to find a closed formula for the total number of nodes, $N(S)$, up to the step S. The problem I shall endeavor to solve is to find the time required for the greedy algorithm to complete. I shall approach the analysis constructively, that is, I shall find the number of nodes that will be informed as time progresses, and I shall stop when all nodes N have been informed.

Denote by S the termination time (in units of t_m). Then starting from an arbitrary node n_0, the nodes that will be informed and assuming no reconfiguration, belong to a k-ary tree rooted at node n_0 and of depth S. There are $N_1 = \frac{k^{S-1} - 1}{k - 1}$ nodes in this tree, and I shall reference them as belonging to the first generation. Each of the nodes in this tree, once it has broadcast the message to its own children, will reconfigure and will become the root of a new tree over which a new wave of broadcasting will commence and proceed concurrently with the broadcasting in the first generation tree. This can only happen if $S \geq d + 2$ ensuring that the first node to be reconfigured (node n_0) will have enough time to reconfigure and broadcast to its k children.

I shall refer to the nodes belonging to the trees rooted at nodes which were included in the first generation tree and reconfigured, as the second generation nodes. Thus, node n_0 can send its message again at time $d + 1$ after its router has been reconfigured to connect to a set of k new nodes. By sending this new message, n_0 actually embeds a new k-ary tree at depth $d + 1$. The next k nodes at depth 1 of the first generation of trees embed k different k-ary trees at depth $d + 2$. Using this concept, the k^{S-d-2} nodes at depth $S - d - 2$ of the first generation embed the last k^{S-d-2} different trees at depth $S - 1$ in the second generation. Figure 5.3 depicts the embedding of the first two generations of the nodes.

Denote by N_2 the total number of new nodes in the second generation, and by M_i the total number of new nodes in the trees of the second generation rooted at depth i.

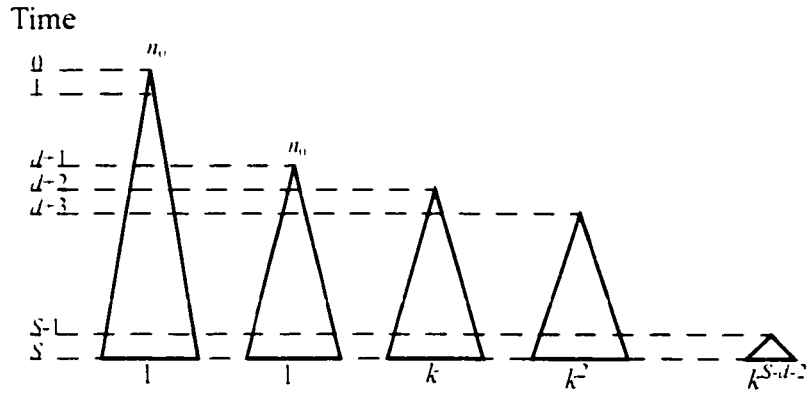


Figure 5.3: First and second generation trees. The numbers underneath each tree denote the number of trees having the same height. These trees are rooted at nodes that were at the same level in the first generation tree.

Therefore,

$$M_{d-1} = k + k^2 + \ldots + k^{S-(d-1)} = \frac{k^{S-d} - k}{k-1} \tag{5.3}$$

$$M_{d-2} = k(k + k^2 + \ldots + k^{S-(d-2)}) = \frac{k^{S-d} - k^2}{k-1} \tag{5.4}$$

$$M_{d-3} = k^2(k + k^2 + \ldots + k^{S-(d-3)}) = \frac{k^{S-d} - k^3}{k-1} \tag{5.5}$$

This continues until depth S - 1 where:

$$M_{S-1} = k^{S-d-2}(k) = \frac{k^{S-d} - k^{S-d-1}}{k-1} \tag{5.6}$$

Therefore, the total number of new nodes in the second generation, N_2, will be:

$$N_2 = \sum_{i=d-1}^{S-1} M_i = \sum_{j=1}^{S-d-1} \frac{k^{S-d} - k^j}{k-1} \tag{5.7}$$

The process of reconfiguring the optical interconnects continues by the nodes as soon as they have broadcast the message to their children. Each generation of trees embeds a new generation that commences at depth $d + 1$ from its parent generation. It is clear that the total number of generations is $\left\lceil \frac{S}{d+1} \right\rceil$.

Let us now count the total number of nodes N_3 in the third generation. The first tree of the third generation is embedded at depth $2(d + 1)$ by n_0. I begin with those trees of this generation which are embedded by the nodes of the first tree in the second generation. Let Q_i^1 denotes the total number of nodes in these trees rooted at depth i.

Therefore,

$$Q^1_{2(d+1)} = k + k^2 + \ldots + k^{S-2(d+1)} = \frac{k^{S-2d-1} - k}{k-1} \tag{5.8}$$

$$Q^1_{2(d+1)+1} = k(k + k^2 + \ldots + k^{S-2(d+1)-1}) = \frac{k^{S-2d-1} - k^2}{k-1} \tag{5.9}$$

$$Q^1_{2(d+1)-2} = k^2(k + k^2 + \ldots + k^{S-2(d+1)-2}) = \frac{k^{S-2d-1} - k^3}{k-1} \tag{5.10}$$

This continues until the depth S -1 where:

$$Q^1_{S-1} = k^{S-2d-3}(k) = \frac{k^{S-2d-1} - k^{S-2d-2}}{k-1} \tag{5.11}$$

Now, consider trees embedded in the third generation by the nodes of the next k trees at depth S - d - 2 in the second generation, and let Q_i^2 denotes the total number of nodes in these trees rooted at depth i. Therefore,

$$Q^2_{2(d-1)-1} = k(k + k^2 + \ldots + k^{S-2(d-1)-1}) = \frac{k^{S-2d-1} - k^2}{k-1} \tag{5.12}$$

$$Q^2_{2(d-1)-2} = k(k(k + k^2 + \ldots + k^{S-2(d-1)-2})) = \frac{k^{S-2d-1} - k^3}{k-1} \tag{5.13}$$

$$Q^2_{2(d-1)-3} = k(k^2(k + k^2 + \ldots + k^{S-2(d-1)-3})) = \frac{k^{S-2d-1} - k^4}{k-1} \tag{5.14}$$

This continues until the depth S - 1 where:

$$Q^2_{S-1} = k(k^{S-2d-4}(k)) = \frac{k^{S-2d-1} - k^{S-2d-2}}{k-1} \tag{5.15}$$

I continue with the trees embedded in the third generation by the nodes of the next k^2 trees of depth S - d - 3 in the second generation, and let Q_i^3 denotes the total number of nodes in these trees rooted at depth i. Therefore,

$$Q^3_{2(d-1)-2} = k^2(k + k^2 + \ldots + k^{S-2(d-1)-2}) = \frac{k^{S-2d-1} - k^3}{k-1} \tag{5.16}$$

$$Q^3_{2(d-1)-3} = k^2(k(k + k^2 + \ldots + k^{S-2(d-1)-3})) = \frac{k^{S-2d-1} - k^4}{k-1} \tag{5.17}$$

$$Q^3_{2(d-1)-4} = k^2(k^2(k + k^2 + \ldots + k^{S-2(d-1)-4})) = \frac{k^{S-2d-1} - k^5}{k-1} \tag{5.18}$$

This continues until the depth S - 1 where:

$$Q^{3}_{S-1} = k^{2}(k^{S-2d-5}(k)) = \frac{k^{S-2d-1} - k^{S-2d-2}}{k-1} \tag{5.19}$$

The process of generating trees in the third generation continues up to the trees embedded at depth S - 1, by the nodes of the trees in the second generation, rooted at depth S - d - 2. Let Q^{S-2d-2}_{S-1} denotes the total number of nodes in these trees. Therefore,

$$Q^{S-2d-2}_{S-1} = k^{S-2d-3}(k) = \frac{k^{S-2d-1} - k^{S-2d-2}}{k-1} \tag{5.20}$$

Now, I am at the stage to sum the number of nodes at each depth in the third generation. Let Q_i denotes the total number of nodes of the trees in the third generation rooted at depth i.

Therefore,

$$Q_{2(d-1)} = \frac{k^{S-2d-1} - k}{k-1} \tag{5.21}$$

$$Q_{2(d-1)+1} = 2\frac{k^{S-2d-1} - k^{2}}{k-1} \tag{5.22}$$

$$Q_{2(d-1)+2} = 3\frac{k^{S-2d-1} - k^{3}}{k-1}. \ldots \tag{5.23}$$

$$Q_{S-1} = (S-2d-2)\frac{k^{S-2d-1} - k^{S-2d-2}}{k-1} \tag{5.24}$$

Hence, the total number of the new nodes in the third generation, N_3, will be:

$$N_3 = \sum_{i=2(d+1)}^{S-1} Q_i = \sum_{j=1}^{S-2d-2} j\left(\frac{k^{S-2d-1}-k^j}{k-1}\right) \tag{5.25}$$

In a similar manner, I can compute the number of nodes for the fourth and fifth generations as:

$$N_4 = \sum_{j=1}^{S-3d-3} \frac{j(j+1)}{2!}\left(\frac{k^{S-3d-2}-k^j}{k-1}\right) \tag{5.26}$$

$$N_5 = \sum_{j=1}^{S-4d-4} \frac{j(j+1)(j+2)}{2!}\left(\frac{k^{S-4d-3}-k^j}{k-1}\right). \tag{5.27}$$

This process implies lemma 1.

Lemma 1 The number of new nodes in generation $i+1$, $i \geq 1$ can be found as:

$$N_{i+1} = \sum_{j=1}^{S-i(d+1)} \binom{j+i-2}{i-1}\left(\frac{k^{S-i(d+1)-1}-k^j}{k-1}\right) \tag{5.28}$$

Proof. I give a combinatorial argument for its validity. Assume a tree belonging to generation i -1 and rooted at depth $(i-1)(d+1)$. This tree will produce a number of trees belonging to generation i and rooted at depth $i(d+1)$. The term $\frac{k^{S-i(d+1)-1}-k^j}{k-1}$ represents the number of new nodes in the first tree of generation i rooted at depth $i(d+1)$. Subsequent trees in this generation, have a decreasing (by one) number of levels, but since they were produced by nodes that are at lower levels in the parent generation, their numbers grow with the power of k. Therefore, the number of nodes within all the trees at each level, remains the same and equal to $\frac{k^{S-i(d+1)-1}-k^j}{k-1}$.

I have however accounted for the number of trees produced by a single tree in a parent generation. There is more than one tree of identical depth in the parent generation, and the multiplicative term $\binom{j+i-2}{i-1}$ accounts for this number based on the Pascal's triangle [27]. ■

The total number of nodes in all generations, $N(S)$, informed up to step S, is equal to:

$$N(S) = N_1 + N_2 + \ldots + N_{\left\lceil \frac{S}{d+1} \right\rceil - 1}, \text{ or} \tag{5.29}$$

$$N(S) = \frac{k^{S+1} - 1}{k-1} + \sum_{i=1}^{\left\lceil \frac{S}{d+1} \right\rceil - 1} \sum_{j=1}^{S-i(d+1)} \binom{j+i-2}{i-1} \frac{k^{S-i(d+1)+1} - k^j}{k-1} \tag{5.30}$$

Note that Equation 5.30 is a closed formula and easier to compute (less computation and memory requirements) than the recurrence Equation 5.1, and Equation 5.2. To determine the termination time S one has to solve Equation 5.30 for S. This equation can be solved numerically. Table 5.1 and Table 5.2 provide a comparison of some numerical examples for the broadcasting time under different broadcasting algorithms, $B1_{Fk}$, $B2_{Fk}$, $B3_{Fk}$, $B4_{Fk}$, and for the best case $\log_{k+1} N$ when there is no reconfiguration delay (i.e. $d = 0$), for a particular number of nodes, N, reconfiguration delay, d, and port modeling, k. It is quite clear that the latency hiding algorithm, $B4_{Fk}$, performs better than the other algorithms.

Table 5.1: Broadcasting time, $k = 2$, $d = 1$

N	$B1_{Fk}$	$B2_{Fk}$	$B3_{Fk}$	$B4_{Fk}$	$\lceil \log_{k+1} N \rceil$
99	12	10	7	5	5
1393	20	14	11	8	7
19601	28	18	15	11	9
114243	32	22	17	13	11

Table 5.2: Broadcasting time, $k = 4$, $d = 3$

N	$B1_{Fk}$	$B2_{Fk}$	$B3_{Fk}$	$B4_F$	$\lceil \log_{k+1} N \rceil$
85	12	12	6	3	3
1369	24	20	9	5	5
22703	32	28	11	7	7
88633	36	32	12	8	8

***Single-port*:** In this case, a node can only use one of its links. Therefore, instead of k-ary trees, linear arrays are embedded. Hence, using the same concept as in the k-port modeling, the total number of nodes for generations 1, 2, 3, 4 are:

$$N_1 = S + 1 \tag{5.31}$$

$$N_2 = \sum_{j=1}^{S-d+1} S - d - j \tag{5.32}$$

$$N_3 = \sum_{j=1}^{S-2(d+1)} j(S - 2d - 1 - j) \tag{5.33}$$

$$N_4 = \sum_{j=1}^{S-3(d+1)} \frac{j(j+1)}{2!}(S - 3d - 2 - j). \tag{5.34}$$

If I continue in a similar manner to the k-port modeling, then the total number of nodes in all generations, $N(S)$, would be:

$$N(S) = S + 1 + \sum_{i=1}^{\left\lceil \frac{S}{d+1} \right\rceil - 1} \sum_{j=1}^{S-i(d+1)} \binom{j+i-2}{i-1}(S - i(d+1) + 1 - j) \tag{5.35}$$

Table 5.3 provides a comparison of some numerical examples for the broadcasting time of the latency hiding algorithm, B_{FI}, of the *spanning binomial algorithm* [114], $(d+1)\lceil \log_2 N \rceil$, and for the best case $\log_2 N$ when there is no reconfiguration delay (i.e. $d = 0$), for a particular number of nodes, N, and reconfiguration delay, d. It is clear that the algorithm, B_{FI}, performs better than the spanning binomial algorithm.

Table 5.3: Broadcasting time, $d = 3$

N	$(d+1)\lceil \log_2 N \rceil$	B_{FI}	$\lceil \log_2 N \rceil$
69	28	12	7
1252	44	21	11
8657	56	27	14
82629	68	34	17

5.3.1.2 Grouping schema

The total number of nodes, $N(S)$, informed up to step S is given as Equation 5.1. Meanwhile, the number of nodes, $r(S)$, that receive the message at each step S is defined as Equation 5.2. The nodes are divided into two groups. The group that has already received the message and the one that has not. The nodes that know the message at any give step can be grouped into those nodes that have already received the message and those that receive at this time step. The nodes that receive at each step, is proportional (k times) to the number of nodes that have received the message at the last step and those that have sent the message $d + 1$ steps ago.

