Abstract of thesis entitled

Performance Studies of High-Speed Communication on Commodity Cluster

submited by

Tam Tat Chun, Anthony

for the degree of Doctor of Philosophy
at The University of Hong Kong
in December 2001

Parallel computing is becoming increasingly accessible through advancement in microprocessors and networking technologies. It is found that the performances of the clusters do not match with their promises, although they are built on the most advanced components. Much effort has been devoted to address the software overhead problem in the past, which is known as the major hindrance in achieving high performance. This thesis shows that having a low-latency communication system does not guarantee high performance, as there are other communication issues that have not been addressed by the use of low-latency communication, such as contention, communication patterns and scheduling of communication events. The development of an efficient parallel application depends upon a realistic prediction of application behavior and the ability to explain the performance characteristics of an application on a parallel system; this requires in-depth understanding of both the application and the architecture characteristics.

This dissertation proposes the use of a realistic communication model to guide the performance understanding and the algorithm design processes, which are the keys to achieve high performance. The model includes a collection of performance parameters which correspond to essential features in the communication architecture, and a collection of benchmark methodologies that quantify these performance features. This model can be used as a framework for programmers to conduct performance studies on various communication issues. We use this framework to examine the performance characteristics of a lightweight messaging system, and show that the model can be effectively used as an evaluation tool as well as an emulating tool.

This thesis explores the congestion problem with the support of the communication model. Through modeling studies and experimental evaluations, we examine how different buffering mechanisms interact with a Go-Back-N reliable protocol, and show that the buffering mechanism dominates the congestion behavior of the communication systems. In the performance studies, we find that the timeout and flow control

settings are the prevailing factors that interact with the buffering mechanism to determine how the congestion evolves. As our model is derived on a resource-centric view of how data move across the communication system, we use it to show that with careful design of communication schedules, we can achieve efficient communication as well as prevent congestion. We have developed a complete exchange algorithm, the Synchronous Shuffle Exchange, which is an optimal algorithm on the non-blocking network. To avoid congestion loss caused by the non-deterministic delays in communication events, a global congestion control scheme is introduced. This scheme uses a global windowing concept to coordinate all participating nodes to monitor and regulate the traffic load, which effectively avoids congestion loss and maintains sufficient throughput to maximize the performance.

In summary, this thesis shows that the key to effective parallel programming on clusters lies with a realistic communication model, which enhances our understanding of the performance issues and supports the development of efficient parallel applications.

Performance Studies of High-Speed Communication on Commodity Cluster

by

Tam Tat Chun, Anthony

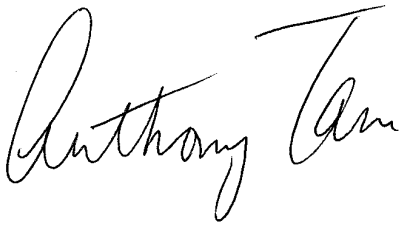
譚達俊

A thesis submitted in partial fulfilment of the requirements for
the Degree of Doctor of Philosophy
at the University of Hong Kong.

December 2001

Declaration

I declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualification.



Signed . . . . . .

Tam Tat Chun, Anthony

Acknowledgments

There are too many friends and colleagues who have offered advice or help to me during my course of studies at HKU. I would like to take this opportunity to express my appreciation on their support.

First and foremost, I would like to express my gratitude to my advisor, Cho-Li Wang, for his guidance, support, and encouragement throughout my graduate studies. Particularly, I would like to thank him for stimulating my interest in performance modeling, evaluation and prediction, and also for his patience in cultivating my research skills over the years. In addition, his technical and editorial advice was invaluable input in the development and completion of this thesis research.

I would also like to thank Dr. Francis Lau, who is my informal advisor and a member of my examination board, for his ability to step back, raise important questions, and explore the big picture during some critical periods on my research studies.

To my other two external examiners, Prof. Chung-Ta King and Dr. Pangfeng Liu, I would like to thank them for giving me excellent advice during my oral examination as well as for reading my dissertation.

My thanks also go to the other members of the System Research Group, notably David Lee and Benny Cheung. The contribution of David has been important to my research work. He is the designer of the Directed Point communication system, which becomes a valuable tool in exploring those communication issues studied in this dissertation. Also to my long-time officemate, Daisy Lee, who made my time at HKU much enjoyable and easier. I would also like to express thanks to all technical staffs in the CSIS Dept., notably Daniel Hung, W.S. Chan, Patrick Au and William Sin, for their helpfulness.

I am greatly indebted to my sibling, Catherine and Henry, for their long-term support and encouragement on pursuing my higher degrees over these years. The most important, I am grateful to my wife, Louisa, for her support and tolerance with me during these years. Without her love and care, I would never have finished off this dissertation.

Contents

List of Figures

x

List of Tables

List of Algorithms

Chapter 1

Introduction

Motivated by the desire to handle larger and complex problems, as well as to solve problems faster, we make use of multiple processing units for parallel computation. Developments of parallel computers, especially focusing on the architectural aspect, have been an active research subject since early 1960. During these forty years of developments, numerous parallel architectures have been developed, evolved and faded out, for examples, systolic architecture, dataflow architecture, and transputer system. Not until recent decades, due to the swiftly improvement of the VLSI technology, there is a clear convergence of parallel machines toward a generic parallel machine organization [32]. In this generic architecture, parallel machines are essentially comprised of a collection of complete computers, each with one or more processors and memory, and are interconnected by a communication network.

Advances in networking technology have accelerated this convergence. We are now capable to transform a pool of off-the-shelf computers to a powerful platform for supporting high-performance computations. This kind of computing platform is commonly known as Cluster of Workstations (COW), Network of Workstations (NOW), or simply Clusters [80]. By the name "commodity cluster", we refer to the clusters that are built on off-the-shelf components, such as high-performance microprocessors and high-speed networks. Better price-per-performance is the incentive of building clusters when compared to traditional parallel machines since they are built on commodity components. In addition, the performance of the cluster systems is getting along with the advances in commodity hardware.

These features are the selling points for using clusters on high-performance computing; however, just putting all state-of-the-art components together does not guarantee to be cost-effective and high-performance. The real challenge is how well can we harness these computing resources to meet our performance needs. As we are building clusters for high-performance computing, we have to face with challenges related to the performance issues on the cluster domain. In particular,

- ❑ Software design for performance - Developing application programs for performance is known to be difficult on traditional parallel machines, we reckon that it is still a big challenge on the cluster domain, as cluster is just another type of parallel platform.

- System design for performance - Building clusters with the most advanced processors and networks provides the most promising performance, however, the actual performance delivered to the end-users most often could not match with their promises. How to exploit these technology resources in the most efficient and cost-effective way imposes great challenges to the system designers.

- Performance scalability - On the architectural aspect, clusters can be easily grown in size, especially on incremental development that often matches with yearly financial plans [7]. This creates another challenge in building clusters for commodity supercomputing as how can we predict the performance of the scaled cluster with information only available on a small-scale prototype.

- Performance evaluation and analysis - Performance evaluation and analysis are the core activities in the software development cycle. A major objective of practical performance evaluation and analysis is to identify potential bottleneck(s) on the target application/platform pair. Based on these analyses, inefficient algorithms are excluded, good candidates are coded, debugged, and tested on the target platform. However, one of the difficulties is how to support the performance evaluation and analysis processes on the cluster systems so as to avoid the high cost of iterative development cycle - design, coding, debugging and testing.

We believe that, to achieve effective parallel programming on the cluster platform, this requires the ability to measure the performance of the parallel applications, the ability to determine the performance capability of the cluster systems, and the ability to explain the performance behavior of a parallel application on a cluster system. This demands system designers and programmers to have in-depth understanding of the interactions between various hardware and software components. In this thesis, we base on a realistic communication model to guide our understanding and structure our reasoning, as well as to perform performance tuning. This model is used as a versatile tool for performance evaluation and predication, as well as for algorithm design and analysis.

The rest of this chapter is organized as follows. We first state our motivation of building commodity cluster, and describe the limitations and challenges we have to tackle in order to achieve our goal. Next, we declare the thesis statement and highlight the contributions of this thesis. Lastly, we present an outline of the organization of this thesis.

1.1 Commodity Supercomputing

Speed is the main reason why we opt for the use of computers - both serial and parallel computers. Building parallel computers to solve large-scale and complex problems bear the same objective - we want to *speed up* the computations. Since parallel computing is becoming more and more accessible through the advances in processors and

network technologies, the use of parallelism is no longer the privilege of a few scientists with access to powerful supercomputers. Now we have pile of PCs sitting in the server room, and they are interconnected by high-speed LAN, these setups become our in-house parallel computer. However, is that the answer to our quest for more processing power? No, this is not. Building an efficient and high-performance cluster is not that simple as plugging in and setting up all components. To construct a high-performance cluster as well as to orchestrate its processing power, we certainly need to adopt a more scientific approach.

Performance Understanding Performance understanding is a process which is used to determine explanations on the performance behavior of a parallel application on a parallel machine. The development of efficient parallel applications depends upon a realistic prediction of application behavior, which requires in-depth understanding of both the application and the architecture characteristics [3, 29, 52]. For example, if the communication overhead of a parallel application dominates its execution time, the programmer may try to improve the communication schedule by rearranging the communication events to reduce the communication overhead. On the other hand, the system designer may try to improve the communication system to achieve the same performance goal. To determine how to perform performance tuning, both the programmer and system designer need to acquire detail knowledge on the operation costs, as well as to identify the potential bottleneck(s).

Therefore, the more we understand a parallel system and its performance characteristics, the more information we have to improve the efficiency of applications, as well as to provide insights on guiding enhancements to the system. This dissertation describes an attempt to improve or accelerate the performance understanding of the commodity clusters by focusing on the communication system. The network is the most critical path of a parallel system as its performance directly influences the capabilities of the system for high-performance computing. On the other hand, the communication efficiency of the network relies on the efficiency of the communication software that drives the interconnect. Therefore, understanding of the performance capabilities of the cluster communication system provides the conceptual framework we need to understand the performance of the cluster system. Here are the main issues related to the performance understanding:

- Performance study - What sort of performance could we get on this platform? Knowing the baseline capability of the cluster system is important for performance understanding. Although this argument seems intuitive, it is an important issue on the commodity clusters, where we have the desire and flexibility to select the best configuration for our cluster systems. Performance is only a relative metric, thus, it has no meaning on its own. In the quest for performance, the primal objective is to have efficient utilization of resources. However, without understanding how well can the system performs.

    - How can we determine whether the performance of the program is acceptable?

- How to distinguish between efficient, cost-effective algorithms and inefficient algorithms?
- How can we identify the weakness of the program and perform performance tuning?

❑ Performance benchmarking - How can we quantify performance? Low-level architectural benchmarks [45] are the most appropriate because their results measure the basic capabilities of the hardware as well as the software that associated with it; and in principle, the performance of higher level programs should be predictable in terms of them. However, designing and interpreting benchmarks correctly can be difficult. This is because the conclusions drawn from a benchmark study depend not only on the basic timing results, but also on the way these are measured and interpreted. In particular, here are a few challenges in the use of benchmarking:

- Could the benchmark routine simulate the same execution condition as it is when the code path is used by real programs? For example, the behavior of cache memory can affect the accuracy of the benchmark measurement.
- Is the benchmark routine measuring something that has a measurable impact on overall system performance?
- Is the benchmark routine sensitive to changes in hardware characteristics? For example, a benchmark routine that is designed on a uniprocessor machine may not work accurately on a multi-way SMP machines; thus, we should have a clear knowledge on the capability of the benchmark routine.

Therefore, it is important to understand low-level details about architectural implementation when interpreting benchmark results. And we believe that, to assist the understanding process, the definition of the performance parameter should be accompanied with the benchmarking methodology that measures it.

❑ Performance modeling - Which performance features are important to our understanding process? Low-level benchmark is designed to measure the quality of a target hardware/software component. The measurements become the quantitative means for us to analyze or reason on the performance of a particular feature of the cluster system. These performance features, or in other words, the performance parameters, become the building blocks of our performance model, such that we try to understand the performance behavior of the application as a function of these performance parameters. We reckon that a useful performance parameter set - the abstract model, must possess these properties:

- Should accurately reflect the performance capability of existing machines.
- Should be able to evaluate the performance of a particular application on a particular platform and provide insights that can suggest possible improvements in the application.

- Should enable the programmer to predict the impact on the program's performance with regard to design changes in the algorithm or changes to the system configuration.

Therefore, we see that the choice of the performance features may influence the understanding process. However, choosing the right features to model and incorporating them simply, elegantly and accurately requires creativity as well as scientific methodology. This is because different applications exhibit different characteristics on different system settings or environments, and the application behaviors are often dynamic. Hence, it is impossible or irrational to expect any set of performance parameters to cover the whole application domain. Thus, the hypothesis becomes, if we have a minimal set of performance parameters that adequately model or describe the system, then the smaller the set is, the better its usefulness will be.

Performance Issues on Communication Traditional approach in assessing the performance of communication systems is by measuring the round-trip latency and the point-to-point bandwidth [20, 55]. However, these simple figures

- ❑ would not give as enough information to evaluate on the real performance of the communication systems;
- ❑ do not provide insight(s) on why these achievements were obtained;
- ❑ could not identify where are the potential bottlenecks of individual subsystems;

since these metrics only report of a single point-to-point measurement under a lightly loaded network. In fact, the network performance differs among applications and is depended on

1. Communication patterns - Besides the one-to-one pattern, example patterns are the one-to-many, many-to-one and many-to-many communications, which are commonly known as the *Collective Operations*. Collective operations play an important role in the development of parallel applications. Different patterns have different performance characteristics and requirements, and demand specific communication algorithms that can efficiently utilize the underlying resources to achieve optimal results.
2. Message sizes - Different factors dominate the communication performance for small, medium and large messages. For examples, with small messages, software overhead and network latency dominate the communication cost, while for large messages, network bandwidth is the more important factor in determining communication performance. For medium messages, they cannot completely amortize the software overhead as compared to long messages; therefore, they require better utilization of communication pipelines to hide away the software cost.

3. Communication schedules - The communication schedule characterizes how we express the communication pattern in terms of communication primitives, such as send and receive operations. By designing communication schedules, we are able to structure our communications to match with the resource constraints and yield highly efficient implementations.

4. Degree of contention - Mild contention may result in slight decrease in performance due to the higher queueing delays in the router. However, under prolonged contention, the network becomes heavily congested and this results in data loss.

All these communication issues contribute to the overall communication performance.

The performance problem related to the communication software has been an active research issue for the past decade. Currently, lightweight messaging systems [21, 58, 59, 82, 101, 109, 114] offer the best communication performance, as they create a fast communication path that bypasses the traditional in-kernel messaging protocol stack (e.g. TCP/IP), which is a serious obstacle in exploiting the high performance of modern network [9, 54, 55]. Although most of these lightweight messaging systems are successful in delivering the raw network performance to higher-level applications, their implementations, functionality and interfaces appear differently even though some are built on the same network technologies. There are issues that are not well addressed by these lightweight messaging systems, such as system behavior under heavy load and general purpose flow control. In particular, they lack of supports on guiding the development of high-level communication primitives atop of their lightweight messaging layer.

To better understand the network performance, we adopt the described systematic approach on performance understanding to explore how the underlying communication system supports these communication issues. In this dissertation, we make use of a communication model, which is derived on a resource-centric view of how data move across the abstract machine, to aid our quantitative and qualitative studies of the communication performance. This model is proposed as a "mid-level" tool, whose tasks are to map all high-level communication primitives properly onto the low-level architectural abstractions. Through a set of performance metrics, the programmers and system designers would understand the power, the strength and the weakness of their cluster communication system. In addition, by selecting appropriate metrics to analyze on the target application, this could bring out new insights as well as efficient quantitative prediction of the communication performance.

Limitations of LANs Having a low-latency communication system that drives the cluster interconnect in high speed is the prerequisite but not the guarantee of achieving high performance. For example, most existing commercial switches adopt packet drop policy on buffer overrun, which is a known performance problem under heavy congestion. Studies on LAN performance [55, 79] have shown that the high overheads in traditional communication protocols limit the ability to generate network load, thus the

observed contention problem was not critical. With low-latency communication mechanisms, applications can now generate higher load to the network, which could result in congestion drop problem even under a well-balanced communication schedule.

The behavior of the networks in response to congestion becomes an important issue in understanding the communication performance. Depend on the switch architectures, type of network technologies and the host communication software, networks react to congestion in different ways. For instances,

- Network routers that employ input-port buffering are known to suffer with the Head-Of-Line (HOL) [44] blocking problem, which is a serious constraint in achieving high-performance.

- Some Ethernet-based networks adopt the 802.3x link-based flow control standard [1]. However, it is a *non-selective* scheme [72], which has the control actions taken to resolve congestion apply to all traffics on that link rather than targeting to the true source of the congestion; therefore, congestion can spread to other parts of the network. Besides, different vendors have their own design strategies, thus, different switches may react differently even under the same contention situation [79].

- Most of the commercial switches or routers employ the Drop Tail discipline [36] as their buffer management strategy under the congestion situation. This is the simplest strategy as the router just drops all excess packets when the buffers are full. However, this is found unfair to bursty traffics, as more number of packets is dropped as compared to other sources.

With the use of commodity components as the cluster interconnect, the performance of the network depends on the type of network technologies (e.g. Fast Ethernet, Gigabit Ethernet, ATM, etc.), as well as depends on the internal architecture of a particular router or switch. This poses a challenge to the system designers and programmers on how to effectively utilize the communication network for high-performance communication.

1.2 Thesis Statement and Contributions

In this dissertation, we show that the key to effective utilization of cluster resources relies on a set of performance parameters, which eloquently delineates the performance capabilities of the target communication system. Through this set of performance parameters - the communication model, we can analyze, predict, evaluate and explain on issues related to high-speed communication on commodity clusters.

This communication model is a performance understanding tool, which captures the performance characteristics of the target machine and facilitates performance analysis as well as algorithm design. The model is based on a resource-centric view of the data flow through the abstract machine during a communication event. Most of

the performance parameters can directly map to events or operations found in traditional message passing programming model. In addition, each performance parameter is associated with a microbenchmark that measures experimentally its associated cost unit, e.g. time measurement in μs or clock cycle, that this parameter spends or takes to support the particular performance feature.

This dissertation exposes performance issues related to high-speed communication on commodity clusters. The principal contributions of this thesis are:

- ❑ We introduce a realistic communication model that supports performance understanding of the cluster communication system. This communication model becomes the foundation of this thesis research, which is used as a tool for supporting performance analysis and algorithm design.

- ❑ We provide a set of benchmark methodologies to quantify all performance parameters of our communication model. This communication model and its associated microbenchmarks make up as a tool for performance evaluation and analysis. As we believe that to assist performance understanding, the definition of the performance parameter should accompany with the benchmarking methodology that measures it.

- ❑ To study the congestion problem in high-speed networking, we exploit how congestion problem could affect the final performance from an architectural viewpoint. We show that how the buffering architectures of the switch interact with the communication protocol, and dominate the behavior of our lightweight messaging system under congestive loss situation. Our analytical and experimental results show that under asymmetric traffic loads, the output-buffered mechanism is more susceptible to the congestion loss problem than the input-buffered mechanism. During the modeling exercise, we identify salient features that enhance our understanding on the packet loss problem, and therefore, are relevant for the design and analysis of efficiency communication schedules.

- ❑ To avoid congestion loss, we have introduced a global congestion control scheme, which is a proactive approach in handling congestion. This scheme makes use of resource information provided by our model to prevent oversubscribing the network and avoid congestion loss. Through a global windowing concept, this scheme works on by collaborating all participating nodes to monitor and regulate the traffic load, which effectively avoids congestion loss and maintains sufficient throughput to maximize the performance.

- ❑ Based on the architectural model of the cluster system, we have devised an efficient communication schedule for complete exchange operation. The spirit of this algorithm is the node contention-free schedule operated at the packet level without explicit synchronization operation. This algorithm, the Synchronous Shuffle Exchange, effectively utilizes the communication pipelines and achieves high-performance. We make use of our communication model to show that this

algorithm is optimal. In the experimental evaluation, we demonstrate that this algorithm is realizable and efficient, as it can reach 97% of the available bandwidth on various hardware platforms.

1.3 Organization of the Thesis

This dissertation shows how a realistic model can help system designers and programmers to understand the performance characteristics of the underlying communication system. The organization of this thesis is as follows. Chapter 2 introduces our communication model in detail. We first describe the architectural model that our communication model is built on, as well as provide a breakdown view on how data flow through this architecture. Then we list out all the parameters of this communication model together with their associated cost formulae. The last section in Chapter 2 is focusing on reviewing other related models, and show how our work is distinguished from other works.

Chapter 3 provides a brief description on the Directed Point communication package [59], which is used as our tool to validate the thesis statement. Then we measure two of the Ethernet-based implementations of Directed Point and perform a systematic analysis on their performance. Through our modeling framework, we easily spot out the strength and weakness of these systems. In Chapter 4, we extend our performance studies from a point-to-point analysis to a highly congested communication pattern, the many-to-one collective operation. We focus on the congestion behavior of how the reliable transmission protocol performs under heavy congestive loss situation. The primary network architectural feature relevant to this study is the switch's buffering architecture, which is one of the performance parameters of our communication model. We conduct both experimental and analytical studies on two different buffering architectures, and investigate on how these buffering mechanisms impact on the resulting performance.

Chapter 5 and 6 highlight another capability of our communication model - algorithm design and analysis. In these chapters, we are focusing on another communication pattern, the many-to-many complete exchange operation. In Chapter 5, we discuss an efficient algorithm, the Synchronous Shuffle Exchange algorithm, for the complete exchange operation with our communication model. We make use of our model parameters to show that this algorithm is optimal on a theoretical non-blocking network. We also show that, in reality, the switch's buffering architecture may hamper the performance of our optimal algorithm, due to the demanding nature of this algorithm.

In Chapter 6, we extend our study on the complete exchange operation to another realistic LAN topology, the Hierarchical network [35]. And we show that there are architectural limitations on this topology. Again, by using the resource information provided by our communication model, we have designed a proactive congestion control scheme to augment the original synchronous shuffle exchange algorithm to work efficiently on this network topology. Chapter 7 gives a summary of this dissertation as well as discusses on directions for future work. Last but not the least, Appendix

A contains the full description of all benchmark methodologies of our communication model.

Chapter 2

Communication Model

2.1 Introduction

All parallel architectures have a common requirement - they all require a fast, high quality interconnect, thus, the unique property of parallel computer architecture is the communication system. Modern network technologies are catching up with the host system performance; however, the overhead of traditional communication protocol (TCP/IP) is so costly that the performance of communication systems lags behind the technological opportunities. This is the reason why lightweight messaging systems are designed to extract the best performance out of the boxes. However, they are relatively low-level from an end-user viewpoint. In addition, programs running directly atop of these communication systems are generally not portable across parallel platforms. Therefore, in general, these lightweight messaging systems are not used directly by the application programmer, but are better for supporting and optimizing high-level massage-passing programming interfaces such as MPI [38] and PVM [97].

Arguably or not, a scalable parallel machine must be in the form of distributed memory architecture. Although this architecture supports all known parallel programming paradigms, the best-fit programming paradigm on this architecture is the message passing paradigm. From the performance and portability points of view, designing parallel programs on top of portable message passing model would be the most effective way to harness the computing resources. Programmers simply express their parallel computations by using the high-level programming interface, whose implementation details are concealed from the end-users. By assumed that the implementations are indeed efficient, much of the burden of managing parallelism is lifted, thus, the end-users can focus on designing and optimizing the applications with respect to the programming interface. Based on this layer structure, we see that our communication model should be aimed as a “mid-level” model, whose tasks are to map all high-level primitives properly onto the low-level architectural abstractions.

A model is just an abstract machine, and therefore, we can have models at many different levels of abstraction [90]. For example, each programming language is a model, since it is an abstract interface that provides a simplified view of the underlying hardware to the programmers. However, due to the diverse nature of the parallel appli-

cations and parallel architectures, it is absurd to think that one model is good enough to represent the whole supercomputing domain. Since the idea of performance modeling is for algorithm designer to develop efficient algorithms for realizable, cost-effective architectures. Therefore, we believe that a realistic performance model (in our case, the communication model) must have its associated programming model as well as the corresponding architectural model, such that algorithms are designed or analyzed with the performance model, coded in the programming model and realized on the architectural model. The above classification of models was introduced by Heywood et al. [43], where they used it as a tool to classify different abstract models.

2.2 Commodity Cluster - Another Message Passing Machine

We agree on the visionary raised by Culler et al. [32] and McColl [66] that the architectural trend of parallel computers is converging toward the following generic model - *Most modern large-scale machines are constructed from general-purpose computers with a complete local memory hierarchy augmented by a communication unit, interfacing to a scalable network.* This architectural insight includes a broad spectrum of parallel architectures, which ranges from Massively Parallel Processor (MPP) machine to the PC and workstation cluster. However, this conceptual architecture is too abstract for building up our communication model. As in our modeling framework, the architectural model lies on the lowest level, which needs to be a flexible and rich model that clearly delineates the performance capability of the target architectures.

2.2.1 Architectural Model

In our model, a cluster is defined as a collection of autonomous machines that are interconnected by two networks, one is driven by standard (STD) communication protocol such as TCP/IP, and the other is for high-performance communication that runs under the lightweight messaging protocol (LMP). Both networks can be comprised of same network technology, but normally the first one is running on a low-end LAN technology while the LMP network is on a high-performance LAN or SAN (System Area Network). Parallel applications running on the cluster make use of both networks. The LMP network is used primarily as the communication network for the parallel applications, while the STD network is used as the control/backup network, e.g. job submission and dispatching, as well as cluster management. For non-parallel applications such as sequential and distributed tasks, they only make use of the STD network if communications between machines are needed. Thus, our communication model is focusing on the performance characteristics of the LMP network under parallel applications.

All cluster nodes are assumed to have the same local characteristics, such as computation power, memory hierarchy, operating system supports, and communication hardware. In particular, we assume that each network adapter equips with one set

of input and output channels, such that, it can simultaneously send and receive one data packet within one communication step. In general, parallel programs are programmed in SPMD (*Single Program Multiple Data*) mode, and follow the compute-interact mode of coarse-grained synchronization. As for the LMP network, it involves a global router/switch, interconnecting all nodes, and assuming a completely connected topology; hence, the network diameter between cluster nodes is not relevant under this architectural model.

For the global router, we assume that it supports an unreliable, packet-switched, pipelined network, and operates in full-duplex configuration under a fixed routing strategy. The logical unit of communication is packet, which is bounded to the range [1..MTU[1]]. To transmit long messages, the system transfers them as sequences of packets. Packets generated from any given node travel along a predetermined fixed route through the network in order to reach their destination. To support completely connected configuration, the router has a deterministic delay in routing a packet from any input port to the destination output port in the absence of conflicts, but the overall performance is affected by the network load. Conflicts take place if more than one packet need to access the same output line, and are resolved by always forwarding the first arrived packet; the rest are buffered and routed later in a FIFO (*First In First Out*) manner. The amount of buffer memory inside the router is assumed to be finite, and therefore, the network can sustain certain level of congestion. Since we assume the network is unreliable, upon heavy congestion and the buffer space is exhausted, the router simply discards excess packets.

At first instance, this global router concept [70] seems to be an unrealistic assumption as this limits the scalability aspect of the cluster systems. However, current network technology allows us to scale the switching system to support hundreds to thousand of nodes interconnected by a single router switch with more or less uniform point-to-point latency. For example, both Extreme Networks [71] and Cisco Systems [98] claim that their high-end chassis switches, BlackDiamond 6816 from Extreme Networks and Catalyst 6500 series from Cisco Systems, can support up to 192 Gigabit Ethernet ports. In particular, the BlackDiamond switch can be configured to support at most 1440 Fast Ethernet ports with wire-speed Layer 2 and Layer 3 switching for consistent performance.

As we believe that an ideal abstract model must correspond to a programming model so that algorithms are designed or analyzed with the abstract model and coded in the programming model. With this architecture, the best-fit programming model would be the Message-Passing programming model, such that processes running in parallel are communicating with one another by sending and receiving messages. In a broader sense, all processes run in private address spaces and cooperate by means of explicit message exchange. Therefore, to delineate the association between the abstract model and programming model, we draw our attention on how data are moving over this architecture.

[1]MTU stands for the Maximum Transfer Unit of the communication system.

2.2.2 Data Transfers

The data transfers have a significant impact on the communication performance, so optimizing this critical path is essential to obtain high-performance, and this becomes the main objective of all lightweight messaging systems. Despite of the vast difference in implementation choices and target network technologies, we observe that data transmissions are commonly going through three phases, from sender address space to receiver address space. A typical scenario would look as follows: the data on the sender side first traverse through the *send* phase, which is usually under the control of the host processor. Then the messages are delivered by the network hardware (network interface cards (NICs) and switches) to the other end during the *transfer* phase of the transmission. Eventually, at the other end, the *receive* phase terminates this transmission by delivering the data to the corresponding receive process.

Send Phase This phase includes all events happened before the network adapter takes over the control in handling the data transfer. We consider this phase starts from the time when the user process is ready to send its message, to the time when the data message is transformed to a hardware dependent format that is ready to be transmitted by the network adapter. With traditional communication protocol, this includes system call handling, cross-domain data copying, checksum computing, protocol stacks traversing, and other processing overheads. However, for all lightweight messaging systems, all those unnecessary overheads have been removed so as to shorten the software gap between the user process and network hardware. Regardless of the optimization techniques used in this phase, all LMP systems must take care on how to "move" the data message from the sender's memory space to the network adapter's address space.

One major difference between these lightweight messaging systems is the obligation of the host processor with respect to the multiplexing task. To support multiprogramming, the LMP systems are required to efficiently multiplex multiple message streams, and deliver these intermixed data packets to the corresponding destinations. For network adapters with a programmable communication co-processor, some LMP systems off-load the multiplexing task to the co-processor. This allows the host processor to do other computation, and thus, effectively reduces the software overhead. On the other hand, without the help from a communication processor, the multiplexing task becomes the sole responsibility of the kernel due to the protection and integrity reasons, and therefore, the observed software overhead would be higher.

For example, BIP [83] is a user-space communication library built on top of the Myrinet network [15]. To deliver near raw speed performance (i.e. the achieved one-way latency is 4.3 μs for an empty message), it simply exposes the network interface to a single user-space process and relies solely on the Myrinet hardware for the reliability. On the other end, PM [101] is another user-space communication library running on Myrinet; it supports multiprogramming as well as adopting flow-control mechanism to avoid receive-buffer overrun. Same as BIP, it relies on the Myrinet hardware to guarantee the data delivery between NICs. Its one-way latency is 7.2 μs for an 8-byte

message. Therefore, we reckon that the overall cost of this send phase could become a performance index, which reflects the functionality of the lightweight messaging protocol as well as the processing capability of the network hardware.

Transfer Phase The transfer phase is wholly a network-dependent part. This phase involves all the events happened in transporting the data through the network to the remote peer. In general, this includes two DMA transfers, one in the host node and the other in the remote node, together with the physical transmission of the data through the wires and switches. Factors attribute to the movement cost of this phase are:

- The I/O bus bandwidth, e.g. PCI, VME, MicroChannel, SBus, etc. For example, Tezuka et al. [101] have reported of the significant difference in the measured network bandwidth between SBus and PCI bus on the Myrinet networks that were driven by the PM protocol.
- The network transmission speed, e.g. 100 Mb/s with Fast Ethernet, 1 Gb/s with Gigabit Ethernet, 2 Gb/s with Myrinet-2000, 10 Gb/s with 10 Gigabit Ethernet, etc. The higher the transmission speed, the shorter the network delay.
- The network interface protocol or firmware, e.g. Myrinet-interface control program, QsNet driver, etc. With the use of programmable network interfaces, more functions can be included, e.g. address translation and reliable support, which significantly reduces the software overhead. On the other hand, this increases the network delay when more complex network interface protocol is used, e.g. AM II [23].
- The network topology, e.g. multistage interconnection network, crossbar network, irregular network, etc. This affects the distance or diameter between the cluster nodes, which in turn affects the transmission delay on the network.
- Underlying switching technique, e.g. store-and-forward switching, virtual cut-through switching, wormhole switching, etc.

It is a non-trivial task to capture all these factors as one has to know all the details of the underlying hardware, which overwhelms the performance understanding process.

Despite of having many factors contribute to the transfer cost, the prevailing factor is the network technology to which these LMP systems are targeting. This is because once we have decided on the network technology, we are confined to the technological constraints of the underlying network. For example, performance of the network is more heavily influenced by the switching technique than by the topology or the routing algorithm [35]. However, most of the Fast Ethernet or Gigabit Ethernet switches are using store-and-forward packet switching. Therefore, those LMP system running on these networks are known to have higher network delay, albeit supporting a larger non-blocking network. In contrast, the Myrinet switches are using wormhole switching, we are therefore expecting a very low hardware latency, e.g. ~0.1μs per switch latency [69], which significantly improves the point-to-point communication performance.

Besides of the hardware delay, the adopted switching technique also affects the way in which the switch handles network conflict. For examples, store-and-forward switch uses buffer queue to store the whole packet, while wormhole switch applies link-by-link flow control to stall the transmission instead of buffering the packet. Although these two techniques both increase the network delay, they give completely different impression to the end-user. For the store-and-forward buffering, the end-user would experience an increase in the per packet network delay, while for the wormhole switching, the end-user would experience an increase in per message injection rate. Nevertheless, these minor differences may affect the design choice in devising efficient communication algorithms.

Receive Phase This phase includes all events happen at the destination node when message is handed over by the network adapter, until it is being dispatched to the corresponding receiver's process space. It looks as if the receive phase is just the opposite of the send phase, but it is not. In the data transfer event, message reception is an asynchronous event as receiving process does not know when will a message arrive. To dispatch the data to the right place, i.e. the reverse of multiplexing, the system needs to know where the data should go. This has to be managed by a privilege process that gets hold of those protected informations, and this becomes the major difference between different LMP implementations. For those LMP systems that do not make use of any communication co-processors, this becomes the sole responsibility of the operating system. To notify the kernel on the message reception, the network adapter needs to raise a hardware interrupt. However, interrupts are high priority events, they have great performance impact on modern CPU architecture, such as flushing pipelines and reducing cache locality.

Even though with the availability of communication co-processor, different LMP systems take different strategies on the message reception. Some LMP do not take advantage on the co-processor, and use the in-kernel approach, e.g. AM-II [23] and FM [75]. Others program the co-processor to move the incoming message directly to a pre-pinned and pre-translated user-space address, e.g. PM [102] and LFC [12], or using a more aggressive approach that integrates a translation look-aside buffer into the network interface to pin and translate user-space addresses dynamically, e.g. UNet/MM [115]. Moreover, no matter which protocols they are using, the performance of this phase is greatly affected by the processing speed of the protocol processor, the lightweight messaging protocol, and the I/O bus bandwidth. It is found that this phase is most likely to be the bottleneck of the whole transmission path, especially if the processing speed is not fast enough to drain-off all incoming messages.

To summarize, we present a breakdown analysis of several user-level or kernel-level lightweight messaging systems in Table 2.2, and categorize their activities with respect to the three-phase data transfer. We observe that different LMP systems have different massing-passing semantics and implementations, and this is being reflected by the differences in involvement within the three phases. However, their events or activities are generally captured by this three-phase data transmission scenario.

	AM II [23]	FM2.x [58]	BIP [83]	PM1.2 [101]	UNet/ATM [108]
pre-communication setup	endpoints creation	context creation		context creation	endpoints creation
possible per-communication overhead	endpoint faults		pin send & receive buffer	gang scheduling overhead	copy-to-DMA
			synchronize on long message	pin-down cache overhead	
send phase	library call	library call	library call	library call	library call
	Short: PIO / Long: copy-to-DMA	acquire device lock	Short: PIO		PIO metadata
	receive polling	PIO with write combine			
transfer phase	endpoint management overhead	Myrinet transfer	DMA to NIC	DMA to NIC	DMA to NIC
	DMA to NIC	DMA from NIC	Myrinet transfer	Myrinet transfer	ATM transfer
	Myrinet transfer		DMA from NIC	DMA from NIC	DMA from NIC
	endpoint management overhead				
	DMA from NIC				
receive phase	polling or interrupt	polling	polling	polling	polling
	PIO / copy-from-DMA	copy-from-DMA	PIO	copy-from-DMA	copy-from-DMA

	GigaE-PM2 [95]	GAMMA [22]	DP	UNET/FE [114]
pre-communication setup	context creation	active port setup	DP endpoint creation	endpoint creation
possible per-communication overhead	acquire send buffer	pin send & receive buffer		copy-to-DMA
send phase	copy-to-DMA	lightweight system call	lightweight system call	lightweight system call
	system call		copy-to-DMA	
transfer phase	DMA to NIC	DMA to NIC	DMA to NIC	DMA to NIC
	GE or FE transfer	GE or FE transfer	GE or FE transfer	FE transfer
	DMA from NIC	DMA from NIC	DMA from NIC	DMA from NIC
receive phase	interrupt or polling	interrupt	interrupt	interrupt
	copy-from-DMA	copy-from-DMA	copy-from-DMA	copy-from-DMA
	copy-to-user		copy-to-user	copy-to-user

Table 2.2: Breakdown analysis of several lightweight messaging systems with respect to the three-phase data transfer. During the data transfer, we assume that the outgoing message is resided on any virtual address space of the sender process, and is going to be delivered to any virtual address space of the receiver process.

2.2.3 The Communication Model

A practical way to understand the performance characteristics of a parallel program on a parallel architecture, is to look from a processor's viewpoint at the different components of time spent executing the program [32]. With each component of time spent reflects the software-*plus*-hardware performance of a specific architectural feature, a collection of these components becomes a set of performance indices that related directly to the performance understanding issues. In the previous section, we have laid out the architectural model of the cluster platform as well as the typical scenarios involved in moving data over this architecture. Since data communication lies on the critical path of a message-passing machine, the straight-forward way in performance understanding is to identify abstract components that characterize this critical path - the data transfer.

To characterize the transfer, a communication model is associated with our architectural model, which delineates the costs induced by moving the data around, both locally and remotely. We consider data communication via the network as an extension to the memory hierarchy concept, such that it is a data movement from the remote memory region to the local memory region, or vice versa. There may have two types of data movements in a communication event: a) remote data transfers and b) local data transfers. It is important to include the local data transfer abstractions to our performance model. This is because, in the associated message-passing model, we are not restricting on the "whereabouts" factor of the data messages. Since data messages can be resided in anywhere of a process address space, and some communication systems require the placement of these data messages be in some pre-defined memory region, e.g. UNet and GigaE-PM2, thus, local data movements are included so as to accomplish this programming abstractions.

Remote Data Transfer We encapsulate all the overheads of the three-phase data transfer by a set of model parameters, and below are the detail description of individual parameters. The associated microbenchmarks for deriving the cost functions of this parameter set are provided in Appendix A. In addition, Figure 2.1 summarizes this abstract model as a schematic drawing that delineates the relations of different performance parameters to the architectural model.

Machine size p This refers to the number of processes participating in this communication event.

Send overhead $O_s(m)$ This parameter stands for the software overhead associated with the send process for sending an m-byte data packet. From the high level perspective, we view it as the time used by the user process to interact with the logical network interface, prepare the message, forward the message to the network adapter, and signal the network hardware. Therefore, it encapsulates all the events happened in send phase. This parameter captures several performance features of the communication system, such as the speed of the host processor, the efficiency of the memory

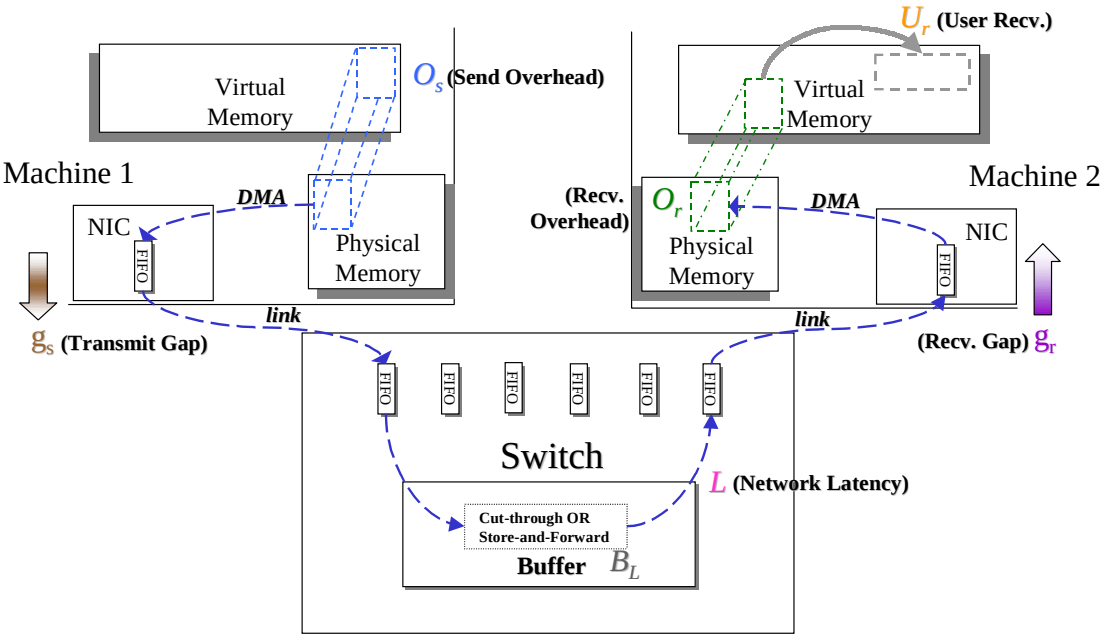


Figure 2.1: Model parameters affiliated with the remote data transfer

subsystem, and the lightweight messaging protocol in use. We model this parameter by a simple linear function,

$$O_s(m) = \kappa_s + \tau_s m \tag{2.1}$$

where κ_s is the startup cost of this event which depends on the node processing power, *m* is the message length bounded by the range [1..MTU], and τ_s is the data transfer rate that depends on the efficiency of the memory subsystem. Subject to the particular LMP protocol, for example, under zero memory copy, this linear function can be reduced to a simple constant, i.e. $O_s(m) = \kappa_s$. As this is a synchronous event, we quantify this parameter by directly measuring the time engaged by the processor in handling those activities.

Inter-packet transmit gap $g_s(m)$ Owing to the difference in data movement speeds between the send and transfer phases, the network hardware is moving data packets with a confined capacity, which is captured by this parameter. For example, the performance of the I/O bus and the network technology are the major factors of this parameter. This inter-packet gap has two slightly different meanings with respect to different perspectives, but in general, it delineates the maximum network throughput available to the user process. From the user process perspective, it views the gap as the minimum service time of the network in transmitting consecutive data packets. Thus, any attempt to send data faster than this gap yields no performance gain, and the differ-

ence between g_s and O_s indicates the amount of CPU cycles available for the processor to do other useful computation or perform the receive operation during bidirectional communication. From the network perspective, it represents the maximum injection rate of the packets to the network. This parameter is delineated as

$$g_s(m) = g_1 + \tau_1 m \tag{2.2}$$

where g_1 stands for the startup cost necessary to initiate the transfer and τ_1 reflects the available communication bandwidth provided by the I/O bus and the network.

Network latency $L(m, p)$ This parameter represents the time used by the network to move an m-byte data packet from the physical memory of the source node to the physical memory of the destination node. It is a network-dependent parameter, which encapsulates the performance of the host I/O bus, the network topology, the network technology and the network interface protocol (or firmware) in use. Since we model the network as a complete graph, the topology and the distance factors can be eliminated. In general, the value of *L* is subjected to the traffic loading at any particular instant in a real network. For example, when routing the packets through the network, conflicts take place if more than one packet accesses the same output line, and temporary buffering is needed. This delay affects the overall network performance perceived by the users. The amount of buffer memory inside the switch is assumed to be finite, thus, the network can sustain certain level of congestion. We model this phase by a bilinear function under the congestion-free condition,

$$L(m, p) = l(p) + m\tau_L(m, p) \tag{2.3}$$

where $l(p)$ is a function representing the cumulative startup cost of this network transfer and $\tau_L(m, p)$ is the available network throughput. Both *l* and τ_L are a function of *p*. Routing a packet involves utilization of some central resources (e.g. buffer control unit and forwarding control unit), therefore, contention for resources may occur if more than one routing request happen concurrently. The extent of this contention is subjected to the switch internal architecture, and different routers may behave differently. Some networks may only have limited aggregate bandwidth, they cannot service too many communicating pairs at a time and contention arises. So the allocated network throughput to the transfer phase depends on the aggregate bandwidth of the network, the number of communicating pairs and the volume of the communication, thus, τ_L is a function of both *p* and *m*.

