Communication Architectures for Parallel-Programming Systems

Raoul A.F. Bhoedjang

Advanced School for Computing and Imaging

This work was carried out in graduate school ASCI.
ASCI dissertation series number 52.

Copyright © 2000 Raoul A.F. Bhoedjang.

VRIJE UNIVERSITEIT

Communication Architectures for Parallel-Programming Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der exacte wetenschappen/
wiskunde en informatica
op dinsdag 6 juni 2000 om 15.45 uur
in het hoofdgebouw van de universiteit,
De Boelelaan 1105

door

Raoul Angelo Félicien Bhoedjang

geboren te 's-Gravenhage

Promotoren: prof.dr.ir. H.E. Bal
prof.dr. A.S. Tanenbaum

Preface

Personal Acknowledgements

First and foremost, I am grateful to my thesis supervisors, Henri Bal and Andy Tanenbaum, for creating an outstanding research environment. I thank them both for their confidence in me and my work.

I am very grateful to all members, past and present, of Henri's Orca team: Saniya Ben Hassen, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, Koen Langendoen, Jason Maassen, Rob van Nieuwpoort, Aske Plaat, John Romein, Tim Rühl, Ronald Veldema, Kees Verstoep, and Greg Wilson.

Henri does a great job running this team and working with him has been a great pleasure. His ability to get a job done, his punctuality, and his infinite reading capacity are greatly appreciated — by me, but by many others as well. Above all, Henri is a very nice person.

Tim Rühl has been a wonderful friend and colleague throughout my years at the VU and I owe him greatly. We have spent countless hours hacking away at various projects, meanwhile discussing research and personal matters. Only a few of our pet projects made it into papers, but the papers that we worked on together are the best that I have (co-)authored.

I would like to thank John Romein for his work on Multigame (see Chapter 8), for teaching me about parallel game-tree searching, for his expert advice on computer hardware, for keeping the DAS cluster in good shape, and for generously donating peanuts when I needed them.

Koen Langendoen is the living proof of the rather disturbing observation that there is no significant correlation between a programmer's coding style and the correctness of his code. Koen is a great hacker and taught me a thing or two about persistence. I want to thank him for his work on Orca, Panda, a few other projects, and for his work as a member of my thesis committee.

The Orca group is blessed with the support of three highly competent research programmers: Rutger Hofman, Ceriel Jacobs, and Kees Verstoep. All three have contributed directly to the work described in this thesis.

I wish to thank Rutger for his work on Panda 4.0, for reviewing papers, for

many useful discussions about communication protocols, and for lots of good espresso. I also apologize for all the boring microbenchmarks he performed so willingly.

There are computer programmers and there are artists. Ceriel, no doubt, belongs to the artists. I want to thank him for his fine work on Orca and Panda and for reviewing several of my papers.

Kees's work has been instrumental in completing the comparison of LFC implementations described in Chapter 8. During my typing diet, he performed a tremendous amount of work on these implementations. Not only did he write three of the five LFC implementations described in this thesis, but he also put many hours into measuring and analyzing the performance of various applications. Meanwhile, he debugged Myrinet hardware and kept the DAS cluster alive. Finally, Kees reviewed some of my papers and provided many useful comments on this thesis. It has been a great pleasure to work with him.

With a 128-node cluster of computers in your backyard, it is easy to forget about your desktop machine ... until you cannot reach your home directory. Fortunately, such moments have been scarce. Many thanks to the department's system administrators for keeping everything up and running. I am especially grateful to Ruud Wiggers, for quota of various kinds and great service.

I wish to thank Matty Huntjens for starting my graduate career by recommending me to Henri Bal. I greatly appreciate the way he works with the department's undergraduate students; the department is lucky to have such people.

I ate many dinners at the University's 'restaurant.' I will refrain from commenting on the food, but I did enjoy the company of Rutger Hofman, Philip Homburg, Ceriel Jacobs, and others.

My thesis committee — prof. dr. Zwaenepoel, prof. dr. Kaashoek, dr. Langendoen, dr. Epema, and dr. van Steen — went through some hard times; my move to Cornell rather reduced my pages-per-day rate. I would like to thank them all for reading my thesis and for their suggestions for improvement. I am particularly grateful to Frans Kaashoek whose comments and suggestions considerably improved the quality of this thesis.

I would also like to thank Frans for his hospitality during some of my visits to the US. I especially enjoyed my two-month visit to his research group at MIT. On that same occasion, Robbert van Renesse invited me to Cornell, my current employer. He too, has been most hospitable on many occasions, and I would like to thank him for it.

The Netherlands Organization for Scientific Research (N.W.O.) supported part of my work financially through Henri Bal's Pionier grant.

Many of the *important* things I know, my parents taught me. I am sure my father would have loved to attend my thesis defense. My mother has been a great example of strength in difficult times.

Finally, I wish to thank Daphne for sharing her life with me. Most statements concerning two theses in one household are true. I am glad we survived.

Per-Chapter Contributions

The following paragraphs summarize the contributions made by others to the papers and software that each chapter of this thesis is based on. I would like to thank all for the time and effort they have invested.

Chapter 2's survey is based on an article co-authored by Tim Rühl and Henri Bal [18].

Tim Rühl and I designed and implemented the LFC communication system described in Chapters 3, 4, and 5. John Romein ported LFC to Linux. LFC is described in two papers co-authored by Tim Rühl and Henri Bal; it is also one of the systems classified in the survey mentioned above.

Panda 4.0, described in Chapter 6, was designed by Rutger Hofman, Tim Rühl, and me. Rutger Hofman implemented Panda 4.0 on LFC from scratch, except for the threads package, OpenThreads, which was developed by Koen Langendoen. Earlier versions of Panda are described in two conference publications [19, 124]. The comparison of upcall models is based on an article by Koen Langendoen, me, and Henri Bal [88]. Panda's dynamic switching between polling and interrupts is described in a conference paper by Koen Langendoen, John Romein, me, and Henri Bal [89].

Chapter 7 discusses the implementation of four parallel-programming systems: Orca, Manta, CRL, and MPICH. Orca and Manta were both developed in the computer systems group at the Vrije Universiteit; CRL and MPICH were developed elsewhere.

Koen Langendoen and I implemented the continuation-based version of the Orca runtime system on top of Panda 2.0. The use of continuations in the runtime system is described in a paper co-authored by Koen Langendoen [14]. Ceriel Jacobs wrote the Orca compiler and ported the runtime system to Panda 3.0. Rutger Hofman ported the continuation-based runtime system to Panda 4.0. An extensive description and performance evaluation of the Orca system on Panda 3.0 are given in a journal paper by Henri Bal, me, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and Frans Kaashoek [9].

Manta was developed by Jason Maassen, Rob van Nieuwpoort, and Ronald Veldema. Their work is described in a conference paper [99]. Rob helped me with the Manta performance measurements presented in Section 7.2.

CRL was developed at MIT by Kirk Johnson [73, 74]; I ported CRL to LFC. It could not have been much easier, because CRL is a really nice piece of software.

MPICH [61] is being developed jointly by Argonne National Laboratories and

Mississippi State University. Rutger Hofman ported MPICH to Panda 4.0.

The comparison of different LFC implementations described in Chapter 8 was done in cooperation with Kees Verstoep, Tim Rühl, and Rutger Hofman. Tim and I implemented the $N_{rx}I_{mc}$ and $H_{rx}H_{mc}$ versions of LFC. Kees implemented $I_{rx}I_{mc}$, $I_{rx}H_{mc}$, and $N_{rx}H_{mc}$ and performed many of the measurements presented in Chapter 8. Rutger helped out with various Panda-related issues.

The parallel applications analyzed in Chapter 8 were developed by many persons. The Puzzle application and the parallel-programming system that it runs on, Multigame [122], were written by John Romein. Awari was developed by Victor Allis, Henri Bal, Hans Staalman, and Elwin de Waal. The linear equation solver (LEQ) was developed by Koen Langendoen. The MPI applications were written by Rutger Hofman (QR and ASP) and Thilo Kielmann (SOR). The CRL applications Barnes-Hut (Barnes), Fast Fourier Transform (FFT), and radix sort (radix) are all based on shared-memory programs from the SPLASH-2 suite [153]. Barnes-Hut was adapted for CRL by Kirk Johnson, FFT by Aske Plaat, and radix by me.

Contents

1 Introduction **1**
- 1.1 Parallel-Programming Systems 2
- 1.2 Communication architectures for parallel-programming systems 5
- 1.3 Problems 6
  - 1.3.1 Data-Transfer Mismatches 7
  - 1.3.2 Control-Transfer Mismatches 8
  - 1.3.3 Design Alternatives for User-Level Communication Architectures 9
- 1.4 Contributions 9
- 1.5 Implementation 10
  - 1.5.1 LFC Implementations 10
  - 1.5.2 Panda 12
  - 1.5.3 Parallel-Programming Systems 12
- 1.6 Experimental Environment 14
  - 1.6.1 Cluster Position 14
  - 1.6.2 Hardware Details 14
- 1.7 Thesis Outline 17

2 Network Interface Protocols **19**
- 2.1 A Basic Network Interface Protocol 20
- 2.2 Data Transfers 23
  - 2.2.1 From Host to Network Interface 23
  - 2.2.2 From Network Interface to Network Interface 25
  - 2.2.3 From Network Interface to Host 26
- 2.3 Address Translation 26
- 2.4 Protection 29
- 2.5 Control Transfers 30
- 2.6 Reliability 31
- 2.7 Multicast 33
- 2.8 Classification of User-Level Communication Systems 34
- 2.9 Summary 36

3 The LFC User-Level Communication System 37
3.1 Programming Interface 37
3.1.1 Addressing 38
3.1.2 Packets 38
3.1.3 Sending Packets 40
3.1.4 Receiving Packets 41
3.1.5 Synchronization 42
3.1.6 Statistics 43
3.2 Key Implementation Assumptions 43
3.3 Implementation Overview 44
3.3.1 Myrinet 44
3.3.2 Operating System Extensions 44
3.3.3 Library 46
3.3.4 NI Control Program 46
3.4 Packet Anatomy 48
3.5 Data Transfer 50
3.6 Host Receive Buffer Management 53
3.7 Message Detection and Handler Invocation 54
3.8 Fetch-and-Add 56
3.9 Limitations 56
3.9.1 Functional Limitations 56
3.9.2 Security Violations 58
3.10 Related Work 59
3.11 Summary 60

4 Core Protocols for Intelligent Network Interfaces 61
4.1 UCAST: Reliable Point-to-Point Communication 62
4.1.1 Protocol Data Structures 62
4.1.2 Sender-Side Protocol 64
4.1.3 Receiver-Side Protocol 65
4.2 MCAST: Reliable Multicast Communication 67
4.2.1 Multicast Forwarding 67
4.2.2 Multicast Tree Topology 68
4.2.3 Protocol Data Structures 68
4.2.4 Sender-Side Protocol 69
4.2.5 Receiver-Side Protocol 70
4.2.6 Deadlock 71
4.3 RECOV: Deadlock Detection and Recovery 73
4.3.1 Deadlock Detection 73
4.3.2 Deadlock Recovery 74
4.3.3 Protocol Data Structures 75

4.3.4 Sender-Side Protocol 77
4.3.5 Receiver-Side Protocol 78
4.4 INTR: Control Transfer 82
4.5 Related Work 83
4.5.1 Reliability 83
4.5.2 Interrupt Management 84
4.5.3 Multicast 85
4.5.4 FM/MC 86
4.6 Summary 88

5 The Performance of LFC 89
5.1 Unicast Performance 89
5.1.1 Latency and Throughput 89
5.1.2 LogGP Parameters 91
5.1.3 Interrupt Overhead 92
5.2 Fetch-and-Add Performance 93
5.3 Multicast Performance 94
5.3.1 Performance of the Basic Multicast Protocol 94
5.3.2 Impact of Deadlock Recovery and Multicast Tree Shape 98
5.4 Summary 101

6 Panda 103
6.1 Overview 104
6.1.1 Functionality 104
6.1.2 Structure 106
6.1.3 Panda's Main Abstractions 109
6.2 Integrating Multithreading and Communication 113
6.2.1 Multithread-Safe Access to LFC 113
6.2.2 Transparently Switching between Interrupts and Polling 115
6.2.3 An Efficient Upcall Implementation 120
6.3 Stream Messages 130
6.4 Totally-Ordered Broadcasting 133
6.5 Performance 133
6.5.1 Performance of the Message-Passing Module 134
6.5.2 Performance of the Broadcast Module 136
6.5.3 Other Performance Issues 137
6.6 Related Work 138
6.6.1 Portable Message-Passing Libraries 138
6.6.2 Polling and Interrupts 139
6.6.3 Upcall Models 140
6.6.4 Stream Messages 141

6.6.5 Totally-Ordered Broadcast 141
6.7 Summary 142

7 Parallel-Programming Systems 145
7.1 Orca 145
7.1.1 Programming Model 146
7.1.2 Implementation Overview 148
7.1.3 Efficient Operation Transfer 151
7.1.4 Efficient Implementation of Guarded Operations 155
7.1.5 Performance 158
7.2 Manta 162
7.2.1 Programming Model 163
7.2.2 Implementation 163
7.2.3 Performance 165
7.3 The C Region Library 166
7.3.1 Programming Model 166
7.3.2 Implementation 167
7.3.3 Performance 170
7.4 MPI — The Message Passing Interface 171
7.4.1 Implementation 171
7.4.2 Performance 172
7.5 Related Work 175
7.5.1 Operation Transfer 175
7.5.2 Operation Execution 176
7.6 Summary 177

8 Multilevel Performance Evaluation 181
8.1 Implementation Overview 182
8.1.1 The Implementations 182
8.1.2 Commonalities 183
8.2 Reliable Point-to-Point Communication 186
8.2.1 The No-Retransmission Protocol (N_{rx}) 186
8.2.2 The Retransmitting Protocols (H_{rx} and I_{rx}) 186
8.2.3 Latency Measurements 187
8.2.4 Window Size and Receive Buffer Space 188
8.2.5 Send Buffer Space 190
8.2.6 Protocol Comparison 193
8.3 Reliable Multicast 193
8.3.1 Host-Level versus Interface-Level Packet Forwarding 194
8.3.2 Acknowledgement Schemes 195
8.3.3 Deadlock Issues 195

8.3.4 Latency and Throughput Measurements 196
8.4 Parallel-Programming Systems 197
8.4.1 Communication Style 198
8.4.2 Performance Issues 200
8.5 Application Performance 201
8.5.1 Performance Results 201
8.5.2 All-pairs Shortest Paths (ASP) 204
8.5.3 Awari 204
8.5.4 Barnes-Hut 206
8.5.5 Fast Fourier Transform (FFT) 207
8.5.6 The Linear Equation Solver (LEQ) 207
8.5.7 Puzzle 208
8.5.8 QR Factorization 208
8.5.9 Radix Sort 209
8.5.10 Successive Overrelaxation (SOR) 209
8.6 Classification 210
8.7 Related Work 211
8.7.1 Reliability 211
8.7.2 Multicast 213
8.7.3 NI Protocol Studies 213
8.8 Summary 214

9 Summary and Conclusions 217
9.1 LFC and NI Protocols 217
9.2 Panda 219
9.3 Parallel-Programming Systems 220
9.4 Performance Impact of Design Decisions 221
9.5 Conclusions 222
9.6 Limitations and Open Issues 223

A Deadlock issues 241
A.1 Deadlock-Free Multicasting in LFC 241
A.2 Overlapping Multicast Groups 243

B Abbreviations 245

Samenvatting 247

Curriculum Vitae 253

List of Figures

1.1 Communication layers studied in this thesis. 6
1.2 Structure of this thesis. 11
1.3 Myrinet switch topology. 15
1.4 Switches and cluster nodes. 15
1.5 Cluster node architecture. 16

2.1 Operation of the basic protocol. 20
2.2 Host-to-NI throughput using different data transfer mechanisms. 24
2.3 Design decisions for reliability. 31
2.4 Repeated send versus a tree-based multicast. 33

3.1 LFC's packet format. 48
3.2 LFC's data transfer path. 50
3.3 LFC's send path and data structures. 51
3.4 Two ways to transfer the payload and status flag of a network packet. 52
3.5 LFC's receive data structures. 53

4.1 Protocol data structures for UCAST. 63
4.2 Sender side of the UCAST protocol. 64
4.3 Receiver side of the UCAST protocol. 66
4.4 Host-level and interface-level multicast forwarding. 67
4.5 Protocol data structures for MCAST. 69
4.6 Sender-side protocol for MCAST. 70
4.7 Receive procedure for the multicast protocol. 71
4.8 MCAST deadlock scenario. 72
4.9 A binary tree. 72
4.10 Subtree covered by the deadlock recovery algorithm. 75
4.11 Deadlock recovery data structures. 76
4.12 Multicast forwarding in the deadlock recovery protocol. 77
4.13 Packet transmission in the deadlock recovery protocol. 79
4.14 Receive procedure for the deadlock recovery protocol. 80
4.15 Packet release procedure for the deadlock recovery protocol. 81

4.16 Timeout handling in INTR. 84
4.17 LFC's and FM/MC's buffering strategies. 86

5.1 LFC's unicast latency. 90
5.2 LFC's unicast throughput. 91
5.3 Fetch-and-add latencies under contention. 94
5.4 LFC's broadcast latency. 95
5.5 LFC's broadcast throughput. 96
5.6 Tree used to determine multicast gap. 97
5.7 LFC's broadcast gap. 97
5.8 Impact of deadlock recovery on broadcast throughput. 98
5.9 Impact of tree topology on broadcast throughput. 100
5.10 Impact of deadlock recovery on all-to-all throughput. 100

6.1 Different broadcast delivery orders. 105
6.2 Panda modules: functionality and dependencies. 107
6.3 Two Panda header stacks. 111
6.4 Recursive invocation of an LFC routine. 114
6.5 RPC example. 118
6.6 Simple remote read with active messages. 121
6.7 Message handler with locking. 124
6.8 Three different upcall models. 125
6.9 Message handler with blocking. 126
6.10 Fast thread switch to an upcall thread. 130
6.11 Application-to-application streaming with Panda's stream messages. 131
6.12 Panda's message-header formats. 131
6.13 Receiver-side organization of Panda's stream messages. 132
6.14 Panda's message-passing latency. 134
6.15 Panda's message-passing throughput. 135
6.16 Panda's broadcast latency. 136
6.17 Panda's broadcast throughput. 137

7.1 Definition of an Orca object type. 147
7.2 Orca process creation. 147
7.3 The structure of the Orca shared object system. 148
7.4 Decision process for executing an Orca operation. 150
7.5 Orca definition of an array of records. 151
7.6 Specialized marshaling code generated by the Orca compiler. 153
7.7 Data transfer paths in Orca. 154
7.8 Orca with popup threads. 155
7.9 Orca with single-threaded upcalls. 157

7.10 The continuations interface. 157
7.11 Roundtrip latencies for Orca, Panda, and LFC. 159
7.12 Roundtrip throughput for Orca, Panda, and LFC. 160
7.13 Broadcast latency for Orca, Panda, and LFC. 161
7.14 Broadcast throughput for Orca, Panda, and LFC. 161
7.15 Roundtrip latencies for Manta, Orca, Panda, and LFC. 165
7.16 Roundtrip throughput for Manta, Orca, Panda, and LFC. 166
7.17 CRL read and write miss transactions. 168
7.18 Write miss latency. 170
7.19 Write miss throughput. 171
7.20 MPI unicast latency. 173
7.21 MPI unicast throughput. 174
7.22 MPI broadcast latency. 174
7.23 MPI broadcast throughput. 175

8.1 LFC unicast latency. 188
8.2 Peak throughput for different window sizes. 189
8.3 LFC unicast throughput. 191
8.4 Host-level and interface-level multicast forwarding. 194
8.5 LFC multicast latency. 196
8.6 LFC multicast throughput. 197
8.7 Application-level impact of efficient broadcast support. 199
8.8 Data and packet rates of $N_{rx}I_{mc}$ on 64 nodes. 203
8.9 Normalized application execution times. 205
8.10 Classification of communication systems for Myrinet. 212

A.1 A biomial spanning tree. 243
A.2 Multicast trees for overlapping multicast groups. 244

List of Tables

1.1 Classification of parallel-programming systems. 4
1.2 Communication characteristics of parallel-programming systems. 13

2.1 Throughputs on a Pentium Pro/Myrinet cluster. 27
2.2 Summary of design choices in existing communication systems. 35

3.1 LFC's user interfac. 39
3.2 Resources used by the NI control program. 46
3.3 LFC's packet tags. 49

5.1 Values of the LogGP parameters for LFC. 93
5.2 Deadlock recovery statistics. 101

6.1 Send and receive interfaces of Panda's system module. 112
6.2 Classification of upcall models. 121

7.1 A comparison of Orca and Manta. 162
7.2 PPS performance summary 179

8.1 Five versions of LFC's reliability and multicast protocols. 183
8.2 Division of work in the LFC implementations. 184
8.3 Parameters used by the protocols. 185
8.4 LogP parameter values (in microseconds) for a 16-byte message. 188
8.5 Fetch-and-add latencies. 189
8.6 Application characteristics and timings. 202
8.7 Classification of applications. 210

Chapter 1

Introduction

This thesis is about *communication support for parallel-programming systems*. A parallel-programming system (PPS) is a system that aids programmers who wish to exploit multiple processors to solve a single, computationally intensive problem. PPSs come in many flavors, depending on the programming paradigm they provide and the type of hardware they run on. This thesis concentrates on PPSs for *clusters* of computers. By a cluster we mean a collection of off-the-shelf computers interconnected by an off-the-shelf network. Clusters are easy to build, easy to extend, scale to large numbers of processors, and are relatively inexpensive.

The performance of a PPS depends critically on efficient communication mechanisms. At present, user-level communication systems offer the best communication performance. Such systems bypass the operating system on all critical communication paths so that user programs can benefit from the high performance of modern network technology. This thesis concentrates on the interactions between PPSs and user-level communication systems. We focus on three questions:

1. Which mechanisms should a user-level communication system provide?

2. How should parallel-programming systems use the mechanisms provided?

3. How should these mechanisms be implemented?

The relevance of these questions follows from the following observations. First, the functionality provided by user-level communication systems varies considerably (see Chapter 2). Systems differ in the types of data transfer primitives they offer (point-to-point message, multicast message, or remote-memory access), the way incoming data is detected (polling or interrupts), and the way incoming data is handled (explicit receive, upcall, or popup thread).

Second, most user-level communication systems provide low-level interfaces that often do not match the needs of the developers of a PPS. This is not a problem as long as higher-level abstractions can be layered efficiently on top of those interfaces. Unfortunately, many systems ignore this issue (see Section 1.3).

Third, where systems do provide similar functionality, they implement it in different ways (see Chapter 2). Many systems, for example, use a programmable network interface (NI) to execute part of the communication protocol. However, systems differ greatly in the type and amount of work that they off-load to the NI. Some systems minimize the amount of work performed on the NI —on the account that the NI processor is much slower than the host processor— while others run a complete reliability protocol on the NI.

This chapter proceeds as follows. Section 1.1 discusses PPSs in more detail and explains which types of PPSs we address exactly. Section 1.2 discusses user-level architectures and introduces the different layers of communication software we study. Section 1.3 introduces the specific problems that this thesis addresses. Section 1.4 states our contributions towards solving these problems. Section 1.5 discusses the software components in which we have implemented our ideas. Section 1.6 describes the environment in which we developed and evaluated our solutions. Finally, Section 1.7 describes the structure of the thesis.

1.1 Parallel-Programming Systems

The use of parallelism in and between computers is by now the rule rather than the exception. Within a modern processor, we find a pipelined CPU with multiple functional units, nonblocking caches, and write buffers. Surprisingly, this intraprocessor parallelism is largely hidden from the programmer by instruction set architectures that provide a more or less sequential programming model. Where parallelism does become visible (e.g., in VLIW architectures), compilers hide it once more from most application programmers. This way, application programmers have enjoyed advances in computer architecture without having to change the way they write their programs.

Programmers who want to exploit parallelism *between* processors have not been so fortunate. In specific application domains, or for specific types of programs, compilers can automatically detect and exploit parallelism of a sufficiently large grain to employ multiple computers efficiently. In most cases, however, programmers must guide compilers by means of annotations or write an explicitly parallel program. Unfortunately, writing efficient parallel programs is notoriously difficult. At the algorithmic level, programmers must find a compromise between locality and load balance. To achieve both, nontrivial techniques such as data replication, data migration, work stealing, load balancing, etc., may have to be

employed. At a lower level, programmers must deal with race conditions, message interrupts, message ordering, and memory consistency.

To simplify the programmer's task, many parallel-programming systems have been developed, each of which implements its own programming model. The most important programming models are:

- Shared memory
- Message passing
- Data-parallel programming
- Shared objects

In the shared-memory model, multiple threads share an address space and communicate by writing and reading shared-memory locations. Threads synchronize by means of atomic memory operations and higher-level primitives built upon these atomic operations (e.g., locks and condition variables). Multiprocessors directly support this programming model in hardware.

In the message-passing model, processes communicate by exchanging messages. Implementations of this model (e.g., PVM [137] and MPI [53]) usually provide several flavors of send and receive methods which differ, for example, in when the sender may continue and how the receiver selects the next message to receive.

In the shared-memory and message-passing models, the programmer has to deal with multiple threads or processes. A data-parallel program has only one thread of control and in many ways resembles a sequential program. For efficiency, however, the programmer provides annotations that indicate which loops can be executed in parallel and how data must be distributed.

In the shared-object model [7], processes can share data, provided that this data is encapsulated in an object. The data is accessed by invoking methods of the object's class. While this model is not as widely used in practice as the previous models, it has received a fair amount of attention in the parallel-programming research community.

Parallel-programming models used to be identified with a specific type of parallel hardware. Today, most vendors of parallel hardware support multiple programming models. The shared-memory model, for example, is a more natural match to a multiprocessor than the message-passing model, yet multiprocessor vendors also provide implementations of message-passing standards such as MPI.

At present, two types of hardware dominate both the commercial market place and parallel-programming research in academia: multiprocessors and workstation clusters. In a multiprocessor, the hardware implements a single, shared memory

Programming model	Multiprocessor implementations	Cluster implementations
Shared memory	Pthreads [110]	Shasta [125], Tempest [119], TreadMarks [78]
Message passing	MPI [53, 140]	MPI [53, 61], PVM [137]
Data-parallel	OpenMP [42]	HPF [81]
Shared objects	Java [60]	Orca [7, 9], CRL [74], Java/RMI [99]

Table 1.1. Classification of parallel-programming systems.

that can be accessed by all processors through memory access instructions. In a workstation cluster, each processor has its own memory. Access to another processor's memory is not transparent and can only be achieved by some form of explicit message passing.

All four programming models mentioned above have been implemented on both types of hardware (see Table 1.1). Implementations on cluster hardware tend to be more complex than multiprocessor implementations because efficient state sharing is more difficult on a cluster than on a multiprocessor. This thesis addresses only communication support for clusters. On clusters, models such as the shared-object model can be implemented only by software that hides the hardware's distributed nature.

All PPSs discussed in this thesis (except Multigame [122], which we discuss only briefly) require the programmer to write a parallel program. With some systems, however, this is easier than with others. The Orca shared-object system [7, 9], for example, is language-based. Programmers therefore need not write any code to marshal objects or parameters of operations because this code is either generated by the compiler or hidden in the language runtime system. In a library-based system such as MPI, in contrast, programmers must either write their own marshaling code or create type descriptors.

At a higher level, consider locality and load balancing. Distributed shared memory systems like CRL [74] and Orca replicate data transparently to optimize read accesses, thus improving locality. Systems such as Cilk [55] and Multigame use a work-stealing runtime system that automatically balances the load among all processors. An MPI programmer who wishes to replicate a data item must do so explicitly, without any help from the system to keep the replicas consistent. Similarly, MPI programmers have to implement their own load balancing schemes.

The conveniences of high-level programming systems do not come for free. In a recent study [97], for example, Lu et al. show that parallel applications written

for the TreadMarks [78] distributed shared-memory system (DSM) send more data and more messages than equivalent message-passing programs. The reasons are that TreadMarks cannot combine data transfer and synchronization in a single message transfer, cannot transfer data from multiple memory pages in a single message, and suffers from false sharing and diff accumulation. (Diff accumulation is specific to the consistency protocol used by TreadMarks. The result of diff accumulation is that processors communicate to obtain old versions of shared data that have already been overwritten by newer versions.)

Orca provides a second example. Although Orca's shared-object programming model is closer to message passing than TreadMarks's shared-memory model, Orca programs also send more messages than is strictly needed. The current implementation of Orca [9], for example, broadcasts operations on a replicated object to *all* processors, even if the object is replicated on only a few processors. Also, operations on nonreplicated objects are always executed by means of remote procedure calls. Each such operation results in a request and a reply message, even if no reply is needed.

These examples illustrate that, with high-level PPSs, application programmers no longer control the implementation of high-level functions and must rely on generic implementations. This results in increased message rates and increased message sizes. In addition, high-level PPSs add message-processing overheads in the form of marshaling and message-handler dispatch. Consequently, efficient communication support is of prime importance to these systems.

1.2 Communication architectures for parallel-programming systems

To achieve efficient communication, we need both efficient hardware and efficient software. While high-bandwidth, low-latency, and scalable network hardware is still expensive, such hardware *is* available and can be used readily outside supercomputer cabinets. Examples of such hardware include GigaNet [130], Memory Channel [57], and Myrinet [27]. On these systems, network bandwidth is measured in Gigabits per second and network latency in microseconds. Most of these networks have low bit-error rates. Finally, network interfaces have become more flexible and are sometimes programmable.

In many ways, *software* is the main obstacle on the road to efficient communication. Software overhead comes in many forms: system calls, copying, interrupts, thread switches, etc. Part of the software problem has been solved by the introduction of *user-level communication architectures* which give user processes direct access to the network. In most user-level communication systems, users

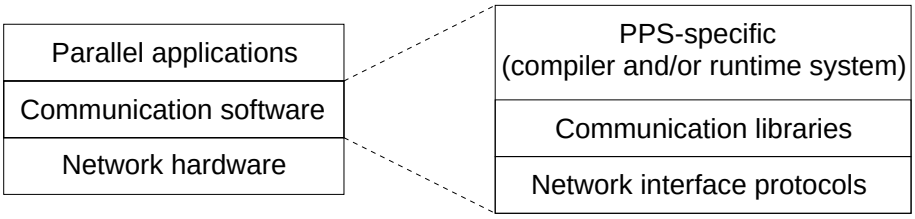


Fig. 1.1. Communication layers studied in this thesis.

must initialize communication segments and channels by means of system calls. After this initialization phase, however, kernel mediation is no longer needed, or needed only in exceptional cases.

Communication support for a PPS can be divided into three layers of software (see Figure 1.1). At the bottom, above the network hardware, we find *network interface protocols*. These protocols perform two functions: they control the network device and implement a low-level communication abstraction that is used by all higher layers.

Since NI protocols are often low-level, most (but not all) PPSs use a *communication library* that implements message abstractions and higher-level communication primitives (e.g., remote procedure call).

The top layer implements a PPS's programming model. In general, this layer consists of a compiler and a runtime system. Not all PPSs are based on a programming language, however, so not all PPSs use a compiler. In principle, PPSs based on a programming language do not need a runtime system: a compiler can generate all code that needs to be executed. In practice, though, PPSs always use some runtime support.

Since communication events frequently cut through all these layers, application-level performance is determined by the way these layers cooperate. In particular, high performance at one layer is of no use if this layer offers an inconvenient interface to higher layers. Our goal in this thesis is to find mechanisms and interfaces that work well at all layers.

1.3 Problems

This thesis addresses three problems:

1. The data transfer methods provided by user-level communication systems often do not match the needs of PPSs.

2. The control transfer methods provided by user-level communication systems do not match the needs of PPSs.

3. Design alternatives for reliable point-to-point and multicast implementations for modern networks are poorly understood.

1.3.1 Data-Transfer Mismatches

Achieving efficient data transfer at all layers in a PPS is hard. A data transfer problem that has received much attention is the high cost of memory copies that take place as data travels from one layer to another. Since redundant memory copies decrease application-to-application throughput (see Chapter 2), much user-level communication work has focused on avoiding unnecessary copies [13, 24, 32, 49]. Most solutions involve the use of DMA transfers between the user's address space and the NI. Unfortunately, these solutions cannot always be used effectively by PPSs.

At the sending side, some systems (e.g., U-Net [147] and Hamlyn [32]) can transfer data efficiently (using DMA) when it is stored in a special *send segment* in the sender's host memory. For a specific application it may be feasible to store the data that needs to be communicated (in a certain period of time) in a send segment. A PPS, however, would have to do this for *all* applications. If this is not possible, senders must first copy their data into the send segment. With this extra copy, the advantage of a low-level high-speed data transfer mechanism is lost.

To improve performance at the receiving side, several systems (e.g., Hamlyn and SHRIMP [24]) offer an interface that allows a sender to specify a destination address in a receiver's address space. This allows the communication system to move incoming data directly from the NI to its destination address. A message-passing system, on the other hand, would first copy the data to some network buffer and would later, when the *receiver* specifies a destination address, copy the data to its final destination. This may appear inefficient, but in many cases the sender simply does not know the destination address of the data that needs to be sent. In such cases, higher layers (e.g., PPSs) are forced to negotiate a destination address before the actual data transfer takes place. Such a negotiation may introduce up to two extra messages per data transfer and is therefore expensive for small data transfers. (Alternatively, the sender can transfer the data to a default buffer with a known address and have the receiver copy the data from that buffer. Such a scheme can be extended so that the receiver can post a destination buffer that replaces the default buffer [49]. If the destination buffer is posted in time, no extra copy is needed.)

The need to avoid copying should be balanced against the cost of managing asynchronous data transfer mechanisms and negotiating destination addresses. Also, not all communication overhead results from a lack of bandwidth. In their study of split-C applications [102], Martin et al. found that applications are sensitive to send and receive overhead, but tolerate lower bandwidths fairly well.

A data transfer problem that has received less attention, but that is important in many PPSs, is the efficient transfer of data from a single sender to multiple receivers. Most existing user-level communication systems focus on high-speed data transfers from a single sender to a single receiver and do not support multicast or support it poorly. This is unfortunate, because multicast plays an important role in many PPSs.

MPICH [61], a widely used implementation MPI, for example, uses multicast in its implementation of *collective communication* operations (e.g., reduce, gather, scatter). Collective communication operations are used in many parallel algorithms. Efficient multicast primitives have also proved their value in the implementation of DSMs such as Orca [9, 75] and Brazos [131], which use multicast to update replicated data.

The lack of efficient multicast primitives in user-level communication systems forces PPSs (or underlying communication libraries) to implement their own multicast on top of point-to-point primitives. This is frequently inefficient. Naive implementations let the sender send a single message to each receiver, which turns the sender into a serial bottleneck. Smarter implementations are based on spanning trees. In these implementations, the sender transmits a message to a few nodes (its children) which then forward the message to their children, and so on until all nodes have been reached. This strategy allows the message to travel between different sender-receiver pairs in parallel.

Even a tree-based multicast can be inefficient if it is layered on point-to-point primitives. At the forwarding nodes, data travels up from the NI to the host. If the host does not poll, forwarding will either be delayed or the data will be delivered by means of an expensive interrupt. To forward the data, the host has to reinject the data into the network. This data transfer is unnecessary, because the data was already present on the NI.

1.3.2 Control-Transfer Mismatches

Control transfer has two components: detecting incoming messages and executing handlers for incoming messages. Most user-level communication systems allow their clients to poll for incoming messages, because receiving a message through polling is much more efficient than receiving it through an interrupt. The problem for a PPS is to decide *when* to poll, especially if the processing of incoming messages is not always tied to an explicit receive call by the application. Again, for a specific application this need not be a hard problem, but getting it right for all applications is difficult.

In DSMs, for example, processes do not interact with each other directly, but only through shared data items. When a process accesses a shared data item that is stored remotely, it typically sends an access request to the remote processor.

The remote processor must respond to the request, even if none of its processes are accessing the data item. If the remote processor is not polling the network when the request arrives, the reply will be delayed. This can be solved by using interrupts, but these are expensive.

Once a message has been detected, it needs to be handled. An interesting problem is in which execution context messages should be handled. Allocating a thread to each message is conceptually clean, but potentially expensive, both in time and space. Moreover, dispatching messages to multiple threads can cause messages to be processed in a different order than they were received. Sometimes this is necessary; at other times, it should be avoided. Executing application code and message handlers in the same thread gives good performance, but can lead to deadlock when message handlers block.

1.3.3 Design Alternatives for User-Level Communication Architectures

The low-level communication protocols of user-level architectures differ substantially from traditional protocols such as TCP/IP. For example, many protocols now use the NI to implement reliability [37, 49], multicast [16, 56, 146], network mapping [100], performance monitoring [93, 103], and address translations and (remote) memory access [13, 24, 32, 51, 83, 126]. Current user-level communication systems make different design decisions in these areas. Specifically, different systems divide similar protocol tasks between the host and the NI in different ways. In some systems, for example, the NI is responsible for implementing reliable communication. In other systems, this task is left to the host processor.

The impact of different design choices on the performance of PPSs and applications has hardly been investigated. Such an investigation is difficult, because existing architectures implement different functionality and provide different programming interfaces. Moreover, many studies use only low-level microbenchmarks that ignore performance at higher layers [5].

1.4 Contributions

The main contributions of this thesis towards solving the problems presented in the previous section are as follows:

1. We show that a small set of simple, low-level communication mechanisms can be employed effectively to obtain efficient PPS implementations that do not suffer from the data and control-transfer mismatches described in the

previous section. Some of these communication mechanisms rely on NI support (see below).

2. We have designed and implemented a new, efficient, and reliable multicast algorithm that uses the NI instead of the host to forward multicast traffic.

3. To gain insight in the tradeoffs involved in choosing between different NI protocol implementations, we have implemented a single user-level communication interface in multiple ways. These implementations differ in whether the host or the NI is responsible for reliability and multicast forwarding. Using these implementations, we have systematically investigated the impact of different design choices on the performance of higher level systems (one communication library and multiple PPSs) and applications.

The communication mechanisms mentioned have been implemented in a new, user-level communication system called LFC. LFC and its mechanisms are summarized in Section 1.5 and described in detail in Chapters 3 to 5. On top of LFC we have implemented a communication library, Panda, that

- provides an efficient (stream) message abstraction
- transparently switches between using polling and interrupts to detect incoming messages
- implements an efficient totally-ordered broadcast primitive

Using LFC and Panda we have implemented and ported various PPSs (see Section 1.5). We describe how we modified some of these PPSs to benefit from LFC's and Panda's communication support.

1.5 Implementation

We have implemented our solutions in various communication systems (see Figure 1.2). These systems cover all communication layers shown in Figure 1.1: NI protocol, communication library, and PPS. The following section introduces the individual systems, layer by layer, bottom-up.

1.5.1 LFC Implementations

We have developed a new NI protocol called LFC [17, 18]. LFC provides reliable point-to-point and multicast communication and runs on Myrinet [27], a modern, switched, high-speed network.

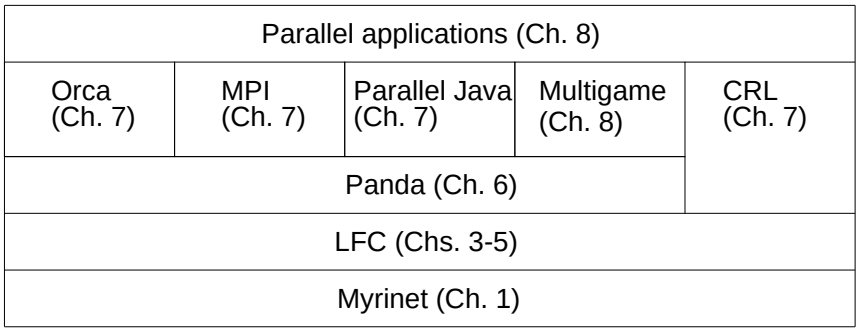


Fig. 1.2. Structure of this thesis.

LFC has a low-level, packet-based interface. By exposing network packets, LFC allows its clients to avoid most redundant memory copies. Packets can be received through polling or interrupts and clients can dynamically switch between the two. LFC delivers incoming packets by means of upcalls, as in Active Messages [148].

We have implemented LFC's point-to-point and multicast primitives in five different ways. The implementations differ in their reliability assumptions and in how they divide protocol work between the NI and the host. Chapters 3 to 5 describe the most aggressive of these implementations in detail. This implementation exploits Myrinet's programmable NI to implement the following:

1. Reliable point-to-point communication.
2. An efficient, reliable multicast primitive.
3. A mechanism to reduce interrupt overhead (a polling watchdog).
4. A fetch-and-add primitive for global synchronization.

This implementation achieves reliable point-to-point communication by means of an NI-level flow control protocol that assumes that the network hardware is reliable. A simple extension of this flow control protocol allows us to implement an efficient, reliable, NI-level multicast. In this multicast implementation, packets need not travel to the host and back before they are forwarded. Instead, the NI recognizes multicast packets and forwards them to children in the multicast tree without host intervention.

The implementation described above performs much protocol work on the programmable NI and assumes that the network hardware does not drop or corrupt network packets. Our other implementations, described in Chapter 8, differ mainly in where and how they implement reliability and multicast forwarding. Our most conservative implementation, for example, assumes unreliable hardware

and implements both reliability (including retransmission) and multicast forwarding on the host.

LFC has been used to implement or port several systems, including CRL [74], Fast Sockets [120], Parallel Java [99], MPICH [61], Multigame [122], Orca [9], Panda [19, 124], and TreadMarks [78]. Several of these systems are described in this thesis; they are introduced in the sections below.

1.5.2 Panda

Panda is a portable communication library that provides threads, messages, reliable message passing, Remote Procedure Call (RPC), and totally-ordered group communication. Using Panda, we have implemented various PPSs (see below).

To implement its abstractions efficiently, Panda exploits LFC's packet-based interface and its efficient multicast, fetch-and-add, and interrupt-management primitives. Panda uses LFC's packet interface to implement an efficient message abstraction which allows end-to-end pipelining of message data and which allows applications to delay message processing without copying message data. All incoming messages are handled by a single, dedicated thread, which solves part of the blocking-upcall problem. To automatically switch between polling and interrupts, Panda integrates thread management and communication such that the network is automatically polled when all threads are idle. Finally, Panda uses LFC's NI-level multicast and synchronization primitives to implement an efficient totally-ordered multicast primitive.

1.5.3 Parallel-Programming Systems

In this thesis, we use five PPSs to test our ideas: Orca, CRL, MPI, Manta, and Multigame.

Orca is a DSM system based on the notion of user-defined shared objects. Jointly, the Orca compiler and runtime system automatically replicate and migrate shared objects to improve locality. To perform operations on shared objects, Orca uses Panda's threads, RPC, and totally-ordered group communication.

Like Orca, CRL is a DSM system. CRL processes share chunks of memory which they can map into their address space. Applications must bracket their accesses to these regions by library calls so that the CRL runtime system can keep the shared regions in a consistent state. Unlike Orca, which updates replicated shared objects, CRL uses invalidation to maintain region-level coherence.

MPI is a message-passing standard. Parallel applications are often developed directly on top of MPI, but MPI is also used as a compiler target. Our implementation of MPI is based on MPICH [61] and Panda. MPICH is a portable and

PPS	Communication patterns	Message detection	# contexts
Orca	Roundtrip + broadcast	Polling + interrupts	1
CRL	Roundtrip	Polling + interrupts	0
MPI	One-way + broadcast	Polling	0
Manta	Roundtrip	Polling + interrupts	≥ 1
Multigame	One-way	Polling	0

Table 1.2. Communication characteristics of parallel-programming systems.

widely used public-domain implementation of MPI that is being developed jointly by Argonne National Laboratories and Mississippi State University.

Manta is a PPS that allows Java [60] threads to communicate by invoking methods on shared objects, as in Orca. Superficially, Manta has simpler communication requirements than Orca, because Manta does not replicate shared objects and therefore requires only RPC-style communication. In reality, however, several features of Java that are not present in Orca —specifically, garbage collection and condition synchronization at arbitrary program points— lead to more complex interactions with the communication system.

Multigame is a declarative parallel game-playing system. Given the rules of a board game and a board evaluation function, Multigame automatically searches for good moves, using one of several search strategies (e.g., IDA* or alpha-beta). During a search, processors push search jobs to each other (using one-way messages). A job consists of (recursively) evaluating a board position. To avoid researching positions, positions are cached in a distributed hash table.

Not only do these PPSs cover a range of programming models, they also have different communication requirements. Table 1.2 summarizes the main communication characteristics of all five PPSs. (A more detailed discussion of these characteristics appears in Chapters 7 and 8.) The second column lists the most common type of communication pattern used in each PPS. The third column shows how each PPS detects incoming messages. All DSMs (Orca, CRL, and Manta) use a combination of polling and interrupts. The fourth column shows how many independent message-handler contexts can be active in each PPS. In CRL, MPI, and Multigame, all incoming messages are processed by handlers that run in the same context as the main computation (i.e., as procedure calls), so there are no independent handler contexts. In Orca, all handlers are run by a dedicated thread. Finally, Manta creates a new thread for each incoming message.

1.6 Experimental Environment

For most of the experiments described in this thesis, we use a cluster that consists of 128 computers, which are interconnected by a Myrinet [27] network. Before describing the hardware in detail, we first position our cluster architecture.

1.6.1 Cluster Position

The cluster used in this thesis is a compromise between a traditional supercomputer and LAN technology. Unlike a traditional supercomputer, the NI connects to the host's I/O bus and is therefore far away from the host processor. This is a typical organization for commodity hardware, but aggressive supercomputer architectures integrate the NI more tightly with the host system by placing it on the memory bus or integrating it with the cache controller. These architectures allow for very low network access latencies and simpler protection schemes [24, 85, 108]. In this thesis, however, we focus on architectures that use off-the-shelf hardware and rely on advanced software techniques to achieve efficient user-level network access. This type of architecture (i.e., with the NI residing on the host's I/O bus) is now also found in several commercial parallel machines (e.g., the IBM SP/2).

The cluster's network, Myrinet, is not as widely used as the ubiquitous Ethernet LAN. As a parallel-processing platform, however, Myrinet is used in many places, both in academia and industry.

Myrinet's NI contains a programmable, custom RISC processor and fast SRAM memory. The network consists of high-bandwidth links and switches; network packets are cut-through-routed through these links and switches. These features make Myrinet (and similar products) much more expensive than Ethernet, the traditional bus-based LAN technology. While Ethernet can be switched to obtain high bandwidth, most Ethernet NIs are not programmable.

Our main reason for using Myrinet is that its programmable NI enables experimentation with different protocols, mechanisms, and interfaces. This type of experimentation is not specific to Myrinet and has also been performed with other programmable NIs [34, 147]. The main problem is often the vendors' reluctance to release information that allows other parties to program their NIs. Myricom, the company that develops and sells Myrinet, does provide the documentation and tools that enable customers to program the NI.

1.6.2 Hardware Details

Each cluster node contains a single Siemens-Nixdorf D983 motherboard, an Intel 440FX PCI chipset, and an Intel Pentium Pro [70] processor. The nodes are connected via 32 8-port Myrinet switches, which are organized in a three-dimensional

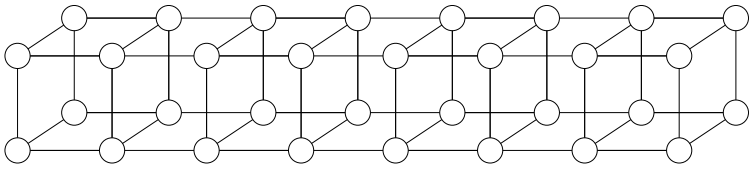


Fig. 1.3. Myrinet switch topology.

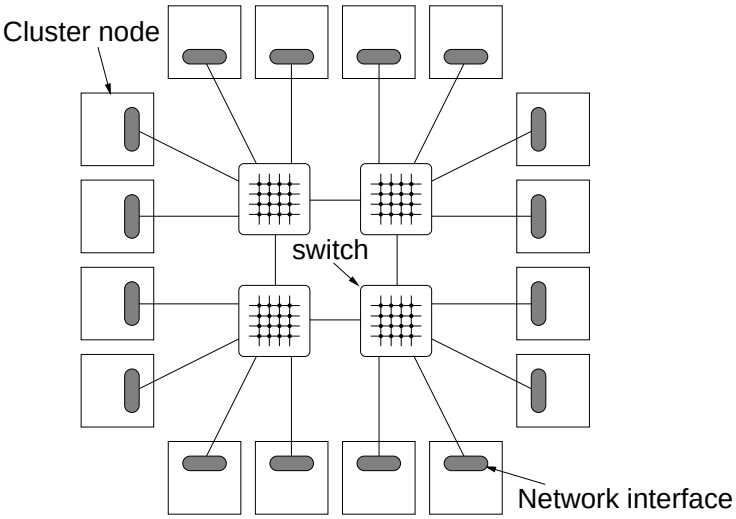


Fig. 1.4. Switches and cluster nodes. Each node has its own network interface which connects to a crossbar switch. Switches connect to network interfaces and other switches.

grid topology. Figure 1.3 shows the switch topology. Figure 1.4 shows one vertical plane of the switch topology. Each switch connects to 4 cluster nodes and to neighboring switches. (Figure 1.4 shows only the connections to switches in the same vertical plane.) The cluster nodes can also communicate through a FastEthernet network (not shown in Figures 1.3 and 1.4). FastEthernet is used for process startup, file transfers, and terminal traffic.

The architecture of a single cluster node is illustrated in Figure 1.5. Each node contains a single Pentium Pro processor and 128 Mbyte of DRAM. The Pentium Pro is a 200 MHz, three-way superscalar processor. It has two on-chip first-level caches: an 8 Kbyte, 2-way set-associative data cache and an 8 Kbyte, 4-way set-associative instruction cache. In addition, a unified, 256 KByte, 4-way set-associative second-level cache is packaged along with the processor core. The cache line size is 32 bytes. The processor-memory bus is 64 bits wide and clocked at 66 MHz. The I/O bus is a 33 MHz, 32-bit PCI bus. The Myrinet NI is attached to the I/O bus.