The same grouping schema as in [20] can be used to find the set of nodes that transmit the message, and the set of the nodes that receive the message at any given step. The set $T(S)$ consists of the nodes transmitting the message at step S. While, the set $R(S)$ consists of the nodes that receive the message at step S. These two sets can be found by Equation 5.36. Note that the same grouping schema can be applied to the multi-broadcasting case to be discussed in the next section.

$$\begin{cases} T(0) = 0 \\ R(S) = \{N(S-1)+1, \ldots, N(S)\} \\ T(S) = T(S-d-1) \cup R(S-1) \end{cases} \tag{5.36}$$

5.3.2 Multi-broadcasting

If there are M messages to be broadcast by a node to all other nodes, the simplest algorithm is to use the above latency hiding broadcasting algorithms ($B4_{Fk}$ or B_{F1}) M times in sequence. This algorithm, denote it by $MB1$, gives an upper bound for multi-broadcasting and takes $M \times (d + B4_{Fk})$, and $M \times (d + B_{F1})$ time units under k-port and single-port modeling, respectively. A lower bound for multi-broadcasting, $M - 1 + MB_m$ (MB_m is the broadcasting time for an optimal algorithm), can be achieved by pipelining the messages through the network. That is, node n_0 sends its M messages in sequence in an optimal broadcasting algorithm.

One may think of another algorithm, $MB2_{FK}$, where the first message embeds a broadcasting tree (first generation tree) rooted at node n_0; Each of the subsequent messages use this embedded tree to broadcast thus bypassing the reconfiguration costs that the first message incurred. Hence, the first message will incur a delay of $d + (\lceil \log_k N(k-1)+1 \rceil - 1)$ time units to broadcast over all N nodes and to embed the broadcast tree, while the second and subsequent messages would only incur a broadcast delay of $\lceil \log_k N(k-1)+1 \rceil - 1$ each. Therefore, the total cost is

$$MB2_{FK} = d + M(\lceil \log_k N(k-1)+1 \rceil - 1) \tag{5.37}$$

Table 5.4 compares the two algorithms $MB1_{FK}$ and $MB2_{FK}$. Note that an optimal algorithm for multi-broadcasting is to be devised such that messages are pipelined through the embedded trees using the latency hiding broadcasting algorithms ($B4_{Fk}$ or B_{F1}).

Table 5.4: Multi-broadcasting time, $k = 4$, $d = 3$, $M = 10$

N	$MB1_{Fk}$	$MB2_{Fk}$
85	60	33
1369	80	63
22703	100	83
88633	110	93

5.4 Communication Modeling for other Collective Communications

In this section, I define the second communication modeling used for scattering, multinode broadcasting, and total exchange algorithms. This model supports combining messages into a single larger message as used in these algorithms. Note that the algorithms for scattering, multinode broadcasting, and total exchange are quite efficient but they do not hide the reconfiguration delay in the network.

As stated in Section 5.2, the communication time to send a unit length message from one node to another in the network is equal to $T = d + t_s + l_m\tau$. Without loss of generality, I normalize the time T with respect to $l_m\tau$. Thus, a representative length message transmission takes $T = d + t_s + 1$. The communication time to send an M representative length message from one node to another would be $T = d + t_s + M$. Note that, sending a combined message (that is a larger message) does not affect the start-up time, t_s, and the reconfiguration delay, d. For simplicity, I incorporate both t_s and d into a single message delay $\tilde{d} = d + t_s$.

5.5 Scattering

The scattering operation, is used basically to distribute data to the nodes of a parallel computer. The easiest algorithm for the scattering operation is based on the *sequential tree* [101]. In this case, the source node sends its different messages to each of the other nodes

sequentially, as shown in Figure 5.4 for single-port modeling. As the source of communication is the same for the whole scattering operation, this node should reconfigure its links after each step. Therefore, the scattering time, SI_{F1}, is $(N-1)(\tilde{d}+1)$ time units.

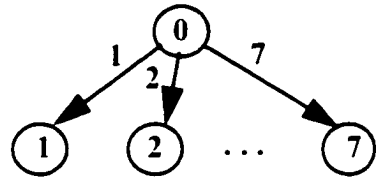


Figure 5.4: Sequential tree algorithm

The *spanning binomial tree algorithm* [91] used for broadcasting/multicasting operations can also be used for scattering operation. In this algorithm, the number of informed nodes doubles at each step, and each node stores its own message and forwards the rest of the messages it received, if necessary, to its children. As illustrated in Figure 5.5, the source node sends its messages for the upper half of the nodes to the node 4. In the second step, nodes 0 and 4 are responsible for sending messages to the nodes in their halves. That is, to the node 2 (messages for nodes 2, and 3), and node 6 (messages for nodes 6, and 7), respectively. In the third step, all nodes send the remaining messages to the remaining nodes. These three steps (actually $\log_2 N$ steps) takes each $(\tilde{d}+4)$, $(\tilde{d}+2)$, and $(\tilde{d}+1)$ time units, respectively. Generally, this algorithm has a scattering time:

$$S2_{F1} = N - 1 + \tilde{d}\log_2 N \tag{5.38}$$

Note that I have neglected the data permutation time at each node. It should be noted that he spanning binomial algorithm has a much better termination time than the sequential algorithm for the *RON (k, N)* (except for the trivial case, $N = 2$, where they have the same termination time).

***k-port*:** The sequential tree algorithm can be extended for k-port modeling. That is, at each step the source node sends its k different messages to k other different nodes. Therefore, $S1_{Fk} = (\tilde{d}+1)\left(\frac{N-1}{k}\right)$.

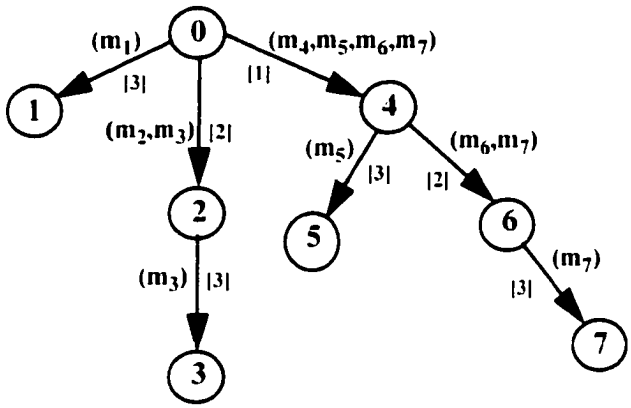


Figure 5.5: Spanning binomial tree algorithm

Desprez and his colleagues have extended the spanning binomial algorithm for the k-port modeling [40]. In this algorithm, the scattering node n_0, sends k messages of $\frac{N}{k+1}$ length each, to its k children. Therefore, there are $(k+1)$ nodes having $\frac{N}{k+1}$ different messages. These nodes, at step 2, communicate each with their k children and send one $(k+1)$-th of their initial message to each one. This process continues and all nodes are informed after $\log_{k+1}N$ communication steps. Thus the scattering time is equal to

$$S2_{FK} = \sum_{i=1}^{\log_{k+1}N} \left(\tilde{d} + \frac{N}{(k+1)^i} \right) = \frac{N-1}{k} + \tilde{d}\log_{k+1}N \qquad (5.39)$$

5.6 Multinode Broadcasting

In multinode broadcasting, also called gossiping [53], all nodes send their unique messages to all other nodes, and this is basically used in parallel algorithms when all nodes need to exchange their data. The simplest algorithm for multinode broadcasting is to use the latency hiding broadcasting algorithm N times, one for each node. Another algorithm is to consider the multinode broadcasting as a degenerate case of total exchange, to be discussed in the next section. However, better algorithms exist.

Single-port: In the direct algorithms [109, 120], at any step i, a node p sends its message to node $(p + i)$ *mod* N. Clearly, the cost of this algorithm, $G1_{F1}$, is $(N-1)(\tilde{d} + 1)$.

One may use a better algorithm, just like the *standard exchange* algorithms for the total exchange operation [71, 24], where during each step, the complete network is recursively divided into halves, and messages are exchanged across new divisions at each step. This algorithm combines messages into larger messages to be transmitted as a single unit. Actually, each node sends its message along with the other messages it received at the previous steps. Hence, the multinode broadcasting has $\log_2 N$ steps, and a cost of

$$G2_{F1} = N - 1 + \tilde{d}\log_2 N \tag{5.40}$$

Figure 5.6 shows pairwise communications and the length of messages at each step for multinode broadcasting on an 8 node message-passing multicomputer. Unfortunately, latency hiding cannot improve this cost.

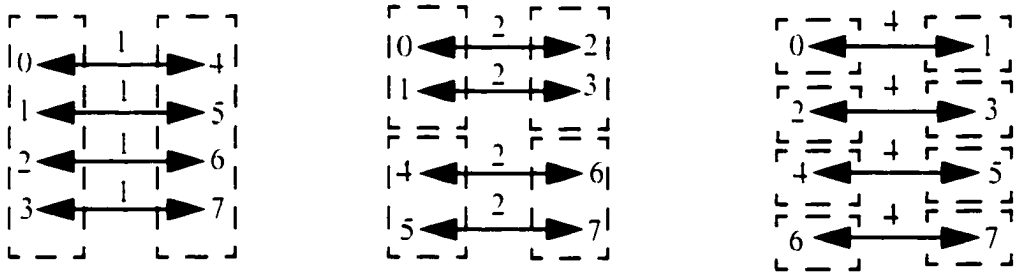


Figure 5.6: Multinode broadcasting on an 8-node *RON (k, N)* under single-port modeling

k-port: A simple algorithm is based on the extension of the direct algorithm for k-port modeling. That is, at step i, node p sends its message to the nodes $(p + (i - 1)k + 1)$ *mod* N, $(p + (i - 1)k + 2)$ *mod* N, ..., $(p + ik)$ *mod* N. This algorithm has a cost of: $G1_{Fk} = (\tilde{d} + 1)\left(\frac{N-1}{k}\right)$.

Desprez and his colleagues [40] extended the $G2_{FI}$ algorithm for k-port modeling by letting the nodes combine the messages to reduce the effect of reconfiguration delay. Figure 5.7 illustrates this algorithm when N =9 and k = 2. I divide the nodes into $\frac{N}{k+1}$ groups of $(k+1)$ nodes each. Nodes are grouped as $(0, 1, \ldots, k)$, $(k+1, k+2, \ldots, 2(k+1)-1)$, $\ldots$, $(N-(k+1), N-(k+1)+1, \ldots, N-1)$. At step 1, all nodes within a group exchange their messages. At the end of this step, each node has $(k+1)$ messages. At step 2, node p exchanges all its messages with nodes $(p+(k+1)) \bmod N$, $(p+2(k+1)) \bmod N$, $\ldots$, $(p+k(k+1)) \bmod N$. At the end of this step, each node has $(k+1)^2$ messages. Let $S = \log_{k+1} N$. This process continues to step s, where node p exchanges its messages with nodes $(p+(k+1)^{S-1}) \bmod N$, $(p+2(k+1)^{S-1}) \bmod N$, $\ldots$, $(p+k(k+1)^{S-1}) \bmod N$. It is clear that at each step i of this algorithm, each node sends $(k+1)^{i-1}$ messages to k other nodes. Hence, this algorithm has a multinode broadcasting time:

$$G2_{FK} = \sum_{i=1}^{\log_{k+1} N} (\tilde{d} + (k+1)^{i-1}) = \frac{N-1}{k} + \tilde{d}\log_{k+1} N \qquad (5.41)$$

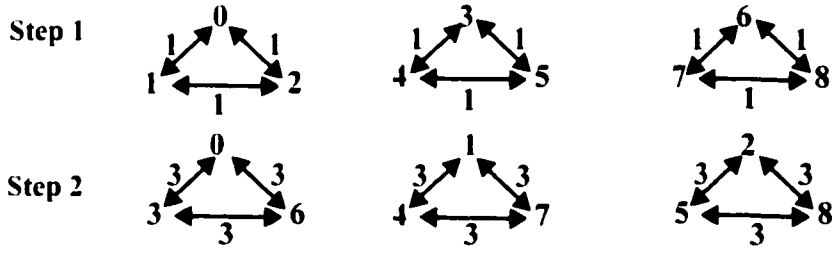


Figure 5.7: Multinode broadcasting on an 9-node *RON (k, N)* under 2-port modeling

5.7 Total Exchange

In total exchange, all nodes send their different messages to all other nodes. A naive algorithm for total exchange is to perform a scattering operation N times in sequence. However, better algorithms exist.

Single-port: In the direct algorithms [109, 120], at any step i, a node p sends the message to destined node $(p + i) \bmod N$. Clearly, the cost of this algorithm, $TE1_{FI}$, is equal to $(N - 1)(\tilde{d} + 1)$.

One may also use the standard exchange algorithm for total exchange similar to the ones used in hypercubes, and meshes [71, 24], where during each step, the complete network is recursively divided into halves, and messages are exchanged across new divisions at each step. Nodes combine messages into larger messages to be transmitted as a single unit. Consider this algorithm for an 8-node multicomputer, as shown in Figure 5.8. There are $N/2$ messages to be sent by each node at any step in this algorithm. I only describe this for node 0. Node 0 sends all its messages for the nodes at the upper half (that is, nodes 4, 5, 6, and 7) to node 4 at step 1. At the same time, it receives the messages for its half from node 4. At the second step, node 0 sends its message, along with the messages from node 4, destined to nodes 2 and 3, to node 2. At the same time, it receives the messages from the nodes 2, and 6 for itself and node 1. At the third step (actually, $\log_2 N$ steps), node 0 sends its message along with the other messages from nodes 2, 4, and 6 to node 1. It is clear that at the end of this step all nodes have exchanged all their messages. Thus, this algorithm, $TE2_{FI}$, has a cost of $\left(\tilde{d} + \frac{N}{2}\right)\log_2 N$.

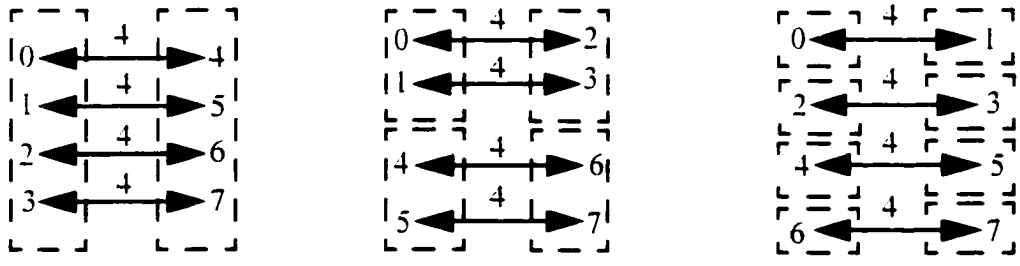


Figure 5.8: Total exchange on an 8-node *RON (k, N)* under single-port modeling

Which algorithm, $TE1_{F1}$ or $TE2_{F1}$, is faster depends on the number of nodes N, and the term, $\tilde{d}$. I propose another algorithm, called *combined total exchange algorithm*, $TE3_{F1}$, which is a combination of these two algorithms.

I begin this algorithm by doing some of (or even none of) the steps involved in the standard total exchange algorithm, and then continue with the direct algorithm. That is, I divide the nodes in the complete network in half and do the steps involved in the standard total exchange algorithm up to a point(s) that there is no gain in continuing to do so. From that step(s) on, the direct algorithm is used for all the nodes in each of the created sub-groups at the same time. Actually, the goal is to find the number of steps, or a bound for the number of steps, before switching to the direct algorithm such that the time associated with this algorithm is less than (or at least equal to) the other two (direct, and standard exchange) algorithms.