Inter-packet receive gap $g_r(m)$ This parameter stands for the minimum time interval between two consecutive receptions experienced by the receiving host, which is limited by the performance of the I/O bus and the network technology in use. Similar to the g_s parameter, it is used to delineate the maximum packet arrival rate delivered by the network or the maximum service rate of the network hardware. This parameter has two uses. First, the inter-packet receive gap reflects the CPU cycles available to handle arrived packets, so this information should be taken into consideration during

LMP design. For example, should we implement an address translation mechanism to support true "zero-copy" on the receive path, or simply using a one-copy semantic? We can make use of the receive gap parameter to justify on various design trade-offs. Second, as we cannot receive more than one packet within the receive gap, this information is particularly useful in designing communication schedules. For example, we can use this information to schedule a collective operation, which involves both send and receive events.

Same as the g_s parameter, this receive gap is captured by a linear function,

$$g_r(m) = g_2 + \tau_2 m \tag{2.4}$$

For simplicity, on a homogeneous cluster, we can generally assume $\tau_1 = \tau_2$, as both are related to the transfer capability of the network and the I/O bus.

Network buffer capacity B_L Resource contention is the major cause of congestion, which in turn, affects the delay experienced by the applications. In reality, congestion is a fact that we need to face with. The B_L parameter corresponds to the available buffer capacity in the global router, which is a measure of the network tolerance of the router in handling contention. For a router/switch, we only have one B_L value, either it is associated to the whole switch if it is a shared-buffered switch, or is associated to a switch port if it is an input-buffered or output-buffered switch. By capturing the finite capacity of the network buffers, algorithm designers can calculate the network endurance, and avoid congestion loss with appropriate communication schedules.

Asynchronous receive overhead $O_r(m)$ This parameter captures the software overhead in handling incoming messages. As there is no central coordination between communicating parties, and the messages can be arrived at anytime on packet-switched network; thus, message reception is considered as an asynchronous event which does not involve the receiving process. This parameter captures the costs of all kernel events including interrupt, memory copy and context switch, and its efficiency is affected by the processing speed of the processor and the lightweight messaging protocol in use. In our model, we express it as a linear equation,

$$O_r(m) = \kappa_r + \tau_r m \tag{2.5}$$

In which κ_r represents the minimal cost of this asynchronous event, such as interrupt cost, buffer management, and protocol overhead; while τ_r mainly reflects the speed of memory movement between different memory regions if needed.

User receive overhead $U_r(m)$ Due to the asynchronous nature of the communication, the receiving process needs to find some means to check for data arrival, e.g. polling, block & wake-up by signal, or hybrid approaches; and consumes the data, e.g. copy to other memory segment. This parameter reflects the software overhead spent by the receiving process after arrival of packets. In most of the performance evaluation

reports, due to the artificial nature of the benchmark programs, this parameter is of insignificantly low cost. However, in reality, this overhead reflects the performance loss due to improper coordination of communication events. For example, in a multi-user environment, polling is a user-level event that is affected by the regular CPU scheduling policy. If the receiving process cannot be scheduled frequently to poll for its data, the overall performance may degrade significantly.

Local Data Transfer

Memory copy overheads M_{ctc}**,** M_{ctm}**, &** M_{mtm} Memory copy issue has been extensively studied in the past, and is being classified as a high overhead event. To avoid this overhead, most of the low-latency communication systems have removed it from their protocol stacks. However, in reality, memory copy operations cannot be avoided completely. For example, some high-level communication schemes such as *Gather* and *All-to-all*, require to have the resulting messages be returned in contiguous memory buffer. Thus, extra memory copy operations are needed. To quantify these costs, we provide three memory copy parameters - M_{ctc}, M_{ctm}, and M_{mtm} to represent the costs induced by data movement between different memory hierarchies, such as the cache-to-cache, cache-to-memory, and memory-to-memory data movement.

2.2.4 Simple Examples

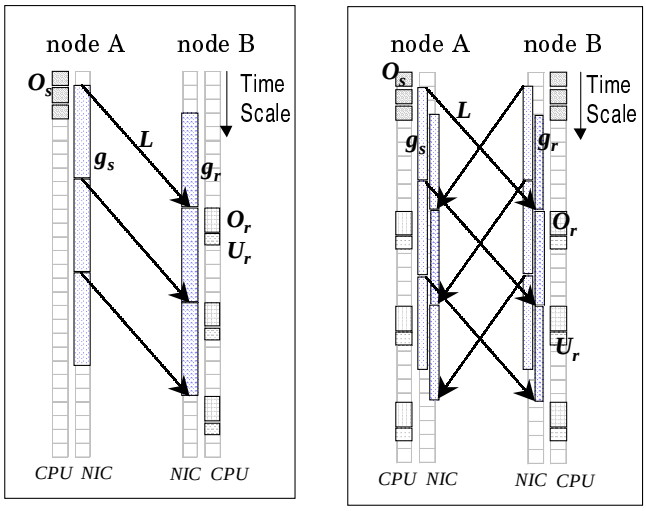
Point-to-point communication For a homogeneous cluster, we generally assume that $g_s \approx g_r$ and simplify the expression by $g = \max(g_s, g_r)$. To take advantage of the full-duplex communication, we assume that the cluster communication system satisfies this condition, $(O_s + O_r + U_r) < g < L$. This assumption is generally true under the current microprocessor technologies and the adoption of low-latency communication. As a result, under no conflict, the one-way point-to-point communication cost (T_{p2p}) in transferring an M-byte "long" message between two remote user processes is:

$$T_{p2p}(M) = O_s + (k-1)g + L + O_r + U_r \tag{2.6}$$

where $k = \frac{M}{b}$, which corresponds to the fragmentation of an M-byte message to *k* data packets of size *b* bytes. For optimal performance, *b* usually stands for the maximum transfer unit (MTU) of the underlying network technology. Similarly, the bi-directional exchange communication cost (T_{xchg}) between two nodes can be expressed as:

$$T_{xchg}(M) = O_s + (k-1)g + L + O_r + U_r \tag{2.7}$$

It is interested to see that both one-way and two-way exchanges have the same cost. Figure 2.2 presents the graphical breakdown of these point-to-point communications and shows that when full-duplex condition is met, both send and receive phases can be happened concurrently without interfering each other. Although O_r is a high-priority event which always preempts other activities, as we have $(O_s + O_r + U_r) < g$,



(a) uni-directional flow (b) bi-directional flow

Figure 2.2: Point-to-point communications with $(O_s + O_r + U_r) < g$

the processor is capable to perform both send and receive activities within one g interval. Thus, we see that the bottleneck of the communication falls on the network component. In theory, their costs would remain the same as long as the bi-directional network throughput is within the I/O bus throughput constraint. The next example will focus on how the assumption of the full-duplex condition affects other high-level communication issue.

Broadcast operation This one-to-many communication pattern appears in many parallel and distributed applications, thus, it is included in most of the high-level massage-passing libraries, e.g. MPI. Most of these libraries implement the broadcast operation on top of the point-to-point primitive with the tree-based algorithm. To broadcast a long message of size *M* bytes (larger than the MTU), we simply repeat the broadcast algorithm $\frac{M}{MTU} = k$ times. Based on our communication model, we have graphically constructed this broadcast communication tree and present it in Figure 2.3. The corresponding cost formula expressed in our model terminology is:

$$\begin{aligned} B_k(p) &= ctm(M) + O_s + g(k-1)log_2p + (L + O_r + U_r + O_s)log_2p - O_s \\ &= ctm(M) + log_2p\,((k-1)g + L + O_r + U_r + O_s) \end{aligned} \quad (2.8)$$

The above formula models the communication cost of sending an M-byte message that resides in an arbitrary virtual address on node 1 to *p-1* nodes. Although, with this parameter set, the full-duplex condition is not reached[2], we find that there is no inter-

[2] U_r includes an mtm copy so as to reconstruct an M-byte message before forwarding to the user process

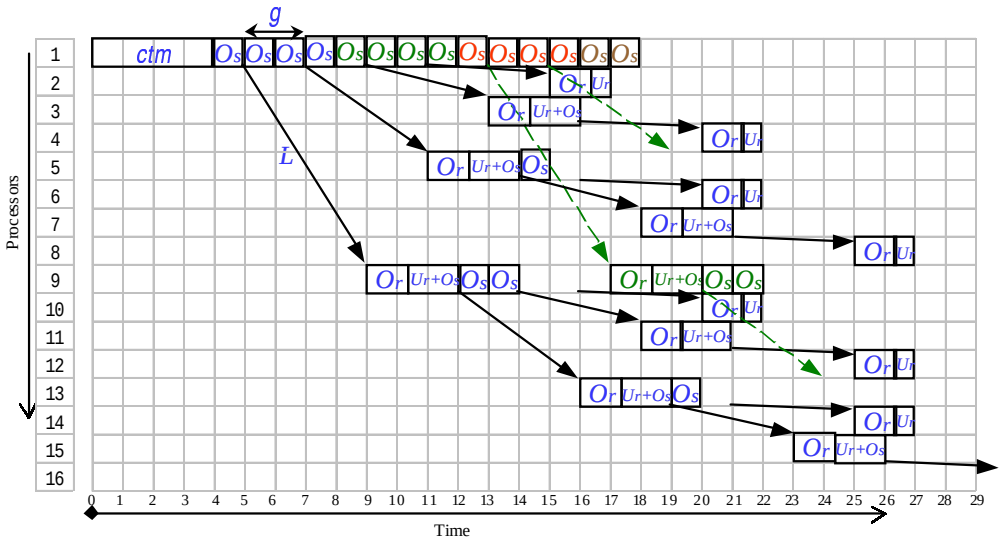


Figure 2.3: Tree-based broadcast operation expressed in our model terminology with $p = 16$, $L = 4$, $g = 2$, $O_s = 1$, $O_r = 1.5$, $U_r = 0.5$; however, the full-duplex constraint cannot be satisfied.

ference between the send and receive events. In addition, we notice that the broadcast root is the bottleneck of this communication schedule. Nevertheless, since no interference exists, this derived cost formula should cover other cluster configurations that satisfy the full-duplex condition.

Figure 2.4 delineates the same broadcast tree that works on a different cluster, which exhibits different characteristics. From this broadcast tree, we identify that the bottleneck region of this broadcast pattern is at the $\frac{p}{2}^{th}$ processor, but not at the broadcast root. Of the previous broadcast communication schedule (Figure 2.3), we find no interference exists in successive broadcast, such that the reception of the i^{th} packet does not overlap with the transmission of the $(i-1)^{th}$ packet to its subtree. But under the new cluster setting, as shown in Figure 2.4, at processor 9 - the $\frac{p}{2}^{th}$ processor of this broadcast event, the transmission of the i^{th} packet is always overlapped with the reception of the j^{th} packet for any $j > i$. With this observation, the new cost formula of this broadcast pattern under this new cluster setting becomes:

$$\begin{aligned} B_k(p) &= ctm(M) + O_s + L + (k-1)\cdot(O_r + U_r + (log_2 p - 1)\,O_s) \\ &\quad + O_r + U_r + O_s + (log_2 p - 1)(L + O_r + U_r + O_s) - O_s \\ &= ctm(M) + (k-1)\cdot(O_r + U_r + O_s + (log_2 p - 2)O_s) \\ &\quad + log_2 p\,(L + O_r + U_r + O_s) \\ &= ctm(M) + (k-1)\,(T_0 + (log_2 p - 2)O_s) + log_2 p\,(L + T_0) \\ &= ctm(M) + (log_2 p + k - 1)T_0 + L log_2 p + (k-1)(log_2 p - 2)O_s \end{aligned} \tag{2.9}$$

where $T_0 = O_r + U_r + O_s$

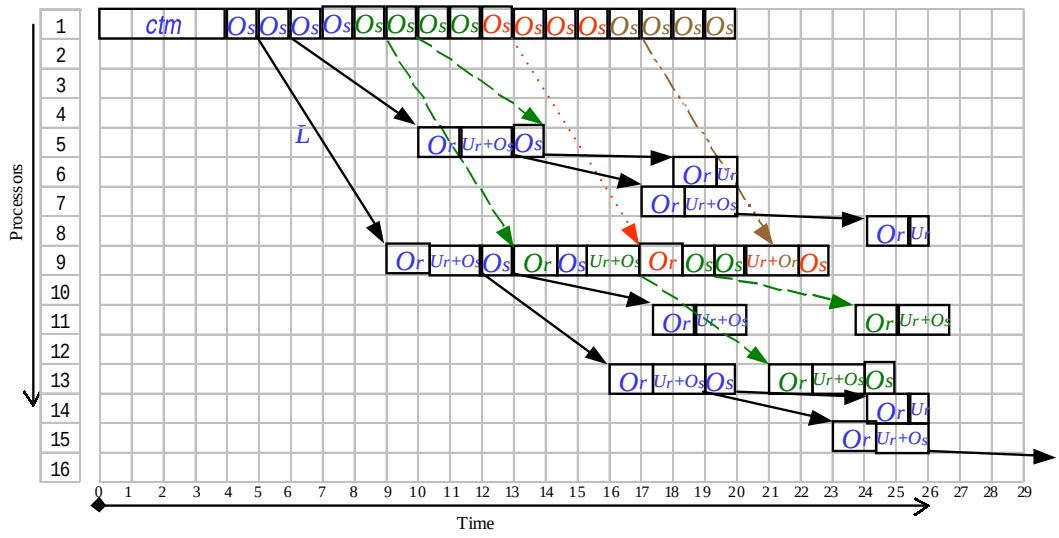


Figure 2.4: The same broadcast schedule on another cluster configuration with g reduced from 2 to 1

When comparing both cost formulae, we find that the extra overhead induced by the interference between successive broadcast is proportional to the size of the broadcast message (k). Hence, when k becomes large, the observed delay on this broadcast operation becomes longer. Interestingly, one can determine which cost formula is more appropriate to their cluster configurations by simply check on the parameter set. Observed that no interference occurs whenever the arrival interval between successive broadcast is longer than the software handling cost, thus

$$\begin{aligned} & O_r + U_r + O_s + \left(log_2\frac{p}{2} - 1\right) * max(g, O_s) < log_2 p * max(g, O_s) \\ \Rightarrow \quad & O_r + U_r + O_s < 2max(g, O_s) \end{aligned}$$

From this derivation, we conclude that if $(O_r + U_r + O_s) < 2g$ when $O_s < g$, or $(O_r + U_r) < O_s$ when $g < O_s$, cost formula (2.8) is more appropriate to estimate the cost of this broadcast operation. Otherwise, use cost formula (2.9).

2.3 Related Models

The lack of the central unifying model in parallel computation as compared to the unique von Neumann model in the field of sequential computation has resulted in a long debate of selecting the representative model(s) for parallel computation. Consequently, there exist enormous number of models for parallel computation. Some of them look different, but were shown to be quantitatively equivalent [14]. Some of them look similar, but they could be completely different. In general, models have tended towards undesirable extremes. On the one hand, they are of highly theoretical qualities

but be unrealistic or difficult to map onto real machines. At the other extreme, models may be too machine-oriented or complex which limit their long-term usage and portability.

Bear in mind that the definition of the model is "*simply an abstract view of a system or a part of a system, obtained by removing details in order to allow one to discover and work with the basic principles*" [43]. On the other hand, the complexity of designing and analyzing parallel systems requires that models be used at various levels of abstraction that are highly related to the application characteristics. Strictly speaking, it is hard to make a head-to-head comparison on models as they may involve different levels of abstraction. As our quest for a performance model is to have effective exploitation of commodity clusters for high-performance computation, so we focus our exposition of models that are based on the same architectural foundation as compared to our communication model, as well as targeting to a similar programming abstraction as we are.

The BSP Model It is the first to be called as bridging model [105]. Essentially, it agrees with the generic architectural model described above, but requires an extension that provides efficient global synchronization on all processors. The performance of the BSP-style program can be characterized by three parameters: *p*, *L* and *g*, where *p* stands for the number of processors; *L* is the cost of global synchronization in unit of time step; and *g* corresponds to network throughput in terms of the ratio between the number of local computational operations performed per second by all processors, to the total number of data words delivered per second by the router. A parallel algorithm is expressed in BSP model as a sequence of parallel supersteps. Each superstep consists of a sequence of local computation steps plus any message exchanges, followed by a global synchronization. The cost of a single superstep phase is represented by the formula $w + gh + L$, where *w* is the maximum cost of the local computation on each processor, and *h* is the maximum number of packets both sent and received by any processor. The cost of the parallel algorithm is just the sum of each individual superstep cost that comprising the algorithm.

Although BSP model does not explicitly stress on data locality, the *gh* parameter shows us how the importance of data distribution (locality) in influencing performance. Furthermore, the *gh* parameter implicitly captures the contention issue but inadequately, as in reality, *g* may be affected under congestion condition. One important performance feature it has missed out is the communication cost related to the message size, as it does not distinguish between a message of length *kb* and *k* messages of length *b*, but in reality, this could be a significant factor to the performance prediction and analysis. Another limitation of this model is the restricted framework - supersteps, as some parallel applications cannot fit into this programming structure, e.g. task-parallel model. The global synchronization operation between supersteps would impose a stringent requirement on the cluster communication system. This is because almost all commodity clusters are not coupled with hardware synchronization primitive, any realization of the global synchronization has to be done by software approach, which means it would contend with normal data communications. Thus, this

becomes a costly overhead, especially this operation appears once in each superstep.

The LogP Model The LogP model [30] tends to be more empirical and network-oriented. Its includes four parameters to characterize the system: *L*, o, g and *P*, where *P* stands for the number of processors involved; o represents the software overhead associated with the transmission/reception of message; *L* is the upper bound on the hardware delay in transmitting a fixed but small size message between two endpoints; and g is the minimal time interval between consecutive messaging events at a processor, which corresponds to the network throughput available to the processor. By simply exposes these architectural parameters, we can directly derive the performance/cost when using it to analyze parallel algorithms.

An interesting concept of LogP model is the idea of finite capacity of the network, such that no more than certain amount of messages ($\left\lceil \frac{L}{g} \right\rceil$) can be in transit from any processor or to any processor at any time. And any attempts to exceed the limit will stall the processor. However, the model does not provide any clear idea on how to quantify, avoid and take advantage of this information in algorithm design. Similarly, LogP model does not address on the issue of message size, even the worst is the assumption of all messages are of "small" size; however, this has been addressed on their follow-up study [31].

Despite of the shortcoming, this model is the pioneer model that breaks the synchrony of parallel execution as oppose to the PRAM model [37], even though it is not the first to do so. Consequently, other studies tried to extend its capabilities to support more constructive features. For examples, LogGP model [4] augments the LogP model with a linear model for long messages; LoGPC model [67] further extends the LogGP model to include contention analysis using queuing model on the *k*-ary *n*-cubes network; LogPQ model [103] augments the LogP model on the stalling issue of the network constraint by adding buffer queues in the communication lines.

The Postal Model The Postal model [8] is similar to LogP model with the exception of more abstractly expressing the network. The system is characterized by two parameters: n and λ, where n stands for the number of processors and λ represents the communication latency. The communication latency λ is expressed as a ratio between total time spent in transmitting the message from sender to receiver with the time spent by the sender in initiating the transfer. This ratio captures both the software and hardware costs, and effectively reduces the dimension of analysis. Similarly, to simplify the analysis, this model sacrifices the performance accuracy by neglecting the importance of message size over communication latency. Therefore, their cost models are better for asymptotic analysis than for prediction, when porting the resulting algorithm to a particular platform, significant efforts have to be made for tuning the algorithm for performance.

The C^3 Model The C^3 model [41] comes with the BSP superstep notion and also requires to have synchronization events between supersteps. However, on the selection

of the performance parameter set, this model adopts a tactic that lies midway between the Postal and LogP models, which explicitly expressing the costs spent in sending and receiving messages. The unique feature of this model is the introduction of the congestion measures to the performance set, which measure the congestion over communication links (C_l) and congestion at the processors (C_p). The authors admitted that congestion is difficult to evaluate, and they approached this problem by a rather phenomenal way.

Observed that congestion depends on the total amount of data sent between all processor pairs (*cong*). This model relates the link congestion by simply estimated the cost as the per-processor delay in routing L_a packets across the bisection width (*b*), which is shared by all processor pairs (*cong*), i.e. $C_l = L_a * \lceil \frac{cong}{b} \rceil$. And the processor congestion is estimated as $C_p = L_a * \left\lceil \frac{cong}{p} \right\rceil * h$, where *h* is the average distance between processors. Their rationale is that a message of size L_a traversing a distance h links would compete for the resources with other messages at each of the $h-1$ intermediate processors, therefore is slowed down by a factor of $\left\lceil \frac{cong}{p} \right\rceil$ at each processor. However, it is easy to find out that these congestion measures are quite unrealistic. For example, the many-to-one and one-to-many communications suffer with the same degree of link and processor congestion, which is obviously not true in real networks.

The CCM model Motivated by the inadequacy of BSP model and the restrictive framework, the Collective Computing Model (CCM) [86] transforms the BSP-superstep framework to support more high-level programming model, such as MPI and PVM. Although CCM follows the superstep terminology of the BSP model, it waives the requirement of global synchronization between supersteps, but combines the message exchanges and synchronization properties into the execution of a collective communication function. As a result, this model provides a finite set ($\vec{F}$) of collective communication functions, which sincerely maps to the collective operations found in MPI. Besides, it also provides a set ($\vec{T}_F$) of cost functions for each collective function in $\vec{F}$, such that performance analysis can be made on these cost functions.

As this model is aiming for a higher level programming model, its abstraction is more closely resembled to those common high-level message-passing programming interfaces; therefore, it consists of a larger set of performance parameters. As these performance parameters are directly related to some concrete operations, quantitative analysis is therefore possible, and the prediction quality is usually high, albeit the intricacy of the analysis. Besides, the parameter set can be a useful tool for evaluating message-passing software. However, this approach only contributes minimally in designing efficient message-passing library as they cannot provide information to guide on the design process, as they assume that the abstract machine supports these high-level primitives. Since this model is oriented to a high-level model, it can actually be built atop of existing abstract models. For example, to derive the $\vec{T}_F$ set, one can measure the performance of those collective operations directly out of the boxes, or we can determine $\vec{T}_F$ from LogP model or from C^3 model.

Phase Parallel Model This model [118] is similar to the BSP model, but functionally closed to the CCM model. Under this model, a parallel execution is divided into sequence of phases. The next phase begins only after all operations in the current phase have finished, however, there are no synchronization primitive to enforce this synchrony. There are three types of phase: (1) Parallelism phase - performs process management; (2) Computation phase - executes local computation; (3) Interaction phase - executes interaction operation. There is no stringent framework in confining the sequence of phases, such that an interaction phase or another computation phase can follow a computation phase. However, different interaction operations, e.g. point-to-point communication or collective communication, may take different times. There is a general cost formula for an interaction operation:

$$T_{interact} = t_0(n) + \frac{m}{r_\infty(n)}$$

which reflects that the cost of interaction depends on the message length (*m*), startup overhead ($t_0(n)$) for an operation involves n processors, and the asymptotic bandwidth ($r_\infty(n)$) under this communication profile. Likewise, to derive these formulae for different interactions, the authors performed direct measurements on the target machines [117].

Table 2.3 highlights these abstract models according to the modeling framework outlined in the Section 2.1. Although all these models are based on the same architectural foundation, different models have different levels of abstraction that make them look differently. In general, they have the following similarities:

- ❑ Emphasize the importance of communication costs on this architecture.
- ❑ Assume fully connected network and the exact structure of the underlying communication network is ignored.
- ❑ Communication is based on point-to-point semantics, with the latency between any pair of processors roughly the same time for all cases.
- ❑ Performance characteristics of the communication systems are abstracted by a set of parameters.
- ❑ Most of the described communication models are message-oriented, such that the logical unit of transfer is message.

When comparing our model with these models, we notice that our performance model lies midway between the BSP and Phase Parallel models, and its functionality is closed to the LogGP model. However, our model has some remarkable differences as compared to these models. First, our model is a packet-oriented model. Therefore, it facilitates communication pipelining and overlapping of communications, and provides more freedom in designing efficient communication schedules. Second, our model supports simultaneously send and receive operations but does not assume that they are

Related Models	Programming Model	Abstract Model	Architectural Model	Characteristics
Bulk-Synchronous Parallel	PRAM or Message-Passing Model or BSP programming [65]	(p, g, L, *h-relations*)	generic architecture + synchronization primitive	Tightly-synchronized computation mode
Postal Model	Message-Passing Model	(n, λ)	generic architecture	Full connectivity; simultaneous I/O
LogP	Message-Passing Model	(L, o, g, P)	generic architecture	Implicit network constraint; overlapping of computation and communication
C^3 Model	Message-Passing Model	(p, s, h, l, $C_l()$, $C_p()$)	generic architecture + synchronization primitive	Introduces congestion overheads
LogGP	Message-Passing Model	(L, o, g, G, P)	generic architecture	Support long messages transfer
Phase Parallel Model	High-level Message-Passing Model	($\vec{t_0}$, $\vec{t_c}$, t_f, $\vec{t_p}$, w, σ, α)	generic architecture	Includes costs for collective operations and process management
LoGPC	Message-Passing Model	LogGP Model + $C_n()$	generic architecture	Contention factor on k-ary n-cubes network
Collective Computing Model	High-level Message-Passing Programming Model - MPI	(P, $\vec{F}$, $\vec{T_F}$, $\vec{P}$, $\vec{T_P}$)	generic architecture or BSP machine or LogP machine	Includes costs for collective operations and process management
Our Cluster Model	Message-Passing Model	(p, $O_s()$, $g_s()$, $O_r()$, $g_r()$, $U_r()$, $L()$, B_L, memcpy())	generic architecture + packet-switched network	Packet-oriented; explicitly exposes network constraint; facilitates communication overlapping

Table 2.3: Summary of related abstract models which built on the same architectural foundation. With simple message-passing model, the programming interface is based on the send and receive operations, or their variants. While for high-level message-passing model, the programming interface includes simple point-to-point communication as well as complex collective communications.

achieved in unit time step, e.g. Postal model. It depends on how well the full-duplex rule can be achieved, i.e. $(O_s + O_r + U_r) < g < L$. This assumption has significant impact on the performance of some communication operations, as in reality, interference exists between send and receive events. Third, our parameter set includes crucial communication issues such as message size, and network constraints - g_s, g_r & B_L, which capture the communication performance as well as congestion performance of the commodity interconnects.

2.4 Summary

In this chapter, we introduce a simple communication model for parallel computing on the cluster platform. The aim of our model is for performance analysis together with the ability to be an algorithm design tool, i.e. feasible for complexity analysis. In particular, the main objective of this model is for the development of efficient high-level communication primitives on top of those lightweight messaging systems. With this objective, we are able to develop portable parallel programs that run efficiently on a range of commodity clusters.

In the selection of the model parameters, we have two slightly conflicting considerations. First, the information reflected by our model should be easily assessable. This is because it is useless to include features that appear to be simple but are difficult to quantify in practice. For example, if one wants to reflect the cost spent on the DMA transfer, one needs to rely on the hardware supports since no simple software solution is available. Second, we must consider on the weighting factor of a target architectural feature on the performance issue. This is because, for the performance tuning aspect, we are tempted to provide more details; however, an overwhelming set will be too tedious for practical analytical use. We believe that the use of the model in algorithm analysis should be done straightforwardly and easily, whenever the users are provided with some systematic means of analysis. Therefore, emphasis has been made on the derivation of our model parameters by software approach, which is the key to the whole analytical process. Based on these measurable parameters, higher level primitives can be built or analyzed, and these primitives can also be used as some high-level performance parameters in analyzing complicated applications.

In our model, communication events are abstracted as some means of local and remote data movements, and each movement should have an associated cost and may be related to the length of the data items. To be realistic, we have included a rich parameter set to the model; however, the used of those parameters are subjected to the target level of abstraction that we are going to work on. Therefore, under some circumstances, a few performance parameters are proved to be adequate for modeling the parallel system. And on other occasions, there are other issues that need to be studied or included for making the correct judgment. For instance, using a simple latency parameter may be good enough to capture the cost of the point-to-point communication, but is too simple for explaining the many-to-one or many-to-many issues.

When compared to other models, we opt to expose the contention issue explicitly

and capture them in our parameters, e.g. the network latency and buffer capacity parameters, thus enhance the programmers' awareness on the contention issues. Besides, our model facilitates communication pipelining and overlapping of communications, which is useful for accurate performance analysis as well as for designing of efficient communication schedules.

Chapter 3

Performance Signatures

The design goal of our communication model is for performance understanding of the cluster communication system, which is driven by a lightweight messaging protocol on a commodity interconnect. As advances in networking technology demands for a low-overhead communication system, that delivers the best performance to high-level applications. When designing lightweight messaging systems, system designers are facing with design options that affect the overall performance, such as balance of workload between different hardware resources, or support multiprogramming verse dedicated use, etc. However, some design issues may turn out to have serious impedance on the performance than others when porting across different network technologies. In this chapter, we show how our communication model is used to delineate the performance characteristics of a lightweight messaging system, as well as to calibrate the performance results and assess various design tradeoffs.

We first give a general introduction on the sample lightweight communication package, the Directed Point. Next, we characterize the performance of Directed Point protocol with respect to our parameter set. During the analysis, we are comparing two different implementations of Directed Point that based on different network technologies, and we comment on their relative strength and weakness. We then make use of the available information to evaluate and analyze on the observed throughput of these communication systems. Finally, we summarize the results of this chapter.

3.1 Directed Point

Directed Point (DP) is a kernel-level lightweight communication system that aims at supporting high-performance communication in a multiprogramming environment over a broad spectrum of commodity interconnects. The adopted programming model of DP is the message passing model, in which data are exchanged between communicating pairs by explicit send and receive operations. Although the DP abstraction model supports group communications, they are being realized with matching send and receive calls, e.g. *dp_write()* and *dp_read()*. This send-receive paradigm supports unreliable, asynchronous communication between communicating processes. The users need to implement a reliable layer atop of DP if they want to work directly on the DP

messaging layer. Besides, it is the users responsibility to handle the fragmentation, re-assembly, multiplexing and demultiplexing of messages. DP provides a simple but flexible interfaces for system developers to build efficient high-level communication interfaces.

The system architecture of DP consists of three layers - the DP Application Programming Interface (DP API) Layer in the user space, the DP Services Layer and the DP Network Interface Layer in the kernel space. The DP API Layer consists of lightweight system calls and user-level function calls, which are operations provided to the users to program their communication codes. The DP Network Interface Layer consists of network driver modules. This layer is responsible for all hardware-specific messaging setup, and signaling the hardware to receive/inject messages from/to the network. Currently, supported network driver modules include Intel EEPro, Digital 21140A, 3Com 905C Fast Ethernet, Hamachi Gigabit Ethernet, and FORE PCA-200E ATM.

The DP Services Layer implements services for passing the packets from user space to the network hardware, as well as delivering the incoming packets to the user space buffers of the receiving processes. This layer realizes the DP messaging protocol and is hardware independent. To support asynchronous communication, a dedicated buffer is pre-allocated to each DP endpoint, which stores incoming messages that are directed to this endpoint. In DP abstraction model, an endpoint is the network abstraction for addressing a communication partner. Although DP supports dynamic binding of the same endpoint to different partners, at a particular instant, each endpoint corresponds to one communicating partner only.

The dedicated buffer is named as *Token Buffer Pool* (TBP) and is a fixed size memory area. Each TBP is shared by the kernel and the associated process through page remapping, which eliminates the delay caused by data copying from the kernel space to user space during the reception event. However, on the transmission event, DP writes the messages directly to the NIC address space without using the TBP. One of the design strategies of DP is the efficient utilization of memory resources. Incoming messages to the same endpoint are queued at the TBP in the form of FIFO linked list, with each segment corresponds to a variable-length message.

With the DP messaging protocol, to adhere to the message-passing semantic of sending messages from any arbitrary address and delivering them to any arbitrary address, we need to have one memory copy operation to move the data to a DMA-able memory region on the sender side, and two memory copies from the DMA-able memory region to the destination address on the receiver side. And the whole data transfer scenario of DP is summarized in Table 2.2 of Section 2.2.

3.2 The Performance of DP

We make use of two implementations of Directed Point to demonstrate how our communication model could be used for performance evaluation and analysis. As the model parameters represent some forms of software overheads and hardware latency,

changes in communication hardware and software are being revealed by the changes in these model parameters. This gives us better insights on the performance impacts of various design choices.

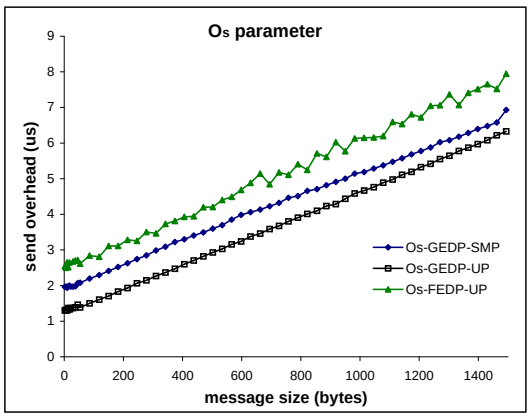
We have two clusters that are driven by two different Ethernet implementations of Directed Point, one is a Fast Ethernet cluster (FEDP) and the other is a Gigabit Ethernet cluster (GEDP). The FEDP cluster consists of 16 PCs running Linux 2.0.36. Each node is equipped with a 450MHz Pentium III processor with 512 KB L2 cache, a Intel 440BX PIIX4e chipset that supports a 66/100 MHz system bus, 128 MB of PC100 SDRAM, and uses a Digital 21140A Fast Ethernet adapter for high-speed communication. The whole cluster is connected to a 24-port IBM 8275-326 Fast Ethernet switch which has 5 Gbps backplane capacity. For the GEDP cluster, it consists of four Dell PowerEdge 6300 SMP servers with four Pentium III Xeon processors sharing 1 GB of EDO memory. The Xeon processor consists of 512KB L2 cache and operates at 500 MHz. This Dell system is using the Intel 450NX controller chipset with a 100 MHz front-side bus and has 64-bit 33 MHz PCI slots for the interconnects. All servers are running on Linux 2.2.12 kernel. In addition, each server is equipped with one Packet Engine G-NIC II Gigabit Ethernet adapter, and is connected to the Packet Engine PowerRail 2200 Gigabit Ethernet switch, which has a backplane capacity of 22 Gbps.

To review the performance issues related to high-speed communication, we have performed a series of microbenchmark tests on these clusters. To achieve beyond-microsecond precision, all timing measurements are calculated by using the hardware time-stamp counters in the Intel Pentium processors. If applicable, all data presented in this section are derived from a statistical calculation with multiple iterations of the same benchmark routine. Each test is conducted with at least 200 iterations with the first and last 10% of the measured timing excluded. Only the middle 80% of the timings is used to calculate the average.

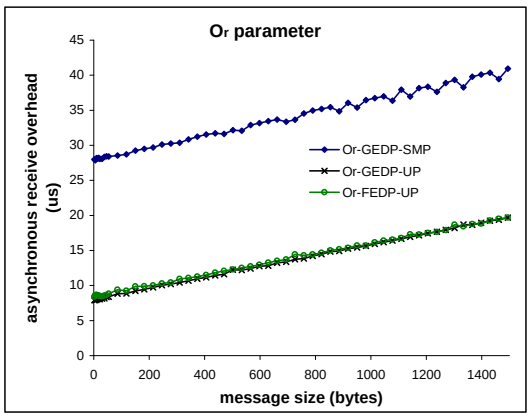
3.2.1 Latency with Performance Breakdowns

By executing the associated benchmark routines (Appendix A), we construct a set of model parameters for the two clusters, as shown in Figure 3.1. In the figure, there are two sets of parameters for the Gigabit Ethernet implementation (GEDP), one is obtained when using an SMP kernel, i.e. with the SMP support on the Linux 2.2.12 kernel (GEDP-SMP), and the other is without SMP support (GEDP-UP), i.e. uniprocessor mode on an SMP server. The purpose of this comparison is to reveal the differences in performance with respect to different OS modes and hardware platforms.

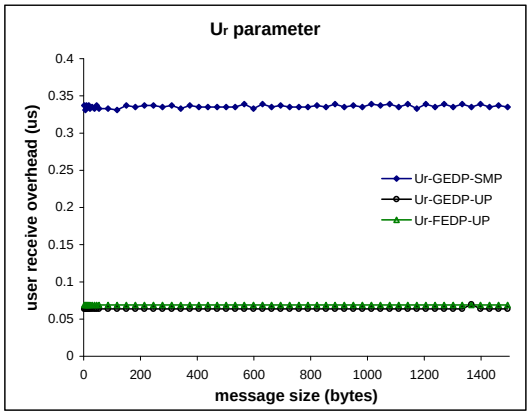
The O_s parameter With DP messaging library, the O_s parameter reflects the time used by the host CPU to initiate the transmission while performing the send (*dp_write()*) operation. Figure 3.1(a) shows the cost associated with the *dp_write()* operation. It involves a lightweight system call and a cross-domain data movement. We see that the processor speed does affect the software cost with 500MHz Xeon processor performs marginally better than the 450MHz Pentium III processor, as both



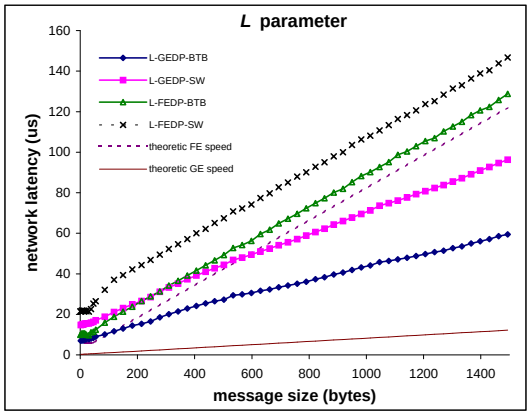
(a) Send overhead: O_s



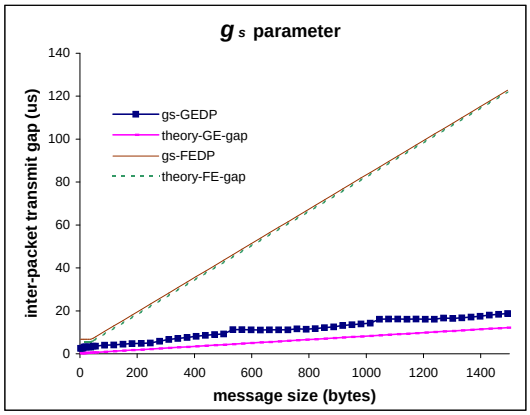
(b) Asynchronous receive overhead: O_r



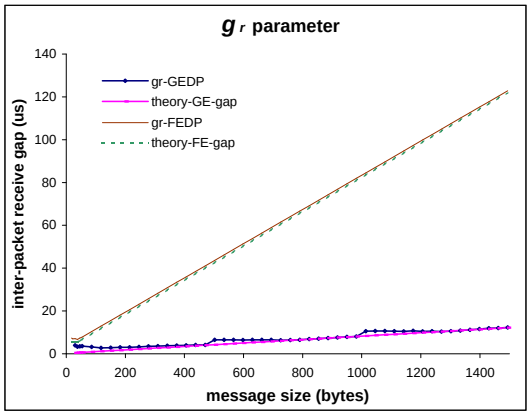
(c) User receive overhead: U_r



(d) Network latency: L



(e) Inter-packet transmit gap: g_s



(f) Inter-packet receive gap: g_r

Figure 3.1: Performance breakdown of two DP implementations - Fast Ethernet (FEDP) and Gigabit Ethernet (GEDP) expressed in the form of our model parameters.

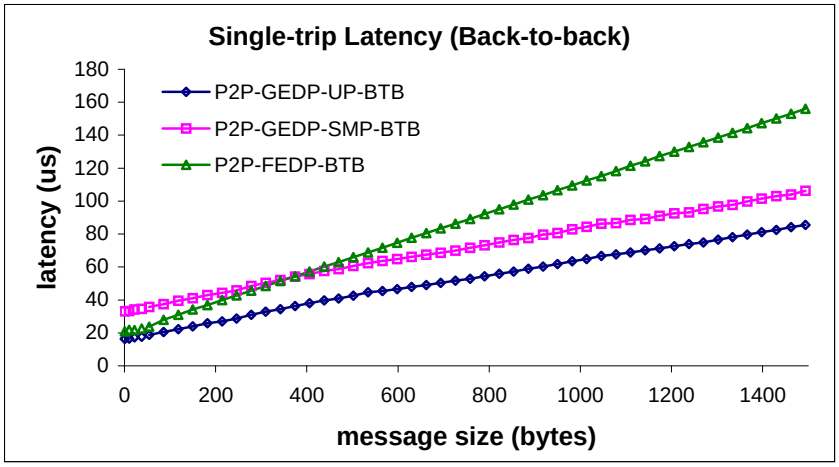


Figure 3.2: Single-trip latency performance with back-to-back (BTB) connection

hardware platforms are operated with a 100 MHz system bus. However, the prevailing software cost of the O_s parameter is coming from the data movement overhead in the send operation. We believe that with the DP protocol, any improvement in processor speed would be offset by the data movement cost; hence, using a system with a faster system bus and memory subsystem would benefit most.

Nevertheless, we see that DP still manages to minimize the send overhead and achieve good performance in driving both networks, especially it looks promising on the Gigabit communication. For example, the cost to send a full-size Ethernet packet is about 7 μs under the SMP OS, while the theoretical speed in transmitting such an Ethernet packet under Gigabit performance is around 12.3 μs. Therefore, an active sending process could saturate the network by continuous transmission. Lastly, observed from the GEDP measurements, with the SMP mode, there is an extra 0.5 μs overhead associated to it due to the locking mechanism for integrity control.

The O_r parameter When examining the O_r parameter in Figure 3.1(b), we find that the cost associated to this parameter is proportional to the message size, and the memory copy overhead is higher than that in the case of the O_s parameter. This reflects the different nature of the memory copy operation. For example, in the microbenchmark test of the O_s parameter, the involved memory copy operation is a M_{ctm} operation, while for the O_r parameter, it involves a M_{mtm} operation. Besides, we find that the SMP kernel has an extra 20 μs overhead added on to the GEDP-SMP, while both GEDP-UP and FEDP-UP have similar performance. This is also observed in the single-trip latency of the GEDP as shown in Figure 3.2, which is measured with the traditional *pingpong* test with back-to-back connection. There is a large performance gap appearing between the two OS modes. We conclude that this extra overhead is induced by the support of symmetric I/O and locking mechanism in the SMP kernel.

Besides the SMP overhead, we also observe that the current architecture of this Gigabit Ethernet adapter has a limitation on the achievable performance. Due to the

lack of intelligence network processor, incoming messages are not delivered to the user process directly. Instead, they are moved by the DMA engine to a pre-allocated network buffers area. This requires an extra memory copy done in the interrupt handler to deliver the messages to the destined user process. The one-copy cost together with the interrupt overhead ($\sim$ 8 μs) would become a threat to the overall performance, e.g. the total interrupt cost for a full size packet is 19.7 μs under GEDP-UP. This O_r overhead is larger than the theoretical transmission delay of Gigabit network. Thus, would hinder on the achievable performance.

There are several methods to work out this problem. First, introduces a network processor to the network adapter such that it can be programmed to move the incoming messages directly to their destined buffers. This approach is taken by other lightweight messaging systems that built on top of Myrinet or Giganet [39], e.g. BIP and FM 2.x. Thus, the interrupt and data movement overheads can be eliminated completely. However, almost all commodity Gigabit/Fast Ethernet cards do not provide the luxury to solve this problem. This is because having a network processor together with the associated SRAM is so expensive that the cost of the memory is roughly half of the production cost. From the commercial point of view, this is not justifiable for improvement of just a few microseconds.

Another method is by the mitigation of interrupt overhead through multiple packet receptions - interrupt coalescing. Most Gigabit Ethernet adapters provide a mechanism to perform tuning on the inter-interrupt gap. For examples, wait until there are *x* incoming packets before raise the interrupt signal, or hold off any pending interrupts until *y* μs has elapsed since handling last packet. The GAMMA messaging system takes a slightly different approach. Whenever the network adapter raises the interrupt signal, the GAMMA protocol blocks off further interrupts by clear the processor's interrupt flag. By this way, the interrupt handler manually checks on further arrival of incoming messages and handles them in one shot. But this method only works under UP kernel. Nevertheless, interrupt coalescing is useful in cases where packets are arriving in back-to-back, but comes at the expense of increased per-packet latency.

The U_r parameter Since the token buffer pool is accessible by both kernel and user processes, the receiving process can simply check on the TBP for picking up and consuming the messages. As these are done in the user space, no kernel events such as block and wake-up signals are needed. Figure 3.1(c) shows the U_r cost of picking up a DP message directly from the TBP without any data movement or system call overheads. Constant overheads, 0.34 μs, 0.06 μs and 0.07 μs were measured for GEDP-SMP, GEDP-UP, and FEDP, respectively.