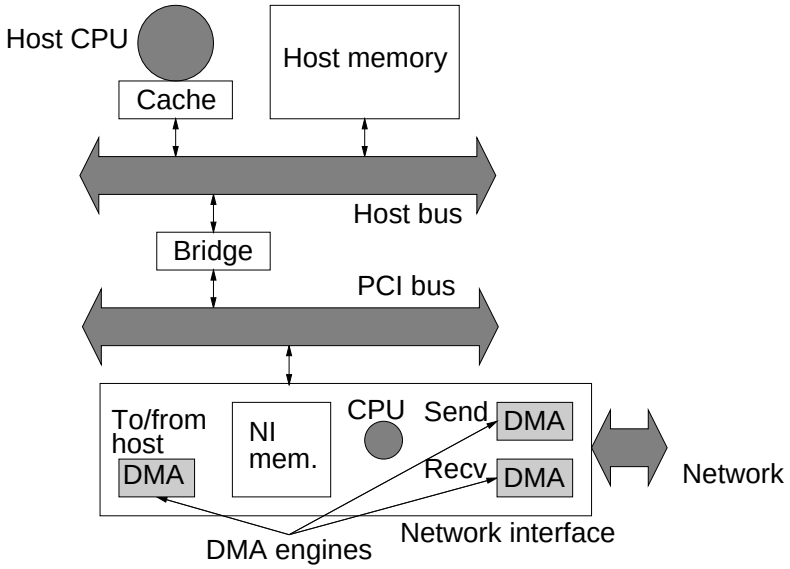


Fig. 1.5. Cluster node architecture.

To obtain accurate timings in our performance measurements, we use the Pentium Pro's 64-bit *timestamp counter* [71]. The timestamp counter is a clock with clock cycle (5 ns) resolution. A simple pseudo device driver makes this clock available to unprivileged users. The counter can be read (from user space) using a single instruction, so the use of this fine-grain clock imposes little overhead (approximately 0.17 μs).

Myrinet is a high-speed, switched LAN technology [27]. Switched technologies scale well to a large number of hosts, because bandwidth increases as more hosts and switches are added to the network. Unlike Ethernet, Myrinet provides no hardware support for multicast. General-purpose multicasting for wormhole-routed networks is a complex problem and an area of active research [136, 135].

Unlike traditional NIs, Myrinet's NI is programmable; it contains a custom processor which can be programmed in C or assembly. The processor, a 33 MHz LANai4.1, is controlled by the LANai control program. The processor is effectively an order of magnitude slower than the 200 MHz, superscalar host processor.

The NI is equipped with 1 Mbyte of SRAM memory which holds both the code and the data for the control program. The currently available Myrinet NIs require that all network packets be staged through this memory, both at the sending and the receiving side. SRAM is fast, but expensive, so the amount of memory is relatively small. Other networks allow data to be transferred directly between host memory and the network (e.g., using memory-mapped FIFOs or DMA).

The NI contains three DMA engines. The *host DMA engine* transfers data

between host memory and NI memory. Host memory buffers that are accessed by this DMA engine should be pinned (i.e., marked unswappable) to prevent the operating system from paging these pages to disk during a DMA transfer. The *send DMA engine* transfers packets from NI memory to the outgoing network link; the *receive DMA engine* transfers incoming packets from the network link to NI memory. The DMA engines can run in parallel, but the NI's memory allows at most two memory accesses per cycle, one to or from the PCI bus and one to or from the NI processor, the send DMA engine, or the receive DMA engine.

Myrinet NIs connect to each other via 8-port crossbar switches and high-speed links (1.28 Gbit/s in each direction). The Myrinet hardware uses routing and switching techniques similar to those used in massively parallel processors (MPPs) such as the Thinking Machines CM-5, the Cray T3D and T3E, and the Intel Paragon. Network packets are cut-through-routed from a source to a destination network interface. Each packet starts with a sequence of routing bytes, one byte per switch on the path from source to destination. Each switch consumes one routing byte and uses the byte's value to decide on which output port the packet must be forwarded. Myrinet implements a hardware flow control protocol between pairs of communicating NIs which makes packet loss very unlikely. If all senders on the network agree on a deadlock-free routing scheme and the NIs' control programs remove incoming packets in a timely manner, then the network can be considered reliable (see also Section 3.9 and Chapter 8).

While in transit, a packet may become blocked, either because part of the path it follows is occupied by another packet, or because the destination NI fails to drain the network. In the first case, Myrinet will kill the packet if it remains blocked for more than 50 milliseconds. In the second case, the Myrinet hardware sends a reset signal to the destination NI after a timeout interval has elapsed. The length of this interval can be set in software. Both measures are needed to break deadlocks in the network. If the network did not do this, a malicious or faulty program could block another program's packets.

1.7 Thesis Outline

This chapter introduced our area of research, communication support for PPSs. We sketched the problems in this area and our approach to solving them. Chapter 2 surveys the main design issues for user-level communication architectures and shows that existing systems resolve these issues in widely different ways, which illustrates that the tradeoffs are still unclear. Chapter 3 describes the design and implementation of our most optimistic LFC implementation. Chapter 4 gives a detailed description of the NI-level protocols employed by this LFC implementation. Chapter 5 evaluates the implementation's performance. Chapter 6

describes Panda and its interactions with LFC. Chapter 7 describes the implementation of four PPSs: Orca, CRL, MPI, and Manta. We show how LFC and Panda enable efficient implementations of these systems and describe additional optimizations used within these systems. Chapter 8 compares the performance of five LFC implementations at multiple levels: the NI protocol level, the PPS level, and the application level. Finally, in Chapter 9, we draw conclusions.

Chapter 2

Network Interface Protocols

High-speed networks such as Myrinet offer great potential for communication-intensive applications. Unfortunately, traditional communication protocols such as TCP/IP are unable to realize this potential. In the common implementation of these protocols, all network access is through the operating system, which adds significant overheads to both the transmission path (typically a system call and a data copy) and the receive path (typically an interrupt and a data copy). In response to this performance problem, several user-level communication architectures have been developed that remove the operating system from the critical communication path [48, 147]. This chapter provides insight into the design issues for communication protocols for these architectures. We concentrate on issues that determine the performance and semantics of a communication system: data transfer, address translation, protection, control transfer, reliability, and multicast.

In this chapter, we use the following systems to illustrate the design issues: Active Messages II (AM-II) [37], BIP [118], Illinois Fast Messages (FM) [112, 113], FM/MC [10, 146], Hamlyn [32], PM [141, 142], U-Net [13, 147], VMMC [24], VMMC-2 [49, 50], and Trapeze [154]. All systems aim for high performance and all except Trapeze offer a user-level communication service. Interestingly, however, they differ significantly in how they resolve the different design issues. It is this variety that motivates our study.

This chapter is structured as follows. Section 2.1 explains the basic principles of NI protocols by describing a simple, unreliable, user-level protocol. Next, in Sections 2.2 to 2.7, we discuss the six protocol design issues that determine system performance and semantics. We illustrate these issues by giving performance data obtained on our Myrinet cluster.

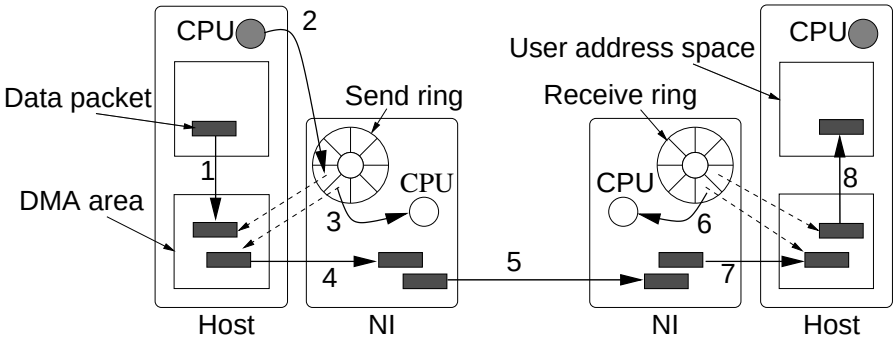


Fig. 2.1. Operation of the basic protocol. The dashed arrows are buffer pointers. The numbered arrows represent the following steps. (1) Host copies user data into DMA area. (2) Host writes packet descriptor to send ring. (3) NI processor reads packet descriptor. (4) NI DMAs packet to NI memory. (5) Network transfer. (6) NI reads receive ring to find empty buffer in DMA area. (7) NI DMAs packet to DMA area. (8) Optional memory copy to user buffer by the host processor.

2.1 A Basic Network Interface Protocol

The goal of this section is to explain the basics of NI protocols and to introduce the most important design issues. To structure our discussion, we first describe the design of a simple, user-level NI protocol for Myrinet. The protocol ignores several important problems, which we address in subsequent sections.

To avoid the cost of kernel calls for each network access, the basic protocol maps all NI memory into user space. User processes write their send requests directly to NI memory, without operating system (OS) involvement. The basic protocol provides no protection, so the network device cannot be shared among multiple processes.

User processes invoke a simple send primitive to send a data packet. The basic protocol sends packets with a maximum payload of 256 bytes and requires that users fragment their data so that each fragment fits in a packet. The signature of the send primitive is as follows:

```
void send(int destination, void *data, unsigned size);
```

Send() performs two actions (see Figure 2.1). First, it copies the user *data* to a packet buffer in a special staging area in host memory (step 1). The NI will later fetch the packet from this *DMA area* by means of a DMA transfer. Unlike normal user pages, pages in the DMA area are never swapped to disk by the OS. By only DMAing to and from such *pinned* pages, the protocol avoids corruption of user memory and user messages due to paging activity of the OS.

Second, the host writes a send request into a descriptor in NI memory (step 2). These descriptors are stored in a circular buffer called the *send ring*. *Send*() stores the identifier of the destination machine, the size of the packet's payload, and the packet's offset in the DMA area into the next available descriptor in this send ring. To inform the NI of this event, *send*() also sets a *DescriptorReady* flag in the descriptor. This flag prevents races between the host and the NI: the NI will not read any other descriptor fields until this flag has been set. The descriptor is written using programmed I/O; since the descriptor is small, DMA would have a high overhead.

The NI repeatedly polls the *DescriptorReady* flag of the first descriptor in the send ring. As soon as this flag is set by the host, the NI reads the offset in the descriptor and adds it to the physical address of the start of the DMA area, resulting in the physical address of the packet (step 3). Next, the NI initiates a DMA transfer over the I/O bus to copy the packet's payload from host memory to NI memory (step 4). Subsequently, it reads the destination machine in the descriptor and looks up the route for the packet in a routing table. The route and a packet header that contains the packet's size are prepended to the packet. Finally, the NI starts a second DMA to transmit the packet (step 5).

When the sending NI detects that the network DMA for a given packet has completed, it sets a *DescriptorFree* flag to release the descriptor, and polls the next free descriptor in the ring. If the host wants to send a packet while no descriptor is available, it busy-waits by polling the *DescriptorFree* flag of the descriptor at the tail of the ring.

NIs use receive DMAs to store incoming packets in their memory. Each NI contains a *receive ring* with descriptors that point to free buffers in the host's DMA area. The NI uses the receive ring's descriptors to determine where (in host memory) to store incoming packets. After receiving a packet, the NI tries to acquire a descriptor from the receive ring (step 6). Each descriptor contains a flag bit that is used in a similar way as for the send ring. If no free host buffer is available, the packet is simply dropped. Otherwise, the NI starts a DMA to transfer the packet to host memory (step 7). Each host buffer also contains a flag that is set by the NI as the last part of its NI-to-host DMA transfer. The host can check if there is a packet available by polling the flag of the next unprocessed host receive buffer. Once the flag has been set, the receiving process can safely read the buffer and optionally copy its contents to a user buffer (step 8).

The data transfers in steps 1, 4, 5, 7, and 8 can all be performed concurrently. For example, if the host sends a long, multipacket message, one packet's network DMA can be overlapped with the next packet's host-to-NI DMA. Exploiting this concurrency is essential for achieving high throughput.

Since the delivery of network interrupts to user-level processes is expensive on current OSs, the basic protocol does not use interrupts, but requires users to

poll for incoming messages. A successful call to *poll*() results in the invocation of a user function, *handle_packet*(), that handles an inbound packet:

```
void poll(void);
void handle_packet(void *data, unsigned size);
```

Our (unoptimized) implementation of the basic protocol achieves a one-way host-to-host latency of 11 μs and a throughput of 33 Mbyte/s (using 256-byte packets). For comparison, on the same hardware the highly optimized BIP system [118], achieves a minimum latency of 4 μs and can saturate the I/O bus (127 Mbyte/s). For large data transfers, BIP uses DMA, both at the sending and the receiving side. In contrast with the basic protocol, however, BIP does not stage data through DMA areas. Section 2.3 discusses different techniques to eliminate the use of DMA areas.

The basic protocol avoids all OS overhead, keeps the NI code simple, and uses little NI memory. It is clear, however, that the protocol has several shortcomings:

- All inbound and outbound network transfers are staged through a DMA area. For applications that need to send and receive from arbitrary locations, this introduces extra memory copies.
- The protocol provides no protection. If the basic protocol allowed multiple users to access the NI, these users could read and modify each other's data in NI memory. Users can even modify the NI's control program and use it to access any host memory location.
- The receiver-side control transfer mechanism, polling, is simple, but not always effective. For many applications it is difficult to determine a good polling rate. If the host polls too frequently, it will have a high overhead; if it polls too late, it will not reply quickly enough to incoming packets.
- The protocol is unreliable, even though the Myrinet hardware is highly reliable. If the senders send packets faster than the receiver can handle them, the receiving host will run out of buffer space and the NI will drop incoming packets.
- The protocol supports only point-to-point messages. Although multicast can be implemented on top of point-to-point messages, doing so may be inefficient. Multicast is an important service by itself and a fundamental component of collective communication operations such as those supported by the message-passing standard MPI.

Below, we will discuss these problems in more detail and look at better design alternatives.

2.2 Data Transfers

On Myrinet, at least three steps are needed to communicate a packet from one user process to another: the packet must be moved from the sender's memory to its NI (host-NI transfer), from this NI to the receiver's NI (NI-NI transfer), and then to the receiving process's address space (NI-host transfer). Network technologies that do not require data to be staged through NI memory use only two steps: host-to-network and network-to-host. Below, we discuss the Myrinet case, but most issues (programmed I/O versus DMA, pinning, alignment, and maximum packet size) also apply to the two-step case.

The data transfers have a significant impact on the latency and throughput obtained by a protocol, so optimizing them is essential for obtaining high performance. As shown in Figure 2.1, the basic protocol uses five data transfers to communicate a packet, because it stages all packets through DMA areas. Below, we discuss alternative designs for implementing the host-NI, NI-NI, and NI-host transfers.

2.2.1 From Host to Network Interface

On Myrinet, the host-to-NI transfer can use either DMA or Programmed I/O (PIO). Steenkiste gives a detailed description of both mechanisms [133]. With PIO, the host processor reads the data from host memory and writes it into NI memory, typically one or two words at a time, which results in many bus transactions. DMA uses special hardware (a DMA engine) to transfer the entire packet in large bursts and asynchronously, so that the data transfer can proceed in parallel with host computations. One thus might expect DMA to always outperform PIO. The optimal choice, however, depends on the type of host CPU and on the packet size. The Pentium Pro, for example, supports *write combining* buffers, which allow memory writes to the same cache line to be merged into a single 32-byte bus transaction. We can thus boost the performance of host-to-NI PIO transfers by applying write combining to NI memory. (This use of the Pentium Pro's write combining facility was suggested to us by the Fast Messages group of the University of Illinois at Urbana-Champaign [31].)

Figure 2.2 shows the throughput obtained by PIO (with and without write combining) and cache-coherent DMA, for copying data from a Pentium Pro to a Myrinet NI card. PIO with write combining quickly outperforms PIO without write combining. For buffer sizes up to 1024 bytes, PIO with write combining even outperforms DMA (which suffers from a startup cost).

For small messages, PIO with write combining is slower than PIO without write combining. This is due to an expensive extra instruction (a serializing instruction) that our benchmark executes after each host-to-NI memory copy with

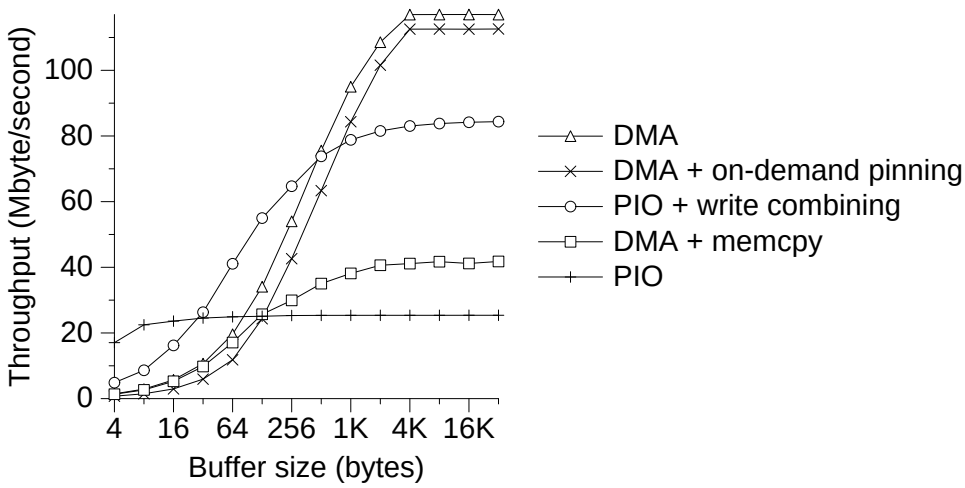


Fig. 2.2. Host-to-NI throughput using different data transfer mechanisms.

write combining. We use this instruction to model the common situation in which a data copy to NI memory is followed by another write that signals the presence of the data. Clearly, the NI should not observe the latter write before the data copy has completed. With write combining, however, writes may be reordered and it is necessary to separate the data copy and the write that follows it by a serializing instruction.

In user-level communication systems, DMA transfers can be started either by a user process or by the network interface without any operating system involvement. Since DMA transfers are performed asynchronously, the operating system may decide to swap out the page that happens to be the source or destination of a running DMA transfer. If this happens, part of the destination of the transfer will be corrupted. To avoid this, operating systems allow applications to *pin* a limited number pages in their address space. Pinned pages are never swapped out by the operating system. Unfortunately, pinning a page requires a system call and the amount of memory that can be pinned is limited by the available physical memory and by OS policies. In the best case, all pages that an application transfers data to or from need to be pinned only once. In this case the cost of pinning the pages can be amortized over many data transfers. In the worst case, an application transfers data to or from more pages than can be pinned simultaneously. In this case, two system calls are needed for every data transfer: one to unpin a previously pinned page and one to pin the page involved in the next data transfer. SHRIMP provides special hardware that allows user processes to start DMA transfers without pinning [25]. This user-level DMA mechanism, however, works only for host-initiated transfers, not for NI-initiated transfers, so pinning is still required at the receiving side.

NI protocols that use DMA often choose to copy the data into a reserved (and pinned) DMA area, which costs an extra memory copy and thus may decrease the throughput. Figure 2.2, shows that a DMA transfer preceded by a memory copy is consistently slower than PIO with write combining. On processors that do not support cache-coherent DMAs, the DMA area needs to be allocated in uncached memory, which also decreases performance. (Most modern CPUs, including the Pentium Pro, support cache-coherent DMA, however.) With PIO, pinning is not necessary. Even if the OS swapped out the page during the transfer, the host's next memory reference would generate a page fault, causing the OS to swap the page back in. In practice, many protocols use DMA; other protocols (AM-II, Hamlyn, BIP) use PIO for small messages and DMA for large messages. FM uses PIO for all messages.

With both DMA and PIO, data transfers between unaligned data buffers can be much slower than between aligned buffers. This problem is aggravated when the NI's DMA engines *require* that source and destination buffers be properly aligned. In that case, extra copying is needed to align unaligned buffers.

Another important design choice is the maximum packet size. Large packets yield better throughput, because per-packet overheads are incurred fewer times than with small packets. The throughput of our basic protocol, for example, increases from 33 Mbyte/s to 48 Mbyte/s by using 1024-byte instead of 256-byte packets. The choice of the maximum packet size is influenced by the system's page size, memory space considerations, and hardware restrictions.

2.2.2 From Network Interface to Network Interface

The NI-to-NI transfer traverses the Myrinet links and switches. All Myrinet protocols use the NI's DMA engine to send and receive network data. In theory, PIO could be used, but DMA transfers are always faster. To send or receive a data word by means of PIO, the processor must always move that data item through a processor register, which costs at least two cycles. A DMA engine can transfer one word per cycle. Moreover, using DMA frees the processor to do other work during data transfers.

To prevent network congestion, the receiving NI should extract incoming data from the network fast enough. On Myrinet, the hardware uses *backpressure* to stall the sending NI if the receiver does not extract data fast enough. To prevent deadlock, however, there is a time limit on the backpressure mechanism. If the receiver does not drain the network within a certain time period, the network hardware will reset the NI or truncate a blocked packet. Many Myrinet control programs deal with this real-time constraint by copying data fast enough to prevent resets. Other protocols avoid the problem by using a software flow control scheme, as we will discuss later.

2.2.3 From Network Interface to Host

The transfer from NI to host at the receiving side can again use either DMA or PIO. On Myrinet, however, only the host (not the NI) can use PIO, making DMA the method of choice for most protocols. Some systems (e.g., AM-II) use PIO on the host to receive small messages. For large messages, all protocols use DMA, because reads over the I/O bus are typically much slower than DMA transfers. Whether we use DMA or PIO, in both cases the bottleneck for the NI-to-host transfer is the I/O bus.

Using microbenchmarks, we measured the attainable throughput on the important data paths of our Pentium Pro/Myrinet cluster, using DMA and PIO. Table 2.1 summarizes the results and also gives the hardware bandwidth of the bottleneck component on each data path. For transfers between the host and the NI, the bottleneck is the 33.33 MHz PCI bus; both main memory and the memory bus allow higher throughputs. With DMA transfers, the microbenchmarks can almost saturate the I/O bus. Our throughput is slightly less than the I/O bus bandwidth because in our benchmarks the NI acknowledges every DMA transfer with a one-word NI-to-host DMA transfer. For network transfers from one NI's memory to another NI's memory, the bottleneck is the NIs' memory. The send and receive DMA engines can access at most one word per I/O bus clock cycle (i.e., 127 Mbyte/s). The bandwidth of the network links is higher, 153 Mbyte/s. Finally, for local host memory copies, the bottleneck component is main memory.

An interesting observation is that a local memory copy on a single Pentium Pro obtains a lower throughput than a remote memory copy over Myrinet (52 Mbyte/s versus 127 Mbyte/s). This problem is largely due to the poor memory write performance of the Pentium Pro [29]; on a 450 MHz Pentium III platform, we measured a memory-to-memory copy throughput of 157 Mbyte/s. Nevertheless, memory copies have an important impact on performance. For comparison, recall that the basic protocol achieves a throughput of only 33 Mbyte/s. The reason is that the basic protocol uses fairly small packets (256 bytes) and performs memory copies to and from DMA areas, which interferes with DMA transfers. Several systems (e.g., BIP and VMMC-2) can saturate the I/O bus: the key issue is to avoid the copying to and from DMA areas. The next section describes techniques to achieve this.

2.3 Address Translation

The use of DMA transfers between host and NI memory introduces two problems. First, most systems require that every host memory page involved in a DMA transfer be pinned to prevent the operating system from replacing that page. Pinning,

Source	Destination	Method	Hardware bandwidth	Measured throughput
Host memory	Host memory	PIO	170	52
Host memory	NI memory	PIO	127	25
		PIO + WC	127	84
		DMA	127	117
NI memory	Host memory	PIO	127	7
		DMA	127	117
NI memory	NI memory	DMA	127	127

Table 2.1. Bandwidths and measured throughputs (in Mbyte/s) on a Pentium Pro/Myrinet cluster. WC means write combining.

however, requires an expensive system call which should be kept off the critical path. The second problem is that, on most architectures, the NI's DMA engine needs to know the physical addresses of each page that it transfers data to or from. Operating systems, however, do not export virtual-to-physical mappings to user-level programs, so users normally cannot pass physical addresses to the NI. Even if they could, the NI would have to check those physical addresses, to ensure that users pass only addresses of pages that they have access to.

We consider three approaches to solve these problems. The first approach is to avoid all DMA transfers by using programmed I/O. Due to the high cost of I/O bus reads, however, this is only a realistic solution at the sending side.

The second approach, used by the basic protocol, requires that users copy their data into and out of special DMA areas (see Figure 2.1). This way, only the DMA areas need to be pinned. This is done once, when the application opens the device, and not during send and receive operations. The address translation problem is then solved as follows. The operating system allocates for each DMA area a *contiguous* chunk of physical memory and passes the area's physical base address to the NI. Users specify send and receive buffers by means of an offset in their DMA area. The NI only needs to add this offset to the area's base address to obtain the buffer's physical address. Several systems (e.g., AM-II, Hamlyn) use this approach, accepting the extra copying costs. As shown in Figure 2.2, however, the extra copy reduces throughput significantly.

In the third approach, the copying to and from DMA areas is eliminated by dynamically pinning and unpinning user pages so that DMA transfers can be performed directly to those pages. Systems that use this approach (e.g., VMMC-2, PM, and BIP) can track the 'DMA' curve in Figure 2.2. The main implementation problem is that the NI needs to know the current virtual-to-physical mappings of individual pages. Since NIs are usually equipped with only a small amount of

memory and since their processors are slow, they do not store information for every single virtual page. Some systems (e.g., BIP) provide a simple kernel module that translates virtual addresses to physical addresses. Users are responsible for pinning their pages and obtaining the physical addresses of these pages from the kernel module. The disadvantage of this approach is that the NI cannot check if the physical addresses it receives are valid and if they refer to pinned pages.

An alternative approach is to let the kernel and NI cooperate such that the NI can keep track of valid address translations (either in hardware or in software). Systems like VMMC-2 and U-Net/MM [13] (an extension of U-Net) let the NI *cache* a limited number of valid address translations which refer to pinned pages (this invariant must be maintained cooperatively by the NI and the operating system). This caching works well for applications that exhibit locality in the pages they use for sending and receiving data. When the translation of a user-specified address is found in the cache, the NI can access that address using a DMA transfer. In the case of a miss, special action must be taken. In U-Net/MM, for example, the NI generates an interrupt when it cannot translate an address. The kernel receives the interrupt, looks up the address in its page table, pins the page, and passes the translation to the NI.

In VMMC-2, address translations for user buffers are managed by a library. This user-level library maps users' virtual addresses to *references* to address translations which users can pass to the NI. The library creates these references by invoking a kernel module. This module translates virtual addresses, pins the corresponding pages, and stores the translations in a *User-managed TLB* (UTLB) in kernel memory.

To avoid invoking the operating system every time a reference is needed, the library maintains a user-level lookup data structure that keeps track of the addresses for which a valid UTLB entry exists. The library invokes the UTLB kernel module only when it cannot find the address in its lookup data structure. When the NI receives a reference, it can find the translation using a DMA transfer to the kernel module's data structure. To avoid such DMA transfers on the critical path, the NI maintains its own cache of references. The 'on-demand pinning' curve in Figure 2.2 shows the throughput obtained by a benchmark that imitates the miss behavior of a UTLB. To simulate a miss in its lookup data structure, the host invokes, for each page that is transferred, a system call to pin that page. To simulate a miss in the NI cache, the NI fetches a single word from host memory before it transfers the data.

2.4 Protection

Since user-level architectures give users direct access to the NI, the OS can no longer check every network transfer. Multiplexing and demultiplexing must now be performed by the NI and special measures are needed to preserve the integrity of the OS and to isolate the data streams of different processes from one another.

In a protected system, no user process should be given unrestricted access to NI memory. With unrestricted access to NI memory, a user process can modify the NI control program and use it to read or write any location in host memory. NI memory access must also be restricted to implement protected multiplexing and demultiplexing. In the basic protocol, for example, user processes directly write to NI memory to initialize send descriptors. If multiple processes shared the NI, one process could corrupt another process's send descriptors. Similarly, no process should be able to read incoming network packets that are destined for another process. The basic protocol prevents these problems by providing user-level network access to at most one user at a time, but this limitation is clearly undesirable in a multi-user and multiprogramming environment.

A straightforward solution to these problems is to use the virtual-memory system to give each user access to a different part of NI memory [48]. When the user opens the network device, the operating system maps such a part into the user's address space. Once the mapping has been established, all user accesses outside the mapped area will be trapped by the virtual-memory hardware. Users write their commands and network data to their own pages in NI memory. It is the responsibility of the NI to check each user's page for new requests and to process only legal requests.

Since NI memory is typically small, only a limited number of processes can be given direct access to the NI this way. To solve this problem, AM-II virtualizes network endpoints in the following way. Part of NI memory acts as a *cache* for active communication endpoints; inactive endpoints are stored in host memory. When an NI receives a message for an inactive endpoint or when a process tries to send a message via an inactive endpoint, the NI and the operating system cooperate to activate the endpoint. Activation consists of moving the endpoint's state (send and receive buffers, protocol status) to NI memory, possibly replacing another endpoint which is then swapped out to host memory.

The sharing problem also exists on the host, for the DMA areas. To maintain protection, each user process needs its own DMA area. Since the use of a DMA area introduces an extra copy, some systems eliminate it and store address translations on the NI. VMMC-2 and U-Net/MM do this in a protected way, either by letting the kernel write the translations to the NI or by letting the NI fetch the translations from kernel memory. BIP, on the other hand, eliminates the DMA area, but does not maintain protection.

2.5 Control Transfers

The control transfer mechanism determines how a receiving host is notified of message arrivals. The options are to use interrupts, polling, or a combination of these.

Interrupts are notoriously expensive. With most operating systems, the time to deliver an interrupt (as a signal) to a user process even exceeds the network latency. On a 200 MHz Pentium Pro running Linux, dispatching an interrupt to a kernel interrupt handler costs approximately 8 μs. Dispatching to a user-level signal handler costs even more, approximately 17 μs. This exceeds the latency of our basic protocol (11 μs).

Given the high costs of interrupts, all user-level architectures support some form of *polling*. The goal of polling is to give the host a fast mechanism to check if a message has arrived. This check must be inexpensive, because it may be executed often. A simple approach is to let the NI set a flag in its memory and to let the host check this flag. This approach, however, is inefficient, since every poll now results in an I/O bus transfer. In addition, this polling traffic will slow down other I/O traffic, including network packet transfers between NI and host memory.

A very efficient solution is to use a special device status register that is shared between the NI and the host [107]. Current hardware, however, does not provide these shared registers.

On architectures with cache-coherent DMA, a practical solution is to let the NI write a flag in cached host memory (using DMA) when a message is available. This approach is used by our basic protocol. The host polls by reading its local memory; since polls are executed frequently, the flag will usually reside in the data cache, so failed polls are cheap and do not generate memory or I/O traffic. When the NI writes the flag, the host will incur a cache miss and read the flag's new value from memory. On a 200 MHz Pentium Pro, the scheme just described costs 5 nanoseconds for a failed poll (i.e., a cache hit) and 74 nanoseconds for a successful poll (i.e., a cache miss). For comparison, each poll in the simple scheme (i.e., an I/O bus transfer) costs 467 nanoseconds.

Even if the polling mechanism is efficient, polling is a mixed blessing. Inserting polls manually is tedious and error-prone. Several systems therefore use a compiler or a binary rewriting tool to insert polls in loops and functions [114, 125]. The problem of finding the right polling frequency remains, though [89]. In multiprocessor architectures this problem can be solved by dedicating one of the processors to polling and message handling.

Several systems (AM-II, FM/MC, Hamlyn, Trapeze, U-Net, VMMC, VMMC-2) support both interrupts and polling. Interrupts usually can be enabled or disabled by the receiver; sometimes the sender can also set a flag in each packet that determines whether an interrupt is to be generated when the packet arrives.

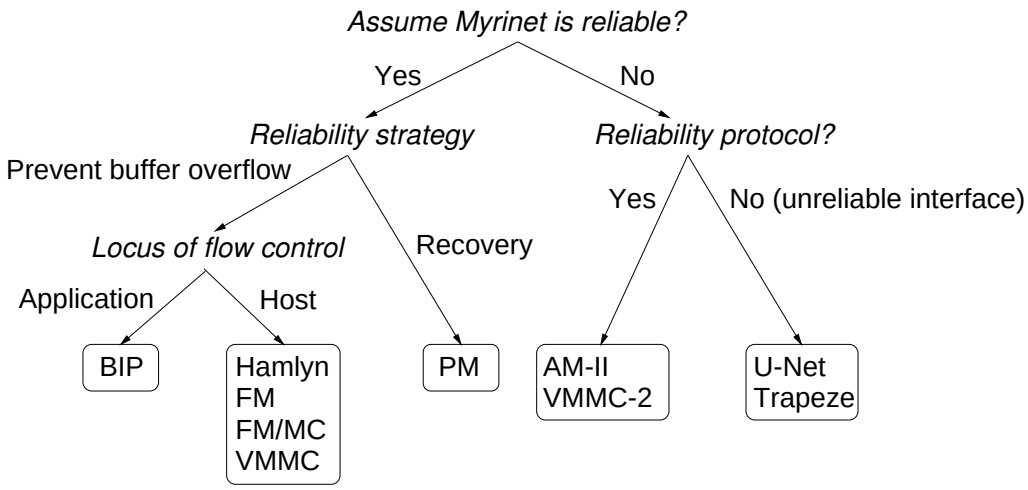


Fig. 2.3. Design decisions for reliability.

2.6 Reliability

Existing Myrinet protocols differ widely in the way they address reliability. Figure 2.3 shows the choices made by various systems. The most important choice is whether or not to assume that the network is reliable. Myrinet has a very low bit-error rate (less than 10^{-15} on shielded cables up to 25 m long [27]). Consequently, the risk of a packet getting lost or corrupted is small enough to consider it a 'fatal event' (much like a memory parity error or an OS crash). Such events can be handled by higher-level software (e.g., using checkpointing), or else cause the application to crash.

Many Myrinet protocols indeed assume that the hardware is reliable, so let us look at these protocols first. The advantage of this approach is efficiency, because no retransmission protocol or time-out mechanism is needed. Even if the network is fully reliable, however, the software protocol may still drop packets due to lack of buffer space. In fact, this is the most common cause of packet loss. Each protocol needs communication buffers on both the host and the NI, and both are a scarce resource. The basic protocol described in Section 2.1, for example, will drop packets when it runs out of receive buffers. This problem can be solved in one of two ways: either *recover* from buffer overflow or *prevent* overflow to happen.

The first idea (recovery) is used in PM. The receiver simply discards incoming packets if it has no room for them. It returns an acknowledgement (ACK or NACK) to the sender to indicate whether or not it accepted the packet. A NACK indicates a packet was discarded; the sender will later retransmit that packet. This process continues until an ACK is received, in which case the sender can release

the buffer space for the message. This protocol is fairly simple; the main disadvantages are the extra acknowledgement messages and the increased network load when retransmissions occur.

A key property of the protocol is that it never drops acknowledgements. By assumption, Myrinet is reliable, so the network hardware always delivers acknowledgements to their destination. When an acknowledgement arrives, the receiver processes it completely before it accepts a new packet from the network, so at most one acknowledgement needs to be buffered. (Myrinet's hardware flow control ensures that pending network packets are not dropped.)

Unlike an acknowledgement, a data packet cannot always be processed completely once it has been received. A data packet needs to be transferred from the NI to the host, which requires several resources: at least a free host buffer and a DMA engine. If the wait time for one of these resources (e.g., a free host buffer) is unbounded, then the NI cannot safely wait for that resource without receiving pending packets, because the network hardware may then kill blocked packets.

The second approach is to prevent buffer overflow by using a *flow control* scheme that blocks the sender if the receiver is running out of buffer space. For large messages, BIP requires the application to deal with flow control by means of a rendezvous mechanism: the receiver must post a receive request and provide a buffer before a message may be sent. That is, a large-message send never completes before a receive has been posted. FM and FM/MC implement flow control using a host-level *credit scheme*. Before a host can send a packet, it needs to have a credit for the receiver; the credit represents a packet buffer in the receiver's memory. Credits can be handed out in advance by pre-allocating buffers for specific senders, but if a sender runs out of credits it must block until the receiver returns new credits.

A host-level credit scheme prevents overflow of host buffers, but not of NI buffers, which are usually even scarcer (because NI memory is smaller than host memory). With some protocols, the NI temporarily stops receiving messages if the NI buffers overflow. Such protocols rely on Myrinet's hardware, link-level, flow control mechanism (backpressure) to stall the sender in such a case.

The protocols described so far implement a reliable interface by depending on the reliability of the hardware. Several other protocols do not assume the network to be reliable, and either present an unreliable programming interface or implement a retransmission protocol. U-Net and Trapeze present an unreliable interface and expect higher software layers (e.g., MPI, TCP) to retransmit lost messages. Other systems do provide a reliable interface, by implementing a timeout-retransmission mechanism, either on the host or the NI. The cost of setting timers and processing acknowledgements is modest, typically no more than a few microseconds.

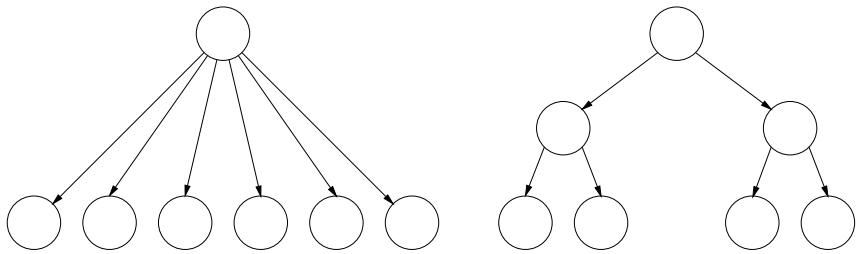


Fig. 2.4. Repeated send versus a tree-based multicast.

2.7 Multicast

Multicasting occurs in many communication patterns, ranging from a straightforward broadcast to the more complicated all-to-all exchange [84]. Message-passing systems like MPI [45] directly support such patterns by means of collective communication services and thus rely on an efficient multicast implementation. In addition, various higher-level systems use multicasting to update replicated shared data [9].

Today's wormhole-routed networks, including Myrinet, do not support reliable multicasting at the hardware level; multicast in wormhole-switched networks is a hard problem and the subject of ongoing research [56, 128, 136, 135]. The simplest way to implement a multicast in software is to let the sender send a point-to-point message to each multicast destination. This solution is inefficient, because the point-to-point startup cost is incurred for every multicast destination; this cost includes the data copy to NI memory (possibly preceded by a copy to a DMA area). With some NI support, the repeated copying can be avoided by passing all multicast destinations to the NI, which then repeatedly transmits the same packet to each destination. Such a 'multisend' primitive is provided by PM.

Although more efficient than a repeated send on the host, a multisend still leaves the network interface as a serial bottleneck. A more efficient approach is to organize the sender and receivers into a multicast tree. The sender is the root of the tree and transmits each multicast packet to all its children (a subset of the receivers). These children, in turn, forward each packet to their children, and so on. Figure 2.4 contrasts the repeated-send and the tree-based multicast strategies. Tree-based protocols allow packets to travel in parallel along the branches of the tree and therefore usually have logarithmic rather than linear complexity.

Tree-based protocols can be implemented efficiently by performing the forwarding of multicast packets on the NI instead of on the host [56, 68, 80, 146]. Verstoep et al. implemented such an NI-level spanning tree multicast as an extension of the Illinois Fast Messages [113] substrate; the resulting system is called FM/MC (*F*ast *M*essages/*M*ulti*C*ast) [146].

An important issue in the design of a multicast protocol is flow control. Multicast flow control is more complex than point-to-point flow control, because the sender of a multicast packet needs to obtain buffer space on all receiving NIs and hosts. FM/MC uses a central buffer manager (implemented on one of the NIs) to keep track of the available buffer space at all receivers. This manager handles all requests for multicast buffers and allows senders to prefetch buffer space for future multicasts. An advantage of this scheme is that it avoids deadlock by acquiring buffer space in advance; a disadvantage is that it employs a central buffer manager. An alternative scheme is to acquire buffers on the fly: the sender of a multicast packet is responsible only for acquiring buffer space at its children in the multicast tree. A more detailed discussion of FM/MC is given in Section 4.5.4.

2.8 Classification of User-Level Communication Systems

Table 2.2 classifies ten user-level communication systems and shows how each system deals with the design issues discussed in this chapter. These systems all aim for high performance and all provide a lean, low-level, and more or less generic communication facility. All systems except VMMC and U-Net were developed first on a Myrinet platform.

Most systems implement a straightforward message-passing model. The systems differ mainly in their reliability and protection guarantees and their support for multicast. Several systems use *active messages* [148]. With active messages, the sender of a message specifies not only the message's destination, but also a *handler function*, which is invoked at the destination when the message arrives.

VMMC, VMMC-2, and Hamlyn provide *virtual memory-mapped* and *sender-based* communication instead of message passing. In both models the sender specifies where in the receiver's address space the communicated data must be deposited. This model allows data to be moved directly to its destination without any unnecessary copying.

In addition to the research projects listed in Table 2.2, industry has recently created a draft standard for user-level communication in cluster environments [51, 149]. Implementations of this *Virtual Interface* (VI) architecture have been constructed by UC Berkeley, GigaNet, Intel, and Tandem. Giganet sells a hardware VI implementation. The others implement VI in device driver software, in NI firmware, or both. Speight et al. discuss the performance of one hardware and one software implementation [130].

System	Data transfer (host-NI)	Address translation	Protection	Control transfer	Reliability	Multicast support
AM-II	PIO + DMA	DMA areas	Yes	Polling + interrupts	Reliable. NI: alternating bit. Host: sliding window.	No
FM	PIO	DMA area (receive)	No	Polling	Reliable. Host-level credits.	No
FM/MC	PIO	DMA area (receive)	No	Polling + interrupts	Reliable. Ucast: host-level credits. Mcast: NI-level credits.	Yes (on NI)
PM	DMA	Software TLB on NI	Yes (gang scheduling)	Polling	Reliable. ACK/NACK protocol on NI.	Multiple sends
VMMC	DMA	Software TLB on NI	Yes	Polling + interrupts	Reliable. Exploits hardware backpressure.	No
VMMC-2	DMA	UTLB in kernel, cached on NI	Yes	Polling + interrupts	Reliable.	No
Hamlyn	PIO + DMA	DMA areas	Yes	Polling + interrupts	Reliable. Exploits hardware backpressure.	No
Trapeze	DMA	DMA to page frames	No	Polling + interrupts	Unreliable.	No
BIP	PIO + DMA	User translates	No	Polling	Reliable. Rendezvous and backpressure.	No
U-Net	DMA	TLB on NI (U-Net/MM)	Yes	Polling + interrupts	Unreliable.	No

Table 2.2. Summary of design choices in existing communication systems.

2.9 Summary

This chapter has discussed several design issues and tradeoffs for NI protocols, in particular:

- how to transfer data between hosts and NIs (DMA or programmed I/O);
- how to avoid copying to and from DMA areas by means of address translation techniques;
- how to achieve protected, user-level network access in a multi-user environment;
- how to transfer control (interrupts, polling, or both);
- how and where to implement reliability (application, hosts, NIs);
- whether to implement additional functionality on the NIs, such as multicast.

An interesting observation is that many novel techniques exploit the programmable NI processor (e.g., for address translation). Programmable NIs are flexible, which compensates for the lack of hardware support present in the more advanced interfaces used by MPPs. Eventually, hardware implementations may be more efficient, but the availability of a programmable NI has enabled fast and easy experimentation with different protocols for commodity network interfaces. As a result, the performance of these protocols has substantially increased. In combination with the economic advantages of commodity networks, this makes such networks a key technology for parallel cluster computing.

An efficient network interface protocol, however, does not suffice; what counts is application performance. As discussed in Section 1.1, most application programmers use a parallel-programming system to develop their applications. There is a large gap between the high-level abstractions provided by these programming systems and the low-level interfaces of the communication architectures discussed in this chapter. It is not a priori clear that all types of programming systems can be implemented efficiently on all of these low-level systems.

Chapter 3

The LFC User-Level Communication System

This chapter describes LFC, a new user-level communication system. LFC distinguishes itself from other systems by the way it divides protocol tasks between the host processor and the network interface (NI). In contrast with communication systems that minimize the amount of protocol code executed on the NI, LFC exploits the NI to implement flow control, to forward multicast traffic, to reduce the overhead of network interrupts, and to perform synchronization operations. This chapter describes LFC's user interface and gives an overview of LFC's implementation on the computer cluster described in Section 1.6. LFC's NI-level protocols are described separately in Chapter 4.

This chapter is structured as follows. Section 3.1 describes LFC's programming interface. Section 3.2 states the key assumptions that we made in LFC's implementation. Section 3.3 gives an overview of the main components of the implementation. Subsequent sections describe LFC's packet format (Section 3.4), data transfer mechanisms (Section 3.5), host buffer management (Section 3.6), the implementation of message detection and dispatch (Section 3.7), and the implementation of a fetch-and-add primitive (Section 3.8). Section 3.9 discusses limitations of the implementation. Finally, Section 3.10 discusses related work.

3.1 Programming Interface

LFC aims to support the development of parallel runtime systems rather than the development of parallel applications. Therefore, LFC provides an efficient, low-level interface rather than a high-level, user-friendly interface. In particular, LFC does not fragment large messages and does not provide a demultiplexing mechanism. As a result, the user of LFC has to write code for fragmenting, reassembling,

and demultiplexing messages. This interface requires extra programming effort, but at the same time offers high performance to all clients. The most important functions of the interface are listed in Table 3.1.

3.1.1 Addressing

LFC's addressing scheme is simple. Each participating process is assigned a unique process identifier, a number in the range $[0 \ldots P-1]$, where P is the number of participating processes. This number is determined at application startup time and remains fixed during the application's lifetime. LFC maps process identifiers to host addresses and network routes.

3.1.2 Packets

LFC clients use packets to send and receive data. Packets have a maximum size (1 Kbyte by default); messages larger than this size must be fragmented by the client. LFC deliberately does not provide a higher-level abstraction that allows clients to construct messages of arbitrary sizes. Such abstractions impose overhead and often introduce a copy at the receiving side when data arrives in packet buffers that are not contiguous in memory. Clients that need only sequential access to message fragments can avoid this copy by using *stream messages* [90]. An efficient implementation of stream messages on top of LFC is described in Section 6.3. Even with stream messages, however, some overhead remains in the form of procedure calls, auxiliary data structures, and header fields that some clients simply do not need. Our implementation of CRL, for example, is constructed directly on LFC's packet interface, without intermediate message abstractions.

LFC distinguishes between *send packets* and *receive packets*. Send packets reside in NI memory. Clients can allocate a send packet and store data into it using normal memory writes (i.e., using programmed I/O). At the receiving side, LFC uses the DMA area approach described in Section 2.1, so receive packets reside in pinned host memory.

Both send and receive packets form a scarce resource, for which only a limited amount of memory is available. Send packets are stored in NI memory, which is typically small: the memory sizes of current NIs range from a few hundred kilobytes to a few megabytes. Receive buffers are stored in pinned host memory. In a multiprogramming environment, where user processes compete for CPU cycles and memory, most operating systems do not allow a single user to pin large amounts of memory.

void *lfc_send_alloc(int upcalls_allowed)	Allocates a send packet buffer in NI memory and returns a pointer to the packet's data part. Parameter *upcalls_allowed* has the same meaning in all functions. It indicates whether upcalls can be made when network draining occurs during execution of the function.
int lfc_group_create(unsigned *members, unsigned nmember)	Creates a multicast group and returns a group identifier. The process identifiers of all *nmembers* members are stored in *members*.
void lfc_ucast_launch(unsigned dest, void *pkt, unsigned size, int upcalls_allowed)	Transmits the first *size* bytes of send packet *pkt* to process *dest* and transfers ownership of the packet to LFC.
void lfc_bcast_launch(void *pkt, unsigned size, int upcalls_allowed)	Broadcasts the first *size* bytes of send packet *pkt* to all processes except the sender and transfers ownership of the packet to LFC.
void lfc_mcast_launch(int gid, void *pkt, unsigned size, int upcalls_allowed)	Multicasts the first *size* bytes of send packet *pkt* to all members of group *gid* and transfers ownership of the packet to LFC.
void lfc_poll(void)	Drains the network and invokes the client-supplied packet handler (*lfc_upcall*()) for each packet removed from the network.
void lfc_intr_disable(void) void lfc_intr_enable(void)	Disable and enable network interrupts. These functions must be called in pairs, with *lfc_intr_disable*() going first.
int lfc_upcall(unsigned src, void *pkt, unsigned size, lfc_upcall_flags_t flags)	*Lfc_upcall*() is the client-supplied function that LFC invokes once per incoming packet. *Pkt* points to the data just received (*size* bytes) and *flags* indicates whether or not *pkt* is a multicast packet. If *lfc_upcall*() returns nonzero, the client becomes the owner of *pkt* and must later release *pkt* using *lfc_packet_free*(). Otherwise, LFC recycles *pkt* immediately.
void lfc_packet_free(void *pkt)	Returns ownership of receive packet *pkt* to LFC.
unsigned lfc_fetch_and_add(unsigned var, int upcalls_allowed)	Atomically increments the F&A variable identified by *var* and returns the value of *var* before the increment.

Table 3.1. LFC's user interface. Only the essential functions are shown.

3.1.3 Sending Packets

LFC provides three send routines: *lfc_ucast_launch*() sends a point-to-point packet, *lfc_bcast_launch*() broadcasts a packet to all processes, and *lfc_mcast_launch*() sends a packet to all processes in a *multicast group*. Each multicast group contains a fixed subset of all processes. Multicast groups are created once and for all at initialization time, so processes cannot join or leave a multicast group dynamically. For reasons described in Appendix A, multicast groups may not overlap.

To send data, a client must perform three steps:

1. Allocate a send packet with *lfc_send_alloc*(). *Lfc_send_alloc*() returns a pointer to a free send packet in NI memory and transfers ownership of the packet from LFC to the client.

2. Fill the send packet, usually with a client-specific header and data.

3. Launch the packet with *lfc_ucast_launch*(), *lfc_mcast_launch*(), or *lfc_bcast_launch*(). These functions transmit the packet to one, multiple, or all destinations, respectively. LFC transmits as many bytes as indicated by the client. Ownership of the packet is transferred back to LFC. This implies that the allocation step (step 1) must be performed for each packet that is to be transmitted. No packet can be transmitted multiple times.

Steps 1 and 3 may block due to the finite number of send packet buffers and the finite length of the NI's command queue. To avoid deadlock, LFC continues to receive packets while it waits for resources (see Section 3.1.4).

The send procedure avoids intermediate copying: clients can gather data from various places (registers, different memory regions) and use programmed I/O to store that data directly into send packets. An alternative is to use DMA transfers, which consume fewer processor cycles and require fewer bus transfers. For message sizes up to 1 Kbyte, however, programmed I/O moves data from host memory to NI memory faster than the NI's DMA engine (see Figure 2.2). Furthermore, programmed I/O can easily move data from any virtual address in the client's address space to the NI, while DMA can be used only for pinned pages.

The send procedure separates packet allocation and packet transmission. This allows clients to hide the overhead of packet allocation. The CRL implementation on LFC, for example, allocates a new send packet immediately after it has sent a packet. Since the CRL runtime frequently waits for a reply message after sending a request message, it can often hide the cost of allocating a send packet.