Let me explain this algorithm with $i = 1$ (number of doing the standard exchange algorithm) for the example shown in Figure 5.8. At the step 1, the nodes in the complete network are divided in halves. Each node exchanges 4 messages with its corresponding node at the other half. This takes $\tilde{d} + 4$, and at this point, each of the network halves contain messages destined to the half itself. As a matter of fact, each node now has two messages for each of the nodes in its half. These messages can be distributed to their destinations using a direct algorithm. There are 4 nodes in each half and 2 messages to be exchanged at a time for a cost of $(4 - 1)(\tilde{d} + 2) = 3\tilde{d} + 6$. Hence, this algorithm has a total cost of $4\tilde{d} + 10$.

Lemma 2 The *combined total exchange algorithm* under single-port modeling on *RON (K, N)* has a cost of

$$TE3_{F1} = i\left(\tilde{d} + \frac{N}{2}\right) + \left(\frac{N}{2^i} - 1\right)(\tilde{d} + 2^i) \tag{5.42}$$

where i is the number of steps to do the standard exchange algorithm before switching to the direct algorithm.

Proof. In the combined total exchange algorithm, each time a standard exchange algorithm step is done a cost of $\tilde{d} + \frac{N}{2}$ is added. This brings up the term $i\left(\tilde{d} + \frac{N}{2}\right)$. The first part of the second term, $\left(\frac{N}{2^i} - 1\right)$, is for the number of nodes in the groups doing the direct algorithms simultaneously. The second part, $(\tilde{d} + 2^i)$, stands for the delay associated with the transfer of messages which is doubled at each steps. ■

It is clear that this algorithm is exactly the same as the direct algorithm when $i = 0$, and the standard exchange algorithm when $i = \log_2 N$.

k-port: The direct algorithm for the k-port modeling requires node p at step i to send its message to the nodes $(p + (i - 1)k + 1) \bmod N$, $(p + (i - 1)k + 2) \bmod N$, ..., $(p + ik) \bmod N$. This algorithm has a cost of, $TE1_{Fk} = (\tilde{d} + 1)\left(\frac{N-1}{k}\right)$.

The same grouping and algorithm as $G2_{Fk}$ can be used for total exchange with the exception that this time each node sends $\frac{N}{k+1}$ messages at a time. Therefore, the cost of this algorithm, $TE2_{Fk}$, is $\sum_{i=1}^{\log_{k+1} N} \left(\tilde{d} + \frac{N}{k+1}\right) = \left(\tilde{d} + \frac{N}{k+1}\right)\log_{k+1} N$. Figure 5.9 illustrates the above algorithm when N =9 and k = 2.

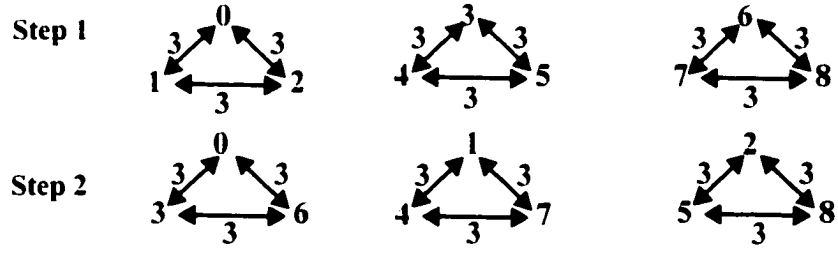


Figure 5.9: Total exchange on an 9-node *RON (k, N)* under 2-port modeling

Which algorithm, $TE1_{Fk}$ or $TE2_{Fk}$, is faster depends on the number of nodes N, number of input/output channels, k, and the term, $\tilde{d}$. Just like the single-port modeling, a *combined total exchange algorithm*, $TE3_{Fk}$, is proposed which is a combination of the above two algorithms.

Lemma 3 The *combined total exchange algorithm* under k-port modeling on *RON (k, N)* has a cost of

$$TE3_{Fk} = i\left(\tilde{d} + \frac{N}{k+1}\right) + \frac{1}{k}\left(\frac{N}{(k+1)^i} - 1\right)(\tilde{d} + (k+1)^i) \tag{5.43}$$

where i is the number of steps to do the standard exchange algorithm before switching to the direct algorithm.

Proof. In the combined total exchange algorithm and under k-port modeling, each time a standard exchange algorithm step is done a cost of $\tilde{d} + \frac{N}{k+1}$ is added. This brings up the term $i\left(\tilde{d} + \frac{N}{k+1}\right)$. The first part of the second term, $\frac{1}{k}\left(\frac{N}{(k+1)^i} - 1\right)$, is for the number of nodes in the groups doing the direct algorithms simultaneously. The second part, $(\tilde{d} + (k+1)^i)$, stands for the delay associated with the transfer of messages. ■

It is clear that this algorithm is exactly the same as the direct algorithm when $i = 0$, and the standard exchange algorithm when $i = \log_{k+1} N$. I haven't found any mathematical proof that this algorithm is better than the known algorithms. However, in all the numerical examples (more than one hundred thousand examples) that I have performed for the comparison of these algorithms, I have always found a step, i, for which, the combined total exchange algorithm had a shorter or equal exchange time than both the direct algorithm, $(\tilde{d} + 1)\left(\frac{N-1}{k}\right)$, and the standard exchange algorithm, $\left(\tilde{d} + \frac{N}{k+1}\right)\log_{k+1} N$. The

above statement is also true for single-port modeling. Therefore, It is conjectured that the proposed algorithm is better than (or at least equal to) both known algorithms. Table 5.5 and Table 5.6 summarize some typical examples with optimal costs for $TE3_{F1}$, and $TE3_{Fk}$.

Table 5.5: Total exchange time, $N = 1024$, single-port

$\tilde{d}$	$TE1_{F1}$	$TE2_{F1}$	$TE3_{F1}$
2	3069	5140	2558 ($i = 1, 2$)
5	6138	5170	3202 ($i = 3$)
20	21483	5320	4272 ($i = 5$)
50	52173	5620	5082 ($i = 6$)

Table 5.6: Total exchange time, $N = 1024$, $k = 3$

$\tilde{d}$	$TE1_{Fk}$	$TE2_{Fk}$	$TE3_{Fk}$
2	1023	1290	768 ($i = 1$)
5	2046	1305	963 ($i = 2$)
20	7161	1380	1248 ($i = 3$)
50	17391	1530	1466 ($i = 3$)

5.8 Summary

In this chapter, I presented and analyzed a broadcasting algorithm [20] that could effectively hide the reconfiguration delay d in the network, $RON(k, N)$. Essentially, in this algorithm, the reconfiguration phase of some of the nodes is overlapped with the message transmission phase of the other nodes which ultimately reduces the broadcasting time. The analysis of the broadcasting algorithm includes a closed formulation that yields the termination time.

The solution for the total exchange problem combines two known algorithms, *direct* [109, 120], and *standard exchange* [71, 24], and it includes an optimization phase that determines the number of steps after which the first algorithm terminates and the second

one is engaged. This ensures a termination time that is better than what can be accomplished by either of the two algorithms. Meanwhile, known algorithms for scattering and all-to-all broadcasting from literature [40, 21] have been adapted to the network, *RON (k, N)*.

The scattering, multinode broadcasting, and total exchange algorithms discussed in this chapter assumed that the number of nodes in the *RON* (k, N) is a power of 2, or a power of $(k + 1)$ under single-port and k-port modeling, respectively. However, when the number of processors is not a power of 2, or a power of $(k + 1)$, dummy nodes can be assumed to exist until the next power of 2 or $(k + 1)$ with a little performance loss.

So far, in this thesis, I have been concerned about efficient communications in message-passing parallel computer systems using reconfigurable interconnects. I have used knowledge of the next destination (either by prediction or algorithmically) to hide the reconfiguration latency of the interconnect. In Chapter 6, regardless of the type of the interconnection network, I utilize prediction techniques in general, and more specifically the proposed predictors in Chapter 3, to remove the redundant message copying at the receiving side of communications in message-passing systems.

Chapter 6

Efficient Communication Using Message Prediction for Clusters of Multiprocessors

A significant portion of the software communication overhead belongs to a number of message copying operations. Ideally, it is desirable to have a true zero-copy protocol where the message moves directly from the send buffer in its user space to the receive buffer in the destination without any intermediate buffering. However, due to the fact that message-passing applications at the send side do not know the final receive buffer addresses, early arriving messages have to be buffered in a temporary area.

I explain the motivation behind this work and discuss related work in Section 6.2. In Section 6.3, I elaborate on how prediction would help eliminate message copying at the receiving side of communications. I explain the experimental methodologies to gather communication traces of the parallel applications in Section 6.4. I characterize some communication properties of the parallel application benchmarks by presenting the frequency and distributions of receive communication calls in Section 6.5. I show that there is a message reception communication locality in message-passing parallel applications [5]. Having this communication locality at the receiver sides, I use the proposed predictors introduced in Chapter 3 to predict the next consumable message. This chapter contributes by arguing that these message predictors can be efficiently used to drain the network and cache the incoming messages even if the corresponding receive calls have not been posted yet. This way, there is no need to unnecessarily copy the early arriving messages into a temporary buffer. As shown in Section 6.6, the performance of these predictors, in terms of hit ratio, on some parallel applications is quite promising [5] and suggest that prediction has the potential to eliminate most of the remaining message copies. I compare the performance and storage requirements of the predictors in Section 6.7. Finally, I summarize this chapter in Section 6.8.

6.1 Introduction

With the increasing uniprocessor and SMP computation power available today, interprocessor communication has become an important factor that limits the performance of workstations clusters. Essentially, communication overhead is one of the most important factors affecting the performance of parallel computers. Many factors affect the performance of communication subsystems in parallel systems. Specifically, communication hardware and its services, communication software, and the user environment (multiprogramming, multiuser) are the major sources of the communication overhead.

The communication hardware aspect includes the architecture and placement of the network interface, and the interconnection network and its services. Many architectures have been proposed for the network interfaces. They are classified as (1) direct [52, 7, 63, 80, 97, 88] and (2) memory-based [48, 112, 126, 23]. Direct network interfaces allow a processor to directly access the network queue. However, they mostly ignore the issue of multiprogramming. That is, a single thread can only use the network interface at a time. Memory-based interfaces provide protection but have high latency. Interconnection networks themselves are another source of communication hardware latency. Communication services including flow control, and message delivery also add to this latency.

Communication software overhead currently dominates the communication time in clusters of workstations. In the current generation of parallel computer systems, the software overheads are tens of microseconds [43]. This is worse in clusters of workstations. Even with high performance networks [23, 67, 111] available today, there is still a gap between what the network can offer and what the user application can see. The communication software overhead cost comes mainly from three different sources; crossing protection boundaries several times between the user space and the kernel space, passing several protocol layers, and involving a number of memory copying operations.

Several researchers are working to minimize the cost of crossing protection boundaries, and using simple protocol layers by utilizing *user-level messaging* techniques such as *Active Messages* (AM) [125], *Fast Messages* (FM) [102], *Virtual Memory-Mapped Communications* (VMMC-2) [48], *U-Net* [126], *LAPI* [110], *Basic Interface for Parallel-*

ism (BIP) [105], *Virtual Interface Architecture* (VIA) [49], and *PM* [121]. A significant portion of the software communication overhead belongs to a number of message copying. Ideally, message protocols should transfer messages in a single copy (this is usually called a true zero-copy). In other words, the protocol should copy the message directly from the send buffer in its user space to the receive buffer in the destination without any intermediate buffering. However, applications at the send side do not know the final receive buffer addresses and, hence, the communication subsystems at the receiving end still copy messages unnecessarily from the network interface to a system buffer, and then from the system buffer to the user buffer when the receiving application posts the receive call.

Some researchers have tried to avoid memory copying [48, 79, 106, 14, 119, 118]. While they have been able to remove the memory copying between the application buffer space and the network interface at the send side by using user-level messaging techniques, they haven't been able to remove the memory copying at the receiver sides completely. They may achieve a zero-copy messaging at the receiver sides only if the receive call is already posted, a rendez-vous type communication is used for large messages, or the destination buffer address is already known by a pre-communication. Note, however, that MPI-2 [93] supports a remote memory access (RMA) operation but this is mostly suitable for receiver-initiated communications arising from the shared-memory paradigm.

I am interested in bypassing the memory copying at the destination in the general case, eager or rendez-vous and for sender-initiated communications as in MPI [92, 93]. In this chapter, I argue that it is possible to address the message copying problem at the receiving side by speculation. I support my claim by showing that messages display a form of locality at the receiving ends of communications.

I introduce here, for the first time, the notion of message prediction for the receiving side of message-passing systems. By predicting the next receive communication call, and hence the next destination buffer address, before the receiving call is posted one will be able to copy the message directly into the CPU cache speculatively before it is needed so that in effect a zero-copy transfer can be achieved.

I am interested in utilizing the proposed predictors in Chapter 3 [3, 2], but this time at the receiver sides to predict the next consumable message and drain the network as soon as the message arrives. Upon a message arrival, a user-level thread is invoked. If the receive call has not been issued yet, the message will be cached, but efficient cache mapping mechanisms need to be devised to facilitate binding at the moment the receive call is issued. If the receive call has already been issued, then the message can be written to its final destination.

This chapter concentrates on message predictions at the destinations in message-passing systems using MPI in isolation. This is analogous to branch prediction, and coherence activity prediction [97] in isolation. Our tools are not ready for measuring the effectiveness of the predictors on the application run-time yet. My preliminary evaluation measures the accuracy of the predictors in terms of hit ratio. The results are quite promising and suggest that prediction has the potential to eliminate most of the remaining message copies.

6.2 Motivation and Related Work

High performance computing is increasingly concerned with efficient communication across the interconnect due to the availability of high-speed highly-advanced processors. Modern switched networks, called *System Area Networks* (SAN), such as Myrinet [23] and ServerNet [67], provide high communication bandwidth and low communication latency. However, because of high processing overhead due to communication software including network interface control, flow control, buffer management, memory copying, polling and interrupt handling, users cannot see much difference compared to traditional local area networks.

Fortunately, several user-level messaging techniques have been developed to remove the operating system kernel and protocol stack from the critical path of communications [125, 102, 48, 126, 49, 105, 110, 121]. This way, applications can send and receive messages without operating system intervention which often greatly reduces the communication latency.

Data transfer mechanisms and message copying, control transfer mechanisms, address translation mechanisms, protection mechanisms, and reliability issues are the key factors for the performance of a user-level communication system. In this chapter, I am particularly interested to avoid message copying at the receiver side of communications.

A significant portion of the software communication overhead belongs to a number of message copying. With the traditional software messaging layers, there are usually four message copying operations from the send buffer to the receive buffer, as shown in Figure 6.1. These copies are namely from the send buffer to the system buffer (1), from the system buffer to the network interface (NI) (2), and at the other end of communication from the network interface to the system buffer (3), and from the system buffer to the receive buffer (4) when the receive call is posted. Note that, I haven't considered data transfer from the network interface (NI) at the sending process to the network interface at the receiving process as a separate copy. Also, the network interface's place can be either on the I/O bus or on the memory bus.

At the send side, some user-level messaging layers use programmed I/O to avoid system buffer copying. FM uses programmed I/O while AM-II and BIP do so only for small messages. Some other user-messaging layers use DMA. VMMC-2, U-Net, and PM use DMA to bypass the system buffer copy while AM-II and BIP do so only for large messages. In systems that use DMA, applications or a library dynamically pins and unpins pages in the user space that contain the send and the receive buffers. Address translation can be done using a kernel module as in BIP, or by caching a limited number of address translations for the pinned pages as in VMMC-2, U-Net/MM [17], and PM. Some network interfaces also permit bypassing message copying at the network interface by directly writing into the network.