Moreover, U_r is not necessarily a constant value. With real communication events, we need to employ another memory copy operation to move the data from the TBP to the destination buffers. This is because the TBP is a pre-allocated memory region dedicated for incoming messages, hence, it does not directly conform to the desire message-passing semantic. Besides, consecutive messages are stored in TBP, which are not aligned in contiguous memory region. To re-assemble long message, one needs to re-construct the message segments back to one large trunk. Therefore, an add-on

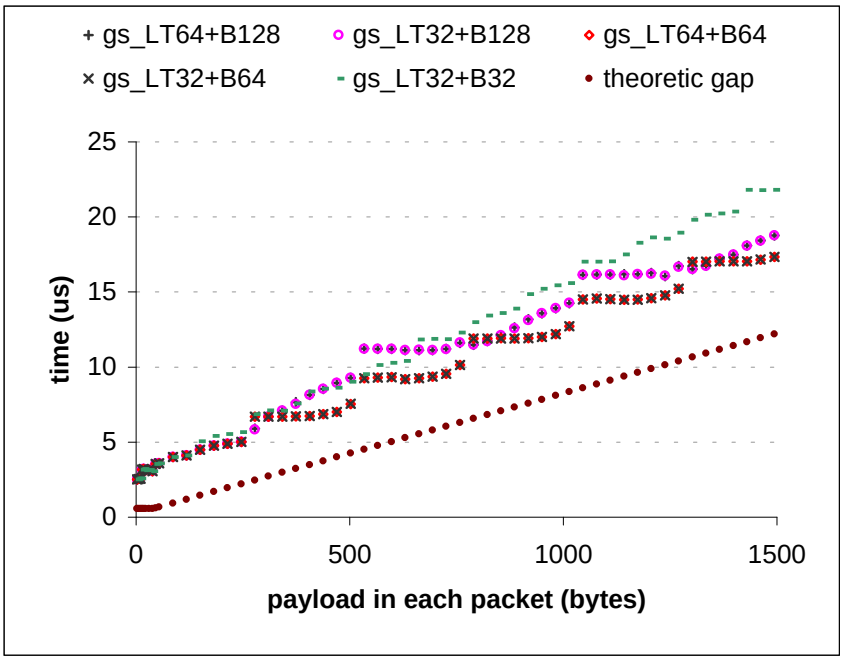
M_{ctm} software cost is expected for each arrived segment. On the GEDP platform, this costs an extra overhead of $\sim 10\ \mu s$ for a 1500-byte packet.

The g_s and g_r parameters Figure 3.1 (e) and (f) show two other network-dependent parameters, they are the inter-packet transmit gap g_s and inter-packet receive gap g_r. To justify their relative performance, all parameters are compared with their theoretical limits. Looking at the FEDP data, we find that with modern PC or server hardware and lightweight communication system, we are able to drive the Fast Ethernet network to its full capacity. For example, the measured g_s and g_r for *m* = 1500 bytes is 122.75 μs and 122.84 μs, while the theoretical transmission speed is 123.04 μs. This means that the critical path of the communication system falls on the Fast Ethernet network.

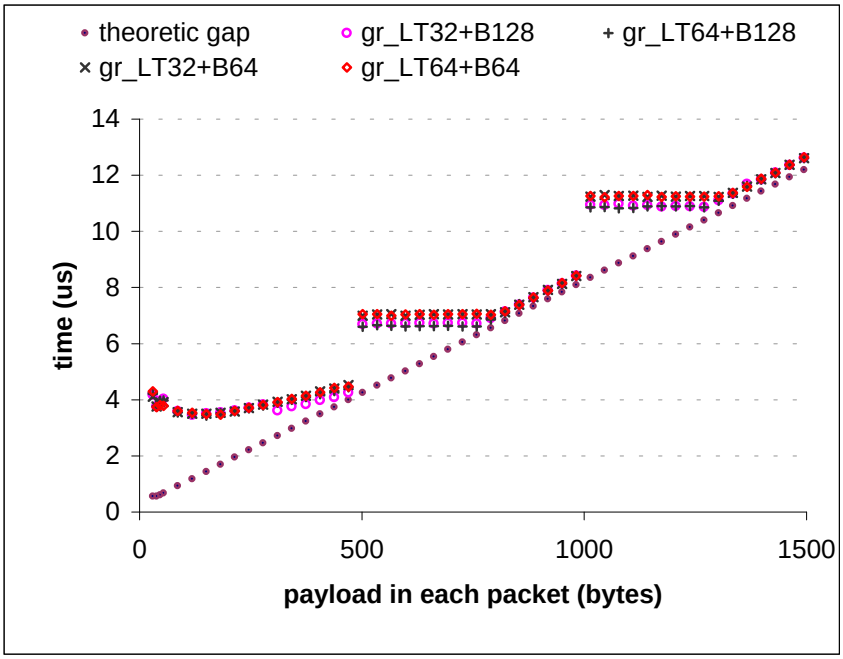
For the Gigabit Ethernet, due to the 10-fold increase in network speed, limitations within the host machine start to pop up. The graph with g_s-GEDP data (Figure 3.1e) shows that the network adapter cannot transmit data in full gigabit performance. The measured g_s value for *m* = 1500 bytes is 18.76 μs but the theoretical speed is 12.3 μs. Since the value of g_s reflects how fast can the network adapter inject a packet into the network, we clearly see that there exists some bottleneck problem. To explore the problem further, we have performed some investigations on this area, and find that the problem seems related to the PCI performance, even though our Dell server is equipped with a 64bit 33MHz PCI bus. It is known that inefficient use of the PCI bus would result in poor system/network performance [48]. Factors such as the PCI burst size and the PCI latency time are of the most importance, since they can be directly manipulated by the system programmer. In our experiments, we varied the burst size and latency time, and conducted our standard microbenchmarks to measure the resulting g_s and g_r values. Figure 3.3(a) shows the differences in g_s values under different PCI settings. It is clear that the PCI setting has significant influence on the resulting g_s values. From the experimental results, we find that the setting of PCI burst size has a paramount effect on the g_s value, while the latency time looks quite inert on our tests. For example, the best network performance is observed with PCI burst size using 64 d-words (256 bytes) with different latency settings.

A similar pattern also appears in the g_r-GEDP data, but the problem is not as clear as that of the g_s parameter. We find that the measured g_r value for m = 1024 bytes is 10.6 μs but the theoretical gap is 8.5 μs; on the other hand, the measured g_r value for m = 1500 is 12.6 μs while the theoretical gap is 12.3 μs. Although we still observe the variation of g_r values under different PCI settings (in Figure 3.3(b)), it appears to be less drastic than its g_s counterpart, and the measured results look quite independent of the burst size and latency time. Part of the reason may be due to the difference in read and write performance of the PCI bus, in particular, under the Intel 82450NX chipset [49]. For example, the observed throughput for PCI read (from memory) is approximately 47% less than the PCI write (to memory) throughput on a 64-bit PCI bus.

The L parameter When look at the *L* parameter (Figure 3.1(d)), the derived network latency of the GEDP with back-to-back connection is 6.9 μs for a 1-byte message,



(a) Inter-packet transmit gap



(b) Inter-packet receive gap

Figure 3.3: The measured g_s and g_r values on the GEDP platform under various PCI settings. With LTXX stands for setting the PCI latency to XX bus cycles; and BYY stands for the PCI burst size (YY d-words).

while the network latency of the FEDP with back-to-back connection is 9.9 μs for the same size message. We observe that the add-on latency by the GE hardware is much higher than that of the FE, when compare to the theoretical wire delay for the smallest packet size of the GE and FE, which are 0.67 μs and 6.7 μs respectively. In addition, the time gaps between the network latency measurements with FEDP back-to-back and FEDP through switch, and between FEDP back-to-back and theoretical FE speed are almost constant, while the corresponding gaps on the GE platform seem to be increasing with the message size. This indicates that there exists some store-and-forward stage(s) along the GE network path.

Lastly, Figure 3.2 compares the single-trip latency of the two DP implementations. To avoid add-on latencies from the switches, we connect two nodes back-to-back and measure their single-trip latencies. The GEDP-UP achieves single-trip latency of 16.3 μs for sending 1-byte message, while GEDP-SMP achieves 33.4 μs and FEDP achieves 20.8 μs respectively.

From the above analysis, we obtain two sets of performance metrics, which clearly delineate the performance characteristics of the two DP implementations. In summary, the host/network combination of the FEDP implementation has the performance limitation on its network component. This is being observed by comparing the O_s, O_r, and U_r parameters with the g_s, g_r and L parameters. And since their performance characteristics satisfy the full-duplex condition, i.e. $(O_s + O_r + U_r) < g < L$, we can directly adopt the previous defined point-to-point communication costs (Eq. 2.6 & 2.7) whenever we want to evaluate on its long message performance. Moreover, the host/network combination of the GEDP implementation has the performance limitation not falling on the network component. For instance, the O_r parameter is higher than the g_s and g_r parameters for both GEDP-SMP and GEDP-UP, which means the performance bottleneck may fall on this region. Therefore, when predicting their long message performance, alternate point-to-point communication cost formulae are required. For example, since the bottleneck stage falls on the receive phase, the new cost formula for predicting the one-way point-to-point communication cost of the GEDP-UP implementation becomes:

$$T_{p2p-GEDP-UP}(M) \approx O_s + L + k(O_r + U_r) \tag{3.1}$$

3.2.2 Uni-directional Bandwidth

In this section, we are going to explore the one-way bandwidth performance of the two DP implementations with respect to different hardware and OS mode. In the analysis, we try to apply the acquired knowledge from the above section to explain and evaluate the measured performance.

Two sets of uni-directional bandwidth measurements for each DP platform - FEDP, GEDP-UP and GEDP-SMP, are presented in Figure 3.4. To calculate the *raw* DP bandwidth, we measure the time spent in transmitting 10 MB data from one process to another remote process, plus the time for the receive process to send back a 4-byte

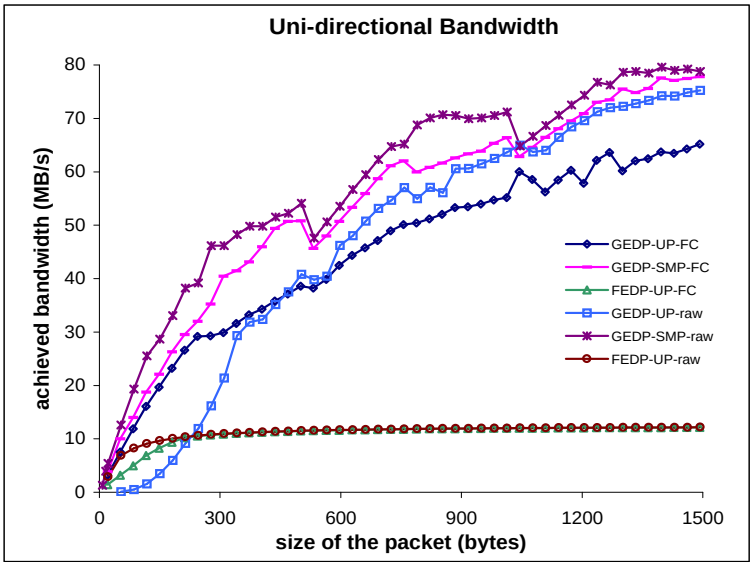


Figure 3.4: Uni-directional bandwidth performance of DP
(FC - with flow control on; raw - unreliable mode)

acknowledgment. By subtracting the measured time with the single-trip latency of a 4-byte message, we calculate the achieved bandwidth as the number of bytes transferred in the test divided by the result timing. As DP supports unreliable communication only, we have implemented a simple Go-Back-N protocol on top of DP to provide flow control and support limited reliable communication. Since all the protocol works are done in the user space, it has add-on overheads to the O_s and U_r parameters. For example, the O_s-GEDP-SMP value for sending a full load packet is increased from 7 μs to 12 μs. To calculate the flow-controlled bandwidth of DP, we performed a set of tests similar to what we have done to obtain the raw DP bandwidth.

From the figure, we see that the maximum achieved bandwidth for GEDP is 79.5 MB/s, which is the raw DP performance measured under the SMP kernel. Under the UP kernel, the raw GEDP achieves at most 75.2 MB/s. Despite the fact that the SMP kernel has a higher O_r overhead, it has a better throughput than the UP kernel. This shows the advantage of sharing the token buffers between the kernel process and user process. Under the UP mode, the user process can only pick up its arrived messages after the interrupt thread returns, so the whole interrupt overhead is included in the delay calculation. However, with the SMP mode, we have more CPU resources and the user process can check out its messages even before the interrupt thread returns. This is because when the receive process gets the CPU cycles and detects that there are arrived messages, it can immediately consume the shared data. Besides, due to the large interrupt overhead on SMP kernel, it is likely that an interrupt thread would pick up more than one arrived packet. Hence, in long run where packets are arriving in back-to-back, this effectively amortizes the interrupt overhead across multiple arrivals.

For the FEDP, the achieved maximum raw DP bandwidth is 12.2 MB/s, which is 97% of the Fast Ethernet theoretical performance, while the raw GEDP-SMP per-

formance only achieves 63.6% of the theoretical gigabit throughput. This shows that there are limiting factors in the host machines which hinder the GE performance. We have seen in Figure 3.1(e) that the network adapter cannot transmit data in full gigabit performance, by dividing the payload size with the corresponding g_s value, we have a useful meter to estimate maximum performance we can get. Take the value of g_s = 18.7 μs at *m* = 1500 bytes as an example, we find that the maximum transmission throughput is around 80MB/s, which is closely matched with the GEDP-SMP measurement. Similarly, if we assume that the bottleneck is on the O_r part, let's take the value of O_r = 19.7 μs for *m* = 1500 bytes, we should have the transmission throughput bounded by 76 MB/s. Again, this is closed to the measured performance on GEDP-UP. From these analysis, we can conclude that the performance of the GEDP is limited by the g_s parameter when operates under the SMP kernel, but the bottleneck is shifted to the O_r parameter when operates under the UP kernel.

To reveal how much improvement we could achieve if we adopt a zero-copy semantic in the send path, we have done some tests that simulated a zero-copy send operation, (simply by removing the *memcpy()* operation and sending out garbage content). The resulting send gap (g_s) is approximately 16.4 μs for *m* = 1500 bytes, which would correspond to a bandwidth of 91.5 MB/s. By eliminating the memory copy operation, it should only affect the O_s parameter. However, we find that the g_s parameter has changed too. This simple experiment suggested that the g_s parameter is sensitive to other bus activities, since memory copy operation involves bus transaction on the system bus, which in theory, interferes with the other data movements on the bus network.

With the add-on reliable layer, the FEDP performs almost as good as the raw performance for medium to large-sized messages, which achieves a throughput of 12.1 MB/s. But for the GEDP, the higher protocol overhead does affect the overall performance, especially under the UP kernel mode. Our result shows that under the SMP mode, the maximum achieved GEDP bandwidth with flow control is 77.8 MB/s, with an average drop of 3.4% performance for the packet size ranged between 1K and 1.5K when compared with the raw speed. While for the performance under UP mode, the maximum achieved bandwidth with flow control is 65.2 MB/s and the average performance drop is 13% of the raw speed for the same data range. This further supports our argument that the performance of GEDP-UP is more susceptible to software overheads.

3.2.3 Bi-directional Bandwidth

Most networks support bi-directional communication and lots of communication patterns require concurrent send and receive operations to achieve optimal results, e.g. complete exchange operation, shift operation, tree-based broadcast, etc. We extend the tests used for uni-directional bandwidth to evaluate the communication performance of the bi-directional communication. During the experiment, two nodes are involved in each test, but they are both sender and receiver. To measure the raw bi-directional bandwidth of DP, both processes are synchronized by a barrier operation before starting the exchange. We measure the time spent by each process in exchanging 10 MB

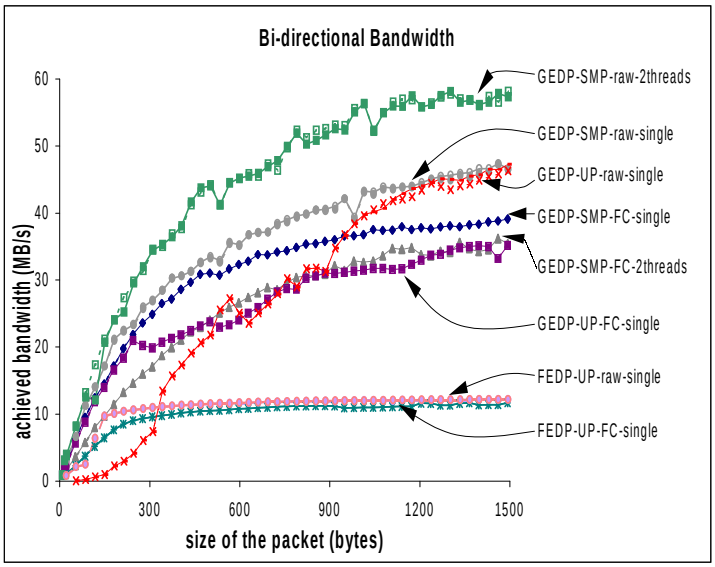


Figure 3.5: Bi-directional bandwidth performance
(FC - flow control; 2threads - multi-thread mode; single - single thread mode)

of data, and calculate the bandwidth by dividing the exchange message size with the measured time. Similarly, we perform the same set of tests with the add-on reliable layer. When testing the bi-directional bandwidth on the SMP kernel, we also try to explore the effect of using multiple CPUs in driving the communication. We have performed a set of tests with two threads per process, which share the same DP endpoint, one thread takes up the job as the sender while the other acts as the receiver. All the experimental results are summarized in Figure 3.5.

For the GEDP, the best bi-directional performance is observed to be about 58 MB/s per process, which is measured with raw DP using multi-thread mode on SMP kernel. Comparing with the uni-directional bandwidth, we have a performance loss of 22 MB/s, which is a 27.5% drop of the peak point-to-point performance. We attribute this performance loss to the contention on the PCI and system buses as there are concurrent DMA transfers to and from the host memory as well as memory copy operations on both send and receive phases.

When compared with the single thread mode on GEDP-SMP and GEDP-UP, which only achieve 47 MB/s per process, we believe that the software overhead induced in the concurrent send and receive operations is the main cause of this performance loss. Therefore, with the add-on reliable layer that adds more software overhead, it is sensible to see that all GEDP-FC performance suffers more. However, it is surprising to find that the bi-directional performance of GEDP-SMP-FC with multiple thread support is worse than the single thread mode. This performance difference is coming from the extra memory contention and synchronization needed in accessing shared data structures on the reliable layer as both threads are concurrently updating these shared information. Finally, similar to the conclusion as appeared in the uni-directional benchmark, the performance of the FEDP on bi-directional communication has achieved a near-

optimal result, which attains 12.1 MB/s per process on the raw bandwidth, and 11.7 MB/s per process with the add-on flow control support.

3.3 Summary

Focusing on issues related to performance understanding, we show that by benchmarking the communication system in a structural way, we can smoothly construct a logical breakdown of the communication system. As individual components correspond to certain architectural features, it is easier to explore their strength and weakness, assess design tradeoffs, and suggest remedy actions if appropriate.

The FEDP implementation shows that additional memory copy performed in a relatively high-speed host causes no performance degradation while driving a slow network. On the contrary, even with a high-end server, when applying the same mechanism on a high-speed network, the DP protocol fails to drive the Gigabit Ethernet in full speed. Therefore, we believe that when designing lightweight messaging systems, one should consider the performance gaps between processor, memory, and the network, especially should have some visions on the future development. Although the above analysis shows that part of the GEDP deficiency is coming from the PCI performance, the evaluation result also demonstrates that there still have rooms for us to further improve its performance, especially on the bi-directional bandwidth which is involved in many collective operations.

Based on our model parameter set, it shows that the data movements on the O_s and O_r parameters affect the overall performance in the Gigabit communication. Thus, further reduction of the protocol handling overheads (O_s, O_r & U_r) are needed. In particular, as data movements are inevitable, one should focus on coordinating these data movement to minimize the memory copy costs. Besides of the data movement overhead, the interrupt overheads are shown as another weak point in current situation. Although interrupt coalescing may improve the overheads for long messages, it also increases the per-packet latency; thus, is not good for small size messages as well as infrequent communications. One should adopt some heuristic method to dynamically handle the interrupt issue in a more efficient way.

The above analysis has focused on the point-to-point issue over a lightly loaded network, therefore, the network congestion issue is not addressed yet. Networks react to congestion in different ways, depending on the switch hardware as well as the adopted communication protocol. In next chapter, we are going to extend our performance studies from the point-to-point analysis to a highly congested communication pattern, the many-to-one collective operation.

Chapter 4

Congestive Loss on High-Speed Communication

Understanding the contention phenomenon is crucial to high performance computing, as contention can happen in the hosts, network links and within the routers. Furthermore, the degree of contention has a direct implication on the sustainable performance for a particular architecture-application pair. Different combination of hardware and software, together with different communication patterns and schedules may stimulate different congestion behavior. These make modeling of congestion behavior on a global communication event a challenging task. Although direct quantifying the target architecture-application pair could tell us the degree of performance loss induced by the contention, it does not provide information on the actual phenomenon that induces the loss.

In this chapter, we are focusing on the congestion behavior of those Ethernet-based lightweight communication systems under heavy congestive loss problem. In particular, we try to model the error path of a user-space Go-Back-N reliable transmission protocol, which is built on top of the Directed Point (DP) low-latency communication system. During the modeling exercises, we identify salient features that enhance our understanding on the packet loss problem, and therefore, are relevant for the design and analysis of "contention-friendly" communication schemes.

We start this chapter by discussing the importance of the reliability issue on lightweight communication systems. A brief survey is given in Section 4.2 on how other Ethernet-based low-latency communication schemes support the reliability issue, as well as the description of our Go-Back-N protocol. Then, in Section 4.3, we examine and model the congestion dynamic of different buffering architectures under the many-to-one congestion loss problem. These analyses are augmented by experimental evaluations on real platforms. In Section 4.3.4, we further corroborate our analysis by extending the model to cover different network configuration and communication patterns. Finally, we conclude this chapter with a discussion of related studies in Section 4.4, and provide a summary on our contributions in Section 4.5.

4.1 Reliability

The development of lightweight messaging systems is to support low-latency communication on high-performance computer. These systems aim at delivering the best communication performance to the application layers. However, subtle issues in design and implementation of these communication libraries could impact on their usability and the ultimate performance available to the application level. One important aspect that most of these lightweight messaging systems have trade-off for performance is the reliability issue, which is a critical issue that affects the final performance delivered to the real applications.

Most clusters use standard system software to manage message passing. Particularly, TCP/IP protocol suite is the most commonly used communication protocol found in system software that delivers reliability to the application layer. However, TCP was initially designed to run on unreliable, wide-area network, and therefore, it is not optimized for the high-performance domain. Many performance studies on TCP/IP implementations [24, 54, 55] have reported on the high software overheads associated to those supporting functions of this protocol stack, e.g. data movement and buffer management overheads. They showed that these software overheads are the major hindrance to the achievable performance, especially when most of the data traffics are small size messages. As a result, all low-latency communication libraries have opted to bypass this layer and provide their own lightweight communication schemes. However, this brings up another question on how these new communication systems are going to handle the reliability issue.

When talking about high-performance cluster computing, people commonly have the following assumptions on the cluster interconnect. First, the interconnection networks are composed of commodity local area network (LAN) or system area network (SAN), which are characterized as low propagation delay, high bandwidth networks. Second, they assume that the underlying network is almost reliable, such that the underlying hardware has extremely low transmission error rate. Based on these assumptions, different communication packages provide difficult levels of reliability. For some low-latency communication packages, they just offer an unreliable programming interface, and require higher level software to think over on the reliability issue. On the other hand, some packages assume that the hardware is reliable, all data loss issues are the result of receiver buffer overruns. Therefore, an effective measure to prevent buffer overrun is by using of flow control mechanisms, e.g. sliding window and/or credit-based flow control.

While choosing to provide an unreliable or a lightweight reliable communication interface to the higher level applications is a decision issue that strives for the best practical compromise; however, one of the major achievements of the TCP protocol is its congestion control mechanism. Previous studies on high-speed LANs [55, 79] have shown that the high overheads in traditional communication software limit the ability to generate load; therefore, the observed contention problem was not critical. With the availability of low-latency communication mechanisms, applications can now generate higher load to the network, that could result in congestion build up in some part of the

network, and of the worst scenario, this would induce congestion loss problem. As the TCP/IP protocol suit is not included in these lightweight messaging systems, the system designers have to consider about supporting the congestion control mechanism on these communication systems.

Some authors of these low-latency communication packages argued that their underlying networks support link-level flow control, transient buildup of congestion can be spotted and back-pressure could be applied link-by-link all the way back to the sender; therefore, avoid data loss problem in the network. This may be true on some type of interconnects, e.g. Myrinet, but is a dangerous assumption for Ethernet type network. Since there could have data loss problem in the network, having end-to-end flow control alone could not resolve the problem, an error recovery mechanism should be incorporated in these low-latency communication schemes so as to provide a reliable communication interface to higher-level applications.

4.2 Reliable Transmission Protocol for Low-Latency Communication System

Before discussing on our reliable transmission protocol, we have a brief survey on several low-latency communication packages with regard to how they handle the reliability issue. Existing low-latency communication systems for clusters fall into two main families. One is based on programmable network devices, such as Myrinet and GigaNet [39], which make use of the embedded co-processors to offload the host processors in running their customized messaging protocols. Another type is based on non-programmable network devices, such as Fast Ethernet and ATM, which rely on the host processors to carry out their customized messaging protocols. In the following discussion, we use the Myrinet-based systems as the representatives of the first type, and the Ethernet-based systems as the representatives of the second type of messaging systems.

Myrinet-based systems

- AM-II [23] - it implements the reliable protocol in the NIC firmware, which supports NIC to NIC flow control, retransmission, and detecting and recovering from errors. The AM-II system addresses flow control at three levels:

  1. uses of credits at the user-level for each endpoint; this allows multiple outstanding requests to fill the communication pipeline.
  2. between NIC to NIC, it uses stop-and-wait flow control for each logical channel; but there have multiple independent logical channels between each NIC pairs.
  3. relies on the Myrinet link-level back-pressure to ensure the network does not drop packets.

A timer event is associated with each packet transmission, which is used to ensure reliable delivery by a simple timeout and retransmission mechanism. Upon successfully delivery of an in-order packet, the receiver positive acknowledges the sender; hence, the sender clears the timer event and frees the corresponding logical channel. When error conditions occur, e.g. buffer overflow or nonresident endpoint, the protocol drops packets and sends back a negative acknowledgement to the sender. However, the sender does not have immediate reaction to the negative acknowledge, the protocol relies on the timeout and retransmission mechanism to resend the packet.

- FM [75] - Myrinet FM's design supports reliable, in-order delivery with end-to-end credit-based flow control and FIFO queuing at all (host and NIC) levels. By assuming the network is reliable, no timeout and retransmission mechanism is implemented. Each sender is allocated portion of a receiving node's host memory queue size (credits), and this guarantees that the pre-pinned message queue will not be overrun, and no message data will be lost.

- BIP [82] - to deliver more than 96% of the raw Myrinet bandwidth with a very low one-way latency timing, BIP does not support any forms of error recovery and flow control mechanisms. It relies completely on the Myrinet hardware to ensure reliable, in-order delivery.

- LFC [13] - it assumes that the network does not drop or corrupt packets, the only cause of packet loss is the lack of buffer space on receiving nodes' NICs. It adopts a credit-based flow control between each pair of NICs, and send credits are returned by means of explicit acknowledgements or by piggybacking on application-level returned traffic. However, credits (NIC buffer space) are statically partitioned among all nodes, so it needs more NIC memory as the cluster size grows.

Ethernet-based systems

- U-Net/FE [114] - it provides an unreliability interface and leaves behind the reliability issue to the higher level applications. To prove that U-Net abstraction is able to support legacy systems, a proof-of-concept implementation of TCP over U-Net/ATM [108] had been implemented. It was shown to achieve better performance on both bandwidth and round-trip latency as compared to the traditional in-kernel protocol under lightly loaded condition. However, nothing had been done on investigation whether the TCP congestion control mechanism is still suitable for supporting high-performance low-latency communication.

- GAMMA [21] - the initial version that based on shared Fast Ethernet network relied on the CSMA/CD property of the Ethernet network to support limited reliable service with error detection. The redesign GAMMA prototype [22] has included a credit-based flow-control protocol to prevent receiver overrun problem after migration from the shared-based to the switched-based network. However,

no error recovery mechanism is provided as the designers assumed that transmission error, the source of data loss other than receiver overrun, is extremely rare with current Ethernet technology.

- GigaE-PM [96] - as it assumes that the Ethernet network does not guarantee message arrival, it adopts a Go-Back-N protocol with STOP and GO flow control to guarantee in-order message delivery on high-speed network without using the TCP/IP protocol. Besides using timeout retransmission mechanism to recover from data loss, upon detection of out-of-order messages, the receiver immediately sends back a LOSE message to the sender for quick recovery from the loss.

- Pupa [106] - it provides reliability guarantee by using optimistic credit-based flow control, positive and negative acknowledgements, and timeout retransmission. Initially the flow control restriction is not enforced by giving unlimited credit to each sender. Upon detection of buffer overflows on the receiver, a credit update with zero credit is sent to the sender, which causes the sender to stop sending until it receives a subsequent positive credit update. Besides, Pupa adopts a piggybacked acknowledgement scheme with sender-controlled acknowledgement, which is a mechanism to reduce the amount of acknowledgement traffics but still asserts effective response.

- PARMA^2 [62] - is designed as a socket compatibility layer with no mechanism associated to flow control and data recovery. This is because the authors believed that the percentage of packets incorrectly delivered by a homogeneous local area network is negligible. Therefore, typical flow control and acknowledge-retransmission schemes become obsolete, and they believed that lost packet detection is sufficient for all common user-level applications, e.g. MPI-based applications.

In summary, most of these low-latency communication libraries have adopted flow control strategy to avoid receiver buffer overflow problem, which is commonly happened in an asynchronous, distributed network environment. However, there is no prevalent approach to handle the error recovery as most of these systems only focused on the fast/normal path of the communication and neglect the error path, since those authors believed that the underlying networks are reasonably reliable. In the next subsection, we will have detail discussion on our reliable transmission protocol, which is a variant of the Go-Back-N Automatic Repeat Request (GBN ARQ) [11, 91]. We are adopting approaches similar to other packages, as well as we lay more emphasis on the error recovery aspect.

4.2.1 Our Go-Back-N ARQ Protocol Definitions

To support reliable communication, we have implemented a user-level reliable transmission layer atop of DP. From our experiences, we observed that modern networking

technologies are quite reliable, as almost all data loss problems happened in the network are due to the congestion problem. As we believe that by using our communication model, we can devise efficient schedules to avoid congestion problem for most of the collective communications. To keep the reliable layer as lean as possible, we try to avoid unnecessary memory copy. Thus, we adopt the *in-order accept policy* [92] on the receiver side so as to reduce the U_r overhead, and a variant of Go-Back-N ARQ as the error recovery scheme on the sender side.

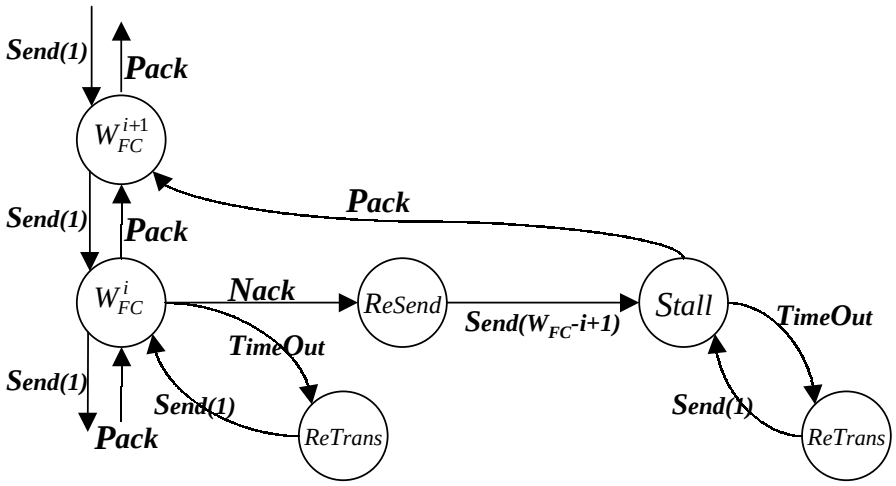
Another commonly used ARQ is the Selective-Repeat ARQ [92]. With this scheme, the only frames retransmitted are those that receive a negative acknowledgement. Although Selective Repeat ARQ provides better retransmission strategy, it requires complex buffer management and/or an extra memory copy for handling out-of-order data arrival. Therefore, this becomes costly if the underlying architecture does not support it. For example, to implement selective repeat ARQ on top of DP, we need to have an extra memory copy (M_{mtm}) and an extra buffer pool which is large enough to temporarily buffer all out-of-order arrivals. Since we believe that with well-coordinated and well-scheduled communication schemes, congestion loss would be rare. This could not justify for consuming extra resources and adding extra overheads to the lightweight messaging system.

Go-Back-N ARQ is a fairly straight-forward protocol and has been adopted in other lightweight messaging systems, e.g. [96, 113]. However, simple variations of the protocol could have significant impact on the final performance under congestive loss situation. For example, TCP is basically one type of ARQ protocol[1], however, its complexity and importance on the Internet has attracted lots of researches and investigations within the past decades [63, 74, 84]. Before we layout our analyses on how different buffering architectures affect on the congestion behavior, we need to provide a clear picture on the GBN protocol that we are using.

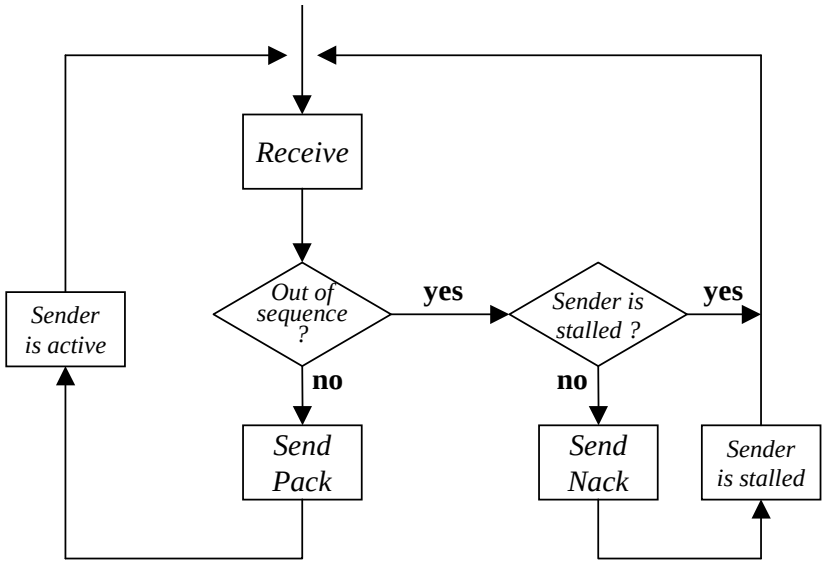
We are using a fixed size credit-based sliding window flow control and let W_{FC} be the size of the window, this bounds on the maximum number of (outstanding) packets that a source/sink process pair is allowed to transmit without waiting for an acknowledgment. To detect for loss packets or loss acknowledgements, our reliable transmission protocol set up a timeout value for *each* outstanding packet. Since we assume that the network is relatively error free, a static timeout value is used.

Figure 4.1 shows the state transition diagram of the sender and the logic flow diagram of the receiver according to this GBN protocol. When the sink process receives a packet, an acknowledgment may or may not be sent out that depends on (1) the current state of the source with respect to this sink, (2) the validity of this packet, and (3) instruction from upper level application. One of the design goals of this reliable layer is for efficient implementation of structural communication schemes, e.g. collective communication. Therefore, we provide a mechanism for higher level application to decide on the most effective acknowledgment methods, such as having immediate

[1]The TCP protocol (specified in RFC 793 [85]) does not define the accept policy on the out-of-order data arrival and the retransmission policy on the error recovery, therefore, we could have a Go-Back-N like TCP if a particular implementation uses *In-order* accept policy and *Batch* retransmit policy (page 663 in [92]). However, most of the implementations adopt the Selective Repeat ARQ approach.



(a) Sender



(b) Receiver

Figure 4.1: Go-Back-N protocol with window flow control - (a) state transition diagram of sender, (b) logic flow of receiver

acknowledgment, piggyback acknowledgment, or delay acknowledgment whenever possible.

There are two types of acknowledgments. An in-order packet will result in a positive acknowledgment (***Pack***), otherwise, a negative acknowledgment (***Nack***) is generated. This Nack packet becomes a loss signal to the source when an out-of-sequence packet is detected, then this packet is simply dropped. Now the receiver labels this sender as has transited to the ***Stall*** state, and all subsequent out-of-sequence packets from this sender will be discarded unconditionally without generating any more negative acknowledgements. This receiver only resumes the acknowledging process until it receives the first in-order packet from the corresponding sender. The rationale behind this approach is to minimize the number of control packets. Even though control packets are small-size packets, they still consume network resources, e.g. network buffer and host receive buffer, which are critical resources under the congestion situation.

On the other side, whenever the source process receives a Pack packet, it advances its sliding window, and thus, allows injection of one more packet to the network. Error situations are detected at the sender side by either receiving a Nack packet or a timeout situation is raised. On receiving a timeout incident, the sender retransmits the corresponding packet and reset its timer. On receiving a negative acknowledgement, the sender backs up to the first error packet (indicated on the Nack reply) and resends all outstanding packets. Then it transits into the Stall state. Under the Stall state, the sender stops sending out new packets until the first Pack reply is returned from the receiver, then it transits back to the normal transmission state. Otherwise, it waits for the timeout situation and retransmits those timeout packet(s).

The unique features of this reliable transmission protocol are the use of fast retransmission mechanism and the present of the Stall state. Although standard GBN protocol is known to be inefficient if error rate is high, we believe that under well-organized communication schedules, the probability of having error situations is extremely low. Moreover, we still attempt to improve the error recovery path in a way that the extra work-done would not hurt the performance of the common/fast path, but would improve the performance of the GBN protocol under heavy congestion. First, we make use of the negative acknowledgement to serve as a quick recovery signal which is similar to the use of *triple-duplicate* acknowledgement in the Fast Retransmit [93] scheme of TCP protocol. By this method, the protocol optimistically retransmits all outstanding packets without waiting for the retransmission timer to expire. The rationale behind this scheme is, since the receiver receives some out-of-sequence packet(s), the network congestion problem may not be too severe, thus the sender tries to recover from the loss immediately.

If the network congestion problem is really harsh, the influx of retransmission packets will make it even worst as we are using the GBN strategy. In order to avoid wasting of bandwidth, when a communicating pair fails to resynchronize themselves by using the fast retransmission, the protocol refrains the sender by keeping it in the Stall state. Now the sender cannot send out any packet until the retransmission timers expire, thus, reduces the traffic load. With the decrease in network load, we hope that the congestion problem would be resolved soon, and the sender could resume the

transmission after idling for a timeout interval.

4.3 Congestion Loss under the Many-To-One Data Flow

In this section, we are exploring the contention behavior of our lightweight reliable transmission protocol on different buffering architectures under the same traffic pattern - the many-to-one flow. The function of a reliable protocol in the communication system is for prevention of and/or recovery from any data loss. Most of the data loss situations in modern networks are caused by dropping of packets in the switches, which in turn are induced by the contention problem. We believe that with a better understanding on how buffering within switches affect the performance of the communication system, especially under heavy contention situation, more effective congestion avoidance schedules can be designed for the cluster domain.

4.3.1 Many-To-One Data Flow

Although this communication pattern induces the heaviest congestion on the network link(s), the congestion behavior of this traffic pattern is easier to comprehend than that of the many-to-many communication pattern. The obvious result of this many-to-one traffic is the congestion build-up at the outgoing port(s), to which directly or indirectly connected to the gather root (common sink) of this traffic pattern. As the outgoing link is over-subscribed, excessive packets must be buffered. If the congestion persists for a “long” duration, congestion loss will be the result. However, if the volume of the data bursts were within the storage capacity associated with this congested port, no packet would be loss. Therefore, the circumstance that induces packet loss under such a traffic pattern is the arrival of a large burst of data packets targeting to the same outgoing port within a close interval, and the volume of this burst overshoots the storage capacity as well as the drainage speed of the congested port. Moreover, on steady-state condition, the incoming and outgoing flows should become balance since data traffics are regulated by the flow-control protocol. This is commonly known as the "self-clocking" [51] effect of the flow-control protocol. Therefore, we deduce that bursts of data packets are only generated by the sources either at the start of the many-to-one traffic or after any packet loss incident.

In the beginning, all sources have a full window, so packets are injecting into the network at full speed. Therefore, we observe that the volume of the first burst of data packets is proportional to the flow control window size (W_{FC}) and the number of concurrent senders (P). If the buffering capacity (B_L) of the bottleneck region along the route of this flow is not large enough to accommodate this burst, a "cluster of packets" is dropped. In this study, we assume that all switches employ the drop-tail FIFO discipline[2] [36] as the buffer management strategy; thus, when the buffer is full,

[2]This is the most commonly used dropping discipline in commercial products.

newcomers are discarded. Due to this dropping discipline, packet losses are exhibiting some form of temporal correlation. This is inferred from the following observations. First, if a packet arrives at a congested port is dropped by the switch, packets from different sources arriving to this bottleneck region in close apart are likely to be lost too. Second, if a particular data stream loses a packet, most of the subsequence packets originated from the same sender within the same window session are likely to be lost too. The implication of this packet loss behavior is that these sources will receive the packet loss signals (negative acknowledgments or timeouts) after sometime later but again in close apart. This triggers the error recovery layer to recover from the loss and induces another burst of data packets. Depends on the protocol used by the error recovery layer, this phenomenon may continue to evolve and results in poor network performance as the efficiency of the communication system deteriorates significantly.

When analyzing the contention behavior of the many-to-one traffic in a cluster interconnect, two features should be considered. First, congestion loss only occurs when the first burst of data packets overruns the switch buffer. If the buffer capacity can tolerate the flooding, we would not see any congestion loss problem. Second, the reliable transmission protocol has significant impact on the available throughput especially when congestion loss occurs. Therefore, the overall congestion behavior with respect to the many-to-one flow would be a combined effect of the buffering mechanism and the reliable transmission protocol in use.

For a switched network, the buffering's ability to tolerate congestion is determined by three factors:

1. The total amount of buffer memory in each switch.
2. The way how buffer memory is associated to individual ports.
3. The way buffer memory is organized to store packets of different sizes.

With an enclosed network, if we assume that all traffics are coming from the same application running on the cluster, we can safely narrow our investigation by assuming that the data packet size is fixed[3]. Therefore, we can eliminate the third factor and focus primarily on the effects of having different buffer architectures as on the congestion behavior. Therefore, our investigations are focusing on understanding the relationship between the buffering architecture of the switched network and the reliable transmission protocol on the congestion behavior. In particular, we would like to make use of this information to predict how much performance loss caused by the congestion.

4.3.2 Congestion Behavior on Input-Buffered Architecture

In this section, we are going to devise a simple empirical formula, which predicts the performance observed by the end-user, when suffered from contention loss on an input-buffered switch using the GBN scheme described in previous section. Assumed

[3]Normally, the traffic should have a bimodal distribution, one is peaked at the control packet size, which is of small size packet, and the other is at the maximum packet size.

that there are *P* data sources and the many-to-one flow starting at time $t = 0$, where all sources start sending their data packets more or less at the same time. Initially, all sources have an open window of size W_{FC} (in unit of fixed-size packet), and they continually send out their packets to the gather root. Congestion starts to build-up as we have more than one active sender, so packets are buffered at the dedicated memory associated to each ingress port - the input-buffered architecture. As the buffer size is finite which is of B_L unit, thus, at any given time $t > 0$, we observe that the queue size (Q_t) must satisfy this constraint, $0 \leq Q_t \leq B_L$. Furthermore, after the initial burst, all data flows should be regulated by the gather root under normal circumstances. Therefore, we could see that the congestion loss situation on the input-buffered architecture would happen only if the initial burst were larger than the buffering capacity, i.e. when $W_{FC} > B_L$. In other words, to avoid congestion loss on the many-to-one data flow under an input-buffered switch, one should have a window size (W_{FC}) which is smaller than or equal to the input port buffering size (B_L).

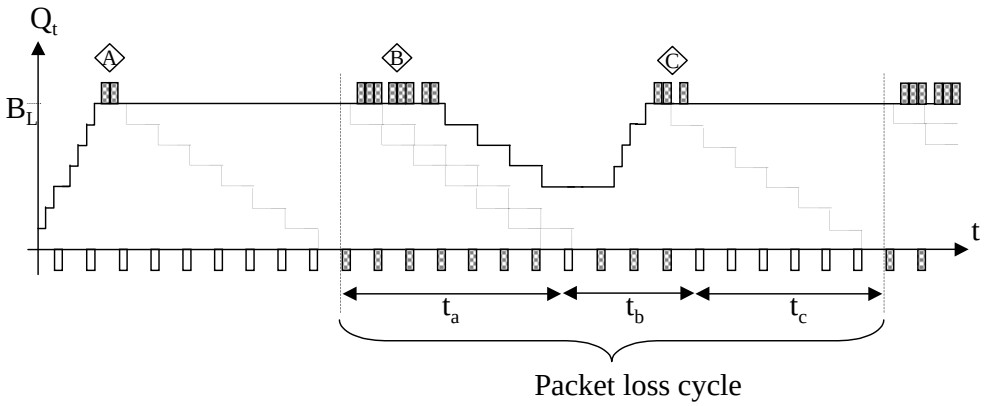


Figure 4.2: Evolution of the queue size (Q_t) over time on an input FIFO queue with congestion loss problem

As we are interested in studying how the congestion loss evolved under our GBN scheme on this architecture, so in order to study the congestion loss situation, we assume that $W_{FC} > B_L$ and each source has a steady stream of fixed-size data packets waiting to be transmitted to the gather root, i.e. all have unlimited amount of data to send. We also assume that the return paths for the acknowledgement packets are noiseless and congestion-free, so the data loss problem is mainly coming from the contention loss on the forward flows. To model the effective throughput delivered to the end-user by this many-to-one flow, we focus primarily on the evolution of a particular input queue, and generalize the result to all input queues involved in this flow. Figure 4.2 shows the queue size evolution of the input port buffer under such a congestion loss situation. Another assumption we have to make is on the departure rate of this buffering system. As there are P senders in this flow, and assume fair

scheduling on the switching engine, then the expected (average) departure rate $E[D]$ for each input queue would be of one packet per *P time slots*, with each time slot is measured as the time used by the egress port to transfer the fixed-size data packet, which can be approximated by the g_r parameter.