3.1.4 Receiving Packets

LFC copies incoming network packets from the NI to receive packets in host memory. LFC delivers receive packets to the receiving process by means of an *upcall* [38]. An upcall is a function that is defined in one software layer and that is invoked from lower-level software layers. Upcalls are used to perform application-specific processing at times that cannot be predicted by the application. Since LFC does not know what a client aims to do with its incoming packets, the client must supply an upcall function. This function, named *lfc_upcall*(), is invoked by LFC each time a packet arrives; it transfers ownership of a receive packet from LFC to the client. Usually, this function either copies the packet or performs a computation using the packet's contents. (The computation can be as simple as incrementing a counter.) The upcall's return value determines whether or not ownership of the packet is returned to LFC. If the client keeps the packet, then the packet must later be released explicitly by calling *lfc_packet_free*(), otherwise LFC recycles the packet immediately. The main advantage of this interface is that it does not force the client to copy packets that cannot be processed immediately. Panda exploit this property in its implementation of stream messages (see Section 6.3).

Many systems based on active messages [148], invoke message handlers while draining the network. (Draining the network consists of moving packets from the network to host memory and is necessary to avoid congestion and deadlock.) The clients of such systems must be prepared to handle packets whenever draining can occur. Unfortunately, draining sometimes occurs at inconvenient times. Consider, for example, the case in which a message handler sends a message. If the system drains the network while sending this message, it may recursively activate the same message handler (for another incoming message). Since the system does not guarantee atomic handler execution, the programmer must protect global data against interleaved accesses by multiple handlers. Using a lock to turn the entire handler into a critical region leads to deadlock. The recursive invocation will block when it tries to enter the critical section occupied by the first invocation. The first invocation, however, cannot continue, because it is waiting for the recursive invocation to finish. Chapter 6 studies this upcall problem in more detail and compares several solutions.

LFC separates network draining and handler invocation [28]. Draining occurs whenever an LFC routine waits for resources (e.g., a free entry in the send queue), when the user polls, or when the NI generates an interrupt. During draining, however, LFC will invoke *lfc_upcall*() only if one of the following conditions is satisfied:

1. The client has not disabled network interrupts.

2. The client invokes *lfc_poll*().

3. The client invokes an LFC primitive with its *upcalls_allowed* parameter set to true (nonzero).

All LFC calls that potentially need to drain the network take an extra parameter named *upcalls_allowed*. If the client invokes a routine with *upcalls_allowed* set to false (zero), then LFC will drain the network if it needs to, but will not make any upcalls. If the parameter is true, then LFC will drain the network *and* make upcalls.

Separating draining and handler invocation is no panacea. If a client enables draining but disables upcalls, LFC must buffer incoming network packets. To prevent LFC from running out of receive packets the client must implement flow control. This is crucial, because there is nothing LFC, or any communication system, can do against clients that send an unbounded amount of data and do not consume that data. The system will either run out of memory or deadlock because it stops draining the network. Since LFC's current clients do not implement flow control —at least, not for all their communication primitives— they cannot disable upcalls and must therefore always be prepared to handle incoming packets.

With the current interface, clients cannot specify that the network must be drained asynchronously, using interrupts, without LFC making upcalls. If a process enters a phase in which it cannot process upcalls and in which it does not invoke LFC primitives, then the network will not be drained. The MPI implementation described in Section 7.4, for example, does not use interrupts. Draining occurs only during the invocation of LFC primitives. Since these primitives are invoked only when the application invokes an MPI primitive, there is no guarantee that a process will frequently drain the network. If a process sends a large message to a process engaged in a long-running computation —i.e., a process that does not drain— then LFC's internal flow-control mechanism (see Section 4.1) will eventually stall the sender, even if the receiver has space to store incoming packets.

3.1.5 Synchronization

LFC provides an efficient fetch-and-add (F&A) primitive [127]. A fetch-and-add operation fetches the current value of a logically shared integer variable and then increments this variable. Since these actions are performed atomically, processes can use the F&A primitive to synchronize their actions (e.g., to obtain the next free slot in a shared queue).

LFC stores F&A variables in NI memory. The implementation provides one F&A variable per NI and initializes all F&A variables to zero. There is no function

to set or reset F&A variables to a specific value, but such a function could easily be added.

Panda uses a single F&A variable to implement totally-ordered broadcasting (see Chapter 6). Totally-ordered broadcasting, in turn, is used by the Orca run-time system to update replicated objects in a consistent manner (see Section 7.1). Karamcheti et al. describe another interesting use of fetch-and-add in their paper on pull-based messaging [76]. In pull-based messaging senders do not push message data to receivers. Instead, a sender transmits only a message pointer to the receiver. The receiver pulls in the message data when it is ready to receive. Pull-based messaging reduces contention.

3.1.6 Statistics

Besides the functions listed in Table 3.1, LFC provides various statistics routines. LFC can be compiled such that it counts events such as packet transmissions, DMA transfers, etc. The statistics routines are used to reset, collect, and print statistics.

3.2 Key Implementation Assumptions

We have implemented LFC on the computer cluster described in Section 1.6. The implementation makes the following key assumptions:

1. The network hardware is reliable.
2. Multicast and broadcast are not supported in hardware.
3. Each host is equipped with an intelligent NI.
4. Interrupt processing is expensive.

Assumptions 1–3 pertain to network architecture. Among these three, the first assumption, reliable network hardware, is the most controversial. We assume that the network hardware neither drops nor corrupts network packets. With the exception of custom supercomputer networks (e.g., the CM-5 network [91]), however, network hardware is usually considered unreliable. This may be due to the network material (e.g., insufficient shielding from electrical interference) or to the network architecture (e.g., lack of flow control in the network switches).

The second assumption, no hardware multicast, is satisfied by most switched networks (e.g., ATM), but not by token rings and bus-based networks (e.g., Ethernet). We assume that multicast and broadcast services must be implemented in software. Efficient software multicast schemes use spanning tree protocols in

which network nodes forward the messages they receive to their children in the spanning tree (see Section 2.7).

The third assumption, the presence of an intelligent NI, enables the execution of nontrivial protocol code on the NI. Intelligent NIs are used both in supercomputers (e.g., FLASH [85] and the IBM SP/2 [129]) and in cluster architectures (e.g., Myrinet and ATM clusters). NI-level protocols form an important part of LFC's implementation (see Chapter 4).

The fourth assumption, expensive interrupts, is related to host-processor architecture and operating system implementations. Modern processors have deep pipelines, which must be flushed when an interrupt is raised. In addition, commercial operating systems do not dispatch interrupts efficiently to user processes [143]. We therefore assume that interrupt processing is slow and that interrupts should be used only as a last resort.

3.3 Implementation Overview

LFC consists of a library that implements LFC's interface and a *control program* that runs on the NI and takes care of packet transmission, receipt, forwarding, and flow control. The control program implements the NI-level protocols described in Chapter 4. In addition, several device drivers are used to obtain efficient access to the network device. The library, the NI control program, and the device drivers are all written in C.

3.3.1 Myrinet

Myrinet implements hardware flow control on the network links between communicating NIs and has a very low bit-error rate [27]. If all NIs use a single, deadlock-free routing algorithm and are always ready to process incoming packets, then Myrinet will not drop any packets. In a single administrative domain of modest size, such as our Myrinet cluster, these conditions can be satisfied. Consequently, LFC assumes that Myrinet neither drops nor corrupts packets. The limitations of this approach are discussed in more detail in Section 3.9.

3.3.2 Operating System Extensions

All cluster nodes run the Linux operating system (RedHat distribution 5.2, kernel version 2.0.36). To support user-level communication, we added several device drivers to the operating system.

Myricom's Myrinet device driver was modified so that at most one user at a time can open the Myrinet network device. When a user process opens the

device, the driver maps the NI's memory contiguously into the user's address space, so the user process can read and write NI memory using normal load and store instructions (i.e., programmed I/O). The driver also processes all interrupts generated by the NI. Each time the driver receives an interrupt, it sends a SIGIO software signal to the process that opened the device.

At initialization time, LFC's library allocates a fixed number of receive packets from the client's heap. The pages on which these packets are stored are pinned by means of the *mlock*() system call. The client can specify the number of receive packets that the library is to allocate. To transfer data from its memory into the receive packets in host memory, the NI's control program needs the *physical* addresses of the receive packets. To obtain these addresses, we implemented a small pseudo device driver that translates a page's virtual address to the corresponding physical address. (A pseudo device driver is a kernel module with a device driver interface but without associated hardware.) LFC's library invokes this driver at initialization time to compute each packet's physical address.

Writes to device memory (e.g., NI memory) are normally not cached by the writing processor, because the data written to device memory is unlikely to be read back again by the processor. Moreover, in the case of write-back caching, the device will not observe the writes until they happen to be flushed from the cache. By disabling caching for device memory, however, performance is reduced because a bus transaction must be set up for each word that is written to the device. Write-back caches, in contrast, write complete cache lines to memory, which is beneficial when a large contiguous chunk of data is written.

To speed up writes to NI memory, LFC uses the Pentium Pro's ability to set the caching properties of specific virtual memory ranges [71]. A small pseudo device driver enables *write combining* for the virtual memory range to which the NI's memory is mapped. The Pentium Pro combines all writes to a write-combining memory region in on-processor write buffers. Writes are delayed until the write buffer is full or until a serializing instruction is issued. By aggregating writes to NI memory, the processor can transfer these writes in larger bursts to the I/O bus, which reduces the number of I/O bus arbitrations.

As discussed in Section 2.2, the use of write combining considerably speeds up programmed I/O data transfers from host memory to NI memory. The main disadvantage of applying write combining to a memory region is that writes to that region may be reordered. Two sucessive writes are executed in program order only if they are separated by a serializing instruction. When necessary, we use the Pentium Pro's atomic increment instruction to order writes.

Type	Resource	Description
Hardware	Send DMA engine	Transfers packets from NI memory to the network.
	Receive DMA engine	Transfers packets from the network to NI memory.
	Host/NI DMA engine	Mainly used to transfer packets from NI memory to host memory. Also used to retrieve and update status information.
Software	Send credits	A send credit represents a data packet buffer in some NI's memory. For each destination NI, there is a separate resource queue.
	Host receive buffer	A receive buffer in host memory.

Table 3.2. Resources used by the NI control program.

3.3.3 Library

LFC's library implements all routines listed in Table 3.1. The library manages send and receive buffers, communicates with the NI control program, and delivers packets to clients. Host-NI communication takes place through shared variables in NI memory, through DMA transfers between host and NI memory, and through signals generated by the kernel in response to NI interrupts.

3.3.4 NI Control Program

The main task of the NI control program is to send and receive packets. Outgoing and incoming packets use various hardware and software resources as they travel through the NI. The control program acquires and activates these resources in the right order for each packet. If the control program cannot acquire the next resource that a packet needs, then it queues the packet on a resource-specific (software) *resource queue*. All resources listed in Table 3.2, except the host receive buffers, have an associated resource queue.

Three hardware resources are available for packet transfers: the send DMA engine, the receive DMA engine, and the host/NI DMA engine. These engines operate asynchronously: typically, the NI processor starts a DMA transfer and continues with other work, periodically checking if the DMA engine has finished its transfer. (Alternatively, a DMA engine can generate an interrupt when it has completed a transfer. LFC's control program, however, does not use interrupts.)

Software resources are used to implement flow control, both between communicating NIs and between an NI and its host. Send credits represent buffer space

in some NI's memory and are used to implement NI-to-NI flow control (see Section 4.1). Before an NI can send a data packet to another NI, it must acquire a send credit for that NI. Similarly, before an NI can copy a data packet to host memory, it must obtain a host buffer. Host buffer management is described in Section 3.6.

To utilize all resources efficiently, LFC's control program is structured around resources rather than packets. The program's main loop polls several work queues and checks if resources are idle. When the control program finds a task on one of its work queues, it executes the task up to the point where the task needs a resource that is not available. The task is then appended to the appropriate resource queue. When the control program finds that a resource is idle, it checks the resource's queue for new work. If the queue is nonempty, a task is dequeued and the resource is put back to use again.

The resource-centric structure optimizes resource utilization. In particular, all DMA engines can be active simultaneously. By allowing the receive DMA of one packet to proceed concurrently with the host DMA of another packet, we pipeline packet transfers and obtain better throughput. A packet-centric program structure would guide a single packet completely through the NI before starting to work on another packet. With this approach, the control program uses at most one DMA engine at a time, leaving the other DMA engines idle.

The resource-centric approach increases latency, because packets are enqueued and dequeued each time they use some resource. To attack this problem, LFC's control program contains a fast path at the sending side, which skips all queueing operations if all resources that a packet needs are available. We also experimented with a receiver-side fast path, but found that the latency gains with this extra fast path were small. To avoid code duplication, we therefore removed this fast path.

The control program acts in response to three event types:

1. send requests
2. DMA transfer completions
3. timer expiration

The control program's main loop polls for these events and processes them.

Send requests are created by the host library in response to a client invocation of one of the packet launch routines. Each send request is stored (using programmed I/O) in a queue in NI memory. The control program's main loop polls this queue for new requests.

DMA transfers are used to send and receive packets and to move received packets to host memory. DMA transfer completion is signaled by means of bits in the NI's interrupt status register. When a packet transfer completes, the control program moves the packet to its next processing stage and checks if it can put

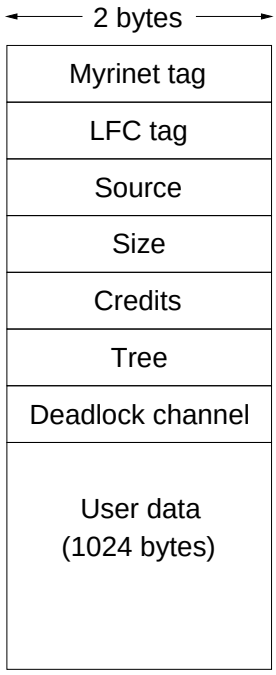


Fig. 3.1. LFC's packet format.

the DMA engine to work again. The control program always restarts the receive DMA engine so that it is always prepared for incoming packets.

Myrinet's NI contains a timer with a granularity of 0.5 μs. The control program uses this timer to delay interrupts (see Section 4.4). Timer expiration is signaled by means of certain bits in the NI's interrupt status register.

3.4 Packet Anatomy

LFC uses several packet types to carry user and control data across the network. As shown in Figure 3.1, each packet consists of a 12-byte header and a 1024-byte *user data* buffer. The buffer stores the data that a client wishes to transfer. LFC does not interpret the contents of this buffer. When LFC allocates a send packet (in *lfc_send_alloc*()) or passes a receive packet to *lfc_upcall*(), the client receives a pointer to the data buffer. Clients must neither read nor write the packet header, but this is not enforced.

The header's tag field consists of a 2-byte Myrinet tag. Every Myrinet communication system can obtain its own tag range from Myricom. This tag range is different from the tag ranges assigned to other registered Myrinet communication systems. NIs can use the Myrinet tag to recognize and discard misrouted pack-

Packet class	LFC tag	Function
Control	CREDIT	Explicit credit update
	CHANNEL_CLEAR	Deadlock recovery
	FA_REPLY	Fetch-and-add reply
Data	UCAST	Unicast packet
	MCAST	Multicast packet
	FA_REQUEST	Fetch-and-add request

Table 3.3. LFC's packet tags.

ets from different communication systems. All LFC packets carry Myrinet tag 0x0450.

LFC uses an additional tag field to identify different types of LFC packets. Unicast and multicast packets, for example, use different tags, because the control program treats unicast and multicast packets differently; multicast packets are forwarded, whereas unicast packets are not. Table 3.3 lists the most important packet tags.

We distinguish between control packets and data packets. Control packets require only a small amount of NI-level processing. Unlike data packets, control packets are never queued; when the NI receives a control packet, it processes it immediately and then releases the buffer in which the packet arrived. Two packet buffers suffice to receive and process all control packets. A second buffer is needed because the control program always restarts the receive DMA engine (with a free packet buffer as the destination address) before it processes the packet just received.

Data packets go through more processing stages and use more resources. Each UNICAST packet, for example, needs to be copied to host memory, but this can be done only when a free host buffer and the DMA engine are available. When either resource is unavailable, the control program queues the packet and starts working on another packet. Since some resources (e.g., host buffers) may be unavailable for an unpredictable amount of time, the control program may have to buffer many data packets. The total number of packets that needs to be buffered is bounded by LFC's flow control protocol, which stalls senders when receiving NIs run out of free packet buffers.

The *source* field in the packet header identifies the sender of a packet. At LFC initialization time, the client passes to each LFC process the number of participating processes and assigns a unique process identifier to each process. Since LFC allows only one process per processor, this process identifier also identifies the processor and the NI. Each time an NI transmits a packet, LFC stores the NI's identifier in the *source* field.

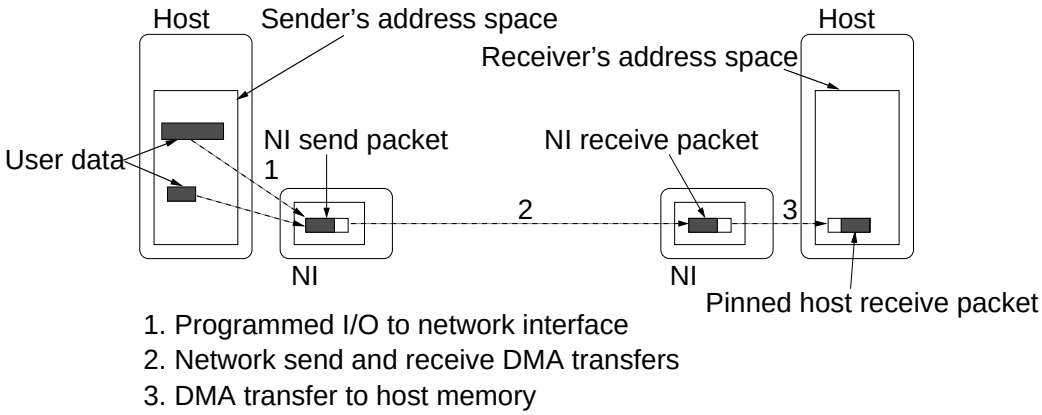


Fig. 3.2. LFC's data transfer path.

The *user data* part of send packets has a maximum size of 1024 bytes, but clients need not fill the entire data part. The *size* field specifies how many bytes have been written into the user data part of a packet. These bytes are called the valid user bytes and they must be stored contiguously at the beginning of a packet. When LFC transfers a packet, it transfers the packet header and the valid user bytes, but *not* the unused bytes at the end of the packet. This yields low latency for small messages and high throughput for large messages.

The remaining header fields are described in detail in Chapter 4. The *credits* field is used to piggyback flow control information to the packet's destination. The *tree* field identifies multicast trees. The *deadlock channel* field is used during deadlock recovery.

LFC's outgoing packets are tagged (in hardware) with a CRC that is checked (in software) by LFC's control program at the receiving side. CRC errors are treated as catastrophic and cause LFC to abort. Lost packets are not detected.

3.5 Data Transfer

LFC transfers data in packets which are allocated from three packet buffer pools. Each NI maintains a send buffer pool and a receive buffer pool. Hosts maintain only a receive buffer pool; all buffers in this pool are stored in pinned memory. The data transfer path in Figure 3.2 for unicast packets shows how these three pools are used.

At the sending side, the client allocates send packets from the send buffer pool in NI memory (transfer 1 in Figure 3.2). Programmed I/O is used to copy client data directly, without operating system intervention, into a send packet. The client

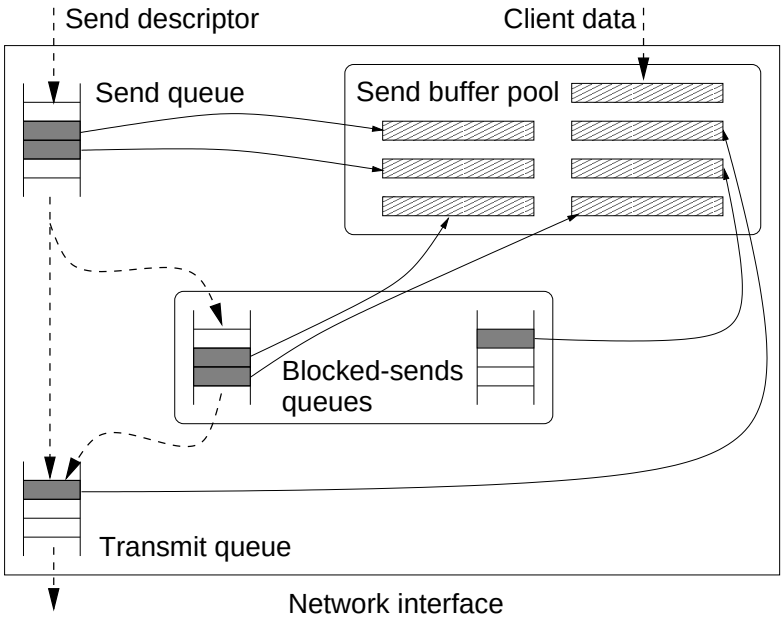


Fig. 3.3. LFC's send path and data structures.

calls one of LFC's launch routines, which writes a send descriptor into the NI's *send queue* (see Figure 3.3). The send descriptor contains the packet's destination and a reference to the packet. The NI, which polls the send queue, inspects the send descriptor. When credits are available for the specified destination, the descriptor is moved to the *transmit queue*; the packet will be transmitted when the descriptor reaches the head of this queue (transfer 2 in Figure 3.2). After the packet has been transmitted, it is returned to the NI's send packet pool.

When no send credits are available, the descriptor is added to a *blocked-sends queue* associated with the packet's destination. When credits flow back from some destination, descriptors are moved from that destination's blocked-sends queue to the transmit queue. The details of the credit algorithm are described in Chapter 4.

The destination NI receives the incoming packet into a free packet buffer allocated from its receive buffer pool. This pool is partitioned evenly among all NIs: each NI owns a fixed number of the buffers in the receive pool and can fill these buffers —and no more— by sending packets. Each buffer corresponds to one send credit and filling a buffer corresponds to consuming a send credit.

Each host is responsible for passing the addresses of free host receive buffers to its NI. These addresses are stored in a shared queue in NI memory (see Section 3.6). As soon as a free host receive buffer is available, the NI copies the packet to host memory (transfer 3 in Figure 3.2). After this copy, the NI receive buffer is returned to its pool. When no host receive buffers are available, the NI

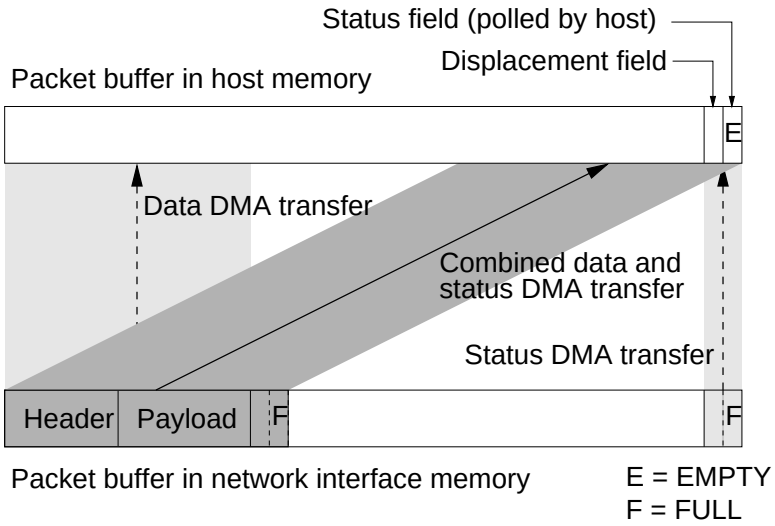


Fig. 3.4. Two ways to transfer the payload and status flag of a network packet. The dashed arrows illustrate a simple, but expensive manner. The solid arrow indicates the optimized transfer.

simply does not copy packets to the host and does not release any of its receive buffers. Eventually, senders will be stalled and remain stalled until the receiving host posts new buffers.

Each host receive buffer contains a status field that indicates whether the control program has copied data into the buffer. The host uses this field to detect incoming messages. Copying a packet from an NI receive buffer to a host receive buffer involves two actions: copying the packet's payload to host memory and setting the status field in the host buffer to FULL. For efficiency, LFC folds both actions into a single DMA transfer. This is illustrated in Figure 3.4, which also shows a naive implementation that uses two DMA transfers to copy the packet's payload and status field. To avoid transferring the unused part of the receive packet's data field, the status field must be stored right after the packet's payload. (If it were stored in front of the data, the host could believe it received a packet before all of that packet's data had arrived.) The position of the FULL word therefore depends on the size of the packet's payload. The host, however, needs to know in advance where to poll for the status field. The control program therefore transfers the packet's payload and the FULL word to the end of the host receive buffer. In addition, the control program transfers a word that holds the payload's displacement relative to the beginning of the host's receive buffer.

Once filled, the host buffer is passed to the client-supplied upcall routine. The client decides when it releases this buffer. Once released, the buffer is returned to the host receive buffer pool.

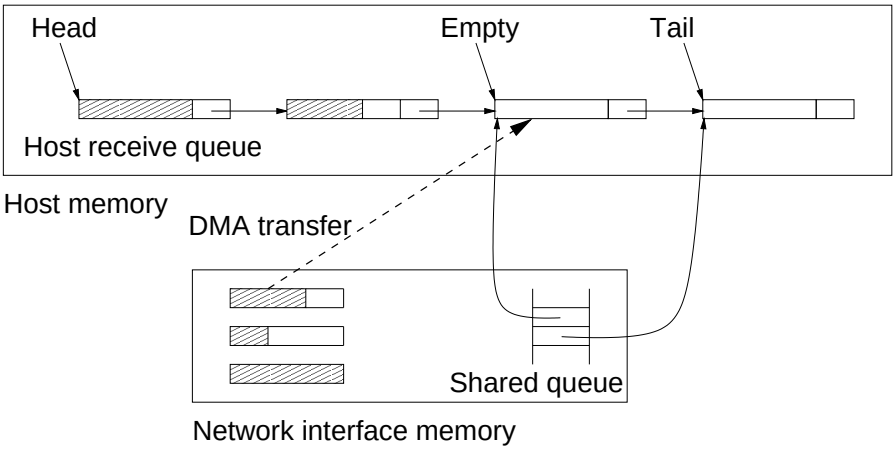


Fig. 3.5. LFC's receive data structures.

Multicast packets follow the same path as unicast packets, but may additionally be forwarded by receiving NIs to other destinations. An NI receive buffer that contains a multicast packet is not returned to its pool until the multicast packet has been forwarded to all forwarding destinations.

3.6 Host Receive Buffer Management

Each host has a pool of free host receive buffers. The client specifies the initial number of buffers in this pool. At initialization time, the library obtains the physical[1] address of each buffer in the pool and stores this address in the buffer's descriptor.

Besides the pool, the library manages a *receive queue* of host receive buffers (see Figure 3.5). The library maintains three queue pointers. *Head* points to the first FULL, but unprocessed packet buffer. This is the next buffer that will be passed to *lfc_upcall*(). *Empty* points to the first EMPTY receive buffer. This is the next buffer that the library expects to be filled by the NI. *Tail* points to the last buffer in the queue. When the library allocates new receive buffers from the pool, it appends them to the receive queue and advances *tail*.

Each time the library appends a buffer to its receive queue, it also appends the physical address of the buffer to a shared queue in NI memory. When the NI control program receives a network packet, it tries to fetch an address from

[1]To be precise, we obtain the *bus* address. This is a device's address for a particular host memory location. On some architectures, this address can be different from the physical address used by the host's CPU, but on Intel's x86 architecture the bus address and the physical address are identical.

the shared queue and copy the packet to this address. When the shared queue is empty, the control program defers the transfer until the queue is refilled by the host library.

It is not necessary to separate the receive queue and the pool, but doing so often reduces the memory footprint of the host receive buffers. In most cases, we keep only a small number of buffers on the receive queue. As long as the client is able to use and reuse these buffers, only a small portion of the processor's data cache is polluted by host receive buffers. In some cases, however, clients need more buffers, which are then allocated from the pool and added to the receive queue. In those cases, a larger portion of the cache will be polluted by host receive buffers.

When the library needs to drain (see Section 3.1.4), it performs two actions:

1. If the packet pointed to by *empty* has been filled, the library advances *empty*. If *empty* cannot be advanced (because all packets are full) *and* upcalls are not allowed, then the library appends a new buffer from the pool to both queues.

2. If *head* does not equal *empty* and upcalls are allowed, the library dequeues *head* and calls *lfc_upcall*(), passing *head* as a parameter. *Head* is then advanced. If there are no buffers left in the receive queue, then the library appends a new buffer from the pool to both queues before calling *lfc_upcall*().

Packets freed by clients are appended to both queues, unless the queue already contains a sufficient number of empty packets. In the latter case, the packet is returned to the buffer pool.

3.7 Message Detection and Handler Invocation

LFC delivers packets to a receiving process by invoking *lfc_upcall*(). This routine is either invoked sychronously, in the context of a client call to a draining LFC routine, or asynchronously, in the context of a signal handler. Here, we discuss the implementation of both mechanisms.

Each invocation of *lfc_poll*() checks if the control program has copied a new packet to host memory. If *lfc_poll*() finds a new packet, it invokes *lfc_upcall*(), passing the packet as a parameter. To allow fair scheduling of the client computation and the packet handler, *lfc_poll*() delivers at most one network packet.

The check for the next packet is implemented by reading the status field of the next undelivered packet buffer (*empty* in Figure 3.5). When this buffer's address was passed to the control program, its status field is set to EMPTY. The poll succeeds when the host detects that this field has been set to FULL. This polling

strategy is efficient, because the status field resides in host memory and will be cached (see Section 2.5).

Asynchronous packet delivery is implemented cooperatively by LFC's control program, the Myrinet device driver, and LFC's library. In principle, LFC's control program generates an interrupt after it has copied a packet to host memory. However, since the receiving process may poll, the control program delays its interrupts. Myrinet NIs contain an on-board timer with 0.5 μs granularity. When a packet arrives, the control program sets this timer to a user-specified timeout value (unless the timer is already running). An interrupt is generated only if the host fails to process a packet before the timer expires. The details of this *polling watchdog* mechanism are described in Chapter 4.

Interrupts generated by the control program are received by the operating system kernel, which passes them to the Myrinet device driver. The driver transforms the interrupt into a user-level software signal and sends this *network signal* to the client process. These signals are normally caught by the LFC library. The library's signal handler checks if the client is running with interrupts enabled. If this is the case, the signal handler invokes *lfc_poll*() to process pending packets; otherwise it returns immediately.

Recall that clients may dynamically disable and enable network interrupts. For efficiency, the interrupt status manipulation functions are implemented by toggling an interrupt status flag in host memory, without any communication or synchronization with the control program. Before sending an interrupt, the control program fetches (using a DMA transfer) the interrupt status flag from host memory. No interrupt is generated when the flag indicates that the client disabled interrupts.

Since the library does not synchronize with the control program when it manipulates the interrupt status flag, two races can occur:

1. The control program may raise an interrupt just after the client disabled interrupts. Unless precautions are taken, the signal handler will invoke the packet handler even though the client explicitly disabled interrupts. To solve this problem, LFC's signal handler always checks the interrupt status flag before invoking *lfc_poll*() and ignores the signal if the race occurred. In practice, this race does not occur very often, so the advantage of inexpensive interrupt manipulation outweighs the cost of spurious interrupts.

2. The control program may decide not to raise an interrupt just after the client (re-)enabled interrupts. This race is fatal for clients that rely on interrupts for correctness. To avoid lost interrupts, the control program continues to generate periodic interrupts for a packet until the host has processed that packet (see Chapter 4).

3.8 Fetch-and-Add

The F&A implementation is straightforward. Every NI stores one F&A variable, which is identified by the *var* argument in *lfc_fetch_and_add*() (see Table 3.1). *Lfc_fetch_and_add*() performs a remote procedure call to the NI that stores the F&A variable. The process that invoked *lfc_fetch_and_add*() allocates a request packet, tags it with tag FA_REQUEST, and appends it to the NI's host send queue.

When the destination NI receives the request, it increments its F&A variable and stores the previous value in a reply packet. The reply packet, a control packet tagged FA_REPLY, is then appended to the NI's transmit queue.

The implementation assumes that each participating process can issue at most one F&A request at a time. To allow multiple outstanding requests, we need a form of flow control that prevents the NI that processes those requests from running out of memory as it creates and queues FA_REPLY packets. FA_REQUEST packets consume credits, but those credits are returned as soon as the request has been received and possibly before a reply has been generated.

When the reply arrives at the requesting NI, the control program copies the F&A result from the reply to a location in NI memory that is polled by *lfc_fetch_-and_add*(). When *lfc_fetch_and_add*() finds the result, it returns it to the client.

This implementation is simple and efficient. The FA_REQUEST packet is handled entirely by the receiving NI, without any involvement of the host to which the NI is attached. This is particularly important when processes issue many F&A requests. Many of these requests would either be delayed or generate interrupts if they had to be handled by the host.

The implementation uses a data packet for the request and a control packet for the reply. The request is a data packet only because the current implementation of LFC does not allow hosts to send control packets. Although this is easy to add, it will increase the send overhead for *all* packets, because the NI will have to check if the host sends a data or a control packet.

3.9 Limitations

The implementation of LFC on Myrinet makes a number of simplifying assumptions, some of which limit its applicability in a production environment. We distinguish between functional limitations and security violations.

3.9.1 Functional Limitations

LFC's functional limitations are all related to the assumption that LFC will be used in a homogeneous, dedicated computing environment.

Homogeneous Hosts

LFC assumes that all host processors use the same byte order. Adding byte swapping for packet headers is straightforward and will add little overhead. (It is the client's responsibility to convert the *data* contained in network packets; LFC does not know what type of data clients store in their packets.) Since our cluster's host processors are little-endian and the NI is big-endian, some byte swapping is already used for data that needs to be interpreted both by the host processor and the NI. This concerns mainly shared descriptors and packet headers; user data contained in packets is not interpreted by LFC and is never byte-swapped.

Device Sharing

LFC can service at most one user process per host, mainly because LFC's control program does not separate the packets from different users. In our current environment (DAS), the Myrinet device driver returns an error if a second user tries to obtain access to the Myrinet device.

Multiple users can be supported by giving different users access to different parts of the NI's memory. This can be achieved by setting up appropriate page mappings between the user's virtual address space and the NI's memory. If this is done, the NI's control program must poll multiple command queues located on different memory pages, which is likely to have some impact on performance.

Newer Myrinet network interfaces provide a *doorbell* mechanism. This mechanism detects host processor writes to certain regions and appends the target address and value written to a FIFO queue. With this doorbell mechanism, the NI need not poll all the pages that different processes can write to; it need only poll the FIFO queue.

Recall that LFC partitions the available receive buffer space in NI memory among all participating nodes. If the NI's memory is also partitioned to support multiprogramming, then the available buffer space per process will decrease noticeably. Each process will be given fewer send credits per destination process, which reduces performance. A simple, but expensive solution is to buy more memory. However, this may only be possible to a limited extent. A less expensive and more scalable solution is to employ swapping techniques such as described in Section 2.4.

Fixed Process Configuration

LFC assumes that all of an application's processes are known at application startup time; users cannot add or remove processes to the initial set of processes. The main obstacle to adding processes dynamically is that LFC evenly divides all NI receive buffers (or rather, the send credits that represent them) among the initial

set of processes. When a process is added, send credits must be reallocated in a safe way.

Reliable Network

LFC assumes that the network hardware never drops or corrupts packets. While LFC tests for CRC errors in incoming packets, it does not recover from such errors. Lost packets are not detected at all.

A retransmission mechanism would allow an LFC application to tolerate transient network failures. In addition, retransmission allows communication protocols to drop packets when they run out of resources. With retransmission, LFC would not need to partition NI receive buffers among all senders. Chapter 8 considers alternative implementations which do retransmit dropped and corrupted packets.

3.9.2 Security Violations

LFC does not limit a user process's access to the NI and is therefore not secure. Any user process with NI access can crash any other process or even the kernel. With a little kernel support, though, LFC can be made secure; the techniques to achieve secure user-level communication are well-known. Moreover, LFC's send and receive methods require relatively simple and inexpensive security mechanisms. Below, we describe the security problems in more detail. Where necessary, we also provide solutions. These solutions have not been implemented.

Network Interface Access

LFC (or rather, the Myrinet device driver used by LFC) allows an arbitrary process to map all NI memory into its address space and does not restrict access to this memory. As a result, users can freely modify the NI control program. Once modified, the control program can both read and write *all* host memory locations.

Access to the NI can be restricted by mapping only part of the NI's memory into a process's address space. The process is not given any access to the control program or to another process's pages. LFC can easily be modified to work with a device driver that implements this type of protection and the expected the performance impact is small. The page mappings can be set up at application initialization time so that the cost of setting up these mappings is not incurred on the critical communication path.

Address Translation

The LFC library passes the physical memory addresses of free host receive buffers to the control program. Since the control program does not check if these addresses are valid, a malicious program can pass an illegal address, which the control program will use as the destination of a DMA transfer.

A simple solution is to have all host receive buffers allocated by a trusted kernel module. This module assigns to each buffer an index (a small integer) and maps this index to the buffer's physical address. The mapping is passed to the NI, but not to the user process. The user process, i.e., the LFC library, is only given each buffer's virtual address and its index. Instead of passing a buffer's physical address to the NI, the library will pass the buffer's index. The NI control program can easily check if the index it receives is valid. This way, DMA transfers to host memory can only target the buffers allocated by the trusted kernel module.

Another solution is to use a secure address translation mechanism such as VMMC-2's user-managed TLB [49] (see Section 2.3). This scheme is more involved, but supports dynamic pinning and unpinning of any page in the user's address space, whereas the scheme above would pin all host receive buffers once and for all.

Foreign Packets

LFC's control program makes no serious attempt to reject network packets that originate from another application. LFC rejects only packets that do not carry LFC's Myrinet tag or that carry an out-of-range LFC tag. Consequently, malicious or faulty programs can send a packet to any other application as long as the packet carries tags known by LFC. The Hamlyn system [32] provides each parallel application's process with a shared random bit pattern that is appended to each message and verified by the receiver. The bit pattern is sufficiently long that an exhaustive search for its value is computationally infeasible.

3.10 Related Work

The design of LFC was motivated in part by the shortcomings of existing systems. Since many of these systems were discussed in Chapter 2, we will not treat them all again here.

LFC uses *optimistic interrupt protection* [134] to achieve atomicity with respect to network signals. This technique was used in the Mach operating system kernel to reduce the overhead of hardware interrupt-mask manipulation. With this technique, interrupt-masking routines do not truly disable interrupts, but manipulate a software flag that represents the current interrupt mask. It is the responsi-

bility of the interrupt handler to check this flag. If the flag indicates that interrupts are not allowed, the interrupt handler creates an *interrupt continuation*, some state that allows the interrupt-handling code to be executed at a later time. LFC uses the same technique, but does not need to create interrupt continuations, because the NI will automatically regenerate the interrupt.

Some systems emulate asynchronous notification using a background thread that periodically polls the network [54]. The problem is to find a good polling frequency. Also, adding a thread to an otherwise single-threaded software system is often difficult.

A similar idea is used in the Message-Passing Library (MPL) for the IBM SP/2. This library uses periodic timer interrupts to allow the network to be drained even if senders and receivers are engaged in long-running computations [129]. The main difference is that MPL's timeout handler does not invoke user code, because MPL does not provide an asynchronous notification interface.

Another emulation approach is to use a compiler or binary rewriter to insert polls automatically into applications. Shasta [125], a fine-grain DSM, uses binary instrumentation. In a simulation study [114], Perkovic and Keleher compare automatic insertion of polls by a binary rewriter and the polling-watchdog approach. Their simulation results, performed in the context of the CVM DSM, indicate that both approaches work well and perform better than an approach that relies exclusively on polling during message sends.

3.11 Summary

In this chapter, we have presented the interface and implementation of LFC, a new user-level communication system designed to support the development of parallel programming systems. LFC provides reliable point-to-point and multicast communication, interrupt management, and synchronization through fetch-and-add. LFC's interface is low-level. In particular, there is no message abstraction: clients operate on send and receive packets.

This chapter has described the key components of LFC's implementation on Myrinet (library, device drivers, and NI control program). We have described LFC's data transfer path, buffer management, control transfer, and synchronization. The NI-level communication protocols, which form an important part of the implementation, are described in the next chapter.

Chapter 4

Core Protocols for Intelligent Network Interfaces

This chapter gives a detailed description of LFC's NI-level protocols: UCAST, MCAST, RECOV, and INTR. The UCAST protocol implements reliable point-to-point communication between NIs. UCAST assumes reliable network hardware and preserves this reliability using software flow control implemented at the data link level, between NIs. LFC is named after this property: *L*ink-level *F*low *C*ontrol.

Link-level flow control also forms the basis of MCAST, an NI-level spanning tree multicast protocol. MCAST forwards multicast packets on the NI rather than on the host; this prevents unnecessary reinjection of packets into the network and removes host processing —in particular interrupt processing and copying— from the critical multicast path.

MCAST works for a limited, but widely used class of multicast trees (e.g., binary and binomial trees). For trees not in this class, MCAST may deadlock. The RECOV protocol extends MCAST with a deadlock recovery protocol that allows the use of arbitrary multicast trees.

UCAST, MCAST, and RECOV are concerned with data transfer between NIs. The last protocol described in this chapter, INTR, deals with NI-to-host control transfer. INTR reduces the overhead of network interrupts by delaying network interrupts.

This chapter is structured as follows. Sections 4.1 through 4.4 discuss, respectively, UCAST, MCAST, RECOV, and INTR. Section 4.5 discusses related work.

4.1 UCAST: Reliable Point-to-Point Communication

The UCAST protocol implements reliable and FIFO point-to-point transfers of *packets* between NIs. Since we assume that the hardware does not drop or corrupt packets (see Section 3.2), the only cause of packet loss is lack of buffer space on receiving nodes (both NIs and hosts). This chapter deals only with NI-level buffer overflow; host-level buffering was discussed in Section 3.6.

UCAST uses a variant of sliding window flow control between each pair of NIs to prevent NI-level buffer overflows. The protocol guarantees that no NI will send a packet to another NI before it knows the receiving NI can store the packet. To achieve this, each NI assigns an equal number of its receive buffers to each NI in the system. An NI S that needs to transmit a packet to another NI R may do so only when it has at least one *send credit* for R. Each send credit corresponds to one of R's free receive buffers for S. Each time S sends a packet to R, S consumes one of its send credits for R. Once S has consumed all its send credits for R, S must wait until R frees some of its receive buffers for S and returns new send credits to S. Send credits are returned by means of explicit acknowledgements or by piggybacking them on application-level return traffic.

4.1.1 Protocol Data Structures

Figure 4.1 shows the types and variables used by UCAST. All variables are be stored in NI memory. UCAST uses two types of network packets: UNICAST packets carry user data; CREDIT packets are used only to return credits to a sender and do not carry any data. Each packet carries a header of type *PacketType* that identifies the packet's type (*tag*) and sender (*src*). In addition, the protocol uses a *credits* field in the header to return send credits to an NI.

Transmission requests are stored in send descriptors of type *SendDesc* that identify the packet to be transmitted (*packet*) and the destination NI (*dest*). Multiple send descriptors can refer to the same packet; this is important for the multicast protocol, which can forward a single packet to multiple destinations.

Each NI maintains protocol state for each other NI. This state is stored in a protocol control block of type *ProtocolCtrlBlock*. The protocol state for a given NI consists of the number of remaining send credits for that NI (*send_credits*), a queue of blocked transmission requests (*blocked_sends*), and the number of credits that can be returned to that NI (*free_credits*).

Finally, each NI maintains several global variables. Each NI stores its unique id in *LOCAL_ID*, the number of NIs in the system in *NR_NODES*, and a credit refresh threshold in *CREDIT_REFRESH*. The refresh threshold is used by receivers to determine when credits must be returned to the sender that consumed them. The protocol control blocks are stored in array *pcbtab*. Variable *nr_received* counts

```
typedef enum {UNICAST, CREDIT} PacketType;

typedef struct {
        PacketType tag;     /* packet type */
        unsigned src;     /* sender of packet */
        unsigned credits;     /* piggybacked credits */
} PacketHeader;

typedef struct {
        PacketHeader hdr;
        Byte data[PACKET_SIZE];
} Packet;

typedef struct {
        unsigned dest;
        Packet *packet;
} SendDesc;

typedef struct {
        unsigned send_credits;
        SendDescQueue blocked_sends;
        unsigned free_credits;
} ProtocolCtrlBlock;

unsigned LOCAL_ID, NR_NODES, CREDIT_REFRESH;     /* runtime constants */
ProtocolCtrlBlock pcbtab[MAX_NODES];     /* per-node protocol state */
unsigned nr_pkts_received;
```

Fig. 4.1. Protocol data structures for UCAST.

```
void event_unicast(unsigned dest, Packet *pkt) {
        SendDesc *desc = alloc_send_desc();

        pkt→hdr.tag = UNICAST;
        desc→dest = dest;
        desc→packet = pkt;
        credit_send(desc);
}

void credit_send(SendDesc *desc) {
        ProtocolCtrlBlock *pcb = &pcbtab[desc→dest];

        if (pcb→send_credits > 0) {
                pcb→send_credits−−;     /* consume a credit */
                send_packet(desc);
                return;
        }
        enqueue(&pcb→blocked_sends, desc);     /* wait for a credit */
}

void send_packet(SendDesc *desc) {
        ProtocolCtrlBlock *pcb = &pcbtab[desc→dest];
        Packet *pkt = desc→packet;

        pkt→hdr.src = LOCAL_ID;
        pkt→hdr.credits = pcb→free_credits;     /* piggyback credits */
        pcb→free_credits = 0;

        transmit(desc→dest, pkt);
}
```

Fig. 4.2. Sender side of the UCAST protocol.

how many packets an NI has received; this variable is used by the INTR protocol (see Section 4.4).

4.1.2 Sender-Side Protocol

Figure 4.2 shows the reliability protocol executed by each sending NI. All protocol code is event-driven; this part of the protocol is activated in response to *unicast* events which are generated when the NI's host launches a packet. How LFC generates this event was discussed in Chapter 3.

Each unicast event is dispatched to *event_unicast*(). This routine creates a send descriptor and passes this descriptor to *credit_send*(), which checks if a send credit is available for the destination specified in the send descriptor. If this is the case,

the packet is transmitted; otherwise the send descriptor is queued on the blocked-sends queue for the destination NI. It will remain queued until the destination NI has returned a sufficient number of send credits.

Packets are transmitted by *send_packet*(). This routine is also used to return send credits to the destination NI. In the protocol control block of each NI we count how many credits can be returned to that NI; *send_packet*() copies the counter into the *credits* field of the packet header and then resets the counter. This way, any packet travelling to an NI will automatically return any available send credits to that NI.

Send packets can be returned to the NI send buffer pool as soon as transmission has completed. We do not show send packet deallocation, because it does not trigger any significant protocol actions.

4.1.3 Receiver-Side Protocol

The receiver side of the reliability protocol is shown in Figure 4.3. This part of the protocol handles two events: packet arrival and packet release. Incoming packets are dispatched to *event_packet_received*(). This routine accepts any new send credits that the sender may have returned and then processes the packet. If the packet is a UNICAST data packet, it is delivered to the host. In this protocol description, we are not concerned with the details of NI-to-host delivery. CREDIT packets serve only to send credits to an NI; they do not require any further processing and are discarded immediately.

Routine *accept_new_credits*() adds the credits returned in packet *pkt* to the send credits for NI *src*. Next, this routine walks the blocked-sends queue to transmit to *src* as many blocked packets as possible. Since the blocked-sends queue is checked immediately when credits are returned, packets are always transmitted in FIFO order.

Although UCAST is not concerned with the exact way in which packets are delivered to the host, it needs to know when a packet buffer can be reused for new incoming packets. The send credit consumed by the packet's sender cannot be returned until the packet is available for reuse. We therefore require that a *packet release* event be generated when a receive packet is released. This event is dispatched to *event_packet_released*(), which returns a send credit to the packet's sender and then deallocates the packet. Credits are returned by *return_credit_to*(), which increments a counter (*free_credits*) in the protocol control block of the packet's sender. If this counter reaches the credit refresh threshold, *return_credit_to*() sends an explicit CREDIT packet to the sender. This happens only if data packets flow mainly in one direction. If there is sufficient return traffic *free_credits* will never reach the refresh threshold, because *send_packet*() will already have piggybacked all free credits on an outgoing UNICAST packet.

```
void event_packet_received(Packet *pkt) {
        accept_new_credits(pkt);

        switch(pkt→hdr.tag) {
        case UNICAST:
                deliver_to_host(pkt);
                nr_pkts_received++;
                break;
        case CREDIT:
                break;
        }
}

void accept_new_credits(Packet *pkt) {
        ProtocolCtrlBlock *pcb = &pcbtab[pkt→hdr.src];
        SendDesc *desc;

        /* Retrieve (piggybacked) credits and unblock blocked senders. */
        pcb→send_credits += pkt→hdr.credits;
        while (! queue_empty(&pcb→blocked_sends) && pcb→send_credits > 0) {
                desc = dequeue(&pcb→blocked_sends);
                credit_send(desc);
        }
}

void event_packet_released(Packet *pkt) {
        return_credit_to(pkt→hdr.src);
}

void return_credit_to(unsigned sender) {
        ProtocolCtrlBlock *pcb = &pcbtab[sender];

        pcb→free_credits++;
        if (pcb→free_credits ≥ CREDIT_REFRESH) {
                Packet credit_packet;
                SendDesc *desc = alloc_send_desc();

                credit_packet.hdr.tag = CREDIT;      /* send explicit credit packet */
                desc→dest = sender;
                desc→packet = &credit_packet;
                send_packet(desc);
        }
}
```

Fig. 4.3. Receiver side of the UCAST protocol.

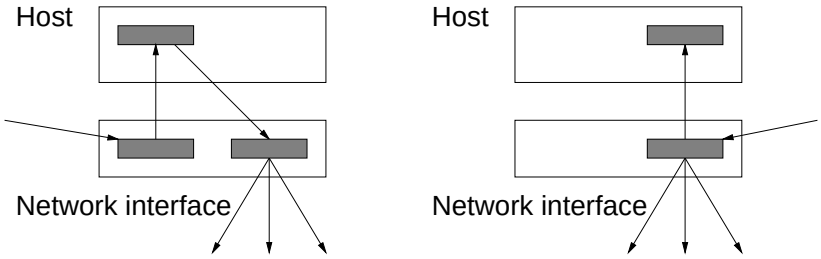


Fig. 4.4. Host-level (left) and interface-level (right) multicast forwarding.

UCAST is simple and efficient. Since the protocol preserves the reliability of the underlying hardware by avoiding receiver buffer overruns, senders need not buffer packets for retransmission, nor need they manage any timers. The main disadvantage of UCAST is that it statically partitions the available receive buffer space among all senders, so it needs more NI memory as the number of processors is increased. Chapter 8 studies this issue in more detail.