Contrary to the send side, bypassing the system buffer copying at the receiving side may not be achievable. Processes at the sending sides do not know the destination buffer addresses. Therefore, when a message arrives at the receiving side it has to be buffered if the receive call has not been posted yet. VMMC [22] for the SHRIMP multicomputer is a communication model that provides direct data transfer between the sender's and

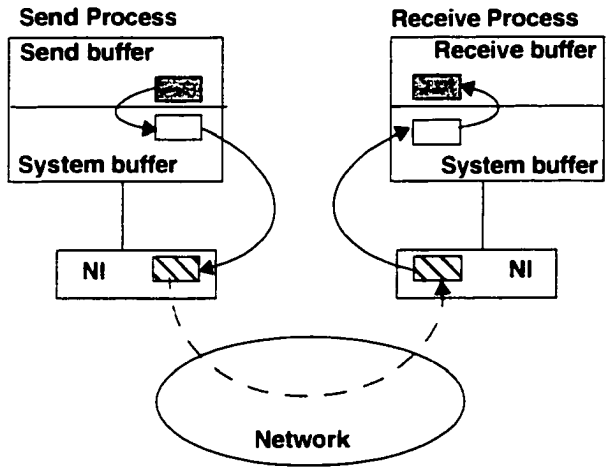


Figure 6.1: Data transfers in a traditional messaging layer

receiver's virtual address space. However, it can achieve zero-copy transfer only if the sender knows the destination buffer address. Therefore, the receiver exports its buffer address by scouting a message to the sender before the actual transmission can take place. This leads to a 2-phase rendez-vous protocol which adds to the network traffic, and network latency especially for short messages.

VMMC-2 [48], uses a *transfer redirection* mechanism instead. It uses a default, redirectable receive buffer for a sender who does not know the address of the receive buffer. When a message arrives at the receiving network interface, the redirection mechanism checks to see if the receiver has already posted its buffer address. If the receive buffer has been posted earlier than the message arrival, the message will be directly transferred to the user buffer. Thus it achieves a zero-copy transfer. If the buffer address is not posted, the message must be buffered in the default buffer. It will then be transferred when the receive buffer is posted. Thus, it achieves a one-copy transfer. However, if the receiver posts its buffer address when the message arrives, part of the message is buffered at the default buffer and the rest is transferred to the user buffer.

Fast sockets [106] has been built using active messages. It uses a mechanism at the receiver side called *receive posting* to avoid the message copy in the fast socket buffer. If the message handler knows that the data's final memory destination is already known upon message arrival the message is directly moved to the application user space. Otherwise, it has to be copied into the fast socket buffer.

FM 2.x [79] uses a similar approach as fast sockets, namely *layer interleaving*. FM collaborates with the handler to direct the incoming messages into the destination buffer if the receive call has already been posted.

MPI-LAPI [14] is an implementation of MPI on top of LAPI [110] for the IBM SP machines. In the implementation of the eager protocol, the header handler of the LAPI returns a buffer pointer to LAPI which tells LAPI where the packets of the message must be reassembled. If a receive call has been posted, the address of the user buffer is returned to LAPI. If the header handler doesn't find a matching receive, it will return the address of an *early arrival buffer* and hence a one-copy transfer is accomplished. Meanwhile, message sizes of larger than eager size is transferred using 2-phase rendez-vous protocol.

Some research projects have proposed solutions to multi-protocol message-passing interfaces on *clusters of multiprocessors* (Clumps) using both shared-memory for intra-node communications and message-passing for inter-node communications [118, 55, 87].

MPICH-PM/CLUMP [118] is an MPI library implemented on a cluster of SMPs. It uses a message-passing only model where each process runs on a processor of an SMP node. For inter-node communications, it uses *eager* and *rendez-vous* protocols. For short messages, it achieves one-copy using eager protocol as the message is copied into a temporary buffer if the MPI receive primitive has not been issued. For large message, it uses rendez-vous protocol to achieve zero-copy by using a remote write operation but it needs an extra communication. For intra-node communications, it achieves a one-copy using a kernel primitive that allows to copy messages from the sender to the receiver without involving the communication buffer.

BIP-SMP [55], for intra-node communications, uses shared memory for small messages with two memory copy, and direct copy for large messages with a kernel overhead. For inter-node communications, it works like MPI-BIP which is a port of MPICH [57].

TOMPI [38] is a threaded implementation of MPI on a single SMP node. It copies a message only once by utilizing multiple threads on an SMP node. Unfortunately, it is not scalable to a cluster of SMP machines.

Other techniques to bypass extra copying are the *re-mapping*, and *copy-on-write* techniques [31, 45]. Both techniques require switching to the supervisor mode, acquiring necessary locks to virtual memory data structure, and changing virtual memory mapping at several levels for each page, and then performing *Translation Lookaside Buffer* (TLB)/ cache consistency actions, and finally returning to the user mode. This limits the performance of the page re-mapping, and copy-on-write techniques. A zero-copy TCP stack is implemented in Solaris by using copy-on-write pages and re-mapping to improve communication performance [31]. It achieves a relatively high throughput for large messages. However, it does not have a good performance for small messages. This work is also solely dedicated to the SUN Solaris virtual memory system.

fbufs [45] is also using the re-mapping technique to avoid the penalty of copying large messages across different layers of protocol stack. However, fbufs allows re-mapping only for a limited range of user virtual memory.

It is quite clear that even user-level messaging techniques may not achieve a zero-copy communication all the time at the receiver side of communications. Meanwhile, the major problem with all page re-mapping techniques is their poor performance for short messages which is extremely important for parallel computing.

As stated in Chapter 3, many prediction techniques have been proposed in the past to predict the future accesses of sharing patterns and coherence activities in distributed shared memory (DSM) by looking at their observed behavior [96, 77, 73, 133, 34, 107]. Recently, Afsahi and Dimopoulos proposed some heuristics to predict the destination target of subsequent communication requests at the send side of communications in mes-

sage-passing systems [3, 4]. However, to the best of my knowledge, no prediction technique has been proposed for the receive side of communications in message-passing systems to reduce the latency of a message transfer.

This chapter of the thesis, reports on an innovative approach for removing message copying at the receiving ends of communications for message-passing systems. I argue that it is possible to address the message copying problem at the receiving sides by speculation. I introduce message prediction techniques such that messages can be directly transferred to the cache even if the receive calls have not been posted yet.

6.3 Using Message Predictions

In this section, I analyze the problem with the early arrival of messages at the destinations in message-passing systems. In such systems, a number of messages arrive in arbitrary order at the destinations. The consuming process or thread will consume one message at a time. If I know which message is going to be consumed next, then I can move the message upon its arrival to near the place that it is to be consumed (e.g. a staging cache), or I could schedule which thread to execute next preferably at the same processor as the consuming thread to enhance the chances that the data will be in the processor cache when it is accessed by the consumer.

For this, one has to consider three different issues. First, deciding which message is going to be consumed next. This can be done by devising receive call predictors, history-based predictors that predict subsequent receive calls by a given process in a message-passing program. Second, deciding where and how this message is to be moved in the cache. Third, efficient cache re-mapping and late binding mechanisms need to be devised for when the receive call is posted.

In this chapter, I am addressing the first problem. That is, utilizing message predictors and evaluating their performance. I am working on several methods to address the remaining issues.

6.4 Experimental Methodology

In exploring the effect that different heuristics have in predicting the next receive call, I used a number of parallel benchmarks, and extracted their communication traces on which I applied the predictors. Specifically, I used BT, SP, and CG benchmarks from NPB suite [13], and PSTSWM application [128], introduced in Chapter 2. I didn't use the MG and LU benchmarks form the NPB suite because these benchmarks use *MPI_ANY_SOURCE* in some of their receive calls (MPI_Recv and MPI_Irecv). This means that the applications may receive a particular message from different sources depending on the order of arrival. I also didn't use the QCDMPI application as this application uses the synchronous communication primitive, *MPI_Sendrecv_replace*, where the sender waits for the receive call to be posted. Then it transmits the message. In this case, prediction wouldn't help as the receive call is already posted.

I experimented with the workstation class "W", and the larger class "A" of the NPB suite, and the default problem size for the PSTSWM application. Note that because of space and access limitations I did not experiment with the larger classes "B", and "C". The NPB results are almost the same for "W" and "A" classes. Hence, I report only for the "A" class here. Note that I also removed the initialization part from the communication traces of the PSTSWM application.

6.5 Receiver-side Locality Estimation

The applications use blocking and nonblocking standard MPI receive primitives, namely *MPI_Recv* and *MPI_Irecv* [92]. *MPI_Recv (buf, count, datatype, source, tag, comm, status)* is a standard blocking receive call. When it returns, data is available at the destination buffer. The PSTSWM application uses this type of receive call. *MPI_Irecv (buf, count, datatype, source, tag, comm, request)* is a standard nonblocking receive call. It immediately posts the call and returns. Hence, data is not available at the time of return. It needs another call to complete the call. All applications in this study use this type of receive call.

As noted earlier in Chapter 3, one of the communication characteristics of any parallel application is the frequency of communications. Figure 6.2 illustrates the minimum, average, and maximum number of receive communication calls in the applications under different system sizes. I executed the applications once for each different system size and counted the number of receive calls for each process of the applications. Hence, in Figure 6.2, by average, minimum, and maximum, I mean the average, minimum, and maximum number of receive calls taken over all processes of each application. It is clear that all processes in the BT, SP, and CG applications have the same number of receive communication calls for each different system size. While processes in the PSTSWM application have different number of receive communication calls.

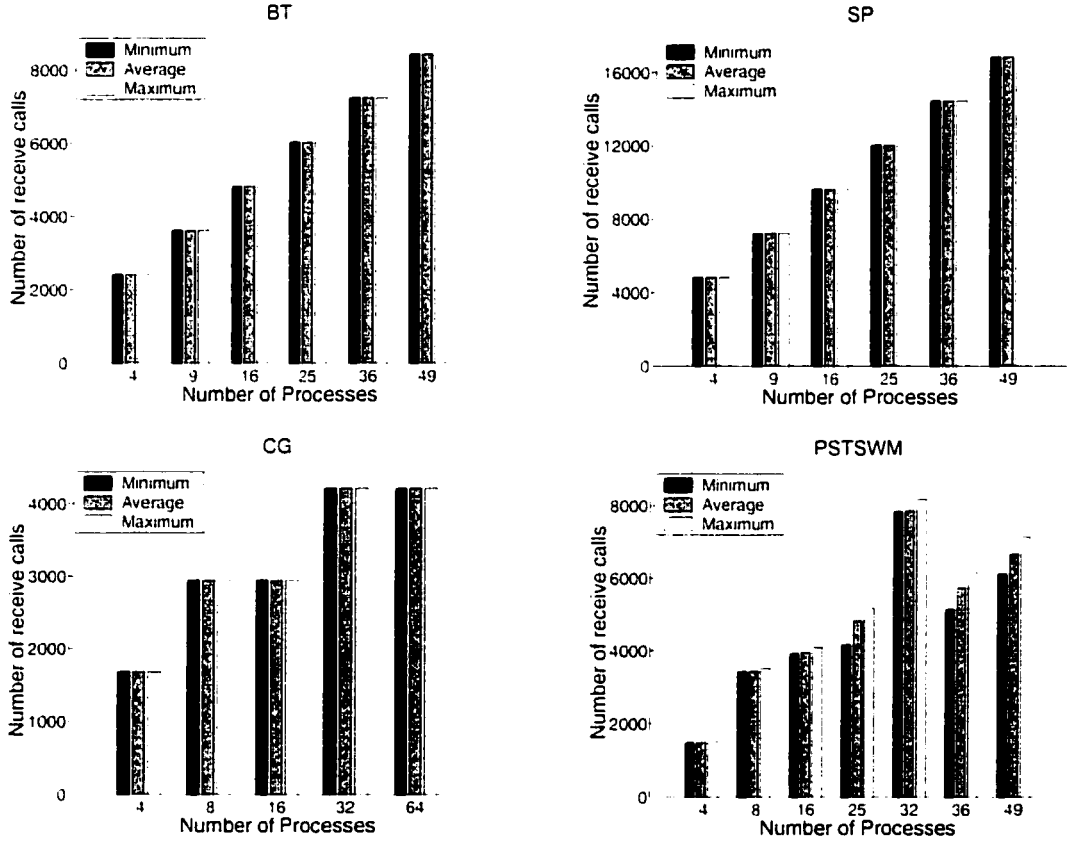


Figure 6.2: Number of receive calls in the applications under different system sizes

MPI_Recv and MPI_Irecv calls have a 7-tuple set consisting of *source*, *tag*, *count*, *datatype*, *buf*, *comm*, and *status* or *request*. In order to choose precisely one of the received messages at the network interface and transfer it to the cache, the predictors need to consider all the details of a message envelop. That is, *source*, *tag*, *count*, *datatype*, *buf*, and *comm* (I don't consider *status* and *request* as they are just a handle when the calls return). I did not rely only on the buffer address, *buf*, of a receive call as many processes may send their messages to the same buffer address of a particular destination process. Nor I could depend only on the sender, *source*, of a message, or on the length, *count*, of a message. Therefore, I assigned a different identifier for each unique 6-tuple found in the communication traces of the applications. Figure 6.3 shows the number of *unique message identifiers* in the applications under different system sizes. By average, minimum, and maximum, I mean the average, minimum, and maximum number of unique identifiers taken over all processes of each application. It is evident that all processes in the BT, and CG applications have the same number of unique message identifiers while processes in the SP, and PSTSWM applications have different number of unique message identifiers (except when the number of processes is four for the SP benchmark).

Figure 6.4 shows the distribution of each unique message identifier for process zero of the applications when the number of processes is 64 for CG and 49 for the other applications. I chose process zero because this process almost always had the largest number of unique message identifiers among all processes in the applications and is also responsible for distributing data and verifying the results of the computation. As it is shown in Figure 6.4, the message identifiers are evenly distributed in BT. However, the distribution of the message identifiers in CG and PSTSWM are almost bimodal with two separated peaks. The SP benchmark shows four different peaks for the message identifiers. Similar distributions have been found for other system sizes [6].

6.5.1 Communication Locality

As noted in Chapter 3, some researchers have tried to find or use the *communications locality* properties of parallel applications [3, 4, 75, 30, 36]. I define the term *message reception locality* in conjunction with this work. By message reception locality I mean that

if a certain message reception call has been used it will be re-used with high probability by a portion of code that is "near" the place that was used earlier, and that it will be re-used in the near future.