After the initial burst, as the departure rate of this buffering system is slower than the input rate and the volume of the burst is larger than the buffering capacity, packets at the end of the burst are dropped (label A in Figure 4.2). However, the sender wouldn't detect the loss until the first out-of-sequence packet reaches the gather root (start of period t_a), and stimulates a packet loss signal (Nack). As presented in Figure 4.1, the immediate response of the sender to the reception of the Nack is to transit to the *ReSend* state, and retransmits all outstanding packets (another burst of full window packets). Then it waits at the Stall state until it receives the first positive acknowledgement.

Under self-clocking principle, removal of one packet at the head of the queue will induce the arrival of another packet to fill the space. However, with an out-of-sequence packet, more than one data packets are injected to the network which causes another period of congestion loss (label B in Figure 4.2). For this burst, as there is only one buffer space left in the queue, thus, only the first packet can get a buffer slot. Since this is the expected packet that the gather root is waiting for, this results in changing the sender from the Stall state back to the normal state. In Figure 4.2, we have labeled the period starting from the detection of the first out-of-sequence packet to the time slot just before the reception of the correct retransmitted packet be period t_a. Within this period, all packets received by the gather root are discarded as they are out-of-sequence packets (the consequence of packet loss at label A). Moreover, since the retransmission burst (label B) only appears at the start of this period, and there is only limited space left behind in the queue, most of those packets are dropped. The queue size Q_t will gradually drop off as packets are continually drained away without replacement.

Although the sender transits from the Stall state back to the normal state at the start of period t_b, it immediately changes back to the ReSend state after the gather root receives the next packet, which is an out-of-sequence packet. Then another burst of packets is generated by the sender, which starts to fill all buffer memory again, and eventually induces another overflow situation (label C). We depict the period t_b to be a period between the two transitions of the sender from the Stall state back to the normal state. Like period t_a, all packets except the first packet received by the gather root in period t_b are out-of-sequence packets (the consequence of packet loss at label B). Switching back to the normal state marks the start of period t_c. Within this period, the gather root receives a sequence of in-order packets, which is the outcome of the retransmission burst happened during period t_b. And period t_c ends with the gather root switches back to the ReSend state when it detects an out-of-sequence packet, which is the consequence of the overflow situation at the end of the retransmission burst at label C.

When we carefully look at the evolution of the queue size as well as the changes of packet statuses, we could find that the packet sequences in periods t_a to t_c form a pattern that is recurred over time. Thus, we could simplify our analysis by focusing on the derivation of the throughput efficiency observed in this recurrent cycle - the *packet*

loss cycle. We define the throughput efficiency (T_{Eff}) to be

$$T_{Eff} = \frac{\tilde{I}}{\tilde{I} + \tilde{O}} \tag{4.1}$$

where Ĩ denotes the average number of in-order packets delivered to the gather root by this sender during a packet loss cycle, and Õ be the average number of error packets received but originated from this sender in the same period. To derive Ĩ and Õ, we need to consider the three subintervals in the packet loss cycle and count their corresponding number of good and error packets in each period. We have seen that period t_a is consisted of solely out-of-sequence packets. The start of this period marks the first error packet caused by the congestion loss at label A. Intuitively, this error packet must move forward $B_L - 1$ unit before it gets to the head of the queue. When the gather root receives this error packet, this causes the gather root to send back a Nack to the sender, which results in filling up the last buffer space by a good packet, and this good packet denotes the end of the period t_a. Therefore, we deduce that within the t_a period, the gather root receives B_L units of error packets.

Similarly, the error packets found in period t_b are induced by the retransmission burst at label B. As the expected arrival rate of this burst is one packet per time slot, while the average departure rate of this buffering system is one packet per *P* time slots, we deduct that only $\frac{W_{FC}-1}{P}$ packets could be buffered when they arrive to the input FIFO queue. However, all of these buffered packets are out-of-sequence packets. After this retransmission burst (label B), the sender transits to the Stall state. No more packets are sent to the queue until it changes back to the normal state at the start of period t_b, then the sender injects another packet to the queue which again is an out-of-sequence packet. Therefore, we could see that the expected number of good packets received by the gather root during period t_b is one packet, and the expected number of error packets received in this period is $\frac{W_{FC}-1}{P} + 1$ packets. Based on the same principle, we estimate the amount of good packets buffered within period t_c. Before the retransmission burst arrived to the FIFO queue at the start of period t_b, there are $\frac{W_{FC}-1}{P}$ error packets in the queue. So we would expect to have $\frac{B_L - \frac{W_{FC}-1}{P}}{1-\frac{1}{P}}$buffer space to accommodate those in-order packets before the buffer overflow situation happens again, and this becomes the amount of good packets received in period t_c. Thus, the throughput efficiency observed in a packet loss cycle becomes

$$\begin{aligned} T_{Eff} &= \frac{\frac{B_L - \frac{W_{FC}-1}{P}}{1-\frac{1}{P}} + 1}{\frac{B_L - \frac{W_{FC}-1}{P}}{1-\frac{1}{P}} + 2 + \frac{W_{FC}-1}{P} + B_L} \\ &\approx \frac{B_L + 1 - \frac{W_{FC}}{P}}{2B_L + 2 - \frac{B_L+2}{P}} \end{aligned} \tag{4.2}$$

Although T_{Eff} is derived by observing the dynamic behavior within a single input FIFO queue, if we assume that the same situation happens to all input FIFO queues

and each sender gets a fair share (i.e. $\frac{1}{P}$) of the available bandwidth, then the expected throughput efficiency observed on this many-to-one flow should be

$$T_{Eff}^{input} = P * (T_{Eff} * \frac{1}{P}) = T_{Eff} \tag{4.3}$$

In conclusion, from formula (4.2), we observe that with the congestion loss problem, the sustained throughput of this many-to-one flow would be less than 50% of the available bandwidth, since the denominator is at least twice as large as the numerator. As congestion loss problem only happens when $W_{FC} > B_L$, thus we just need to consider the effect of W_{FC} only. From the formula, we observe that the W_{FC} factor has a negative linear relation with the throughput efficiency, such that if we increase W_{FC}, we would expect to have a decrease in the sustained throughput. As for the factor P, it seems to have insignificant effect on the final throughput.

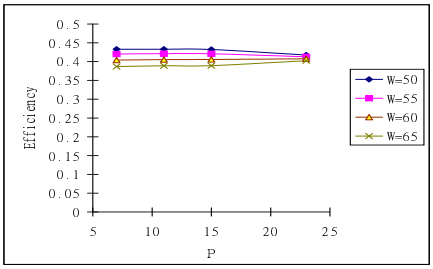
4.3.2.1 Experimental Evaluations

We validate T_{Eff}^{input} by using measurement data obtained from our experimental cluster platform. We use a cluster with 24 high-end PCs (PIII 733) connected by the IBM 8275-326 Fast Ethernet switch, which is revealed as an input-buffered architecture with $B_L = 43$ units per input port (Appendix A). To drive the Fast Ethernet network, we use the Directed Point (DP) low-latency communication system and implement the described Go-Back-N scheme to support reliability on top of DP. We have shown in Chapter 3 that with such a high-end cluster, the performance bottleneck falls on the network component.

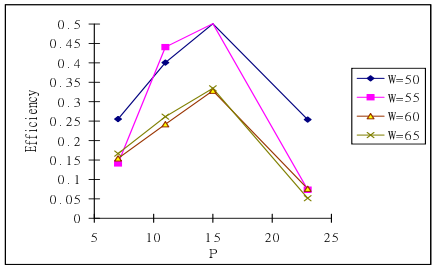
To simulate the many-to-one data flow under steady state streaming, we gathered all performance data by running the Gather collective operation with $P + 1$ processes and each process sends out 30000 full size data packets to the gather root, e.g. for $P = 15$, the total message length received by the gather root is approximately 640MB[4]. To induce the congestion loss problem, we started the experiments with $W_{FC} > B_L$ for various W_{FC}, P and timeout (TO) combinations, and observe their effects on the final performance. To simplify our analysis, we assume that the timeout value is set to a sufficient large value to avoid false retransmission[5]. Each test is conducted with P senders send out their messages "continuously" to the sole receiver and we measured how long would it take for all processes to complete this collective operation. By dividing the theoretical performance of the gather operation with the measured result, we calculate the throughput efficiency of this gather operation under congestion loss problem.

[4]The size of the received message is beyond the memory capacity of a single cluster node, which only has 128 MB of primary memory. To avoid paging overheads that affect the final throughput, all incoming data to the gather root are discarded immediately after protocol checking, thus without copying from the staging buffer to the user-space buffer. This avoids the need to create such a large receive buffer, and hence, avoids the paging overhead.

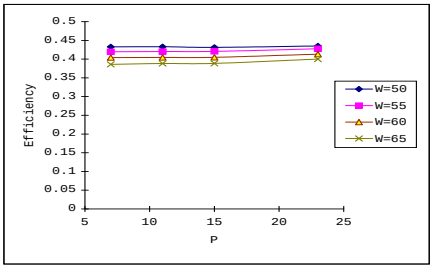
[5]The setting of timeout value is depended on the available network information. As this protocol is designed for communications on an enclosed network with negligible transmission error, we can make use of the g_s, g_r and B_L parameters to estimate the timeout setting, instead of using the round-trip



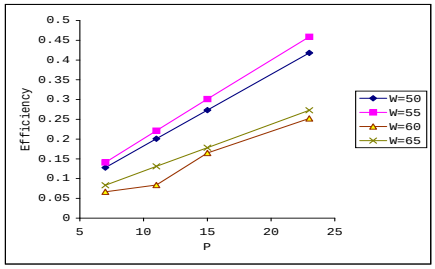
(a) our GBN - Timeout = 1000



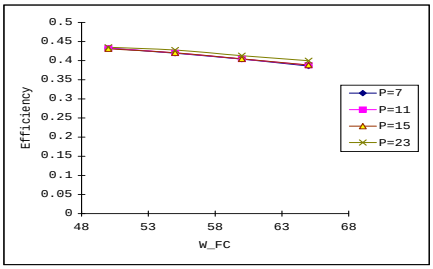
(b) simple GBN - Timeout = 1000



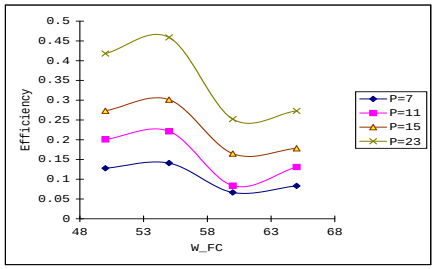
(c) our GBN - Timeout = 2000



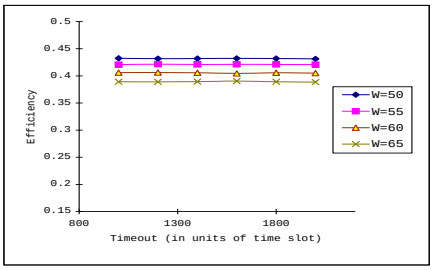
(d) simple GBN - Timeout = 2000



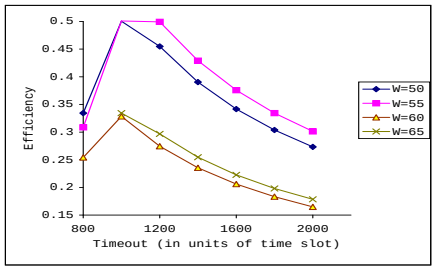
(e) our GBN - Timeout = 2000



(f) simple GBN - Timeout = 2000



(g) our GBN - P = 15



(h) simple GBN - P = 15

Figure 4.3: The measured throughput efficiency of our GBN reliable transmission protocol as compared to the measured throughput efficiency of the *simple* GBN scheme on the IBM 8275-326 switch.

The first set of experimental results is presented in Figure 4.3. In this figure, we are comparing the throughput efficiency (T_{Eff}^{input}) of our GBN scheme and a *simple* GBN scheme under the above experimental settings on an input-buffered switch. The simple GBN scheme is the classical version of the GBN scheme which does not have the fast retransmission mechanism. Therefore, whenever the receiver receives an out-of-sequence packet, it simply drops it and waits for the timeout retransmission. In general, we see that the performance of our GBN scheme is better than the simple GBN scheme, except on a few data points. Particularly, our GBN scheme is quite insensitive to the number of senders and the timeout parameter (Figure 4.3(a)(c)(g)), while the simple GBN scheme varies substantially with different *P*, W_{FC} and timeout parameters (Figure 4.3(b)(d)(f)(h)). With the simple GBN scheme, we observe that the timeout value is closely related to the number of senders (Figure 4.3(b)(d)). Such that with each P value, there is a specific timeout setting that fits it most, otherwise, the performance deteriorates considerably (Figure 4.3(h)). Another interesting observation on Figure 4.3 is that the achieved maximum throughput efficiency is no better than 50% of the available bandwidth on both GBN measurements.

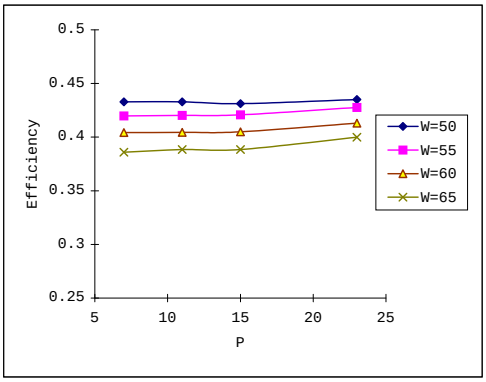
Figure 4.4 presents another comparison about the measured throughput efficiency of our GBN scheme with the predicted performance using Eq. 4.2. Although our predictions do not accurately match the measure performance, (the 95% confidence level of the prediction error is $13\% \pm 0.8\%$), our empirical formula does capture those salient features that we have described in previous subsection. First, when there is congestion loss problem, the measured performance is less than 45% of the available bandwidth, and the behavior is independent on the timeout setting if it is being set to a sufficient large value to avoid false alarm (shown in Figure 4.4(e)). This finding shows the importance of adopting congestion avoidance control in the first place as more than 50% of the performance is wasted. Second, both the measured and predicted results show that the throughput efficiency is only slightly affected by the number of senders, though we find that the throughput efficiency improves with increase in P (Figure 4.4(a)(c)). The measured results on different W_{FC} (Figures 4.4(c)) show that an inverse relation exists between the throughput efficiency and W_{FC}, however, the degradation in performance (slope) is greater than what we have estimated.

4.3.3 Congestion Behavior on Output-Buffered Architecture

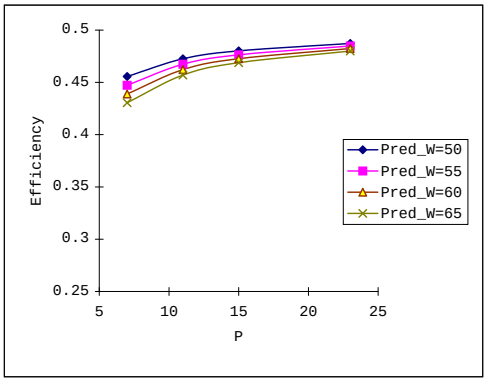
We have shown that with an input-buffered architecture, we lose more than 50% of the theoretical performance under congestion loss problem. In this section, we switch our analysis to the congestion dynamic happened on the output-buffered architecture under the same GBN ARQ scheme. Then based on these analytical studies, we compare on the performance differences between different buffering architectures.

With the output-buffered architecture, packets start to accumulate in the buffers associated to the egress port that leads to the gather root. As we assume that the buffer size is finite, at any given time $t > 0$, we observe that the queue size (Q_t) must

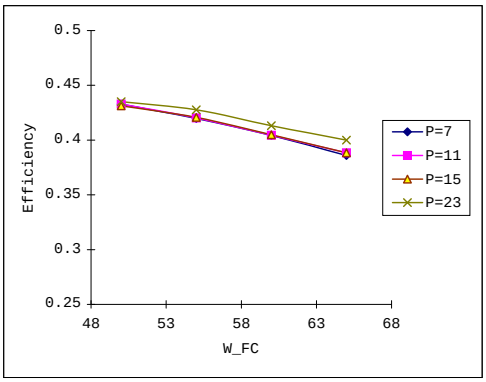
estimation.



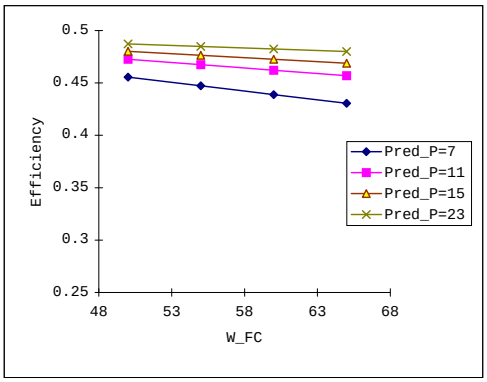
(a) Measured - Timeout=2000



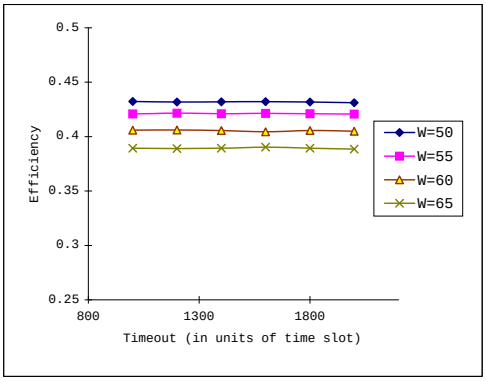
(b) Predicted -Timeout=2000



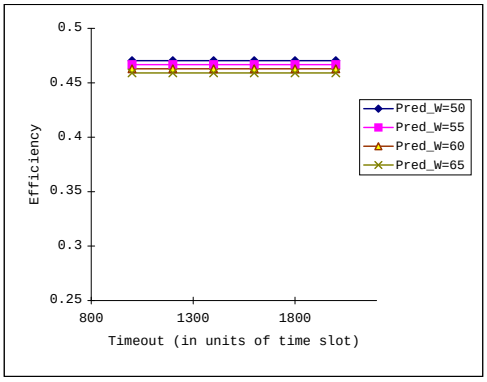
(c) Measured - Timeout=2000



(d) Predicted - Timeout=2000



(e) Measured - P=15



(f) Predicted - P=15

Figure 4.4: Comparisons of the measured and predicted performance on the congestion loss problem under input-buffered architecture with our GBN reliable transmission protocol.

satisfy this constraint, $0 \leq Q_t \leq B_L$. From this observation, we conclude that the congestion loss situation would happen on this architecture if and only if we have an initial burst of data packets which is larger than B_L, such that, when $W_{FC} * P > B_L$. As compare to input-buffered architecture with the same per port buffering capacity and under the same number of sources, the output-buffered architecture is more sensitive to the congestion loss problem under the many-to-one flow.

To study the steady state contention behavior, we assume that the communication event starts with $W_{FC} * P > B_L$ and all sources have unlimited amount of data to send. We also assume that our switched network adopts the drop tail discipline, and packets that arrive at a full buffer are dropped unconditionally. This dropping policy induces some form of temporally correlation amongst the senders, which results in creating wave of retransmission bursts. Under such scenario, some senders fall through to Stall state as their first packets cannot find an empty slot in the output queue, and thus, get into hibernation (i.e. remain inactive until retransmission timer expire). Some senders may alternate between cycles of Pack^i $\rightarrow$Nack events and eventually get into hibernation too. Only limited number of senders could get through this contention period and remain active until those slumbering senders receive their timeout signals and start another cycle of congestion. Therefore, if we collect the activity profile of a particular sender, we would see that its activities are cycling between Pack, Nack and Timeout events. Figure 4.5 shows a sample trace of the sender's activities which corresponds to such an arbitrary sequence of state transitions over time .

$$P^6\ N\ TO^2\ P^2\ N\ TO\ P^{32}\ N\ P^6\ N\ P^{106}\ N\ TO\ P^{13}\ N\ P^6$$

Figure 4.5: The sample trace of a sender's activities
(Legend: P^i- *i* consecutive *Packs*; N - *Nack*; TO^j - *j Timeout* events)

Based on the above observations, we model the congestion behavior of our reliable protocol in terms of *activity cycles* of the sender, with each cycle is characterized by some recurrent patterns which we believe, are statistical independent but are probabilistic replicas of one another. A sample cycle is given in Figure 4.6 which is an abstract representation of the sample trace shown in Figure 4.5. The cycle begins with the transition of the sender from the Stall state back to the normal state by receiving a Pack response after a series of the timeout retransmissions[6]. After getting back to the normal state, the sender continues with a sequence of recurrent pattern, with each pattern composes of a series of Pack events and ends on a Nack event. Each PNP transition corresponds to the detection of packet loss situation, but successfully recovers by fast retransmission. However, under severe congestion, fast retransmission would

[6]Although our reliable protocol implements a timeout timer for each outstanding packet, in this model, a timeout retransmission event corresponds to a batch retransmission of all outstanding packets, instead of the individual retransmission. This is because, our reliable protocol is implemented in the user-space, and therefore the inter-message submission gap is governed by the O_s parameter. Since under normal circumstance, $O_s < g_s$, we can safely assume that the individual retransmission timers are coalesced to form one single batch retransmission timer.

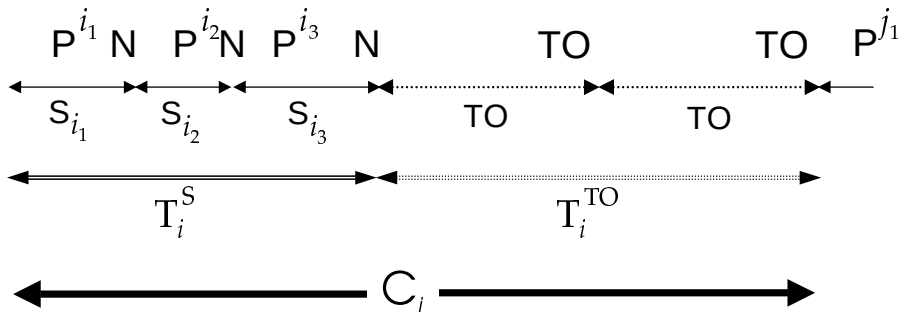


Figure 4.6: A typical activity cycle of a sender which is composed of a sequence of recurrent patterns

not help, then the sender falls through to the Stall state and waits until next timeout signal. This is reflected in the activity cycle as a series of TO events that follow the Nack event. Finally, the cycle is terminated by the last timeout retransmission, which marks the onset of the next activity cycle.

Now we are going to derive an analytical model that captures the throughput efficiency observed by a particular sender under this buffering architecture. Let T_i^{TO} denotes the duration of a sequence of timeout events and T_i^S be the time interval between two consecutive timeout sequences. When adding up these two intervals, we have $C_i = T_i^S + T_i^{TO}$, which becomes the elapsed time of a particular activity cycle. We also define U_i be the number of in-order packets received by the gather root with respect to this sender during this C_i interval. By viewing $\{(U_i, C_i)\}_i$ as a stochastic sequence of random variables, we define the throughput efficiency be

$$T_{Eff} = \frac{E[U]}{\frac{E[C]}{g_r}} = \frac{E[U] * g_r}{E[C]}$$

where E[U] denotes the expected number of packets accepted by the gather root in one activity cycle and E[C] be the expected duration of one activity cycle. Without having packet loss, we expect that the gather root can handle one packet per g_r time units, thus, the maximum number of packets that a the gather root can handle during E[C] time units is $\frac{E[C]}{g_r}$packets.

To derive E[U], we have to estimate the total number of Pack responses received within those $P^{i_n}N$ sequences. Let y_i be the number of $P^{i_n}N$ sequences in the interval T_i^S. For the k-th $P^{i_n}N$ sequence, we define x_{i_k} to be the number of Pack responses received in that period, S_{i_k} to be the duration of that period. Assume if the number of Pack responses received in one $P^{i_n}N$ sequence is statistically independent to the number of Pack responses in another $P^{i_n}N$ sequence, then we can view $\{x_{i_k}\}$ as a sequence of independent and identically distributed (i.i.d) random variables. The same assumption can be applied to y_i and S_{i_k}. Now we have

$$U_i = \sum_{k=1}^{y_i} x_{i_k} \;\Rightarrow\; E[U] = E\left[\sum_{k=1}^{y_i} x_{i_k}\right] \;\Rightarrow\; E[U] = E[y] * E[x]$$

$$T_i^S = \sum_{k=1}^{y_i} S_{i_k} \;\Rightarrow\; E[T^S] = E\left[\sum_{k=1}^{y_i} S_{i_k}\right] \;\Rightarrow\; E[T^S] = E[y] * E[S]$$

As for deriving E[C], let m_i be the number of timeout periods in the interval T_i^{TO} and since the duration of each timeout period is fixed, we have

$$C_i = T_i^S + T_i^{TO} \;\Rightarrow\; E[C] = E[T^S] + E[T^{TO}] \;\Rightarrow\; E[C] = E[y] * E[S] + E[T^{TO}]$$

Assume that the number of timeout periods in each activity cycle is also an i.i.d random variable, then we have

$$E[C] = E[y] * E[S] + E[m] * TO$$

and thus,

$$T_{Eff} = \frac{E[y] * E[x] * g_r}{E[y] * E[S] + E[m] * TO} \tag{4.4}$$

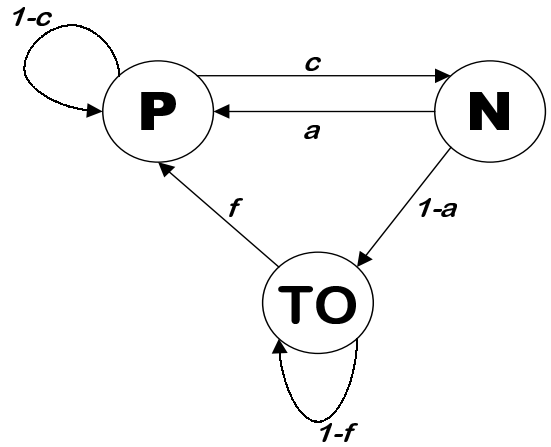


Figure 4.7: 3-state Markov chain model

To derive E[y], E[x] and E[m], we make use of concepts from discrete-time Markov chain model [50] to model the activity profile of a sender. A 3-state Markov chain model (see Figure 4.7) is constructed which represents the three different events happened in a sender profile, they are **P** - Pack, **N** - Nack and **TO** - timeout events. With reference to the state transition diagram (Figure 4.1) of our GBN protocol, a Pack event means working in normal state or transit to normal state, a Nack event means transit to the ReSend state (fast retransmission), and the TO event means slumbering in the Stall

state but being waked up by the timeout signal. However, transition from one event status to another means the detection of or recovery from some loss events. Therefore, the transition probabilities between certain events are directly related to the loss probabilities on the congested path.

Hence, we define the stochastic process in which this 3-state loss model is based on as, $X_i(\omega)$ = the type of events delivered to the sender at the time of receiving the i^{th} event with a finite state space, $\ddot{S} = \{P, N, TO\}$, and the corresponding *transition probability matrix* is

$$\begin{bmatrix} 1-c & c & 0 \\ a & 0 & 1-a \\ f & 0 & 1-f \end{bmatrix},$$

with individual row represents

First row Transition probabilities from state P to state ω for $\omega \in \ddot{S}$, such that $Pr[X_i = P \mid X_{i-1} = P] = 1 - c$, $Pr[X_i = N \mid X_{i-1} = P] = c$, and $Pr[X_i = TO \mid X_{i-1} = P] = 0$.

Second row Transition probabilities from state N to state ω for $\omega \in \ddot{S}$, such that $Pr[X_i = P \mid X_{i-1} = N] = a$, $Pr[X_i = N \mid X_{i-1} = N] = 0$, and $Pr[X_i = TO \mid X_{i-1} = N] = 1 - a$.

Third row Transition probabilities from state TO to state ω for $\omega \in \ddot{S}$, such that $Pr[X_i = P \mid X_{i-1} = TO] = f$, $Pr[X_i = N \mid X_{i-1} = TO] = 0$, and $Pr[X_i = TO \mid X_{i-1} = TO] = 1 - f$.

And the transition probabilities *(1-a)*, *c* and *(1-f)* reflect the different loss probabilities experience by a particular sender during different congestion stages. The probability *c* measures the likelihood of a sender to encounter a lost packet event, so that a high value for it would indicate that the congestion loss problem occurs frequently. On the other hand, the probability *(1-a)* measures the severity of the congestion problem, so a high value means the competition is indeed fierce. In addition, a high value of *(1-f)* would indicate a high degree of clustering of retransmission bursts, which means a high degree of temporal correlation between losses.

Now we first derive E[x]. Given our loss model and the assumption of being a *stationary*[7] discrete-time Markov chain, the probability that $x_{i_k} = n$ is equal to the probability that there appears to have n P events before a PN transition occurs. Thus,

$$Pr[x_{i_k} = n] = (1-c)^{n-1}c \qquad \text{for n} = 1, 2, \ldots$$

and the expected value of x is

[7]The probability of going from one state to another is independent of the time at which the step is being made [50].

$$E[x] = \sum_{n=1}^{\infty}(1-c)^{n-1}cn = \frac{1}{c} \tag{4.5}$$

Likewise, the probability that $y_i = h$ is equal to the probability that there appears to have $h-1$ consecutive $P^{i_n}N$ sequences before a Timeout event occurs, that is

$$Pr[y_i = h] = a^{h-1}(1-a) \qquad \text{for } h = 1, 2, \ldots$$

and the expected value of y becomes

$$E[y] = \sum_{h=1}^{\infty} a^{h-1}(1-a)h = \frac{1}{1-a} \tag{4.6}$$

And the probability mass function and the expected value of the random variable m are

$$Pr[m_i = q] = (1-f)^{q-1}f$$

$$E[m] = \sum_{q=1}^{\infty}(1-f)^{q-1}fq = \frac{1}{f} \tag{4.7}$$

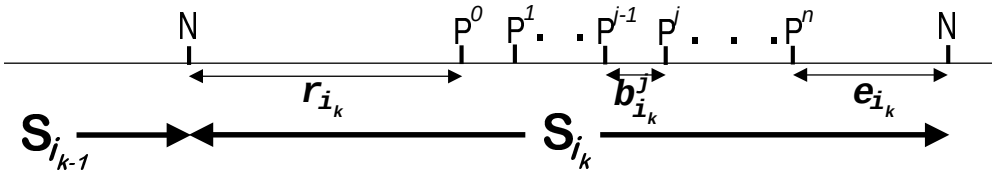


Figure 4.8: Events happened in a typical $P^{i_n}N$ sequence

Finally, to derive an expression for E[S], we look into the time scale of a particular $P^{i_n}N$ sequence (see Figure 4.8). Let $b_{i_k}^j$ be the time gap between the appearances of *j-1*th and the j^{th} Pack events; e_{i_k} be the time gap between the last Pack event and the Nack event; and r_{i_k} be the elapsed time between the Nack event of the last $P^{i_n}N$ sequence and the first Pack event of current sequence. Then we can express S_{i_k} be

$$S_{i_k} = r_{i_k} + \sum_{j=1}^{x_{i_k}-1} b_{i_k}^j + e_{i_k}.$$

For simplicity, we also assume that e_{i_k}, r_{i_k} and $b_{i_k}^j$ are random variables and are independent on each other, as well as assume that the sequence of Pack events is modeled as a Poisson process, thus $b_{i_k}^j$ can be modeled as exponentially distribution with parameter λ. It follows that the expected value of S become

$$E[S] = E[r] + E\left[\sum_{j=1}^{x_{i_k}-1} b_{i_k}^{j}\right] + E[e]$$

$$E[S] = E[r] + (E[x] - 1)E[b] + E[e]$$

From Figure 4.8, we see that r_{i_k} is the time lapsed between a Nack event and a Pack event. When a sender receives an Nack event, it responses by re-sending all outstanding packets, and we would expect that the next Pack event is triggered by the reception of the first packet from this ReSend burst. Therefore the elapsed time between this Nack/Pack pair depends on the current queue size (Q_t) as the first packet of the ReSend burst needs to move forward to the head of the queue before a Pack event is generated. Since Q_t is bounded by B_L, and under steady state condition on the many-to-one flow, the buffer queue should operate under almost full condition. Thus, we have

$$E[r] \approx B_L * g_r.$$

To derive $E[b]$, which is equal to λ, we have to determine the average inter-arrival time λ of the Pack events in a $P^{i_n}N$ sequence. Consider that if there is no congestion loss problem, all (P) senders get a fair share of the bandwidth, we would expect that the average inter-arrival time observed by a particular sender be $P * g_r$. However, under congestion loss situation with our reliable protocol, not all senders are in active sending mode as some are forced to stay in hibernation condition. Hence we postulate that the expected number of senders ($E[A]$) be $\frac{B_L}{W_{FC}}$, which is derived on the ground that with a buffering capacity of B_L units, after vigorous competition, on average only $E[A]$ senders can win a place for their full window load of packets. Therefore, we have

$$E[b] = E[A] * g_r = \frac{B_L * g_r}{W_{FC}}.$$

We follow the same logic to deduce $E[e]$. Remember that a Pack is sent to a sender only when the gather root receives an in-order packet from that sender. So the inter-arrival time ($E[b]$) between Pack events has a direct relationship with the elapsed time between consecutive packets from the same sender. To determine $E[e]$, we have to estimate how many packets from this sender are lost before the gather root detects this loss problem. If we consider the loss is uniformly distributed between 1 and $W_{FC} - 1$, then on average, we would expect to have $\frac{W_{FC}-1}{2}$lost packets before the gather root detects the loss situation. Since each consecutive packet is separated by $E[b]$ time units, $E[e]$ becomes

$$E[e] = \frac{W_{FC} - 1}{2} * E[b] = \frac{(W_{FC} - 1) * B_L * g_r}{2W_{FC}}.$$

Now we have

$$
\begin{aligned}
E[S] &= B_L * g_r + (\frac{1}{c} - 1)\left(\frac{B_L * g_r}{W_{FC}}\right) + \frac{(W_{FC} - 1) * B_L * g_r}{2W_{FC}} \\
&\approx \left(\frac{3}{2} + \frac{1-c}{cW_{FC}}\right) * B_L * g_r
\end{aligned} \tag{4.8}
$$

By substitute (4.5), (4.6), (4.7) and (4.8) to (4.4), we have

$$
\begin{aligned}
T_{Eff} &= \frac{g_r}{\left(\frac{3c}{2} + \frac{1-c}{W_{FC}}\right) * B_L * g_r + \frac{(1-a)*c*TO}{f}} \\
&= \frac{1}{\left(\frac{3c}{2} + \frac{1-c}{W_{FC}}\right) * B_L + \frac{(1-a)*c*\overline{TO}}{f}}
\end{aligned} \tag{4.9}
$$

where $\overline{TO} = \frac{TO}{g_r}$. As T_{Eff} represents the throughput efficiency observed by one particular sender, then the aggregate performance of this many-to-one flow under congestion loss problem becomes

$$
T_{Eff}^{output} = P * T_{Eff}. \tag{4.10}
$$

The above empirical equation (4.9) for predicting the throughput efficiency on the output-buffered architecture is formulated in terms of 3 transition probabilities - *a*, *c* and *f*. From the formula, we observe that transition probability *c* has a significant role on the final performance, such that if we can find some way to minimize *c*, the higher throughput we get. To utilize this equation, we need to provide some methods to determine these probabilities. However, we cannot identify a clean association between our target performance parameters (W_{FC} , P, B_L & *TO*) and those transition probabilities. Consequently, we have to rely on some inferential approach [50] to statistically estimate these transition probabilities, which is commonly used in other performance studies [63, 74, 112, 119]. In particular, we look at the information gathered from a sample trace and use this information to estimate on the transition probabilities. For examples, in our case, we have

$$
\tilde{c} = \frac{\text{total number of } PN \text{ transitions}}{\text{total number of } P} \tag{4.11}
$$

$$
\widetilde{1-a} = \frac{\text{total number of transitions from } N \text{ to } TO}{\text{total number of } P^{i_n}N \text{ sequences}} \tag{4.12}
$$

$$
\tilde{f} = \frac{\text{total number of transitions from } TO \text{ to } P}{\text{total number of } TO} \tag{4.13}
$$

4.3.3.1 Experimental Evaluations

A series of experiments are conducted to validate the above empirical formula. For all of these tests, the same cluster that we have used in Subsection 4.3.2.1 is used, however, the number of cluster nodes involved depends on the configuration of the switch or at most 32 nodes. Furthermore, the same GBN reliable transmission protocol and the DP package are used. The first set of experiments is conducted with this cluster interconnected by the 16-port IBM 8275-416 Ethernet switch. Under our benchmark tests, this switch is revealed to have an output-buffered architecture with $B_L = 95$ units per output port. As this switch can only support 16 full 100 Mb/s connections, we use a cluster size of 16 nodes to conduct all tests. The second set of experiments are conducted by interconnecting this cluster with the Cisco Catalyst 2980G Ethernet switch, which has 80 Fast Ethernet ports and 2 Gigabit Ethernet ports. By applying our benchmark tests on this switch, we uncover that the switch internal is adopting an output-buffered allocation scheme on a shared memory architecture[8] (with $B_L = 128$ units per output port). We only connect 32 cluster nodes to this switch as this is the maximum size we have for this homogeneous cluster.

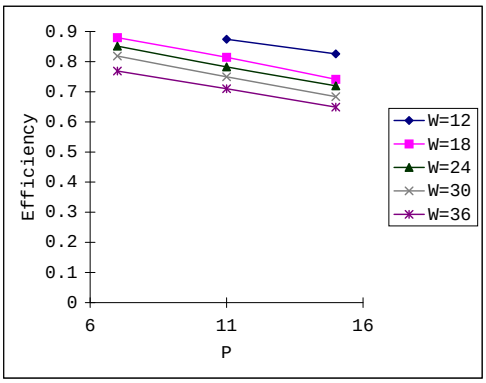
We employ a similar testing methodology as appeared in Subsection 4.3.2.1 on each platform, however, to induce the congestion loss problem, we have a different window sizing constraint, that is $W_{FC} * P > B_L$. Since the two switching platforms have different buffering capacity and supported port number, thus the extent of varying the W_{FC} and P parameters are different too. In order to apply our empirical formula (Eq. 4.9), we collect activity traces from those processes during the tests. We obtain the measured performance on different parameter settings (different combinations of W_{FC}, P and TO) by running the same tests for 30 iterations and take the average timing as the result measurement. To minimize the required runtime memory to store the activity traces, only the activity traces of all senders on the last iteration are returned. Then the corresponding transition probabilities are calculated for each sender, and finally, the mean values from all the senders are taken as the transition probabilities for this particular parameter set. Table 4.1 displays a sample set of data collected from the IBM 8275-416 platform by this method.

We start the analysis of our GBN scheme by comparing its performance with the simple GBN scheme (as described in Subsection 4.3.2.1) on the IBM 8275-416 switch, and the results are shown in Figure 4.9. In contrast with the input-buffered architecture, we find that our GBN scheme works effectively on the output-buffered architecture, while the simple GBN scheme performs extremely inefficient on this architecture. The measured results on the simple GBN scheme show that its performance is insensitive to the *P* and W_{FC} parameters, but is slightly depended on the timeout setting. The major difference between our GBN scheme and the simple GBN scheme is on the existence of fast retransmission mechanism. However, fast retransmission is simply a stochastic approach on improving the efficiency, as each sender attempts to recover from the loss before falling through to the Stall state. Through this kind of random selection, some

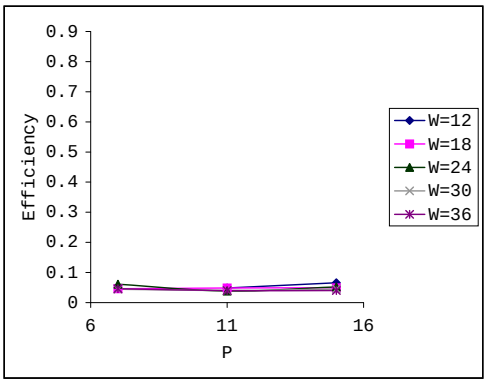
[8]On the supporting document of this switch [99], Cisco claims that this switch has a low-latency, centralized, shared memory switching fabric architecture.

P	W_{FC}	$\overline{TO}$	$\tilde{c}$	$\widetilde{1-a}$	$\tilde{f}$	*Measured* T_{Eff}^{output}	*Predicted*
7	18	400	0.0218	0.0547	0.5329	0.8375	0.7639
7	24	400	0.0216	0.1383	0.5300	0.7846	0.7598
7	36	400	0.0227	0.2344	0.5420	0.6971	0.7189
11	18	400	0.0357	0.2097	0.6109	0.7345	0.7294
11	24	400	0.0338	0.2863	0.5763	0.6947	0.7162
11	36	400	0.0305	0.3956	0.5207	0.6380	0.6798
15	18	400	0.0495	0.2907	0.5346	0.6309	0.6573
15	24	400	0.0441	0.3674	0.4977	0.5975	0.6501
15	36	400	0.0341	0.4675	0.4053	0.5596	0.6477
7	18	1600	0.0112	0.0578	0.6810	0.8929	0.8397
7	24	1600	0.0118	0.0834	0.6091	0.8822	0.8550
7	36	1600	0.0126	0.1106	0.5628	0.8027	0.8363
11	18	1600	0.0202	0.0931	0.6390	0.8366	0.8621
11	24	1600	0.0163	0.1522	0.6444	0.8222	0.8884
11	36	1600	0.0164	0.1798	0.5789	0.7516	0.8390
15	18	1600	0.0204	0.1952	0.6922	0.7976	0.8670
15	24	1600	0.0174	0.2230	0.5918	0.7785	0.8905
15	36	1600	0.0175	0.2635	0.5641	0.6877	0.8238

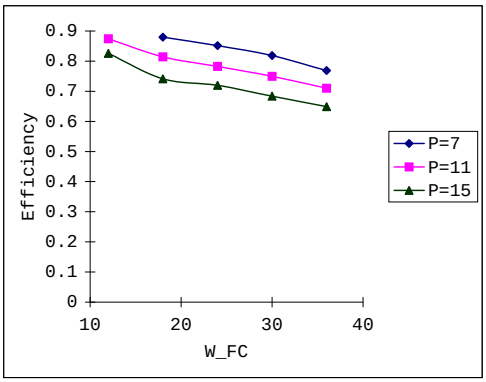
Table 4.1: Sample data collected on the 416 platform



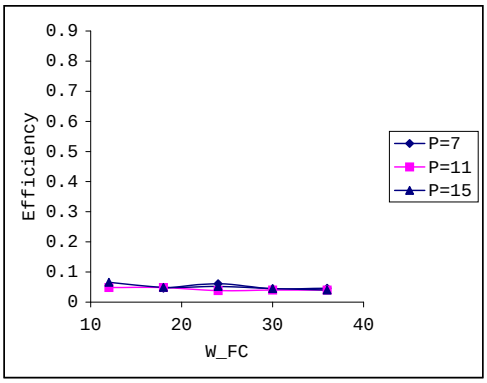
(a) our GBN - Timeout = 1000



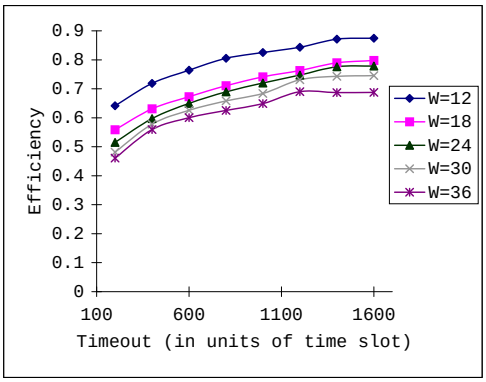
(b) simple GBN - Timeout = 1000



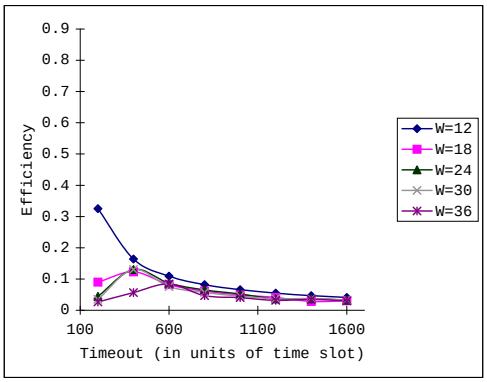
(c) our GBN - Timeout = 1000



(d) simple GBN - Timeout = 1000



(e) our GBN - P = 15



(f) simple GBN - P = 15

Figure 4.9: Comparing the performance of our GBN scheme with the simple GBN scheme on the IBM 8275-416 switch

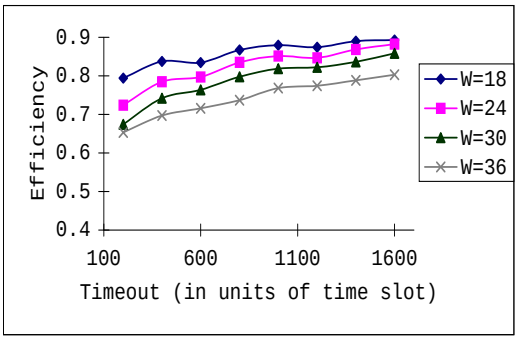
senders get their chances in continuing their transmissions and effectively utilize the bandwidth, while the rest have to wait for their chances after a timeout interval.