4.2 MCAST: Reliable Multicast Communication

Since we assume that multicast is not supported in hardware, we provide a software spanning-tree multicast protocol. This protocol, MCAST, implements a reliable and FIFO-ordered multicast. Before describing MCAST, we discuss different multicast forwarding strategies and the importance of multicast tree topology.

4.2.1 Multicast Forwarding

Most spanning-tree protocols use *host*-level forwarding. With host-level forwarding, the sender is the root of a multicast tree and sends a multicast packet to each of its children. Each child's NI passes the packet to its host, which reinjects the packet to forward it to the next level in the multicast tree (see Figure 4.4).

Host-level forwarding has three drawbacks. First, since each reinjected packet was already available in NI memory, host-level forwarding results in an unnecessary host-to-NI data transfer at each internal tree node of the multicast tree, which wastes bus bandwidth and processor time. Second, no forwarding takes place unless the host processes incoming packets. If one host does not poll the network in a timely manner, all its children will be affected. Instead of relying on polling, the NI can raise an interrupt, but interrupts are expensive. Third, the critical sender-receiver path includes the host-NI interactions of all the nodes between the sender and the receiver. For each multicast packet, these interactions consist of copying the packet to the host, host processing, and reinjecting the packet.

MCAST addresses these problems by means of NI-level forwarding: multicast packets are forwarded by the NI without intervention by host processors.

4.2.2 Multicast Tree Topology

Multicast tree topology matters in two ways. First, different trees have different performance characteristics. Many different tree topologies have been described in the literature [30, 77, 80]. Deep trees (i.e., trees with a low fanout), for example, give good throughput, because internal nodes need to forward packets fewer times, so they use less time per multicast. Latency increases because more forwarding actions are needed before the last destination is reached. Depending on the type of multicast traffic generated by an application, one topology gives better performance than another. Our goal is not to invent or analyze a particular topology that is optimal in some sense. No topology is optimal under all circumstances. It is important, however, that a multicast protocol does not prohibit topologies that are appropriate for the application at hand.

Second, with MCAST's packet forwarding scheme (described later), some trees can induce buffer deadlocks. A multicast packet requires buffer space at all its destinations. This buffer space can be obtained *before* the packet is sent or it can be obtained more dynamically, as the packet travels from one node in the spanning tree to another. The first approach is conceptually simple, but requires a global protocol to reserve buffers at all destination nodes. The second approach, taken by MCAST, introduces the danger of buffer deadlocks. A sender may know that its multicast packet can reach the next node in the spanning tree, but does not know if the packet can travel further once it has reached that node. Section 4.3 gives an example of such a deadlock.

MCAST avoids buffer deadlocks by restricting the topology of multicast trees. An alternative approach is to detect deadlocks and recover from them. This approach is taken by RECOV, an extension of MCAST that allows the use of arbitrary multicast trees.

4.2.3 Protocol Data Structures

In MCAST, NIs multicast inside multicast groups that are created at initialization time; MCAST does not include a group join or leave protocol. Each member of a multicast group has its own spanning tree which it uses to forward multicast packets to the other group members.

Figure 4.5 shows MCAST's data structures. MCAST builds on UCAST, so we reuse and extend UCAST's data structures and routines. MCAST introduces a new packet type, MULTICAST, and extend the packet header with a *tree* field. Each process is given a unique index for each multicast group that it is a member

```
typedef enum {UNICAST, MULTICAST, CREDIT} PacketType;

typedef struct {
        ... /* as before */
        unsigned tree;     /* identifies multicast tree */
} PacketHeader;

typedef struct {
        unsigned parent;
        unsigned nr_children;
        unsigned children[MAX_FORWARD];
        /* MAX_FORWARD depends on tree size and shape */
} Tree;

Tree treetab[MAX_TREES];
```

Fig. 4.5. Protocol data structures for MCAST.

of. This index identifies the process's multicast tree for that particular multicast group. When the process transmits a packet to the other members of that multicast group, it stores the tree identifier in the packet's *tree* field (see Figure 3.1). Each NI stores a multicast forwarding table (*treetab*) that is indexed by this *tree* field. The table entry for a given tree specifies how many children the NI has in that tree and lists those children.

4.2.4 Sender-Side Protocol

An NI initiates a multicast in response to *multicast* events which are handled by *event_multicast*(). This routine marks the packet to be multicast as a MULTICAST packet, stores the multicast tree identifier in the packet header, and tries to transmit the packet to all first-level children in the multicast tree by calling *forward_packet*().

Routine *forward*() looks up the table entry for the multicast tree that the packet was sent on and creates a send descriptor for each forwarding destination listed in the table entry. From then on, exactly the same procedure is followed as for a unicast packet. The packet will be transmitted to a forwarding destination only if credits are available for that destination; otherwise, the send descriptor is moved to the destination's blocked-sends queue.

```
void event_multicast(unsigned tree, Packet *pkt) {
        pkt→hdr.tag = MULTICAST;
        pkt→hdr.tree = tree;
        forward_packet(pkt);
}

void forward_packet(Packet *pkt) {
        Tree *tree = &treetab[pkt→hdr.tree];
        SendDesc *desc;
        unsigned i;

        for (i = 0; i < tree→nr_children; i++) {
                desc = alloc_send_desc();
                desc→dest = tree→children[i];
                desc→packet = pkt;

                credit_send(desc);
        }
}
```

Fig. 4.6. Sender-side protocol for MCAST.

4.2.5 Receiver-Side Protocol

Figure 4.7 shows the receiving side of MCAST. When a multicast packet is received, it is delivered to the host, just like a unicast packet. In addition, the packet is forwarded to all of the receiving NI's children in the multicast tree. This is done using the same forwarding routine described above (*forward_packet*()).

An NI receive buffer that holds a multicast packet is released when it has been delivered to the host *and* when it has been forwarded to all children in the multicast tree. At that point, *event_packet_released*() is called. As before, this routine returns a send credit to the (immediate) sender of the packet. In the case of a multicast packet, the immediate sender is the receiving NI's parent in the multicast tree. We do not use the packet header's source field (*src*) to determine the packet's sender, because this field is overwritten during packet forwarding (by *send_packet*()). Instead, we use the *tree* field to find this NI's parent; this field is never modified during packet forwarding.

To make this modified packet release routine work for unicast packets, we also define *unicast trees*. These trees consist of two nodes, a sender and a receiver; the sender is the parent of the receiver. One unicast tree is defined for each sender-receiver pair. UNICAST is modified so that it writes the unicast tree identifier in each outgoing unicast packet. Unicast trees are not defined only to make packet release work smoothly, but are also important in the RECOV protocol.

```
void event_packet_received(Packet *pkt) {
        accept_new_credits(pkt);

        switch(pkt→hdr.tag) {
        case UNICAST: ...; break /* as before */
        case CREDIT: ...; break /* as before */
        case MULTICAST:
                deliver_to_host(pkt);
                nr_pkts_received++;
                forward_packet(pkt);
                break;
        }
}

void event_packet_released(Packet *pkt) {
        return_credit_to(treetab[pkt→hdr.tree].parent);
}
```

Fig. 4.7. Receive procedure for the multicast protocol.

4.2.6 Deadlock

MCAST's simplicity results from building on reliable communication channels between NIs. Unfortunately, MCAST is susceptible to deadlock if an arbitrary topology is used. Figure 4.8 illustrates a deadlock scenario. The figure shows four NIs, each with its own send and receive pool. Recall that the NI receive buffer pool is partitioned. In this scenario, each processor is the root of a degenerate multicast tree, a linear chain. Initially, each NI owns one send credit for each destination NI. All processors (not shown in the figure) have injected a multicast packet and each multicast packet has been forwarded to the next NI in the sender's multicast chain. That is, each NI has spent its send credit for the next NI in the multicast chain. Now every packet needs to be forwarded again to reach the next NI in its chain, but no packet can make progress, because the target receive buffer on the next NI is occupied by another blocked packet. Each NI has free receive buffers, but UCAST's flow control scheme does not allow one sending NI to use another sending NI's receive buffers.

This deadlock is the result of the specific multicast trees used to forward packets (linear chains). By default, LFC uses binary trees to forward multicast packets. Figure 4.9 shows the binary multicast tree for processor 0 in a 16-node system. The multicast tree for processor p is obtained by renumbering each tree node n in the multicast tree of processor 0 to $(n+p) \bmod P$, where P is the number of processors. With these binary trees, it is impossible to construct a deadlock cycle (see Appendix A).

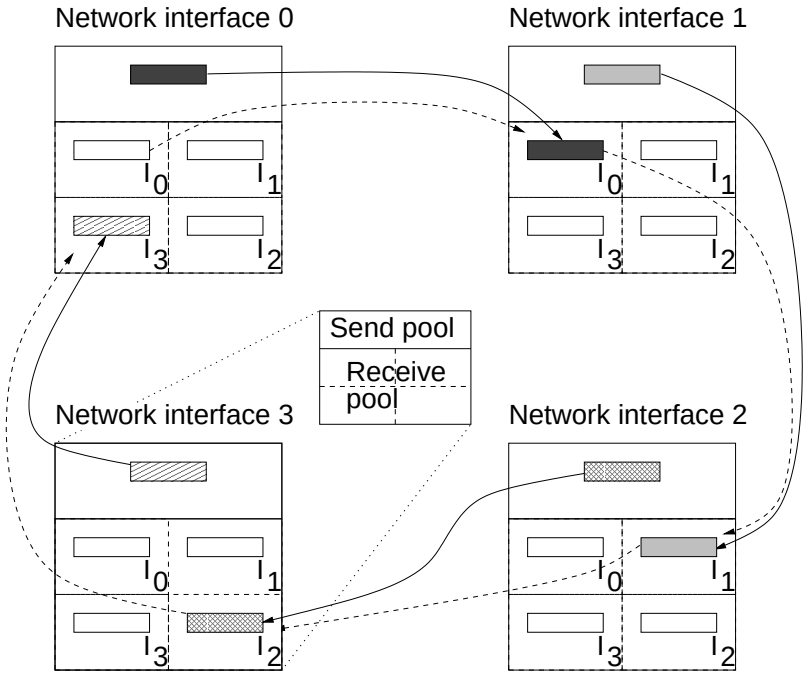


Fig. 4.8. MCAST deadlock scenario.

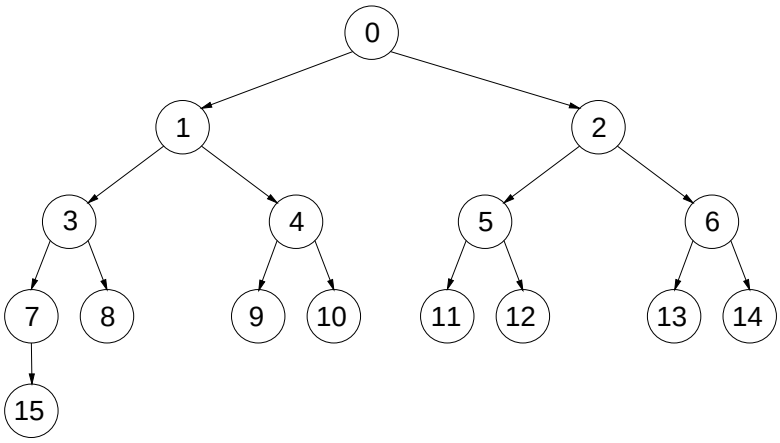


Fig. 4.9. A binary tree.

To avoid buffer deadlocks, MCAST imposes two restrictions:

1. Multicast groups may not overlap.
2. Not all multicast tree topologies are valid.

The first restriction implies that we cannot use broadcast and multicast at the same time, because the broadcast group overlaps with every multicast group. The reasons for these restrictions and a precise characterization of valid multicast trees are given in Appendix A.

4.3 RECOV: Deadlock Detection and Recovery

MCAST's restrictions on multicast tree topology are potentially cumbersome, because some topologies that are not deadlock-free are more efficient than other combinations [80]. For large messages, for example, linear chains give the best throughput, but as illustrated in the previous section, chains are not deadlock-free. Several sophisticated multicast protocols build trees that contain short chains. These trees are not deadlock-free either, yet we would like to use them if deadlock is unlikely.

To solve this problem, we have extended MCAST with a deadlock recovery mechanism. The extended protocol, RECOV, switches to a slower, but deadlock-free protocol when a deadlock is suspected. In this section, we first give a high-level overview of RECOV's deadlock detection and recovery protocol. Next, we give a detailed description of the protocol.

4.3.1 Deadlock Detection

RECOV uses the feedback from UCAST's flow control scheme to detect potential deadlocks. In the case of a buffer deadlock, each NI in the deadlock cycle has consumed all send credits for the next NI in the cycle. In RECOV, an NI therefore signals a potential deadlock when it has run out of send credits for a destination that it needs to send a packet to. That is, nonempty blocked-sends queues signal deadlock. With this trigger mechanism an NI can signal a potential deadlock unnecessarily, because an NI may run out of send credits even if there is no deadlock. The advantage of this detection mechanism, however, is that NIs need only local information (the state of their blocked-sends queues) to detect potential deadlocks. No communication is needed to detect a deadlock, which keeps the detection algorithm simple and efficient.

Since a deadlock need not involve all NIs, each NI must monitor all its blocked-sends queue; progress on some, but not all queues does not guarantee freedom of

deadlock. Checking for a nonempty blocked-sends queue is efficient, but may signal deadlock sooner than a timeout-based mechanism.

When an NI has signaled a potential deadlock, it initiates a recovery action. No new recoveries may be started by that NI while a previous recovery is still in progress. When a recovery terminates, the NI that initiated it will search for other nonempty blocked-sends queues and start a new recovery if it finds one. To avoid livelock, the blocked-sends queues are served in a round-robin fashion.

Multiple NIs can detect and recover from a deadlock simultaneously. Simultaneous recoveries do not interfere, but can be inefficient. In a deadlock cycle, only one NI needs to use the slower escape protocol to guarantee forward progress. With simultaneous recoveries, *all* NIs that detect deadlock will start using the slower escape protocol [96].

4.3.2 Deadlock Recovery

To ensure forward progress when a deadlock is suspected, RECOV requires $P-1$ extra buffers on each NI, where P is the number of NIs used by the application. Each NI N owns one buffer on every other NI. These buffers form a dedicated *recovery channel* for N. Communication on this recovery channel can result only from a recovery initiated by N.

When an NI suspects deadlock, it will use its recovery channel to make progress on one multicast or unicast tree. (Recall that we have defined both unicast and multicast tree, so that every data packet travels along some tree.) Other NIs will use this channel only when they receive a packet that was transmitted on the channel. Specifically, when an NI receives, on recovery channel C, a packet p transmitted along tree T, then that NI may forward one packet to each of its children in T. The forwarded packets must also belong to T. This guarantees that we cannot construct a cycle within a recovery channel, because all communication links used in the recovery channel form a subtree of T.

Once an NI has initiated a recovery on its recovery channel, it may not initiate another recovery until the recovery channel is completely clear. The recovery channel is cleared bottom-up. When a leaf in the recovery tree releases its escape buffer (because it has delivered the packet to its host), it sends a special acknowledgement to its parent in the recovery tree. When the parent has released *its* escape buffer and when it has received acknowledgements from all its children in the recovery tree, then it will propagate an acknowledgement to *its* parent. When the root of the recovery tree receives an acknowledgement, the recovery channel is clear.

Each NI can initiate at most one recovery at a time, but different NIs can start a recovery simultaneously, even in a single multicast tree. Such simultaneous recoveries do not interfere with each other, because they use different recovery

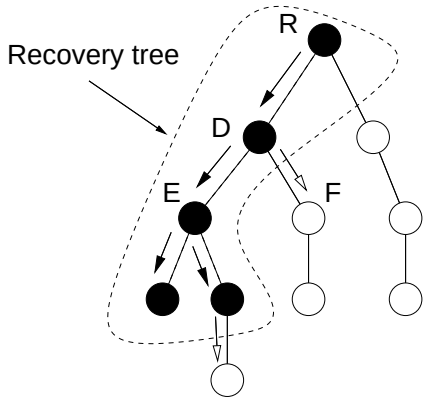


Fig. 4.10. Subtree covered by the deadlock recovery algorithm.

channels: each recovery uses the recovery channel owned by the NI that initiates the recovery.

The protocol's operation is illustrated in Figure 4.10. In this figure, vertices represent NIs and the edges show how these NIs are organized in a particular spanning tree. Dashed edges between a parent and a child indicate that the parent has no send credits left for the child.

NI R triggers a recovery because it has no send credits for its child D. R selects a blocked packet p at the head of one of its blocked-sends queues and determines the tree T that this packet is being forwarded on. R now becomes the root of a *recovery tree* and will use its recovery channel to forward p. R marks p and sends p to D. D then becomes part of the recovery tree. If D has any children for which it has no send credits, then D may further extend the recovery tree by using R's recovery channel to forward p. If p can be forwarded to an NI using the normal credit mechanism (e.g., to F), then D will do so, and the recovery tree will not include that NI. (If F cannot forward p to its child, then F will initiate a recovery on its own recovery channel.)

4.3.3 Protocol Data Structures

We now present the RECOV protocol as an extension of MCAST. Figure 4.11 shows the new and modified data structures and variables used by RECOV. A new type of packet, CHANNEL_CLEAR, is used to clear a recovery channel. CHANNEL_CLEAR packets travel from the leaves to the root of a recovery tree.

The packet header contains two new fields. (In reality LFC combines these two fields in a single *deadlock channel* field (see Section 3.4)). Field *deadlocked* is set to TRUE when a packet needs to use a recovery channel. The *channel* field identifies the recovery channel. This field contains a valid value for all UNICAST

```
typedef enum {UNICAST, MULTICAST, CREDIT, CHANNEL_CLEAR} PacketType;

typedef struct {
        ... /* as before */
        int deadlocked;     /* TRUE iff transmitted over recovery channel */
        unsigned channel;     /* identifies owner of the recovery channel */
} PacketHeader;

typedef struct {
        ... /* as before */
        int deadlocked;     /* saves deadlocked field of incoming packets */
        unsigned channel;     /* saves channel field of incoming packets */
} SendDesc;

typedef struct {
        ... /* as before */
        unsigned nr_acks;     /* need to receive this many recovery acks */
} ProtocolCtrlBlock;

typedef struct {
        ... /* as before */
        UnsignedQueue active_channels;     /* active recovery channels in this tree */
} Tree;

int recovery_active;     /* TRUE iff the local node initiated a recovery */
```

Fig. 4.11. Deadlock recovery data structures.

```
void forward_packet(Packet *pkt) {
        Tree *tree = &treetab[pkt→hdr.tree];
        int deadlocked = pkt→hdr.deadlocked;      /* save! */
        unsigned channel = pkt→hdr.channel;      /* save! */
        SendDesc *desc;
        unsigned i;

        for (i = 0; i < tree→nr_children; i++) {
                desc = alloc_send_desc();
                desc→dest = tree→children[i];
                desc→packet = pkt;
                desc→deadlocked = deadlocked;
                desc→channel = channel;

                credit_send(desc);
        }
}
```

Fig. 4.12. Multicast forwarding in the deadlock recovery protocol.

and MULTICAST packets that travel across a recovery channel (i.e., packets that have *deadlocked* set to TRUE) and for all CHANNEL_CLEAR packets. The same fields are added to send descriptors.

The protocol control block is extended with a counter (*nr_acks*) that counts how many CHANNEL_CLEAR packets remain to be received before a channel-clear acknowledgement can be propagated up the recovery tree.

The *Tree* data structure is extended with a queue of active recovery channels (*active_channels*). Each time an NI participates in a recovery on some recovery channel C for some tree T, it adds C to the queue of tree T.

4.3.4 Sender-Side Protocol

The sender side of RECOV consists of several modifications to the MCAST and UCAST routines. The modified multicast forwarding routine is shown in Figure 4.12. For each child, RECOV must know whether the packet to be forwarded arrived on a deadlock recovery channel. At the time the packet is forwarded to a specific child, we can no longer trust all information in the packet header, because this information may have been overwritten when the packet was forwarded to another child. Therefore, we have modified *forward_packet*() to save the *deadlocked* and *channel* fields of the incoming multicast packet in the send descriptors for the forwarding destinations.

The saved *deadlocked* field is used by *credit_send*() (see Figure 4.13). As before, this routine first tries to consume a send credit. If all send credits have been

consumed, however, *send_credit*() no longer immediately enqueues the packet — or rather, the descriptor that references the packet— on a blocked-sends queue. Instead, we first check if the packet arrived on a recovery channel (using the saved *deadlocked* field). If this is the case, we forward a packet that belongs to the same tree on the same recovery channel. We cannot always forward the packet referenced by *desc*, because other packets may be waiting to travel to the same destination. If we bypass these packets and one of them arrived on the same tree as the bypassing packet, then we violate FIFOness. To prevent this, *next_in_tree*() (not shown) first appends *desc* to the blocked-sends queue for the packet's destination. Next, it removes from this queue the first packet that is travelling along the same tree as the packet referenced by *desc* and it returns the send descriptor for this packet. This packet is transmitted on the recovery channel.

If the packet that *credit_send*() tries to transmit did not arrive on a deadlock recovery channel, then *credit_send*() will start a new recovery on this NI's recovery channel. However, it can do so only if it is not already working on an earlier recovery. If this NI has already initiated a recovery and this recovery is still active, then the packet is simply enqueued on its destination's blocked-sends queue. Otherwise the NI starts a new recovery on its own recovery channel.

For completeness, we also show the modified *send_packet*() routine. This routine now also copies the *deadlocked* field and the *channel* field into the packet header.

4.3.5 Receiver-Side Protocol

The receiver-side code of RECOV is shown in Figure 4.14 and Figure 4.15. Only a few modifications to *event_packet_received*() are needed.

First, special action must be taken when a packet arrives on a recovery channel. In that case, the receiving NI creates some state for the recovery that this packet is part of. In the protocol control block of the recovery channel's owner, the NI sets the channel-clear counter to 1. This counter is used when the recovery channel is cleared and indicates how many parties need to agree that the channel is clear before a channel-clear acknowledgement can be sent up the recovery tree. As long as the receiving NI does not extend the recovery tree, only it needs to say that the channel is clear, hence the counter is set to 1. If, however, the recovery tree is extended by forwarding the packet on its recovery channel, then *credit_send*() will increase the counter to indicate that an additional channel-clear acknowledgement is needed from the child that the packet was forwarded to.

Routine *event_packet_received*() also stores the identity of the recovery channel's owner (the NI that initiated the recovery) on a queue associated with the tree that the incoming packet travelled on. This information is used when packets are released.

```
void credit_send(SendDesc *desc) {
        ProtocolCtrlBlock *pcb = &pcbtab[desc→dest];

        if (pcb→send_credits > 0) {
                desc→deadlocked = FALSE;
                pcb→send_credits−−;     /* consume a credit */
                send_packet(dest);
                return;
        }

        if (desc→deadlocked) {
                /* desc→packet arrived on a deadlock channel. We forward desc→packet,
                * or a predecessor in the same tree, on the same channel.
                */
                pcbtab[desc→channel].nr_acks++;
                desc = next_in_tree(pcb, desc);     /* preserve FIFOness in tree */
                send_packet(desc);     /* no credit consumed by this send */
        } else if (recovery_active) {
                /* Cannot do more than one recovery at a time. */
                enqueue(&pcb→blocked_sends, desc);
        } else {
                /* Initiate recovery on my deadlock channel */
                recovery_active = TRUE;
                pcbtab[LOCAL_ID].nr_acks = 1;
                desc→deadlocked = TRUE;
                desc→channel = LOCAL_ID;     /* my recovery channel */
                send_packet(desc);     /* no credit consumed by this send */
        }
}

void send_packet(SendDesc *desc) {
        ProtocolCtrlBlock *pcb = &pcbtab[desc→dest];
        Packet *pkt = desc→packet;

        pkt→hdr.src = LOCAL_ID;
        pkt→hdr.credits = pcb→free_credits;     /* piggyback credits */
        pkt→hdr.deadlocked = desc→deadlocked;
        pkt→hdr.channel = desc→channel;
        pcb→free_credits = 0;     /* we returned all credits to dest */

        transmit(desc→dest, pkt);
}
```

Fig. 4.13. Packet transmission in the deadlock recovery protocol.

```
void event_packet_received(Packet *pkt) {
        accept_new_credits(pkt);

        if (pkt→hdr.deadlocked) {
                ProtocolCtrlBlock *channel_owner = &pcbtab[pkt→hdr.channel];
                Tree *tree = &treetab[pkt→hdr.tree];

                /* Register my state for this recovery in the pcb of the node that
                * initiated the recovery. Add the owner's id to a queue of ids
                * that is maintained for the tree that the packet belongs to.
                */
                channel_owner→nr_acks = 1;      /* one 'ack' for local delivery */
                enqueue(&tree→active_channels, pkt→hdr.channel);
        }

        switch(pkt→tag) {
        case UNICAST: ...; break;      /* as before */
        case MULTICAST: ...; break;      /* as before */
        case CREDIT: ...; break;      /* as before */
        case CHANNEL_CLEAR:
                release_recovery_channel(pkt→channel);
                break;
        }
}
```

Fig. 4.14. Receive procedure for the deadlock recovery protocol.

```
void release_recovery_channel(unsigned channel) {
        ProtocolCtrlBlock *channel_owner = &pcbtab[channel];

        channel_owner→nr_acks−−;
        if (channel_owner→nr_acks > 0) return;      /* channel not clear yet */

        /* Below and at this tree node, the recovery channel is clear. */
        if (channel == LOCAL_ID) {
                /* Recovery completed; the entire recovery channel is clear. */
                recovery_active = FALSE;
                try_new_recovery();      /* search for new nonempty blocked-sends queue */
        } else {
                Packet deadlock_ack;
                SendDesc *desc = alloc_send_desc();

                /* Propagate channel-clear ack to parent */
                deadlock_ack.hdr.tag = CHANNEL_CLEAR;
                desc→dest = pcb→parent;
                desc→packet = &deadlock_ack;
                desc→deadlocked = FALSE;
                desc→channel = channel;
                send_packet(desc);
        }
}

void event_packet_released(Packet *pkt) {
        Tree *tree = &treetab[pkt→hdr.tree];

        if (queue_empty(&tree→active_channels)) {
                return_credit_to(tree→parent);
        } else {
                /* Do not return a credit, but try to clear a deadlock channel */
                unsigned channel = dequeue(&tree→active_channels);
                release_recovery_channel(channel);
        }
}
```

Fig. 4.15. Packet release procedure for the deadlock recovery protocol.

Second, we must process incoming channel-clear acknowledgements. This is done by *release_recovery_channel*() (see Figure 4.15). This routine propagates channel-clear acknowledgements up the recovery tree and detects recovery completion. If the channel-clear counter does not drop to zero, then this routine does nothing. If the counter drops to zero, there are two possibilities. If the channel-clear acknowledgement has reached the root of the recovery tree, then the recovery is complete. The owner of the recovery channel knows that the entire recovery channel is free and will try to initiate a new recovery on its recovery channel. This is done by *try_new_recovery*() (not shown), which scans all blocked-sends queues for a blocked packet. If such a queue is found, a new recovery is started. If the channel-clear acknowledgements have not yet reached the recovery channel's owner (the root of the recovery tree), then we create a new channel-clear acknowledgement and send it up the recovery tree.

We have now described how a recovery is initiated, how a recovery tree is built, and how channel-clear acknowledgements travel back up the recovery tree, but we have not explained how the clearing of a recovery channel is initiated. This is done by the leaves of the recovery tree when they release a packet. The modified *event_packet_released*() first checks if this NI is participating in any recovery in the tree that the packet arrived on. If it is, then it will have stored the identities of the recovery channel owners in the *active_channels* queue associated with the tree. If this queue is not empty, *event_packet_released*() dequeues one channel owner and returns the packet to that owner's recovery channel. If, on the other hand, this NI is not participating in a recovery on the tree that the packet arrived on, then it will simply return a send credit to the packet's last sender (as before).

4.4 INTR: Control Transfer

Since interrupts are expensive, LFC tries to avoid generating interrupts unnecessarily by means of a *polling watchdog* [101]. This is a mechanism that delays interrupts in the hope that the target process will soon poll the network. The idea is to start a watchdog timer when a packet arrives at the NI. The NI generates an interrupt only if the host does not poll before the timer expires,

Where the original polling watchdog proposal by Maquelin et al. is a hardware design, LFC uses the programmable NI processor to implement a software polling watchdog. In addition, LFC refines the original design in the following way. When the watchdog timer expires, LFC does not immediately generate an interrupt if the host has not yet processed all packets delivered to it. Instead, the NI performs two additional checks to determine whether it should generate an interrupt: an interrupt status check and a client progress check.

The purpose of the interrupt status check is to avoid generating interrupts when

the receiving process runs with network interrupts disabled. Recall that clients may dynamically disable and enable network interrupts. For efficiency, the interrupt status manipulation functions are implemented by toggling an interrupt status flag in host memory, without any communication or synchronization with the control program (see Section 3.7). Before sending an interrupt, the control program fetches (using a DMA transfer) the interrupt status flag from host memory. No interrupt is generated when the flag indicates that the client disabled interrupts.

The goal of the client progress check is to avoid generating interrupts when the receiving process recently processed some (but not necessarily all) of the packets delivered to it. The host maintains a count of the number of packets that it has processed. When the watchdog timer expires, the control program fetches this counter. No interrupt is generated if the host processed at least one packet since the previous watchdog timeout.

INTR starts the watchdog timer each time a packet arrives and the timer is not already running on behalf of an earlier packet. Figure 4.16 shows how INTR handles polling watchdog timeouts. The NI fetches the host's interrupt flag and packet counter using a single DMA transfer; both are stored into *host*. The NI then performs the first part of the client progress check: if the client has processed all (*nr_pkts_received*) packets delivered by the NI, then the NI cancels the polling watchdog timer. In this case, the host processed all packets that were delivered to it and there is no need to keep the timer running. If this check fails, the NI performs both the interrupt status check and the second part of the client progress check. If the host recently disabled interrupts or if the host consumed some of the packets delivered to it, then the NI does not generate an interrupt, but restarts the polling watchdog timer. If all checks fail, the NI generates an interrupt and restarts the timer. Finally, the NI saves the host's packet counter in *last_processed*, so that on the next timeout it can check if the client made progress.

4.5 Related Work

LFC's NI-level protocols implement reliability, interrupt management, and multicast forwarding. We discuss related work in all three areas.

4.5.1 Reliability

LFC assumes that the network hardware is reliable and preserves this reliability by means of its link-level flow control protocol (UCAST). Several other systems (BIP [118], Hamlyn [32], Illinois Fast Messages [112, 113], FM/MC [146], VMMC [24], and PM [141]) make the same hardware reliability assumption (see Figure 2.3). These systems, however, implement flow control in a different way.

```
typedef struct {
        unsigned nr_pkts_processed;
        int intr_disabled;
} HostStatus;

unsigned last_processed;

void event_watchdog_timeout(void) {
        HostStatus host;

        fetch_from_host(&host);
        if (host.nr_pkts_processed == nr_pkts_received) {
                timer_cancel();     /* client processed all packets */
        } else if (host.intr_disabled || host.nr_pkts_processed > last_processed) {
                timer_restart();    /* client disabled interrupts or polled recently */
        } else {
                send_interrupt_to_host();
                timer_restart();
        }
        last_processed = host.nr_pkts_processed;
}
```

Fig. 4.16. Timeout handling in INTR.

With the exception of PM (see Section 2.6) none of these systems implements link-level flow control. Instead, they assume that the NI can copy data to the host sufficiently fast to prevent buffer overflows (see Section 1.6).

With the exception of FM/MC, none of the above systems supports a complete NI-level multicast. With NI-level forwarding, multicast packets occupy NI receive buffers for a longer time. As a result, the pressure on these buffers increases, and special measures are needed to prevent blocked-packet resets. FM/MC treats high NI receive buffer pressure as a rare event and swaps buffers to host memory when the problem occurs. For applications that multicast very heavily, however, swapping occurs frequently and degrades performance [15]. LFC is more conservative and uses NI-level flow control to solve the problem.

4.5.2 Interrupt Management

LFC supports both polling and interrupts as control-transfer mechanisms. Some communication systems provide only a polling primitive. The main problem with this approach is that many parallel-programming systems, notably DSM systems, cannot rely on polling by the application. Interrupts provide asynchronous notification, but are expensive on commercial architectures and operating systems: the

time to field an interrupt often exceeds the latency of a small message. Research systems such as the Alewife [2] and the J-machine [132] support fast interrupts, but the mechanisms used in these machines have not found their way into commercial processor architectures. Even without hardware support, operating systems can, in principle, dispatch interrupts efficiently [94, 143]. The key idea is to save minimal processor state, and leave the rest (e.g., floating-point state) up to the application program. In practice, however, interrupt processing on commodity operating systems has remained expensive.

LFC does not attack the overhead of individual interrupts, but aims to reduce the interrupt frequency by optimistically delaying interrupts. Maquelin et al. proposed the polling watchdog, a hardware implementation of this idea [101]. We augmented the polling watchdog with the interrupt-status and client-progress checks described in Section 4.4. These two checks further reduce the number of spurious interrupts.

4.5.3 Multicast

Several researchers have proposed to use the NI instead of the host processor to forward multicast packets [56, 68, 80, 146]. Our NI-level protocol is original, however, in that it integrates unicast and multicast flow control and uses a deadlock-recovery scheme to avoid routing restrictions.

Verstoep et al. describe a system, FM/MC, that implements an NI-level multicast for Myrinet [146]. FM/MC runs on the same hardware as LFC, but implements buffer management and reliability in a very different way. Section 4.5.4 compares LFC and FM/MC in more detail.

Huang and McKinley propose to exploit ATM NIs to implement collective communication operations, including multicast [68]. In their symmetric broadcast protocol, NIs use an ATM multicast channel to forward messages to their children; multicast acknowledgements are also collected via the NIs. The sending host maintains a sliding window; it appears that a single window is used per broadcast group. LFC, in contrast, uses sliding windows between each pair of NIs. This allows LFC to integrate unicast and multicast flow control; Huang and McKinley do not discuss unicast flow control and present simulation results only.

Gerla et al. [56] also discuss using NIs for multicast packet forwarding. They propose a deadlock avoidance scheme that divides receive buffers in two classes. The first class is used when messages travel to higher-numbered NIs, the second when going to lower-numbered NIs. This scheme requires buffer resources per multicast tree, which is problematic in a system with many small multicast groups.

Kesavan and Panda studied optimal multicast tree shapes for systems with programmable NIs [80]. They describe two packet-forwarding schemes, first-child-first-served and first-packet-first-served (FPFS), and show that the latter performs

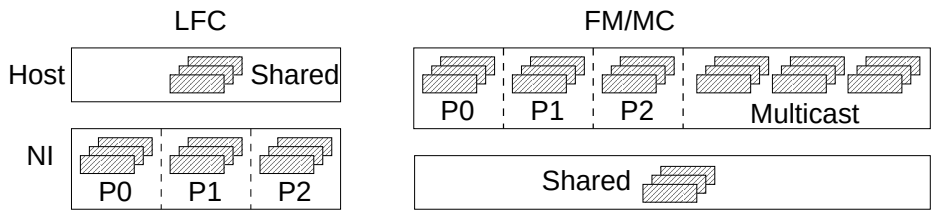


Fig. 4.17. LFC's and FM/MC's buffering strategies.

best. The paper does not discuss flow control issues. MCAST integrates FPFS with UCAST's flow control scheme. MCAST usually forwards packets to all children as these packets arrive, just like in FPFS. When an NI has to forward a packet to a destination for which no send credits are available, however, the NI will queue that packet and start working on another packet.

All spanning-tree multicast protocols must deal with the possibility of deadlock. Deadlock is often avoided by means of a deadlock-free routing algorithm; the literature on such algorithms is abundant. Most research in this area, however, applies to routing at the hardware level. Also, as pointed out by Anjan and Pinkston [4], most protocols *avoid* deadlock by imposing routing restrictions [43]. This is the approach taken by MCAST. RECOV was inspired by the DISHA protocol for wormhole-routed networks [4]. DISHA, however, is a deadlock recovery scheme for unicast worms, whereas RECOV deals with buffer deadlocks in a store-and-forward multicast protocol.

4.5.4 FM/MC

FM/MC implements a reliable NI-level multicast for Myrinet, but does so in a very different way than LFC. The most important differences are related to buffer management and the reliability protocol.

Figure 4.17 shows how LFC and FM/MC allocate receive buffers. FM/MC uses a single queue of NI buffers for all inbound network packets. LFC, on the other hand, statically partitions its NI buffers among all senders: one sender cannot use another sender's receive buffers. At the host level, the situation has almost been reversed. FM/MC allocates a fixed number of unicast buffers per sender. These unicast buffers are managed by a standard sliding-window protocol that runs on the host. In addition, FM/MC has another, separate class of multicast buffers which are not partitioned statically among senders. LFC does not distinguish between unicast and multicast buffers and uses a single pool of host receive buffers.

FM/MC's multicast buffers are allocated to senders by a central credit manager that runs on one of the NIs. A multicast credit represents a buffer on *every*

receiving host. Before a process can multicast a message, it must obtain credits from the credit manager. To avoid the overhead of a credit request-reply pair for every multicast packet, hosts can request multiple credits and cache them. Credits that have been consumed are returned to the credit manager by means of a token that rotates among all NIs.

At the NI level, no software flow control is present. Each NI contains a single receive queue for all inbound packets. Under conditions of high load, this queue fills up. FM/MC then moves part of the receive queue to host memory to make space for inbound packets. Eventually, senders will run out of credits and the memory pressure on receiving NIs will drop; packets can then be copied back to NI memory and processed further. The idea is that this type of swapping to host memory will occur only under exceptional conditions, and it is assumed that swapping will not take too much time. Eventually, however, FM/MC relies on hardware back-pressure to stall sending NIs.

By default, FM/MC uses the same binary trees as LFC to forward multicast packets. The protocol, however, allows the use of arbitrary multicast trees, because the credit and swapping mechanism avoid buffer deadlocks at the NI level. When NI buffers fill up, they are copied to host memory. The centralized credit mechanism guarantees that buffer space is available on the host. Since packets are not forwarded from host memory, there is no risk of buffer deadlock at the host level.

Both LFC and FM/MC have their strengths and weaknesses. An important advantage of LFC over FM/MC is its simplicity. A single flow control scheme is used for unicast and multicast traffic. There is no need to communicate with a separate credit manager. No request messages are needed to obtain credits: receivers know when senders are low on credits and send credit update messages without receiving any requests. Finally, since LFC implements NI-level flow control, it does not need to swap buffers back and forth between host and NI memory. The price we pay for this simplicity is the restriction on the tree topologies that can be used by MCAST. FM/MC can use arbitrary multicast trees.

LFC and FM/MC also differ in the way credits are obtained for multicast packets. LFC is optimistic in that a sender waits only for credits for its children in the multicast tree. In FM/MC, a sender waits until every receiver has space before sending the next multicast packet(s). On the other hand, host buffers are not as scarce a resource as NI buffers, so waiting in FM/MC may occur less frequently.

Both protocols have potential scalability problems. LFC partitions NI receive buffers among all senders. FM/MC, in contrast, allows NI buffers to be shared, which is attractive when the amount of NI memory is small and the number of nodes in the system large. Given a reasonable amount of memory on the NI and a modest number of nodes, however, LFC's partitioning of NI buffers poses no problems, and obviates the need for FM/MC's buffer swapping mechanism.

FM/MC has other scalability problems. First, all credit requests are processed by a single NI, which introduces a potential bottleneck. Second, to recover multicast credits, FM/MC uses a rotating token. The credit manager may have to wait for a complete token rotation until it can satisfy a request for credits.

4.6 Summary

This chapter has given a detailed description of LFC's NI-level protocols for reliable point-to-point communication, reliable multicast communication, and for interrupt management. The UCAST protocol provides reliable point-to-point communication between NIs. MCAST extends UCAST with multicast support, but is not deadlock-free for all multicast trees. RECOV extends MCAST with deadlock recovery and the resulting protocol is deadlock-free for all multicast trees. Finally, INTR is a refined software polling watchdog protocol. INTR delays the generation of network interrupts as much as possible by monitoring the behavior of the host.

Chapter 5

The Performance of LFC

This chapter evaluates the performance of LFC's unicast, fetch-and-add, and broadcast primitives using microbenchmarks. The performance of client systems and applications, which is what matters in the end, will be studied in subsequent chapters.

5.1 Unicast Performance

First we discuss the latency and throughput of LFC's point-to-point message-passing primitive, *lfc_ucast_launch*(). Next, we describe LFC's point-to-point performance in terms of the LogGP [3] parameters.

5.1.1 Latency and Throughput

Figure 5.1 shows LFC's one-way latency for different receive methods. All messages fit in a single packet and are received through polling. We used the following receive methods:

1. No-touch. The receiving process is notified of a packet's arrival (in host memory), but never reads the packet's contents. This type of receive behavior is frequently used in latency benchmarks and gives a good indication of the overheads imposed by LFC.

2. Read-only. The receiving process uses a for loop to read the packet's contents into registers (one 32-bit word per loop iteration), but does not write the packet's contents to memory. This type of behavior can be expected for small control messages, which can immediately be interpreted by the receiving processor.

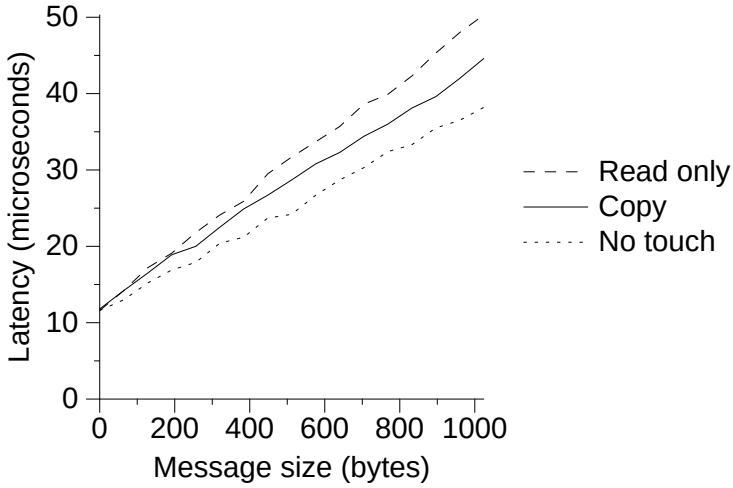


Fig. 5.1. LFC's unicast latency.

3. Copy. The receiving process copies the packet's contents to memory. (This is done using an efficient string-move instruction.) This is the typical behavior for larger messages that need to be copied to a data structure in the receiving process's address space.

The one-way latency of an empty packet is 11.6 μs (for all receive strategies). This does not include the cost of packet allocation at the sending side and packet deallocation at the receiving side, because both can be performed off the critical path. Send packet allocation costs 0.4 μs, receive packet deallocation costs 0.7 μs.

As expected, the read-only and copy latencies are higher than the no-touch latencies, because the receiving process incurs cache misses when it reads the packet's contents. Surprisingly, however, the copy variant is faster than the read-only variant. This has two causes. First, the copy variant always copies incoming packets to the same, write-back cached destination buffer, so the writes that result from the copy do not generate any memory traffic (as long as the buffer fits in the cache). Second, the read-only variant accesses the data by means of a for loop, while the copy variant uses a fast string-copy instruction.

Figure 5.2 shows the one-way unicast throughput using the same three receive methods. (Note that the message size axis has a logarithmic scale.) The sawtooth shape of the curves is caused by fragmentation. Messages larger than 1 Kbyte are split into multiple packets. If the last of these packets is not full, the overall efficiency of the message transfer decreases. This effect is strongest when the last fragment is nearly empty.

With the no-touch receive strategy, the maximum throughput is 72.0 Mbyte/s. Reading the data reduces the throughput significantly. For message sizes up to 128 Kbyte, copying the data is faster than just reading it, for the reasons described

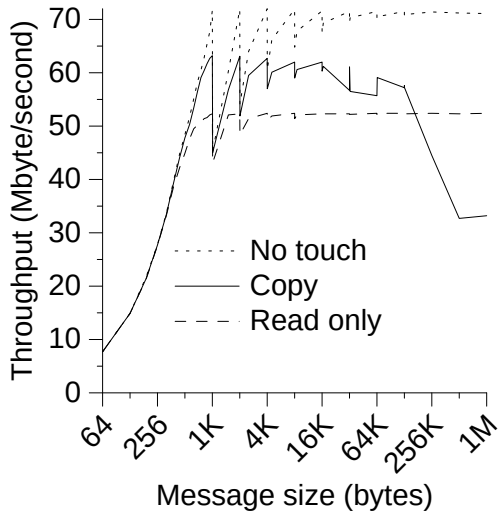


Fig. 5.2. LFC's unicast throughput.

above (write-back caching and a fast copy instruction). For large messages, however, the throughput of the copy variant drops noticeably (e.g., from 63.2 Mbyte/s for 1 Kbyte messages to 33.2 Mbyte/s for 1 Mbyte messages). For messages up to 256 Kbyte, the receive buffer used by the copy variant ought to fit in the processor's second-level data cache. Since the L2 cache is physically indexed, however, some of the buffer's memory pages may map to the same cache lines as other pages. The exact page mappings vary from run to run, but for larger buffers, conflicts are more likely to occur. Conflicting page mappings result in conflict misses during the copy operation and these misses generate memory traffic. This traffic competes for bus and memory bandwidth with incoming network packets and reduces the overall throughput. Messages larger than 256 Kbyte no longer fit in the L2 cache. Cache misses are guaranteed to occur when such a message is copied.

Although LFC achieves good throughput, it is not able to saturate the hardware bottleneck, the I/O bus, which has a bandwidth of 127 Mbyte/s. The main reason is that LFC uses programmed I/O (with write combining) at the sending side. As shown in Figure 2.2, this data transfer mechanism cannot saturate the I/O bus. This problem can be alleviated by switching to DMA, but DMA introduces its own problems (see Section 2.2).

5.1.2 LogGP Parameters

While the throughput and latency graphs are useful, additional insight can be gained by measuring LFC's LogGP parameters. LogGP [3, 102] is an extension of the LogP model [40, 41], which characterizes a communication system in terms of

four parameters: latency (L), overhead (o), gap (g), and the number of processors (P).

Latency is the time needed by a small message to propagate from one computer to another, measured from the moment the sending processor is no longer involved in the transmission to the moment the receiving processor can start processing the message. Latency includes the time consumed by the sending NI to transmit the message and the time consumed by the receiving NI to deliver the message to the receiving host. This definition of latency is different from the sender-to-receiver latency discussed above. The sender-to-receiver latency includes send and receive overhead (described below), which are not included in L.

Overhead is the time a processor spends on transmitting or receiving a message. We will make the usual distinction between send overhead (o_s) and receive overhead (o_r). Overhead does not include the time spent on the NI and on the wire. Unlike latency, overhead cannot be hidden.

The *gap* is the reciprocal of the system's small-message bandwidth. Successive packets sent between a given sender-receiver pair are always at least g microseconds apart from each other.

The LogP model assumes that messages carry at most a few words. Large messages, however, are common in many parallel programs and many communication systems can transfer a single large message more efficiently than many small messages. LogGP therefore extends LogP with a parameter, G, that captures the bandwidth for large messages. For LFC, we define G as the peak throughput under the no-touch receive strategy.

To measure the values of the LogGP parameters for LFC, we used benchmarks similar to those described by Iannello et al [69]. The resulting values are given in Table 5.1. All values, except the value of G, were measured using messages with a payload of 16 bytes.

The sender-to-receiver latencies reported in Figure 5.1 are related to the LogGP parameters as follows:

$$\text{sender-to-receiver latency} = o_s + L + o_r.$$

The sender-to-receiver latency of a 16-byte message is thus $1.6 + 8.2 + 2.2 = 12.0$ μs, slightly more than the latency of a zero-byte message (11.6 μs). The largest part (L) of the sender-to-receiver latency is spent on the sending and receiving NIs; this part can in principle be overlapped with useful work on the host processor.

5.1.3 Interrupt Overhead

All previous measurements were performed with network interrupts disabled. We now briefly consider interrupt-driven message delivery. As explained in Sec-

LogGP parameter	Value
Send overhead (o_s)	1.6 μs
Receive overhead (o_r)	2.2 μs
Latency (L)	8.2 μs
Gap (g)	5.6 μs
Big gap (G)	72.0 Mbyte/s

Table 5.1. Values of the LogGP parameters for LFC.

tion 3.7, LFC delays interrupts in the hope that the destination process will poll before the interrupt needs to be raised. The default delay is 70 μs. This value, approximately twice the interrupt overhead (see below), was determined after experimenting with some of the applications described in Chapter 8.

To measure the overhead of interrupt-driven message delivery, we set the delay to 0 μs and measure the null unicast latency in two different ways: first, using a program that does not poll for incoming messages and that uses interrupts to receive messages; second, using a program that does poll and that does not use interrupts. The difference in latency measured by these two programs stems from interrupt overhead. Using this method, we measure an interrupt overhead of 31 μs, which is more than twice the unicast null latency. This time includes context-switching from user mode to kernel mode, interrupt processing inside the kernel and the Myrinet device driver, dispatching the user-level signal handler, and returning from the signal handler (using the *sigreturn*() system call). Notice that the last overhead component, the signal handler return, need not be on the critical communication path: a message can be processed and a reply can be sent before returning from the signal handler.

This large interrupt overhead is not uncommon for commercial operating systems. In user-level communication systems with low-latency communication, however, the high cost of using interrupts should clearly be avoided whenever possible. It is exactly this high cost that motivated the addition of a polling watchdog mechanism to LFC.

5.2 Fetch-and-Add Performance

A contention-free fetch-and-add operation on a variable in local NI memory costs 20.7 μs. Since we have not optimized this local case, an FA_REQUEST and an FA_REPLY message will be sent across the network. Each message travels to the switch to which the sending NI is attached; this switch then sends the message back to the same NI.

A contention-free F&A operation on a remote F&A variable costs 19.8 μs,

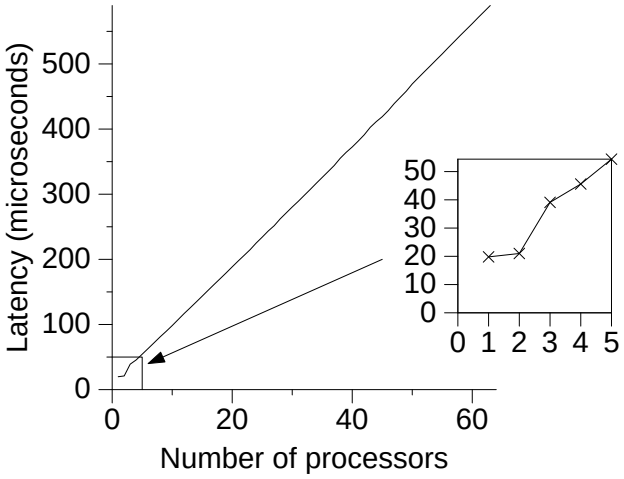


Fig. 5.3. Fetch-and-add latencies under contention.

which is slightly less expensive than the local case. To detect the F&A reply, the host that issued the operation polls a location in its NI's memory. In the local case, this polling steals memory cycles from the local NI processor which needs to process the F&A request and send the F&A reply (to itself). In the remote case, receiving the request, processing it, and sending the reply are all performed by a remote NI, without interference from a polling host.