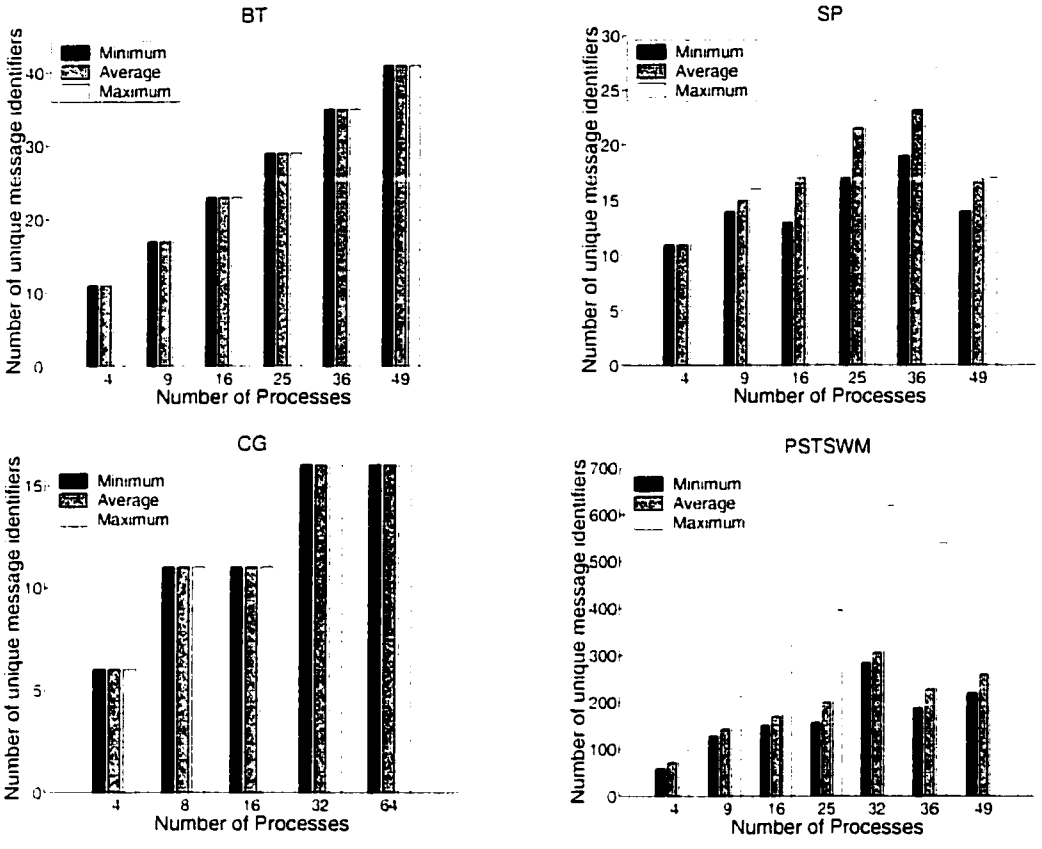


Figure 6.3: Number of unique message identifiers in the applications under different system sizes

In the following subsection, I present the performance of the classical LRU, LFU, and FIFO heuristics on the applications to see the existence of locality or repetitive receive calls. I use the *hit ratio* to establish and compare the performance of these heuristics. As a hit ratio, I define the percentage of the times that the predicted receive call was correct out of all receive communication requests.

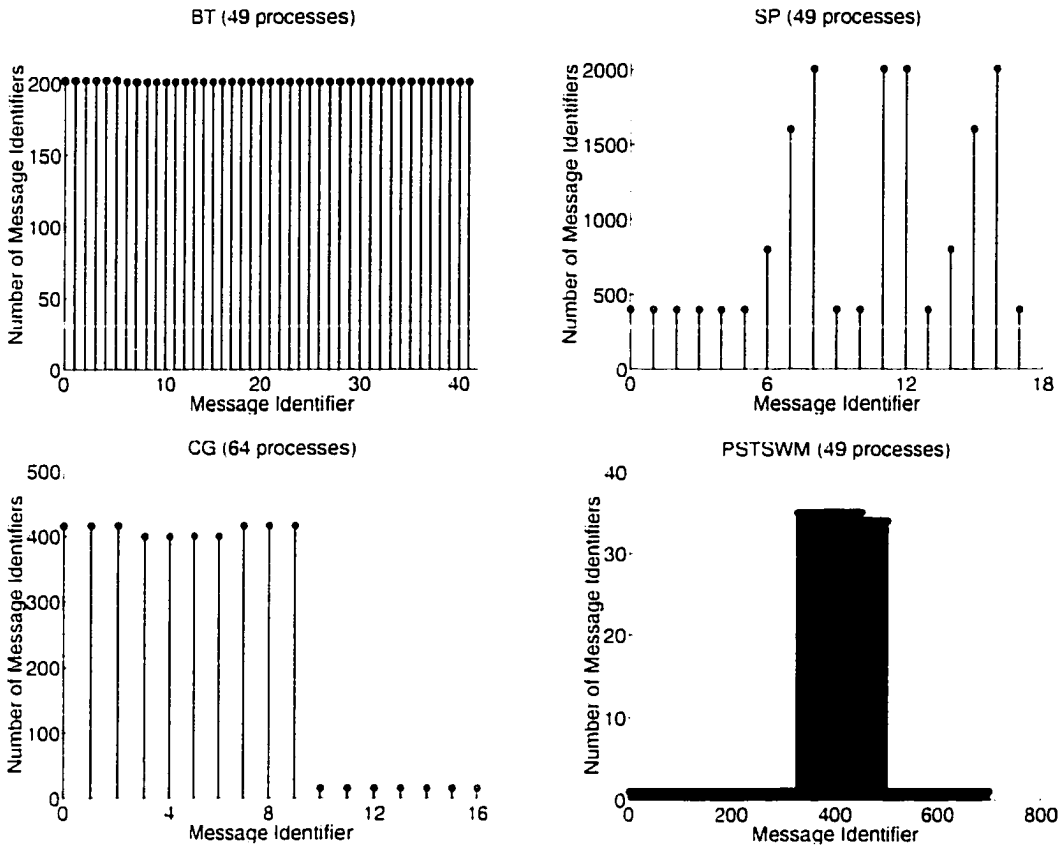


Figure 6.4: Distribution of the unique message identifiers for process zero in the applications

6.5.2 The LRU, FIFO and LFU Heuristics

The *Least Recently Used* (LRU), *First-In-First-Out* (FIFO), and *Least Frequently Used* (LFU) heuristics, all maintain a set of k (k is the window size) unique message identifiers. If the next message identifier is already in the set, then a hit is recorded. Otherwise, a miss is recorded and the new message identifier replaces one of the identifiers in the set according to which of the LRU, FIFO or LFU strategies is adopted.

Figure 6.5 shows the results of the LRU, FIFO, and LFU heuristics on the application benchmarks when the number of processes is 64 for CG and 49 for all other applications. It is clear that the hit-ratios in all benchmarks approach 1 as the window size increases. The performance of the FIFO algorithm is the same as the LRU for BT, and PSTSWM benchmarks, and almost the same for the SP and CG benchmarks. The LFU algorithm

consistently has a better performance than the LRU and FIFO heuristics on the BT, CG, and PSTSWM applications. It also has a better performance than the LRU and FIFO heuristics on the SP benchmark for window sizes of greater than five. It is interesting to see that a real application like PSTSWM needs window sizes of greater than 150 to achieve a good performance (hit ratios above 80%) under the LFU policy. Similar performance results for the LRU, FIFO, and LFU heuristics on other system sizes can be found in [6].

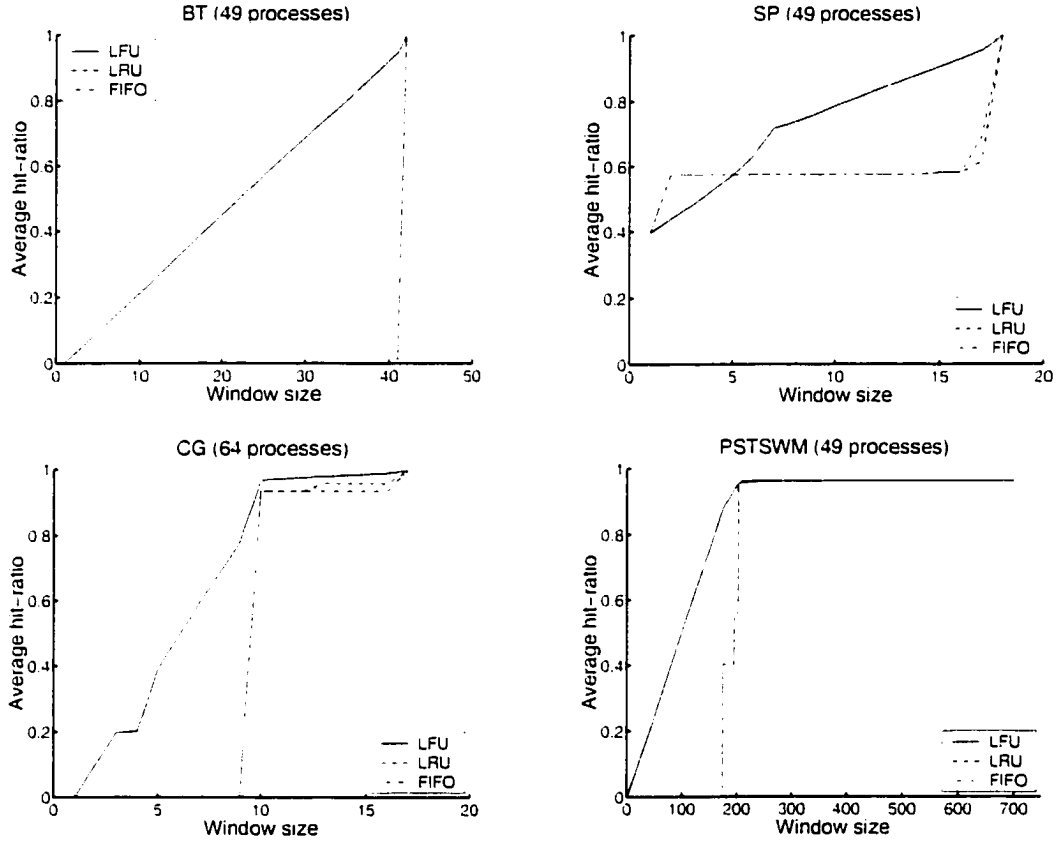


Figure 6.5: Effects of the LRU, FIFO, and LFU heuristics on the applications

Essentially, the LRU, FIFO and LFU heuristics do not predict exactly the next receive call but shows the probability that the next receive call might be in the set. For instance, the SP benchmark shows nearly a 60% hit ratio for a window size of five under the LRU heuristic. This means that 60% of the time one of the five most recently issued calls will be issued next. These heuristics perform better when the window size k is sufficiently large.

However, this large window adds to the hardware and software implementation complexity as one needs to move all messages in the set to the cache in the likelihood that one of them is going to be used next. This is prohibitive for large window sizes.

I am interested in having predictors that can predict the next receive call with a high probability. In Section 6.6, I utilize the novel message predictors proposed in Chapter 3 employing different heuristics and evaluate their performance on the applications.

6.6 Message Predictors

The set of predictors used in this section predict the subsequent receive calls based on the past history of communication patterns on a per process basis. These predictors were proposed in Chapter 3 to predict the destination target of subsequent communication requests at the send side of communications. It is worth mentioning that the message reordering effect [77] (messages from different processes may arrive out-of-order even if messages from the same processes may arrive in-order in most networks) has no effect on the predictions as the predictors predict the next receive calls based on the patterns of the receive calls in the program that runs on the same process and not on the arriving messages unless the order of receive calls depends on the order of message arrival. Note that in the following figures, by average, minimum, and maximum, I mean the average, minimum, and maximum hit ratio taken over all processes of each application.

6.6.1 The Tagging Predictor

As described earlier in Chapter 3, the *Tagging* predictor assumes a static communication environment in the sense that a particular communication receive call in a section of code, will be the same one with a large probability. I attach a different *tag* to each of the receive calls found in the applications. This can be implemented with the help of a compiler or by the programmer through a *pre-receive (tag)* operation which will be passed to the communication subsystem to predict the next receive call before the actual receive call is issued.

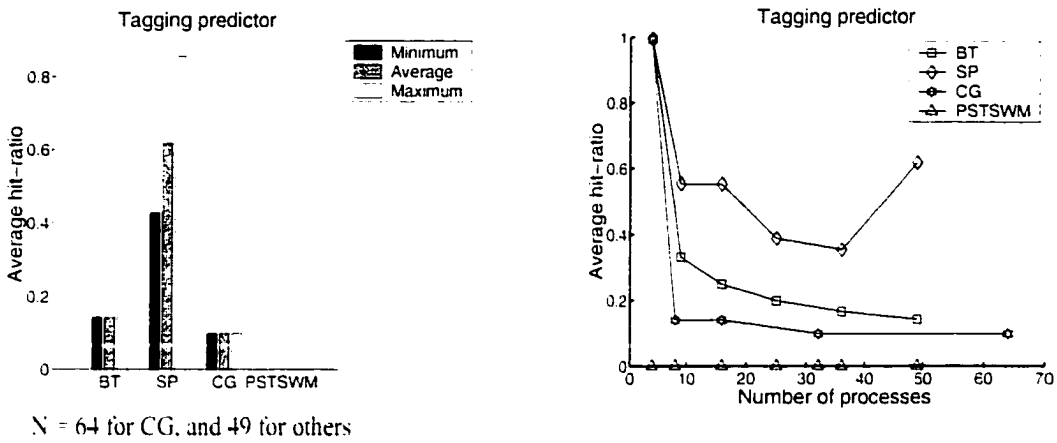


Figure 6.6: Effects of the Tagging predictor on the applications

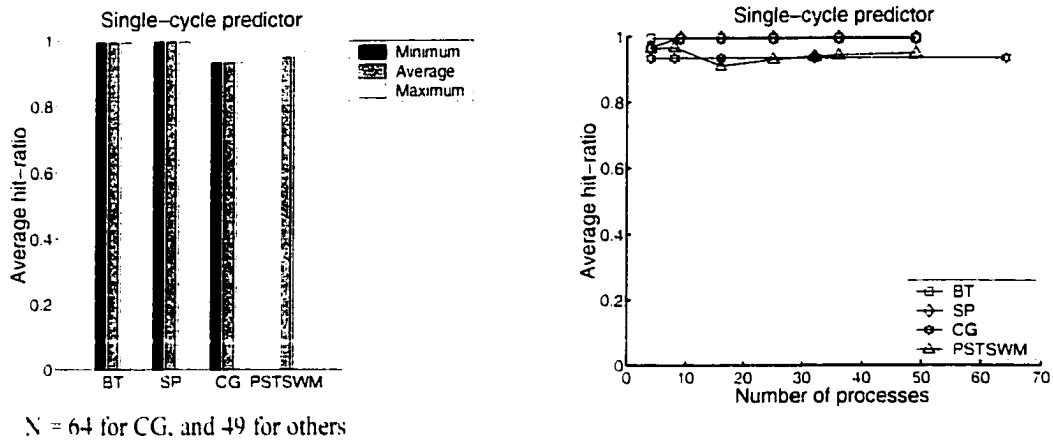
The performance of the Tagging predictor is shown in Figure 6.6. It is evident that this predictor doesn't have a good performance for the applications studied. It cannot predict the communication patterns of PSTSWM at all, and has a degrading performance for all other applications when the number of processes increases.

6.6.2 The Single-cycle Predictor

The *Single-cycle* predictor, proposed in Chapter 3, is based on the fact that if a group of receive calls are issued repeatedly in a cyclical fashion, then I can predict the next request one step ahead. The performance of the Single-cycle predictor is shown in Figure 6.7. It is evident that its performance is consistently very high (hit ratios of more than 0.9). Note that for the PSTSWM application, the Single-cycle predictor has a zero hit-ratio for one of the processes. However, it doesn't affect the average hit-ratio over all the processes. It is worth mentioning that all Cycle-based predictors proposed in Chapter 3, (Single-cycle, Single-cycle2, Better-cycle, and Better-cycle2) have the same performance for the applications studied. Thus, I just reported the results for the Single-cycle predictor here.