In Figures 4.10 and 4.11, we have the comparisons of the measured and predicted performance of our GBN scheme on the IBM 8275-416 platform for various parameter sets. In general, the empirical formula (Eq. 4.10) correctly reflects the macroscopic behavior of our GBN reliable transmission protocol on the output-buffered architecture. Specifically, both the measured and predicted results (Figure 4.10) show that having a larger *TO* setting improves the throughput efficiency. This behavior is a result of the addition of the fast retransmission mechanism, which randomly selects a few senders to continue and denies the rest. Besides the *TO* parameter, we also observe that both the W_{FC} and P parameters are inversely related to the throughput efficiency.

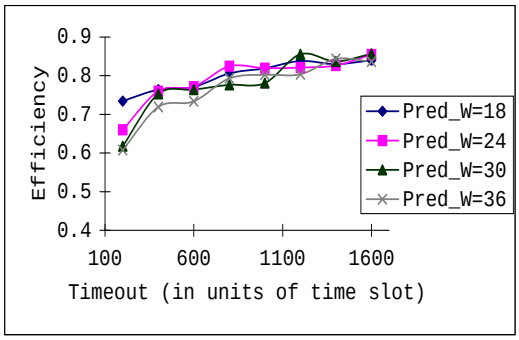
On the other hand, observing from the results shown in Table 4.1 and Figure 4.11, we find that the accuracy of our predictions is deteriorating along with the increase in P, W_{FC} and TO, albeit the low error rate (the 95% confidence level of the prediction error falls on $7\% \pm 0.9\%$). When the timeout setting is small (subgraphs (a),(b),(c)&(d) of Figure 4.11), we clearly see that our empirical formula correctly reflects the relationships between P, W_{FC} & T_{Eff}^{output}, but when the timeout setting is large, the relationships become unclear. This is because, when look into the data shown in Table 4.1, our empirical formula tends to over-estimate the improvement made by the increase in timeout parameter.

Figure 4.12 shows the results of the second set of experiments using the Cisco Catalyst 2980G switch with the 32-node cluster. The same findings could be observed from this set of data as compared to the IBM 8275-416 case, but the accuracy of our predictions on this platform is better than that of the IBM 8275-416 case (the 95% confidence level of the prediction error is $6\% \pm 0.7\%$). To uncover the performance difference between the two switches, we extract performance results from these two sets which are collected with the same parameter settings, and present them in Table 4.2. In general, we find that increase in buffering capacity would improve the congestion performance. However, it is interesting to see that the Cisco switch, which has a larger buffering capacity, could result in having poorer congestion performance than the 416 switch on some scenarios.

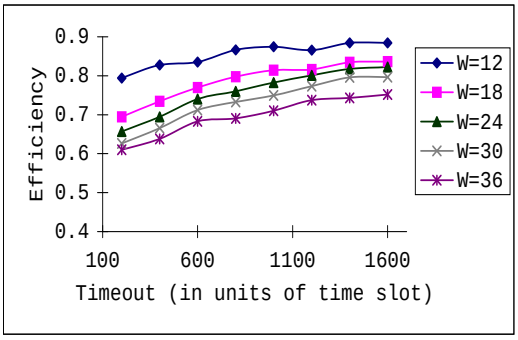
Our assumption on the timeout (TO) setting is that it should be sufficiently large to avoid false retransmission. Therefore, for the timeout setting on all tests, it satisfies this constraint - $\overline{TO} \geq 2B_L$. This ensures that the timeout timer is set to a value larger than the round-trip delay on a saturated network; and if the timer expires, this highly indicates that the network is under heavy congestion. To quantify the effect of the timeout parameter on the throughput efficiency, we intentionally include timeout settings that are out of our assumed range, and the result is shown in Figure 4.13. We observe that the throughput efficiency is in a logarithmic scale with the timeout parameter. This finding supports our initial assumption and indicates that the performance is severely degraded by the false retransmission phenomena. On the other hand, the observed performance does not improve a lot even if we use a very large timeout setting. This justifies that picking a reasonable large timeout setting is sufficient to guard against the congestion loss problem.



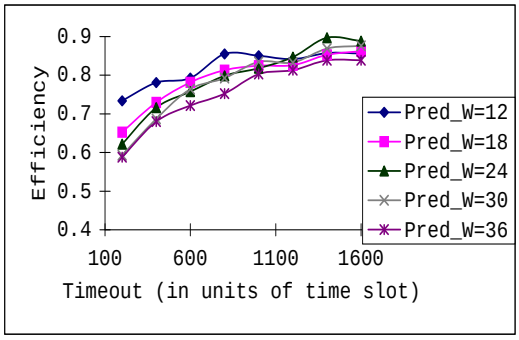
(a) Measured - P=7



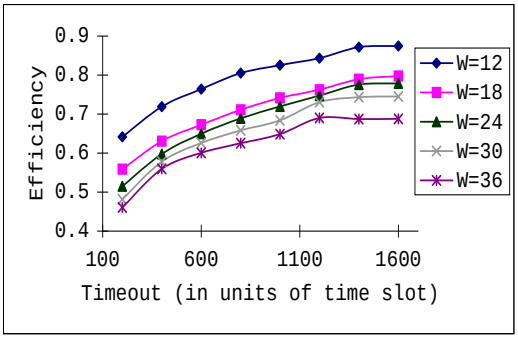
(b) Predicted - P=7



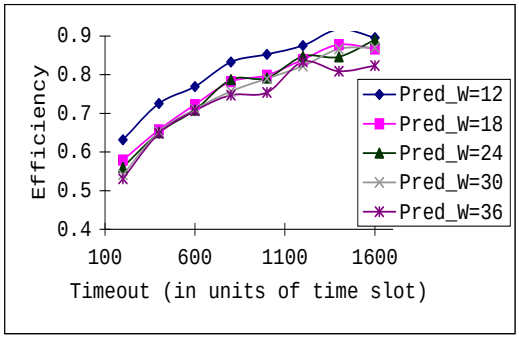
(c) Measured - P=11



(d) Predicted - P=11

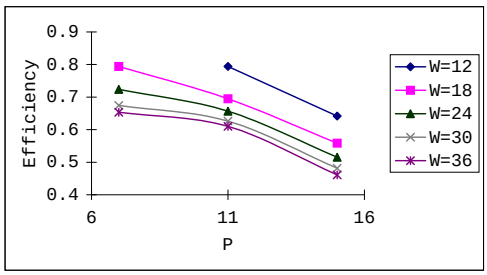


(e) Measured - P=15

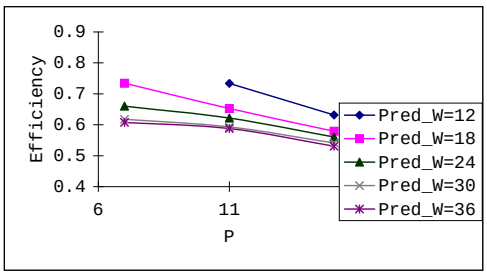


(f) Predicted - P=15

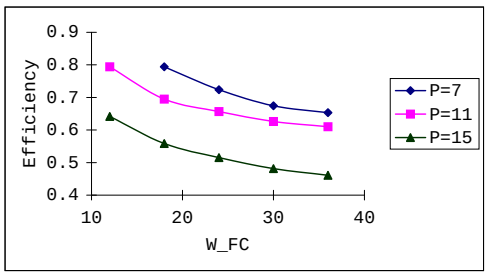
Figure 4.10: Comparison of the measured and predicted performance of the IBM 8275-416 switch under heavy congestion loss problem with our GBN reliable transmission protocol subjected to different timeout settings



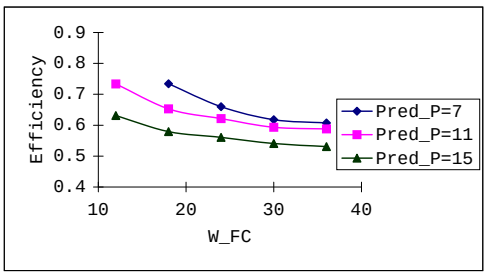
(a) Measured - Timeout=200



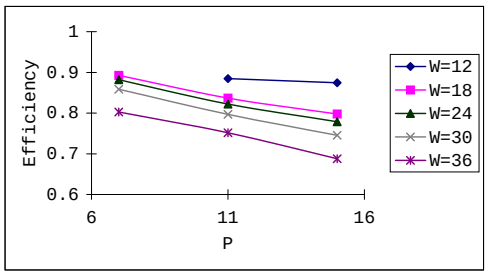
(b) Predicted - Timeout=200



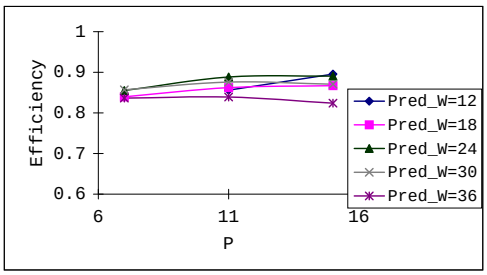
(c) Measured - Timeout=200



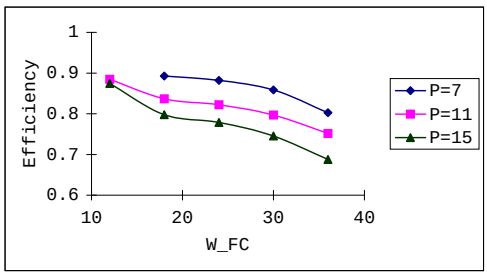
(d) Predicted - Timeout=200



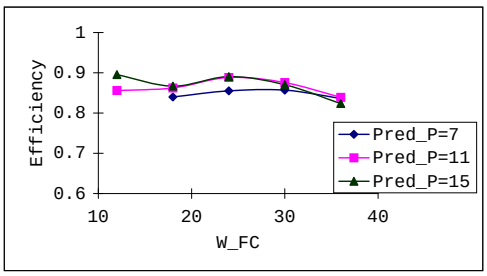
(e) Measured - Timeout=1600



(f) Predicted - Timeout=1600

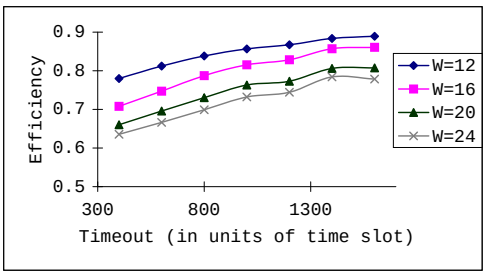


(g) Measured - Timeout=1600

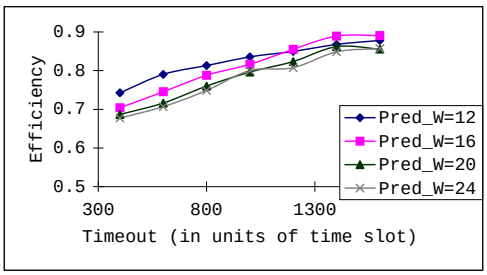


(h) Predicted - Timeout=1600

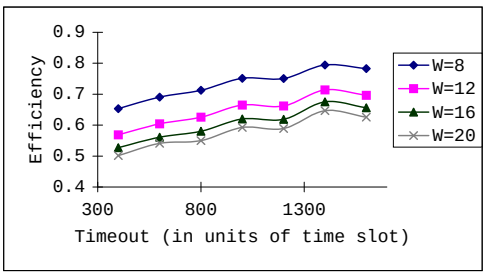
Figure 4.11: Comparisons of the measured and predicted performance of the IBM 8275-416 switch with our GBN reliable transmission protocol. The main focus is on revealing the effects of the P and W_{FC} parameters on the final performance.



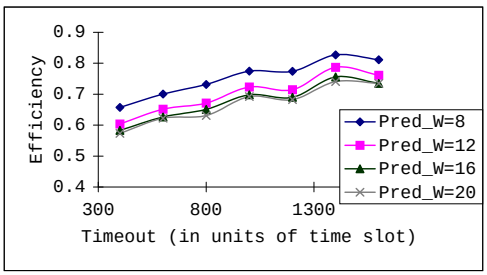
(a) Measured - P=15



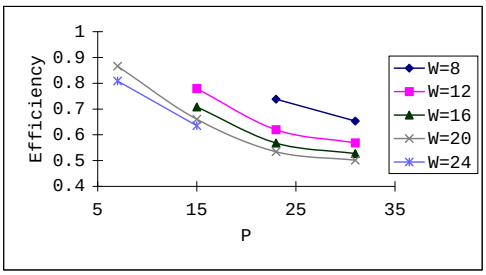
(b) Predicted - P=15



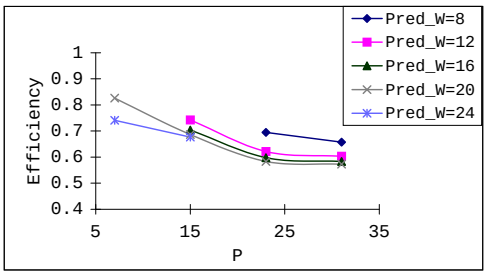
(c) Measured - P=31



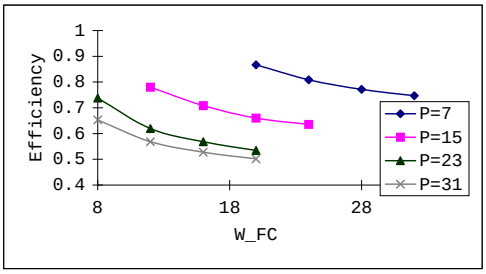
(d) Predicted - P=31



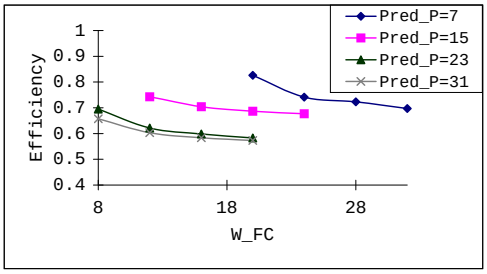
(e) Measured - Timeout=400



(f) Predicted - Timeout=400



(g) Measured - Timeout=400



(h) Predicted - Timeout=400

Figure 4.12: Comparisons of the measured and predicted performance of the Cisco Catalyst 2980G switch under heavy congestion loss problem with our GBN reliable transmission protocol

Switch	B_L	P	W_{FC}	$\overline{TO}$	Measured	Predicted
IBM416	95	7	24	400	0.7846	0.7598
Cisco2980	128	7	24	400	0.8088	0.7413
IBM416	95	7	24	1600	0.8822	0.8550
Cisco2980	128	7	24	1600	0.8602	0.8046
IBM416	95	15	24	400	0.5975	0.6501
Cisco2980	128	15	24	400	0.6354	0.6772
IBM416	95	15	24	1600	0.7785	0.8905
Cisco2980	128	15	24	1600	0.7785	0.8571
IBM416	95	15	12	400	0.719	0.7256
Cisco2980	128	15	12	400	0.7797	0.7427
IBM416	95	15	12	1600	0.8745	0.8956
Cisco2980	128	15	12	1600	0.8891	0.8782

Table 4.2: Comparisons of the congestion behavior observed on the IBM416 and the Cisco2980 switches under the same parameter sets

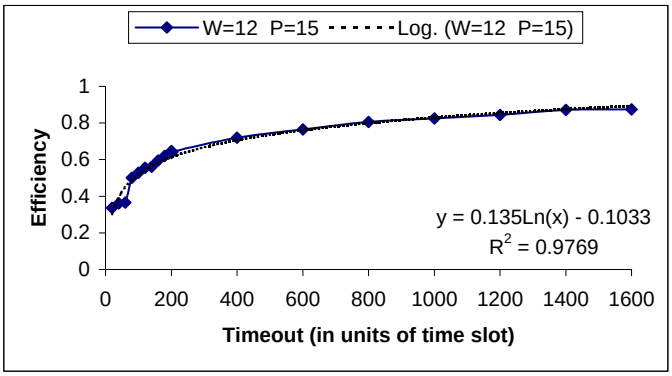


Figure 4.13: Effect of the timeout (TO) parameter on the throughput efficiency when the network is under heavy congestion loss problem. The data are collected on the IBM 8275-416 platform with $P = 15$, $W_{FC} = 12$.

4.3.4 Discussion of the Models and Their Implications

In previous subsections, we have examined on the performance behavior of two buffering architectures under heavy congestion loss. Regards to the buffering architectures, another commonly used buffering scheme should also be considered - the shared-buffered [110] or common output-buffered architecture [42]. This scheme makes use of dynamic buffer allocation from a common pool of buffers, and therefore each output port virtually has a larger buffer storage. The main advantages of this scheme as compared to output-buffered architecture are (1) the higher buffer utilization can be achieved and (2) a smaller total buffering capacity is required. However, to avoid unfair utilization, most implementations set up an upper limit on each output port. Due to the architectural similarity between the shared-buffered and the output-buffered schemes, we consider that the above analytical study on the output-buffered scheme is directly applicable to the shared-buffered architecture.

We have two models that describe the congestion behavior of our communication system under different buffering architectures; however, one can show that the 3-state Markov chain model can be used to derive the input-buffered case, if we view the input-buffered case as a special case of this Markov chain model, where $a = 1$ & $c = \frac{2P-2}{B_L * P + P - W_{FC}}$. Although using stochastic process to model system dynamic is a powerful technique, it has some known limitations. For some cases, to capture real world phenomena but still keeping the model equation tractable, one has to rely on some known and well-behaved statistical or probabilistic models. Then, the question becomes how close are these assumptions matched with the reality? Besides, we may find that on some cases, there exists no general probability model that suits for our needs. Then one has to collect information from the running system, or if a system does not exist, collect from the simulator. Therefore, in all cases, analysts are required to have statistical expertise. Depending on the techniques used, these skills are usually not common to the parallel programmers [116]. Furthermore, of our study on the output-buffered architecture, we are using the second method to derive those transition probabilities. Although the prediction results are within acceptable accuracy, we believe that the information revealed by the empirical equation (4.10) for output-buffered is not as expressive as that by the empirical equation (4.3) for input-buffered. For example, one can directly estimate the effect of varying the W_{FC} parameter from equation (4.3), while equation (4.10) only shows part of the picture as we cannot take hold of the relationship between W_{FC} and those transition probabilities.

The primary objective of this study is to explore the relationship between buffering architecture and congestion behavior, and through the analysis, we could devise better strategies in handling the contention problem. Although our studies are based on the many-to-one pattern with a single switched network, we believe that our findings can be extended to capture the congestion behavior of different network configurations and communication scenarios. In short summary, here are what we have observed from our analytical studies and previous experiments:

1. The number of attributed sources (P) on the contention problem has a negative impact on the throughput efficiency with our GBN reliable protocol, except on

the input-buffered case. However, it has the least weight on the performance aspect when compare to other performance parameters.

2. The larger flow control window size (W_{FC}) we set, the more susceptibly we are when facing with the congestion loss problem for all cases. Therefore, the best tactic is to avoid the loss completely. Since different buffering architectures have different overflow conditions, i.e. $W_{FC} > B_L$ for the input buffering and $W_{FC} * P > B_L$ for the output buffering, we should observe these rules in selecting the optimal window size. However, if we are too conservative in setting the window size, we lose the benefit of having pipelining. Besides, even with a small window setting, we still experience performance problem on a large cluster if the network traffic becomes asymmetric.

3. Under bulk data transfer, the throughput efficiency improves logarithmically with the increase in timeout value with our GBN reliable protocol, except on the input-buffered case. Therefore, we find that the setting of the timeout value has a significant weight on the resulting throughput. From our experimental results, we observe that it is good enough to set the timeout values to the range between $B_L * \exp < \overline{TO} \leq B_L * \exp^2$ on the output-buffered case, since the throughput efficiency only scales up logarithmically. While with the input-buffered case, the range between $2 * B_L * P \leq \overline{TO} \leq 3 * B_L * P$ should work petty good on our GBN scheme.

To further our understanding on the congestion behavior, we extend this study to another type of network configuration - Hierarchical network[9] [35]. The hierarchical network makes use of faster technology as the backbone network to support full-connectivity between many smaller subnetworks, while these subnetworks can be composed of a single switched network or another hierarchical network. Figure 4.14 gives an example of a two-level hierarchical network that composes of Fast Ethernet (FE) and Gigabit Ethernet (GE) switches.

For the first set of tests on the hierarchical network, we show that the congestion behavior of the output-buffered case can be used to explain on the congestion behavior of an input-buffered port, which happens to be the bridging port between the two Ethernet technologies. Using the same cluster as in previous experiments, we connect 10 cluster nodes to one IBM 8275-326 switch and there are total three such subnets in this setup. Each IBM 8275-326 switch is connected to a Gigabit Ethernet switch - the Alcatel PowerRail 2200 (PR2200), through a Gigabit uplink port. With this configuration, in theory, we have full-connectivity for all 30 machines. After running our benchmark tests, we find that this Gigabit uplink port has an input-buffered architecture with $B_L = 45$ units (which matches with buffer size of other FE ports). However, we also uncover a serious problem of this uplink port. Although it is capable to sustain ten full FE streams on the upstream flow[10], it could only manage to sustain at most

[9] A formal definition of the Hierarchical network will appear on Chapter 6

[10] Upstream - movement of data from the low-level to upper-level of the hierarchy; while downstream is just the reverse.

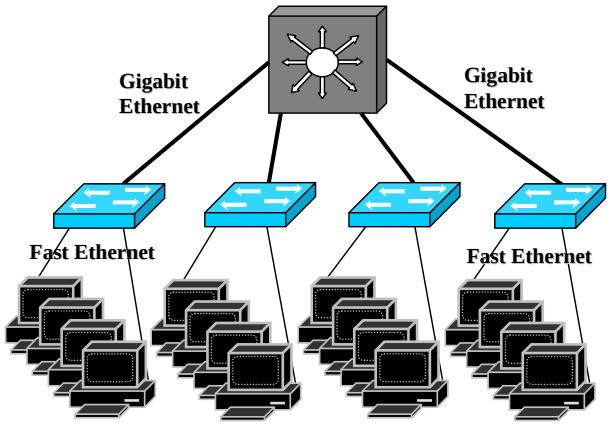


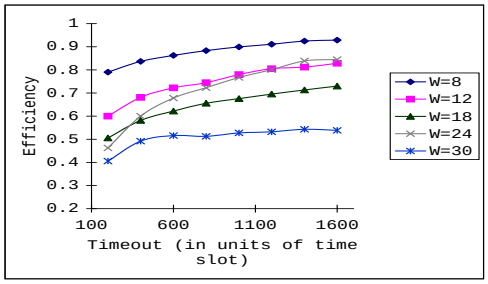
Figure 4.14: A hierarchical network composes of Gigabit Ethernet and Fast Ethernet switches

seven full FE streams on the downstream flow without packet loss. More discussion will be given later in this subsection. As for the PR2200 GE switch, our benchmark tests show that it has a shared-buffered architecture with $B_L = 820$ units.

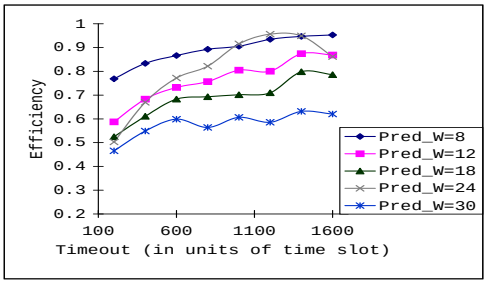
No. of senders P	Subnet A(10 nodes)	Subnet B(10 nodes)	Subnet C(10 nodes)
7	4 senders	3 senders	target receiver
11	6 senders	5 senders	target receiver
15	8 senders	7 senders	target receiver
19	10 senders	9 senders	target receiver

Table 4.3: The setting used in emulating the many-to-one flow over a congested uplink port

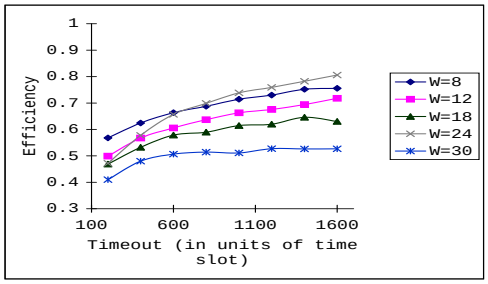
To emulate the many-to-one congestion loss problem on the uplink port, we use the following experimental settings, which are summarized in Table 4.3. With such settings, we remove the upstream performance limitation, and by directing all traffics to the same uplink port, we create a scenario similar to the output-buffered congestion problem on an input-buffered uplink port. Therefore, we can apply our previous technique to analyze the congestion behavior induces by this traffic condition, and the results are presented in Figure 4.15. Again, we observe a similar congestion dynamic when compare with the previous experiments. The only exception we have observed is the extraordinary performance improvement with the timeout parameter when we have $W_{FC} = 24$ for different TO settings. We could not identify any clue except that from the activity traces on the transition probabilities (a, c and f), with $W_{FC} = 24$ and increase in timeout value, a sender is less likely to encounter the Nack event, but once it transits to the ReSend state, it is more prone to be kept in the Stall state, i.e. c



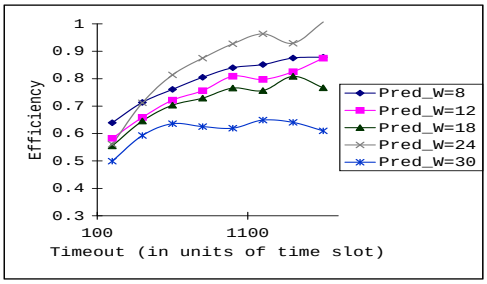
(a) Measured - P=7



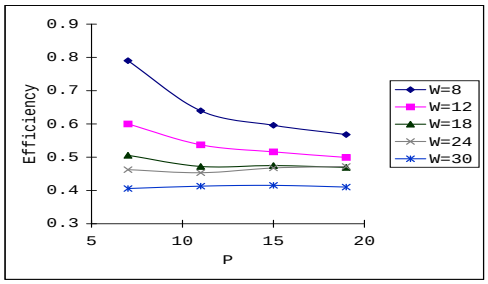
(b) Predicted - P=7



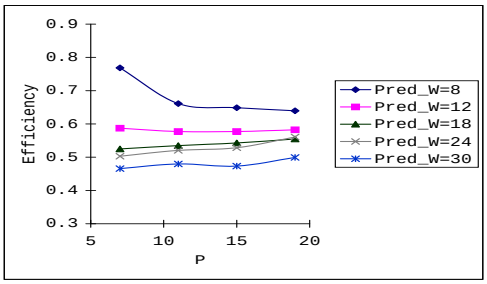
(c) Measured - P=19



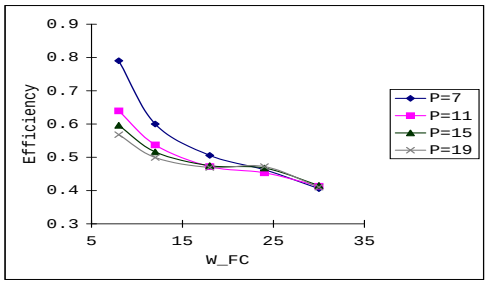
(d) Predicted - P=19



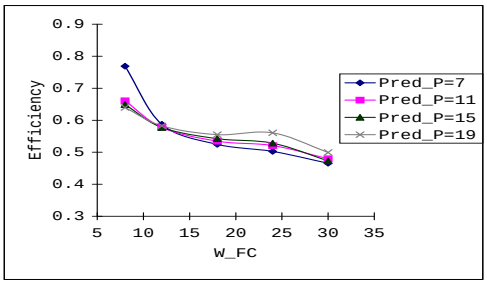
(e) Measure - Timeout=200



(f) Predicted - Timeout=200



(g) Measured - Timeout=200



(h) Predicted - P=200

Figure 4.15: The measured and predicted results of the many-to-one congestion loss problem on the uplink port under our GBN reliable transmission protocol

and *a* become smaller when increase in timeout setting. Besides, we observe that the performance behavior becomes governed by window flow control (W_{FC}) setting rather than on the number of attributed sources (P).

In the previous experiments, we could not show that having a large B_L value would be benefit under the congestion problem. To look for supporting evident, we compare the measured performance of the three setups under the same parameter settings, IBM 8275-416, Cisco 2980 and the IBM 8275-326 uplink. The results (as presented in Table 4.4) show that having a larger buffer capacity does provide better performance when subjects to the same traffic loading.

Switch	B_L	P	W_{FC}	$\overline{TO}$	Measured	Predicted
uplink	45	15	12	400	0.6068	0.6691
IBM416	95	15	12	400	0.719	0.7256
Cisco2980	128	15	12	400	0.7797	0.7427
uplink	45	15	12	1600	0.7309	0.8666
IBM416	95	15	12	1600	0.8745	0.8957
Cisco2980	128	15	12	1600	0.8891	0.8782
uplink	45	11	30	200	0.4129	0.4798
IBM416	95	11	30	200	0.6263	0.5931
uplink	45	11	30	1600	0.5564	0.6430
IBM416	95	11	30	1600	0.7967	0.8759

Table 4.4: Comparisons of the measured performance on IBM416, Cisco2980 and the IBM uplink port under the same parameter settings.

Although the above experimental setup looks rather artificial, it shows that the input-buffered uplink port behaves like an output-buffered port when it is subjected to heavy congestive flow across the hierarchy. To further examine on the congestion behavior of the uplink port under realistic traffics, we re-configure the network setup of the cluster to emulate multiple concurrent data flows across the hierarchical network.

No. of Senders	Subnet A(8 nodes)	Subnet B(4 nodes)	Subnet C(12 nodes)
$P \leq 8$	P senders	not used	P receivers
$12 \geq P > 8$	8 senders	$P - 8$ senders	P receivers

Table 4.5: The network configuration used to emulating the multiple one-way data transfer over a congested uplink port

Two sets of tests are carried out. With network configuration shown in Table 4.5, we create multiple one-to-one one-way data flows across the target uplink port; while with network configuration shown in Table 4.6, we create multiple one-to-one

No. of Senders	Subnet A(12 nodes)	Subnet B(12 nodes)
$8 \leq P \leq 24$	$\frac{P}{2}$ as both senders and receivers	$\frac{P}{2}$as both senders and receivers

Table 4.6: The network configuration used to emulating the multiple bi-directional data flow over a congested uplink port

bi-directional data flows across the uplink ports. The main reason why we use different network settings is to keep our focus on the congestion loss problem at the target uplink port(s) only, and try to avoid other sources of loss. To mimic the bulk transfer data flow, all tests are conducted with each sender sends out 30000 full size packet to its partner.

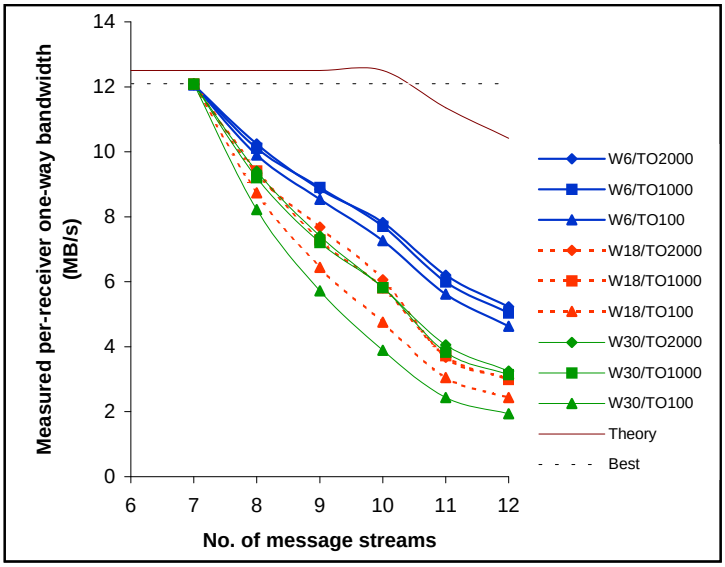


Figure 4.16: The congestion dynamic of the uplink port under multiple one-way bulk transfers

Figures 4.16 and 4.17 show the per-receiver bandwidths measured on the one-way and bi-directional tests with different window flow control and timeout settings. In general, the performance behaviors under such data flows agree with our conclusion made on the studies of the many-to-one congestion loss problem. Particularly, the test results match with our analyses that the flow control window size has the heaviest weight on the final performance, and the timeout setting comes next, while the number of attributed sources has the least weight. Furthermore, Figure 4.16 uncovers the intrinsic limitation of the uplink port, as the measurements show that it can only sustain at most 7 full FE message streams regardless of the window size setting. Once we move beyond its throughput limitation, the flow control setting becomes significant, and this indicates that the uplink port is now working under overload condition. Of the worst, Figure 4.17 further demonstrates that under heavy loading with bi-directional data flows, the circuitry of the uplink port cannot support more than 8 streams, i.e.

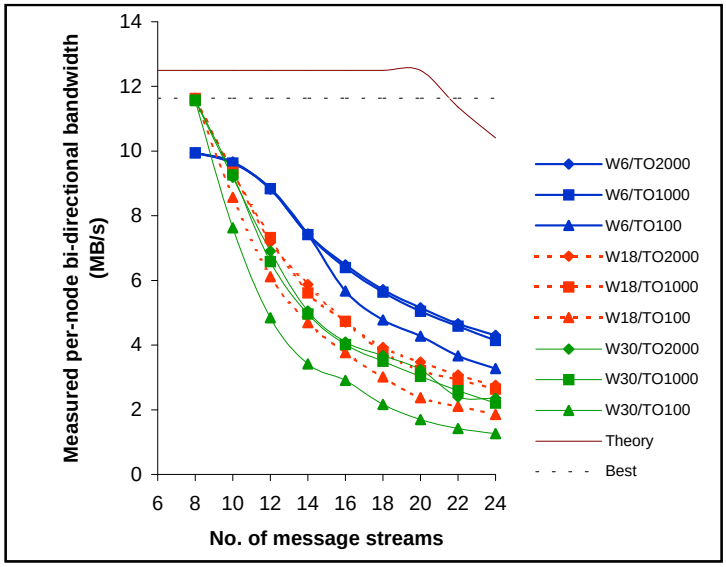


Figure 4.17: The congestion dynamic of the uplink port under multiple bi-directional data transfers

only supports up to 4 concurrent duplex streams. Under such condition, both the data and control packets are subjected to lose as traffics on both directions are heavily congested, however, we still observe a similar congestion dynamic as identified in previous experiments. This fortifies our belief that the studies on the many-to-one congestion loss problem do capture those salient features of our GBN reliable transmission protocol.

4.4 Related Work

As TCP is the most widely used reliable protocol in today's network, there exist numerous studies on performance modeling and analysis of this protocol, especially on the congestion behavior over wide-area networks, e.g. the Internet [56, 63, 74, 84, 122]. Although we are not adopting TCP as our lightweight reliable layer, their works provide valuable insight on building up our work. In particular, the work in [74] analyzes and models the Fast Retransmit and the Timeout features of the TCP protocol, which happens that our reliable protocol also supports similar features. However, they assume that packet losses are correlated, and if a packet is lost, all subsequent packets from the same congestion window are lost too. This loss assumption simplifies the modeling task, but cannot reflect the real situation under the drop-tail discipline.

Due to the uniqueness of our GBN reliable transmission protocol, a 3-state Markov chain model is used for the loss behavior analysis. This 3-state model can be viewed as an extension of the Gilbert's 2-state error model, which is commonly used for modeling of channel error [60, 61] or packet loss [2, 74] on the communication network. By delineating the loss process as the likelihood in staying or transiting between states,

this provides a method to reflect the temporal correlation between packet loss events. The difference between the 2-state model and our model is that the Gilbert's model characterize the error state directly, while our model characterize the error state indirectly, i.e. each state in our model only represents the occurrence of an event and it is the transition between states that delineates the loss process. Besides, our error structure is built on the interactions between the underlying buffer architecture and the reliable transmission protocol, this gives us a solid ground to reason on our results.

Moreover, there are other approaches to characterize the error or loss dynamic [112, 119]. Of particular interest, the work in [119] examines the temporal correlation between packet loss on the Internet connections by collecting real measurements. It found that within 1000ms time-scale, the losses are highly correlated, otherwise, they are independent of each other. By using the same dataset, it also evaluates three loss models of increasing complexity: the Bernoulli (random) model, the 2-state Markov chain model and the *k*-th order Markov chain model. The results show that the finer the correlation time-scale, the better accuracy we have with more complex model.

4.5 Summary

In this chapter, we analyze and model the congestion loss problem experienced on today's high-speed commodity network. Two different analytical models for input-buffered and output-buffered architectures are constructed and analyzed. Through the modeling process, we study how different buffering architectures, as well as different settings of our Go-Back-N reliable protocol, affect on the congestion dynamic. To show our contributions, we organize our results around areas on:

1. Of the modeling aspect, we show that the deterministic model provide more accessible information than the stochastic model, but requires explicit delineation of relationships. However, stochastic model is more powerful and flexible, despite the needs of statistical expertise. Through the comparison on our two performance models, we point out that the behavioral different between the two buffering architectures on the congestion loss problem lies on how they interact with the reliable protocol.

2. In the analysis arena, we show that our study on the congestion loss behavior of the output-buffered case can explain other architectural situations and communication scenarios. In addition, we find that under the many-to-one traffic, input-buffered architecture has a higher threshold on the overflow problem; however, once the overflow situation occurs, the performance suffers significantly (both measured and predicted results showed that we could only achieve less than 50% of the bandwidth under congestion loss).

3. Of the contention studies, we find that buffering architecture has a significant impact on the congestion behavior, as a result, some measures on congestion control may find to be effective under one architecture but futile in others. In

particular, the addition of the Stall state over classic GBN scheme has a positive impact on the congestion performance under the output-buffered case, but shows no effect on the input-buffered case. Besides, we find that the larger buffer capacity associated to a switch port, the better congestive loss performance we have on the output-buffered case.

4. Of the protocol design issue for cluster computing, we show that with many-to-one bulk data transfers, setting a longer timeout duration (e.g. $B_L * \exp < \overline{TO} \leq B_L * \exp^2$) help relieving the congestion problem with our GBN reliable protocol. Besides, we should take great care on the selection of the flow control window size, as its setting pays a critical role on the whole performance spectrum, i.e. over all traffic conditions. In particular, one should make use of the available information on the buffering architecture and capacity, the communication pattern and the number of participating nodes to derive the corresponding settings. This shows that we need to have a global prospective, rather than a simple end-to-end view when designing communication protocols.

The above results are valuable information for us to devise effective strategies to handle the congestion loss problem. However, the best answer to the congestion problem is we should try to prevent any packet loss. This is because, no matter how efficient the congestion recovery protocol is, the performance still suffers.

Chapter 5

Complete Exchange on Non-Blocking Network

In previous chapter, we show that congestion loss is a serious threat to the communication performance. A good rule of thumb in achieving high-performance on communication is to avoid congestion build up. Hence, we should avoid contention in the first place. We have stressed in early chapters that contention is a phenomenon which exists everywhere. Contention can happen in host node, network link and within the switch. Node contention happens when multiple data packets contend for the receive channel of a node, while link contention occurs when two or more packets share a communication link. And switch contention is induced by the unbalance of traffic flow through the switch, which results in overflow of the switch buffer. In fact, it is impractical to think that we can eliminate contention completely, unless we sacrifice our performance objective.

In this chapter, we carry on with our studies of the contention issue as well as the performance issue by focusing on the most demanding communication pattern on all message-passing machines - the Complete Exchange operation. We make use of our communication model to design and analyze several complete exchange algorithms, and accurately demonstrate their relative performance. These algorithms feature their own communication schedules to avoid node, link and switch contention on a *non-blocking* network. The network is said to be non-blocking if all disjoint point-to-point connections are compatible, such that there exists a disjoint path between each node-pair which are interconnected by this network. Besides the contention issue, our modeling works uncover other internal factors that have significant influence on the communication performance.

This chapter is organized as follows. We start the discussion by having a brief overview on the complete exchange operation together with some related work. Then, follow by some discussions on the network issue in Section 5.2. In Section 5.3, we present and analyze various communication schedules for the complete exchange operation with respect to our communication model. Section 5.4 contains our experimental validations and analyses of these algorithms on a real cluster platform. Finally, summaries are presented in Section 5.5.

5.1 Complete Exchange

Complete exchange, also known as *all-to-all personalized communication*, is a collective operation takes place with a set of processes, and each process has a distinct set of data to transmit to every other process in the set. To minimize the communication delay, all processes are actively participating in the communication. It is known to be the most stringent communication requirement imposed on the interconnection network. Such a communication pattern occurs in numerous numerical and scientific applications.

Due to its importance, complete exchange operation has been extensively studied in the past. Most of the studies are focused on designing communication schedules to avoid contention delay induced by the topological constraints of the underlying networks, such as hypercubes [10], meshes [94], tori [104], fat-trees [81], multistage interconnection networks [121] and multi-dimension networks [33]. These algorithms exploit the full performance of the underlying networks by carefully scheduling communications to avoid both node and link contention. Thus, these communication schedules are almost shaped to the target network constraints.

General speaking, algorithms for complete exchange can be classified into two categories, the direct or indirect approaches. For the direct algorithm, each process directly sends those data blocks to each of the destination processes using separate communication steps. A clear advantage is that the messages are delivered right to the destinations without going through any intermediate nodes, hence, each message appears only once in the network. This favors networks with higher connectivity such as multistage interconnection networks and the crossbar networks, since the major issue is to schedule the transmissions such that no link contention takes place. For the indirect algorithm, data blocks for a set of destination processes are combined to a larger block and are sent to a representative process, this is then forward to the correct destination processes. The indirect approach reduces the number of communication steps to reduce the startup cost; however, it introduces more traffic in the network and extra software overheads in performing data permutation. Thus, indirect approach favors small size messages exchange, while direct approach favors long messages exchange [16, 17, 19].

To avoid link and node contention, some complete exchange algorithms split the communication schedule into multiple phases. Each phase corresponds to a contention-free routing of messages between nodes. This approach restrains the parallelism between different phases, as a process would not enter next phase unless it has finished the message exchange of the current phase. Besides, not all processes are active in each phase [57, 104], therefore, inactive processes have to be kept idle for the whole phase. Furthermore, to achieve this contention-free synchronism, the schedule would induce substantial synchronization overhead. First, processes ahead of schedule cannot continue, this means some of the links carry no data. Second, it is hard to enforce this synchronism on a distributed system. In particular, synchronization achieved by software solutions (e.g., barrier) could contend with normal data transfer and this may become a waste of bandwidth. Bokhari et al. [18] have pointed out that by using a relaxed synchronization scheme that possibly increases the network

contention, the overall performance of the complete exchange communication could be improved.

5.2 Non-Blocking Switch

Demanding applications such as multimedia and data-intensive applications, require higher network transfer rates. Traditional bus-based LANs are not capable to serve for the needs. Therefore, the use of switches in LANs becomes an effective way to increase the network bandwidth. Besides of the improvement in network performance, these switches provide greater flexibility and interconnect scalability in network design. Driven by the market demands and trends, industrial sectors have invested considerable efforts in improving the quality of their commercial products. Particularly, some commercial network products even support non-blocking switching capability up to hundreds or even thousand ports [71]. A switch is said to be non-blocking if the switching fabric is capable of handling the theoretical total of all ports, such that any routing request to any free output port can be established successfully without interfering other traffics.

Theoretically, connecting all cluster nodes via a single non-blocking switch provides the best performance. However, to achieve good communication performance, balance of traffic flows as well as scheduling of communications should be stressed, as any misjudgment may result in congestion loss. Although having a non-blocking switching fabric guarantees high-performance, there are other internal factors that hinder the switch performance. In particular, the buffering mechanism used within the switches is one of the crucial factors. There are many variations of switch's buffering architecture, most commodity switches fall into one or a combination of these three basic types: input-buffered, output-buffered and shared-buffered. In Chapter 4, we have investigated and reported on how the switch's buffering architecture affects the congestion behavior under heavy congestive loss.