Figure 5.3 shows how F&A latencies increase under contention. In this benchmark, each processor, except the processor where the F&A variable is stored, executes a tight loop in which an F&A operation is issued. With as few as three processors (see inset), the NI that services the F&A variables becomes a bottleneck and the latencies increase rapidly.

5.3 Multicast Performance

The multicast performance evaluation is organized as follows. First, we present latency, throughput, and forwarding overhead results for LFC's basic protocol. Next, we evaluate the deadlock recovery protocol under various conditions. In Chapter 8 we will also compare the performance of LFC's multicast protocol with protocols that use host-level forwarding.

5.3.1 Performance of the Basic Multicast Protocol

We first examine the latency, throughput, and forwarding overhead of LFC's basic multicast protocol. The basic protocol uses binary trees and therefore does not need the deadlock recovery mechanism. In the following measurements, the

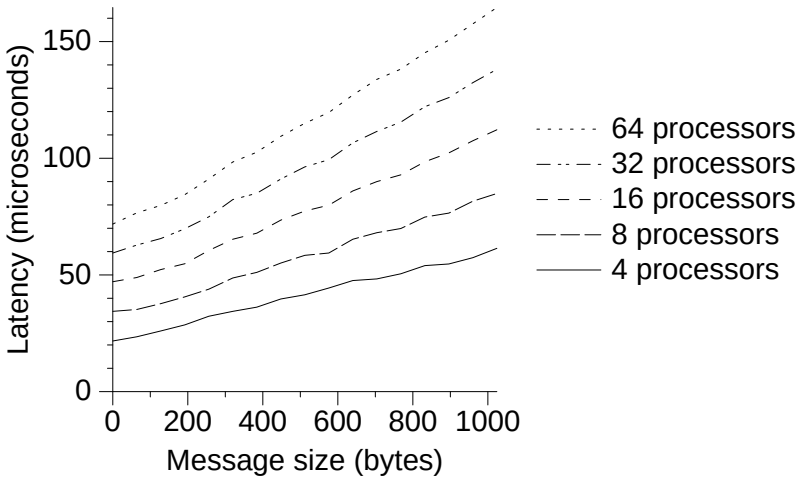


Fig. 5.4. LFC's broadcast latency.

maximum packet size is 1 Kbyte and each sender is allocated $\lfloor \frac{512}{P} \rfloor$ send credits, where P is the number of processors. All packets are received through polling. We use the copy receive strategy described in Section 5.1.

Figure 5.4 shows the broadcast latencies of the basic protocol. We define broadcast latency as the latency between the sender and the *last* receiver of the broadcast packet. As expected, the latency increases linearly with increasing message size and logarithmically with increasing numbers of processors. With a 4-node configuration, we obtain a null latency of 22 μs; with 64 nodes, the null latency is 72 μs.

Figure 5.5 shows the throughput of the basic protocol. Throughput is measured using a straightforward blast test in which the sender transmits many messages of a given size and then waits for an (empty) acknowledgement message from all receivers. We report the throughput observed by the sender.

All curves decline for large messages. This is due to the same cache effect as described in Section 5.1. For messages that fit in the cache, the maximum multicast throughput (41 Mbyte/s for 4 processors) is less than the maximum unicast throughput (63 Mbyte/s) and decreases as the number of processors increases. These effects are due to the following throughput-limiting factors:

1. NI memory supports at most two memory accesses per NI clock cycle (33.33 MHz). Memory references are issued by the NI's three DMA engines and by the NI processor. None of these sources can issue more than one memory reference per clock cycle. As a result, the maximum packet receive and send rate is 127 Mbyte/s (33.33 MHz $\times$ 32 bits). This maximum can be obtained only with large packets; with LFC's 1 Kbyte packets, the maximum DMA speed is approximately 120 Mbyte/s.

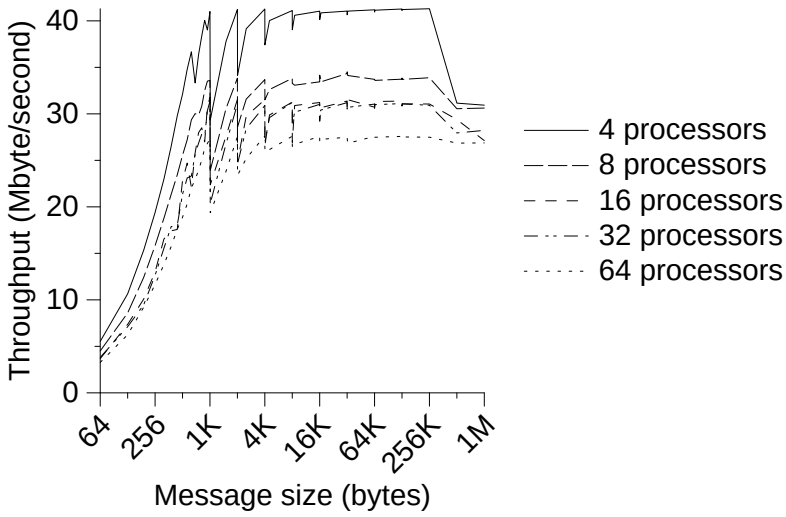


Fig. 5.5. LFC's broadcast throughput.

2. The fanout of the multicast tree determines the number of forwarding transfers that an NI's send DMA engine must make. Since these forwarding transfers must be performed serially, the attainable throughput is proportional to the reciprocal of the fanout. Binary trees have a maximum fanout of two. The maximum throughput that can be obtained is therefore $120/2 = 60$ Mbyte/s.

3. High throughput can only be obtained if the NI processor manages to keep its DMA engines busy.

4. Multicast packets travelling across different logical links in the multicast tree may contend for the same physical links. The amount of contention depends on the mapping of processes to processors. Since different mappings can give very different performance results, we used a simulated-annealing algorithm to obtain reasonable mappings for all single-sender throughput benchmarks. The resulting mappings give much better performance than random mappings, but are not necessarily optimal, because simulated annealing is not guaranteed to find a true optimum and because we did not take acknowledgement traffic into account.

The first three factors explain why we do not attain the unicast throughput, but not why throughput decreases when the number of processors is increased. Since the amount of forwarding work performed by the bottleneck nodes of the multicast tree —i.e., internal nodes with two children— is independent of the size of the multicast tree, we would expect that throughput is also independent of the size

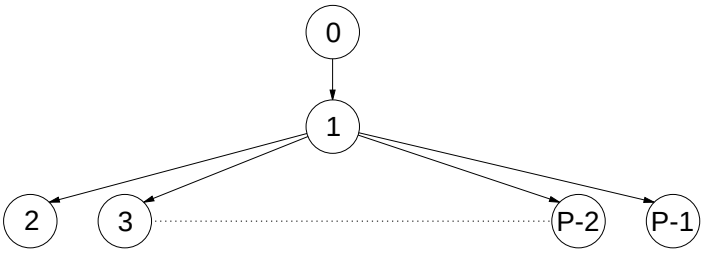


Fig. 5.6. Tree used to determine multicast gap.

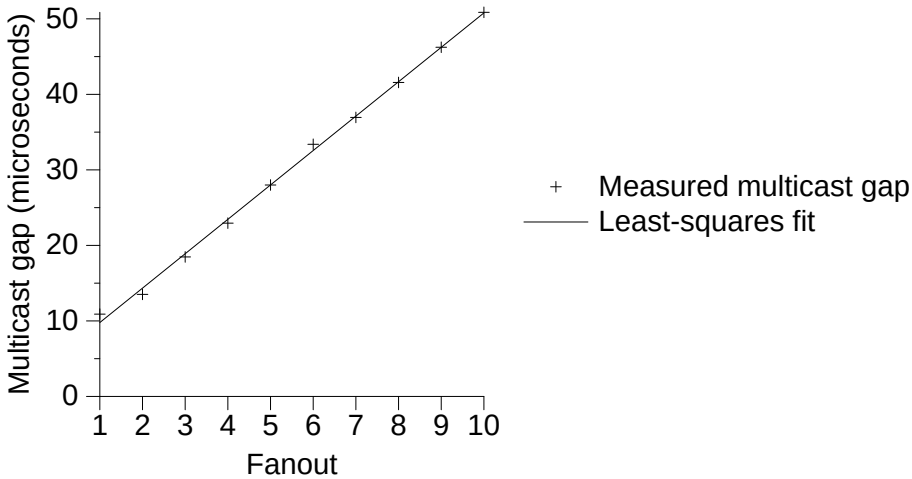


Fig. 5.7. LFC's broadcast gap.

of the tree. In reality, adding more processors increases the demand for physical network links. The resulting contention leads to decreased throughput.

To estimate the amount of work performed by the NI to forward a multicast packet, we can measure the *multicast gap* $g(P)$. This is the reciprocal of the multicast throughput on P processors for small (16-byte) messages. We assume that the multicast gap is proportional to the fanout $F(P)$ of the bottleneck node(s) in the multicast tree. That is, we assume $g(P) = \alpha \cdot F(P) + \beta$, for some α and β. By varying $F(P)$ and measuring the resulting multicast gaps $g(P)$, we can find α and β. LFC's binary trees, however, have a fixed fanout of two. To vary $F(P)$, we therefore use multicast trees that have the shape shown in Figure 5.6. In such trees, there is exactly one forwarding node and the fanout of the tree is $F(P) = P - 2$.

We measured the multicast gaps for various numbers of processors and used a least-squares fit to determine α and β from these measurements. The measured gaps and the resulting fit are shown in Figure 5.7; we found $\alpha = 4.6\,\mu$s per forward and $\beta = 5.2\,\mu$s. The forwarding time per destination (α) is fairly large; this time

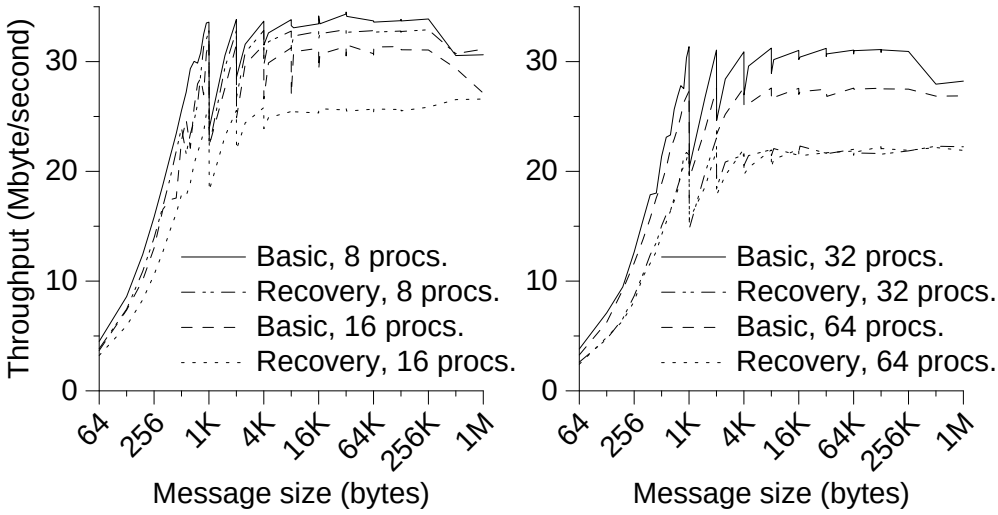


Fig. 5.8. Impact of deadlock recovery on broadcast throughput.

is used to look up the forwarding destination, to allocate a send descriptor, to enqueue this descriptor, and to initiate the send DMA.

5.3.2 Impact of Deadlock Recovery and Multicast Tree Shape

We assume that most programs will rarely multicast in such a way that deadlock recovery is necessary. It is therefore important to know the overhead that the protocol imposes without recovery.

Enabling deadlock recovery does not increase the broadcast latency. In the latency benchmark, the condition that triggers deadlock recovery (an NI has run out of send credits) is never satisfied, so the recovery code is never executed. Also, the test for the condition does not add any overhead, because it takes place both in the basic and in the enhanced protocol.

Throughput *is* affected by enabling deadlock recovery. Figure 5.8 shows single-sender throughput results for 8, 16, 32, and 64 processors, with and without deadlock recovery. For the recovery protocol measurements, we used almost the same configuration as for the basic protocol measurements shown in Figure 5.5. The only difference is that we enabled deadlock recovery and added $P-1$ NI receive buffers on each NI (see Section 4.3). We use the same binary trees as before, so no true deadlocks can occur. Since our deadlock recovery scheme makes a local, conservative decision, however, it still signals deadlock.

Our measurements indicate that, with a few exceptions, all recoveries are initiated by the root of the multicast tree. The deadlocked subtrees that result from these recoveries are small. With 64 processors and 64 Kbyte messages, for ex-

ample, each deadlock recovery results (on average) in 1.1 packets being sent via a deadlock recovery channel. In almost all cases, the deadlocked subtree consists of the root and one of its children. The root performs less work than its children, because it does not have to receive and deliver incoming multicast packets. Consequently, the root can send out multicast packets faster than its children can process them and therefore runs out of credits, which triggers a deadlock recovery.

For 8 processors, the performance impact of these recoveries is small, but for larger numbers of processors, the number of recoveries increases and the throughput decreases significantly. With 64 Kbyte messages, for example, the number of recoveries per multicast increases from 0.02 for 8 processors to 0.37 for 64 processors (on average). This increase in the number of recoveries and the decrease in throughput appear to be caused by increased contention on the physical network links (due to the larger number of processors). This contention delays acknowledgements, which causes senders to run out of credits more frequently. The problem is aggravated slightly by a fast path in the multicast code that tries to forward incoming multicast packets immediately. If this succeeds, the send channel is blocked by an outgoing data packet and cannot be used by an acknowledgement. Removing this optimization improves performance when deadlock recovery is enabled, but reduces performance when deadlock recovery is disabled.

Figure 5.9 shows the impact of tree shape on broadcast throughput. We compare binary trees and linear chains on 64 processors. We use a large number of processors, because that is when we expect the performance characteristics of different tree shapes to be most visible. In both cases, deadlock recovery was enabled, because with chains, deadlocks can occur when multiple senders multicast concurrently. With a single sender, however, deadlocks cannot occur, and we expect low-fanout trees to perform best. Figure 5.9 confirms this expectation. The linear chain (fanout 1) performs better than the binary tree (fanout 2).

To test the behavior of the deadlock recovery scheme in the case that deadlocks *can* occur, we performed an all-to-all benchmark in which all senders broadcast simultaneously. In this case, true deadlocks may occur for the chain. Figure 5.10 shows the per-sender throughput on 64 processors for the chain and the binary tree. Due to contention for network resources (links, buffers, and NI processor cycles), the per-sender throughput is much lower than in the single-sender case.

As expected, the chain performs worse than the binary tree. Table 5.2 reports several statistics that illustrate the behavior of the recovery protocol for both tree types. These statistics are for an all-to-all exchange of 64 Kbyte messages on 64 processors. In this table, L_{used} is the number of logical links between pairs of NIs that are used in the all-to-all test; L_{av} is the total number of logical links available ($64 \cdot 63$). $L_{\text{used}}/L_{\text{av}}$ indicates how well a particular tree spreads its packets over different links. RC is the number of deadlock recoveries; M is the number of multicasts. RC/M indicates how often a recovery occurs. D is the number of pack-

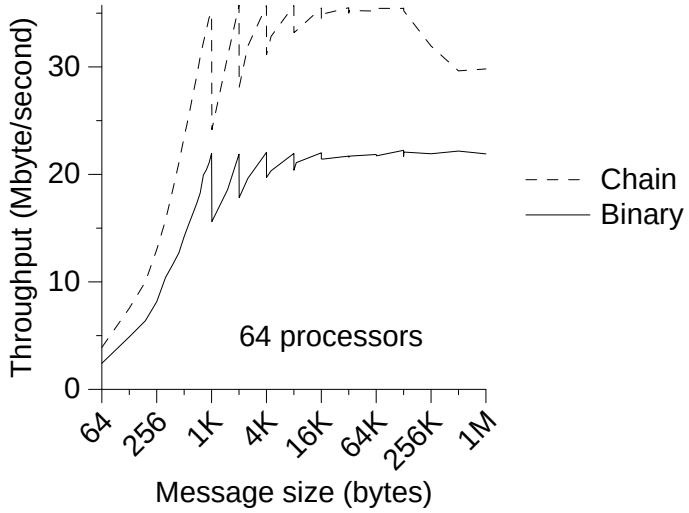


Fig. 5.9. Impact of tree topology on broadcast throughput.

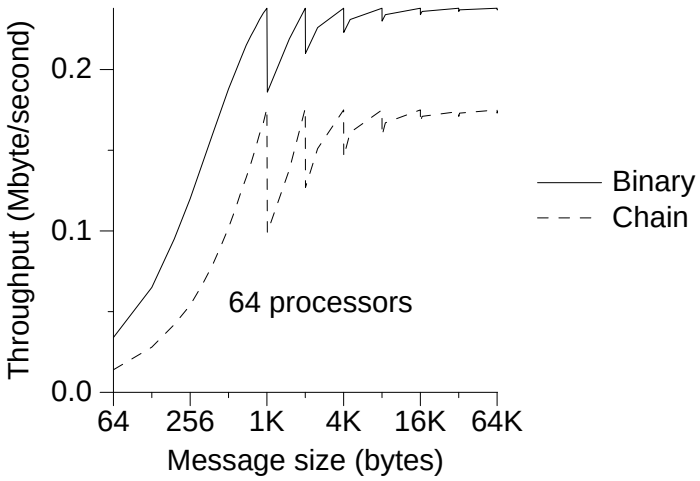


Fig. 5.10. Impact of deadlock recovery on all-to-all throughput.

Tree shape	$L_{\mathbf{used}}/L_{\mathbf{av}}$	RC/M	$D/RC+1$
Binary	0.51	0.38	3.3
Chain	0.02	1.00	63.9

Table 5.2. Deadlock recovery statistics. L_{used} — #NI-to-NI links used in this test; L_{av} — #NI-to-NI links available; RC — #deadlock recoveries; M — #multicasts; D — #packets that use a deadlock recovery channel.

ets that are forced to travel through a slow deadlock recovery channel. $D/RC+1$ is the average size of deadlocked subtrees. These statistics show that, with chains, every multicast triggers a deadlock recovery action. What is worse, these recoveries involve all 64 NIs, so each multicast packet uses a slow deadlock channel. With chains, this benchmark can use only a small fraction (0.02) of the available NI receive buffer space. In combination with the high communication load, this causes senders to run out of credits and triggers deadlock recoveries. With binary trees, which make better use of the available NI buffer space, deadlock recoveries occur less frequently and the size of the deadlocked subtrees is much smaller.

Summarizing, we find that deadlock recovery is triggered frequently, but that its effect on performance is modest when the communication load is moderate and true deadlocks do not occur. To reduce the number of false alarms, a more refined, timeout-based mechanism could be used [96]. This mechanism does not signal deadlock immediately when a blocked-sends queue becomes nonempty, but starts a timer and waits for a timeout to occur. The timer is canceled when a send descriptor is removed from the blocked-sends queue.

5.4 Summary

In this chapter, we have analyzed the performance of LFC using microbenchmarks. LFC's point-to-point performance is comparable to that of existing user-level communication systems and does not suffer much from the NI-level flow control protocol. Specifically, on the same hardware LFC achieves similar point-to-point latencies as Illinois Fast Messages (version 2.0), which uses a host-level variant of the same flow control protocol. Running the protocol between NIs, however, allows for a simple and efficient multicast implementation.

Due to our use of programmed I/O at the sending side, LFC cannot attain the maximum available throughput, but we believe that LFC's throughput is still sufficiently high that this is not a problem in practice. At least one study suggests that parallel applications are more sensitive to send and receive overhead than to throughput [102].

LFC's NI-level fetch-and-add implementation is not much faster than a host-level implementation would be, but has the advantage that it need never generate an interrupt.

LFC's NI-level multicast also avoids unnecessary interrupts (during packet forwarding) and eliminates an unnecessary data transfer. The binary trees used by the basic multicast protocol give good latency and throughput. Client systems that wish to optimize for either latency or throughput can use other tree shapes if they use LFC's extended multicast protocol which includes a deadlock recovery mechanism. We showed that the extended protocol allows clients to obtain the advantages of a specific tree shape such as the chain.

Chapter 6

Panda

This chapter describes the design and implementation of the Panda communication system on top of LFC. Panda is a multithreaded communication library that provides flow-controlled point-to-point message passing, remote procedure call, and totally-ordered broadcast. Since 1993, versions of Panda have been used by implementations of the Orca shared object system [9, 19, 124], the MPI message-passing system, a parallel Java system [99], an implementation of Linda tuple spaces on MPI [33], a subset of the PVM message-passing system [124], the SR programming language [124], and a parallel search system [121].

Portability and efficiency have been the main goals of all Panda implementations. Panda has been ported to a variety of communication architectures. The first Panda implementation [19] was constructed on a cluster of workstations, using the UDP datagram protocol. Since then, versions of Panda have been ported to the Amoeba distributed operating system [111], to a transputer-based system [65], to MPI for the IBM SP/2, to active messages for the Thinking Machines CM-5, to Illinois Fast Messages for Myrinet clusters [10], and to LFC, also for Myrinet clusters. This chapter, however, focuses on a single Panda implementation: Panda 4.0 on LFC. Panda 4.0 differs substantially from the original system which had no separate message-passing layer and had a different message interface [19].

The second goal, efficiency, conflicts with the first, portability. Whereas portability favors a modular, layered structure, efficiency dictates an integrated structure. In this chapter we demonstrate that Panda can be implemented efficiently (on LFC) without sacrificing portability. This efficiency results from Panda's flexible internal structure, carefully designed interfaces (both in Panda and LFC), exploiting LFC's communication mechanisms, and integrating multithreading and communication.

In this chapter, we show how Panda uses LFC's packet-based interface, NI-level multicast and fetch-and-add, and polling watchdog to implement the following abstractions:

1. Asynchronous upcalls. Since many PPSs require asynchronous message delivery, Panda delivers all incoming messages by means of asynchronous upcalls. To avoid the use of interrupts for each incoming message, Panda transparently and dynamically switches between using polling and interrupts. To choose the most appropriate mechanism, Panda uses thread-scheduling information.

2. Stream messages. LFC's packet-based communication interface allows a flexible implementation of *stream messages* [112]. Stream messages allow end-to-end pipelining of data transfers. Our implementation avoids unnecessary copying and decouples message notification and message consumption.

3. Totally-ordered broadcast. Panda provides a simple and efficient implementation of a totally-ordered broadcast primitive. The implementation builds on LFC's multicast and fetch-and-add primitives.

The first section of this chapter gives an overview of Panda 4.0. It describes Panda's functionality and its internal structure. Section 6.2 describes how Panda integrates communication and multithreading. Section 6.3 studies the implementation of Panda's message abstraction. Section 6.4 describes Panda's totally-ordered broadcast protocol. Section 6.5 reports on the performance of Panda on LFC and Section 6.6 discusses related work.

6.1 Overview

This section gives an overview of Panda's functionality, describes the internal structure of Panda, and discusses the key performance issues in Panda's implementation.

6.1.1 Functionality

Panda extends LFC's low-level message-passing functionality in several ways. First, processes that communicate using Panda exchange messages of arbitrary size rather than packets with a maximum size.

Second, Panda implements demultiplexing. Each Panda module (described later) allows its clients to construct communication endpoints and to associate a handler function with each endpoint. Senders direct their messages to such endpoints. When a message arrives at an endpoint, Panda invokes the associated message handler and passes the message to this handler. LFC, in contrast, dispatches all network packets that arrive at a processor to a single packet handler.

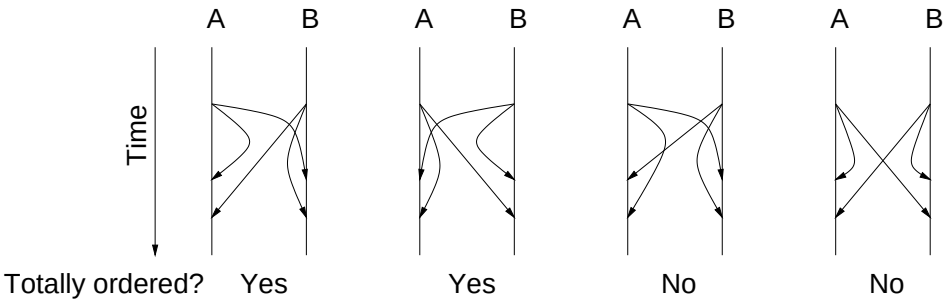


Fig. 6.1. Different broadcast delivery orders.

Third, Panda supports multithreading. Panda clients can create threads and use them to structure a system or to overlap communication and computation.

Finally, Panda supports several high-level communication primitives, all of which operate on messages of arbitrary length. With *one-way message passing*, messages can be sent from one process to an endpoint in another process. Such messages can be received implicitly, by means of upcalls, or explicitly, by means of blocking downcalls. *Remote procedure call*, a well known communication mechanism in distributed systems [23], allows a process to invoke a procedure (an upcall) in a remote process and wait for the result of the invocation. A *totally-ordered broadcast* is a broadcast with strong ordering guarantees. It has many applications in systems that need to maintain consistent replicas of shared data [75]. Specifically, a totally-ordered broadcast primitive guarantees that when n processes receive the same set of broadcast messages, then

1. All messages sent by the same process will be delivered to their destinations in the same order they were sent.

2. All messages (sent by any process) will be delivered in the same order to all n processes.

The first requirement (FIFOness) is also satisfied by LFC's broadcast primitive, but the second is not.

Figure 6.1 illustrates the difference between a FIFO broadcast and a totally-ordered broadcast. Two processes A and B concurrently broadcast a message. The figure shows all four possible delivery orders. With a FIFO broadcast, all delivery orders are valid. With a totally-ordered broadcast, however, only the first two scenarios are possible. In the third and fourth scenario, processes A and B receive the two messages in different orders.

Both message passing and RPC are subject to flow control. If a receiving process does not consume incoming messages, Panda will stall the sender(s) of

those messages. At present, no flow control is implemented for Panda's totally-ordered broadcast. Scalable multicast flow control is difficult due to the large number of receivers that a sender needs to get feedback from. LFC's current set of clients and applications, however, works fine without multicast flow control, for two reasons. First, due to application-level synchronization or due to the use of collective-communication operations, many applications have at most one broadcasting process at any time, which reduces the pressure on receiving processes. Second, a receiver that disables network interrupts and that does not poll, will eventually stall all nodes that try to send to it. If there is only a single broadcasting process, this back-pressure is sufficient to stall that process when needed. This is an all-or-nothing mechanism, though: either the receiver accepts incoming packets from all senders or it does not receive any packets at all.

Since Panda adds considerable functionality to LFC's simple unicast and multicast primitives, the question arises why this functionality is not part of LFC itself. The answer is that not all client systems *need* this extra functionality. The CRL DSM system (discussed in Section 7.3), for example, needs little more than efficient point-to-point message passing and interrupt management. Adding extra functionality would introduce unnecessary overhead.

6.1.2 Structure

Panda consists of several modules, which can be configured at compile time to build a complete Panda system. Figure 6.2 illustrates the module structure. Each box represents a module and lists its main functions. The arrows represent dependencies between modules. A detailed description of the module interfaces can be found in the Panda documentation [62].

The System Module

The most important module is the *system module*. For portability, Panda allows only the system module to access platform-specific functionality such as the native message-passing primitives. All other modules must be implemented using only the functionality provided by the system module.

The system module implements threads, endpoints, messages, and Panda's low-level unicast and multicast primitives. The thread and message abstractions are used in all Panda modules and by Panda clients. The unicast and multicast functions, however, are mainly used by Panda's message-passing and broadcast module to implement higher-level communication primitives. Panda clients use these higher-level primitives.

One way to achieve both portability and efficiency would be to define only a high-level interface that every Panda system must implement. The fixed interface

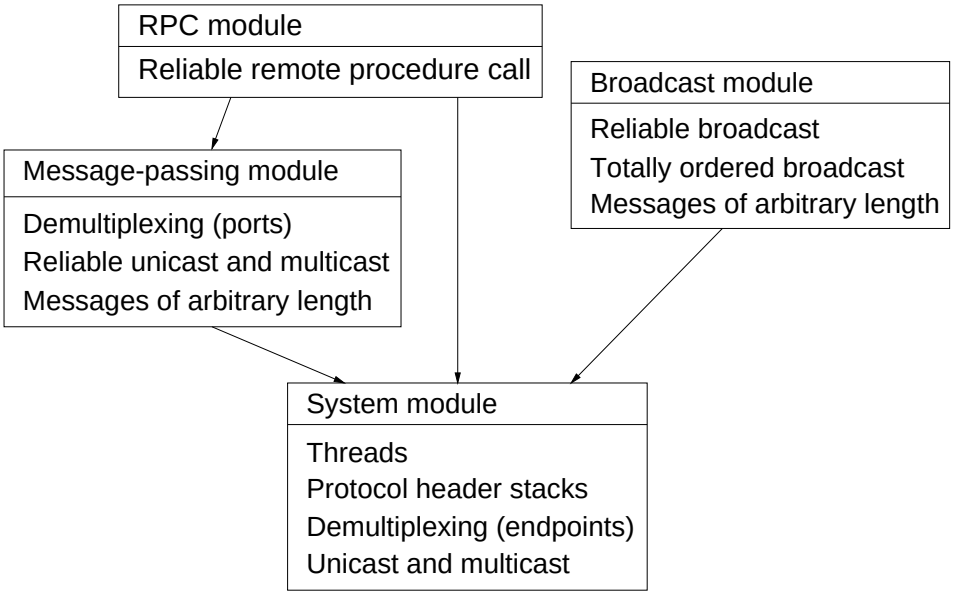


Fig. 6.2. Panda modules: functionality and dependencies.

ensures that Panda clients will run on any platform that the interface has been implemented on. Also, if only the top-level interface is defined, the implementation of Panda can exploit platform-specific properties. For example, if the message-passing primitive of the target platform is reliable, then Panda need not buffer messages for retransmission.

The main problem with this single-interface approach is that Panda must be implemented from scratch for every target platform. In practice platforms vary in a relatively small number of ways, so implementing Panda from scratch for every platform would lead to code duplication. To allow code reuse, Panda hides platform-specific code in the system module and requires that all other modules be implemented in terms of the system module. These modules can thus be reused on other platforms. The system-module functions have a fixed signature (i.e., name, argument types, and return type), but the exact semantics of these functions may vary in a small number of predefined ways from one Panda implementation to another. The system module exports the particular semantics of its functions by means of compile-time constants that indicate which features or restrictions apply to the target platform. This way, the system module can convey the main properties of the underlying platform to higher-level modules. The properties exported by the system module do not have to match those of the underlying system. For some platforms, including LFC, the system module sometimes exports a stronger interface than provided by the underlying system.

Variations of semantics is possible along the following dimensions:

1. Maximum packet size. The system module can export a maximum packet size and leave fragmentation and reassembly to higher-level modules, or it can accept and deliver messages of arbitrary size.

2. Reliability. The system module can export an unreliable or a reliable communication interface.

3. Message ordering. The system module can export FIFO or unordered communication primitives. The system module can also indicate whether or not its broadcast primitive is totally-ordered.

Panda's system module for LFC implements fragmentation and reassembly, both for unicast and broadcast communication. (In this respect, the system module thus exports a stronger interface than LFC.) The system module preserves the reliability and FIFOness of LFC's unicast and multicast primitives, and exports a reliable and FIFO interface to higher-level Panda modules. In addition, the system module implements a totally-ordered broadcast using LFC's fetch-and-add and broadcast primitives and exports this totally-ordered broadcast to higher-level modules. With this configuration, the implementation of the higher-level modules is relatively simple. Fragmentation, reliability, and ordering are all implemented in lower layers.

High-Level Modules

We discuss three higher-level modules: the message-passing module, the RPC module, and the broadcast module. These modules can exploit the information conveyed by the system module's (compile-time) parameters. This is the *only* information about the underlying system available to the higher-level modules. These modules can therefore be reused in Panda implementations for another platform, provided that the system module for that platform implements the same (or stronger) semantics. It is not necessary to build implementations of higher-level modules for every possible combination of system-module semantics. First, many combinations are unlikely. For example, no system module provides reliable broadcast and unreliable unicast. Second, we only need an implementation for the system module that exports the weakest semantics (i.e., unreliable and unordered communication). Such an implementation is suboptimal for system modules that offer stronger primitives, but it will function correctly and can serve as a starting point for optimization.

The *message-passing module* implements reliable point-to-point communication and an endpoint (demultiplexing) abstraction. Since LFC is reliable and Panda's system module performs fragmentation and reassembly, the message-

passing module need only implement demultiplexing, which amounts to adding a small header to every message.

The *RPC module* implements remote procedure calls on top of the message-passing module. An RPC is implemented as follows. The client creates a request message and transmits it to the server process using the message-passing module's send primitive. The RPC module then blocks the client (using a condition variable). When the server receives the request, it invokes the request handler. This handler creates a reply message and transmits it to the client process. At the client side, the reply is dispatched to a reply handler (internal to the RPC module) which awakens the blocked client thread and which passes the reply message to the awakened client. Since the implementation is built upon the message-passing module's reliable primitives, it is small and simple.

The *broadcast module* implements a totally-ordered broadcast primitive. All broadcast messages are delivered in the same total order to all processes, including the sender. Since the system module already implements a totally-ordered broadcast —so that it can exploit LFC's multicast and fetch-and-add— the implementation of the broadcast module is also very simple.

6.1.3 Panda's Main Abstractions

Below we describe the abstractions and mechanisms that are used throughout the Panda system and by Panda clients. On LFC, most of these abstractions are implemented in the system module.

Threads

Panda's multithreading interface includes the data types and functions that are found in most thread packages: thread create, thread join, scheduling priorities, locks, and condition variables. Since most thread packages provide similar mechanisms, Panda's thread interface can usually be implemented using wrapper routines that invoke the routines supplied by the thread package available on the target platform (e.g., POSIX threads [110]). The LFC implementation of Panda's thread interface uses a thread package called OpenThreads [63, 64]. Section 6.2 describes how Panda orchestrates the interactions between LFC and OpenThreads in an efficient way.

Addressing

The system module implements an addressing abstraction called Service Access Points (SAPs). Messages transmitted by the system module are addressed to a SAP in a particular destination process. Since Panda does not implement a

name service, all SAPs must be created by all processes at initialization time. The creator of a SAP (a Panda client or one of Panda's modules) associates a receive upcall with each SAP. When a message arrives at a SAP, this upcall is invoked with the message as an argument. Panda executes at most one upcall per SAP at a time. Upcalls of different SAPs may run concurrently.

Panda's high-level modules use similar addressing and message delivery mechanisms as the system module. The message-passing module, for example, implements an endpoint abstraction called ports. On LFC, ports are implemented as system module endpoints (SAPs), but on other platforms there need not be a one-to-one correspondence between ports and SAPs. In the Panda 4.0 implementation on top of UDP, for instance, SAPs are implemented as UDP sockets, but ports are not. The reason for this difference is that communication to UDP sockets is unreliable, while communication to a port is reliable.

As to message delivery, almost all modules deliver messages by means of upcalls associated with communications endpoints (SAPs, ports, etc.). The RPC module is an exception, because RPC replies are delivered synchronously to the thread that sent the request. This thread is blocked until the reply arrives. RPC requests, on the other hand, are delivered by means of upcalls.

Message Abstractions

Panda provides two message abstractions: I/O vectors at the sending side and *stream messages* at the receiving side. Both abstractions are used by all Panda modules.

A sending process creates a list of pointers to buffers that it wishes to transmit and passes this list to a send routine. The buffer list is called an I/O vector. Panda's low-level send routines (described below) *gather* the data referenced by the I/O vector by copying the data into network packets. The main advantage of a gather interface over interfaces that accept only a single buffer argument, is that they do not force the client to copy data into a contiguous buffer before invoking the send routine (which will have to copy the message at least once more, to the NI).

At the receiving side, Panda uses *stream messages*. Stream messages were introduced in Illinois Fast Messages (version 2.0) [112]. A stream message is a message of arbitrary length that can be accessed only sequentially (i.e., like a stream). A stream message behaves like a TCP connection that contains exactly one message. This property allows receivers of a message to begin consuming that message before it has been fully received. Incoming data can be copied to its final destination in a pipelined fashion, which is more efficient than first reassembling the complete message before passing it to the client.

Panda consists of multiple protocol layers and Panda clients may add their own protocol layers. Each layer typically adds a header to every message that

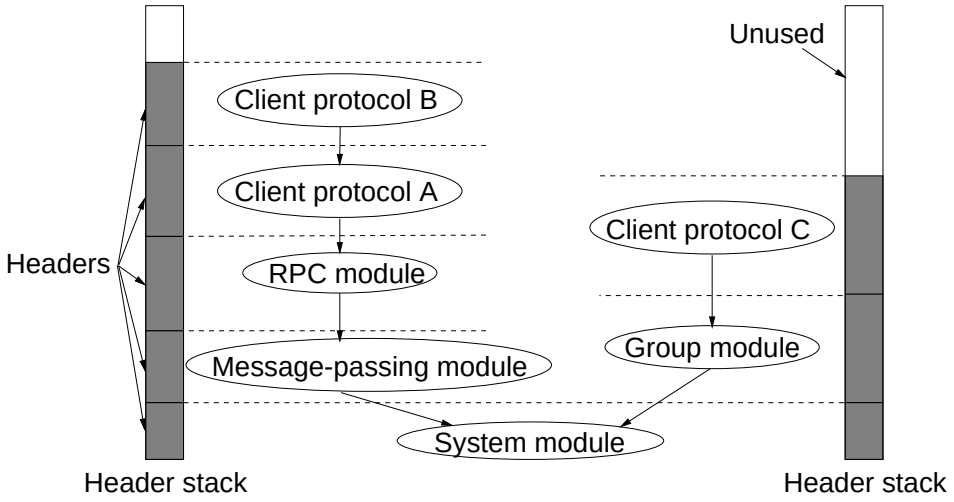


Fig. 6.3. Two Panda header stacks.

it transmits. To support efficient header manipulation, we borrow an idea from the *x*-kernel [116]. All protocol headers that need to be prepended to an outgoing message are stored in a single, contiguous buffer. By using a single buffer to store protocol headers, Panda avoids copying its clients' I/O vectors just to prepend pointers to its own headers (which are small). Such buffers are called header stacks and are used both at the sending and the receiving side. Senders write (push) their headers to the header stack. Receivers read (pop) those headers in reverse order. Both pushing and popping headers are simple and efficient operations.

Figure 6.3 shows two Panda protocol stacks and the corresponding header stacks. Each protocol layer, whether internal or external to Panda, exports how much space in the header stack it and its descendants in the protocol graph need. Other protocols can retrieve this information and use it to determine where in the header stack buffer they must store their header.

Send and Receive Interfaces

Table 6.1 gives the signatures of the system module's communication routines. *Pan_unicast*() gathers and transmits a header stack (*proto*) and an I/O vector (*iov*) to a SAP (*sap*) in the destination process (*dest*). When the destination process receives the first packet of the sender's message, it invokes the handler routine associated with the service access point parameter (*sap*). The handler routine takes a header stack and a stream message as its arguments. The receiving process can read the stream message's data by passing the stream message to *pan_msg_-consume*(). The header stack can be read directly.

void pan_unicast(unsigned dest, pan_sap_p sap, pan_iovec_p iov, int veclen, void *proto, int proto_size) Constructs a message out of all *veclen* buffers in I/O vector *iov*. Buffer *proto*, of length *proto_size*, is prepended to this message and is used to store headers. The message is transmitted to process *dest*. When *dest* receives the message, it invokes the upcall associated with *sap*, passing the message, its headers, and its sender as arguments.
void pan_multicast(pan_pset_p dests, pan_sap_p sap, pan_iovec_p iov, int veclen, void *proto, int proto_size) Behaves like *pan_unicast*(), but multicasts the I/O vector to all processes in *dests*, not just to a single process.
int pan_msg_consume(pan_msg_p msg, void *buf, unsigned size) Tries to copy the next *size* bytes from message *msg* to *buf* and returns the number of bytes that have been copied. This number equals *size* unless the client tried to consume more bytes than the stream contains. When the last byte has been consumed, the message is destroyed.

Table 6.1. Send and receive interfaces of Panda's system module.

Pan_msg_consume() consumes the next *n* bytes from the stream message and copies them to a client-specified buffer. If less than *n* bytes have arrived at the time that *pan_msg_consume*() is called, *pan_msg_consume*() blocks and polls until at least *n* bytes have arrived.

In many cases, the message handler reads all of a stream message's data. If this is inconvenient, however, then the receiving process can store the stream message, return from the handler, and consume the data later. Storing the stream message consists of copying a pointer to the message data structure; the contents of the message need not be copied.

Other Panda modules have similar send and receive signatures. In all cases, the send routine accepts an I/O vector with buffers to transmit and the receive upcall delivers a stream message and a protocol header stack.

Upcalls

The system module delivers every incoming message by invoking the upcall routine associated with the message's destination SAP. The upcall routine is usually a routine inside the higher-level module that created the SAP. In most cases, this routine passes the message to a higher-level layer (either the Panda client or an-

other Panda module). This is done by invoking another upcall or by storing the message (e.g., an RPC reply) in a known location. In other cases, the message (e.g., an acknowledgement) serves only control purposes internal to the receiving module and will not be propagated to higher-level layers.

When the system module receives the first packet of a message, it schedules an upcall. Each SAP maintains a queue of pending upcalls, which are executed in-order and one at a time. Clients and higher-level modules do not have precise control over the scheduling of upcalls. They must assume that every upcall is executed asynchronously and protect their global data structures accordingly.

An interesting question is whether an upcall should be viewed as an independent thread or as a subroutine of the running 'computation' thread. The former view is more natural, because the next incoming message may have nothing to do with the current activity of the running thread. Unfortunately, this view has some problems associated with it. These problems and their resolution in Panda are discussed in Section 6.2.

6.2 Integrating Multithreading and Communication

This section studies the interactions between LFC's communication mechanisms and multithreading in Panda. Section 6.2.1 explains how Panda and other multithreaded clients can be implemented *safely* on LFC. Next, in Section 6.2.2 we show how the knowledge that is available in a thread scheduler can be exploited to reduce interrupt overhead. Finally, in Section 6.2.3 we discuss design options and implementation techniques for upcall models. The design of an upcall model is related to multithreading and efficient implementations of some upcall models require cooperation of the multithreading system. After discussing different design options, we describe Panda's upcall model.

6.2.1 Multithread-Safe Access to LFC

LFC is not multithread-safe: concurrent invocations of LFC routines can corrupt LFC's internal data structures. LFC does not use locks to protect its global data, because several LFC clients (e.g., CRL, TreadMarks, and MPI) do not use multithreading and because we are reluctant to make LFC dependent on specific thread packages. Nevertheless, Panda and other multithreaded systems can safely use LFC without modifications to LFC's interface or implementation.

With single-threaded clients LFC has to be prepared for two types of concurrent invocation of LFC routines: recursive invocations and invocations executed by LFC's signal handler. Recursive invocations occur (only) when an LFC routine drains the network (see Section 3.6). The routine that sends a packet, for exam-

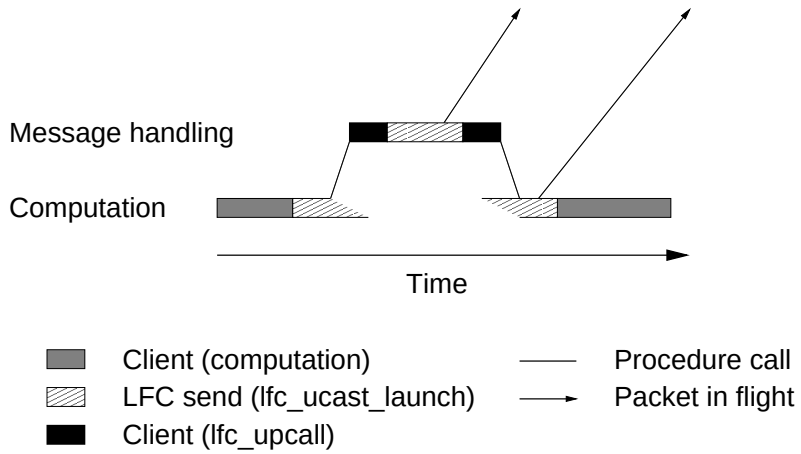


Fig. 6.4. Recursive invocation of an LFC routine.

ple, will poll when no free send descriptors are available. During a poll, LFC may invoke *lfc_upcall*() to handle incoming network packets. This upcall is defined by the client and may invoke LFC routines, including the routine that polled. This scenario is illustrated in Figure 6.4. To prevent corruption of global state due to recursion, LFC always leaves all global data in a consistent state before polling.

When we allow network interrupts, LFC routines can be interrupted by an upcall at any point, which can easily result in data races. To avoid such data races, LFC logically[1] disables network interrupts whenever an LFC routine is entered that accesses global state.

The two measures described above are insufficient to deal with multithreaded clients, because they do not prevent two client threads from concurrently entering LFC. To solve this problem, Panda's system module employs a single lock to control access to LFC routines. Panda acquires this lock before invoking any LFC routine and releases the lock immediately after the routine returns.

The lock introduces a new problem: recursive invocations of an LFC routine will deadlock, because Panda does not allow a thread to acquire a lock multiple times (i.e., Panda does not implement recursive locks). To prevent this, Panda releases the lock upon entering *lfc_upcall*() and acquires it again just before returning from *lfc_upcall*(). This works, because all recursive invocations have *lfc_upcall*() in their call chain. When *lfc_upcall*() releases the lock, another thread can enter LFC. This is safe, because the polling thread that invoked *lfc_upcall*() always leaves LFC's global variables in a consistent state.

With these measures, we can handle concurrent invocations from multiple Panda threads and recursive invocations by a single thread. There remains one

[1] As explained in Section 3.7, LFC does not truly disable network interrupts.

problem: network interrupts. Network interrupts are processed by LFC's signal handler (see Section 3.7). When invoked, this signal handler checks that interrupts are enabled (otherwise it returns) and then invokes *lfc_poll*() to check for pending packets. *Lfc_poll*(), in turn, may invoke *lfc_upcall*() and will do so before Panda has had a chance to acquire its lock. When this occurs, *lfc_upcall*() will release the lock even though the lock had never been acquired.

The problem is that the signal handler is another entry point into LFC that needs to be protected with a lock/unlock pair. To do that, Panda overrides LFC's signal handler with its own signal handler. Panda's signal handler acquires the lock, invokes LFC's signal handler, and releases the lock. This closes the last atomicity hole.

Summarizing, Panda achieves multithread-safe access to LFC through three measures:

1. LFC leaves its global variables in a consistent state before polling.
2. The client brackets calls to LFC routines with a lock/unlock pair to prevent multiple client threads from executing concurrently within LFC.
3. The client brackets invocations of LFC's network signal handler with a lock/unlock pair.

Two properties of LFC make these measures work. First, LFC does not provide a blocking receive downcall, but dispatches all incoming packets to a user-supplied upcall function. Systems like MPI, in contrast, provide a blocking receive downcall. If we bracket calls to such a blocking receive with a lock/unlock pair, we block entrance to the communication layer until a message is received. This causes deadlock if a thread blocked in a receive downcall waits for a message that can be received only if another thread is able to enter the communication system (e.g., to send a message). This type of blocking is different from the transient blocking that occurs when LFC has run out of some resource (e.g., send packets). The latter type of blocking is always resolved if all participating processes drain the communication network and release host receive-packets.

Second, LFC dispatches all packets to a *single* upcall function, which allows lock management to be centralized. If the communication system would directly invoke user handlers, then it would be the responsibility of the user to get the locking right, or, alternatively, LFC would have to know about the locks used by the client system.

6.2.2 Transparently Switching between Interrupts and Polling

In Section 2.5 we discussed the main advantages and disadvantages of polling and interrupts. This section presents an approach, implemented in Panda and its

underlying thread package OpenThreads, that combines the advantages of polling and interrupts. In this approach polling is used when the application is known to be idle. Interrupts are used to deliver messages to a process that is busy executing application code. Using the thread scheduler's knowledge of the state of all threads, Panda dynamically (and transparently) switches between these two strategies. As a result, Panda never polls unnecessarily and we use interrupts only when the application is truly busy.

The techniques described in this section were initially developed on another user-level communication system (FM/MC [10]). The same techniques, however, are used on LFC. In addition, LFC supports the mixing of polling and interrupts by means of its polling watchdog.

Polling versus Interrupts

Choosing between polling and interrupts can be difficult. There are difficulties in two areas: ease of programming and performance cost. We consider two issues related to ease of programming: matching the polling rate to the message-arrival rate and concurrency control.

With polling, it is necessary to roughly match the polling rate to the message-arrival rate. Unfortunately, many parallel programming systems cannot predict when messages arrive and when they need to be processed. These systems must either use interrupts or insert polls automatically. Since interrupts are expensive, a polling-based approach is potentially attractive. Automatic polling, however, has its own costs and problems.

Polls can be inserted statically, by a compiler, or dynamically, by a runtime system. A compiler must be conservative and may therefore insert far too many polling statements (e.g., at the beginning of every loop iteration). A runtime system can poll the network each time it is invoked, but this approach works only if the application frequently invokes the runtime system. Alternatively, the runtime system can create a background thread that regularly polls the network. This, however, requires that the thread scheduler implement a form of time-slicing, and introduces the overhead of switching to and from the background thread. Polling is also troublesome from a software-engineering perspective, because *all* code, including large standard libraries, may have to be processed to include polls.

The second issue is concurrency control. Unlike interrupts, using polls gives precise control over message-handler execution. Consequently, a single-threaded application that polls the network only when it is not in a critical section need not use locks or interrupt-status manipulation to protect its shared data. However, to exploit the synchronous nature of polling, one must know exactly where a poll may occur. Calling a library routine from within a critical section can be dangerous, unless it is guaranteed that this routine does not poll.

Quantifying the difference in cost between using interrupts and polling is difficult due to the large number of parameters involved: hardware (cache sizes, network adapters), operating system (interrupt handling), runtime support (thread packages, communication interfaces), and application (polling policy, message-arrival rate, communication patterns). The discussion below considers the base costs of polling and interrupts, and the relationship with the message-arrival rate.