6.6.3 The Tag-cycle2 Predictor

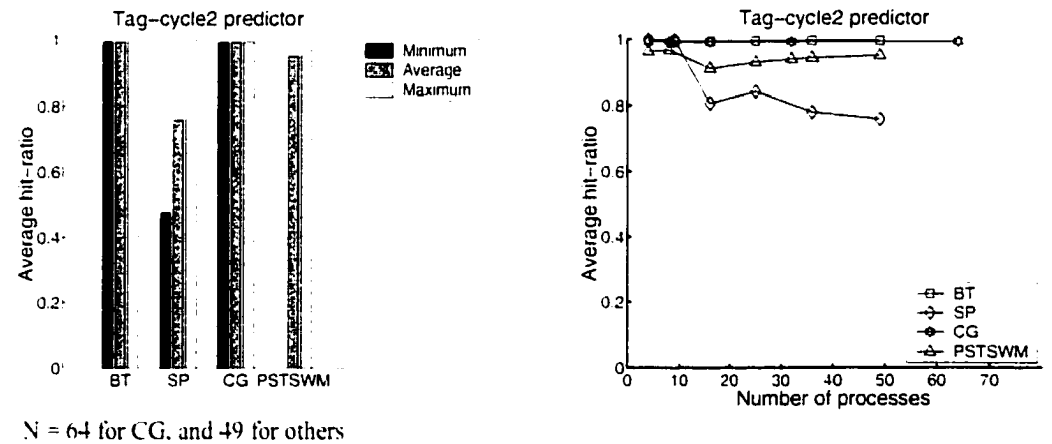
The Tag predictor didn't have a good performance on the applications while the Single-cycle predictor had a very good performance. The *Tag-cycle2* predictor, proposed in Chapter 3, is a combination of the Tag predictor and the Single-cycle2 predictor. In the Tag-cycle2 predictor, I attach a different tag to each of the communication requests found



N = 64 for CG, and 49 for others

Figure 6.7: Effects of the Single-cycle predictor on the applications

in the benchmarks and do a Single-cycle2 discovery algorithm on each tag. The performance of the Tag-cycle2 predictor is shown in Figure 6.8. The Tag-cycle2 predictor performs well on all benchmarks. Its performance is the same as the Single-cycle predictor on BT and PSTSWM. However, it has a better performance on CG and a lower performance on SP.



N = 64 for CG, and 49 for others

Figure 6.8: Effects of the Tag-cycle2 predictor on the applications

6.6.4 The Tag-bettercycle2 Predictor

In the Single-cycle and Tag-cycle2 predictors, as soon as a receive call breaks a cycle I remove the cycle and form a new cycle. In the *Tag-bettercycle2* predictor, proposed in Chapter 3, I keep the last cycle associated with each tagbettercycle-head encountered in

the communication patterns of each process. This means that when a cycle breaks I maintain the elements of this cycle in memory for later references. The performance of the Tag-bettercycle2 predictor is shown in Figure 6.9. The Tag-bettercycle2 predictor performs well on all benchmarks. Its performance is the same as the Single-cycle and Tag-cycle2 predictors on the BT and PSTSWM. However, it has a better performance on the CG and a lower performance on the SP relative to the Single-cycle predictor. The Tag-bettercycle2 predictor has a better performance on the SP application compared to the Tag-cycle2 predictor. I also found that the applications have very small number of tagbettercycle-heads (at most 2) under the Tag-bettercycle2 predictor and different system sizes.

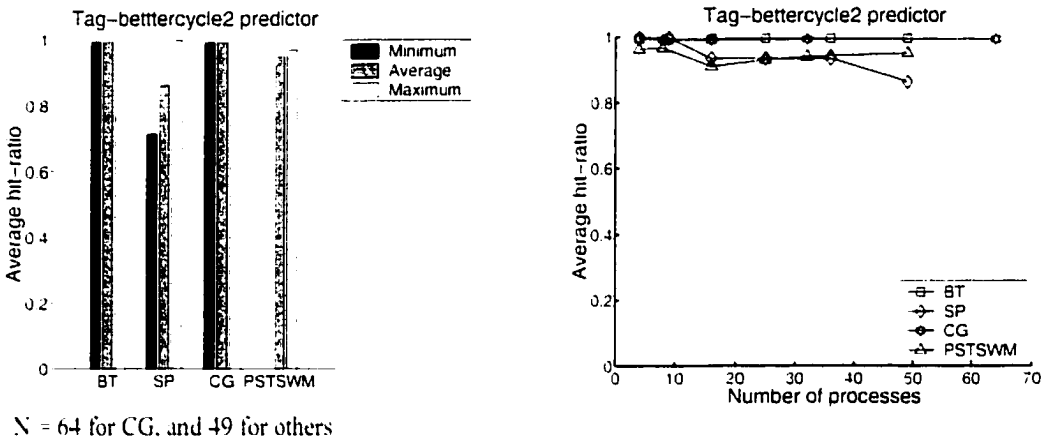


Figure 6.9: Effects of the Tag-bettercycle2 predictor on the applications

6.7 Message Predictors' Comparison

Figure 6.10 presents a comparison of the performance of the predictors on the applications under some typical system sizes. As we have seen so far, Single-cycle, Tag-cycle2 and Tag-bettercycle2 all perform exceptionally well on the benchmarks. However, the performance of the Single-cycle is better on the SP benchmark while Tag-cycle2 and Tag-bettercycle2 have better performance on the CG benchmark.

6.7.1 Predictor's Memory Requirements

Table 6.1 compares the maximum memory requirement of the message predictors on the application benchmarks when the number of processes is 64 for CG, and 49 for BT, SP, and PSTSWM. I have found that the memory requirement of the predictors decrease grad-

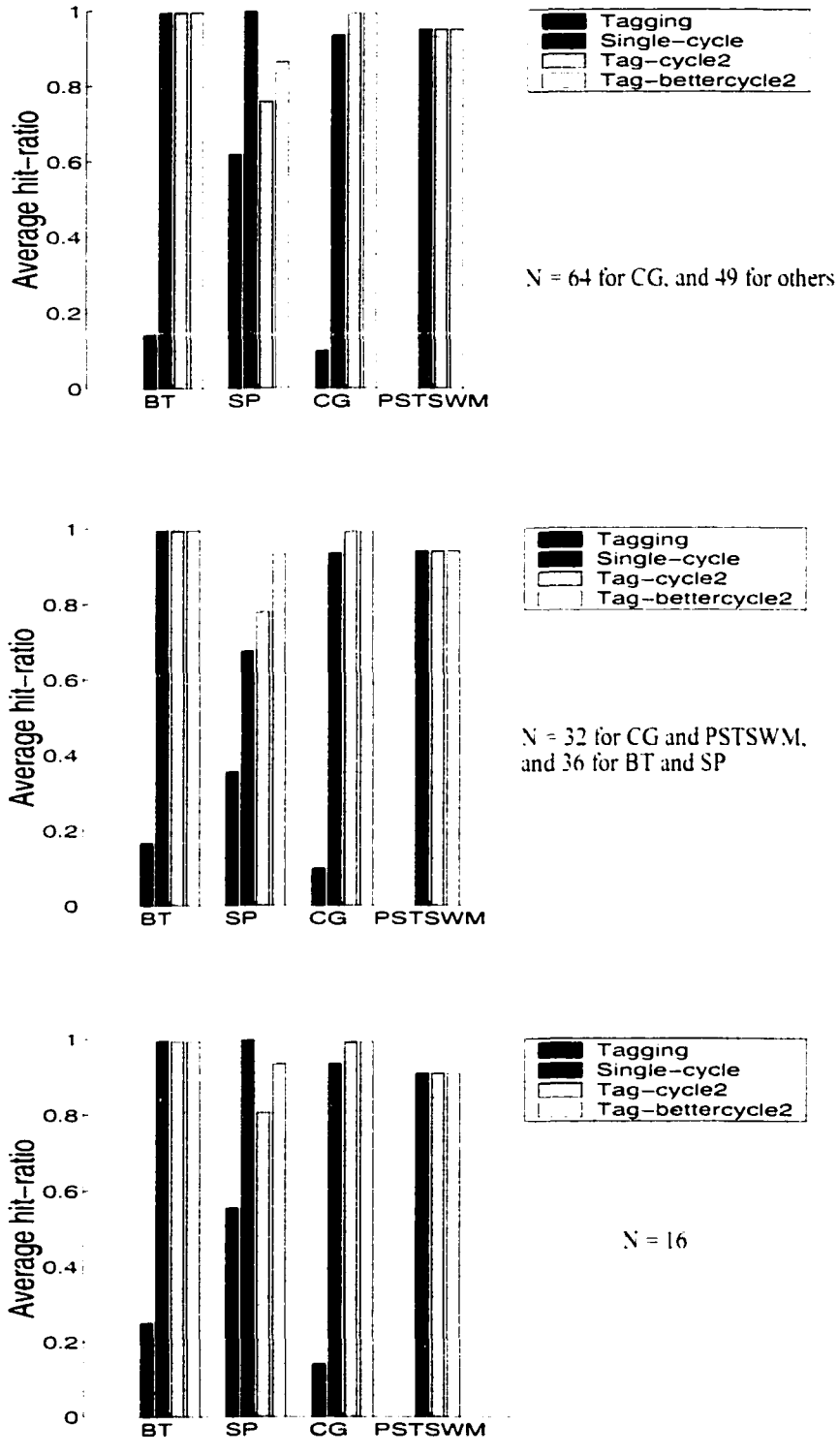


Figure 6.10: Comparison of the performance of the predictors on the applications

ually when the number of processes decreases. The numbers in the table are the multiplication factor for the amount of storage needed to maintain the message 6-tuple sets. It is quite clear that the memory requirements of the predictors is low. That makes them very attractive for the implementation at the network interface. Comparatively, predictors (Single-cycle, Tag-cycle, and Tag-bettercycle) need higher memory requirement for the PSTSWM application. Although, the classical LRU, LFU, and FIFO heuristics need less memory requirements, but as stated earlier, the beauty of the predictors lies on the fact that they predict with high accuracy and transfer only one message to the cache which should dramatically reduce the cache pollution effect, if any. This should also bring down the software cost of the implementation.

Table 6.1: Memory requirements (in 6-tuple sets) for the predictors when $N = 64$ for CG, and $N = 49$ for BT, SP, and PSTSWM

	BT	**SP**	**CG**	**PSTSWM**
Tagging	12	12	10	7
Single-cycle	43	43	138	204
Tag-cycle2	60	72	40	693
Tag-bettercycle2	60	108	40	693

6.8 Summary

Communication latency adversely affects the performance of networks of workstations. A significant portion of the software communication overhead belongs to a number of message copying operations. Ideally, it is very desirable to have a true zero-copy protocol where the message is moved directly from the send buffer in its user space to the receive buffer in the destination without any intermediate buffering. However, this is not always possible as a message may arrive at the destination where the corresponding receive call has not been issued yet. Hence, the message has to be buffered in a temporary buffer.

In this chapter of the dissertation, I have shown that there is a message reception communication locality in message-passing applications. I have utilized the different predictors proposed in Chapter 3 to predict the next receive call at the receiver side of

communications. By predicting receive calls early, a process can perform the necessary data placement upon message reception and move the message directly into the cache. I presented the performance of these predictors on some parallel applications. The performance results are quite promising and justify more work in this area.

I envision these predictors to be used to drain the network and place the incoming messages in the cache in such a way so as to increase the probability that the messages will still be in cache when the consuming thread needs to access them.

Chapter 7

Conclusions and Directions for Future Research

Parallel processing is the key to the design of high performance computers. However, with the availability of fast microprocessors and small-scale multiprocessors, internode communication has become an increasingly important factor that limits the performance of parallel computers. In essence, parallel computers require extremely short communication latency such that network transactions have minimal impact on the overall computation time. This thesis uses a number of techniques to achieve efficient communications in message-passing systems. This thesis makes five contributions.

The first contribution of this thesis is the design and evaluation of two different categories of prediction techniques for message-passing systems. I present evidence that message destinations display a form of locality. This thesis utilizes the message destination locality property of message-passing parallel applications to devise a number of heuristics that can be used to predict the target of subsequent communication requests.

Specifically, I propose two sets of message destination predictors: *Cycle-based* predictors, which are purely dynamic predictors, and *Tag-based* predictors, which are static/dynamic predictors. In cycle-based predictors, *Single-cycle*, *Single-cycle2*, *Better-cycle*, and *Better-cycle2*, predictions are done dynamically at the network interface without any help from the programmer or compiler. In Tag-based predictors, *Tagging*, *Tag-cycle*, *Tag-cycle2*, *Tag-bettercycle*, and *Tag-bettercycle2*, predictions are done dynamically at the network interface as well, but they require an interface to pass some information from the program to the network interface. This can be done with the help of programmer or compiler through inserting instructions in the program such as *pre-connect (tag)*. The performance of the proposed predictors, specially Better-cycle2 and Tag-bettercycle2, are very

well on all application benchmarks. Meanwhile, the memory requirements of the predictors is very low. The proposed predictors should be easily implemented on the network interface due to their simple algorithms and low memory requirements.

The heuristics proposed are only possible because of the existence of communications locality that can be used in establishing a communication pathway between a source and a destination in reconfigurable interconnects before this pathway is to be used. This is a very desirable property since it allows us to effectively hide the cost of establishing such communications links, providing thus the application with the raw power of the underlying hardware (e.g. a reconfigurable optical interconnect).

As the second contribution of this thesis, I show that the majority of reconfiguration delays in single-hop reconfigurable networks can be hidden by using one of the proposed high hit ratio predictors. In other words, by comparing the inter-send computation times of some parallel benchmarks with some specific reconfiguration times, most of the time, we are able to fully utilize these computation times for the concurrent reconfiguration of the interconnect when we know, in advance, the next target using one of the proposed high hit ratio target prediction algorithms. This thesis also states that by utilizing the predictors at the send side of communications, applications at the receiver sides would also benefit as messages arrive earlier than before.

As the third contribution of this thesis, I analyze a broadcasting algorithm that utilizes latency hiding and reconfiguration in the network to speed the broadcasting operation under single-port and k-port modeling. In this algorithm, the reconfiguration phase of some of the nodes is overlapped with the message transmission phase of the other nodes which ultimately reduces the broadcasting time. The analysis brings up closed formulations that yield the termination time of the algorithms.

The fourth contribution of this thesis is a new total exchange algorithm in single-hop reconfigurable networks under single-port and k-port modeling. I conjecture that this algorithm ensures a better termination time than what can be achieved by either of the direct, and standard exchange algorithms.

Ideally, message protocols should copy the message directly from the send buffer in its user space to the receive buffer in the destination without any intermediate buffering. However, Applications at the send side do not know the final receive buffer addresses and, hence, the communication subsystems at the receiving end still copy messages unnecessarily at a temporary buffer.