A known phenomenon that comes with the input-buffered switch is the *Head-Of-Line* (HOL) blocking problem. Packets block at the head of the queue also block the packets behind them, even if some of these packets are destined for idle output ports. By using queuing analysis, HOL blocking is shown to reduce available throughput to 58% even under uniform traffic pattern. However, input-port buffering is the simplest to design as the internal speed of the buffer only operates at the same speed as the input/output links. Therefore, they are cheap albeit have some physical constraint. While for the other two architectures, output-buffered and shared-buffered, they do not suffer from the HOL problem and thus could support higher throughput than input buffered switch on some traffic patterns. Due to technological constraints, the performance of the buffers must be fast enough to sustain simultaneous access [110], and this requires more complex and stringent design. Thus, these switches are usually more expensive than the input-buffered switch.

5.3 Complete Exchange Algorithms

We first derived the lower bound cost for the complete exchange operation on our abstract model - the completely connected cluster. Assuming that each node is capable to send and receive a message in one time unit, such that $(O_s+O_r+U_r) < max(g_s,\ g_r) < L$. With this capability, a process can actively send and receive at the same time, thus can fully utilize the communication network. In this study, we are only focusing on the communication performance in exchanging large data block, which is the general scenario happened in most scientific and numerical computations on message passing machines. To simplify the analysis, we assume that each data block corresponds to *k* data packets. Therefore, the minimum amount of packets being sent and received in the complete exchange operation per process is $2k(p-1)$ packets or $2kb(p-1)$ bytes if each data packet is of size *b* bytes.

As the minimal time in sending or receiving a packet of size b bytes is bounded by the send gap (g_s) and receive gap (g_r), and each machine can inject or receive no more than one packet within this gap. Therefore, we deduce that the minimal time required for the complete exchange operation under such a cluster communication abstraction is

$$\begin{aligned} T_{ata} &= O_s + max((k(p-1)-1)g_s, (k(p-1)-1)g_r) + L + O_r + U_r \\ &= (k(p-1)-1)\, max(g_s,\ g_r) + O_s + L + O_r + U_r \\ &= k(p-1)g + T_w \qquad (5.1) \\ &\qquad \text{where } T_w = O_s + L - g + O_r + U_r \end{aligned}$$

Thus, any solution to the *k*-item complete exchange operation on the cluster is optimal if it takes T_{ata} time units to finish the operation. Carry on with the deduction, we see that the necessary conditions to satisfy the above optimality are:

1. Each data packet must be sent directly to the target node without detour.
2. Each cluster node is actively sending and receiving the data packets without network stalling during the whole course of operation.

Condition 1 ensures that all messages appear once in the network, and therefore, minimizes the total transmission delay and messaging overhead. While with Condition 2, we ensure that no bubble exists in the network pipelines. As our performance goal is to minimize the communication time, existence of bubbles in the network pipeline means that we have to take longer time to complete the transmission. Thus, any schedule that ensures no bubble appears in the network pipelines achieves better performance.

With the assumption of complete-connected topology, links and bandwidth are sufficient but contention still exists if message transmissions are not well scheduled. In particular, any efficient algorithm in realizing the complete exchange operation must balance between synchronization and contention. This is because, due to the distributed nature of the clusters, it is difficult to impose a lock-step schedule. As most

of the synchronization operations are implemented by software means, this further impedes on normal data communication and contends for network resources. In the following subsections, we review the communication schedules used by different algorithms for the complete exchange operation, and use our performance model to evaluate their performance.

5.3.1 Shift Exchange

This algorithm is the simplest way to schedule communications without node contention. It takes *p*-1 communication rounds, and during each round, each process sends out *k* items to one partner, and receives *k* items from another partner, which is determined by a shift pattern.

Algorithm 1 Shift Exchange

```
for i = 1 to p-1 do
     from_partner = (myid + p - i) % p
     to_partner = (myid + i) % p
     for (s = 1 to k) & (r = 1 to k) in parallel do
          if ( send_item_to( to_partner_s , to) = SUCCESS)
          then
               inc s
          fi
          if ( recv_item_from( from_partner_r ,from) = SUCCESS)
          then
               inc r
          fi
     endfor
endfor
```

As depicted in Algorithm 1, during each round, each node uniquely maps to one *sendto* and *recvfrom* partners, thus exhibits no node contention. However, the non-stalling condition (Condition 2) is not enforced under this scheme. Although there is no explicit synchronization appeared between consecutive rounds and both send and receive operations are of non-blocking semantics, the *p*-1 rounds have an implicit synchronization cost that introduces bubbles to the network pipelines. For example, Figure 5.1 depicts a typical communication round. Both the send and receive channels are idle until the first byte of the first packet is being injected into the network. Similarly, after receiving the last byte of the last packet, both channels are idle until the cluster node has finished handling the last packet of this round. The predicted communication cost for this complete exchange operation is

$$\begin{aligned} T_{shift} &= (p-1)(O_s + kg + L - g + O_r + U_r) \\ &= kg(p-1) + (p-1)T_w \end{aligned} \tag{5.2}$$

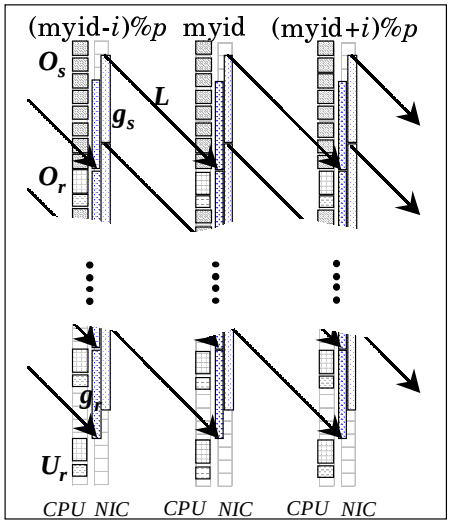


Figure 5.1: Shift Exchange pattern

From the cost formula, we notice that this algorithm is not optimal as there is a messaging overhead which is proportional to the number of cluster nodes, as denoted by $(p-1)T_w$.

5.3.2 Generalized Pairwise Exchange

In the pairwise exchange algorithm, nodes are pairing up for direct exchange in each round. Traditionally, the pairing pattern is based on the Exclusive Or (XOR) binary operation. From high-level prospective, the communication cost of the pairwise exchange algorithm coincides with that of the shift exchange as both algorithms involve the same number of communication rounds and communication load. Therefore, we can consider that $T_{pair} = T_{shift}$. We will discuss later how they may be different from an implementation view, such that their cost formulae may be different.

The major drawback of the XOR bitwise operation is the requirement of $p = 2^X$ in order to symmetrically pairing up all the nodes. For the case with $p \neq 2^X$, the number of rounds becomes $2^{\lceil log_2 p \rceil} - 1$, and during each round, not all the nodes find a matching partner. The solution to the pairing problem is to find an algorithm that matches our completely-connected topology. The pairing problem can be formulated as an edge-coloring problem on any connected graph. It is defined as:

> Given a graph $G = (V, E)$, with $|V| = p$ nodes connected by a set of edges (E), what is the minimum number of colors needed to color the edges of G so that no two adjacent edges are assigned the same color.

In graph theory, this is also defined as the edge-chromatic index $\chi'(G)$ of G. Thus, the solution to our schedule problem is to find an algorithm for edge-coloring the complete graph, with the edge-chromatic index represents the number of communication

Algorithm 2 Edgecolor Pairing Algorithm

```
PROCEDURE EdgeColor( round, myid, p )
{
     χ′ = odd(p) ? p : p-1
     if (myid < χ′) then
         v = (round + χ′ – myid) % χ′
     else
         v = odd(round) ? ⌊(round + χ′)/2⌋ % χ′ : round/2
     fi
     if (odd(p) AND v = myid) then
         return –1            // idle for this round
     else if (v = myid)
         return χ′
     else
         return v
     fi
}
```

rounds. Due to its uniqueness, there exists a simple numerable solution comparable to the XOR bitwise operation for $p \geq 3$, and is being described and proved in [40]. By incorporated this algorithm (Algorithm 2) to the pairwise exchange scheme, we have the generalized pairwise exchange algorithm. Under this mapping scheme, the performance is only slightly deteriorated with *p* communication rounds for all odd cases, instead of having *p*-1 communication rounds for all even cases.

5.3.3 Synchronous Shuffle Exchange

The above two algorithms have a messaging overhead which is depended on the number of communication rounds. If *k* is small and *p* is large, they would perform poorly. A simple solution to this problem is by removal or reduction of this messaging overhead. We observe that the previous two schedules are arranged to avoid node contention at the message level, such that during the exchange, the whole message (*k* data packets) is being sent continually to the destination.

The synchronous shuffle schedule (Algorithm 3), effectively multiplexes all the *p*-1 messages in a single round by applying a contention-free schedule at the packet level without explicit synchronization operation. Based on that packet-level scheduling, at a particular instant i^j (assume logically synchronized), each process is sending its j^{th} packet to the process p_i directly. And p_i is derived from a node contention-free permutation scheme (φ), e.g. the shift pattern, the XOR pattern or the edgecolor pattern. As each process can uniquely match to different process at each packet transmission step, it guarantees no two packets are directed to the same destination at the same instant, thus no node contention. The predicted communication cost for this complete exchange operation is

Algorithm 3 Synchronous Shuffle Exchange

```
for (s = 1 to k) & (r = 1 to k) in parallel do
    for (i_s = 1 to p-1) & (i_r = 1 to p-1) do
        to = φ_s(myid, i_s)
        from = φ_r(myid, i_r)
        status = send_item_to( to_s, to )
        if (status = SUCCESS) then
            inc i_s
        fi
        status = recv_item_from( from_r, from )
        if (status = SUCCESS) then
            inc i_r
        fi
    endfor
endfor
```

$$\begin{aligned} T_{sync} &= O_s + kg(p-1) + L - g + O_r + U_r \\ &= k(p-1)g + T_w \end{aligned} \tag{5.3}$$

From the cost formula, we notice that the messaging overhead is kept constant, and is not depended on *p* or *k*. In addition, this cost formula matches exactly to our optimal formula T_{ata}. This shows that the scheme can effectively utilize the send and receive channels by multiplexing all the messages seamlessly to a single pipeline flow without unnecessary synchronization delay.

5.3.4 Group Shuffle Exchange

If every operation is executed on schedule, and the network resources are scalable, then, the permutation scheme of the synchronous shuffle exchange could be finished in minimal time. However, in reality, logical synchronization is not enforced due to the distributed nature of the cluster system. Random delays between communication events, such as scheduling delays, could break this harmony and result in "transient hot-spot" in the switch. Observed that the more packets are targeting to the same output link, which are arriving from different sources at different time period, the higher chance of having conflicts even under a regular and uniform pattern. When two or more packets contend for the same output link, buffering of conflicting packets would result in routing delay. As the buffering technique within the switch has an enormous impact on the network performance, we reckon that the synchronous shuffle scheme could suffer on clusters with input-buffered switches due to the head-of-line blocking problem.

Group shuffle exchange (Algorithm 4) is a hybrid approach that combines the pairwise exchange and the synchronous shuffle exchange algorithms. The main idea is to

Algorithm 4 Group Shuffle Exchange

```
round = ⌈(p-1)/ω⌉                    // assume p is even
for i = 1 to round do
    group = []
    for j = 1 to ω do
        group[j] = EdgeColor((i-1)*ω+j, myid, p )
    endfor
    for (s = 1 to k) & (r = 1 to k) in parallel do
        for (j_s = 1 to ω) & (j_r = 1 to ω) do
            to = group[j_s]
            status = send_item_to( to_s, to )
            if (status = SUCCESS) then
                inc j_s
            fi
            from = group[j_r]
            status = recv_item_from( from_r, from )
            if (status = SUCCESS) then
                inc j_r
            fi
        endfor
    endfor
endfor
```

overcome the HOL problem but still achieving comparable performance as compared to the synchronous shuffle scheme. In pure pairwise exchange scheme, packets appear in each input port are destined to a unique outgoing port in each round, thus HOL blocking does not exist even under input-buffered switch. However, in the pairwise scheme, the startup overhead is linearly proportional to the number of communication rounds, which hinders its efficiency. For the group shuffle exchange, we reduce the number of communication rounds to $\lceil \frac{p-1}{\omega} \rceil$. In each round, a processor is performing a synchronous shuffle exchange with at most ω partners. The main idea of this scheme is to limit the degree of *fan-out* (ω) during individual shuffle exchange phases, while keeping the number of communication rounds to a minimum.

As this algorithm comprises of more communication rounds, the startup overhead would be higher than that of the synchronous shuffle scheme but lower than the pairwise scheme. The predicted communication cost for this algorithm is, (assume ω divides *p*-1)

$$\begin{aligned} T_{group} &= \frac{p-1}{\omega}(O_s + kg\omega + L - g + O_r + U_r) \\ &= kg(p-1) + \frac{p-1}{\omega}T_w \end{aligned} \tag{5.4}$$

Table 5.1 summaries the performance characteristics of all four complete exchange schemes that we have discussed so far.

	Synchronous	Group	Pairwise	Shift
Fan out degree (ω)	p-1	ω	1	1
No. of communication rounds	1	$\lceil \frac{p-1}{\omega} \rceil$	p-1	p-1
Pipeline stall	0	$\lceil \frac{p-1}{\omega} \rceil - 1$	p-2	p-2

Table 5.1: Performance characteristics of different Complete Exchange schemes

parameters	O_s	g_s	g_r	O_r	U_r	L for 326	L for 416
Time (μs)	12.5	122	123	20	7	0.3387p+149	0.3413p+264

Table 5.2: Model parameters for the experiment cluster

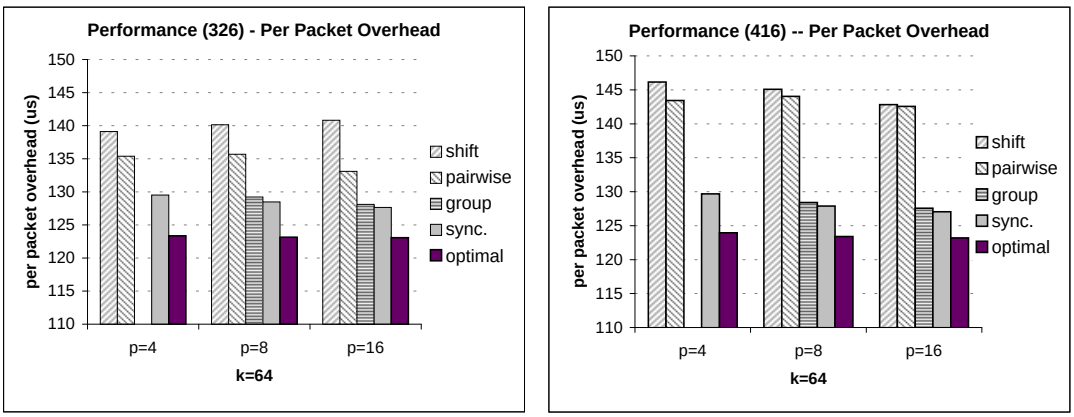
5.4 Experimental Results

Our experimental platform is a cluster consists of 16 standard PCs running Linux 2.0.36. Each node is equipped with a 450MHz Pentium III processor with 512 KB L2 cache and 128 MB of main memory. The interconnection network is the Fast Ethernet driven by the Directed Point communication system. Each node includes a DEC21140-based Ethernet card and connects to a Fast Ethernet Switch. Two IBM switches with different internal architectures are tested. One is the model 8275-326, which is a 24-port virtual cut-through switch, and is revealed by our microbenchmark as an input-buffered architecture. Another switch is the model 8275-416 which is a 16-port store-and-forward switch, and is revealed as an output-buffered architecture. We have implemented the above algorithms on this platform and compared their performances with the analytical formulae.

To evaluate the performance of these algorithms, model parameters of our experimental cluster are required. Table 5.2 shows all the necessary model parameters for this cluster, which are derived from the microbenchmark tests as described in Appendix A. As we are assuming *k*-item complete exchange, we can simplify the analysis by using a constant value for most of the parameters. This is because all experiments are conducted with a fixed size data packet, which is the maximum payload (1492 bytes) available to a Directed Point packet.

5.4.1 Complete Exchange Performance

We validate the performance of these algorithms by comparing the measured results with the optimal prediction, which is derived by using Eq. 5.1. Figure 5.2 shows the analysis on the per packet overhead of these direct algorithms for p = 4,8,16 with k = 64 on the same cluster but interconnected by the two IBM switches respectively. The per packet overhead is a metric used to measure the average latency experiences by the processor in exchanging a data item. This is calculated by dividing the measured time with the total number of data packets that each process has sent out. The measured



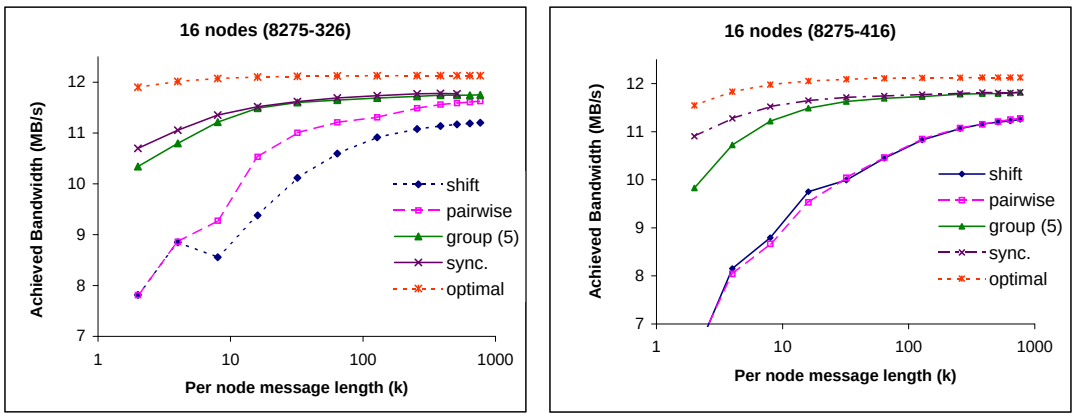
(a) Connected by the IBM 8275-326 (b) Connected by the IBM 8275-416

Figure 5.2: Performance of different complete exchange algorithms with k=64 on a 16 nodes cluster

results conform with our analytical studies, that is, the synchronous shuffle exchange algorithm is the best amongst all of these direct algorithms, and their performance rankings match with their cost formulae in the previous section. For the group shuffle exchange, we use ω=5 at p=16, i.e. there are $\lceil \frac{16-1}{5} \rceil = 3$ rounds, and ω=4 at p=8 (2 rounds). The group shuffle scheme works as the second best which is almost closed to the performance of the synchronous shuffle exchange algorithm (which only has one communication round). We do not provide any data points for group shuffle at p=4 in Figure 5.2 as we believe that it is meaningless to use group shuffle scheme when p is small.

One of the features of our GBN reliable layer is the support of piggyback acknowledgement scheme. Since the communication schedule of pairwise exchange involves only a single partner in each round, we can apply optimization techniques such as piggyback and delay acknowledgement to reduce the control traffics. Based on this optimized implementation, we see that the pairwise exchange algorithm works faster than the shift exchange algorithm in all cases, although they both have the same communication complexity. This is a typical example on the tradeoff between simplicity against accuracy on the performance analysis issue. In Section 5.3, all analyses on different communication schemes are based on the assumption of the *send-and-forget* nature [8] of asynchronous communication. However, as the underlying network is unreliable, we have to build a reliable protocol layer to accomplish reliability. This protocol layer would induce some control traffics that inevitably interferes with the normal data flows.

Figure 5.3 shows the achieved bandwidth of these algorithms for p=16 against different message sizes (k) range from 2 to 768 (data packets) on the two switches. Achieved bandwidth is a metric which measures the efficiency of the algorithm in utilizing the network. This is calculated by dividing the total data message sizes per node



(a) Connected by the IBM 8275-326 (b) Connected by the IBM 8275-416

Figure 5.3: Achievable bandwidth per node for different algorithms as compared to the optimal prediction

with the measured communication time. The results show that synchronous shuffle exchange performs the best with the achieved bandwidth for each node reaches 11.82 MB/s (on IBM 8275-416), which is 97% of the available bandwidth. While the measured best achievement for shift exchange is 11.25 MB/s on IBM 8275-416, pairwise exchange is 11.6 MB/s on IBM 8275-326 and group shuffle is 11.8 MB/s on IBM 8275-416 respectively.

For very long messages (large k), both shift exchange and pairwise exchange are catching up with the performance of the synchronous shuffle exchange algorithm. However, for exchanging small to medium size messages in shift exchange and pairwise exchange, the waiting time incurred by each communication round cannot be masked away and results in poor performance. For example, at k=4, we observed that the achieved bandwidth of the pairwise exchange is 72% of the optimal performance, while the achieved bandwidth of the synchronous shuffle exchange reaches 92%. Furthermore, it is interested to see that the optimization techniques adopted in the pairwise exchange algorithm looks more promising on the virtual cut-through switch (IBM 8275-326) then on the store-and-forward switch (IBM 8275-416). We have repeated the same experiment on IBM 8275-326 with the cut-through mode being switched off, and we experienced the same performance pattern as reported on the IBM 8275-416 switch (in Figure 5.4).

On the other hand, the synchronous shuffle and the group shuffle exchanges perform much better even up to k=100. The synchronous shuffle scheme logically schedules all communications at the packet level in a pattern that avoids node and switch contentions. As waiting time is removed, the network links are better utilized, and we can exchange all the messages by minimal time, hence, achieved better performance. Theoretically, at the same instant, all packets arrived to different input ports are destined to different output ports according to our contention-free schedule. Therefore,

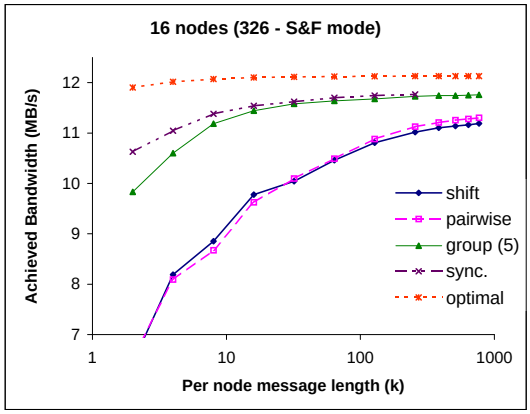


Figure 5.4: Achieved bandwidth on IBM 8275-326 with store-and-forward switching

this schedule should operate efficiently on any non-blocking network.

In reality, no global clock is implemented and the operations are not lock-step synchronized. The shuffle pattern of the synchronous exchange could induce head-of-line blocking on the input-buffered switch. The group shuffle exchange algorithm limits the degree of fan out and introduces minimal waiting time during the communication. As shown in Figure 5.3, it performs almost as good as the synchronous shuffle exchange algorithm even at small *k*.

5.4.2 Effects on Group Size ω

In Figure 5.5 we compare the efficiency of the group shuffle exchange algorithms with respect to different group size ω for *p*=16 on various per node message length *k*=4, 40, 400. Technical speaking, with $\omega = 1$, the synchronous shuffle exchange reduces to the pairwise exchange, while $\omega = 15$ corresponds to the synchronous shuffle exchange. Clearly, this figure shows that synchronous shuffle exchange always performs the best in all aspects. The group shuffle exchange performs considerably well even under small ω for medium to large messages as compares to pairwise exchange. As the performance of the synchronous shuffle algorithm deteriorates under heavy traffic when p is large (will be discussed in the next subsection), the group shuffle algorithm becomes an alternative in achieving higher performance.

5.4.3 Scalability on Problem Size *k*

We compare the scalability of these algorithms when operated on the input-buffered switch (IBM 8275-326) with *p*=16 and ω=5 while varying the problem size *k*. This test reveals the effect of HOL problem while transmitting long messages. Figure 5.6 shows the results of this experiment. Clearly, we see that synchronous shuffle exchange achieves the best performance, however, the performance degraded significantly after $k > 512$ (out of the range shown in the graph), which corresponds to the transmission of total message length 11 MB per node. The performance of group shuffle exchange is

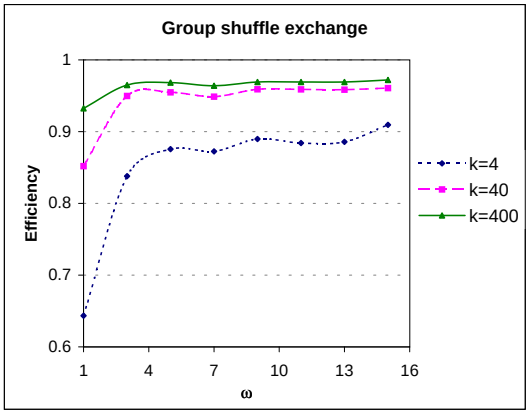


Figure 5.5: The efficiency of the group shuffle exchange algorithm as a function of group size ω for various message size k on 16 nodes over the IBM 8275-326.

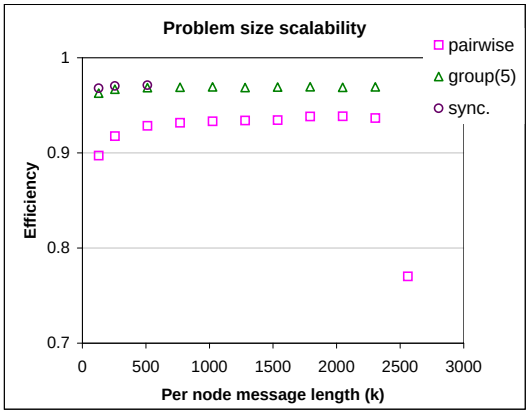


Figure 5.6: Comparison of the problem size scalability on the input-buffered switch (IBM 8275-326) for p=16

only slightly worse than the synchronous shuffle exchange, and it continues to operate at high efficiency until $k > 2304$ (around 49 MB per node). Lastly, we observe that the pairwise exchange continues to work for very large message length (around 54 MB per node), but the performance drops dramatically as the required total messaging buffer size is approaching the machine limit, which is around 128MB. In summary, the group shuffle exchange algorithm shows its robustness in dealing with the HOL problem and can retain good performance for very large message sizes.

5.4.4 Comparing Switching Mechanisms

Since the complete exchange operation induces heavy network loading that stresses on the communication network severely, it could be used as a tool to evaluate the relative performance of different hardware components. Figure 5.7 shows the achieved bandwidth of the group shuffle scheme and synchronous shuffle scheme for p=16 on our two

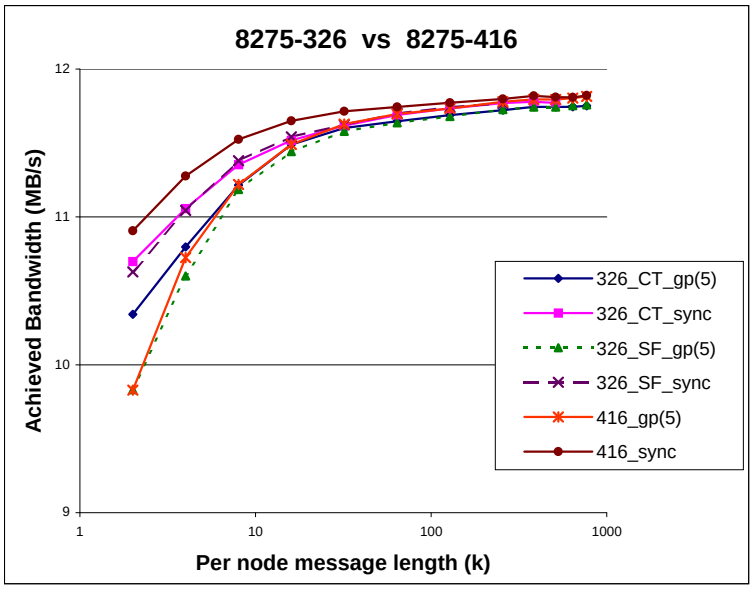


Figure 5.7: Comparison of performance between a virtual cut-through switch (IBM 8275-326) and a store-and-forward switch (IBM 8275-416) on the synchronous shuffle and group shuffle algorithms

switches. Generally, both switches have the similar performance for long messages, so not much difference between the cut-through or store-and-forward modes. However, on short message exchanges, if we cannot fully utilize the network, we cannot effectively mask away the higher latency of store-and-forward switching. This is being reflected by the slower performance of the group shuffle scheme on both switches when they are working in store-and-forward mode. Furthermore, we observe that the switch internal buffering mechanism does affect the overall performance. The performance of the synchronous shuffle exchange algorithm on the output-buffered switch (IBM 8275-416) is always better than the same algorithm on the input buffered switch (IBM 8275-326) with both packet forwarding modes.

5.4.5 Comparison with MPICH

In Chapter 4, we observe that under heavy congestion loss, our reliable protocol may not be working efficiently due to its simplicity. For examples, we are using static window size and retransmission timer, while the sophisticated TCP/IP protocol adopts many features [93] in handling the congestion issue. Since congestion loss occurs mainly at high network load, one would think of switching back to the traditional protocol stack while we are having intensive communication scheme, such as the complete exchange operation.

Our objective of devising efficient communication schemes atop of lightweight messaging system is to support high-level programming model such as MPI. Therefore, we have a direct comparison of our DP implementation of synchronous shuffle

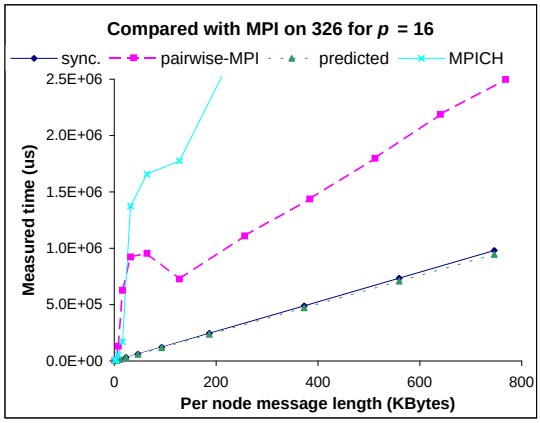


Figure 5.8: Comparing the performance of the synchronous shuffle complete exchange algorithm with two MPICH implementations

exchange algorithm against the MPICH [68] implementation of the complete exchange operations. The results are shown in Figure 5.8. There are two complete exchange MPICH implementations in this graph. The curve labeled as "MPICH" represents the original implementation found in the MPICH package, while the curve labeled as "pairwise-MPI" represents our implementation of the generalized pairwise exchange algorithm with the *MPI_sendrecv()* communication primitive.

First, we clearly see that the original implementation of the *MPI_Alltoall()* function by MPICH performs extremely inefficient. This is because their implementation is based on simple non-blocking *MPI_Isend()* and *MPI_Irecv()* functions, which are issued in an uncoordinated manner. Therefore, it would subject to both node and switch contention as well as the high overhead problem that inherits from the TCP protocol stack. To avoid the node and switch contention, the "pairwise-MPI" carefully coordinates the communications, and achieves considerable improvement. However, our DP implementation of synchronous shuffle exchange even outperforms the "pairwise-MPI" implementation significantly. This shows that the traditional protocol stack has severe limitation on achieving high-speed communication. In other words, it is inadvisable to drive the high-performance communication network with the conventional communication protocols.

5.5 Summary

In this chapter, we focus on the practical issues of designing high-speed complete exchange algorithms on a commodity cluster interconnected by a non-blocking network. As the performance of interconnection networks is the limiting factor of most existing clusters, improving the communication performance by well scheduling of communication events could leverage the overall performance of the clusters. Four complete exchange algorithms, including, shift exchange, pairwise exchange, group shuffle exchange and synchronous shuffle exchange algorithms are studied and implemented on

our experimental platform. Their common characteristic is that they all arrange the communications or message exchanges in a node and link contention-free manner.

Based on the architecture and communication model of cluster, we observe that to achieve optimal result, the network pipes must be fully utilized. Any waiting stage would stall the communication pipelines and decrease the overall communication performance. This optimum could only be fulfilled by the synchronous shuffle exchange algorithm, which adopts a contention-free schedule at the packet level. Due to its effectiveness, our experimental results show that the performance of synchronous shuffle exchange reaches 97% of the available bandwidth. In particular, we show that by careful scheduling of all communication events to balance the usage of available network resources, this hides away the synchronization overheads, and gain considerably improvement even when exchanging small size messages. While both shift and pairwise exchanges show their effectiveness on exchanging long messages, they become inefficient when exchanging small messages as they cannot fill up those network pipelines effectively.

Theoretically, under a contention-free schedule over a non-blocking network, all packets arrived to different input ports are destined to different output ports; therefore, can be routed instantaneously. However, in reality, no global clock is implemented to coordinate all cluster nodes and their events. Thus, their operations are not lock-step synchronized. Any variations in communication schedules will result in drifting from the theoretical harmony. The consequence is packets start to cumulate in the network due to experience of conflicts. Under this situation, the buffering architecture of the switch plays a critical role in the performance issue. In our experiments, we show that the performance of the synchronous shuffle exchange algorithm is affected by the Head-Of-Line blocking problem. To counteract the HOL blocking, we have devised the group shuffle exchange algorithm. Lastly, we show that the group shuffle exchange performs almost as good as the synchronous shuffle exchange algorithm and scales better when it works on an input-buffered switch.

Chapter 6

Complete Exchange on Hierarchical Network

In this chapter, we propose an efficient communication schedule for running the complete exchange operation on clusters, which are interconnected by the Ethernet-based hierarchical network. The essential feature of this communication scheme is the *proactive* approach in handling congestion. We consider the contention effect at three different stages. At the first stage, we experience no contention and the performance of the network is bounded by its hardware performance only. For example, the network load is light and there are no conflicting traffic. At the second stage, we have mild contention with slight decrease in performance due to the experience of contention delay. At the third stage, we experience significant performance loss as the severity of the contention has induced the congestion loss problem. Therefore, to achieve good performance, we try to avoid contention in the first place by having a communication schedule, which prevents contention at the node and switch. If congestion does develop, our communication scheme regulates the traffic to avoid further building up of congestion that results in congestion loss.

Our complete exchange algorithm on hierarchical network is based on the synchronous shuffle exchange algorithm, which is developed on a theoretical non-blocking network. By introducing a *global windowing* concept to all participating nodes, they are responsible to monitor and regulate the traffic loads to avoid congestion. Based on architectural features like the network buffering capacity and the balancing of upstream and downstream flows, we derive the global windowing scheme and the contention-aware permutation, which transform the algorithm to work efficiently on the hierarchical network.

This chapter is organized as follows. Section 6.1 lays down the architectural characteristics of the Ethernet-based hierarchical network, and describes a simple abstract model of the hierarchical network to aid our analysis. Section 6.2 describes how to augment the original synchronous shuffle exchange algorithm to run efficiently on both Ethernet-based hierarchical networks and non-blocking networks. Then, the experimental results of the modified algorithm are presented in Section 6.3. Finally, summaries are presented in Section 6.5.

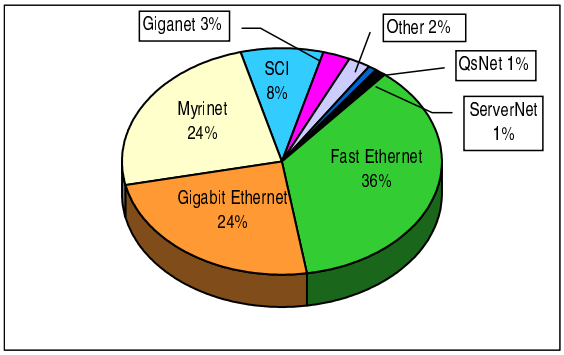


Figure 6.1: Poll results on the subject: "*What interconnect do you use or would use in your cluster?*" (Source: Clusters@TOP500 [28])

6.1 Hierarchical Network

Ethernet-based network is the most widely used local area networking technology. Although standard Ethernet has limited bandwidth in supporting cluster computing, its enhanced versions, such as Fast Ethernet (FE), Gigabit Ethernet (GE) and 10 Gigabit Ethernet provide sufficient bandwidth with a steady upgrade path in building large-scale commodity clusters. Therefore, many self-made large-scale clusters [25–27] are using Fast Ethernet and/or its successor as the base of their interconnections. For example, Cluster@TOP500 [28] has conducted an informal survey on choice of cluster interconnects. There are total 380 votes as of July 2001 on this poll, and the result is shown in Figure 6.1. Roughly speaking, 60% of the system designers have voted for Ethernet-based interconnect.

Currently, there are two approaches in building a large-scale cluster using the Ethernet-based interconnections. First, using a single high-performance, high port density chassis switch to connect all machines [71]. Second, using a hierarchical network, in which cluster nodes are connected to Fast Ethernet switches and using the Gigabit Ethernet as a backbone network to interconnect all Fast Ethernet switches. Given that the backbone capacity of the interconnection network is greater than the bandwidth demands of the whole cluster, both approaches support a fully connected network with similar performance. In reality, both approaches are suffered from similar architectural constraints that limit their actual performance. In previous chapter, our study of complete exchange on a single router switch has revealed that the buffering mechanism used within the switch could hinder its actual performance, and we have proposed a sub-optimal algorithm (group shuffle exchange) in dealing with the situation.

The hierarchical approach makes use of compatible network technologies, which is a more cost-effective method in incremental scaling of the cluster. With current Ethernet installations, connections between different technologies are commonly bridged by one or more uplink ports, which are add-ons to those “lower” end devices. Since faster Ethernet technologies are mutated from the standard Ethernet, so they are using the same mechanism in switching packets. Packet received on an ingress port is

switched to the corresponding egress port according to the destination MAC address found at the head of each packet. The switch uses its address lookup table to make this forwarding decision.

Within the Fast Ethernet switch, there are two types of network traffics, either internal traffics or cross-switch traffics. For internal flows, packets are destined to cluster nodes that connect to the same FE switch directly; while for the cross-switch flows, packets are destined to remote cluster nodes that are resided across the network hierarchy. The requirement of having higher channel bandwidth for the uplink limits the switching mechanism adopted on this type of interconnection. As cut-through switching is only possible for ports operate at the same data rate or slower, this is not suitable for the uplink connections and makes the store-and-forward switching be the only feasible solution. However, the change of channel bandwidth between two technologies may induce hot spot since store-and-forward switching causes cumulating of upstream and downstream packets over those uplink ports.

Apparently, even under a node contention-free schedule, sharing of uplinks is needed on a hierarchical network. For instance, all cross-switch traffics are going via the uplinks to the GE switch, packets have to contend for the shared links although they are from distinct sources and to distinct destinations. In theory, under a node contention-free schedule, the distribution of data packets should be well balanced. Thus, any transient congestion over the uplinks could be handled by the buffers in the switches as well as the higher throughput of the uplink connections. However, the distributed nature of the clusters does not guarantee to adhere to a tightly synchronized schedule. Any random delay on scheduling communication events of the complete exchange operation may result in considerably contention. As congestion is handled by the buffers in the switches, therefore, we see that the buffering mechanism plays a critical role in the overall performance of the hierarchical network.

6.1.1 System Model

To simplify the discussion, a two-level switch hierarchy as depicted in Figure 6.2 is discussed. For this tree topology, all cluster nodes are the leaf nodes, and are grouped into disjoint sets with d_1 members in each set. Members of the same set are connected to a parent which is a switch node located at Level 1, and all communications generated by the set - both within set and across set, have to go through this switch node. Communications between sets are established through the root switch node, which fully connects all Level 1 switch nodes. To support high performance communication, we assume that the switch-to-switch link bandwidth c_2 and the node-to-switch link bandwidth c_1 satisfy this constraint, $d_1c_1 \leq c_2$, which ensures that the throughput capacity of the uplink is able to handle all upstream/downstream traffics generated by the whole set at any particular instant. We also assume that the backbone bandwidth of those Level 1 switches are greater than or equal to $d_1c_1 + c_2$, and the backbone bandwidth of the root switch is greater than or equal to d_2c_2. With these assumptions, the aggregated bandwidth available to a cluster with p nodes (where $p = d_1d_2$) is bounded by $d_1d_2c_1$.

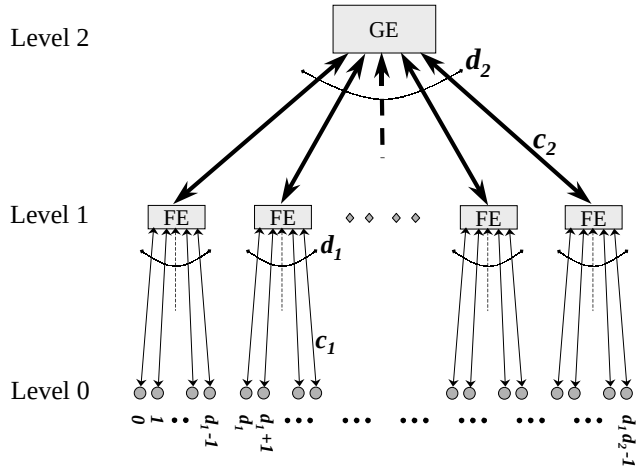


Figure 6.2: Interconnection topology of the two-level switch hierarchy

6.2 Modified Synchronous Shuffle Exchange

In Chapter 5, we have presented the Synchronous Shuffle complete exchange algorithm, which is a bandwidth-optimal algorithm on any non-blocking network. The spirit of this algorithm is the node contention-free schedule operated at the packet level without explicit synchronization operation. By effectively utilizing the send and receive channels, this scheme multiplexes all the messages seamlessly to a single pipeline flow by scheduling consecutive packets to different destination nodes according to a node contention-free permutation (φ).

If every operation is executed on schedule, the permutation scheme of the synchronous shuffle exchange can be finished in minimal time. However in reality, as this schedule induces intensive communications and demands on logical synchrony, any non-deterministic delays between events could break the synchronism and result in congestion developed in the switch. For example, non-coordinated process scheduling would introduce randomness. We have shown in previous chapter that not all switches can withstand such an intensive communication pattern for an extended period.

Generally, having logical synchrony on all cluster nodes is an idealistic assumption for the case of commodity clusters, which have no hardware synchronization support. To impose this synchrony, explicit synchronization operations can be used. However, this brings on extra synchronization overhead to the total communication time, and also stalls the communication pipelines as no data communications are taking place during synchronization. Since performance loss is caused by oversubscription to the network, which induces packet loss at the bottleneck region, the best solution to avoid congestion loss is to prevent oversubscription to the network. That can be done by applying some form of traffic control on each node to minimize the contention problem.

6.2.1 Global Window Congestion Control

The conjecture behind the contention problem induced by the synchronous shuffle exchange algorithm is the non-deterministic delays on communication events. With the hierarchical network, two more sources of delay could contribute to this non-determinism.

- ❑ Queuing delay at the uplinks: with the hierarchical network, inter-switch traffics have to go through shared uplinks and conflicting packets are buffered; therefore, increases the network delay.

- ❑ The difference of network latencies between nodes: even with the use of faster technology for upper level interconnections, additional transmission delays on delivering the data across the hierarchy would induce the contention problem.

To achieve optimal performance on the hierarchical network, sharing of links is necessary. Thus, having link contention is a fact that we must confront with. Although mild contention increases network delay, it does not severely degrade the performance, unless the congestion persists for a long period of time, which results in buffer overflow. Therefore, it would be useful to have a congestion control scheme to prevent oversubscribing the network.

In this study, we adopt a proactive approach in the congestion control. This congestion control scheme is different from traditional approaches. Conventional mechanisms for controlling congestion are based on end-to-end windowing schemes [89], however, they are not suitable for collective operations in high-speed networks. This is because they are usually *reactive* schemes. They probe for congestion signals, such as packet loss and timeout signals, and respond by recovering the loss and regulating the traffic load to avoid further loss. However, we have already lost some packets, and this has affected the performance. Our GBN reliable protocol described in Chapter 4 is an example of a reactive scheme. Besides, the feedback information from the network is usually outdated due to the propagation and transmission delays. Hence, any reactive action taken may be too late to avoid further loss. Furthermore, end-to-end windowing only provides traffic information on individual connection. It lacks in a global picture of the network, such as the number of traffic sources and the communication pattern in used.

However, in cluster computing, the traffic pattern is predictable in the case of collective operations on an enclosed network. Therefore, we can utilize those available information, such as the network buffer capacities (B_L) of the switched network, the communication pattern and communication volume, to derive some resource-aware congestion control scheme. With our global congestion control scheme, each source is assigned with a predefined resource limit, and our scheme forces them to regulate their traffic loads below this limit. By having a fair share of resources, this ensures that no source will exceed its allowed traffic capacity and avoids congestion loss.

We have observed that during the execution flow, at i^{th} communication step, a process is sending a data packet to another process according to the node contention-free

permutation scheme φ. If every operation is on schedule, the number of outstanding data packets (η) in transit from a process to other process is bounded by $\left\lceil \frac{L}{g_s} \right\rceil$. Under mild congestion, the process experiences slight increase of η. If congestion persists, this eventually induces packet loss, and η will increase considerably. The above observation implies that to avoid overflowing the network buffers, we need to regulate the number of outstanding packets (η).