First, executing a single poll is typically much cheaper than taking an interrupt, because a poll executes entirely in user space without any context switching (see Section 2.5). Dispatching an interrupt to user space on a commodity operating system, on the other hand, is expensive. The main reason is that software interrupts are typically used to signal exceptions like segmentation faults, events for which operating systems do not optimize [143]. In LFC, a successful poll costs 1.0 μs; dispatching an interrupt and a signal handler costs 31 μs.

Second, comparing the cost of a single poll to the cost of a single interrupt does not provide a sufficient basis for statements about application performance. A single interrupt can process many messages, so the cost of an interrupt can be amortized over multiple messages. Also, each time a poll fails, the user program wastes a few cycles. Since matching the polling rate to the message-arrival rate can be hard, an application may either poll too often (and thus waste cycles) or poll too infrequently (and delay the delivery of incoming messages).

For Panda, the following observations apply. Since Panda is multithreaded, many Panda clients will already use locks to protect accesses to global data. Such clients will have no trouble when upcalls are scheduled preemptively and do not get any benefit from executing upcalls at known points in time. Moreover, several of Panda's clients cannot predict when messages will arrive, yet need to respond to those messages in a timely manner. These two (ease-of-use) observations suggest an interrupt-driven approach. However, since we expect that the exclusive use of interrupts will lead to bad performance for clients that communicate frequently, polling should also be taken into consideration. Below, we describe how Panda tries to get the best of both worlds.

Exploiting the Thread Scheduler

Both polling and interrupts have their advantages and it is often beneficial to combine the two of them. LFC provides the mechanisms to switch dynamically between polling and interrupts: primitives to disable and enable interrupts, and a polling primitive. Using these primitives, a sophisticated programmer can get the advantages of both polling and interrupts. The CRL runtime system, for example, takes this approach [73]. Unfortunately, this approach is error-prone: the programmer may easily forget to poll or, worse, to disable interrupts.

To solve this problem, we have devised a simple strategy for switching be-

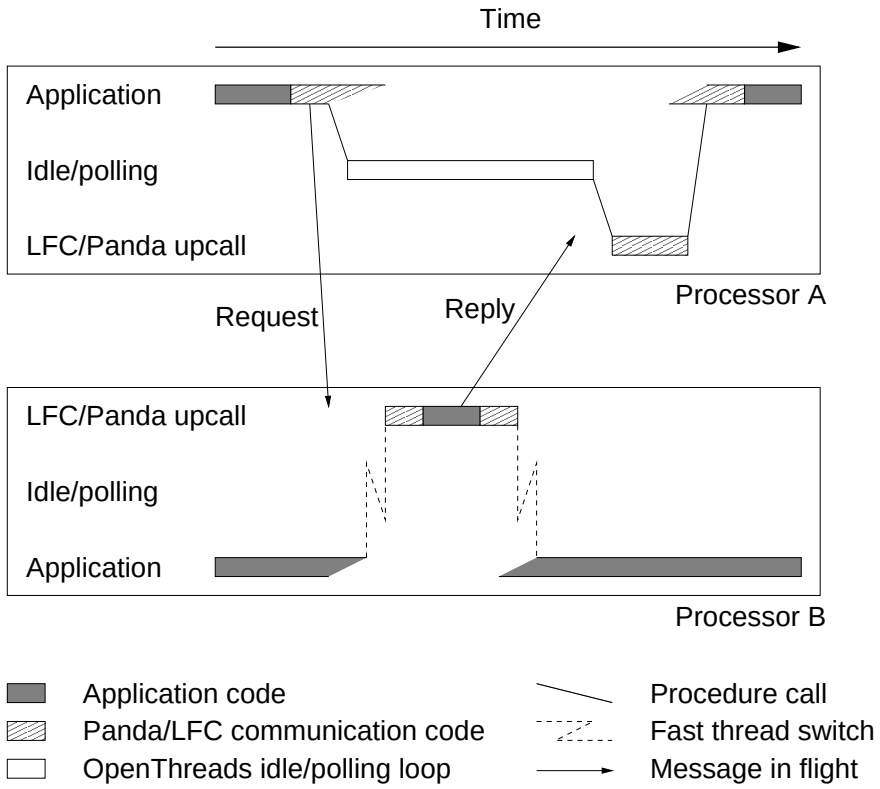


Fig. 6.5. RPC example.

tween the two message extraction mechanisms. The key idea is to integrate thread management and communication. This integration is motivated by two observations. First, polling is always beneficial when the application is idle (i.e., when it waits for an incoming message). Second, when the application is active, polling may still be effective from a performance point of view, but inserting polls into computational code and finding the right polling rate can be tedious. Thus, our strategy is simple: we poll whenever the application is idle and use interrupts whenever runnable threads exist. To be able to poll when the application is idle, however, we must be able to detect this situation. This suggests that the decision to switch between polling and interrupts be taken in the thread scheduler.

We implemented this strategy in the following way. By default, Panda uses interrupts to receive messages. However, when OpenThreads's thread scheduler detects that all threads are blocked, it disables network interrupts and starts polling the network in a tight loop. A successful poll results in the invocation of *lfc_upcall*(). If the execution of the handler wakes up a local thread, then the scheduler will re-enable network interrupts and schedule the awakened thread. If an application thread is running when a message arrives, LFC will generate a network signal. Since LFC delays the generation of network interrupts, it is unlikely that interrupts are generated unnecessarily (see Section 3.7).

OpenThreads does not detect all types of blocking: a thread is considered blocked only if it blocks by invoking one of Panda's thread synchronization primitives (*pan_mutex_lock*(), *pan_cond_wait*(), or *pan_thread_join*()). If a thread spin-waits on a memory location then this is not detected by OpenThreads. Also, OpenThreads is not aware of blocking system calls. If a system call blocks, it will block the entire process and not just the thread that invoked the system call. Some thread packages solve this problem by providing wrappers for blocking system calls. The wrapper invokes the asynchronous version of the system call and then invokes the scheduler, which can then block the calling thread. When a poll or a signal indicates that the system call has completed, the scheduler can reschedule the thread that made the system call.

Figure 6.5 illustrates a typical RPC scenario. A thread on processor A issues an RPC to remote processor B, which also runs a computation thread. After sending its request, the sending thread blocks on a condition variable and Panda invokes the thread scheduler. The scheduler finds no other runnable threads, disables interrupts, and starts polling the network.

When the request arrives at processor B, it interrupts the running application thread. (Since processor B has an active thread, interrupts are enabled.) The request is then dispatched to the destination SAP's message handler. This handler processes the request, sends a reply, and returns.

On processor A, the polling thread receives the reply, enters the message-passing library, and then signals the condition variable that the application thread

blocked on. When the handler thread dies, the scheduler re-enables interrupts, because the application thread has become runnable again. Next, the scheduler switches to the application thread and starts to run it.

Summarizing, the behavior of the integrated threads and communication system is that synchronous communication, like the receipt of an RPC reply, is usually handled through polling. Asynchronous communication, on the other hand, is mostly handled through interrupts. This mixing of polling and interrupts combines nicely with LFC's polling watchdog.

6.2.3 An Efficient Upcall Implementation

In this section we discuss the design options for upcall implementations. Next, we discuss three upcall models that take different positions in the design space. The properties of these models have influenced the design of Panda's upcall model, which is discussed toward the end of this section.

The two main design axes for upcall systems are upcall context and upcall concurrency. Upcalls can be invoked from different processing contexts. A simple approach is to invoke upcall routines from the thread that happens to be running when a message arrives. We call this the procedure call approach. The alternative approach, the thread approach, is to give each upcall its own execution context. This amounts to allocating a thread per incoming message. The exact context from which a system makes upcalls affects both the programming model and the efficiency of upcalls.

Upcall concurrency determines how many upcalls can be active concurrently in a single receiver process. The most restrictive policy allows at most one upcall to execute at any time; the most liberal strategy allows any number of upcalls to execute concurrently. Again, different choices lead to differences in the programming model and performance.

Table 6.2 summarizes the design axes and the different positions on these axes. The table also classifies three upcall models along these axes: active message, single-threaded upcalls, and popup threads. The *active-messages* model [148] is restrictive in that it prohibits all blocking in message handlers. In particular, active-message handlers may not block on locks, which complicates programming significantly. On the other hand, highly efficient implementations of this model exist. Some of these implementations allow active-message handler invocations to nest, others do not. *Popup threads* [116] allow message handlers to block at any time. Popup threads are often slower than active messages, because conceptually a thread is created for each message. Finally, we consider *single-threaded upcalls* [14], which disallow blocking in some, but not all cases. Below, we compare the expressiveness of these models and the cost of implementing them.

Concurrency	Context	
	Procedure call	**Thread**
One upcall at a time	Active messages	Single-threaded upcalls
Multiple upcalls	Active messages	Popup threads

Table 6.2. Classification of upcall models.

```
int x, y;

void
handle_store(int *x_addr, int y_val, int z0, int z1)
{
        *x_addr = y_val;
}

void
handle_read(int src, int *x_addr, int z0, int z1)
{
        AM_send_4(src, handle_store, x_addr, y, 0, 0);
}

...
/* send read request to processor 5 */
AM_send_4(5, handle_read, my_cpu, &x, 0, 0);
...
```

Fig. 6.6. Simple remote read with active messages.

Active Messages

As explained in Section 2.8, an active message consists of the address of a handler function and a small number of data words (typically four words). When an active message is received, the handler address is extracted from the message and the handler is invoked; the data words are passed as arguments to the handler. For large messages, the active-messages model provides separate *bulk-transfer* primitives.

A typical use of active messages is shown in Figure 6.6. In this example, processor *my_cpu* sends an active message to read the value of variable *y* on processor 5. To send the message, we use the hypothetical routine *AM_send_4*(), which accepts a destination, a handler function, and four word-sized arguments. At the receiving processor, the handler function is applied to the arguments. In this case, the message contains the address of the handler *handle_read*(), the iden-

tity of the sender (*my_cpu*), and the address at which the result of the read should be stored (*&x*). Function *handle_read*() will be invoked on processor 5 with *src* set to *my_cpu* and *x_addr* set to *&x*. The handler replies with another active message that contains the value of *y*. When this reply arrives at processor *my_cpu*, *handle_store*() is invoked to store the value in variable *x*. We assume that reading an integer (variable *y*) is an atomic operation.

Active-message implementations deliver performance close to that of the raw hardware. An important reason for this high performance is that active-message handlers do not have their own execution context. When an application thread polls or is interrupted by a network signal, the communication system invokes the handler(s) of incoming messages in the context of the application thread. That is, the handler's stack frames are simply stacked on top of the application frames (see Figure 6.8(a)). No separate thread is created, which eliminates the cost of building a thread descriptor, allocating a stack, and a thread switch.

The lack of a dedicated thread stack makes active messages efficient, but makes them unattractive as a general-purpose communication mechanism for application programmers. Active-message handlers are run on the stacks of application threads. If an active-message handler blocks, then the application thread cannot be resumed, because part of its stack is occupied by the active-message handler (see Figure 6.8(a)). This type of blocking occurs when the application thread holds a resource (e.g., a lock) that is also needed by the active-message handler. Clearly, a deadlock is created if the active-message handler waits until the application thread releases the resource.

Similar problems arise when a handler waits for the arrival of another message. Consider for example the transmission of a large reply message by an upcall handler. If the message is sent by means of a flow-controlled protocol, then the send routine may block when its send window closes. The send window can be reopened only after an acknowledgement has been processed, which requires another upcall. If the active-messages system allows at most one upcall at a time, then we have a deadlock. If the system allows multiple upcalls, however, then the acknowledgement handler can be run on top of the blocked handler (i.e., as a nested upcall) and no deadlock occurs.

These problems are not insolvable. If it is necessary to suspend a handler, the programmer can explicitly save state in an auxiliary data structure, a *continuation*, and let the handler return. The state saved in the continuation can later be used by a local thread or another handler to resume the suspended computation. We assume that continuations are created manually by the programmer. Sometimes, though, it is possible to create continuations automatically. Automatic approaches either rely on a compiler to identify the state to be saved or save state conservatively. If a compiler recognizes potential suspension points in a program, then it can identify live variables and generate code to save live variables when the computation is

suspended. Blocking a thread is an example of conservative state saving: the saved state consists of the entire thread stack.

To select between these state-saving alternatives, the following tradeoffs must be considered. With manual state saving, no compiler is needed and application-specific knowledge can be exploited to minimize the amount of state saved. However, manual state saving can be tedious and error prone. Compilers can remove this burden, but most communication systems are constructed as libraries rather than languages. Finally, blocking an entire thread is simple, but requires that every upcall be run in its own thread context, otherwise we will also suspend the computation on which the upcall has been stacked. This alternative is discussed below, in the section on popup threads.

In small and self-contained systems, continuations can be used effectively to solve the problem of blocking upcalls (see Section 7.1). On the other hand, continuations are too low-level and error prone to be used by application programmers. In fact, the original active-message proposal [148] explicitly states that the active-message primitives are not designed to be high-level primitives. Active messages *are* used in implementations of several parallel programming systems. A good example is Split-C [39, 148], an extension of C that allows the programmer to use global pointers to remote words or arrays of data. If a global pointer to remote data is dereferenced, an active message is sent to retrieve the data.

Single-Threaded Upcalls

In the single-threaded upcall model [14], all messages sent to a particular process are processed by a single, dedicated thread in that process. We refer to this thread as the *upcall thread*. Also, in its basic form, the single-threaded upcall model allows at most one upcall to execute at a time.

Figure 6.7 shows how single-threaded upcalls can be used to access a distributed hash table. Such a table is often used in distributed game-tree searching to cache evaluation values of board positions that have already been analyzed. To look up an evaluation value in a remote part of the hash table, a process sends a *handle_lookup* message to the processor that holds the table entry. Since the table may be updated concurrently by local worker threads, and because an update involves modifying multiple words (a key and a value), each table entry is protected by a lock. In contrast with the active-messages model, the handler can safely block on this lock when it is held by some local thread. While the handler is blocked, though, no other messages can be processed. The single-threaded upcall model assumes that locks are used only to protect small critical sections so that pending messages will not be delayed excessively.

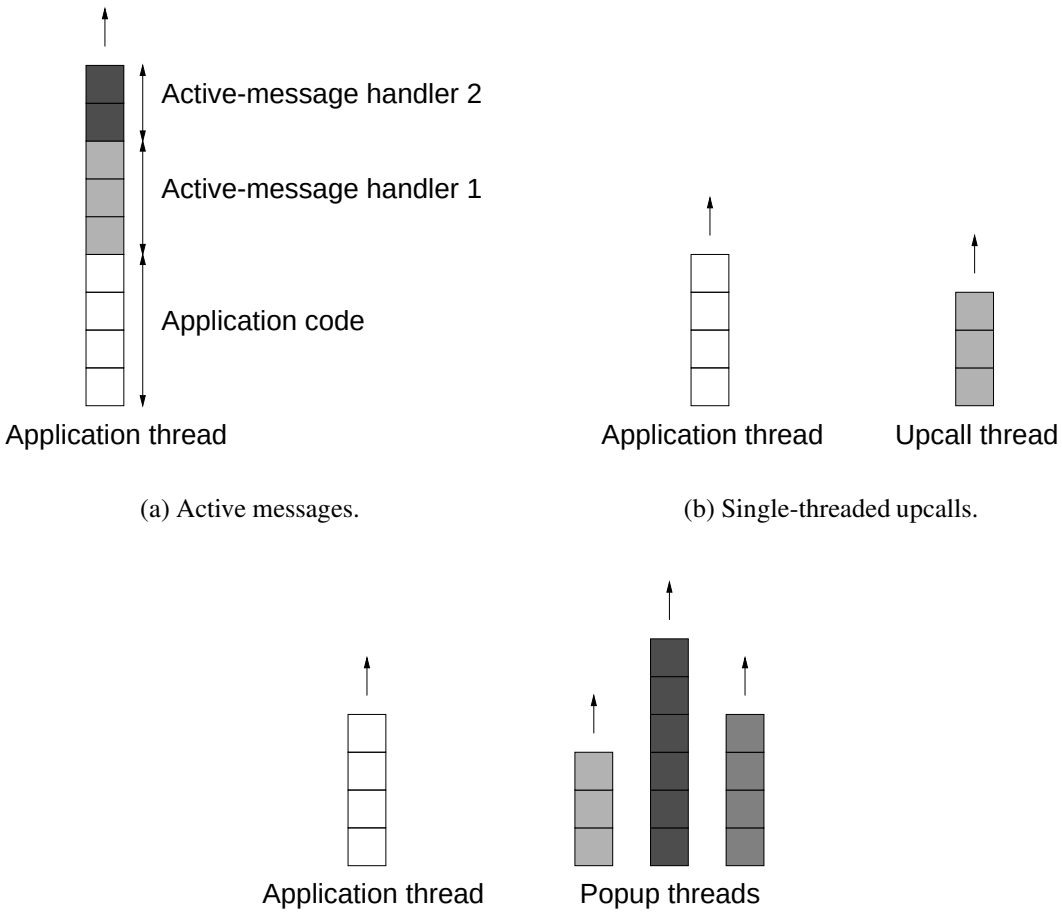
Allowing at most one upcall at a time restricts the set of actions that a programmer can safely perform in the context of an upcall. Specifically, this policy

```
void
handle_lookup(int src, int key, int *ret_addr, int z0)
{
        int index;
        int val;

        index = hash(key);

        lock(table[index].lock);
        if (table[index].key == key) {
                val = table[index].value;
        } else {
                val = -1;
        }
        unlock(table[index].lock);

        AM_send_4(src, handle_reply, ret_addr, val, 0, 0);
}
```

Fig. 6.7. Message handler with locking.

does not allow an upcall to wait for the arrival of a second message, because this arrival can be detected only if the second message's upcall executes. For example, if a message handler issues a remote procedure call to another processor, deadlock would ensue because the handler for the RPC's reply message cannot be activated until the handler that sent the request returns. In practice, the single-threaded upcall model requires that message handlers create continuations or additional threads in cases where condition synchronization or synchronous communication is needed.

The difference between single-threaded upcalls and active messages is illustrated in Figure 6.8. With active messages, upcall handlers are stacked on the application's execution stack. Some implementations allow upcall handlers to nest, others do not. Single-threaded upcall handlers, in contrast, do not run on the stack of an arbitrary thread; they are always run, one at a time, on the stack of the upcall thread. Executing message handlers in the context of this thread allows the handlers to block *without* blocking other threads on the same processor. In particular, the single-threaded upcall model allows message handlers to use locks to synchronize their shared-data accesses with the accesses of other threads. This is an important difference with active messages, where all blocking is disallowed. If an active-message handler blocked, it would occupy part of the stack of another thread (see Figure 6.8(a)), which then cannot be resumed safely.



(a) Active messages.

(b) Single-threaded upcalls.

(c) Popup threads.

Fig. 6.8. Three different upcall models.

```
void
handle_job_request(int src, int *jobaddr, int z0, int z1)
{
        int job_id;

        lock(queue_lock);
        while (is_empty(job_queue)) {
                wait(job_queue_nonempty, queue_lock);
        }
        job_id = fetch_job(job_queue);
        unlock(queue_lock);

        AM_send_4(src, handle_store, jobaddr, job_id, 0, 0);
}
```

Fig. 6.9. Message handler with blocking.

The single-threaded upcall model has been implemented in several versions of Panda. The current version of Panda, Panda 4.0, uses a variant of the single-threaded upcall model that we will describe later.

Popup Threads

While the single-threaded upcall model is more expressive than the active-messages model, it is still a restrictive model because all messages are handled by a single thread. The popup-threads model [116], in contrast, allows multiple message handlers to be active concurrently. Each message is allocated its own thread context (see Figure 6.8(c)). As a result, each message handler can synchronize safely on locks and condition variables and issue (possibly synchronous) communication requests, just like any other thread.

Figure 6.9 illustrates the advantages of popup threads. In this example, the message handler *handle_job_request*() attempts to retrieve a job identifier (*job_id*) from a job queue. When no job is available, the handler blocks on condition variable *job_queue_nonempty* and waits until a new job is added to the queue. While this is a natural way to express condition synchronization, it is prohibited in both the active-messages and the single-threaded upcall model. In the active-messages model, the handler is not even allowed to block on the queue lock. In the single-threaded upcall model, the handler can lock the queue, but is not allowed to wait until a job is added, because the new job may arrive in a message not yet processed. To add this new job, it is necessary to run a new message handler while the current handler is blocked. With single-threaded upcalls, this is impossible.

Dispatching a popup thread need not be any more expensive than dispatching a single-threaded upcall [88]. Popup threads, however, have some hidden costs that do not immediately show up in microbenchmarks:

1. When many message handlers block, the number of threads in the system can become large, which wastes memory.

2. The number of runnable threads on a node may increase, which can lead to scheduling anomalies. In earlier experiments we observed a severe performance degradation for a search algorithm in which popup threads were used to service requests for work [88]. The scheduler repeatedly chose to run high-priority popup threads instead of the low-priority application thread that generated new work. As a result, many useless thread switches were executed. Druschel and Banga found similar scheduling problems in UNIX systems that process incoming network packets at top priority [47].

3. Because popup threads allow multiple message handlers to execute concurrently, the ordering properties of the underlying communication system may be lost. For example, if two messages are sent across a FIFO communication channel from one node to another, the receiving process will create two threads to process these messages. Since the thread scheduler can schedule these threads in any order, the FIFO property is lost. In general, only the programmer knows when the next message can safely be dispatched. Hence, if FIFO ordering is needed, it has to be implemented explicitly, for example by tagging messages with sequence numbers.

Several systems provide popup threads. Among these systems are Nexus [54], Horus [144], and the *x*-kernel [116]. These systems have all been used for a variety of parallel and distributed applications.

Panda's Upcall Model

The first Panda system [19] implemented popup threads. To reduce the overhead of thread switching, however, later versions have used the single-threaded upcall model. Two kinds of thread switching overhead were eliminated:

1. In our early Panda implementations on Unix and Amoeba [139] all incoming messages were dispatched to a single thread, the network daemon, which passed each message to a popup thread. (These systems maintained a pool of popup threads to avoid thread creation costs on the critical path.) The use of single-threaded upcalls allowed the network daemon to process all messages without switching to a popup thread.

2. Multiple upcall threads could be blocked, waiting for an event that was to be generated by a local (computation) thread or a message. When the event occurred, all threads were awakened, even if the event allowed only one thread to continue; the other threads would put themselves back to sleep. It is frequently possible to avoid putting upcall threads to sleep. Instead, upcalls can create continuations which can be resumed by other threads without any thread switching. Once upcalls do not put themselves to sleep any more, they can be executed by a single-threaded upcall instead of popup threads.

In retrospect, the first type of overhead could have been eliminated without changing the programming model, by means of lazy thread-creation techniques such as described below. The second problem occurred in the implementation of Orca and is discussed in more detail in Section 7.1.4.

Panda 4.0 implements an upcall model that lies between active messages and single-threaded upcalls. The model is as follows. First, as in single-threaded upcalls, Panda clients can use locks in upcalls, but should not use condition-synchronization or synchronous communication primitives in upcalls.

Second, Panda clients must release their locks before invoking any Panda send or receive routine. This restriction is the main difference between Panda's upcall model and single-threaded upcalls. It sometimes allows the implementation to run upcalls on the current stack rather than on a separate stack.

Finally, Panda allows up to one upcall per endpoint to be active at any time. Panda clients must therefore be prepared to deal with concurrency between upcalls for different endpoints. In practice, this poses no problems, because programmers already have to deal with concurrency between upcalls and a computation thread or between multiple computation threads. Since Panda runs only upcalls for different endpoints concurrently, the order of messages sent to any single endpoint is preserved, so we avoid a disadvantage of popup threads.

Implementation of Panda's Upcall Model

On LFC, we use an optimized implementation of Panda's upcall model. In many cases, this implementation completely avoids thread switching during message processing. The implementation distinguishes between synchronous and asynchronous upcalls. Synchronous upcalls occur implicitly, as the result of polling by an LFC routine, or explicitly as the result of an invocation of *lfc_poll*() by Panda. Panda polls explicitly when all threads are idle or when it tries to consume an as yet unreceived part of a stream message (see Section 6.3). A successful poll results in the (synchronous) invocation of *lfc_upcall*() in the context of the current thread (or the last-active thread if all threads are idle). This synchronous

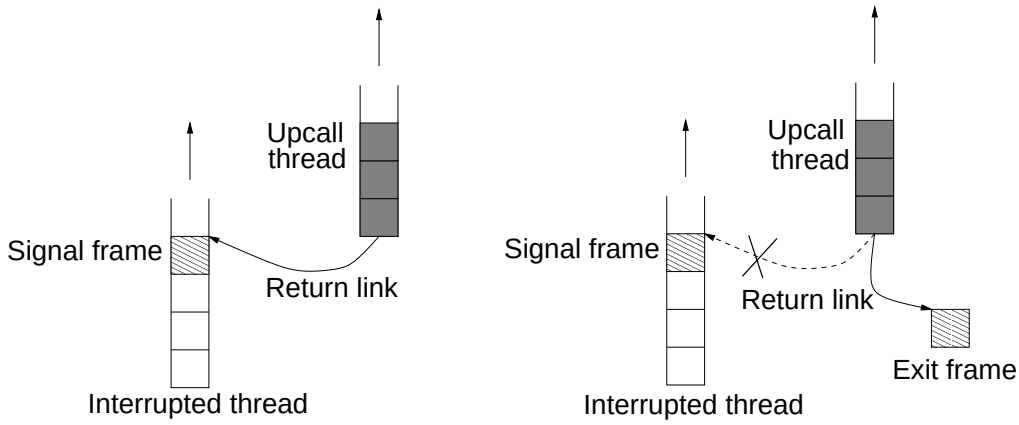
upcall queues the packet just received at the packet's destination SAP. If this is the first packet of a message, and no other messages are queued before this message, then Panda invokes the SAP's upcall routine (by means of a plain procedure call) without switching to another stack. This is safe, because Panda's upcall model requires that the polling thread does not hold any locks.

Asynchronous upcalls occur when the NI generates a network interrupt. Network interrupts are propagated to the receiving user process by means of a UNIX signal. The signal handler interrupts the running thread and executes on that thread's stack. Eventually, the signal handler invokes *lfc_poll*() to process the next network packet. In this case, there is no guarantee that the interrupted thread does not hold any locks, so we cannot simply invoke a SAP's upcall routine. A simple solution is to store the packet just received in a queue and signal a separate upcall thread. Unfortunately, this involves a full thread switch.

To avoid this full thread switch, OpenThreads invokes *lfc_poll*() by means of a special, slightly more expensive procedure-call mechanism. Instead of running *lfc_poll*() on the stack of the interrupted thread, OpenThreads switches to a special upcall stack and runs the poll routine on that stack. At first sight, this is just a thread switch, but there are two differences. First, since the upcall stack is used only to run upcalls, OpenThreads does not need to restore any registers when it switches to this stack. Put differently, we always start executing at the bottom of the upcall stack. Second, OpenThreads sets up the bottom stack frame of the upcall stack in such a way that the poll routine will automatically return to the signal handler's stack frame on the top of the stack of the interrupted thread (see Figure 6.10(a)).

If the SAP handler does not block, then we will leave behind an empty upcall stack, return to the stack of the interrupted thread, return from the signal handler, and resume the interrupted thread. This is the common case that OpenThreads optimizes. If, on the other hand, the SAP handler blocks on a lock, then OpenThreads will disconnect the upcall stack from the stack of the interrupted thread. This is illustrated in Figure 6.10(b). OpenThreads modifies the return address in the bottom stack frame so that it points to a special exit function. This prevents the upcall from returning to the stack of the interrupted thread and it allows the interrupted thread to continue execution independently.

Summarizing, OpenThreads's special calling mechanism optimistically exploits the observation that most handlers run to completion without blocking. In this case, upcalls execute without true thread switches. If the handler does block, then we promote it to an independent thread which OpenThreads schedules just like any other thread.



(a) Interrupted thread and upcall thread before the upcall thread has blocked. The upcall will return to the interrupted thread's stack.

(b) When the upcall thread blocks, OpenThreads modifies the return address.

Fig. 6.10. Fast thread switch to an upcall thread.

6.3 Stream Messages

Panda's stream messages are implemented in the system module. The implementation minimizes copying and allows pipelining of data transfers from application to application process. The key idea is illustrated in Figure 6.11. The data-transfer pipeline consists of four stages, which operate in parallel. In the first stage, Panda copies client data into LFC send packets. In the second stage, LFC transmits send packets to the destination NI. In the third stage, the receiving NI copies receive packets to host memory. Finally, in the fourth stage, the receiving Panda client consumes data from receive packets in host memory. Messages larger than a single packet will benefit from the parallelism in this pipeline. (A stream of short messages also benefits from this pipelining, but the key feature of stream messages is that the same parallelism can be exploited within a single message.)

Stream messages are implemented as follows. At the sending side, the system-module functions *pan_unicast*() and *pan_multicast*() are responsible for message fragmentation. They repeatedly allocate an LFC send packet, store a message header into this packet, and fill the remainder of the packet with data from the user's I/O vector and the header stack.

Unicast header formats are shown in Figure 6.12. The first packet of a message has a different header than the packets that follow. Both headers contain a count of piggybacked credits and a field that identifies the destination SAP. The credits

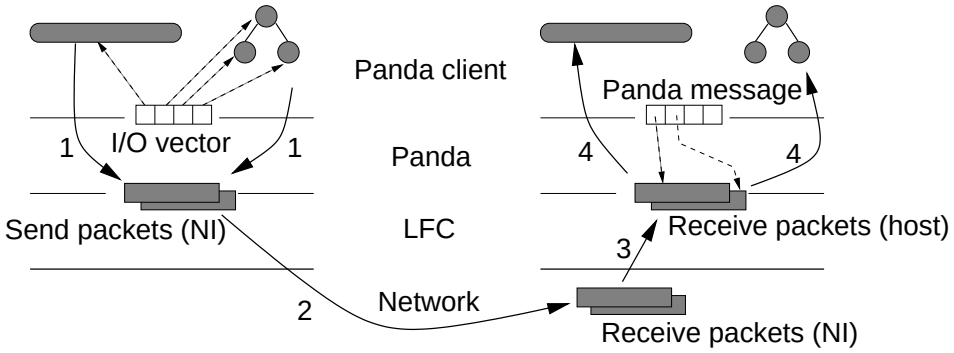


Fig. 6.11. Application to application streaming with Panda's stream messages. (1) Panda copies user data into a send packet. (2) LFC transmits a send packet. (3) LFC copies the packet to host memory. (4) The Panda client consumes data.

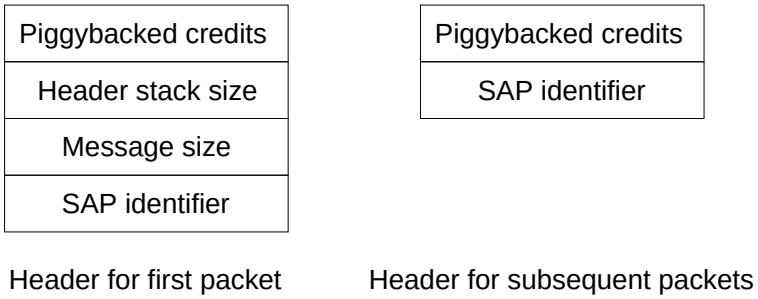


Fig. 6.12. Panda's message-header formats.

field is used by Panda's sliding-window flow-control scheme. The SAP identifier is used for reassembly. Panda does not interleave outgoing packets that belong to different messages. Consequently, senders need not add a message id to outgoing packets. The source identifier in the LFC header (see Figure 3.1) and the SAP identifier in the Panda header suffice to locate the message to which a packet belongs.

The header of a message's first packet contains two extra fields: the size of the header stack contained in the packet and the total message size. The header-stack size is used to find the start of the data that follows the headers. The total message size is used to determine when all of a message's packets have arrived.

Figure 6.13 illustrates the receiver-side data structures. The receiver maintains an array of SAPs. Each SAP contains a pointer to the SAP's handler and a queue of *pending* stream messages. A stream message is considered pending when its first packet has arrived and the receiving process has not yet entirely consumed the stream message. A stream message is represented by a data structure that is created when the stream message's first packet arrives. This data structure con-

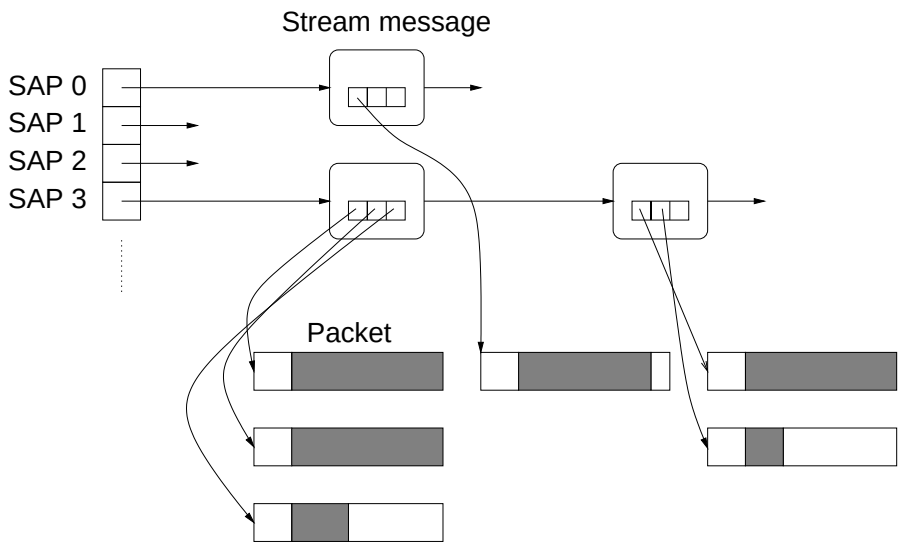


Fig. 6.13. Receiver-side organization of Panda's stream messages.

tains an array which holds pointers to the stream message's constituent packets. Packets are entered into this array as they arrive. In Figure 6.13, one stream message is pending for SAP 0 and two stream messages are pending for SAP 3. The first stream message for SAP 3 consists of three packets. All packets but the last one are full.

When a packet arrives, *lfc_upcall*() searches the destination SAP's queue of pending stream messages. If it does not find the stream message to which this packet belongs, it creates a new stream message and appends it to the queue. (Since LFC delivers all unicast packets in-order, there is no need to store an offset or sequence number in the message headers.) If an incoming packet is not the first packet of its stream message, then *lfc_upcall*() will find the stream message in the SAP's queue of pending messages. The packet is then appended to the stream message's packet array.

When a stream message reaches the head of its SAP's queue, Panda dequeues the stream message and invokes the SAP handler, passing the stream message as an argument. The stream messages queued at a specific SAP are processed one at a time. (That is, a SAP's handler is not invoked again until the previous invocation has returned.)

Pan_msg_consume() copies data from the packets in a message's packet list to user buffers. Each time *pan_msg_consume*() has consumed a complete packet, it returns the packet to LFC. If a Panda client does not want to consume all of a message's data, it can either skip some bytes or discard the entire message without copying any data.

6.4 Totally-Ordered Broadcasting

This section describes an efficient implementation of Panda's totally-ordered broadcast primitive. Totally-ordered broadcast is a powerful communication primitive that can be used to manage shared, replicated data. Panda's totally-ordered broadcast, for example, is used to implement method invocations for replicated shared objects in the Orca system (see Section 7.1).

A broadcast is totally-ordered if all broadcast messages are received in a single order by all receivers and if this order agrees with the order in which senders sent the messages. Most totally-ordered broadcast protocols use a centralized component to implement the ordering constraint and the protocol presented in this section is no exception. The protocol uses a central node, a *sequencer*, to order messages. The protocol we describe uses LFC's fetch-and-add primitive to obtain sequence numbers to order messages.

The protocol is simple. Before sending a broadcast message, the sender just invokes *lfc_fetch_and_add*() to obtain a system-wide unique sequence number. This sequence number is attached to the message and then the message's packets are broadcast using LFC's broadcast primitive. All senders perform their fetch-and-add operations on a single fetch-and-add variable that acts as a shared sequence number. This variable is stored in a single NI's memory.

Receivers assemble the message in the same way they assemble unicast messages (see Section 6.3). The only difference is that a message is not delivered until all preceding messages have been delivered. This enforces the total ordering constraint.

This *G*et-*S*equence-number-then-*B*roadcast (GSB) protocol was first developed for a transputer-based parallel machine [65]. The main difference with the original implementation is that with LFC, requests for a sequence number are handled entirely by the NI. This reduces the latency of such requests in two ways. First, the NI need not copy the request to host memory and the host need not copy a reply message to NI memory. This gain is measurable – an LFC fetch-and-add costs less than an LFC (host-to-host) roundtrip – but small (19.8 μs versus 23.3 μs). The main gain is the reduction in the number of interrupts. Namely, the sequencer does not expect a sequence number request. Therefore, such requests are quite likely to be delivered by means of an interrupt rather than a poll. LFC's fetch-and-add primitive avoids this type of interrupt.

6.5 Performance

In this section we discuss Panda's performance. We measure the latency and throughput for the message-passing and broadcast modules. We do not present

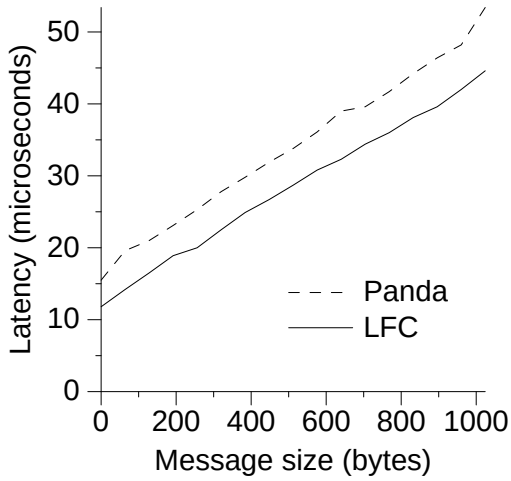


Fig. 6.14. Panda's message-passing latency.

separate results for the RPC module, because RPCs are trivially composed of pairs of message-passing module messages. We compare the Panda measurements with the LFC measurements of Chapter 5, to assess the cost of Panda's higher-level functionality.

In Chapter 5 we measured latency and throughput using three receive methods: no-touch, read-only, and copy. Here we use only the copy method; this is the most common scenario for Panda clients and includes all overhead that Panda's stream-message abstraction adds to LFC's packet interface.

6.5.1 Performance of the Message-Passing Module

Figure 6.14 shows the one-way latency for Panda's message-passing module. For comparison, we also show the one-way latency (with copying at the receiver side) of LFC's unicast primitive. As shown, Panda adds a constant amount (approximately 5 μs) of overhead to LFC's latency. There are several reasons for this increase:

1. Locking. To ensure multithread-safe execution, Panda brackets all calls to LFC routines with lock/unlock pairs.

2. Message abstraction. At the sending side, Panda has to process an I/O vector before sending an LFC packet. At the receiving side, incoming packets have to be appended to a stream message before the receiver upcall can process the data. Panda also maintains several pointers and counters to keep track of the current packet and the current position in that packet.

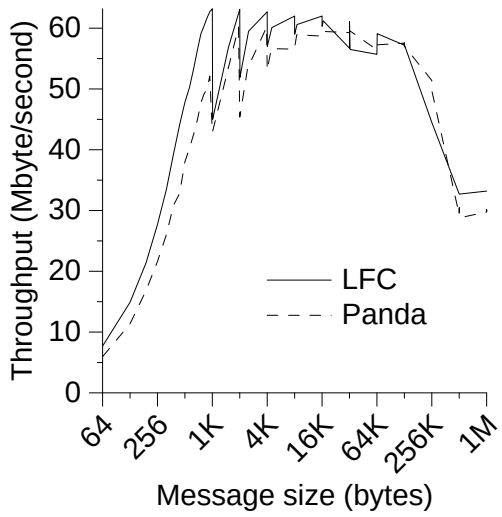


Fig. 6.15. Panda's message-passing throughput.

3. Demultiplexing. Panda adds a header to each LFC packet. The header identifies the destination port and the stream message to which the packet belongs.

4. Flow control. Panda has to check and update its flow-control state for each outgoing and incoming packet.

For 1024-byte messages, the difference between Panda's latency and LFC's latency is larger than for smaller message sizes. For this message size, Panda needs two packets, whereas LFC needs only one; this is due to Panda's header.

Figure 6.15 shows the one-way throughput for Panda's message-passing module and for LFC. Due to the overheads described above, Panda's throughput for small messages is not as good as LFC's throughput. In particular, sending and receiving the first packet of a stream message involves more work than sending and receiving subsequent packets. At the sending side, for example, we need to store the header stack into the first packet; at the receiving side, we have to create a stream message when the first packet arrives. For larger messages, these overheads can be amortized over multiple packets. For this reason, Panda does not reach its peak throughput until a message consists of multiple LFC packets.

For larger messages, Panda sometimes attains higher throughputs than LFC. This is due to cache effects that occur during the copying of data into send packets and out of receive packets.

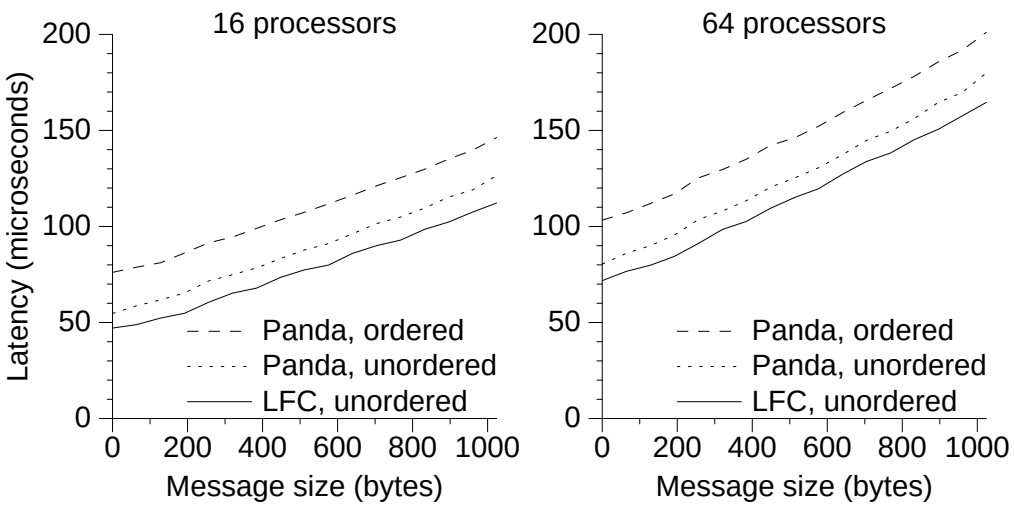


Fig. 6.16. Panda's broadcast latency.

6.5.2 Performance of the Broadcast Module

We measured broadcast performance using three different broadcast primitives:

1. LFC's broadcast using the copy receive strategy (see Section 5.3).

2. Panda's unordered broadcast. Panda provides an option to disable total ordering. When this option is enabled, Panda does not tag broadcast messages with a system-wide unique sequence number. That is, Panda skips the fetch-and-add operation that it normally performs to obtain such a sequence number.

3. Panda's totally-ordered broadcast. This is the broadcast primitive described earlier in this chapter.

Figure 6.16 shows the latency for all three primitives, for message sizes up to 1 Kbyte and for 16 and 64 processors. The difference in latency for a null message between Panda's unordered broadcast and LFC's broadcast is 9 μs. As expected, adding total ordering increases the latency by a constant amount: the cost of a fetch-and-add operation. On 64 processors, the null latency difference between Panda's unordered and ordered broadcasts is 23 μs.

Figure 6.17 shows broadcast throughput for 16 and 64 processors. Adding total ordering reduces the throughput for small and medium-size messages. For larger messages, however, Panda reaches LFC's throughput.

Figure 6.16 and Figure 6.17 show only single-sender broadcast performance. With multiple senders, the performance of totally-ordered broadcasts may suffer

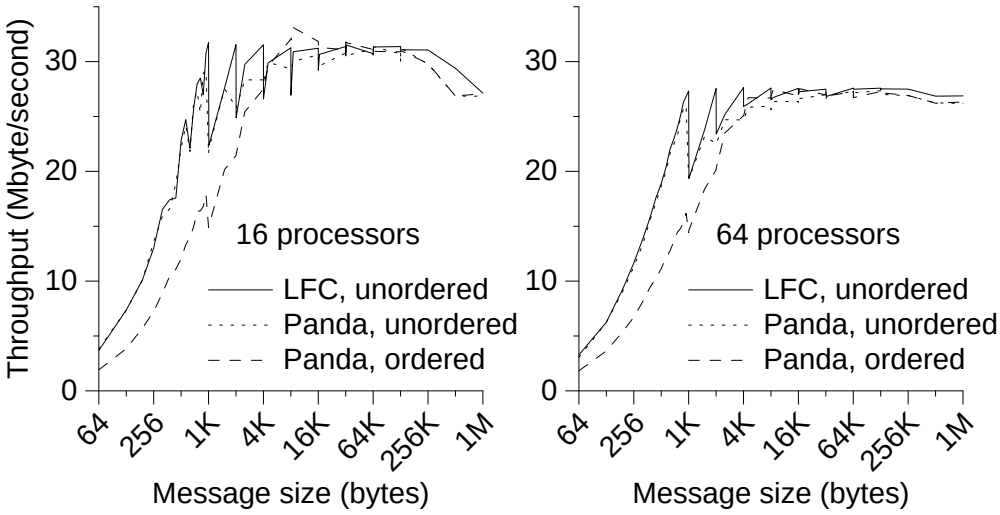


Fig. 6.17. Panda's broadcast throughput.

from the centralized sequencer. In practice, however, this is rarely a problem. In many applications, there is at most one broadcasting process at any time. When processes do broadcast simultaneously, the overall broadcast rate is often low. In an Orca performance study [9], we measured the overhead of obtaining a sequence number from the sequencer. For 7 out of the 8 broadcasting applications considered, the time spent on fetching sequence numbers was less than 0.2% of the application's execution time. Only one application suffered from the centralized sequencer: for this application, a linear equation solver, the time to access the sequencer accounted for 4% of the application's execution time. All applications were run on FM/MC, which uses an NI-level fetch-and-add to obtain sequence numbers, just like LFC.

6.5.3 Other Performance Issues

Some optimizations discussed in this chapter reduce costs that are architecture and operating system dependent. The cost of a thread switch, for example, depends on the amount of CPU state that has to be saved and on the way in which this state is saved. The early Panda implementations ran on SPARC processor architectures on which each (user-level) thread switch requires a trap to the operating system. This trap is needed to flush the register windows, a SPARC-specific cache of the top of the runtime stack, to memory [72, 79]. On other architectures, thread switches are cheaper. On the Pentium Pros used for the experiments in this thesis, a switch from one Panda thread to another costs 1.3 μs. This is still a considerable overhead to add to LFC's one-way null latency, but it is not excessive.

Interrupts and signal processing are other sources of architecture and operating-system dependent overhead. Whereas thread switches are cheap on our experimental platform, interrupts and signals are expensive and should be avoided when possible (see Section 2.5).

6.6 Related Work

We discuss related work in five areas: portable message-passing libraries, polling and interrupts, upcall models and their implementation, stream messages, and totally-ordered broadcasting.

6.6.1 Portable Message-Passing Libraries

PVM [137] and MPI [53] are the most widely used portable message-passing systems. Unlike Panda, these systems target application programmers. The main differences between these systems and Panda are that PVM and MPI are hard to use in a multithreaded environment, use receive downcalls instead of upcalls and do not support totally-ordered broadcast.

PVM and MPI do not provide a multithreading abstraction. Both systems use blocking receive downcalls and do not provide a mechanism to notify an external thread scheduler that a thread has blocked. This is not a problem if the operating system provides kernel-scheduled threads and the client uses these kernel threads. If, on the other hand a client employs a thread package not supported by the operating system, or if the operating system is not aware of threads at all, then the use of blocking downcalls and the lack of a notification mechanism complicate the use of multithreaded systems on top of MPI and PVM.

Multithreaded MPI and PVM clients can avoid the blocking-receive problem by using nonblocking variants of the receive calls, but this means that applications will have to poll for messages, because PVM and MPI do not support asynchronous (i.e., interrupt-driven) message delivery. The same lack of asynchronous message delivery complicates the implementation of PPSs that require asynchronous delivery to operate correctly and efficiently. These PPSs are forced to poll (e.g., in a background thread), but finding the right polling rate is difficult.

LFC does not provide receive downcalls: all packets are delivered through upcalls. Blocking receive downcalls can still be implemented on top of LFC, though. (Panda's message-passing module provides a blocking receive.) When a blocking receive is implemented outside the message-passing system, control can be passed to the thread scheduler when a thread blocks on a receive.

Neither PVM nor MPI provides a totally-ordered broadcast. While a totally-ordered broadcast can be implemented using PVM and MPI primitives, such

an implementation will be slower than an implementation that supports ordered broadcasting at the lowest layers of the system.

The Nexus communication library [54] has the same goals as Panda; Nexus is intended to be used as a compiler target or as part of a larger runtime system. Nexus provides an asynchronous, one-way message-passing primitive called a Remote Service Request (RSR). The destination of an RSR is named by means of a global pointer, which is a system-wide unique name for a communication endpoint. Endpoints can be created dynamically and references to them can be passed around in messages. Unlike Panda, Nexus does not provide a broadcast primitive.

Horus [144] and its precursor Isis [22] were designed to simplify the construction of distributed programs. Horus focuses on message-ordering and fault-tolerance issues. Panda supports parallel rather than distributed programming, provides only one type of ordered broadcast, and does not address fault tolerance. Like Panda, Horus can be configured in different ways. Horus, however, is much more flexible in that it allows protocol stacks to be specified at run time rather than at compile time.

6.6.2 Polling and Interrupts

In their remote-queueing model [28], Brewer et al. focus on the benefits of polling. They recognize, however, that interrupts are indispensable and combine polling with *selective* interrupts. Interrupts are generated only for specific messages (e.g., operating-system messages) or under specific circumstances (e.g., network overflow). In contrast, our integrated thread package chooses between polling and interrupts based on the application's state (idle or not).

CRL [74] is a DSM system that illustrates the need to combine polling and interrupts in a single program. Operations on shared data are bracketed by calls to the CRL library. During or between such operations, a CRL application may not enter the library for a long time, so unless the responsibility for polling is put on the user program, protocol requests can be handled in a timely manner only by using interrupts. CRL therefore uses interrupts to deliver protocol request messages; polling is used to receive protocol reply messages. This type of behavior occurs not only in CRL, but also in other DSMs such as CVM [114] and Orca [9]. It is exactly the kind of behavior that Panda deals with transparently.

Foster et al. describe the problems involved in implementing Nexus's message delivery mechanism (RSRs) on different operating systems and hardware [54]. Available mechanisms for polling and interrupts, and their costs, vary widely across different systems. Moreover, these mechanisms are rarely well integrated with the available multithreading primitives. For example, a blocking read on a socket may block the entire process rather than just the calling thread. We be-

lieve that our system, in which we have full control over thread scheduling and communication, achieves the desired level of integration.