This thesis presents evidence that there exists message reception communications locality in message-passing parallel applications. Having message reception communications locality, the fifth contribution of this thesis is the use and evaluation of the proposed predictors to predict the next consumable message at the receiving ends of communications. This thesis contributes by claiming that these message predictors can be efficiently used to drain the network and cache the incoming messages even if the corresponding receive calls have not been posted yet. This way, there is no need to unnecessarily copy the early arriving messages into a temporary buffer. The performance of the proposed predictors, Single-cycle, Tag-cycle2 and Tag-bettercycle2, on the parallel applications are quite promising and suggest that prediction has the potential to eliminate most of the remaining message copies.

7.1 Future Research

The proposed predictors in Chapter 3 of this thesis such as Tag-bettercycle2 and Better-cycle2 perform exceptionally well on all applications except QCDMPI, under different system sizes. It seems that this application repeatedly changes its message destinations in different cycles that even the best proposed predictors cannot always capture them. Thus, it might be helpful to devise other predictors, called *All-cycle* and *Tag-allcycle*, that could maintain all cycles associated with each cycle-head and tagbettercycle-head found in the communication traces of the applications. In case that these two predictors, All-cycle and Tag-allcycle, have high memory requirements, it might be better to devise predictors that fall somewhere between the extreme cases. That is, predictors that can maintain more than one cycle but less than all of the cycles associated with each cycle-head and tagbettercycle-head. Not to mention that searching in different cycles may add to the performance penalty.

The Tag-based predictors proposed in Chapter 3 can be pure dynamic predictors if another level of prediction is done on the tag themselves at the network interface. This way, there is no need for the program to pass *pre-connect (tag)* (or *pre-receive (tag)* as in Chapter 6) information to the network interface. It is interesting to see what would be the performance of such *2-level Tag-based* predictors.

In Chapter 4, I roughly showed that up to 50% of the times applications at the receiving end might benefit when the predictors are applied at the send side of communications. However, a trace-driven simulator should be written to precisely evaluate the effect that applying the predictors at the send side has on the receive side, and on the total application run-time.

This thesis in Chapter 5 analyzes efficient broadcasting/multi-broadcasting algorithms that utilizes latency hiding to speed these operations. An optimal algorithm for multi-broadcasting is to be devised such that messages are pipelined in the embedded trees using the latency hiding broadcasting algorithms ($B4_{Fk}$, or B_{FI}). In this thesis, although algorithms for scattering, all-to-all broadcasting, and total exchange are very efficient but they do not use latency hiding technique. Although very challenging, efficient algorithms for multicasting, scattering, all-to-all broadcasting, and total exchange should be devised such that they use latency hiding technique to hide the reconfiguration delay in the network.

As stated in Chapter 6, by predicting receive calls early, a node can perform the necessary data placement upon message reception and move the message directly into the cache in such a way so as to increase the probability that the messages will still be in cache when the consuming thread needs to access them. Further issues that should be investigated are deciding where and how this message is to be moved in the cache. Would this cache be a first-level cache, a second-level cache, a third-level cache or even a network-cache? What mechanism should be used to transfer the message into the cache? User-level messaging and/or multithreaded MPI environment. Meanwhile, efficient cache re-mapping and late binding mechanisms need to be devised for when the receive call is posted. Also, cache pollution and inaccurate timing are the other issues that should be addressed.

The performance of the predictors proposed in this thesis were evaluated under single-port modeling. That is the predictors predict one step ahead. However, Cycle-based predictors, Single-cycle, Single-cycle2, Better-cycle, and Better-cycle2, and Tagcycle-based predictors, Tag-cycle, Tag-cycle2, Tag-bettercycle, and Tag-bettercycle2 maintain the message destinations of a cycle. Therefore, it is possible to predict more than one step ahead. It is interesting to find the performance of the predictors under such modeling in terms of hit ratio, and for the total reconfiguration delays, and the application run time.

Finally, all the applications studied in this dissertation are scientific and engineering ones. It is interesting to discover the impact of the predictors on the performance of commercial applications.

Bibliography

[1] A. Afsahi and N. J. Dimopoulos, "Collective Communications on a Reconfigurable Optical Interconnect", *Proceedings of the OPODIS'97, International Conference on Principles of Distributed Systems*, December, 1997, pp. 167-181.

[2] A. Afsahi and N. J. Dimopoulos, "Communications Latency Hiding Techniques for a Reconfigurable Optical Interconnect: Benchmark Studies", *Proceedings of the of PARA98, Fourth International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Springer-Verlag, *Lecture Notes in Computer Science, 1541*, June 1998, pp. 1-6.

[3] A. Afsahi and N. J. Dimopoulos, "Hiding Communication Latency in Reconfigurable Message-Passing Environments", *Proceedings of the IPPS/SPDP 1999, 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, April 1999, pp. 55-60.

[4] A. Afsahi and N. J. Dimopoulos, "Communication Latency Hiding in Reconfigurable Message-Passing Environments: Quantitative Studies", *Proceedings of the HPCS'99, 13th Annual International Symposium on High Performance Computing Systems and Applications, Kluwer Academics Publishers*, June 1999, pp. 111-126.

[5] A. Afsahi and N. J. Dimopoulos, "Efficient Communication Using Message Prediction for Cluster of Multiprocessors", *Proceedings of the CANPC'00, Fourth Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, held in conjunction with the 6th International Symposium on High-Performance Computer Architecture, HPCA-6*, January 2000.

[6] A. Afsahi and N. J. Dimopoulos, "Efficient Communication Using Message Prediction for Clusters of Multiprocessor", *Technical Report ECE-99-5, Department of Electrical and Computer Engineering, University of Victoria*, December 1999.

[7] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowiz, B-H. Lim, K. Mackenzie and D. Yeung, "The MIT Alewife Machine: Architecture and Performance", *Proceedings* e *of the 22th Annual International Symposium on Computer Architecture*, 1998.

[8] A. Alexandrov, M. Ionescu, K. E. Schauser and C. Scheiman, "LogGP: Incorporating Long Messages Into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation", *7th Annual Symposium on Parallel Algorithms and Architecture (SPAA'95)*, July 1995.

[9] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, 1989.

[10] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstation, IEEE Computer, Volume 29, no. 2, February 1996, pp. 18-28.

[11] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team, "A case for Networks of Workstations: NOW", *IEEE Micro*, February 1995.

[12] T. E. Anderson, S. S. Owicki, P. Saxe, and C. P. Thacker, "High Speed Switch Scheduling for Local Area Networks", *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 98-110.

[13] D. H. Bailey, T. Harsis, W. Saphir, R. V. der Wijngaart, A. Woo and M. Yarrow, "The NAS Parallel Benchmarks 2.0: Report NAS-95-020", Nasa Ames Research Center, December 1995.

[14] M. Banikazemi, R. K. Govindaraju, R. Blackmore and D. K. Panda, "Implementing Efficient MPI on LAPI for IBM RS/6000 SP Systems: Experiences and Performance Evaluation, "*Proceedings of the of IPPS/SPDP 1999, 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, April 1999, pp. 183-190.

[15] M. Banikazemi, J. Sampathkumar, S. Prabhu, D. K. Panda, and P. Sadayappan, "Communication Modeling of Heterogeneous Networks of Workstations for Performance Characterization of Collective Operations", *Proceedings of the International Workshop on Heterogeneous Computing, in conjunction with IPPS/SPDP'99*, April 1999, pp. 125-131.

[16] A. Bar-Noy and S. Kipnis, "Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems", *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1992, pp. 11-22.

[17] A. Basu, M. Welsh, T. V. Eicken, "Incorporating Memory Management into User-Level Network Interfaces", *Hot Interconnects V*, August 1997.

[18] C. Berge, *Hypergraphs*, North-Holland, 1989.

[19] P. Berthome and A. Ferreira, Editors, *Optical Interconnections and Parallel Processing: Trends at the Interface*, Kluwer Academic Publishers, 1998.

[20] P. Berthome and A. Ferreira, "Communication Issues in Parallel Systems with Optical Interconnections", *International Journal of Foundations of Computer Science*, Volume 8, Number 2, June 1997, pp. 143-162.

[21] P. Berthome and A. Ferreira, "On Broadcasting Schemes in Restricted Optical Passive Star Systems", Interconnection Networks and Mapping and Scheduling Paral-

lel Computations", *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Volume 21, 1995, pp. 19-29.

[22] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg, "A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer", *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 142-153.

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

[24] S. H. Bokhari and H. Berryman, "Complete Exchange on a Circuit Switched Mesh", *Proceedings of the 1992 Scalable High Performance Computing Conference*, April 1992, pp. 300-306.

[25] H. Bourdin, A. Ferriera, and K. Marcus, "A Comparative Study of One-to-Many WDM Lightwave Interconnection Networks for Multiprocessors", *Proceedings of the Second International Conference on Massively Parallel Processing using Optical Interconnections*, 1995, pp. 257-263.

[26] C. A. Brackett, "Dense Wavelength Division Multiplexing Principles and Applications", *IEEE Journal of Selected Areas in Communications*, August 1990, pp. 948-964.

[27] P. J. Cameron, *Combinatorics: Topics, Techniques, Algorithms*, Cambridge University Press, 1994.

[28] L. J. Camp, "Guided-Wave and Free-Space Optical Interconnects for Parallel Processing Systems: a Comparison", *Applied Optics*, Volume 33, No. 26, September 10 1994, pp. 6168-6180.

[29] D. M. Chiarulli, S. P. Levitan, R. P. Melhem, J. P. Teza and G. Gravenstreter, "Partitioned Optical Passive Star (POPS) Multiprocessor Interconnection Networks with Distributed Control", *IEEE Journal of Lightwave Technology*, Volume 14, No. 7, 1994, pp. 1601-1612.

[30] S. Chodnekar, V. Srinivasan, A. Vaidya, A. Sivasubramaniam and C. Das, "Towards a Communication Characterization Methodology for Parallel Applications", Proceedings of the Third International Symposium on High Performance Computer Architecture, 1997.

[31] H. Chu, "Zero-copy TCP in Solaris, "*Proceedings of the USENIX Annual Technical Conference*, 1996, pp. 253-263.

[32] D. E. Culler, J. P. Singh and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1999.

[33] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation", *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[34] F. Dahlgren, M. Dubois and P. Stenstrom, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, 6(7), 1995.

[35] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, "The Message Driven Processor: A Multicomputer Processing Nodes with Efficient Mechanisms", *IEEE Micro*, April 1992, pp. 23-39.

[36] B. V. Dao, Sudhakar Yalamanchili, and Jose Duato, "Architectural Support for Reducing Communication Overhead in Multiprocessor Interconnection Networks", *Proceedings of the Third International Symposium on High Performance Computer Architecture*, 1997, pp. 343-352.

[37] D. G. de Lahaut and C. Germain, "Static Communications in Parallel Scientific Programs", *Proceedings of PARLE'94, Parallel Architecture and Languages*, July 1994.

[38] E. D. Demaine, "A Threads-Only MPI Implementation for the Development of Parallel Programs", *Proceedings of the 11th International Symposium on High Performance Computing Systems, HPCS'97*, 1997, pp. 153-163.

[39] Department of Energy *Accelerated Strategic Computing Initiative* (ASCI) Project, http://www.llnl.gov/asci/.

[40] F. Desprez, A. Ferriera and B. Tourancheau, "Efficient Communication Operations on Passive Optical Star Networks", *Proceedings of the First International Conference on Massively Parallel Processing using Optical Interconnections*, 1994, pp. 52-58.

[41] V. Dimakopoulos and N. J. Dimopoulos, "Total Exchange in Cayley Networks", *Euro-Par' 96, Parallel Processing, Lecture Notes in Computer Science*, 1996, pp. 341-346.

[42] V. V. Dimakopoulos and N. J. Dimopoulos, "Communications in Binary Fat Trees", *Proceedings of the International Conference on Parallel and Distributed Computing*, September 1995, pp. 383-388.

[43] J. J. Dongarra and T. Dunigan, "Message-Passing Performance of Various Computers", *Concurrency*, Volume 9, No. 10, December 1997, pp. 915-926.

[44] P. W. Dowd, "Wavelength Division Multiple Access Channel Hypercube Processor Interconnection", *IEEE Transactions on Computers*, Volume 41, October 1992, pp. 1223-1241.

[45] P. Druschel and L. L. Peterson, "Fbufs: A High-bandwidth Cross-domain Transfer Facility", *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 1993, pp. 189-202.

[46] J. Duato, S. Yalamanchili and L. Ni, *Interconnection Networks: An Engineering Approach*, IEEE Computer Society Press, 1997.

[47] J. Duato, "A Necessary and Sufficient Condition for Deadlock-free Adaptive Routing in Wormhole Networks", IEEE Transactions on Parallel and Distributed Systems, Volume 6, No. 10, 1995, pp. 1055-1067.

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

[49] 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*, March-April, 1998, pp. 66-76.

[50] L. Fan, M. C. Wu, H. C. Lee and P. Grodzinski, "Optical Interconnection Networks for Massively Parallel Processors using Beam Steering Vertical Cavity Surface-Emitting Lasers.", *Proceedings of the Second International Conference on Massively Parallel Processing using Optical Interconnections*, October 1995, pp. 28-34.

[51] M. R. Feldman, S. C. Esener, C. C. Guest and S. H. Lee, "Comparison Between Optical and Electrical Interconnects Based on Power and Speed Considerations", *Applied Optics*, 27(9), May 1988, pp. 1742-1751.

[52] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich and W. S. Lee, "The M-Machine Multicomputer", *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitectures*", 1995.

[53] P. Fraigniaud and E. Lazard, "Methods and Problems of Communication in Usual Networks", *Discrete Applied Mathematics*, Volume 53, 1994, pp. 79-133.

[54] M. Galles, "Spider: A High-Speed Network Interconnect", *IEEE Micro*, Volume 17, No. 1, January/February 1997.

[55] P. Geoffray, L. Prylli, and B. Tourancheau, "BIP-SMP: High Performance Message Passing Over a Cluster of Commodity SMPs", *SC99: High Performance Networking and Computing Conference*, November, 1999.

[56] C. J. Glass and L. M. Ni, "The Turn Model for Adaptive Routing", *Proceedings of the 17th International Symposium on Computer Architecture*, 1992, pp. 278-287.

[57] W. Gropp and E. Lusk, "User's Guide for MPICH, a Portable Implementation of MPI", *Argonne National Laboratory, Mathematics and Computer Science Division*, June, 1999.

[58] J. W. Goodman, F. I. Leonberger, S-Y. Kung and R. A. Athale, "Optical Interconnections for VLSI Systems", *Proceedings of IEEE*, Volume 72, No. 7, July 1984.

[59] G. Gravenstreter and R. G. Melhem, "Realizing Common Communication Patterns in Patitioned Optical Passive Stars (POPS) Networks", *IEEE Transactions on Computers*, Volume 47, No. 9, 1998, pp. 998-1013.

[60] M. W. Haney and M. P. Christensen, "Fundamental Geometric Advantages of Free-Space Optical Interconnect", *Proceedings of the Third International Conference on Massively Parallel Processing using Optical Interconnections*, 1996, pp. 16-23.

[61] S. M. Hedetneimi et al., "A Survey of Gossiping and Broadcasting in Communication Networks", *Networks*, Volume 18, 1988, pp. 319-349.

[62] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1996.