The principle behind our congestion control scheme is quite simple. When applying this scheme on our complete exchange algorithm, all senders are assigned with a global window (W_g) at the beginning of the communication event. This W_g factor acts as a controller to limit the amount of traffic that a particular sender can inject into the network. If a sender finds that sending out a data packet may overload the network, when $\eta = W_g$, it just halts current transmission and waits until it is safe to transmit, i.e. $\eta < W_g$. By picking the correct value for W_g, this scheme guarantees that during any interval, the total number of packets entering the network does not exceed the sum of a pre-specified limit, which is the network buffer capacity at the bottleneck region. To compute W_g, we need to identify the bottleneck region and measure the buffer capacity (B_L) associated to the bottleneck, then we derive W_g from B_L on the principle of fair sharing.

Based on the communication pattern and schedule, we estimate the average number of packets (ν) generates at each communication step which are forwarded to the bottleneck region. Without lost of generality, let's take the FE/GE hierarchical network as an example. Assume that the uplink ports are the critical bottlenecks and they are of input-buffered architecture. Under the synchronous shuffle schedule, in *p*-1 communication steps, a process generates *p*-1 data packets which are destined to *p*-1 distinct nodes. However, only $d_1 - 1$ packets are switched locally, and the rest, $p - d_1$ packets, are forwarded by the Fast Ethernet switch to its uplink port. Therefore, there are $(p-d_1)d_1$ packets being forwarded upstream by each FE switch in *p*-1 communication steps. Based on the node contention-free permutation, the same amount of data packets are switched from the Gigabit backbone back to each FE switch. Thus, the average number of data packets directed to each uplink port per communication step is

$$\nu = \frac{(p - d_1)d_1}{p - 1} \tag{6.1}$$

From this, we derive the value of W_g, which is

$$W_g = \left\lfloor \frac{B_L}{\nu} \right\rfloor \tag{6.2}$$

6.2.2 Contention-Aware Permutation

However, knowing the value of W_g is a necessary but not sufficient condition to avoid congestion loss. This is because W_g is derived from taking the average traffic load, and unlike traditional end-to-end scheme, global windowing needs to monitor and regulate all traffic flows of a process, not just one connection. If the traffic distribution is not

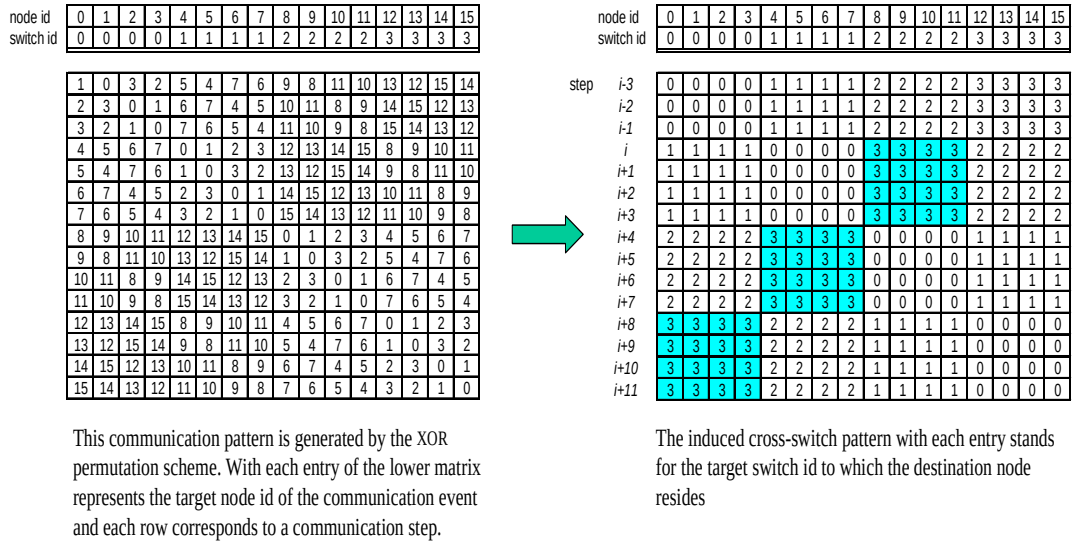


Figure 6.3: An example permutation in which global windowing alone fails to regulate the traffic.

uniformly spread across the network, the global windowing scheme could not fulfill its function correctly. This is being shown in Figure 6.3. In this example, we assume that the bottleneck region of the 4X4 two-level hierarchical network is at the uplink ports with $B_L = 30$. However, under the XOR permutation scheme, we could experience the contention loss problem even though global windowing is adopted.

In this example, the size of W_g is $\lfloor \frac{30}{3.2} \rfloor = 9$. Assume that at communication step *i*, four packets originate from switch 2 and head for switch 3 are blocked by some cause, e.g. HOL, so as those packets that follow in step *i*+1, *i*+2, and *i*+3 from the same switch. However, no process is aware of the congestion problem unless their global windows become saturated. This may only be happened until step *i*+8 when processes in switch 3 detect the congestion problem. By that time, processes in switch 1 have already sent out all their packets to processes in switch 3, which further increases the queue length at switch 3. Moreover, processes in switch 0 are not aware of the problem. This is because global windowing collects traffic information on the base of past events, but none of these past events could indicate the congestion problem in switch 3. As a result, processes in switch 0 continue to send all their packets to processes in switch 3, which finally overflow the buffer.

An obvious reason for this failure is that the feedback information on traffic condition is not regularly gathered from all part of the network. Thus, information on part of the network is outdated. Although the overflow situation could be detected and resolved by both global windowing and individual end-to-end flow control scheme, performance has been suffered as packets are lost inevitably. If we can arrange all communication events in a way that each process is communicating with different processes reside in a node linked to different switches at each communication step. Then, the traffic loads would become more evenly distributed as well as having more regular

node id	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
switch id	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

4	0	12	8	5	1	13	9	6	2	14	10	7	3	15	11
8	12	0	4	9	13	1	5	10	14	2	6	11	15	3	7
12	8	4	0	13	9	5	1	14	10	6	2	15	11	7	3
1	5	9	13	0	4	8	12	3	7	11	15	2	6	10	14
5	1	13	9	4	0	12	8	7	3	15	11	6	2	14	10
9	13	1	5	8	12	0	4	11	15	3	7	10	14	2	6
13	9	5	1	12	8	4	0	15	11	7	3	14	10	6	2
2	6	10	14	3	7	11	15	0	4	8	12	1	5	9	13
6	2	14	10	7	3	15	11	4	0	12	8	5	1	13	9
10	14	2	6	11	15	3	7	8	12	0	4	9	13	1	5
14	10	6	2	15	11	7	3	12	8	4	0	13	9	5	1
3	7	11	15	2	6	10	14	1	5	9	13	0	4	8	12
7	3	15	11	6	2	14	10	5	1	13	9	4	0	12	8
11	15	3	7	10	14	2	6	9	13	1	5	8	12	0	4
15	11	7	3	14	10	6	2	13	9	5	1	12	8	4	0

After applying the contention-aware mapping to re-map each node's logical id, a new communication pattern is generated with the same XOR permutation scheme.

node id	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
switch id	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

1	1	1	1	0	0	0	0	3	3	3	3	2	2	2	2
2	2	2	2	3	3	3	3	0	0	0	0	1	1	1	1
3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
1	1	1	1	0	0	0	0	3	3	3	3	2	2	2	2
2	2	2	2	3	3	3	3	0	0	0	0	1	1	1	1
3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
1	1	1	1	0	0	0	0	3	3	3	3	2	2	2	2
2	2	2	2	3	3	3	3	0	0	0	0	1	1	1	1
3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
1	1	1	1	0	0	0	0	3	3	3	3	2	2	2	2
2	2	2	2	3	3	3	3	0	0	0	0	1	1	1	1
3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0

The induced cross-switch pattern with each entry stands for the target switch id to which the destination node resides

Figure 6.4: The resulting communication pattern after applying the contention-aware permutation scheme.

information feedback between different processes in different part of the network.

An approach in generating this kind of dispersive pattern is by adopting a contention-aware permutation, which includes knowledge on the network constraints to generate the communication schedule. Observed that the original permutation is obtained by some simple functions (φ) which operate on inputs such as current communication step and node id. A simple method to incorporate the network structure into the original permutation is by redefining a mapping between the logical node id to its physical id. One example of such permutation scheme (ϕ) can be as follows:

$$\text{logical id} = \left\lfloor \frac{\text{physical id}}{d_1} \right\rfloor + (\text{physical id} \,\%\, d_1) * d_2 \tag{6.3}$$

Carry on with the previous example. If we apply the XOR permutation on the re-mapped logical id, we get the communication schedule as shown in Figure 6.4, which is a more evenly distributed pattern with respect to both switches and cluster nodes. We observe that with this new communication pattern, a process is communicating with different processes located in different part of the network in consecutive communication steps. This greatly relieves the contention at the uplink ports and improves the effectiveness of our congestion control scheme.

Based on the global windowing and the contention-aware permutation scheme, we have modified the synchronous shuffle exchange algorithm to work efficiently on the two-level hierarchical network, and the modified algorithm is given in Algorithm 5.

Algorithm 5 Modified Synchronous Shuffle Exchange Algorithm

```
set η = 0
for (s = 1 to k) & (r = 1 to k) in parallel do
    for (i_s = 1 to p-1) & (i_r = 1 to p-1) do
        to = φ_s( φ(physical id), i_s)
        from = φ_r( φ(physical id), i_r)
        if (η < W_g) then
             status = send_item_to( to_s, to )
             if (status = SUCCESS) then
                 inc i_s
                 inc η
             fi
        status = recv_item_from( from_r, from )
        if (status = SUCCESS) then
            inc i_r
            dec η
        fi
    endfor
endfor
```

6.3 Experimental Analyses

Our experimental platform is a cluster consists of 32 standard PCs running Linux 2.2.14. Each cluster node equips with a 733MHz Pentium III processor with 256KB L2 cache, 128MB of main memory, an integrated 3Com 905C FE controller and is connected to the Ethernet-based switched network. Once again, we use the Directed Point communication system to drive the network and conduct all our experiments. In this study, we use four Fast Ethernet switches and one Gigabit Ethernet switch to set up various configurations to evaluate our algorithm.

The GE backbone switch is a chassis switch from Alcatel. It is the model PowerRail 2200 (PR2200) with backplane capacity reaches 22 Gigabit per second (Gbps). This switch is equipped with 8 GE ports on 2 modules, but we only use at most 4 ports in our experiments. Four FE switches are from IBM, which are of the model 8275-326. It is a 24-port input-buffered switch with backplane capacity reaches 5 Gbps. A one-port GE uplink module is installed on each FE switch for connecting to the Gigabit backbone switch. Table 6.1 summaries all the buffer parameters of the above switches, which are used in our algorithm to compute the global windows (W_g) on different network configurations.

To analyze and evaluate the performance of our congestion control mechanism, we have set up five different configurations on this cluster - 16x1, 8x2, 8x3, 6x4 and 8x4, with each configuration corresponds to a different degree of contention on the uplink ports (except configuration 16x1). The configuration *A*x*B* corresponds to connect *A* cluster nodes to each FE switch, and there are total *B* FE switches interconnected by the GE switch. This makes up a cluster size of $A * B$ nodes.

Switch/uplink	Architecture	B_L
Alcatel PR2200	Shared-buffered	820
IBM 8275-326	Input-buffered	43
IBM GE uplink	Input-buffered	45

Table 6.1: The B_L parameter of different switches in our experimental setup

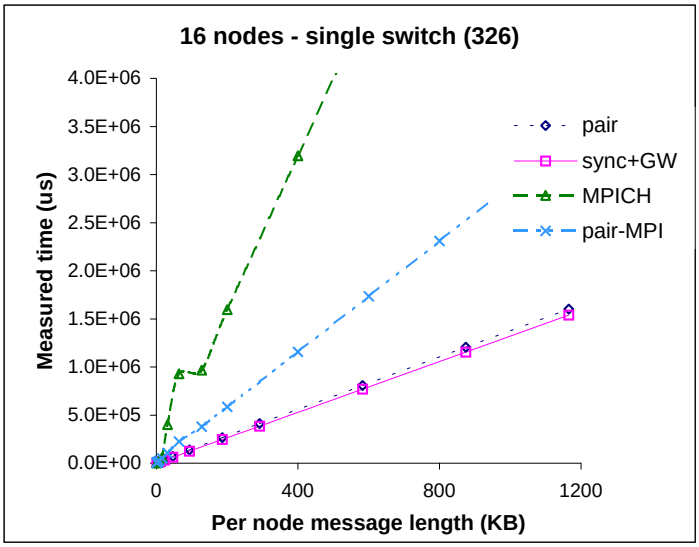
6.3.1 16-Node Single Switch - 16x1

The synchronous shuffle exchange is designed to work efficiently on any non-blocking network. However, in previous chapter, we have shown that there is internal constraint on an input-buffered switch, which limits the problem size scalability of our synchronous shuffle exchange algorithm. Although group shuffle exchange is devised to alleviate the problem, it only works sub-optimally from the analytical point of view. In this chapter, we have devised a new congestion control scheme to make synchronous shuffle exchange works efficiently on the hierarchical network. We consider that the same congestion control scheme can be applied to the single-router network to offset the limitation imposed by the HOL blocking.

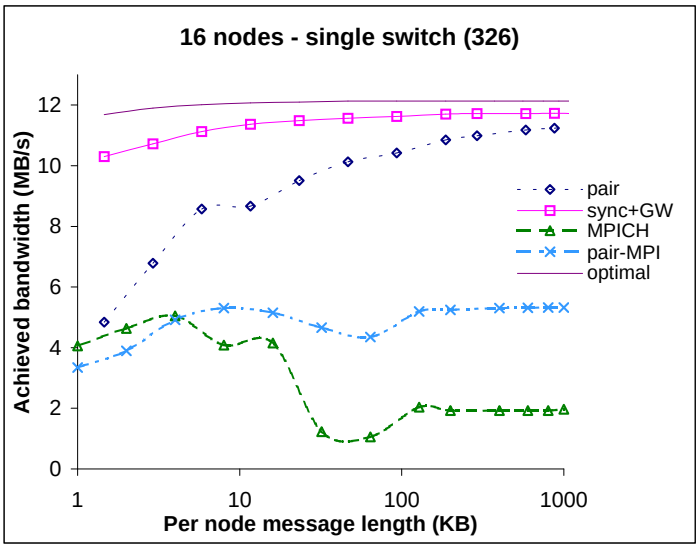
Figure 6.5 presents the measured results of four complete exchange implementations on a 16-node cluster interconnected by a single input-buffered switch (IBM 8275-326). They are the synchronous shuffle with global windowing (sync+GW), the pairwise exchange (pair), the original MPICH implementation (MPICH) and the pairwise exchange MPI implementation (pair-MPI). The experiment is conducted with each node sending a long message to every node in the cluster, which ranges from 1 KB to 1200 KB of data to each node. Both the measured performance and the per-node achieved bandwidth of each implementation are shown in the graphs.

We have shown in Figure 5.6 (Section 5.4) that the performance of the original synchronous shuffle algorithm degrades significantly after k>512, which corresponds to a message length of 746 KB per node. By supplementing the synchronous shuffle algorithm with the global windowing scheme, we show that it continues to operate efficiently as the problem size scales. When compared to the optimal performance (Eq. 5.3), the modified synchronous shuffle exchange algorithm has its efficiency ranged from 87% to 97% of the theoretical bandwidth. When compared with the pairwise exchange, the results show that the modified synchronous shuffle algorithm can effectively mask away synchronization overhead and achieves better performance. This shows that the add-on congestion control scheme does not affect the efficiency of our synchronous shuffle exchange algorithm. Indeed, it effectively guards against the congestion loss.

Not to mention on the poor performance of both MPI implementations, even though we are now using a faster processor and pumping the network with more data, their performances are restrained by the high protocol overheads. Although the pairwise MPI implementation generally performs better than the original MPICH implementation, we observe that the original MPICH implementation is slightly better on small



(a) Measured execution time



(b) Achieved bandwidth

Figure 6.5: Performance of modified synchronous shuffle exchange on a single input-buffered switch. (Legends: sync - synchronous shuffle; pair - pairwise; GW - global windowing)

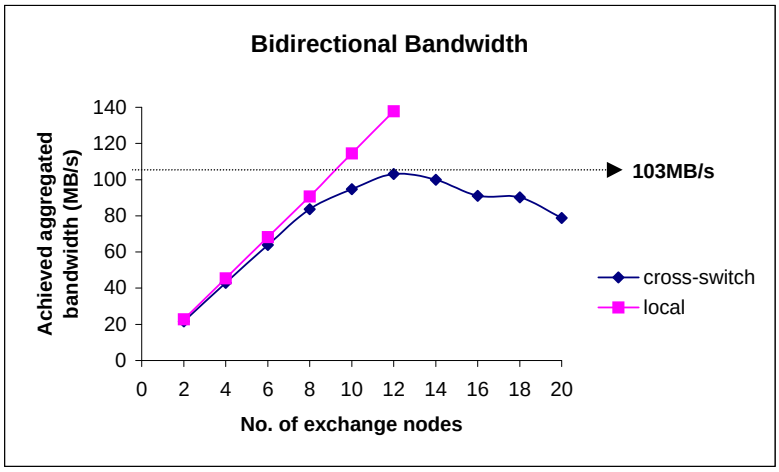


Figure 6.6: The achieved performance of the 10x2 hierarchical network under multiple bidirectional message exchanges. (Legends: cross-switch - measured aggregated bandwidth over the hierarchical network; local - measured aggregated bandwidth on the single switch)

message exchanges. This reflects that the use of non-blocking send and receive operations could hide away part of the synchronization overhead when exchanging small size messages and the induced contention problem is minimal, e.g. the use of eager protocol in the MPICH. However, when exchanging long messages, it would be better to have a well-coordinated schedule.

6.3.2 16-Node Hierarchical Configuration - 8x2

In this subsection, we start our experiments on the hierarchical network by first using a 16-node configuration. We are using two Fast Ethernet switches with eight nodes connect to each switch, and they are interconnected via the Gigabit Ethernet switch. With this setup, the theoretical *bisection bandwidth* [46] is 1 Gb/s, which should be sufficient for the current cluster configuration.

Baseline Studies Our experiment results in Section 4.3.4 have shown that the uplink circuitry of the IBM 8275-326 switch is not as good as it claims. In order to reason on the measured performance, such that we can make the correct judgment on the performance of our implementations, we have performed some baseline measurements to determine the best achievable throughput across these GE uplinks. Instead of using the 8x2 configuration, we have the 10x2 configuration and measure the achieved aggregated bandwidth across the hierarchical network by having multiple concurrent bi-directional data exchanges. Figure 6.6 shows the results of this baseline study. The peak aggregated bandwidth achieved on this setting is 103 MB/s with 12 concurrent bi-directional flows across the uplink ports. Beyond that, the communication performance starts to deteriorate gradually. With the same software and hardware settings, but re-

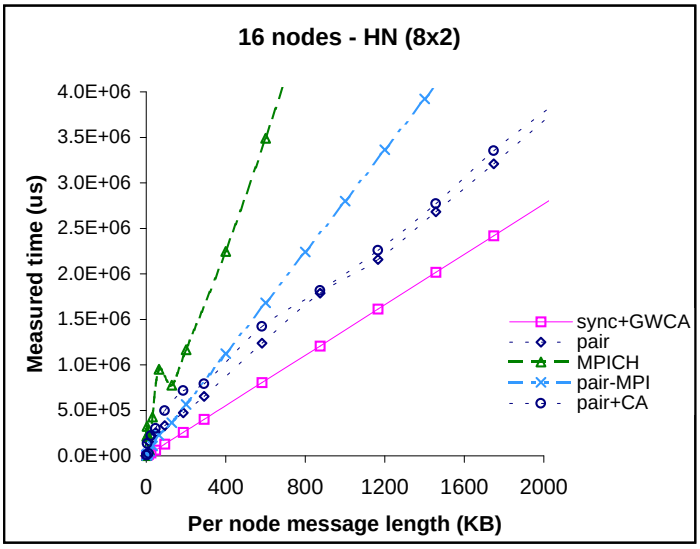
placing the hierarchical network with a single IBM 8275-326 switch, we can achieve a linearly scaled aggregated bandwidth, which is labeled as "local" in the graph. This demonstrates that the limitation is on the uplink circuitry, not on other components. With this baseline measurement, we have a solid foundation to justify on the expected communication efficiency across the problematic uplink ports, such that we have

$$\text{Best cross switch data exchange time} = \frac{\text{Total cross switch volume}}{103\ MB/s} \tag{6.4}$$

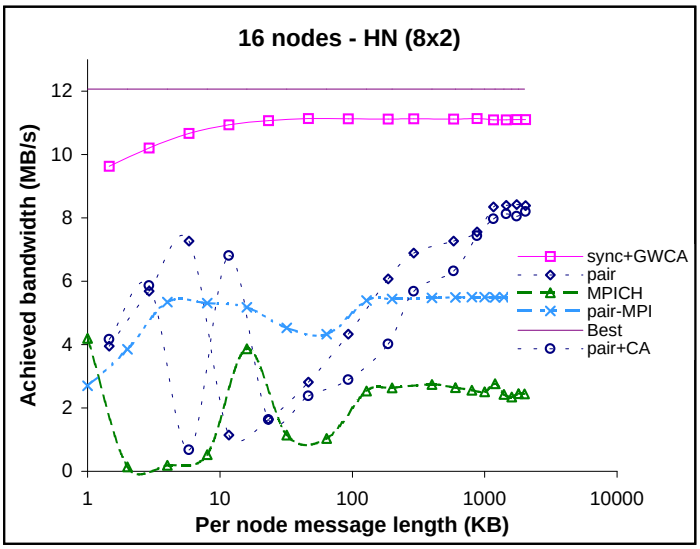
Take an example with the 8x2 configuration, the total cross-switch volume on the *k*-item complete exchange is $2k * \text{MTU} * (16 - d_1) * d_1$ bytes. Thus, the best timing in delivering this volume of data across the uplink connection is $\frac{128*k*\text{MTU}}{103}$seconds. Assumed that an efficient communication schedule should be able to arrange all local and cross-switch communications be happened concurrently. Therefore, the execution time of the *k*-item complete exchange should be bounded by the best cross-switch data exchange time. Then, the best achieved per-node bandwidth for this *k*-item complete exchange operation is $\frac{k*\text{MTU}*15}{\frac{128*k*\text{MTU}}{103}} = 12.07$ MB$/s$.

After understanding about the performance limitation of the network, we carry on with our analysis. With the 8x2 configuration, the theoretical computed value of W_g is 10; however, when considered together with implementation issue, such as the existence of control packets with reliable support, the calculated value of W_g is $\left\lfloor 45 \div \frac{(16-8)*8*2}{15} \right\rfloor = 5$. We have measured the performance of the modified synchronous shuffle algorithm with this global windowing setting, and the results are presented in Figure 6.7. Similarly, we are comparing different implementations of the complete exchange operation on this configuration. Five sets of measurements are shown in the graphs. They are the synchronous shuffle with global windowing and contention-aware permutation (sync+GW+CA), pairwise exchange (pair), pairwise exchange with contention-aware permutation (pair+CA), the original MPICH implementation and the pairwise exchange MPI implementation (pair-MPI). The results show that synchronous shuffle exchange with global windowing and contention-aware permutation performs the best amongst all tested implementations in this configuration. When compared to the expected best achievement, the modified synchronous shuffle shows its effectiveness in utilizing the network pipelines as well as avoiding the congestion loss, since it reaches 93% of the best achieved performance.

However, we find that the performance of the DP pairwise exchange implementation has degraded considerably under this hierarchical configuration when compared to its performance on the single-switch case (Figure 6.5b). Initially, the performance of the DP pairwise implementation (labeled as "pair") increases with the increase in message length until the maximum capacity of the uplink ports has reached. After that, the performance is affected by the congestion loss problem. However, our GBN reliable protocol could only recover from the loss with long message exchanges. This is being shown as the slow increased in the achieved bandwidth after experiencing the congestion loss problem. To investigate on whether the contention-aware permutation



(a) Measured execution time



(b) Achieved bandwidth

Figure 6.7: The performance of modified synchronous shuffle exchange on the 8x2 configuration - 8 nodes connect to each FE switch, which is connected to the PR2200. (Legends: GWCA- global windowing plus contention-aware permutation scheme; CA-contention-aware permutation scheme only)

scheme would also benefit the pairwise exchange algorithm, we have applied the same contention-aware permutation on the DP pairwise implementation. The measured results (labeled as "pair+CA") show that this augmentation exhibits a similar behavior as compared to the pure pairwise exchange. However, the congestion loss problem appears earlier than we have expected, and the overall performance is slightly worse than the pure pairwise exchange implementation.

On the other hand, it is interested to see that the performances of the two MPI implementations do not have significant performance changes on this hierarchical configuration. We find that they both have slight improvements on exchanging long message, but the original MPICH implementation has lost its performance on exchanging small messages. This could be the result of contention over the uplinks as the MPICH implementation does not carefully schedule those communication events. This indicates that under this hierarchical configuration, it demands for a better communication scheme to coordinate the communication events, since the performance is limited by the aggregated bandwidth.

6.3.3 24-Node Hierarchical Configurations - 8X3 and 6X4

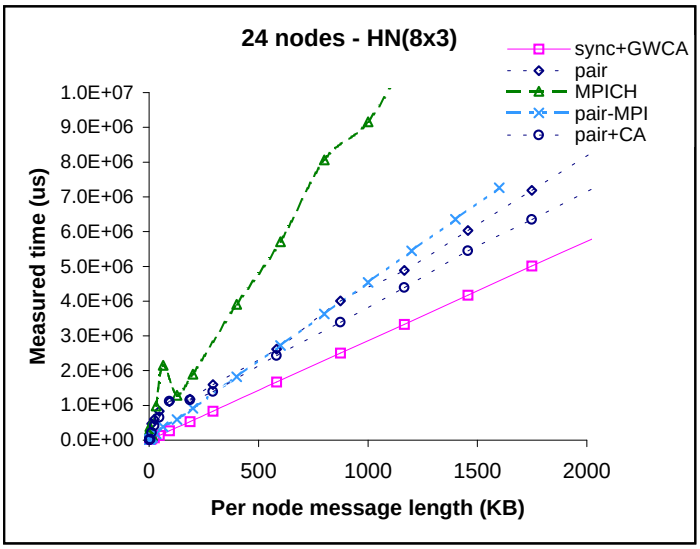
To construct a 24-node cluster with our hardware resources, we can arrange the hierarchical network in two different configurations:

1. 8X3 - By connecting three Fast Ethernet switches to the Gigabit Ethernet switch, and each FE switch has eight cluster nodes connected to it, we get a 24-node cluster. With this configuration, the computed value of W_g is 4 and the experimental results are shown in Figure 6.8.

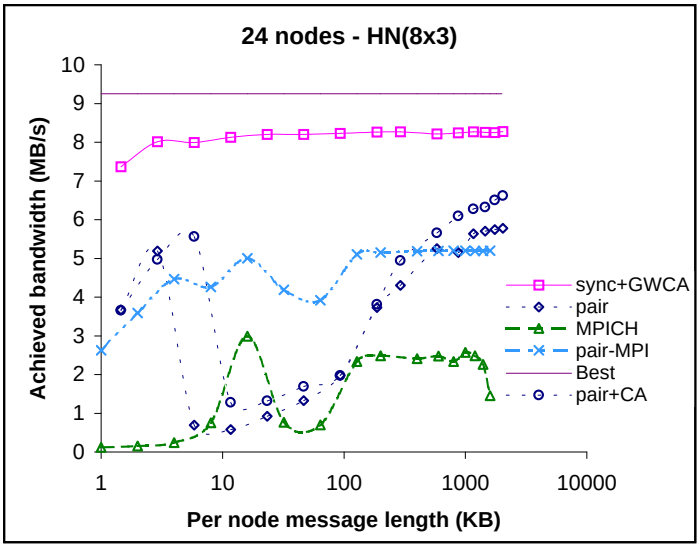
2. 6X4 - All four FE switches are connected to the GE switch with six cluster nodes attached to each FE switch, we have the second 24 nodes configuration. With this configuration, the computed value of W_g is 5 and the experimental results are shown in Figure 6.9.

We have performed the same set of tests on these two configurations as compared to the 8X2 setup. When comparing their expected best achievements on these two configurations, which are of 9.25 MB/s on the 8X3 configuration and of 10.96 MB/s on the 6X4 configuration, we see that the 6X4 configuration has a better throughput performance. This is reasonable since we only attach 6 nodes to each FE switch on the 6X4 setup. Again, we see that the modified synchronous shuffle performs the best on both setups; however, it only reaches 89% and 88% of the expected best achievements on the 8X3 and 6X4 configurations respectively. A possible explanation on the performance degradation is due to the use of small global window settings, which may reduce the pipelining efficiency. However, increase the global window size would increase the congestion loss probability, this could create an adverse effect on the overall performance.

As for the two MPI implementations, their measured performance look similar to the performance observed in the 8X2 configuration. Such that the achieved bandwidth

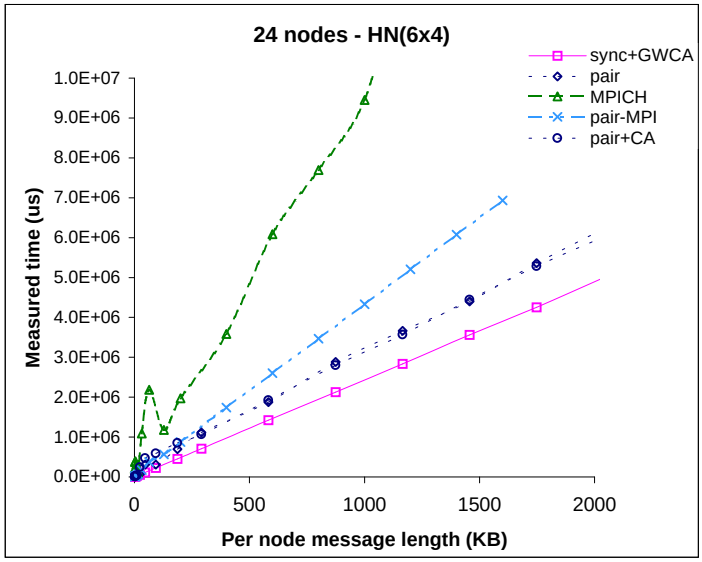


(a) Measured execution time

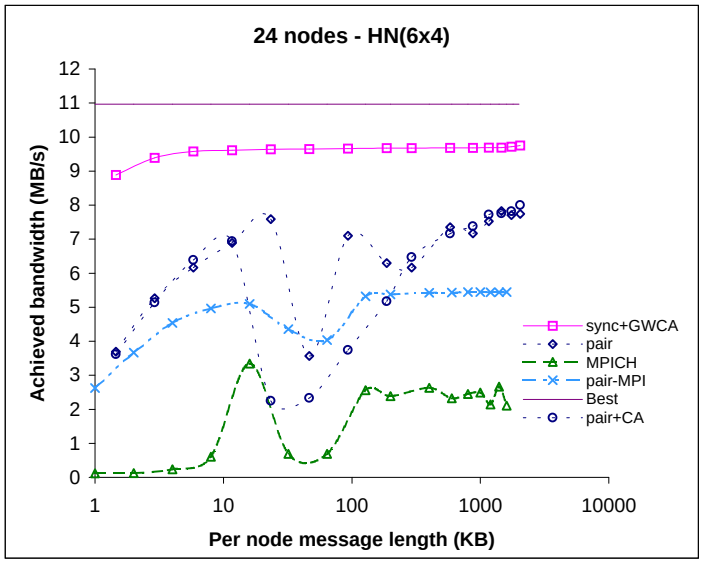


(b) Achieved bandwidth

Figure 6.8: Performance of different complete exchange implementations on the 8x3 hierarchical configuration



(a) Measured execution time



(b) Achieved bandwidth

Figure 6.9: Performance of different complete exchange implementations on the 6x4 hierarchical configuration

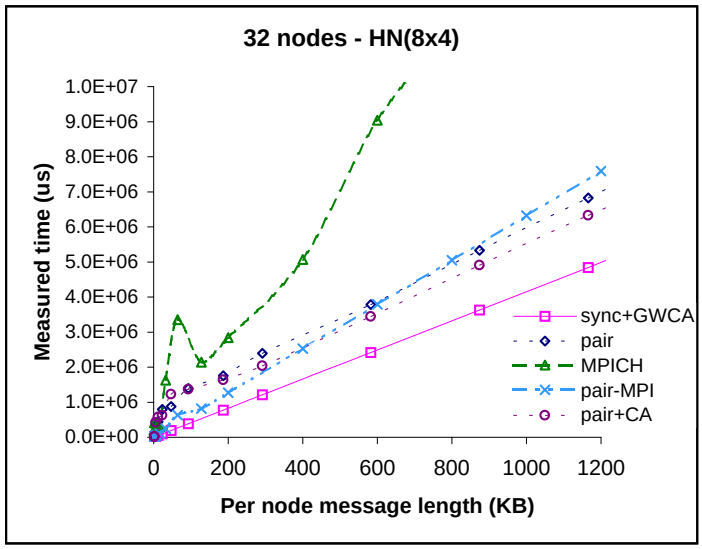
of the pairwise MPI implementation is peaked at 5.4 MB/s and 5.2 MB/s on 6x4 and 8x3 respectively, as compared to 5.5 MB/s on the 8X2 setup. Similarly, the achieved bandwidth of the MPICH implementation is peaked at 2.7 MB/s and 2.6 MB/s on 6X4 and 8X3 respectively, while it achieves 2.7 MB/s on the 8X2 setup. Besides, we observe that as the cluster size has increased from 16 nodes to 24 nodes, the observed contention problem of the original MPICH implementation on small message exchanges is getting worse than the 8X2 configuration. Nevertheless, all these findings support our belief that conventional communication libraries are restrained by the high software overheads.

As for the DP pairwise implementations, their measured results exhibit different performance behaviors on these two configurations. On a configuration that supports less aggregated bandwidth (8X3), we find that we have experienced severe performance loss starting at small size message exchanges. But, with a less restrictive configuration (6X4), the losses start to appear only on medium size message exchanges. After that, the performance slowly increases when exchanging longer messages. On the other hand, we find that the add-on contention-aware scheme (pair+CA) performs better on the 8X3 configuration when compared to the pure pairwise implementation. A possible explanation to this observation is that the contention-aware scheme is more effective when operates on a more stringent configuration.

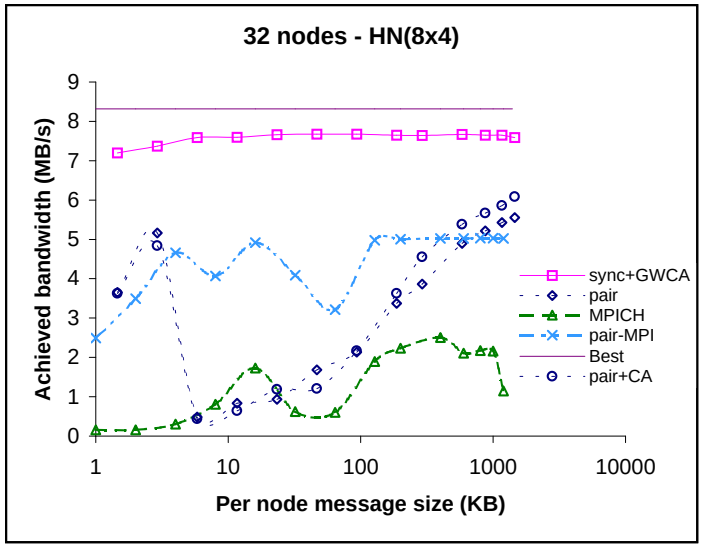
6.3.4 32-Node Hierarchical Configuration - 8X4

With this configuration, we interconnect four FE switches to the GE switch, and each FE switch is attached with 8 cluster nodes. Carry on with the same set of experiments with the global windowing parameter (W_g) set to 3, the measured results are shown in Figure 6.10. Same as other experiments, the modified synchronous shuffle algorithm shows its clear advantage over other implementations when running on this hierarchical configuration. In particular, it achieved per-node bandwidth peaks at 7.67 MB/s, which is around 92% of the expected best achievement (8.32 MB/s) on this configuration.

As for the two MPI implementations, their performances closely match with other configurations, which are peaked at 2.5 MB/s on the per-node bandwidth for the MPICH implementation and 5.03 MB/s on the pairwise MPI implementation. However, the performance of our DP pairwise exchange implementation suffers considerably when exchanging small to medium size messages, as the results show that the pairwise MPI implementation outperforms the DP pairwise implementations on this message range. This indicates that our GBN reliable protocol is not working as effectively as the TCP protocol except with large message exchanges. Once again, we observe that the add-on contention-aware permutation on the pairwise implementation has slight performance improvement on the current configuration. When compared this finding with the results on other configurations, we believe that the contention-aware scheme works better on a more stringent configuration. This observation shows that contention-aware permutation alleviates the congestion build-up at the uplink ports; however, it still has to work together with the global windowing scheme in order



(a) Measured execution time



(b) Achieved bandwidth

Figure 6.10: Performance of different complete exchange implementations on the 8x4 hierarchical configuration

to avoid congestion loss.

6.4 Related Work

The uses of models in designing algorithms, in particular communication algorithms, are not uncommon in the parallel community [47, 53, 73, 76, 88]. These algorithms are mostly shown to be efficient or optimal with respect to the based models by using paper and pen or simulations. For example, Karp et al. [53] had designed an optimal *k*-item broadcast algorithm based on the LogP model [30]. This algorithm is presented as a communication schedule where the root sends each data item only once, but alternates among recipients in order to retain the logarithmic depth of a tree broadcast. The principle behind this communication schedule matches with our synchronous shuffle exchange algorithm, as both schedules try to arrange the communications such that a processor would only at most send out one data item and receive one data item in one communication step. However, as demonstrated in our experimental studies, these types of communication schedules may be too idealistic, which demand to have logical synchronism in order to achieve the optimal bounds; therefore, it is hard to achieve the claimed performance on a real platform unless more control of the communications is enforced.

Similar to our contention-aware permutation, Nupairoj et al. [73] has reported that to ensure the efficiency of their optimal multicast algorithm, which is built on top of a parameterized model [76], it is necessary to consider some architecture-dependent characteristics of a system. This is because in real network, network contention is likely to occur if concurrent message transmissions are not scheduled properly. Their approach makes use of the topological information from the mesh or multistage networks, to order those communications in the multicast tree to avoid network contention. This supports our belief that optimal performance achievable on parallel machines can be first designed by using architecture independent model, and then perform performance tuning of an implementation on the based of some architecture dependent characteristics.

Donaldson et al. [34] also reported that the BSPlib, a communication library for BSP [105] programming, is using information related to the global state to improve the collective use of the communication system. Due to the synchronous nature of the BSP machines, communications are constrained to take place only in certain stages. This constraint, although limits its flexibility, can transform to enhancement on communications as each processor can infer the global state of the communication in which it is involved. For example, a BSP process knows which other processes are about to communicate, and how much they plan to send, and then can estimate the global network loading; this in turn, can be used to determine the ideal schedule and transmission rate. Their objective is similar to our approach, but we are working on an asynchronous model, which makes use of the buffering information and communication pattern to devise the corresponding information.

We make use of the taxonomy introduced by Yang et al. [120] as a reference for

comparing different congestion control schemes. Broadly speaking, our global window congestion control mechanism is an anticipatory, closed loop control scheme with implicit feedback of global information. By anticipatory control, this means that the control is a congestion avoidance scheme, which tends to drive the network toward the optimal operating point but without falling into the danger of congestion. With closed loop control, this stands for the use of implicit feedback information to the sources as the control decisions in regulating the traffics. In our case, the control information is derived from the total number of unacknowledged packets for all communicating partners. Under the same terminology, the slow start scheme used in the TCP [51] could be considered as a reactive, closed loop control with implicit feedback, which based on the round-trip delay of acknowledgements [120] on individual end-to-end flow. Besides using the acknowledgements as the feedback control, other means of feedback can be used. For example, the Warp control scheme [77] makes use of a time-stamp based measure, called Warp, to monitor network utilization, and to control the injection rates for achieving optimal utilization. However, like the slow start scheme, it is also a reactive scheme that based on end-to-end feedback information.

Being an anticipatory scheme, our proactive approach is similar to the congestion control schemes used in some of the ATM networks. These schemes usually operate through input rate regulation [89], which relies on the underlying network devices to support and manage the resource allocation and scheduling. For example, the use of explicit feedback signals that generated or set by the congested devices to alert other network devices or communicating nodes on the growing congestion. With these control schemes, during the call-setup procedure, the information sources are assigned with some predefined rate, and are forced to limit their average input rate below this value. This ensures that no source will exceed for an extensive period of time the rate provided by the network and avoids congestion.

Instead of using the buffer resources as a guide to monitor the congestion problem, other approaches have been proposed to explicitly manage the allocation of scarce resources to different purposes. For example, Roy et al. [87] propose the uses of Quality of Service (QoS) mechanisms at the application level to manage contention, so as to improve the performance of MPI applications. They argue that if appropriate mechanisms can be provided for expressing application requirements, for arbitrating between different requirements, for enforcing allocations, and for providing feedback to applications concerning achieved performance, then applications can adapt their behavior according to resource availability. However, their approach require that the underlying network supports the QoS mechanisms.

6.5 Summary

In this chapter, we have presented an efficient implementation of complete exchange operation on the Ethernet-based hierarchical network. The example hierarchical network is based on a Gigabit switch as the backbone, which interconnects all Fast Ethernet switches. We believe that the hierarchical network model is a practical design to

construct large-scale clusters. With this system configuration, clusters can be scaled up to hundreds or thousands of nodes, and support enough bandwidth for high-speed communication. Our research can be used in any combination of Ethernet-based switched networks, which we belief, over 60% of the self-made clusters are based on. And the concept is applicable to future technologies, such as 10 Gigabit Ethernet, which simply extends the topology to multi-level hierarchy.

We have demonstrated that the contention problems on such network, such as the link, the node, and the switch contention, can severely affect the overall performance of the clusters. To avoid congestion loss on this type of network, we propose the use of synchronous shuffle exchange algorithm with congestion control scheme and contention-aware permutation. With the proactive approach in handling congestion, this algorithm makes use of architectural characteristics to avoid congestion build-up in the first place and reduces congestion whenever it happens. We derive a global window scheme from information on the network buffer capacity, which forces each node to limit their traffic loads and ensures a fair sharing of network resources that avoids congestion overflow. We also make use of information on the network topology to derive a contention-aware permutation in generating a communication schedule, which avoids contention at the node and at the switch, as well as creating a more evenly distributed traffic pattern on the network. This improves the synchronism of the traffic information exchange between cluster nodes, and hence, improves the effectiveness of the global windowing scheme in monitoring the network. The proposed algorithm is implemented on a 32-node cluster with different network configurations. And the results have showed that it can efficiently utilize the network as well as effectively control the congestion problem.

Chapter 7

Conclusions and Direction for Future Works

In this chapter, we summarize and conclude the thesis. To achieve our objective of using commodity clusters for supercomputing, this dissertation proposes the use of a realistic communication model for performance understanding, as well as for algorithm design and analysis. We have concentrated on identifying the essential properties of the communication architecture which have significant impact on the performance, and devising benchmark methodologies to quantify their performance characteristics. By organizing these architectural features as a performance parameter set, we have created a framework for the programmers to conduct performance understanding, performance calibration and performance prediction. We have applied this modeling framework in examining the performance characteristics of two implementations of the Directed Point lightweight messaging system. In particular, we have demonstrated the effectiveness of using the model as an evaluation tool in delineating the strength and weakness of these communication systems, as well as using the model as a emulating tool for assessing various design tradeoffs.

Our communication model is distinguished from other performance or abstract models on its supports of the congestion studies and pipelining communication, because our model is derived on the base of a resource-centric viewpoint. Pipelining communication is the key to achieve high-bandwidth data transfers in modern networks by maximizing the number of concurrent data movements. This is achieved by maximizing the uses of available network resources. However, without well coordination and scheduling, aggressive pipelining may lead to contention development, which results in performance loss. This is being shown in our studies of the congestive loss problem in Chapter 4. Based on the network buffering information provided by our model, we artificially controlled the degree of congestion happened on the cluster network. Observing through both modeling studies and experimental evaluations, we have examined how different buffering architectures interacted with our Go-Back-N reliable protocol and affected on the congestion development of the communication system when subjected to heavy congestion. Through these performance studies, we obtain valuable insights on guiding of the design of efficient reliable transmission pro-

tocol on top of the lightweight messaging systems.

Since all congestion problems are caused by the contention of resources, with the availability of the resource information described by our model, we can utilize these information to devise high-level communication schedules, which optimize the pipelining efficiency as well as guard against the congestion loss problem. In this study, we make use of the Complete Exchange operation to validate on the above conjecture. We have devised the Synchronous Shuffle Exchange algorithm, and have demonstrated by means of analytical and experimental studies that it is an optimal algorithm on any non-blocking network. To maximize the pipelining efficiency, we adopt a contention-free schedule at the packet level. This completely eliminates unnecessary startup and synchronization overheads.