6.6.3 Upcall Models

We have described three message-handling models in which each message results in a handler invocation at the receiving side. Nexus uses two of these models to dispatch RSR handlers. Nexus either creates a new thread to run the handler or else runs the handler in a preallocated thread. The first case corresponds to what we call the popup threads model, the second to the active-messages model. In the second case, the thread is not allowed to block.

A model closely related to popup threads is the optimistic active-messages model [150]. This model removes some of the restrictions of the basic active-messages model. It extends the class of handlers that can be executed as an active-message handler —i.e., without creating a separate thread— with handlers that can safely be aborted and restarted when they block. A stub compiler preprocesses all handler code. When the stub compiler detects that a handler performs a potentially blocking action, it generates code that aborts the handler when it blocks. When aborted, the handler is re-run in the context of a first-class thread (which is created on the fly). Thread-management costs are thus incurred only if a handler blocks; otherwise the handler runs as efficiently as a normal active-message handler.

Optimistic active messages are less powerful than popup threads. First, optimistic active messages require that all potentially blocking actions be recognizable to the stub compiler. Popup threads, in contrast, can be implemented without compiler support and do not rely on programmer annotations to indicate what parts of a handler may block. Second, an optimistic active-messages handler can be re-run safely only if it does not modify any global state before it blocks; otherwise this state will be modified twice. The programmer is thus forced to avoid or undo changes to global state until it is known that the handler cannot block any more.

The stack-splitting and return-address modification techniques we use to invoke message handlers efficiently are similar to the techniques used by Lazy Threads [58]. Lazy Threads provide an efficient fork primitive that optimistically runs a child on its parent's stack. When the child suspends, its return address is modified to reflect the new state. Also, future children will be allocated on different stacks. In our case, a thread that finds a message —either because it polled or because it received a signal— needs to fork a message handler for this message. Unlike Lazy Threads, however, the parent thread (the thread that polled or that was interrupted) does not wait for its child (the message handler) to terminate. Also, we run all children, including the first child, on their own stack.

6.6.4 Stream Messages

Stream messages were introduced in Illinois Fast Messages (version 2.0). Stream messages in Fast Messages differ in several ways from stream messages in Panda. The main difference is that Fast Messages requires that each incoming message be consumed in the context of the message's handler. This implies that the user needs to copy the message if the message arrives at an inconvenient moment. LFC allows a handler to put aside a stream message without copying that message.

This difference results from the way receive-packet management is implemented in Fast Messages. Fast Messages appends incoming packets to a circular queue of packet buffers in host memory. For each packet in this queue, Fast Messages invokes a handler function. When the handler function returns, Fast Messages recycles the buffer. This scheme is simple, because the host never needs to communicate the identities of free buffers to the NI. Both the host and the NI maintain a local index into the queue: the host's index indicates which packet to consume next and the NI's counter indicates where to store the next packet.

LFC explicitly passes the identities of free buffers to the NI (see Section 3.6), which has the advantage that the host can *replace* buffers. This is exactly what happens when *lfc_upcall*() does not allow LFC to recycle a packet immediately. As a result, packets can be put aside without copying, which is convenient and efficient. Clients should be aware, however, that packet buffers reside in pinned memory and are therefore a relatively precious resource. If a client continues to buffer packets and does not return them, then LFC will continue to add new packet buffers to its receive ring and will eventually run out of pinnable memory. To avoid this, clients should keep track of their resource usage and take appropriate measures, either by implementing flow control or by copying packets when the number of buffered packets exceeds a threshold.

6.6.5 Totally-Ordered Broadcast

Totally-ordered broadcast is a well known concept in distributed computing where it is used to simplify the construction of distributed programs. However, totally-ordered broadcast is not used much in parallel programming. As we shall see in Section 7.1, though, the Orca shared object system uses this primitive in an effective way to update replicated objects.

Many different totally-ordered broadcast protocols are described in the literature. Heinzle et al. describe the GSB protocol used by Panda on LFC [65]. The main difference between GSB and Panda's implementation is that Panda handles (through LFC) sequence number requests on the programmable NI; this reduces the number of network interrupts.

Kaashoek describes two other protocols, Point-to-point/Broadcast (PB) and

Broadcast/Broadcast (BB), which also rely on a central, dedicated sequencer node [75]. In PB, the sender sends its message to the sequencer. The sequencer tags the message with a sequence number and broadcasts the tagged message to all destinations. In BB, the sender broadcasts its message to all destinations and the sequencer. When the sequencer receives the message, it assigns a sequence number to the message and then broadcasts a small message that identifies the message and contains the sequence number. Receivers are not allowed to deliver a message until they have received its sequence number and until all preceding messages have been delivered.

The performance characteristics of GSB, PB, and BB depend on the type of network they run on. PB and BB were developed on an Ethernet, where a broadcast has the same cost as a point-to-point message. On switched networks like Myrinet, however, a (spanning-tree) broadcast is much more expensive than a point-to-point message. Consequently, the performance of large messages will be dominated by the cost of broadcasting the data, irrespective of the ordering mechanism.

For small messages, BB's separate sequence-number broadcast is unattractive, because the worst-case latency of a totally-ordered broadcast becomes equal to twice the latency of an unordered broadcast (one for the data and one for the sequence number). Incidentally, BB was also considered unattractive for small messages in an Ethernet setting, but for a different reason: BB generates more interrupts than PB.

PB is much more attractive for small messages, because it adds only a single point-to-point message to the broadcast of the data. GSB uses *two* messages to obtain a sequence number. For large messages, however, PB has the disadvantage that all data packets must travel to and through the sequencer, thus putting more load on the network and on the sequencer. PB also puts more load on the sequencer than GSB if many processors try to broadcast simultaneously. With PB, the occupancy of the sequencer will be high, because the sequencer must broadcast every data packet. Which GSB, the occupancy will be lower, because the sequencer need only respond to fetch-and-add requests. Finally, if all broadcasts originate from the sequencer, there are fewer opportunities to piggyback acknowledgements on multicast traffic.

6.7 Summary

This chapter described the implementation of Panda on LFC. Panda extends LFC with multithreading, messages of arbitrary length, demultiplexing, remote procedure call, and totally-ordered broadcast. The efficient implementation of this functionality is enabled by LFC's performance and interface and by novel tech-

niques employed by Panda. LFC allows packets to be delivered through polling and interrupts. Both mechanisms are useful, but manually switching between them is tedious and error-prone. Panda hides the complexity of managing both mechanisms. Panda clients need not poll, because Panda's threading subsystem, OpenThreads, transparently switches to polling when all client threads are idle. All messages are delivered using asynchronous upcalls. Panda uses the single-threaded upcall model. This model imposes fewer restrictions than the active-messages model, but more than popup threads. Upcalls are dispatched efficiently using thread inlining and lazy thread-creation.

Panda implements stream messages, an efficient message abstraction that enables end-to-end pipelining of data transfers. Panda's stream messages separate notification and the consumption of message data, which allows clients to defer message processing.

Finally, we discussed a simple but efficient implementation of totally-ordered broadcasting. This implementation is enabled by LFC's efficient multicast primitive and its fetch-and-add primitive.

Chapter 7

Parallel-Programming Systems

This chapter focuses on the implementation of parallel-programming systems (PPSs) on LFC and Panda. We will show that the communication mechanisms developed in the previous chapters can be applied effectively to a variety of PPSs. We consider four PPSs:

- Orca [9], an object-based DSM
- Manta, a parallel Java system [99]
- CRL [74], a region-based DSM
- MPICH [61], an implementation of the MPI message-passing standard [53]

These systems offer different parallel-programming models and their implementations stress different parts of the underlying communication systems.

This chapter is organized as follows. Section 7.1 to Section 7.4 describe the programming model, implementation, and performance of, respectively, Orca, Manta, CRL, and MPI. Section 7.5 discusses related work. Section 7.6 compares the different systems and summarizes the chapter.

7.1 Orca

Orca is a PPS based on the shared-object programming model. This section describes this programming model, gives an overview of the Orca implementation, and then zooms in on two important implementation issues: operation transfer and operation execution. Finally, it discusses the performance of Orca.

7.1.1 Programming Model

In shared-memory and page-based DSM systems [78, 82, 92, 155], processes communicate by reading and writing memory words. To synchronize processes, the programmer must use mutual-exclusion primitives designed for shared memory, such as locks and semaphores. Orca's programming model, on the other hand, is based on high-level operations on shared data structures and on implicit synchronization, which is integrated into the model.

Orca programs encapsulate shared data in *objects*, which are manipulated through operations of an *abstract data type*. An object may contain any number of internal variables and arbitrarily complex data structures (e.g., lists and graphs). A key idea in Orca's model is to make each operation on an object *atomic*, without requiring the programmer to use locks. All operations on an object are executed without interfering with each other. Each operation is applied to a single object, but within this object the operation can execute arbitrarily complex code using the object's data. Objects in Orca are passive: objects do not contain threads that wait for messages. Parallel execution is expressed through dynamically created processes.

The shared-object model resembles the use of monitors. Both shared objects and monitors are based on abstract data types and for both models mutual-exclusion synchronization is done by the system instead of the programmer. For condition synchronization, however, Orca uses a higher-level mechanism [12], based on Dijkstra's guarded commands [44]. A *guard* is a boolean expression that must be satisfied before the operation can begin. An operation can have multiple guards, each of which has an associated sequence of statements. If the guards of an operation are all false, the process that invoked the operation is suspended. As soon as one or more guards become true, one true guard is selected nondeterministically and its sequence of statements is executed. This mechanism avoids the use of explicit *wait* and *signal* calls that are used by monitors to suspend and resume processes, simplifying programming.

Figure 7.1 gives an example definition of an object type *Int*. The definition consists of an integer instance variable *x* and two operations, *inc*() and *await*(). Operation *inc*() increments instance variable *x* and operation *await*() blocks until *x* has reached at least the value of parameter *v*. Instances of type *Int* are initialized by the initialization block that sets instance variable *x* to zero.

Figure 7.2 shows how Orca processes are defined and created and how objects are shared between processes. At application-startup time, the Orca runtime system (RTS) creates one instance of process type *OrcaMain*() on processor 0. In this example, *OrcaMain*() creates 15 other processes —instances of process type *Worker*()— on processors 1 to 15. To each of these processes *OrcaMain*() passes a reference to object *counter*. When all processes have been forked, *counter*,

```
object implementation Int;
        x: integer;

        operation inc();
        begin
                x := x + 1;
        end;

        operation await(v : integer);
        begin
                guard x ≥ v do
                od;
        end;

        begin
                x := 0;
        end;
end;
```

Fig. 7.1. Definition of an Orca object type.

```
process Worker(counter: shared Int);
begin
        counter$inc();
        ...
end;

process OrcaMain();
begin
        counter: Int;

        for i in 1..15 do
                fork Worker(counter) on i;
        od;
        counter$await(15);
        ...
end;
```

Fig. 7.2. Orca process creation.

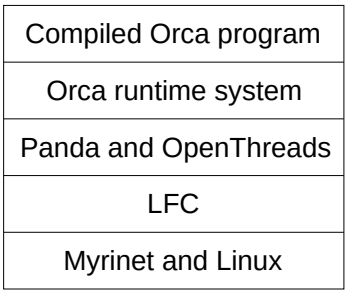


Fig. 7.3. The structure of the Orca shared object system.

which was originally local to *OrcaMain*(), is shared between *OrcaMain*() and all *Worker*() processes. *OrcaMain*() waits until all *Worker*() processes have indicated their presence by incrementing *counter*.

7.1.2 Implementation Overview

Figure 7.3 shows the software components that are used during the execution of an Orca program. The Orca compiler translates the modules that make up an Orca program. For portability, the compiler generates ANSI C. Since the Orca compiler performs many optimizations such as common-subexpression elimination, strength reduction, and code motion, the C code it generates often performs as well as equivalent, hand-coded C programs. The C code is compiled to machine code by a platform-dependent C compiler. The resulting object files are linked with the Orca RTS, Panda, LFC, and other support libraries. Below, we describe how the compiler and the RTS support the efficient execution of Orca programs.

The Compiler

Besides translating sequential Orca code, the compiler supports an efficient implementation of operations on shared objects in three ways. First, the compiler distinguishes between read and write operations. Read operations do not modify the state of the object they operate on. Operation *await*() in Figure 7.1, for example, is a read operation. If the compiler cannot tell if an operation is a read operation, then it marks the operation as a write operation.

Second, the compiler tries to determine the relative frequency of read and write operations [11]. The resulting estimates are passed to the RTS which uses them to determine an appropriate replication strategy for the object. For example, if the compiler estimates that an object is read much more frequently than it is written, then the RTS will replicate the object, because read operations on a replicated object do not require communication. The compiler's estimates are used only as

a first guess. The RTS maintains usage statistics and may later decide to revise a previous decision [87].

Third, the compiler generates operation-specific marshaling code. Marshaling is discussed in Section 7.1.3.

The Runtime System

The Orca RTS implements process creation and termination, shared-object management, and operations on shared objects. Process creation occurs infrequently; most programs create, at initialization time, as many processes as the number of processors. Each Orca process is implemented as a Panda thread. To create a process in response to an application-level **FORK** call, the RTS broadcasts a *fork* message. Upon receiving this message, the processor on which the process is to be created (the forkee) issues an RPC back to the forking processor. This RPC is used to fetch copies of any shared objects that need to be stored on the forkee's processor. When the forkee receives the RPC's reply, it creates a Panda thread that executes the code for the new Orca process. The forker uses a totally-ordered broadcast message rather than a point-to-point message to initiate the fork. Broadcasting forks allows all processors to keep track of the number and type of Orca processes; this information is used during object migration decisions.

The RTS implements two object-management strategies. The simplest strategy is to store a single copy of an object in a single processor's memory. To decide in *which* processor's memory the object must be stored, the RTS maintains operation statistics for each shared object. The RTS uses these statistics and the compiler-generated estimates to migrate each object to the processor that accesses the object most frequently [87].

The second strategy is to replicate an object in the memories of all processors that can access the object. In this case, the main problem is to keep the replicas in a consistent state. Orca guarantees a sequentially consistent [86] view of shared objects. Among others, sequential consistency requires that all processes agree on the order of writes to shared objects. To achieve this, all write operations are executed by means of a totally-ordered broadcast, which leads naturally to a single order for all writes. Fekete et al. give detailed formal description of correct implementation strategies for Orca's memory model [52].

Most programs create only a small number of shared objects. Using the compiler-generated hints and its runtime statistics, the RTS usually decides quickly and correctly on which processor(s) it must store each object [87]. For these reasons, object-management actions have not been optimized: replicating or migrating an object involves a totally-ordered broadcast and an RPC.

Operations are executed in one of three ways: through a local procedure call, through a remote procedure call, or through a totally-ordered broadcast. Figure 7.4

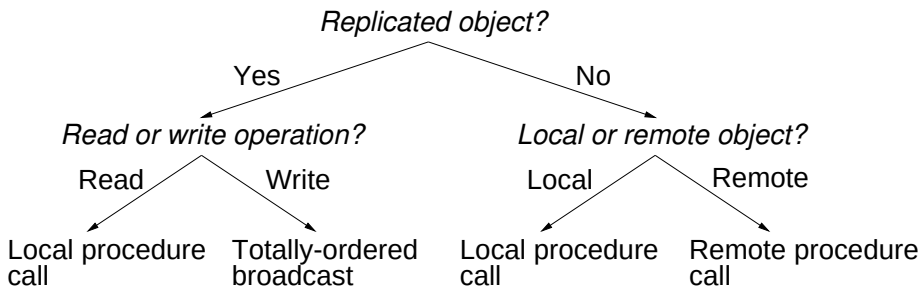


Fig. 7.4. Decision process for executing an Orca operation.

shows how one of these methods is selected. A read operation on a replicated object is executed locally, without communication. This is, of course, the purpose of replication: to avoid communication. Write operations on replicated objects require a totally-ordered broadcast. For operations on nonreplicated objects the RTS does not distinguish between read and write operations, but consider only the object's location. If the object is stored in the memory of the processor that invokes the operation, then the operation is executed by means of a local procedure call; otherwise a remote procedure call is used.

The description so far is correct only for operations that have at most one guard. With multiple guards, the compiler does not know in advance whether the operation is a read or write operation: this depends on the guard alternative that is selected. The compiler could conservatively classify operations that have at least one write alternative as write operations. Instead, however, the compiler classifies individual alternatives as read or write alternatives. When the RTS executes an operation, it first tries to execute a read alternative, because for replicated objects reads are less expensive than writes. The write alternatives are tried only if all read alternatives fail. If the object is replicated, this involves a broadcast.

Operations that require communication are executed by means of *function shipping*. Instead of moving the object to the invoker's processor, the RTS moves the invocation and its parameters to all processors that store the object. This is done using an RPC or a totally-ordered broadcast, as explained above. Since all processors run the same binary program, the code to execute an operation need not be transported from one processor to another; a small operation identifier suffices. In addition to this identifier the RTS marshals the object identity and the operation's parameters. In the case of an operation on a nonreplicated object, this information is stored in the RPC's request message. When the request arrives at the processor that stores the object, this processor's RTS executes the operation and sends a reply that contains the operation's output parameters and return value. In the case of a write operation on a replicated object, the invoker's RTS broadcasts the operation identifier and the operation parameters to all processors that

```
type Person = record
        age: integer;
        weight: integer;
end;

type PersonList = array[integer] of Person;
```

Fig. 7.5. Orca definition of an array of records.

hold a replica. Since the RTS stores each replicated object on all processors that reference it, the invoker's processor always has a copy of the object. All processors that have a replica perform the operation when they receive the broadcast message; other processors discard the message. On the invoker's machine, the RTS additionally passes the operation's output parameters and return value to the (blocked) invoker.

7.1.3 Efficient Operation Transfer

This section describes how the Orca RTS, the Orca compiler, Panda, and LFC cooperate to transfer operation invocations as efficiently as possible.

Compiler and Runtime Support for Marshaling Operations

In earlier implementations of Orca, all marshaling of operations was performed by the RTS. To marshal an operation, the RTS needed the following information:

- the number of parameters
- the parameter modes (**IN**, **OUT**, **IN OUT**, or **SHARED**)
- the parameter types

This information was stored in a recursive data structure in which each component type of a complex type had its own type descriptor node. During operation execution, the RTS made two passes over this data structure: one to find the total size of the data to be marshaled and one to copy the data into a message buffer. For an array of records such as defined in Figure 7.5, for example, the RTS would inspect each record's type descriptor to determine the record's size, even if all records had the same, statically known size.

The current compiler determines object sizes at compile time whenever possible. (Orca supports dynamic arrays and a built-in graph data type, so object sizes cannot always be computed statically.) For each operation defined as part of an

object type, the compiler generates operation-specific marshaling and unmarshaling routines, which are invoked by the Orca RTS. In some cases, these routines call back into the RTS to marshal dynamic data structures and object metastate.

Figure 7.6 shows the code generated to marshal and unmarshal the Orca type defined in Figure 7.5; this code has been edited and slightly simplified for readability. The figure shows only the routines that are used to construct and read an operation request message. Another triple of routines is generated for RPC reply messages that hold **OUT** parameters and return values.

The first routine, *sizeof_iovec_array_record*(), computes the number of pointers in the I/O vector and is used by the RTS to allocate a sufficiently large I/O vector. (The RTS caches I/O vectors, but has to check if there is a cached vector that can hold as many pointers as needed.) The second routine, *fill_iovec_array_record*(), generates a Panda I/O vector that contains pointers to the data items that are to be marshaled. If the array is empty, the routine adds only a pointer to the array descriptor; otherwise it adds a pointer to the array descriptor and a pointer to the array data.

To transmit the operation, the RTS stores its header(s) in a header stack and passes both the I/O vector and the header stack to one of Panda's send routines. Panda copies the data items referenced by the I/O vector into LFC send packets.

At the receiving side, Panda creates a stream message and passes this message to the Orca RTS. The RTS looks up the target object and the descriptor for the operation that is to be invoked, and then invokes *unmarshal_array_record*(), the third routine in Figure 7.6. This operation-specific routine first unmarshals the array descriptor and then decides if it needs to unmarshal any data.

Data Transfers

Above, we discussed the Orca part of operation transfers. This section discusses the entire path, through all communication layers. All data transfers involved in an operation on a remote object are shown in Figure 7.7. At the sending side, Panda copies (using programmed I/O) data referenced by the I/O vector directly from user data structures into LFC send packets. The second stage in the data transfer pipeline consists of moving LFC send packets to the destination NI. In the third stage, LFC uses DMA to move network packets from NI memory to host memory. Panda organizes the packets in host memory into a stream message and passes this message to the Orca RTS. In the fourth stage, the RTS allocates space to hold the data stored in the message and unmarshals (i.e., copies) the contents of the message into this space. All four stages operate concurrently. As soon as Panda has filled a packet, for example, LFC transmits this packet, while Panda starts filling the next packet.

At the sending side, one unnecessary data transfer occurs. Since LFC uses

```
int sizeof_iovec_array_record(PersonListDesc *a)
{
        if (a->a_sz > 0) {
                return 2;               /* pointer to descriptor and to array data */
        }
        return 1;                       /* pointer to descriptor; no data (empty array)*/
}

pan_iovec_p fill_iovec_array_record(pan_iovec_p iov, PersonListDesc *a)
{
        iov->data = a;                  /* add pointer to the array descriptor */
        iov->len = sizeof(*a);
        iov++;

        if (a->a_sz > 0) {              /* add pointer to array data */
                iov->data = (Person *) a->a_data;
                iov->len = a->a_sz * sizeof(Person);
                iov++;
        }

        return iov;
}

void unmarshal_array_record(pan_msg_p msg, PersonListDesc *a)
{
        Person *r;

        pan_msg_consume(msg, a, sizeof(*a)); /* unmarshal array descriptor */
        if (a->a_sz > 0) {
                r = malloc(a->a_sz * sizeof(Person));
                a->a_data = r;
                pan_msg_consume(msg, r, a->a_sz * sizeof(Person));
        } else {
                a->a_sz = 0;
                a->a_data = 0;
        }
}
```

Fig. 7.6. Specialized marshaling code generated by the Orca compiler.

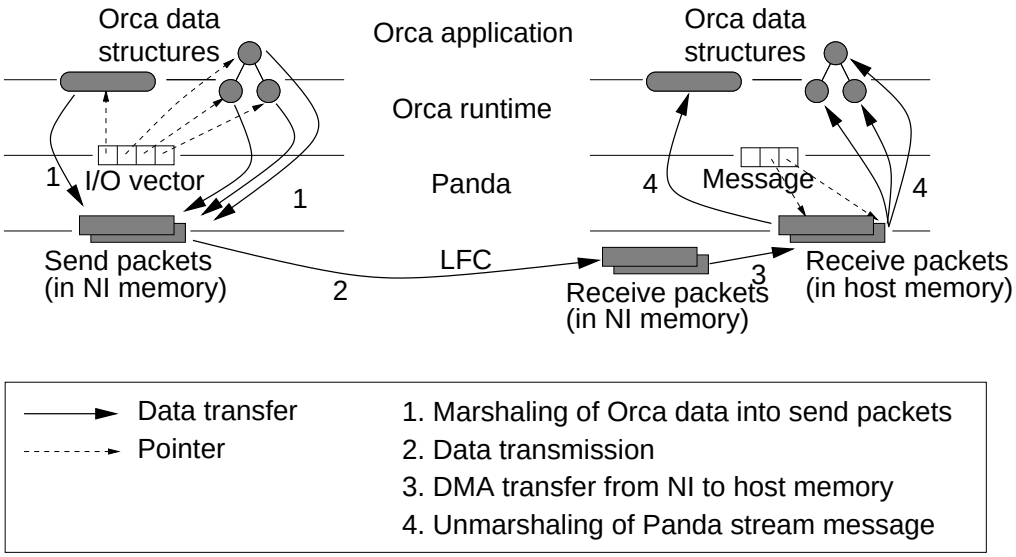


Fig. 7.7. Data transfer paths in Orca.

programmed I/O to move data into send packets, the data crosses the memory bus twice: once from host memory to the processor registers and from those registers to NI memory. By using DMA instead of PIO, the data would cross the memory bus only once. The data in Figure 2.2 suggests that for messages of 1 Kbyte and larger DMA will be faster than PIO. The use of DMA also frees the processor to do other work while the transfer progresses. The same asynchrony, however, also implies that a mechanism is needed to test if a transfer has completed. This requires communication between the host and the NI. Either the host must poll a location in NI memory, which slows down the data transfer, or the NI must signal the end of the data transfer by means of a small DMA transfer to host memory, which increases the occupancy of both the NI processor and the DMA engine.

Another problem with DMA is that it is necessary to pin the pages containing the Orca data structures or to set up a dedicated DMA area. Unless the Orca data structures can be stored in the DMA area, the latter approach requires a copy to the DMA area. On-demand pinning of Orca data is feasible, but complex. To avoid pinning and unpinning on every transfer, a caching scheme such as VMMC-2's UTLB is needed (see Section 2.3). For small transfers, such a scheme is less efficient than programmed I/O.

At the receiving side, the optimal scenario would be for the NI to transfer data from the network packets in NI memory to the data buffers allocated by the RTS (i.e., to avoid the use of host receive packets). In practice, this is difficult. First, the NI has to know to which message each incoming data packet belongs. This implies that the NI has to maintain state for each message and look up this state for

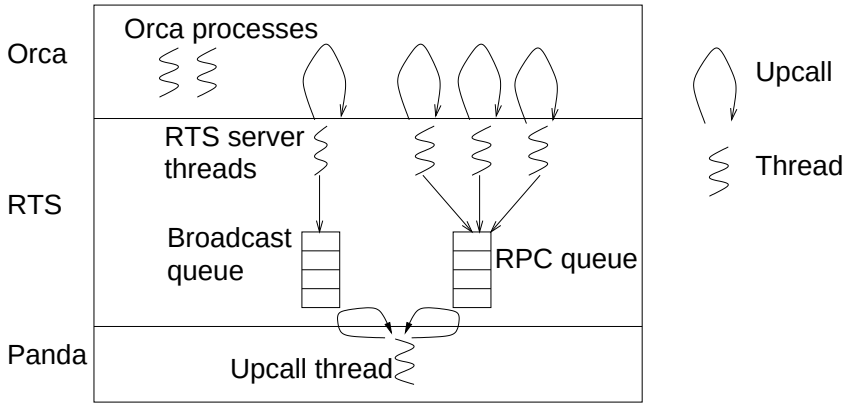


Fig. 7.8. Orca with popup threads.

each incoming packet. Second, the NI has to find a sufficiently large destination buffer in host memory into which the data can be copied. Third, after the data has been copied, the NI has to notify the receiving process. Unless the host and the NI perform a handshake to agree upon the destination buffer, this notification cannot be merged with the data transfer as in LFC. A handshake, however, increases latency.

Summarizing, eliminating all copies is difficult and it is doubtful whether doing so will significantly improve performance. We therefore decided to use a potentially slower, but simpler data path.

7.1.4 Efficient Implementation of Guarded Operations

This section describes an efficient implementation of Orca's guarded operations on top of Panda's upcall model. When Panda receives an operation request, it dispatches an upcall to the Orca RTS. It is not obvious how the RTS should execute the requested operation in the context of this upcall. The problem is that the operation may block on a guard. Panda's upcall model, however, forbids message handlers to block and wait for incoming messages. Such messages may have to be received and processed to make a guard evaluate to true.

To solve this problem, an earlier version of the Orca RTS implemented its own popup threads. The structure of this RTS, which we call RTS-threads, is shown in Figure 7.8. In RTS-threads, Panda's upcall thread does not execute incoming operation requests, but merely stores them in one of two queues. RPC requests for operations on nonreplicated objects are stored in the RPC queue and broadcast messages for write operations on replicated objects are stored in the broadcast queue. These queues are emptied by a pool of RTS-level server threads. When all server threads are blocked, the RTS extends the thread pool with new server

threads. Since the server threads do not listen to the network, they can safely block on a guard when they process an incoming operation request from the RPC queue. (This blocking is implemented by means of condition variables.) Also, it is now safe to perform synchronous communication while processing a broadcast message. This type of nested communication occurs during the creation of Orca processes and during object migration.

RTS-threads uses only one thread to service the broadcast queue. With multiple threads, incoming (totally-ordered) broadcast messages could be processed in a different order than they were received. Using only a single thread, however, implies (once again) that, without extra measures, write operations on replicated objects cannot block.

Besides this ordering and blocking problem, RTS-threads suffers from problems related to the use of popup threads (see Section 6.2.3): increased thread switching and increased memory consumption. Processing an incoming operation request always requires at least one thread switch (from Panda's upcall to a server thread). In addition, when many incoming operation requests block, RTS-threads is forced to allocate many thread stacks, which wastes memory. Finally, when some operation modifies an object that many operations are blocked on, RTS-threads will signal all threads associated with these operations. These threads will then re-evaluate the guard they were blocked on, perhaps only to find that they need to block again. This leads to excessive thread switching.

To solve these problems, we restructured the RTS so that all operation requests can be processed directly by Panda's upcall, without the use of server threads. The new structure is shown in Figure 7.9. The new RTS employs only one RTS server thread (not shown), which is used only during actions that occur infrequently, such as process creation and object migration. In these cases, it is necessary to perform an RPC (i.e., synchronous communication) in response to an incoming broadcast message. This RPC is performed by the RTS server thread, just as in RTS-threads.

Blocking on guards is handled by exploiting the observation that Orca operations can block only at the very beginning. A blocked operation can therefore always be represented by an object identifier, an operation identifier, and the operation's parameters. This *invocation information* is available to the RTS when it invokes an operation. When an operation blocks, the compiler-generated code for that operation returns an error status to the RTS. Instead of blocking the calling thread on a condition variable, as in RTS-threads, the RTS creates a continuation. In this continuation, the RTS stores the invocation information and the name of an RTS function that, given the invocation information, can resume the blocked invocation. Different resume functions are used, depending on the way the original invocation took place. The resume function for an RPC invocation, for example, differs from the resume function for a broadcast invocation, because it needs to send a reply message to the invoking processor.

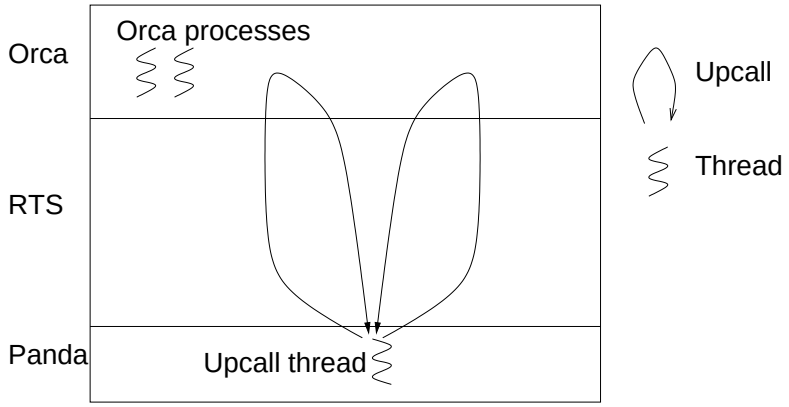


Fig. 7.9. Orca with single-threaded upcalls.

```
cont_init(contqueue, lock)
cont_clear(contqueue)
state_ptr = cont_alloc(contqueue, size, contfunc)
cont_save(state_ptr)
cont_resume(contqueue)
```

Fig. 7.10. The continuations interface.

Each shared object has a *continuation queue* in which the RTS stores continuations for blocked operations. When the object is later modified, the modifying thread walks the object's continuation queue and calls each continuation's resume function. If an operation fails again, its continuation remains on the queue, otherwise it is destroyed.

Figure 7.10 shows the interface to the continuation mechanism. Queues of continuations resemble condition variables, which are essentially queues of thread descriptors. This similarity makes replacing condition variables with continuations quite easy. *Cont_init*() initializes a continuation queue; initially, the queue is empty. Like a condition variable, each continuation queue has an associated lock (*lock*) that ensures that accesses to the queue are atomic. *Cont_clear*() destroys a continuation queue. *Cont_alloc*() heap-allocates a continuation structure and associates it with a continuation queue. (Continuations cannot be allocated on the stack, because the stack is reused.) *Cont_alloc*() returns a pointer to a buffer of *size* bytes in which the client saves its state. After saving its state, the client calls *cont_save*() which appends the continuation to the queue. Together, *cont_alloc*() and *cont_save*() correspond to a wait operation on a condition variable. In the case of condition variables, however, no separate allocation call is needed, because the system knows what state to save: the client's thread stack. Finally, *cont_resume*()

resembles a broadcast on a condition variable; it traverses the queue and resumes all continuations.

Representing blocked operations by continuations instead of blocked threads has three advantages.

1. Continuations consume very little memory. There is no associated stack, just the invocation information.

2. Resuming a blocked invocation does not involve any signaling and switching to other threads, but only plain procedure calls.

3. Continuations are portable across communication architectures. With appropriate system support (such as the fast handler dispatch described in Section 6.2.3), popup threads can be implemented more efficiently than in RTS-threads. Continuations do not require such support *and* they avoid some of the disadvantages of popup threads.

Manual state saving with continuations is relatively easy if message handlers can block only at the beginning (as with Orca operations), because the state that needs to be saved consists essentially of the message that has been received. Manual state saving is more difficult when handlers block after they have created new state. In this case, the handler must be split in two parts: the code executed before and after the synchronization point. For very large systems, this manual function splitting becomes tedious. The difficulty lies in identifying the state that needs to be communicated from the function's first half to its second half (by means of a continuation). This state may well include state stored in stack frames below the frame that is about to block (assuming that stacks grow upward). In the worst case, the entire stack must be saved.

A second complication arises if a handler performs synchronous communication. A synchronous communication call can only be split in two parts if there is a nonblocking alternative for the synchronous primitive. Earlier versions of Panda, for example, did not provide asynchronous message passing primitives, but only a synchronous RPC primitive. In this case, true blocking cannot be avoided and it is necessary to hand off state to a separate thread that can safely perform the synchronous operation. In the continuation-based RTS, such cases are handled by a single RTS server thread.

7.1.5 Performance

This section discusses only operation latency and throughput. An elaborate application-performance study was performed using an Orca implementation that runs on Panda (version 3.0) and FM/MC [9]. Application performance on Panda 4.0

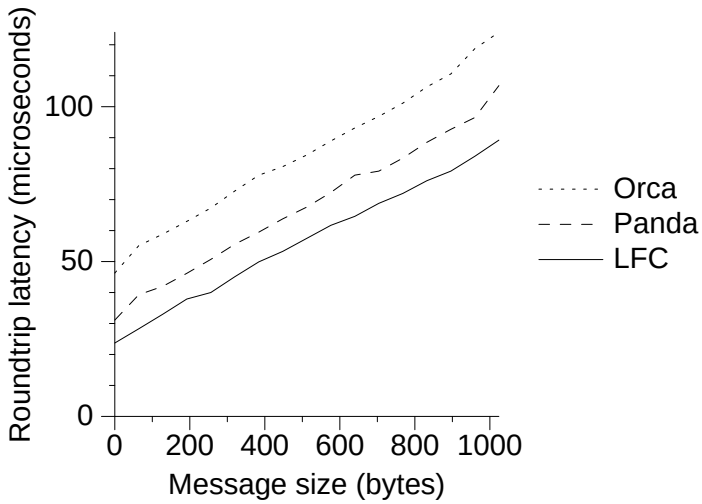


Fig. 7.11. Roundtrip latencies for Orca, Panda, and LFC.

and LFC is discussed in Chapter 8 which studies the application-level effects of different LFC implementations. All measurements in this chapter were performed on the experimental platform described in Section 1.6, unless noted otherwise.

Performance of Operations on Nonreplicated Objects

Figure 7.11 shows the execution time of an Orca operation on a remote, non-replicated object. The operation takes an array of bytes as its only argument and returns the same array. The horizontal axis specifies the number of bytes in the array. The operation is performed using a Panda RPC to the remote machine. For comparison, Figure 7.11 also shows roundtrip latencies for Panda and LFC using messages that have the same size as the Orca array. In all three benchmarks, the request and reply messages are received through polling.

Orca-specific processing costs approximately 15 μs. This includes the time to lock the RTS, to read the RTS headers, to allocate space for the array parameter, to look up the object, to execute the operation, to update the runtime object access statistics, and to build the reply message.

We used the same test to measure operation throughput. The results of this test, and the corresponding roundtrip throughputs for Panda and LFC with the copy receive strategy, are shown in Figure 7.12. We define roundtrip throughput as $(2m)/RTT$, where m is the size in bytes of the array transmitted in the request and reply message and RTT the measured roundtrip time.

Compared to the throughput obtained by Panda and LFC, Orca's throughput is good. This is due to the use of I/O vectors and stream messages, which avoid

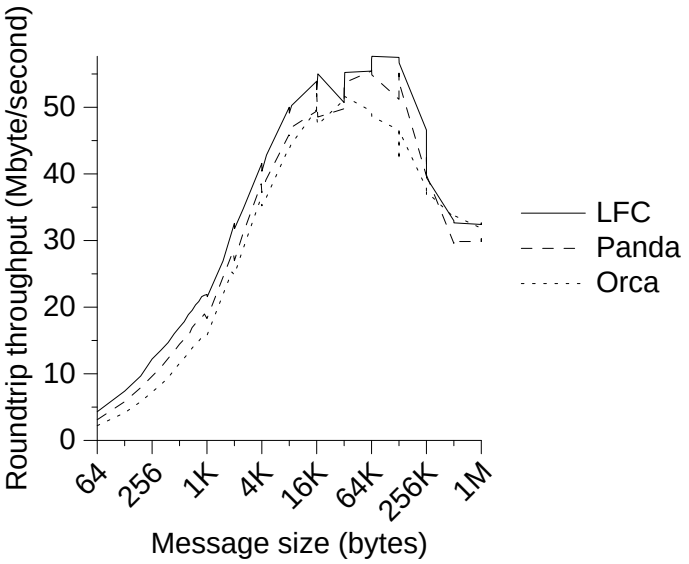


Fig. 7.12. Roundtrip throughput for Orca, Panda, and LFC.

extra copying inside the Orca RTS. For all systems, throughput decreases when messages no longer fit into the L2 cache (256 Kbyte).

Performance of Operations on Replicated Objects

Figure 7.13 shows the latency of a write operation on an object that has been replicated on 64 processors. The operation takes an array of bytes as its only argument and does not return any results. This operation is performed by means of a totally-ordered Panda broadcast. For comparison, the figure also shows the broadcast latencies for Panda (ordered and unordered) and LFC (unordered). In all cases, the latency shown is the latency between the sender and the last receiver of the broadcast. Orca adds approximately 18 μs (17%) to Panda's totally-ordered broadcast primitive.

Figure 7.14 shows the throughput obtained using the same write operation as above. The loss in throughput relative to Panda (with ordering) is at most 15%. This occurs when Orca sends two packets and Panda one (due to an extra Orca header). In all other cases the loss is at most 8%.

The Impact of Continuations

Since we have no implementation of RTS-threads on Panda 4.0, we cannot directly measure the gains of using continuations instead of popup threads. An earlier comparison showed that operation latencies went from 2.19 ms in RTS-threads

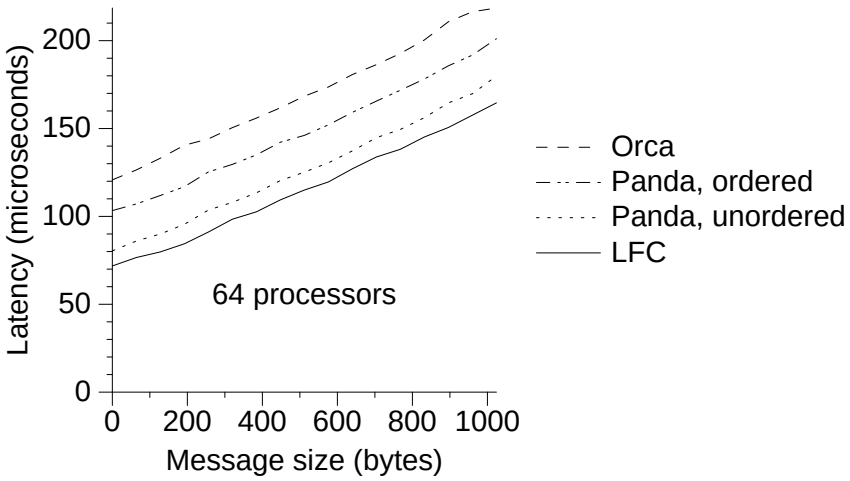


Fig. 7.13. Broadcast latency for Orca, Panda, and LFC.

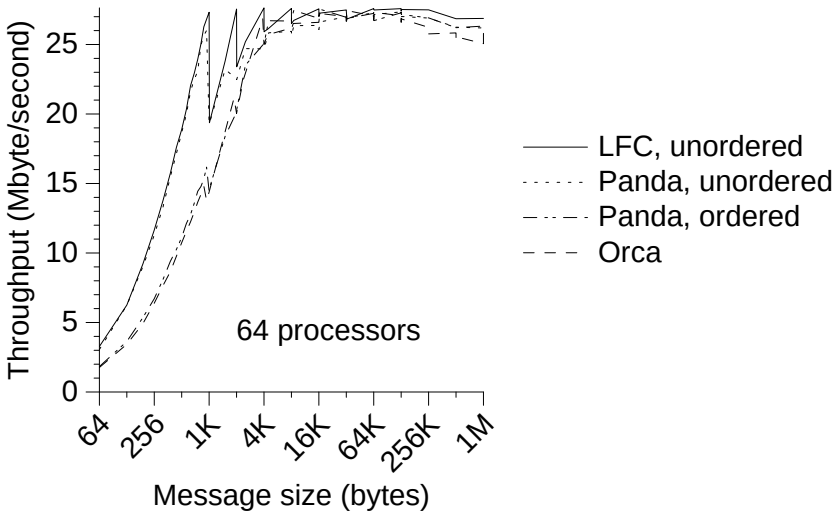


Fig. 7.14. Broadcast throughput for Orca, Panda, and LFC.

	Issue	Orca	Manta
Programming model	Object placement	Transparent	Explicit
	Mutual exclusion	Per operation, implicit	Synchronized methods/blocks
	Condition synchronization	Guards	Condition variables
	Garbage collection	No	Yes
Implementation	Compilation	To C	To assembly (x86 and SPARC)
	Object replication	Yes	No
	Object migration	Yes	No
	Invocation mechanisms	Totally-ordered bcast and RPC	RMI
	Upcall models	Panda upcalls	Popup threads

Table 7.1. A comparison of Orca and Manta.

to 1.90 ms in the continuation-based RTS [14], an improvement of 13%. These measurements were performed on 50 MHz SPARCClassic clones connected by 10 Mbps Ethernet. For communication we used Panda 2.0 on top of the datagram service of the Amoeba operating system. Thread switching on the SPARC was expensive, because it required a trap to the kernel. On the Pentium Pro, thread switches are performed entirely in user space, so the gains of using continuations instead of popup threads are smaller. Nevertheless, the costs of switching from a Panda upcall to a popup thread are significant. In Manta, for example, dispatching a popup thread costs 4 μs and accounts for 9% of the execution time of a null operation on a remote object (see Section 7.2).

7.2 Manta

Manta is a parallel-programming system based on Java [60]. Java is a portable, type-secure, and object-oriented programming language with automatic memory management (i.e., garbage collection); these properties have made Java a very popular programming language. Java is also an attractive (base) language for parallel programming. Multithreading, for example, is part of the language and an RPC-like communication mechanism is available in standard libraries.

Manta implements the JavaParty [117] programming model, which is based on shared objects and resembles Orca's shared-object model. The programming model and its implementation on Panda and LFC are described below. A detailed description of Manta is given in the MSc thesis of Maassen and van Nieuwpoort [98].

7.2.1 Programming Model

Manta extends Java with one keyword: **remote**. Java threads can share instances of any class that the programmer tags with this keyword. Such instances are called *remote objects*. In addition, Manta provides library routines to create threads on remote processors. Threads that run on the same processor can communicate through global variables; threads that run on different machines can communicate only by invoking methods on shared remote objects.

This programming model resembles Orca's shared-object model, but there are several important differences, which are summarized in Table 7.1 and discussed below. Section 7.2.2 discusses the implementation differences.

First, in Manta, object placement is not transparent to the programmer. When creating a remote object, the programmer must specify explicitly on which processor the object is to be stored. Each object is stored on a single processor —Manta does not replicate objects— and cannot be migrated to another processor.

Second, Orca and Java use different synchronization mechanisms. In Orca, all operations on an object are executed atomically. In Java, individual methods can be marked as 'synchronized.' Two synchronized methods cannot interfere with each other, but a nonsynchronized method can interfere with other (synchronized and nonsynchronized) methods. Also, Java uses condition variables for condition synchronization; Orca uses guards.

The last difference is that Java is a garbage-collected language, which has some impact on communication performance (see Section 7.2.3).

7.2.2 Implementation

Like Orca, Manta does not use LFC directly, but builds on Panda. Manta benefits from Panda's multithreading support, Panda's transparent mixing of polling and interrupts, and Panda's stream messages. Since Manta does not replicate objects, it need not maintain multiple consistent copies of an object, and therefore does not use Panda's totally-ordered broadcast primitive.

Manta achieves high performance through a fast remote-object invocation mechanism and the use of compilation instead of interpretation. Both are discussed below.

Remote-Object Invocation

Like Orca, Manta uses function shipping to access remote objects. Method invocations on a remote object are shipped by means of an object-oriented variant of RPC called Remote Method Invocation (RMI) [152]. Each Manta RMI is executed by means of a Panda RPC.

Processing an RMI request consists of executing a method on a remote object. All parameters except remote objects are passed by value. Method invocations can block at any time on a condition variable. This is different from Orca where blocking occurs only at the beginning of an operation. Due to this difference, Manta cannot always use Panda upcalls and continuations to handle incoming RMIs. If a method is executed by a Panda upcall and blocks halfway through, then the Manta runtime system is not aware of the state created by that method and cannot create a continuation.

To deal with blocking methods, Manta uses popup threads to process incoming RMIs. When an RMI request is delivered by a Panda upcall, the Manta runtime system passes the request to a popup thread in a thread pool. This thread executes the method and sends the reply. When the method blocks, the popup thread is blocked on a Panda condition variable; no continuations are created. In short, Manta uses the same system structure as earlier Orca implementations (cf. Figure 7.8).

Compiler Support

Besides defining the Java language, the Java developers also defined the Java Virtual Machine (JVM) [95], a virtual instruction set architecture. Java programs are typically compiled to JVM bytecode; this bytecode is interpreted. This approach allows Java programs to be run on any machine with a Java bytecode interpreter. The disadvantage is that interpretation is slow. Current research projects are attacking this problem by means of just-in-time (JIT) compilation of Java bytecode for a specific architecture and by means of hardware support [104].

Manta includes a compiler that translates Java source code to machine code for SPARC and Intel x86 architectures [145]. This removes the overhead of interpretation for applications for which source code is available. For interoperability, Manta can also dynamically load and execute Java bytecode [99]. The experiments below, however, use only native binaries without bytecode.

The compiler also supports Manta's RMI. For each remote class, the compiler generates method-specific marshaling and unmarshaling routines. Manta marshals arrays in the same way as Orca (see Figure 7.6). At the sending side, the compiler-generated code adds a pointer to the array data to a Panda I/O vector and Panda copies the data into LFC send packets. At the receiving side, the compiler-generated code uses Panda's *pan_msg_consume*() to copy data from LFC host receive packets to a Java array.

For nonarray types, Manta makes an extra copy at the sending side. All nonarray data that is to be marshaled is copied to a single buffer and a pointer to this buffer is added to a Panda I/O vector. Since nonarray objects are often small, the extra copying costs may outweigh the cost of I/O vector manipulations.

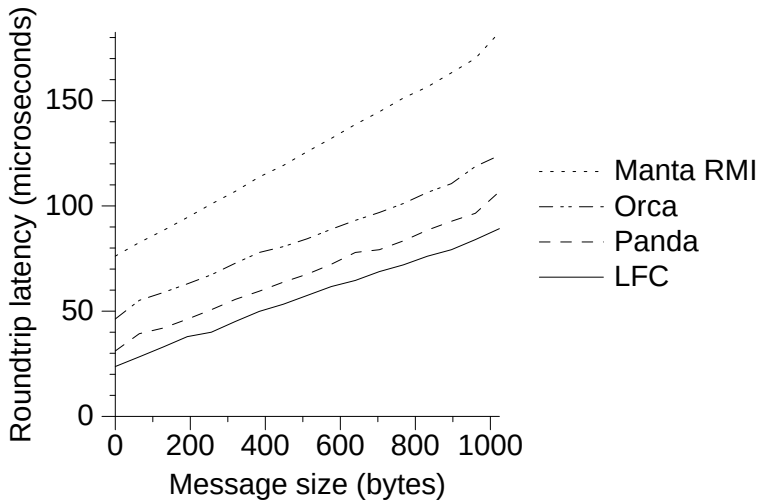


Fig. 7.15. Roundtrip latencies for Manta, Orca, Panda, and LFC.

7.2.3 Performance

Figures 7.15 and 7.16 show Manta's RMI (roundtrip) latency and throughput, respectively. These numbers were measured by repeatedly performing an RMI on a remote object. The method takes a single array parameter and returns this parameter; the horizontal axes of Figure 7.15 and Figure 7.16 specify the size of the array. For comparison, the figure shows also the roundtrip latencies and throughputs for LFC, Panda, and Orca. All four systems make the same number of data copies.

The null latency for an empty array is 76 μs. This high latency is due to unoptimized array manipulations. With zero parameters instead of an empty array parameter, the null latency drops to 48 μs. The thread switch from Panda's upcall to a popup thread in the Manta runtime system costs 4 μs. For nonempty arrays, Manta's latency increases faster than the latency of Orca, Panda and LFC. This is not due to extra copying, but to the cache effect described below.

Figure 7.16 shows the roundtrip throughput for LFC, Panda, Orca, and Manta. Manta's RMI peak throughput is much lower than the peak throughput for Orca, Panda, and LFC. The reason is that Manta does not free the data structures into which incoming messages are unmarshaled until the garbage collector is invoked. In this test, the garbage collector is never invoked, so each incoming message is unmarshaled into a fresh memory area, which leads to poor cache behavior. Specifically, each time a message is unmarshaled into a buffer, the contents of receive buffers used in previous iterations is flushed from the cache to memory. This problem does not occur in the other tests (Orca, Panda, and LFC), because they reuse the same receive buffer in each iteration.