[63] D. S. Henry and C. F. Joerg, "A Tightly-Coupled Processor-Network Interface", *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[64] H. S. Hinton, T. J. Cloonan, F. B. McCormick, Jr., A. L. Lentine and F. A. P. Tooley, "Free-Space Digital Optical Systems", *Proceedings of IEEE. Special Issue on Optical Computing Systems*, Volume 82, No. 11, Nov. 1994, pp. 1632-1649.

[65] S. Hioki, "Construction of Staples in Lattice Gauge Theory on a Parallel Computer", *Parallel Computing*, Volume 22, No. 10, October 1996, pp. 1335-1344.

[66] R. W. Hockney, "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2", *Parallel Computing*, Volume 20, No. 3, March 1994, pp. 389-398.

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

[68] J. Hsu and P. Banjeree, "Performance Measurement and Trace Driven Simulation of Parallel CAD and Numerical Applications on a Hypercube Multicomputer", *Proceedings of the 17th International Symposium on Computer Architecture*, 1990, pp. 260-269.

[69] K. Hwang and Z. Xu, *Scalable Parallel Computing: Parallelism, Scalibility, Programmability*, McGraw-Hill, 1998.

[70] S. L. Johnsson, "Communication in Network Architectures", in *VLSI and Parallel Computation*, ed. R. Suaya and G. Birtwistle, Morgan Kaufmann, 1990.

[71] S. L. Johnsson and C.-T. Ho, "Optimum Broadcasting and Personalized Communication in Hypercubes", *IEEE Transactions on Computers*, Volume C-38, September 1989, pp. 1249-1268.

[72] S. Karlson and M. Brorsson, "A Comparative Characterization of Communication Patterns in Applications Using MPI and Shared Memory on an IBM SP2", *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, International Symposium on High Performance Computer Architecture*, February 1998.

[73] S. Kaxiras and J. R. Goodman, "Improving CC-NUMA Performance Using Instruction-Based Prediction", *International Symposium on High Performance Computer Architecture*, 1999.

[74] F. E. Kiamilev, "Performance Comparison between Optoelectronic and VLSI Multistage Interconnection Networks", *Journal of Lightwave Technology*, Volume 9, No. 12, December 1991, pp. 1674-1692.

[75] J. Kim and D. J. Lilja, "Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs", *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, in conjunction with the International Symposium on High Performance Computer Architecture*, February 1998, pp. 202-216.

[76] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, The Benjamin/Cummings Publishing Company, Inc., 1994.

[77] A.-C. Lai and B. Falsafi, "Memory Sharing Predictor: The Key to a Speculative Coherent DSM", *Proceedings of the 26th Annual International Symposium on Computer Architectures*, 1999, pp. 172-183.

[78] LAM/MPI Parallel Computing, University of Notre Dame, http://www.mpi.nd.edu/lam/.

[79] M. Lauria, S. Pakin and A. A. Chien, "Efficient Layering for High Speed Communication: Fast Messages 2.x", *Proceedings of the 7th High Performance Distributed Computing (HPDC7) Conference*, 1998.

[80] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S-W.Yang and R. Zak, "The Network Architecture of the Connection Machine CM-5", *Proceedings of the 4th AM Symposium on Parallel Algorithms and Architectures*, June 1992, pp. 272-285.

[81] A. L. Lentine, K. W. Goosen, J. A. Walker, L. M. F. Chirovsky, L. A. D'Asaro, S. P. Hui, B. J. Tseng, R. E. Leibenguth, J. E. Cunningham, W. Y. Jan, J.-M. Kuo, D. W. Dahringer, D. P. Kossives, D. D. Bacon, G. Livesue, R. K. Morrison, R. A. Novotny, and D. B. Buchholz, "High-Speed Optoelectronic VLSI Switching Chip with > 4000 Optical I/O Based on Flip-chip Bonding of MQW Modulators and Detectors to Silicon CMOS", *IEEE Journal of Selected Topics in Quantum Electronics*, Volume 2, April, 1996.

[82] K. Li, Y. Pan and S. Q. Zheng, Editors, *Parallel Computing Using Optical Interconnections*, Kluwer Academic Publishers, 1998.

[83] A. Louri and H. K. Sung, "An Optical Multi-Mesh Hypercube: A Scalable Optical Interconnection Network for Massively Parallel Computing", *Journal of Lightwave Technology*, Volume 12, No. 4, 1994, pp. 704-716.

[84] A. Louri and H. K. Sung, "Scalable Optical Hypercube-based Interconnection Network for Massively Parallel Computing", *Applied Optics*, Volume 33, No. 32, Nov. 1994, pp. 7588-7598.

[85] D. B. Loveman, "High Performance Fortran", *IEEE Parallel and Distributed Technology*, Volume 1, February 1993, pp. 25-42.

[86] Lucent's Wavestar LambdaRouter, *IEEE, Computer*, January 2000, pp. 26.

[87] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler, "Multi-Protocol Active Messages on a Cluster of SMPs", *SC97: High Performance Networking and Computing Conference*, November, 1997.

[88] K. Mackenzie, J. Kubiatowiz, M. Frank, W. Lee, V. Lee, A. Agarwal and M. F. Kaashock, "Exploiting Two-Case Delivery for Fast Protected Messaging", *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[89] P. J. Marchand, A. V. Krishnamoorthy, G. I. Yayla, S. C. Esener, and U. Efron, "Optically Augmented 3-D Computer: System Technology and Architecture",

Journal of Parallel and Distributed Computing, Special Issue on Optical Interconnects, February 25, 1997, pp. 20-35.

[90] P. K. McKinley and D. F. Robinson, "Collective Communication in Wormhole-Routed Massively Parallel Computers", *IEEE Computer*, December 1995, pp. 39-50.

[91] P. K. McKinley, H. Xu, A. -H. Esfahanian and L. M. Ni, "Unicast-based Multicast Communication in Wormhole-routed Networks", *IEEE Transactions on Parallel and Distributed Systems*, 5(12): 1252-1265, December 1994.

[92] *Message Passing Interface Forum: MPI: A Message-Passing Interface Standard.* Version 1.1 (June 1995).

[93] *Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface*, (July 1997).

[94] V. N. Morozov, H. Temkin and A. S. Fedor, "Analysis of a Three-Dimensional Computer Optical Scheme Based on Bidirectional Free-Space Optical Interconnects", *Optical Engineering*, Volume 34, No. 2, 1995, pp. 523-534.

[95] T. Mowry and A. Gupta, "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors", *Journal of Parallel and Distributed Computing*, 12(2), 1991, pp. 87-106.

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

[97] S. S. Mukherjee, B. Falsafi, M. D. Hill and D. A. Wood, "Coherent Network Interfaces for Fine-Grain Communication", *Proceedings of the 23th Annual International Symposium on Computer Architecture*, 1996.

[98] National Coordination Office for *Computing, Information, and Communications* (NCO/CIC), http://www.ccic.gov/.

[99] L. M. Ni, "Should Scalable Parallel Computers Support Efficient Hardware Multicast?", *International Conference on Parallel Processing*, Workshop, April 1995.

[100] R. A. Nordin, A. F. Levi, R. N. Nottenburg, J. O'Gorman, T. Tanbun-Ek, and R. A. Logan, "A System Perspective on Digital Interconnection Technology", *IEEE Journal of Lightwave Technology*, Volume 10, June 1992, pp. 801-827.

[101] N. Nupairoj and L. M. Ni, "Benchmarking of Multicast Communication Services", *Technical Report MSU-CPS-ACS-103, Michigan State University*, September

1995.

[102] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstation: Illinois Fast Messages (FM) for Myrinet," *Proceedings of the Supercomputing'95*, Nov., 1995.

[103] K. Panajotov, N. Nieuborg, A. Goulet, I. Veretennicoff and H. Thienpont, "A Free-space Reconfigurable Optical Interconnection based on Polarization-Switching VCSEL's and Polarization-Selective Diffractive Optical Channels", *Proceedings of the Optics in Computing*, 1998, pp. 151-154.

[104] T. M. Pinkston, "Design Considerations for Optical Interconnects in Parallel Computers", *Proceedings of the First International Workshop on Massively Parallel Processing Using Optical Interconnects*, April 1994, pp. 306-322.

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

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

[107] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles, "Predicting Multiprocessor Memory Access Patterns with Learning Models", *Proceedings of the Fourteenth International Conference on Machine Learning*," 1997, pp. 305-312.

[108] I. D. Scherson and A. S. Youssef, *Interconnection Networks for High-Performance Parallel Computers*, IEEE Computer Society Press, 1994.

[109] S. R. Seidel, "Circuit Switched vs. Store-and-Forward Solutions to Symmetric Communication Problems", *Proceedings of the 4th Conference on Hypercube Computers and Concurrent Applications*, 1989, pp. 253-255.

[110] G. Shah, J. Nieplocha, J. Mirza and C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, "Performance and Experience with LAPI -- a New High-Performance Communication Library for the IBM RS/6000 SP", *First Merged Symposium IPPS/SPDP 1998 12th International Parallel Processing symposium & 9th Symposium on Parallel and Distributed Processing*, 1998.

[111] R. Sheifert, "Gigabit Ethernet", *Addison-Wesley*, 1998.

[112] M. Snir and P. Hochschild, "The Communication Software and Parallel Environ-

ment of the IBM SP2", *IBM Systems Journal*, 34(2):205-221, 1995.

[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. Varker, "The SP2 High-Performance Switch", IBM Systems Journal, 34(2): 185-204, 1995.

[114] H. Sullivan and T. R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine", *Proceedings of the 4th Annual Symposium on Computer Architecture*, Volume 5, March 1977, pp. 105-124.

[115] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice and Experience*, Volume 2(4), December 1990, pp. 315-339.

[116] T. Szymanski, "Graph Theoretic Models for Photonic Networks", *Proceedings of the New Frontiers: A workshop on Future Directions of Massively Parallel Processing*, 1993, pp. 85-96.

[117] T. Szymanski, "Hypermeshes: Optical Interconnection Networks for Parallel Computing", *Journal of Parallel and Distributed Computing*, 26, 1995, pp. 1-35.

[118] T. Takahashi, F. O'Carrol, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, P.H. Beckman, "Implementation and Evaluation of MPI on an SMP Cluster", *Proceedings of the PC-NOW'99: International Workshop on Personal Computer based Networks Of Workstations, in conjunction with PPS/SPDP'99*, 1999.

[119] Y. Tanaka, M. Matsuda, M. Ando, K. Kubota and M. Sato, "COMPaS: A Pentium Pro PC-based SMP Cluster and its Experience", *Proceedings of the PC-NOW98: International Workshop on Personal Computer based Networks Of Workstations, in conjunction with PPS/SPDP'98*, 1998.

[120] R. Thakur and A. Choudhary, "All-to-all Communication on Meshes with Wormhole Routing", *Proceedings of the 1994 International Parallel Processing Symposium*, 1994, pp. 561-565.

[121] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication", *First Merged Symposium IPPS/SPDP 1998 12th International Parallel Processing symposium & 9th Symposium on Parallel and Distributed Processing*, 1998.

[122] K. Thulasiraman and M. N. S. Swamy, Graphs: Theory and Algorithms, John Wiley, 1992.

[123] G. Tricoles, "Computer Generated Holograms: A Historical review", *Applied*

Optics, Special Issue on Computer Generated Holograms, Volume 26, No. 20, 1987, pp. 4351-4360.

[124] A. Varma, *Interconnection Networks for Multiprocessors and Multicomputers: Theory and Practice*, IEEE Computer Society Press, 1993.

[125] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation", *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 256-265.

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

[127] D. S. Wills, W. S. Lacy, and J. Cruz-Rivera, "The Offset Cube: An Optoelectronic Interconnection Network", in K. Bolding and L. Synder (ED.) Parallel Computer Rouitng and Communication, Springer-Verlag, LNCS 853, pp. 86-100, 1994.

[128] P. H. Worley and I. T. Foster, "Parallel Spectral Transform Shallow Water Model: A Runtime-tunable parallel benchmark code", *Proceedings of the Scalable High Performance Computing Conference*, 1994, pp. 207-214.

[129] T. Yatagai, "Optical Computing and Interconnect", *Proceedings of IEEE*, Volume 84, No. 6, June 1996, pp. 828-852.

[130] G. I. Yayla, P. J. Marchand, and S. C. Esener, "Speed and Energy Analysis of Digital Interconnections: Comparison of On-chip, Off-chip and Free-Space Technologies", *Applied Optics*, Volume 37, No. 2, January 1998, pp. 205-227.

[131] T-Y Yeh and Y. Patt, "Alternative Implementation of Two-Level Adaptive Branch Prediction", *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp.124-134.

[132] X. Yuan, R. Melhem and R. Gupta, "Compiled Communication for All-Optical TDM Networks", *Proceedings of the Supercomputing'96*, 1996.

[133] Z. Zhang and J. Torrellas, "Speeding Up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching", *Proceedings of the 22nd Annual Symposium on Computer Architecture*, 1995, pp. 188-199.

Appendix A

Removing Timing Disturbances

I wrote my own profiling codes using the wrapper facility of the MPI to gather the communication traces, and the timing profiles of our application benchmarks. In this appendix, I explain how I removed the timing disturbances from the timing profiles of the applications.

Each inter-send computation time is the computation time between two successive communication operations (send operations). In the following example, t_3 - t_2 is the computation time between two successive *MPI_Send* operations where t_3 is the time just before the second call is issued while t_2 is the time just after the first send call finishes.

t_1 *MPI_Send (buf, count, datatype, dest, tag, comm);* t_2

...

... *computation*

...

t_3 *MPI_Send (buf, count, datatype, dest, tag, comm);* t_4

The example above has no other MPI calls between the two send primitives. In cases that other MPI calls exist between successive send calls, we have to take out these extra times to obtain the pure inter_send computation times. In the following example, two other MPI calls, *MPI_Irecv* and *MPI_Wait*, exist.

t_1 *MPI_Send (buf, count, datatype, dest, tag, comm);* t_2

...

... *computation*

...

t_3 *MPI_Irecv (buf, count, datatype, source, tag, comm, request);* t_4

...

... *computation*

...

t_5 *MPI_Wait (request, status);* t_6

...

... *computation*

...

t_7 *MPI_Send (buf, count, datatype, dest, tag, comm);* t_8

Therefore, the pure computation time is equal to $t_7 - t_2 - ((t_4 - t_3) + (t_6 - t_5))$. To compute the pure inter-send computation times, I need to know the exact times before and after each MPI call. For these, I did not insert the *MPI_Wtime* call in the source codes of the applications, but instead I wrote my own profiling codes to gather the timing traces. Thus, each MPI call in the applications calls its own profiling code, as shown in the following example for the *MPI_Send*.

t_a *MPI_Send (buf, count, datatype, dest, tag, comm);* t_b

***Profiling code*:**

start_time [index] = MPI_Wtime(); t_c

PMPI_Send (buf, count, datatype, dest, tag, comm);

end_time [index] = MPI_Wtime(); t_d

(i) *index++;*

(ii) *label = k;*

return;

The *MPI_Wtime* calls give the times, t_c and t_d, before and after the profiling call, *PMPI_Send*, respectively, while what I really need are the times t_a and t_b. It is clear that there are overheads entering and exiting the profiling code in addition to the overhead of the instructions *i* and *ii*. I computed these extra overheads for each type of the MPI calls used in the applications and took them out to find the pure inter-send computation times.