Even with a contention-free communication scheme, other factors could introduce non-deterministic delays to well-scheduled communication events, such as variations in process scheduling and competition with high-priority system activities. We have shown that the buffering architecture of the switch plays a crucial role under such a circumstance. This is because, the synchronous shuffle exchange algorithm achieves high-performance by fully utilizing the network links, any variations in communication schedules will induce contention, and this is handled by the switch's buffers. However, the network buffers are scarce resources. Furthermore, we have demonstrated that switches with input-buffered architecture as well as the hierarchical networks are more vulnerable to these non-deterministic delays. To solve this problem, this dissertation proposes the use of our model parameter (B_L) to derive a congestion control scheme, which limits the traffic loads and ensures a fair sharing of network resources that avoids buffer overflow. To improve the effectiveness of the congestion control scheme when working on the hierarchical network, we have incorporated information on the network topology to devise a contention-aware permutation. This permutation scheme generates a communication schedule, which is both node and switch contention-free as well as distributing the network loads more evenly across the hierarchy. This relieves the congestion build-up at the uplink ports and improves the synchronism of the traffic information exchange between cluster nodes.

7.1 Contributions

Although this dissertation focuses on the performance issues related to commodity clusters, the principle behind this thesis research should be applicable to other parallel computers with the same architecture foundation. To conclude, we organize and discuss our accomplishments around areas of contributions - modeling and performance understanding, congestion studies, and algorithm design and analysis.

Modeling and Performance Understanding Computer systems are evolving rapidly. The development of computer systems is highly complex that demands for a systematic approach in performance understanding. This dissertation demonstrates the importance of having both quantitative and qualitative metrics in performance

understanding. With the quantitative information, we can evaluate, question and analyze how the system performs. While with the qualitative information, we can predict, analyze and explain how the application behaves. One of the approaches on performance understanding is the use of modeling techniques. The foundation of this thesis research is the development of a realistic communication model that captures the performance characteristics of the target machine and serves as a practical tool for design and analysis of algorithms. This includes a set of microbenchmarks to quantify the resources information and a set of performance parameters to delineate the performance characteristics of the communication system. The model parameters hold the quantitative and qualitative information for both programmers and system designers to understand or reason about their program/system design decisions.

Congestion Study This thesis demonstrates that the buffering mechanism within the routers or switches has a paramount influence on the communication performance. We make use of the available information on the network buffering to investigate the congestion problem by two different approaches. To begin with, we explore how the software and hardware components interacted when the communication network is under heavy congestion. Our analytical and experimental results show that under asymmetric traffic loads, the output-buffered mechanism is more susceptible to the congestion loss problem than the input-buffered mechanism. Although input buffering has a higher threshold, once the overflow situation occurs, the resulting communication performance drops significantly. Besides, we find that the behavioral different between the two buffering mechanisms lies on how they interact with the communication protocol.

The second part of our investigations on the congestion problem is on the design of high-performance communication algorithms, which can efficiently utilize the network resources as well as can avoid the building up of congestion. The unique feature of our approach is the uses of resources information provided by our communication model to guide our design and analysis. We have introduced a global congestion control scheme to the complete exchange algorithm, and have demonstrated that it effectively avoids congestion loss and maintains sufficient throughput to maximize the performance. The principle behind the global congestion control scheme is to prevent oversubscribing the network, and the scheme is derived from information related to the communication pattern and volume, a well as from modeled architectural features. Improving the congestion control with the ideas presented in this dissertation has made it an effective solution for high-performance communication on commodity networks.

Algorithms Analysis and Design We have devised an efficient communication schedule for the complete exchange operation - the Synchronous Shuffle Exchange, and have shown that it is an optimal algorithm on any non-blocking networks. Although our experimental results showed that the synchronous shuffle exchange is realizable and efficient, in reality, there are limiting factors that restrain its performance. This is commonly happened when porting theoretical algorithms on to the real platforms, as algorithms are designed and analyzed on a simplified/abstract platform.

In this dissertation, we unknot the problems by augmenting the algorithms with mechanisms that are derived from our communication model. This shows the importance of using a practical performance model to perform algorithm design and analysis.

7.2 Future Directions

This dissertation covers a broad scope on the studying of performance issues, however, certain issues have not been fully addressed, which become areas for our future investigations.

Modeling We believe that our performance model is a general model that covers a broad spectrum of parallel platforms. However, one of the important features of modeling is that models must not be made obsolete by advances in realizing technology. It is interest to see how our model fits to the next waves of technology achievements. For examples, on the networking aspect, we have 10 Gigabit Ethernet [5], WDM optical networks and the InfiniBand architecture [6]; on the system area, we have the massively produced 64-bit microprocessor, VIA communication infrastructure [107] and PCI-X [78]. All these technologies will increase the performance of data communication; however, we reckon that problems induced by congestion and buffering architecture will still have considerable impact on the performance.

User-level Reliable Support Reliable support on lightweight messaging systems needs further studied. Our studies show that the reliable protocol interacts with the network buffering architecture on the congestion development; however, we still lack of information on determining which reliable mechanism is best suit for low-latency high-performance communication. For example, von Eicken et al. [108] has shown that the blame of poor TCP performance is not on the protocol, but on the particular implementation and its integration into the operating system. Their experiments only supported that we may have good TCP performance on light-load communications, but did not provided evidence that the TCP is still effective for handling congestion problem on low-latency communication under heavy congestion. Therefore, it is worthwhile to investigate on how TCP interacts with the network buffering architecture on the congestion development. The results could provide useful information to guide us to achieve better performance on situations under heavy congestion.

Furthermore, in the implementation of the reliable layer on top of DP, we find that one of the hindrances on the performance of user-level reliable support is the delay in delivering the control information. As the underlying network does not differentiate between control and data packets, it handles all packets with the same priority; this could delay some high priority events, e.g. Nack. We are investigating on using existing features of commodity networks, such as priority queues and virtual lanes, to improve the situation. This is importance to the development of congestion control scheme as having faster control information exchanges improves efficiency of the control scheme.

Contention Study Our approach on tackling the congestion problem works the best for scheduling regular communication patterns on a well-structured, enclosed network. Two possible areas of extensions are the studies of irregular communication patterns and the uses of asymmetric/irregular networks. Another limitation of this thesis study is our optimization technique such as global congestion control is implemented in off-line approach. It is worthwhile to implement it in on-line mode, such as automation through compiler directives and runtime supports. For example, a large-scale cluster may be using in a space-shared mode by several concurrent parallel applications. If we assume that each parallel application is allocated with disjoint set of cluster nodes, how can the congestion control scheme adapt to the resulting irregular configuration and irregular traffic pattern?

Appendix A

Benchmark Methodologies

In this Appendix, we briefly describe the benchmark methodologies that we use to derive those modeling parameters (in Chapter 2). Although our microbenchmarks are originated from the programming abstraction of DP, we believe that our techniques apply to other low-latency communication systems, as DP programming interface follows the traditional message passing paradigm [111].

A.1 Microbenchmark for the B_L Parameter

The B_L parameter of our model provides two information related to the buffering system: (1) it displays the maximum number of buffer units associated to a switch's port, and (2) the associated buffering architecture adopted by this switching unit. As the primary usage of providing buffer storage is to handle temporary congestion, the natural starting point of our investigation is to create artificial congestion problem. For this communication microbenchmark, we carry out a set of tests that systematically generate traffics from multiple input ports (*S*) to a prefixed set of output ports (*R*). On each test, we start with a short burst of packets (of predefined volume and packet size), generated simultaneously from all the source nodes, and are flooded to the target destination nodes (ports). After injecting the burst into the network, all source nodes will stall for a specific duration, which is long enough for the switch's buffers to clear off their loading and for the receivers to count how many packets have they got. Then the program increases the burst volume, and continues the same process until the target burst volume is reached. By using different number of source nodes (*S*) and sink nodes (*R*) on each test, we collect a set of loss patterns which becomes the buffering signature of the switching unit in question.

Before discussing on how to draw the conclusion from the buffering signature, we first illustrate the theory behind our microbenchmark. Assuming that the network is error-free, all packet losses are caused by buffer overflow, either in the switch port or at the receiver node. The prerequisite of having buffer overflow is the unbalance of flow across the buffer region, so we simplify the bottleneck region as a staging buffer between input and output pipes, as shown in Figure A.1. Intuitively, the staging buffer starts queuing up data items when the departure rate is slower than the arrival

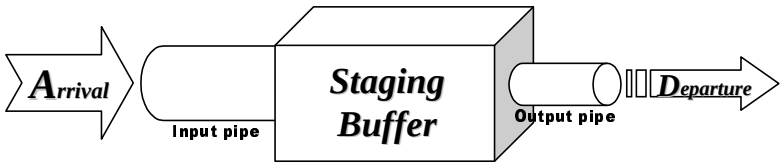


Figure A.1: Bottleneck stage phenomenon

rate. Thus, the ratio between the departure rate and arrival rate becomes a measure of congestion. Due to the limited size, this staging buffer cannot sustain long-term congestion, and eventually starts dropping all newly arrived packets when it is full.

Here we derive a deterministic model to delineate this overflow problem. Let the staging buffer be of size $\tilde{B}$ units, and the arrival and departure rates over this bottleneck stage be *A* and *D* units per second respectively. Assume that at $t = 0$, the staging buffer is empty, and with an unbalance flow ($A > D$), packets start to cumulate over the staging buffer. Now at $t = \beta$, the buffer becomes full, then we have

$$A \cdot \beta = \tilde{B} + D \cdot \beta \qquad \Rightarrow \qquad \beta = \frac{\tilde{B}}{A - D} \tag{A.1}$$

Therefore, when $t \leq \beta$, the system can accommodate all arrivals and the success probability is equal to one. However, after buffer saturation, at $t > \beta$, newly arrived packets can only be accepted and forwarded with a probability of $\frac{D}{A}$. Hence, we expect that the probability of success (ρ) on moving k packets (where $k > \tilde{B}$) across this bottleneck stage when $A > D$ is:

$$\begin{aligned} \rho &= \frac{\left(k - \frac{A*\tilde{B}}{A-D}\right) * \frac{D}{A} + \frac{A*\tilde{B}}{A-D}}{k} \\ &= \frac{D}{A} + \frac{\tilde{B}}{k} \end{aligned} \tag{A.2}$$

We see from equation (A.2) that by varying these parameters, we are expecting to get different packet loss ratio:

- the arrival rate *A* - by controlling the number of network sources *S* and driving them in full speed, we have $A \simeq \frac{S}{g_s}$.
- the departure rate *D* - by choosing the number of sink nodes *R*, we have $D \simeq \frac{R}{g_r}$.
- the burst volume *k*

Thus, by plotting the measured success ratio against the inverse of the burst volume (*k*), we would expect to get a linear equation, which has $\tilde{B}$ as its slope and $\frac{R*g_s}{S*g_r}$as its

Switching unit	Architecture	B_L
IBM 8275-326	Input-buffered	43
IBM 8275-326 (uplink)	Input-buffered	45
IBM 8275-416	Output-buffered‡	95
Intel 510T	Shared-buffered	1990
Intel 510T (uplink)	Shared-buffered	1990+510
Cisco Catalyst 2980G	Output-buffered‡	128
Cisco Catalyst 3524	Shared-buffered	650
Alcatel PowerRail 2200	Shared-buffered	820

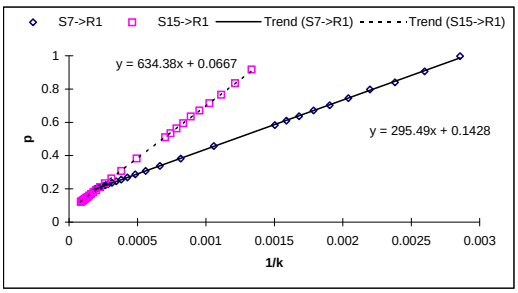
Table A.1: B_L parameters of different Ethernet switches
‡ Both switches could be built on a shared-buffered architecture, but the buffer allocation scheme may impose an upper limit on each output buffer, hence, creates the output-buffered signatures.

y-intercept. Besides, by using different combination of S and R, the resulting $\tilde{B}$ value would reflect the internal buffer architecture adopted by that switching unit. Table A.1 summarizes the B_L parameters for some switches we have measured by using this microbenchmark. We also present the graphical signatures of three switching units in Figure A.2, which serve as the sample signatures to demonstrate how different buffer architectures look like under our microbenchmark.

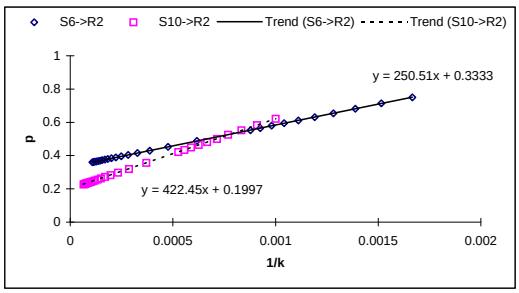
A.2 Microbenchmark for the O_r Parameter

In our model, the arrival of data packets from the network is an asynchronous event, and therefore, we cannot directly probe the source code for measuring the software overhead associated to the reception event. Consequently, we have to rely on indirect probing. Observe that without the support of programmable network co-processor, dispatching of arrived packets has to be carried out by the receiver host processor. This is because of the protection reason, moving data across address spaces should be handled by privilege process only. As the host processor is engaged during message reception, other execution flows would be affected. Based on this observation, consider if we know about the expected execution time of a particular computation segment. And on a particular run of this computation segment, there is message reception happened in the background, we would expect to observe a remarkable increase in execution time. Thence, we have devised a technique to indirectly measure the software overhead associated to any message reception. In summary, Algorithm 6 presents the pseudocode for our O_r microbenchmark.

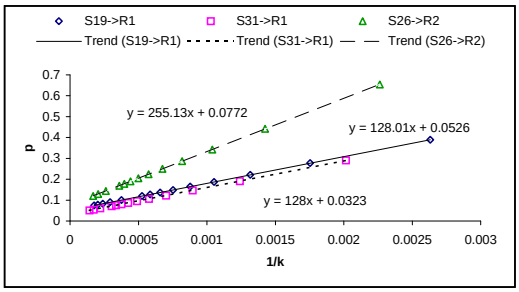
Our technique to measure O_r is quite straightforward. As we need to detect whether a message (packet) has arrived, we have to poll for the data arrival. Thus, we use the polling action to be the computation segment. However, if the software overhead associated to this receive call is too large, we should devise other lightweight computation



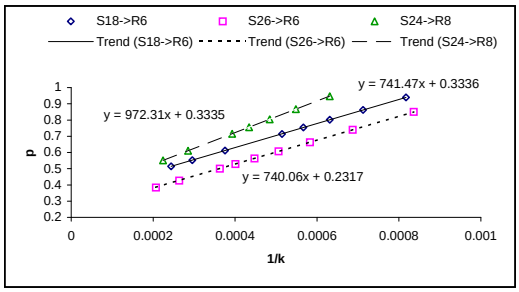
(a) IBM 8275-326 (Input-buffered)



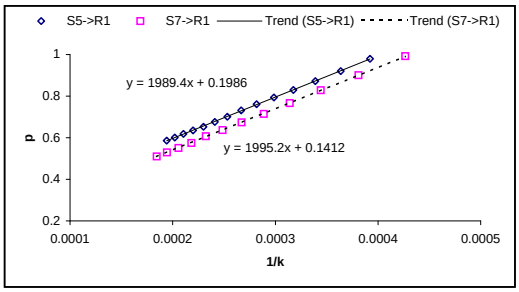
(b) IBM 8275-326 (Input-buffered)



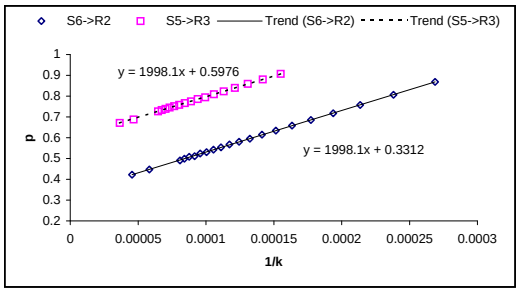
(c) Cisco 2980G (Output-buffered)



(d) Cisco 2980G (Output-buffered)



(e) Intel 510T (Shared-buffered)



(f) Intel 510T (Shared-buffered)

Figure A.2: Microbenchmark signatures for (a)(b) IBM 8275-326, (c)(d) Cisco 2980G, and (e)(f) Intel 510T.

Algorithm 6 The O_r microbenchmark

```
INPUT
        message size x                          (1 ≤ x ≤ PDU)
        master node a
        slave node b                            # echo server

ALGORITHM
        barrier (a, b)
        RTT ⟵ pingpong_test (a, b, x)           # measure roundtrip time
        barrier (a, b)
        for i = 0 to 1000 do
                send_message (a, b, x)
                sleep (2*RTT)
                T0 ⟵ clock ()
                msg ⟵ async_receive (b)         # computation segment
                T1 ⟵ clock ()
                Ur ⟵ Ur + (T1 - T0)
        endfor
        Ur ⟵ Ur ÷1000                           # use average value
        i ⟵ 0
        do {
                T2 ⟵ 0
                send_message (a, b, x)
                T0 ⟵ clock ()
                do {
                        msg ⟵ async_receive (b)
                        T1 ⟵ clock ()
                        T2 ⟵ max (T2, (T1 - T0))
                        T0 ⟵ T1
                } while (msg == NULL)           # just check the clock until msg return
                if ( (T2 > 2*Ur) && (T2 < ½RTT) ) then
                        t[i] ⟵ T2
                        inc i
                else
                        noise measurement, discard T2
                fi
        } while (i < ITERATION)
        Or(x) ⟵ statistical_analysis (t[])
```

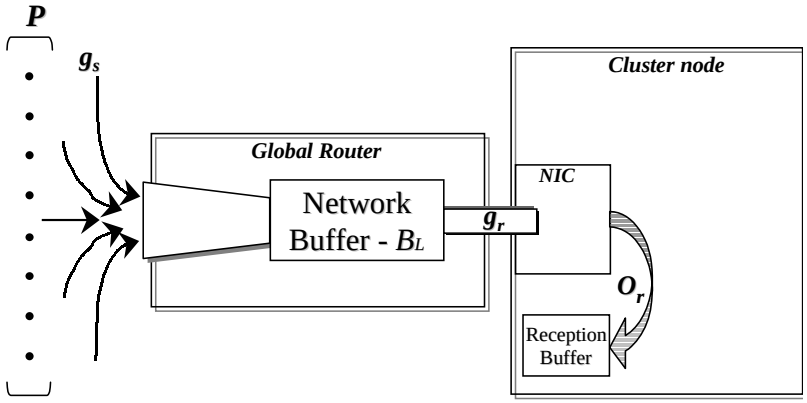


Figure A.3: A saturated inflow pipe

segment. First, we have to measure the expected execution time of this computation segment, as well as the round trip time for current message size. In our pseudocode, we just take the average of these measurements, but we can adopt stringent statistical test to get more accurate value. Then we run another pingpong-like test to obtain a set of O_r measurements. After the master node sends out one message, it immediately evaluates how long has the host processor been engaged in polling for message arrival. It records the maximum time spent in each run until it receives a message. To filter out noise measurements, we use our baseline measurements to select possible candidates. This pingpong-like process is running for many times until we obtain a sufficient large sample size. Then the collected dataset is processed by some statistical routine to extract the information we need.

A.3 Microbenchmark for the g_r Parameter

This microbenchmark measures the average inter-packet arrival time observed by a user-space process. If we can create and maintain a steady stream of packet inflow during the testing period, this measurement becomes a good approximation for the g_r parameter. For this microbenchmark, we simulate a saturated inflow pipe by directing a many-to-one data flow to the target node as shown in Figure A.3. After detecting that there are packet flooding in from the network, the target process arbitrary selects a time point and records the current time (t_0) and arrival count (C_0). Then it waits for another arbitrary time period, say δ unit, and records the current time (t_δ) and arrival count (C_δ) again. By dividing the the time gap $(t_\delta - t_0)$ against the difference between C_δ and C_0, we obtain an approximation for the g_r parameter. We see that the accuracy of this methodology improves by (1) running this test multiple times to obtain the correct statistical distribution, and (2) using a relatively long time duration (δ) to alleviate the measuring mistake.

The remaining issue of this microbenchmark falls on how can we obtain the packet

arrival count at the user-level. Here are some techniques that serves for our need:

1. Directly get the received count from the network interface card (NIC). Most of the high-end NICs provide performance counters for diagnostic and statistical purpose.

2. Monitor and report by the interrupt handler as it is responsible for handling incoming packets.

3. The above two methods need to have support either from the hardware or from the kernel. Otherwise, we can indirectly probe and count the number of arrivals by the user process itself. The following pseudocode provides a sketch on how to achieve this.

```
barrier ()
wait α * RTT time units
repeat {
    msg ⟵ async_receive (*)
} until (msg != NULL)
t0 ⟵ clock ()
i ⟵ 1
t1 ⟵ clock ()
while (t0 + δ > t1 ) {
    msg ⟵ async_receive (*)
    if (msg != NULL)
        inc i
    fi
    t1 ⟵ clock ()
}
```

A.4 Microbenchmark for the O_s and g_s Parameters

Modern network interfaces are operated in the form of queue structures, e.g. the transmission queue, receive queue, free-buffer queue, etc. These queues are simple circular FIFO queues, and operate under the consumer-producer relationship. Consider if the producer generates requests faster than the service rate of the consumer, outstanding requests will be queuing up in the queue; due to the limitation of the queuing space, eventually the system will become full. Upon the full condition, the system will only accept new request if and only if the consumer has serviced the "head" request to make way for the newcomer. Therefore, if there were unlimited supply of requests, the system gradually reaches an equilibrium state, in which the producer is forced to work at the same rate at that of the servicing rate. The algorithm used by this microbenchmark is shown in Algorithm 7.

Algorithm 7 The pseudocode for benchmarking the O_s & g_s parameters

```
INPUT
        message size x                          (1 ≤ x ≤ PDU)
        bandwidth BW
        Transmit ring size Tx

ALGORITHM
        msg_no ⟵ 8
        repeat {
                t0 ⟵ clock ()
                for i = 0 to msg_no do
                        do {
                                req ⟵ send_message (x)
                        } while (req != success)
                endfor
                t1 ⟵ clock ()
                sleep (msg_no * x ÷ BW)          # wait until all packets are served
                S ⟵ (t1 - t0) ÷ msg_no
                msg_no ⟵ (msg_no < Tx) ? (msg_no + 8) : (msg_no * 2)
                println (x, msg_no, S)
        } until (equilibrium rate is reached)
```

For the above microbenchmark to be effective, the prerequisite condition is $g_s > O_s$, without that we could only report of the O_s value. When the number of messages sent (*msg_no*) is less than queue size (*Tx*) of the transmit ring, the observed per message handling time becomes our O_s value for this message size. On the other hand, when the number of messages sent is greater than *Tx*, the observed equilibrium handling time becomes our g_s value for this message size.

A.5 Microbenchmark for the L parameter

This parameter encapsulates all the costs involved in moving the data across the network, thus, it becomes an impracticable task to measure this parameter directly. For example, the physical network may stretch over the building, and therefore, direct signal analysis on two remote ends becomes impossible. A feasible approach to measure this movement cost is by indirect estimation. In this subsection, we first describe the indirect calculation that yields the approximation for the *L* values between two arbitrary cluster nodes, then we will describe the microbenchmark that portrays the *L* parameter for the whole cluster network.

Recall that we can depict the total time for a *m*-byte packet to travel from the source node to its destination as

$$T_{ptp}(m) = O_s(m) + L(m) + O_r(m) + U_r(m) \tag{A.3}$$

If we carry out a simple pingpong test, the measured roundtrip time becomes $RTT(m) = 2 * T_{ptp}(m)$. Since all other parameters can be measured directly or

indirectly, we can therefore deduce the required L value by

$$L(m) = \frac{RTT(m)}{2} - O_s(m) - O_r(m) - U_r(m) \tag{A.4}$$

Thus, equation (A.4) becomes an indirect measurement for the L parameter between two machines over a range of packet sizes.

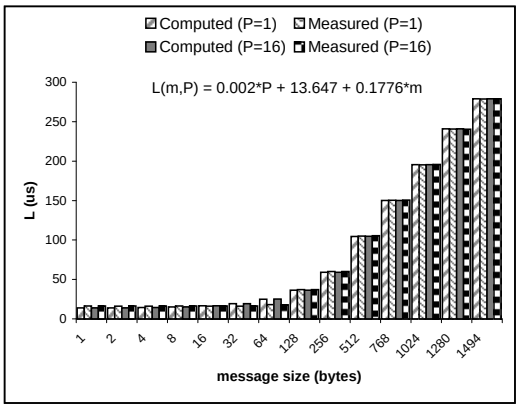
However, showing the L value between two machines of the cluster only delineates part of the cost, since the data are measured under a competition-free condition. To capture the network performance, our microbenchmark works up by generating multiple distinct pairs of pingpong nodes, with nodes of each distinct pair lie across the bisectional plane of the network. When all nodes start the pingpong tests at the same time, this emulates multiple concurrent packets flowing back-and-forth over the bisectional plane. Under this benchmark setting, by varying the number of pingpong pairs, we obtain different sets of L values. Then, the required bilinear function could be obtained by applying multiple regression technique on these datasets. To summarize, Algorithm 8 presents the pseudocode for this microbenchmark.

Algorithm 8 The concurrent pingpong microbenchmark for the L parameter

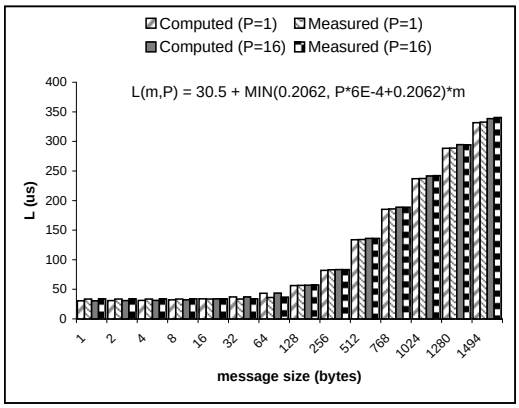
```
INPUT
        message size x                          (1 ≤ x ≤ PDU)
        number of nodes P

ALGORITHM
        myid ⟵ get_node_identity ()
        partner ⟵ pairing ()                    # identify the partner
        barrier ()                              # for all nodes
        for iter = 0 to ITERATION do
            if (left_side (myid)) then          # on the left side of the bisectional plane
                t0 ⟵ clock ()
                send_message (myid, partner, x)
                do {
                    msg ⟵ async_receive (partner)
                } while (msg == NULL)
                t1 ⟵ clock ()
                T[iter] ⟵ t1 - t0
            else
                do {
                    msg ⟵ async_receive (partner)
                } while (msg == NULL)
                send_message (myid, partner, x)
            fi
        endfor
        TIME[] ⟵ gather all timing vectors T[] (total P/2)
        L(x, P/2) ⟵ statistical_analysis (TIME[])
```

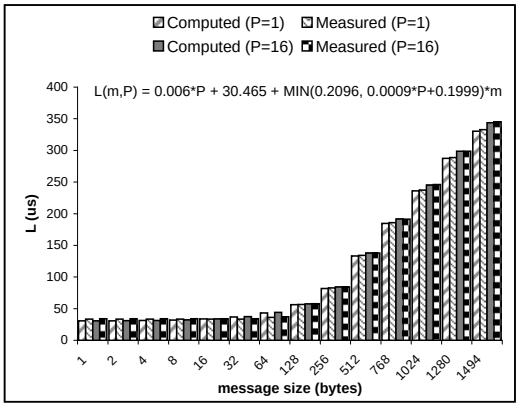
One of the merits of this microbenchmark is that it clearly differentiates between different network configurations. For examples, Figure A.4 shows the resulting parametric functions for three different network configurations that support a 32-node clus-



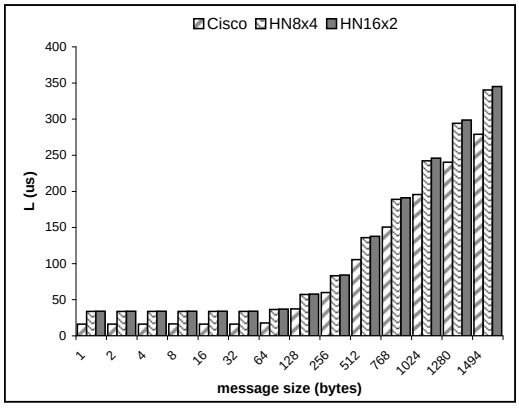
(a) (Cisco 2980G) - Comparison of calculated and measured L values on 2P=2 and 2P=32



(b) (HN 8x4) - Comparison of calculated and measured L values on 2P=2 and 2P=32



(c) (HN 16x2) - Comparison of calculated and measured L values on 2P=2 and 2P=32



(d) Comparison of three different configurations on a 32-node cluster

Figure A.4: The *L* parametric functions of three different network configurations for a 32-node cluster - (a) a single switch (Cisco Catalyst 2980G), (b) a hierarchical network (8x4) and (c) another hierarchical network (16x2)

ter. By exploring those equations, we observe that with current cluster size, the Cisco 2980G switch provides the best performance in terms of latency and scalability. In particular, we find that the bisectional bandwidth of the Cisco switch scales up linearly with the number of nodes, while the hierarchical networks have reached the upper bound when we scale up the cluster size.

Bibliography

[1] IEEE 802.3x. *Annex 31B MAC Control PAUSE operation, IEEE Std 802.3*, 2000 edition.

[2] Alhussein A. Abouzeid, Sumit Roy, and Murat Azizoglu. Stochastic modeling of TCP over lossy links. In *Proceedings of INFOCOM 2000*, pages 1724–1733. IEEE, 2000.

[3] Vikram S. Adve. *Analyzing the Behavior and Performance of Parallel Programs*. PhD thesis, Department of Computer Sciences, University of Wisconsin-Madison, December 1993.

[4] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 95–105, July 1995.

[5] 10 Gigabit Ethernet Alliance. http://www.10gea.org/index.htm.

[6] InfiniBand Trade Association. http://www.infinibandta.org/home.php3.

[7] Mark Baker and Rajkumar Buyya. *Cluster Computing at a Glance*, volume I of *High Performance Cluster Computing*, chapter 1, pages 246–269. Prentice Hall PTR, 1999.

[8] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message passing systems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, June 1992.

[9] Ammon Barak, Ilia Gilderman, and Igor Mctrik. Performance of the Communication Layers of TCP/IP with the Myrinet Gigabit LAN. *Computer Communication*, 22(11), Jul 1999.

[10] D. Bertsekas, C. Ozveren, G. Stamoulis, P. Tseng, and J. Tsitsiklis. Optimal communication algorithms for hypercubes. *Journal of Parallel and Distributed Computing*, 11:263–275, 1991.

[11] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice-Hall International, Inc., second edition, 1992.

[12] Raoul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.

[13] Raoul A.F. Bhoedjang. *Communication Architectures for Parallel-Programming Systems*. PhD thesis, Department of Computer Science, Vrije Universiteit, June 2000.

[14] G. Bilardi, K.T. Herley, A. Pietracaprina G. Pucci, and P. Spirakis. BSP vs LogP. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32, June 1996.

[15] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[16] S. Bokhari. Multiphase complete exchange: a theoretical analysis. *IEEE Transactions on Computers*, 45(2):220–229, February 1996.

[17] S. Bokhari. Multiphase complete exchange on paragon, sp2, and cs-2. *IEEE Parallel and Distributed Technology*, 4(3):45–59, Fall 1996.

[18] S.H. Bokhari and D.M. Nicol. Balancing contention and synchronization on the Intel Paragon. *IEEE Concurrency*, 5(2):74–83, 1997.

[19] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

[20] C. Chiola and G. Ciaccio. *Lightweight Messaging Systems*, volume I of *High Performance Cluster Computing*, chapter 10, pages 246–269. Prentice Hall PTR, 1999.

[21] G. Chiola and G. Ciaccio. Gamma: a low-cost network of workstations based on active messages. In *Proceedings of the 5th EUROMICRO workshop on Parallel and Distributed Processing PDP'97*, January 1997.

[22] G. Chiola, G. Ciaccio, L. V. Mancini, and P. Rotondo. Gamma on dec 2114x with efficient flow control. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.

[23] Brent N. Chun, Alan M. Mainwaring, and David E. Culler. Virtual Network Transport Protocols for Myrinet. *IEEE Micro*, 18(1):53–63, 1998.

[24] David D. Clark, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[25] The Asgard Cluster. http://www.asgard.ethz.ch/.

[26] The Biopendium Cluster. http://www.inpharmatica.co.uk/biopdetail.htm.

[27] The CLiC Cluster. http://www.tu-chemnitz.de/urz/anwendungen/CLIC/.

[28] Cluster@TOP500. http://clusters.top500.org/.

[29] Mark Edward Crovella. *Performance Prediction and Tuning of Parallel Programs*. PhD thesis, Department of Computer Science, The College Arts and Sciences, University of Rochester, 1994.

[30] 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. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[31] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad O. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[32] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[33] V. Dimakopoulos and N. Dimopoulos. A theory for total exchange in multidimensional interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):639–649, July 1998.

[34] S. Donaldson, J. Hill, and D. Skillicorn. Exploiting Global Structure for Performance on Clusters. In *Proceedings of IPPS/SPDP'99*, pages 176–182, 1999.

[35] Jose Duato, Sudhakar Yalmanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach*. IEEE Computer Society, 1997.

[36] Sally Floyd and Van Jacobson. Traffic phase effects in packet-switched gateways. *Journal of Internetworking:Practice and Experience*, 3(3):115–156, Sept. 1992.

[37] Steven Fortune and James Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[38] Massing Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[39] Giganet. http://wwwip.emulex.com/ip/index.html/.

[40] J. Gross and J. Yellen. *Graph Theory and its Applications*. CRC press, 1999.

[41] S. Hambrusch and A. Khokhar. An architecture-independent model for coarse-grained parallel machines. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, pages 544–551, Oct 1994.

[42] Mathias Hein and David Griffiths. *Switching technology in the local network : from LAN to switched LAN to virtual LAN*. International Thomson Computer Press, 1997.

[43] T. Heywood and C. Leopold. Models of parallelism. In J.R. Davy and P.M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*. Oxford Univ. Press, 1995.

[44] Michael G. Hluchyj and Mark J. Karol. Queueing in High-Performance Packet Switching. *IEEE Journal on Selected Areas in Communications*, 6(9):1587–1597, December 1988.

[45] R. W. Hockney. *The Science of Computer Benchmarking*. SIAM Publications, 1996.

[46] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing*. McGraw-Hill, 1998.

[47] G. Iannello. Efficient Algorithms for the Reduce-Scatter Operation in LogGP. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):970–982, 1997.

[48] Intel. PCI - Efficient Use. Web document, Apr 1997. http://support.intel.com/support/chipsets/pc1001.htm.

[49] Intel. Balanced Server Platform Design. Presentation material, 1998. http://developer.intel.com/software/asc/documents/19.IDF-ESGS4.pdf.

[50] Dean L. Isaacson and Richard W. Madsen. *Markov Chains: Theory and Applications*. John Wiley & Sons, 1976.

[51] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM 88*, pages 314–329, 1988.

[52] Wagner Meira Junior. *Understanding Parallel Program Performance Using Cause-Effect Analysis*. PhD thesis, Department of Computer Science, The College Arts and Sciences, University of Rochester, 1997.

[53] R. Karp, A. Sahay, E. Santos, and K. Schauser. Optimal broadcast and summation in the logp model. In *Proceedings of Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 142–153, June 1993.

[54] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of ACM SIGCOMM 93*, pages 259–268, 1993.

[55] K. Keeton, T. Anderson, and D. Patterson. Logp quantified: The case for low-overhead local area networks. In *Hot Interconnects III: A Symposium on High Performance Interconnects*, Aug. 1995.

[56] T. V. Lakshman and Upamanyu Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking*, 5(3):336–350, June 1997.

[57] C. Lam, C. Huang, and P. Sadayappan. Optimal Algorithms for All-to-all Personalized Communication on Rings and Two Dimensional Tori. *Journal of Parallel and Distributed Computing*, 43:3–13, 1997.

[58] M. Lauria, S. Pakin, and A. Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of the 7th High Performance Distributed Computing Conference (HPDC7)*, July 1998.

[59] C.M. Lee, A.T.C. Tam, and C.L. Wang. Directed point: An efficient communication subsystem for cluster computing. In *International Conference on Parallel and Distributed Computing Systems (IASTED)*, Oct. 1998.

[60] Tsern-Huei Lee. The throughput efficiency of go-back-n arq scheme for burst-error channels. In *Proceedings of INFOCOM'91*, pages 773–780. IEEE, 1991.

[61] C. H. C. Leung, Y. Kikumoto, and S. A. Sorensen. The throughput efficiency of the go-back-n arq scheme under markov and related error structures. *IEEE Transactions on Communications*, 36(2):231–234, February 1988.

[62] P. Marenzoni, Giovanni Rimassa, Massimo Bertozzi, Gianni Conte, and P. Rossi. An operating system support to low-overhead communications in NOW clusters. In *Proceedings of Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC97)*, pages 130–143, February 1997.

[63] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion aviodance algorithm. *Computer Communication Review*, 27(3), July 1997.

[64] R. McClellan. Evaluating expected network performance based on multilayer switch performance data. Technical report, McClellan Consulting, August 2000.

[65] W. McColl. BSP programming. In *Proceedings of DIMACS Workshop on Specification of Parallel Algorithms*, pages 25–35, May 1994.

[66] W. McColl. The bsp approach to architecture independent parallel programming. Technical report, Oxford University Computing Laboratory, Dec. 1995.

[67] Csaba Andras Moritz and Matthew Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. In *Measurement and Modeling of Computer Systems*, pages 254–263, 1998.

[68] MPICH. MPICH-A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/.

[69] Myricom. http://www.myri.com/.

[70] J. Nash, P. Dew, and M. Dyer. Scalable and Portable Computing Using the WPRAM Model. In M. Kara, J. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 47–62. IOS Press, April 1996.

[71] Extreme Networks. http://www.extremenetworks.com/products/.

[72] W. Noureddine and F. Tobagi. Selective back-pressure in switched Ethernet lans. In *Proceedings of the Globecom'99*, 1999.

[73] Nupairoj, Ni, Park, and Choi. Architecture-Dependent Tuning of the Parameterized Communication Model for Optimal Multicasting. In *IPPS: 11th International Parallel Processing Symposium*, pages 578–582. IEEE Computer Society Press, 1997.

[74] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling tcp throughput: A simple model and its empirical validation. In *Proceedings of ACM SIGCOMM*, Sept. 1998.

[75] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, 1997.

[76] J.-Y.L. Park, H.-A. Choi, N. Nupairoj, and L.M. Ni. Construction of Optimal Multicast Trees Based on the Parameterized Communication Model. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages 180–187, Aug 1996.

[77] K. Park. Warp Control: a Dynamically Stable Congestion Protocol and Its Analysis. *Journal of High Speed Networks*, 2(4):373–404, 1993.

[78] PCI-X. The PCI Special Interest Group (SIG) . http://www.pcisig.com/.

[79] Stefan Petri, Gunther Lustig, Andreas Döring, and Rainer Hagenau. The impact of switch technologies on high performance cluster communication. Technical report, Institut für Technische Informatik, Medizinische Universität zu Lübeck, Germany, 2000.

[80] Gregory F. Pfister. *In Search Of Clusters*. Prentice Hall, 1995.

[81] R. Ponnusamy, R. Thakur, A. Choudhary, and G. Fox. Scheduling regular and irregualar patterns on the CM-5. In *Proceedings of Supercomputing '92*, pages 394–402, November 1992.

[82] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *PC-NOW Workshop, IPPS/SPDP98*, March 1998.

[83] Loic Prylli and Bernard Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. In *IPPS/SPDP Workshops*, pages 472–485, 1998.

[84] L. Qiu, Y. Zhang, and S. Keshav. On individual and aggregate TCP performance. In *Proceedings of the Seventh Annual International Conference on Network Protocols*, pages 203–212. IEEE, Nov. 1999.

[85] RFC793. Transmission control protocol - protocol specification. Internet Archives, Sept 1981.

[86] J.L. Roda, F. Sande, C. León, J.A. González, and C. Rodriguez. The Collective Computing Model. In *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing PDP'99*, pages 19–26, Feb 1999.

[87] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality-of-Service for Message Passing Programs. In *Proceedings of the IEEE/ACM SC2000 Conference*, 2000.

[88] S. Shibusawa, H. Makino, S. Nimiya, and J. Hatta. Scatter and Gather Operations on an Asynchronous Communication Model. In *ACM Symposium on Applied Computing*, Mar 2000.

[89] M. Sidi, W. Z. Liu, I. Cidon, and I. Gopal. Congestion control through input rate regulation. *IEEE Transactions on Communications*, 41(3):471–477, 1993.

[90] D. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, Jun 1998.

[91] John D. Spragins, Joseph L. Hammond, and Krzysztof Pawlikowski. *Telecommunications - Protocols and Design*. Addison Wesley, 1992.

[92] William Stallings. *Data & Computer Communications*. Prentice Hall, sixth edition, 2000.

[93] W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Internet Archives RFC 2001, January 1997.

[94] Y.J. Suh and S. Yalamanchili. All-to-all communication with minimum start-up costs in 2D/3D tori and meshes. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):442–458, 1998.

[95] S. Sumimoto, A. Hori, H. Tezuka, H. Harada, T. Takahashi, and Y. Ishikawa. GigaE PM II: Design of High Performance Communication Library using Gigabit Ethernet, 1999. http://pdswww.rwcp.or.jp/db/paper-J/1999/swopp99/sumi/sumi.html.

[96] Shinji Sumimoto, Hiroshi Tezuka, Atsushi Hori, Hiroshi Harada, Toshiyuki Takahashi, and Yutaka Ishikawa. The design and evaluation of high performance communication using a gigabit Ethernet. In *International Conference on Supercomputing '99*, pages 243–250. ACM SIGARCH, June 1999.

[97] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency, Practice and Experience*, 2:315–340, 1990.

[98] Cisco Systems. http://www.cisco.com/warp/public/44/jump/switches.shtml.

[99] Cisco Systems. Catalyst 2980G/2980G-A Enterprise Desktop Switches. Data Sheet, 2001.

[100] Anthony T.C. Tam and Cho-Li Wang. Realistic Communication Model for Parallel Computing on Cluster. In *Proceedings of the 1st IEEE International Workshop on Cluster Computing (IWCC'99)*, December 1999.

[101] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: A operating system coordinated high performance communication library. In *High-Performance Computing and Networking '97*, 1997.

[102] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *IPPS/SPDP'98*, pages 308–315, 1998.

[103] Takayoshi Touyama and Susumu Horiguchi. Performance Evaluation of Practical Parallel Computation Model LogPQ. In *Proceedings of the 4th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, pages 216–221, 1999.

[104] Yu-Chee Tseng and Sandeep K. S. Gupta. All-to-all personalized communication in a wormhole-routed torus. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):498–505, 1996.

[105] L. Valliant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[106] M. Verma and T. Chiueh. Pupa: A low-latency communication system for fast Ethernet. Technical report, Computer Science Department, State University of New York at Stony Brook, Apr. 1998.

[107] VIA. http://www.viarch.org/.

[108] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[109] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*. ACM Press, 1992.

[110] J. Walrand and P. Varaiya. *High-Performance Communication Networks*. Morgan Kaufmann Publishers, 1st edition, 1996.

[111] C.L. Wang, A. Tam, B. Cheung, W.Z Zhu, and C.M. Lee. Directed point: High performance communication subsystem for gigabit networking in clusters. *Journal of Future Generation Computer Systems*, 18(4), 2001.

[112] Jonathan L. Wang. Impact of self-similarity on the Go-Back-N ARQ protocols. In *Proceedings of Fourth International Conference on Computer Communications and Networks*, pages 250–257, Sept 1995.

[113] T.M. Warschko, J.M. Blum, and W.F. Tichy. A reliable transmission protocol for Myrinet. In *Proceedings of the 2nd Workshop on Cluster-Computing*, March. 1999.

[114] M. Welsh, A. Basu, and T. Von Eicken. Low-latency communication over Fast Ethernet. In *Lecture Notes in Computer Science*, volume 1123, 1996.

[115] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Hot Interconnects V*, Aug 1997.

[116] Xingfu Wu. *Performance Evaluation, Prediction and Visualization of Parallel Systems*. Kluwer Academic Publishers, 1999.

[117] Zhiwei Xu and Kai Hwang. Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2. *IEEE Parallel and Distributed Technology*, 4(1):9–23, 1996.

[118] Zhiwei Xu and Kai Hwang. MPPs and Clusters for Scalable Computing. In *Proceedings of 2nd International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 117–123, 1996.

[119] M. Yajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and modeling of the temporal dependence in packet loss. In *Proceedings of IEEE INFOCOM (1999)*, pages 345–352, 1999.

[120] C-Q Yang and A.V.S. Reddy. A Taxonomy for Congestion Control Algorithms in Packet Switching Networks. *IEEE Network*, 9(4):34–45, 1995.

[121] Y. Yang and J. Wang. Optimal all-to-all personalized exchange in self-routable multistage networks. *IEEE Transactions on Parallel and Distributed Systems*, 11(3):261–274, 2000.

[122] Lixia Zhang, Scott Schenker, and David Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two Way Traffic. *ACM Computer Communications Review*, 1991.