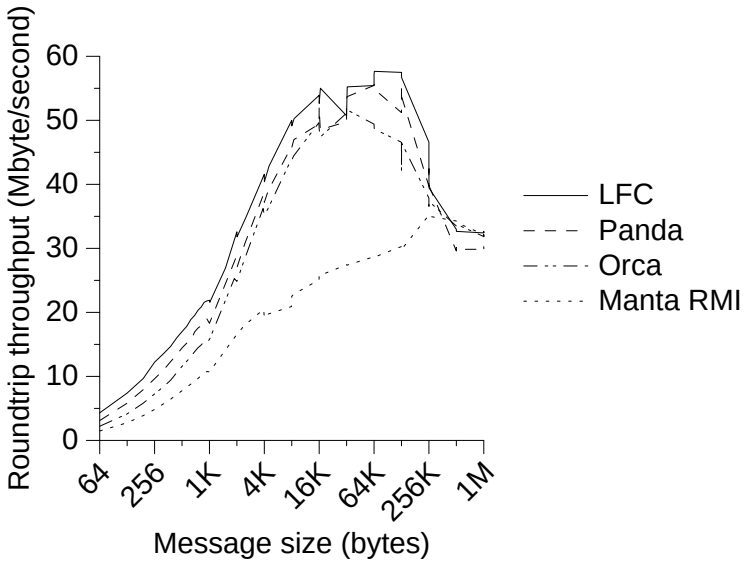


Fig. 7.16. Roundtrip throughput for Manta, Orca, Panda, and LFC.

If Manta can see that the data objects contained in a request message are no longer reachable after completion of the remote method that is invoked, then it can immediately reuse the message buffer without waiting for the garbage collector to recycle it. At present, the Manta compiler is able to perform the required *escape analysis* [36] for simple methods, including the method used in our throughput benchmark. Since we wish to illustrate the garbage collection problem and since escape analysis works only for simple methods, we show the unoptimized throughput. With escape analysis enabled, Manta tracks Panda's throughput curve.

7.3 The C Region Library

The C Region Library (CRL) is a high-performance software DSM system which was developed at MIT [74]. Using CRL, processes can share regions of memory of a user-defined size. The CRL runtime system implements coherent caching for these regions. This section discusses CRL's programming model, its implementation on LFC, and the performance of this implementation.

7.3.1 Programming Model

CRL requires the programmer to store shared data in *regions*. A region is a contiguous chunk of memory of a user-defined size; a region can neither grow nor

shrink. Processes can map regions into their address space and access them using memory-reference instructions.

To make changes visible to other processes, and to observe changes made by other processes, each process must enclose its region accesses by calls to the CRL runtime system. (CRL is library-based and has no compiler.) A series of read accesses to a region *r* should be bracketed by calls to *rgn_start_read(r)* and *rgn_end_read(r)*, respectively. If a series of accesses includes an instruction that modifies a region *r*, then *rgn_start_write(r)* and *rgn_end_write(r)* should be used. *Rgn_start_read*() locks a region for reading and *rgn_start_write*() locks a region for writing. Both calls block if another process has already locked the region in a conflicting way. (A conflict occurs whenever at least on write access is involved.) *Rgn_end_read*() and *rgn_end_write*() release the read and write lock, respectively. If all region accesses in a program are properly bracketed, then CRL guarantees a sequentially consistent [86] view of all regions. However, the CRL library cannot verify that users indeed bracket their accesses properly.

7.3.2 Implementation

Although CRL uses a sharing model that is similar to Orca's shared objects, the implementation is different. First, CRL runs directly on LFC and does not use Panda. Second, CRL uses a different consistency protocol for replicated shared objects. Orca *updates* shared objects by means of *function shipping*. CRL, in contrast, uses *invalidation* and *data shipping*. (A function-shipping implementation of CRL exists [67], but the LFC implementation of CRL is based on the original, data-shipping variant.) CRL uses a protocol similar to the cache coherence protocols used in scalable multiprocessors such as the Origin2000 [151] and the MIT Alewife [1]. The protocol treats every shared region as a cache line and runs a directory-based cache coherence protocol to maintain cache consistency.

The core of CRL's runtime system is a state machine that implements the consistency protocol. The runtime system maintains (meta)state for each region on all nodes that cache the region. The state machine is implemented as a set of handler routines that are invoked when a specific event occurs. An event is either the invocation of a CRL library routine that brackets a region access (e.g., *rgn_start_write*()) or the arrival of a protocol message. A detailed description of the state machine is given in Johnson's PhD thesis [73].

When a process creates a region on a processor *P*, then *P* becomes the region's *home node*. The home node stores the region's directory information; it maintains a list of all processors that are caching the region. It is the location where nodes that are not caching the region can obtain a valid copy of the region. It is also a central point at which conflicting region accesses are serialized so that consistency is maintained.

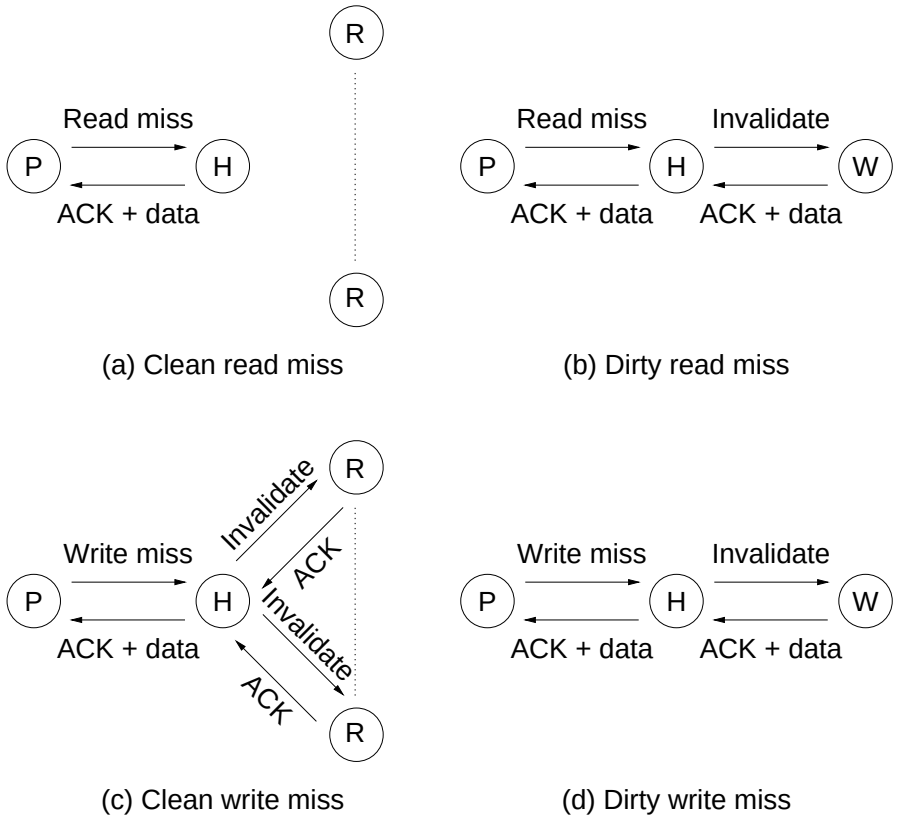


Fig. 7.17. CRL read and write miss transactions.

CRL Transactions

Communication in CRL is caused mainly by read and write misses. Figure 7.17 shows the communication patterns that result from different types of misses. A miss is a *clean miss* if the home node H has an up-to-date copy of the region that process P is trying to access, otherwise the miss is a *dirty miss*. In the case of a clean miss, the data travels across the network only once (from H to P). For a clean write miss, the home node sends invalidations to all nodes R that recently had read access to the region. The region is not shipped to P until all readers R have acknowledged these invalidations. In the case of a dirty miss the data is transferred twice, once from the last writer W to home node H and once from H to the requesting process P. The CRL version we used does not implement the three-party optimization that avoids this double data transfer. (Later versions did implement this optimization).

Polling and Interrupts

CRL uses both polling and interrupts to deliver messages. The implementation is single-threaded. All messages, whether they are received through polling or interrupts, are handled in the context of the application thread.

The runtime system normally enables network interrupts so that nodes can be interrupted to process incoming requests (misses, invalidations, and region-map requests) even if they happen to be executing application code. For some applications, interrupts are necessary to guarantee forward progress. For other applications, they improve performance by reducing response time.

In many cases, though, interrupts can be avoided. All transactions in Figure 7.17 start with a request from node P to home node H and end with a reply from H to P. While waiting for the reply from H, P is idle. Similarly, if H sends out invalidations then H will be idle until the first acknowledgement arrives and in between subsequent acknowledgements. In these idle periods, CRL disables network interrupts and polls. This way the expected reply message is received through polling rather than interrupts, which reduces the receive overhead.

This behavior is a good match to LFC's polling watchdog which works well for clients that mix polling and interrupts. Interrupts are used to handle request messages *unless* the destination node happens to be waiting (and polling) for a reply to one of its own requests. By delaying interrupts, LFC increases the probability that this happens.

Message Handling

CRL sends two types of messages. *Data messages* are used to transfer regions containing user data (e.g., when a region is replicated). These messages consist of a small header followed by the region data. *Control messages* are small messages used to request copies of regions, to invalidate regions, to acknowledge requests, etc. Both data and control messages are point-to-point messages; CRL does not make significant use of LFC's multicast primitive.

Each CRL message carries a small header. Each header contains a demultiplexing key that distinguishes between control messages, data messages, and other (less frequently used) message types. For control messages, the header contains the address of a handler function and up to four integer-sized arguments for the handler. A data message consists of a region pointer, a region offset, and a few more fields. The region pointer identifies a mapped region in the requesting process's address space. Since regions can have any size, they may well be larger than an LFC packet. The offset is used to reassemble large data messages; it indicates where in the region to store the data that follows the header.

Incoming packets are always processed immediately. The data contained in

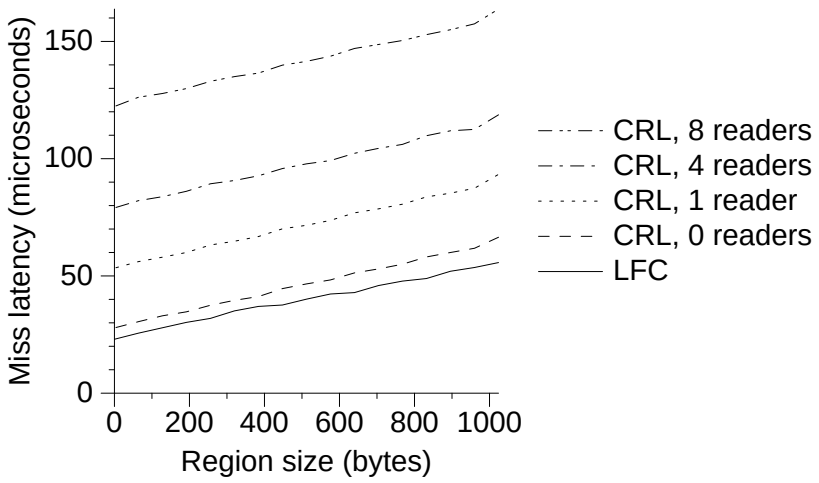


Fig. 7.18. Write miss latency.

data packets is always copied to its destination without delay. Protocol-message handlers, in contrast, cannot always be invoked immediately; in that case the protocol message is copied and queued. (This copy is unnecessary, but protocol messages consist of only a few words.) Since no process can initiate more than one coherence transaction, the amount of buffering required to store blocked protocol handlers is bounded.

7.3.3 Performance

CRL programs send many control messages and are therefore sensitive to send and receive overhead. Moreover, several CRL applications use small regions, which yields small data messages. Since many CRL actions use a request-reply style of communication, communication latencies also have an impact on performance.

Figures 7.18 and 7.19 show the performance of clean write misses for various numbers of readers. The home node H sends an invalidation to each reader and awaits all readers' acknowledgements before sending the data to requesting process P (see Figure 7.17(c)). The LFC curve shows the performance of a raw LFC test program that sends request and reply messages that have the same size as the messages sent by CRL in the case of zero readers (i.e., when no invalidations are needed).

For small regions, CRL adds approximately 5 μs (22%) to LFC's roundtrip latency (see Figure 7.18). CRL's throughput curves are close to the throughput curve of the LFC benchmark.

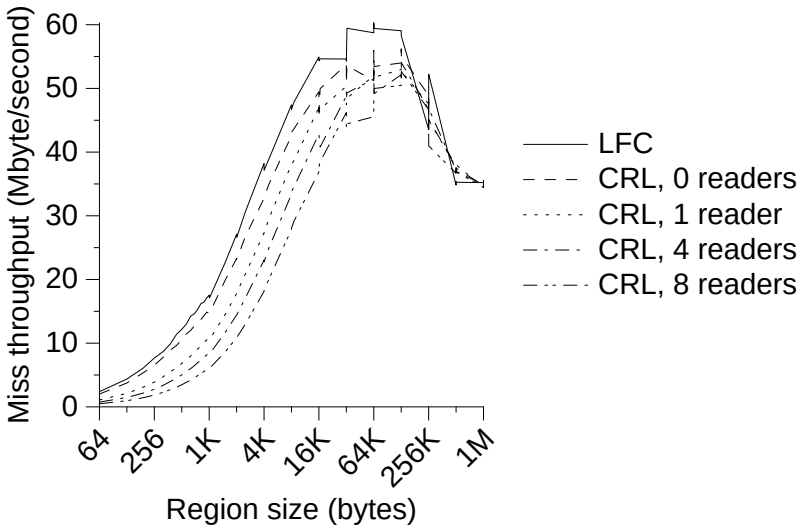


Fig. 7.19. Write miss throughput.

7.4 MPI — The Message Passing Interface

The Message Passing Interface (MPI) is a standard library interface that defines a large variety of message-passing primitives [53]. MPI is likely the most frequently used implementation of the message-passing programming model.

Several MPI implementations exist. Major vendors of parallel computers (e.g., IBM and Silicon Graphics) have built implementations that are optimized for their hardware and software. A popular alternative to these vendor-specific implementations is MPI Chameleon (MPICH) [61]. MPICH is free and widely used in workstation environments. We implemented MPI on LFC by porting MPICH to Panda. The following subsections describe this MPI/Panda/LFC implementation and its performance.

7.4.1 Implementation

MPICH consists of three layers. At the bottom, platform-specific *device channels* implement reliable, low-level send and receive functions. The middle layer, the *application device interface* (ADI), implements various send and receive flavors (e.g., rendezvous) using downcalls to the current device channel. The top layer implements the MPI interface. The ADI and the device channels have fixed interfaces. We ported MPICH version 1.1 and ADI version 2.0 to Panda by implementing a new device channel. This Panda device channel implements the basic send and receive functions using Panda's message-passing module.

Since MPICH is not a multithreaded system, the Panda device channel uses

a stripped-down version of Panda that does not support multithreading and that receives all messages through polling. We refer to this simpler Panda version as Panda-ST (Panda-*S*ingle*T*hreaded). The multithreaded version of Panda described in Chapter 6 is referred to as Panda-MT (Panda*M*ulti*T*hreaded).

By assuming that the Panda client is single-threaded, Panda-ST eliminates all locking inside Panda. Panda-ST also disables LFC's network interrupts. MPICH does not need interrupts, because all communication is directly tied to send and receive actions in user processes. It suffices to poll when a receive statement is executed (either by the user program or as part of the implementation of a complex function). Message handlers do not execute, not even logically, in the context of a dedicated upcall thread. Instead, they execute in the context of the application's main thread, both physically (on the same stack) and logically.

MPICH provides its own spanning-tree broadcast implementation. Unlike LFC's multicast implementation, however, MPICH forwards *messages* rather than packets. This way, the multicast primitive can be implemented easily in terms of MPI's unicast primitives which also operate on messages. An obvious disadvantage of this implementation choice is the loss of pipelining at processors that are internal nodes of a multicast tree and that have to forward messages. Given a large multicast message, such a processor will not start forwarding before it has received the entire message. If done on a per-packet basis, as in LFC, forwarding can begin as soon as the first packet has arrived at the forwarding processor's NI. We experimented both with MPICH's default broadcast implementation and with an implementation that uses Panda's unordered broadcast on top of LFC's broadcast. The performance of these two broadcast implementations, MPI-default and MPI-LFC, is discussed in the next section.

7.4.2 Performance

Below, we discuss the unicast and broadcast performance of MPICH on Panda and LFC.

Unicast Performance

Figure 7.20 and Figure 7.21 show MPICH's one-way latency and throughput, respectively. Both tests were performed using MPI's *MPI_Send*() and *MPI_Recv*() functions. For comparison, the figures also show the results of the corresponding tests at the Panda and LFC level.

Figure 7.20 shows MPI's unicast latency and, for comparison, the unicast latencies of Panda and LFC. MPICH is a heavyweight system and adds 6 μs (42%) to Panda's 0-byte one-way latency.

Except for small messages, MPICH attains the same throughput as Panda and

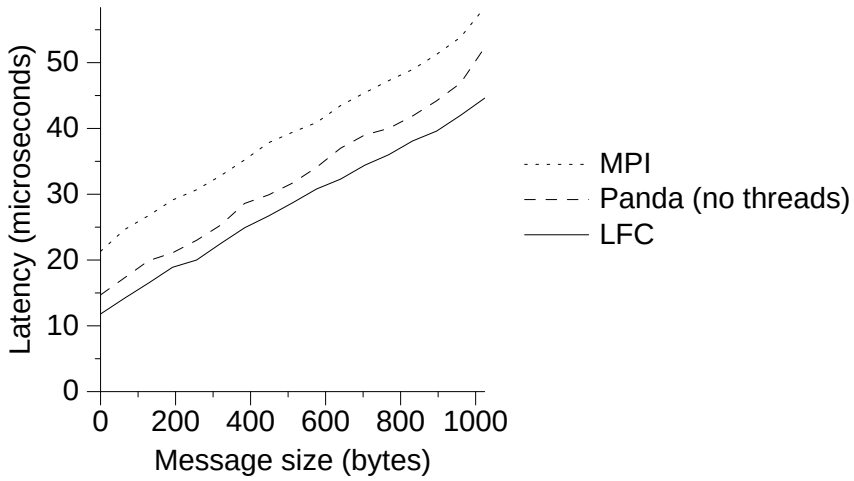


Fig. 7.20. MPI unicast latency.

LFC. The reason is that MPICH makes no extra passes over the data that is transferred. MPICH simply constructs an I/O vector and hands it to Panda, which then fragments the message. As a result, the MPICH layer does not incur any per-byte costs and throughput is good.

Broadcast Performance

Figure 7.22 shows multicast latency on 64 processors for LFC, Panda, and three MPI implementations. Both MPI-default results were obtained using MPICH's default broadcast implementation, which forwards broadcast messages as a whole rather than packet by packet. MPI-default normally uses binomial broadcast trees. Since LFC uses binary trees, we added an option to MPICH that allows MPI-default to use both binomial and binary trees. The MPI-LFC results were obtained using LFC's broadcast primitive (i.e., with binary trees).

As in the unicast case, we find that MPICH adds considerable overhead to Panda. Binary and binomial trees yield no large differences in MPI-default's broadcast performance, because 64-node binary and binomial trees have the same depth. There *is* a large difference between the MPI-default versions and MPI-LFC. With binary trees, MPI-default adds 67 μs (77%) to MPI-LFC's latency for a 64-byte message; with binomial trees, the overhead is 59 μs (68%). In this latency test it is not possible to pipeline multipacket messages. The cause of the performance difference is that MPI-default adds at least two extra packet copies to the criticial path at each internal node of the multicast tree: one copy from the network interface to the host and one copy per child from the host to the network interface.

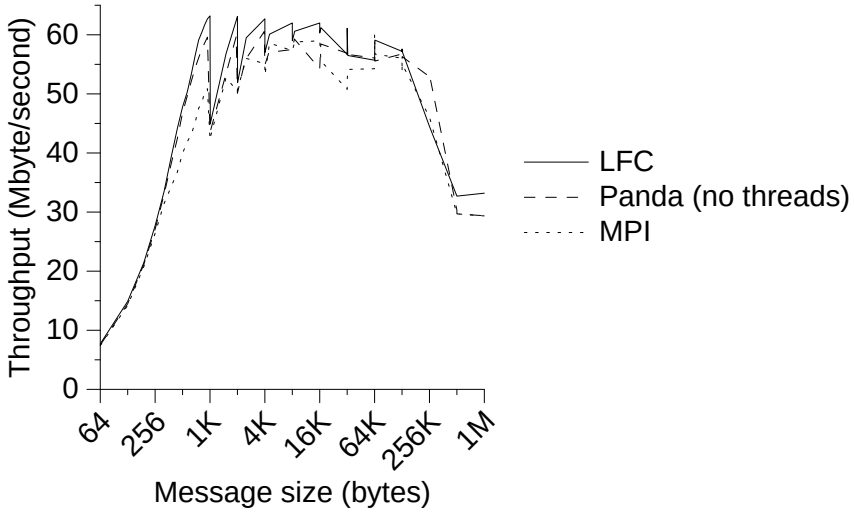


Fig. 7.21. MPI unicast throughput.

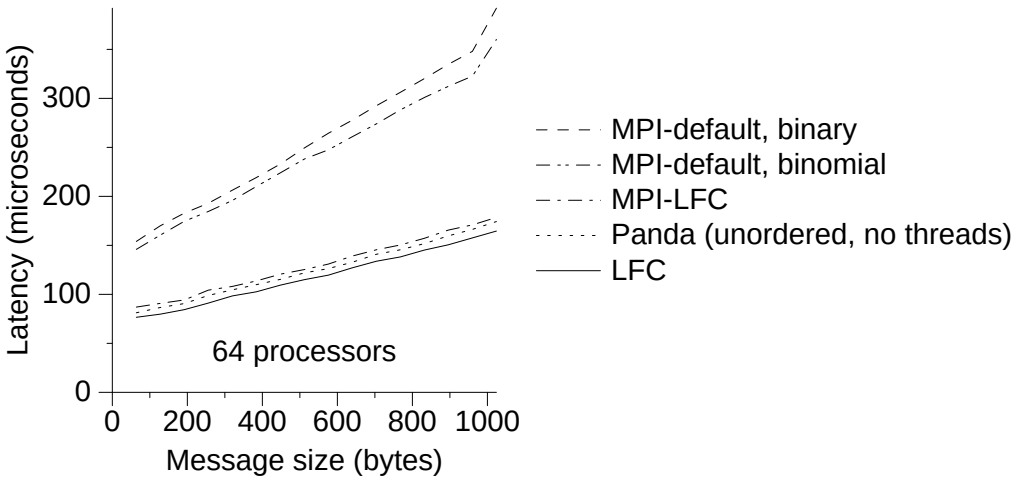


Fig. 7.22. MPI broadcast latency.

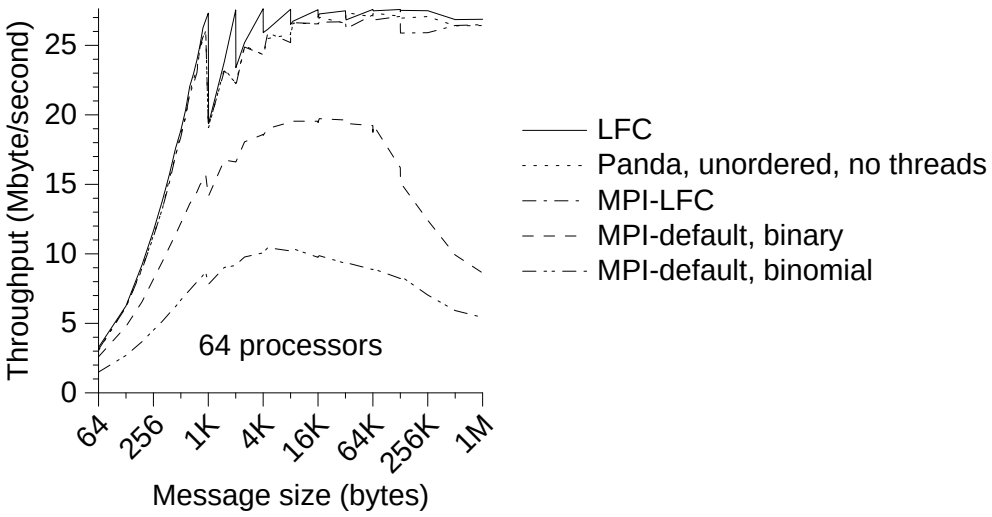


Fig. 7.23. MPI broadcast throughput.

Figure 7.23 compares the throughput of the same configurations. The difference between the MPI-default curves is caused by the difference in tree shape: binomial trees have a higher fanout than binary trees, which reduces throughput. Both MPI-default versions performs worse than MPI-LFC, not so much due to extra data copying, but because MPI-default does not forward multipacket multicast messages in a pipelined manner. In addition, host-level, message-based forwarding has a larger memory footprint than packet-based forwarding. When the message no longer fits into the L2 cache, MPI-default's throughput drops noticeably (from 20 to 9 Mbyte/s for binary trees and from 10 to 5 Mbyte/s for binomial trees). The problem is that the use of MPI-default results in one copying pass over the entire message for each child that a processor has to forward the message to. If the message does not fit in the cache, then these copying passes trash the L2 cache.

7.5 Related Work

We consider related work in two areas: operation transfer and operation execution.

7.5.1 Operation Transfer

Orca's operation-specific marshaling routines eliminate interpretation overhead by exploiting compile-time knowledge. In addition, they do not copy to intermediate RTS buffers, but generate an I/O vector which is passed to Panda's send

routines. The same approach is taken in Manta [99] (see Section 7.2), which generates specialized marshaling code for Java methods.

In their implementation of Optimistic Orca [66] by means of Optimistic Active Messages (OAMs), Hsieh et al. use compiler support to optimize the transfer of simple operations of simple object types (see also Section 6.6.3). For such operations, Optimistic Orca avoids the generic marshaling path and copies arguments directly into an (optimistic) active message. At the receiving side, these optimistic active messages are dispatched in a special way (see Section 7.5.2). We optimize marshaling for *all* operations, using a generic compilation strategy. To attain the performance of Optimistic Orca, however, further optimization would be necessary. For example, we would have to eliminate Panda's I/O vectors (which are expensive for simple operations).

Muller et al. optimized marshaling in Sun's RPC protocol by means of a program specialization tool, Tempo, for the C programming language [109]. Given the stubs generated by Sun's nonspecializing stub compiler this tool automatically generates specialized marshaling code by removing many tests and indirections. We use the Orca compiler instead of a general-purpose specialization tool to generate operation-specific code.

As described in Section 7.1.3, there are redundant data transfer operations on Orca's operation transfer path. These transfers can be removed by using DMA instead of PIO at the sending side, and by removing the receiver-side copy (from LFC packets to Orca data structures). Chang and von Eicken describe a zero-copy RPC architecture for Java, J-RPC, that removes these two data transfers [35]. J-RPC is a connection-oriented system in which receivers associate pinned pages with individual (per-sender) endpoints. Once a sender has filled these pages, the receiver unpins them and allocates new pages. The address of the new receive area is piggybacked on RPC replies.

J-RPC suffers from two problems. First, as noted by its designers, allocating pinned memory for each endpoint does not scale well to large numbers of processors. Second, J-RPC relies on piggybacking to return information about receive areas to a sender. This works well for RPC, but not for one-way multicasting.

7.5.2 Operation Execution

Whereas our approach has been to avoid expensive thread switches by hand, OAMs [66, 150], Lazy Task Creation [105], and Lazy Threads [58] all rely on compiler support. OAMs transform an AM handler that runs into a locked mutex into a true thread. The overhead of creating a thread is paid only when the lock operation fails. Hsieh et al. used OAMs to improve the performance of one of the first Panda-based Orca implementations. Like RTS-threads, this implementation used popup threads. On the CM-5, the implementation based on OAMs reduced

the latency of Orca object invocation by an order of magnitude by avoiding thread switching and by avoiding the CM-5's bulk transfer mechanism for small messages. In contrast with our approach, OAMs use compiler support and require that all locks be known in advance.

Draves and Bershad used continuations inside an operating system kernel to speed up thread switching [46]. Instead of blocking a kernel thread, a continuation is created and the same stack is used to run the next thread. We use the same technique in user space for the same reasons: to reduce thread switching overhead and memory consumption. Orca's continuation-based RTS uses continuations in the context of upcalls and accesses them through a condition-variable like interface.

Rühl and Bal use continuations in a collective-communication module implemented on top of Panda [123]. Several collective-communication operations combine results from different processors using a reverse spanning tree. Rather than assigning to one particular thread (e.g., the local computation thread) the task of waiting for, combining, and propagating data, the implementation stores intermediate results in continuations. Subresults are created either by the local computation thread or by an upcall thread. The first thread that creates a subresult creates a continuation and stores its subresult in this continuation. When another thread later delivers a new subresult, it merges its subresult with the existing subresult by invoking the continuation's resume function. This avoids thread switches to a dedicated thread.

7.6 Summary

This chapter has shown that the communication mechanisms developed in the previous chapters can be applied effectively to a variety of PPSs.

We described portable techniques to transfer and execute Orca operations efficiently. Operation transfer is optimized by letting the compiler generate operation-specific marshaling code. Also, the Orca RTS does not copy data to intermediate buffers: data is copied directly between LFC packets and Orca data structures. Operation execution is optimized by representing blocked invocations compactly by continuations instead of blocked threads. The use of continuations allows operations to be executed directly by Panda upcalls, saves memory, and avoids thread switching during the re-evaluation of the guards of blocked operations.

The Orca RTS exploits all communication mechanisms of Panda and LFC. Panda's transparent switching between polling and interrupts and LFC's polling watchdog work well for messages that carry operation requests or replies. An interrupt is generated when the receiver of a request is engaged in a long-running computation and does not respond to the request. RPC replies are usually received through polling.

Panda's totally-ordered broadcast is used to update replicated objects. This broadcast is implemented efficiently using LFC's NI-supported fetch-and-add and multicast primitives. LFC's NI-level fetch-and-add does not generate interrupts on the sequencer node. LFC's multicast may generate interrupts on the receiving nodes, but these interrupts are not on the multicast-packet forwarding path.

Panda's stream messages allow the RTS to marshal and unmarshal data without making unnecessary copies. Finally, Panda's upcall model works well for Orca, even though blocking upcalls occur frequently in Orca programs.

Manta has a similar programming model as Orca (shared objects) and also uses some of Orca's implementation techniques, in particular function shipping and compiler-generated, method-specific marshaling routines. The communication requirements of Manta and Orca are different, though. First, Manta does not replicate objects and therefore does not need multicast support. Second, Manta uses popup threads to process incoming operation requests, because Java methods can create new state before they block, which makes it difficult to represent a blocked invocation by a continuation (as in Orca). In the current implementation, the use of popup threads adds a thread switch to the execution path of all operations on remote objects.

In terms of its communication requirements, CRL is the simplest of the PPSs discussed in this chapter. CRL runs directly on top of LFC and benefits from LFC's efficient packet interface and its polling watchdog.

The implementation of MPI uses Panda's stream messages and Panda's broadcast. As in Orca and Manta, Panda's stream messages allow data to be copied directly between LFC packets and application data structures. Panda's broadcast is much more efficient than MPICH's default broadcast. MPICH performs all forwarding on the host, rather than on the NI. What is worse, however, is that MPICH forwards message-by-message rather than packet-by-packet, and therefore incurs the full message latency at each hop in its multicast tree.

All PPSs have the same data transfer behavior. At the sending side, each client copies data from client-level objects into LFC send packets in NI memory. At the receiving side, each client copies data from LFC receive packets in host memory to client-level objects.

Table 7.2 summarizes the minimum latency and the maximum throughput of characteristic PPS-level operations on PPS-level data. The table also shows the cost of the communication pattern induced by these operations, at all software levels below the PPS level. The performance results for different levels are separated by slashes; differences in these results are caused by layer-specific overheads. Recall that Panda can be configured with (Panda-MT) or without (Panda-ST) threads.

For CRL, Table 7.2 shows the cost of a clean write miss. The communication pattern consists of a small, fixed-size control message to the home node which replies with a data message containing a region. For MPI, the table shows the cost

PPS	Latency (μs)		Throughput (Mbyte/s)	
	Unicast	**Broadcast**	**Unicast**	**Broadcast**
LFC/CRL	23/28	−/−	60/56	−/−
LFC/Panda-ST/MPI	12/15/21	72/78/87	63/61/60	28/27/27
LFC/Panda-MT/Orca	24/31/46	72/103/121	58/56/52	28/28/27
LFC/Panda-MT/Manta	24/31/76	−/−/−	58/56/35	−/−/−

Table 7.2. PPS performance summary. All broadcast measurements were taken on 64 processors.

of a one-way message and the cost of a broadcast. For Orca, it shows the cost of an operation on a remote, nonreplicated object (which results in a RPC) and the cost of an operation on a replicated object (which results in an F&A operation followed by a broadcast). Finally, for Manta, the table shows the cost of a remote method invocation on a remote object.

The results in Table 7.2 show that our PPSs add significant overhead to an LFC-level implementation of the same communication pattern. These overheads stem from several sources.

1. Demultiplexing. All clients perform one or more demultiplexing operations. In Orca, for example, an operation request carries an object identifier and an operation identifier. In CRL, each message carries the address of a handler function and the name of a shared region. In MPI, each message carries a user-specified tag. MPI and Orca are both layered on top of Panda, which performs additional demultiplexing.

2. Fragmentation and reassembly. Both Panda and CRL implement fragmentation and reassembly. This requires extra header fields and extra protocol state.

3. Locking. Orca has a multithreaded runtime system that uses a single lock to achieve mutual exclusion. To avoid recursion deadlocks, this lock is released when the Orca runtime system invokes another software module (e.g., Panda) and acquired again when the invocation returns. Panda-MT also uses locks to protect global data structures.

4. Procedure calls. Although all PPSs and Panda use inlining inside software modules and export inlined versions of simple functions, many procedure calls remain. This is due to the layered structure of the PPSs. Layering is used to manage the complexity and to enhance the portability of these systems: most systems are fairly large and have been ported to multiple platforms.

The overheads added by the various software layers running on top of LFC are large compared to, for example, the roundtrip latency at the LFC level (e.g., an empty-array roundtrip for Manta adds 217% to an LFC roundtrip). Consequently, we expect that these overheads will dampen the application-level performance differences between different LFC implementations. Application performance is studied in the next chapter.

Chapter 8

Multilevel Performance Evaluation

The preceding chapters have shown that the LFC implementation described in Chapter 3 and Chapter 4 provides effective and efficient support for several PPSs. This implementation, however, is but one point in the design space outlined in Chapter 2. This chapter discusses alternative designs of two key components of LFC: reliable point-to-point communications by means of NI-level flow control and reliable multicast communication by means of NI-level forwarding. Both services are critically important for PPSs and their applications. Existing and proposed communication architectures take widely different approaches to these two issues (see Chapter 2), yet there have been few attempts to compare these architectures in a systematic way.

A key idea in this chapter is to use multiple implementations of LFC's point-to-point, multicast, and broadcast primitives. This allows us to evaluate the decisions made in the original design and implementation. We focus on assumptions 1 and 3 stated in Chapter 4: reliable network hardware and the presence of an intelligent NI. The alternative implementations all relax one or both of these assumptions. The use of a single programming interface is crucial; existing studies compare communication architectures with different functionality and different programming interfaces [5], which makes it difficult to isolate the effects of particular design decisions.

We compare the performance of five implementations at multiple levels. First, we perform direct comparisons between the systems by means of microbenchmarks that run directly on top of LFC's programming interface. Second, we compare the performance of parallel applications by running them on all five LFC implementations. Each of these applications uses one of four different PPSs: Orca, CRL, MPI, and Multigame. Orca, CRL, and MPI were described in the previous chapter; Multigame will be introduced later in this chapter. Our performance comparison thus involves three levels of software: the different LFC implementations, the PPSs, and the applications. A key contribution of this chapter is that it ties

together low-level design issues, the communication style induced by particular PPSs, and characteristics of parallel applications.

This chapter is organized as follows. Section 8.1 gives an overview of all LFC implementations. Section 8.2 and Section 8.3 present, respectively, the point-to-point and multicast parts of the implementations and compare the different implementations using microbenchmarks. Section 8.4 describes the communication style and performance of the PPSs used by our application suite. Section 8.5 discusses application performance. Section 8.6 classifies applications according to their performance on different LFC implementations. Section 8.7 studies related work.

8.1 Implementation Overview

LFC's programming interface and one implementation were described in Chapter 3 and Chapter 4. This section gives an overview of all implementations and describes those implementation components that are shared by all implementations.

8.1.1 The Implementations

We developed three point-to-point reliability schemes and two multicast schemes. Each LFC implementation consists of a combination of one reliability scheme and one multicast scheme. One of these LFC implementations corresponds to the system described in Chapters 3 and 4; we refer to this system as the *default* LFC implementation.

The point-to-point schemes are *no retransmission* (N_{rx}), *host-level retransmission* (H_{rx}), and *NI-level retransmission* (I_{rx}). Some systems, including the NI-level protocols described in Chapter 4, assume that the network hardware and software behave and are controlled as in an MPP environment. These systems exploit the high reliability of many modern networks and use nonretransmitting communication protocols [26, 32, 106, 113, 141, 154]. Such protocols are called *careful* protocols [106], because they may never drop packets, which requires careful resource management. The no-retransmission (N_{rx}) scheme represents these protocols; it assumes that the network hardware never drops or corrupts network packets and will fail if this assumption is violated.

Systems intended to operate outside an MPP environment usually implement retransmission. Retransmission used to be implemented on the host, but several research systems (e.g., VMMC-2 [49]) implement retransmission on a programmable NI. Our retransmitting schemes, H_{rx} and I_{rx}, represent these two types of systems.

Retransmission	**Forwarding**	
	Host forwards (H_{mc})	Interface forwards (I_{mc})
No retransmission (N_{rx})	$N_{rx}H_{mc}$	$N_{rx}I_{mc}$ (default)
Interface retransmits (I_{rx})	$I_{rx}H_{mc}$	$I_{rx}I_{mc}$
Host retransmits (H_{rx})	$H_{rx}H_{mc}$	Not implemented

Table 8.1. Five versions of LFC's reliability and multicast protocols.

The multicast schemes are *host-level multicast* (H_{mc}) and *NI-level multicast* (I_{mc}). On scalable, switched networks, multicasting is usually implemented by means of spanning-tree protocols that forward multicast data. Most communication systems implement these forwarding protocols on top of point-to-point primitives. Other systems use the NI to forward multicast traffic, which results in fewer data transfers and interrupts on the critical path from sender to receivers [16, 56, 68, 146]. H_{mc} and I_{mc} represent both types of systems.

The point-to-point reliability and the multicast schemes can be combined in six different ways (see Table 8.1). We implemented five out of these six combinations. The combination of host-level retransmission and interface-level multicast forwarding has not been implemented for reasons described in Section 8.3.1. The implementation described in Chapter 3 and Chapter 4 is essentially $N_{rx}I_{mc}$. There are, however, some small differences between the implementation described earlier and the one described here. ($N_{rx}I_{mc}$ recycles send descriptors in a slightly different way than the default LFC implementation.)

Table 8.2 summarizes the high-level differences between the five LFC implementations. We have already discussed reliability and multicast forwarding. All implementations use the same polling-watchdog implementation. The retransmitting implementations use retransmission and acknowledgement timers. $I_{rx}I_{mc}$ and $I_{rx}H_{mc}$ use Myrinet's on-board timer to implement these timers. $H_{rx}H_{mc}$ uses both Myrinet's on-board timer and the Pentium Pro's timestamp counter. All protocols except $H_{rx}H_{mc}$ implement LFC's fetch-and-add operation on the NI. Since $H_{rx}H_{mc}$ represents conservative protocols that make little use of the programmable NI, it stores fetch-and-add variables in host memory and handles fetch-and-add messages on the host.

8.1.2 Commonalities

The LFC implementations share much of their code. All implementations transmit data in variable-length packets. Hosts and NIs store packets in packet buffers which all have the same maximum packet size. Each NI has a send buffer pool and a receive buffer pool; hosts only have a receive buffer pool.

We distinguish four packet types: unicast, multicast, acknowledgement, and

Function	$N_{rx}I_{mc}$	$N_{rx}H_{mc}$	$I_{rx}I_{mc}$	$I_{rx}H_{mc}$	$H_{rx}H_{mc}$
Reliability	NI	NI	NI	NI	Host
Mcast forwarding	NI	Host	NI	Host	Host
Polling watchdog	NI + host	NI + host	NI + host	NI + host	NI + host
Fine-grain timer	—	—	NI	NI	NI + host
Fetch-and-add	NI	NI	NI	NI	host

Table 8.2. Division of work in the LFC implementations.

synchronization. Unicast and multicast packets contain client data. Acknowledgements are used to implement reliability. Synchronization packets carry fetch-and-add (F&A) requests and replies. An F&A operation is implemented as an RPC to the node holding the F&A variable. Depending on the implementation, this variable is stored either in host or NI memory.

All implementations organize multicast destinations in a binary tree with the sender as its root and forward multicast packets along this tree, in parallel. One of the problems in implementing a multicast forwarding scheme is the potential for buffer deadlock. Several strategies can be used to deal with this problem: reservation in advance [56, 146], deadlock-free routing, and deadlock recovery [16].

To send a packet, LFC's send routines store one or more transmission requests in send descriptors in NI memory (using PIO). Each descriptor identifies a send packet, the packet's destination, the packet's size, and protocol-specific information such as a sequence number. Multiple descriptors can refer to the same packet, so that the same data can be transmitted to multiple destinations.

LFC clients use PIO to copy data that is to be transmitted into LFC send packets (which are stored in NI memory). To speed up these data copies, all implementations mark NI memory as a write-combined memory region (see Section 3.3).

Each host maintains a pool of free receive buffers. The addresses of free host buffers are passed to the NI control program through a shared queue in NI memory. When the NI has received a packet in one of its receive buffers, it copies the packet to a free host buffer (using DMA). The host can poll a flag in this buffer to test whether the packet has been filled.

The NI control program's event loop processes the following events:

1. Host transmission request. The host enqueued a packet for retransmission. The NI tries to move the packet to its *packet transmission queue*. In some protocols, however, the packet may have to wait until an NI-level send window opens up. If the window is closed, the packet is stored on a per-destination *blocked-sends queue*. Incoming acknowledgements will later open the window and move the packet from this queue to the packet transmission queue.

Parameter	Meaning	Unit
P	#processors that participate in the application	Processors
PKTSZ	Maximum payload of a packet	Bytes
ISB	NI send pool size	Packet buffers
IRB	NI receive pool size	Packet buffers
HRB	Host receive pool size	Packet buffers
W	Maximum send window size	Packets
INTD	Polling watchdog's network interrupt delay	μs
TGRAN	Timer granularity	μs

Table 8.3. Parameters used by the protocols.

2. Packet transmitted. The NI hardware completed the transmission of a packet. If the packet transmission queue is not empty, the NI starts a network DMA to transmit the next packet. For each outgoing packet, Myrinet computes (in hardware) a CRC checksum and appends it to the packet. The receive hardware recomputes and verifies the checksum and appends the result (checksum verification succeeded or failed) to the packet. This result is checked in software.

3. Packet received. The NI hardware received a packet from the network. The NI checks the checksum of the packet just received. If the checksum fails, the NI drops the packet or signals an error to the host. (The retransmitting protocols drop the packet, forcing the sender to retransmit. The non-retransmitting protocols signal an error.) Otherwise, if the packet is a unicast or multicast packet, then the NI enqueues it on its *NI-to-host DMA queue*. Some protocols also enqueue multicast packets for forwarding on the packet transmission queue or, if necessary, on a blocked-sends queue. Acknowledgement and synchronization packets are handled in a protocol-specific way.

4. NI-to-host DMA completion. If the NI-to-host DMA queue is not empty, then a DMA is started for the next packet.

5. Timeout. All implementations use timeouts to implement the NI-level polling watchdog described in Section 3.7. Some implementations also use timeouts to trigger retransmissions or acknowledgements.

Each implementation is configured using the parameters shown in Table 8.3.

8.2 Reliable Point-to-Point Communication

To compare different point-to-point protocols, we have implemented LFC's reliable point-to-point primitive in three different ways. The retransmitting protocols (H_{rx} and I_{rx}) assume unreliable network hardware and can recover from lost, corrupted, and dropped packets by means of time-outs, retransmissions, and CRC checks. In H_{rx}, reliability is implemented by the host processor; in I_{rx}, it is implemented by the NI. As explained above, the three implementations have much in common; below, we discuss how the three implementations differ from one another.

8.2.1 The No-Retransmission Protocol (N_{rx})

N_{rx} assumes reliable network hardware and correct operation of all connected NIs. N_{rx} does not detect lost packets and treats corrupted packets as a fatal error. N_{rx} never retransmits any packet and can therefore avoid the overhead of buffering packets for retransmission and timer management.

To avoid buffer overflow, N_{rx} implements flow control between each pair of NIs (see the UCAST protocol described in Section 4.1). At initialization time, each NI reserves $W = \lfloor IRB/P \rfloor$ receive buffers for each sender. The number of buffers (W) is the window size for the sliding window protocol that operates between sending and receiving NIs. Each NI transmits a packet only if it knows that the receiving NI has free buffers. If a packet cannot be transmitted due to a closed send window, then the NI queues the packet on a blocked-sends queue for that destination. Blocked packets are dequeued and transmitted during acknowledgement processing (see below).

Each NI receive buffer is released when the packet contained in it has been copied to host memory. The number of newly released buffers is piggybacked on each packet that flows back to the sender. If there is no return traffic, then the receiving NI sends an explicit *half-window acknowledgement* after releasing $W/2$ buffers. (That is, the credit refresh threshold discussed in Section 4.1 is set to $W/2$.)

8.2.2 The Retransmitting Protocols (H_{rx} and I_{rx})

The two retransmission schemes make different tradeoffs between send and receive overhead on the host processor and complexity of the NI control program. We first describe the protocols in general terms and then explain protocol-specific modifications. In the following, the terms *sender* and *receiver* refer to NIs in the case of I_{rx} and to host processors in the case of H_{rx}.

Each sender maintains a send window to each receiver. A sender that has

filled its send window to a particular destination is not allowed to send to that destination until some of the packets in the window have been acknowledged by the receiver. Each packet carries a sequence number which is used to detect lost, duplicated, and out-of-order packets. After transmitting a packet, the sender starts a retransmission timer for the packet. When this timer expires, all unacknowledged packets are retransmitted. This go-back-N protocol [115, 138] is efficient if packets are rarely dropped or corrupted (as is the case with Myrinet).

Retransmission requires that packets be buffered until an acknowledgement is received. With asynchronous message passing, this normally requires a copy on the sending host. However, since the Myrinet hardware can transmit only packets that reside in NI memory, all protocols must copy data from host memory to NI memory. The retransmitting protocols use the packet in NI memory for retransmission and therefore do not make more memory copies than N_{rx}. This optimization is hardware-specific; it does not apply to NIs that use FIFOs into the network (e.g., several ATM interfaces).

Receivers accept a packet only if it carries the next-expected sequence number; otherwise they drop the packet. In addition, NIs drop incoming packets that have been corrupted —i.e., packets that yield a CRC error— or that cannot be stored due to a shortage of NI receive buffers.

Since send packets cannot be reused until they have been acknowledged and since NI memory is relatively small, senders can run out of send packets when acknowledgements do not arrive promptly. To prevent this, receivers acknowledge incoming data packets in three ways. As in N_{rx}, receivers use piggybacked and half-window acknowledgements. In addition, each receiver sets an acknowledgement timer whenever a packet arrives. When the timer expires and the receiver has not yet sent any acknowledgement, the receiver sends an explicit *delayed acknowledgement*.

In I_{rx}, receiving NIs keep track of the amount of buffer space available. When a packet must be dropped due to a buffer space shortage, the NI registers this. When more buffer space becomes available, the NI sends a NACK to the sender of the dropped packet. When a NACK is received, the sender retransmits all unacknowledged packets. Without the NACKs, packets would not be retransmitted until the sender's timer expired.

8.2.3 Latency Measurements

Figure 8.1 shows the one-way latency for N_{rx}, H_{rx}, and I_{rx}. All messages fit in a single network packet and are received through polling. N_{rx} outperforms the retransmitting protocols, which must manage timers and which perform more work on outstanding send packets when an acknowledgement is received. All curves have identical slopes, indicating that the protocols have identical per-byte costs.

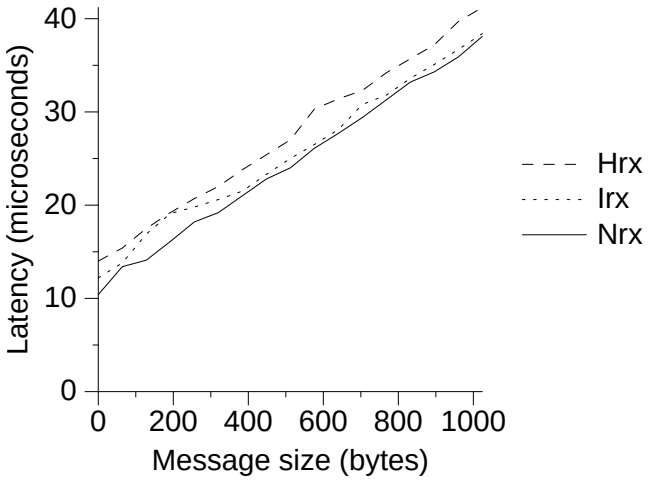


Fig. 8.1. LFC unicast latency.

Protocol	o_s	o_r	L	g	$o_s + o_r + L$
N_{rx}	1.5	2.3	7.2	6.7	11.0
I_{rx}	1.4	2.3	8.3	9.7	12.0
H_{rx}	2.4	5.3	6.5	8.3	14.2

Table 8.4. LogP parameter values (in microseconds) for a 16-byte message.

Table 8.4 shows the values of the LogP [40, 41] parameters and the end-to-end latency ($o_s + o_r + L$) of the three protocols. The LogP parameters were defined in Section 5.1.2. N_{rx} and I_{rx} perform the same work on the host, so they have (almost) identical send and receive overheads (o_s and o_r, respectively). I_{rx}, however, runs a retransmission protocol on the NI, which is reflected in the protocol's larger gap ($g = 9.7$ versus $g = 6.7$) and latency ($L = 8.3$ versus $L = 7.2$). H_{rx} runs a similar protocol on the host and therefore has larger send and receive overheads than N_{rx} and I_{rx}.

Table 8.5 shows the F&A latencies for the different protocols. Again, N_{rx} is the fastest protocol (18 μs). H_{rx} is the slowest protocol (31 μs); it is the only protocol in which F&A operations are handled by the host processor of the node that stores the F&A variable. In an application, handling F&A requests on the host processor can result in interrupts. In this benchmark, however, H_{rx} receives all F&A requests through polling.

8.2.4 Window Size and Receive Buffer Space

To obtain maximum throughput, acknowledgements must flow back to the sender before the send window closes. To avoid sender stalls, all protocols therefore

Protocol	Latency (μs)
N_{rx}	18
I_{rx}	25
H_{rx}	31

Table 8.5. Fetch-and-add latencies.

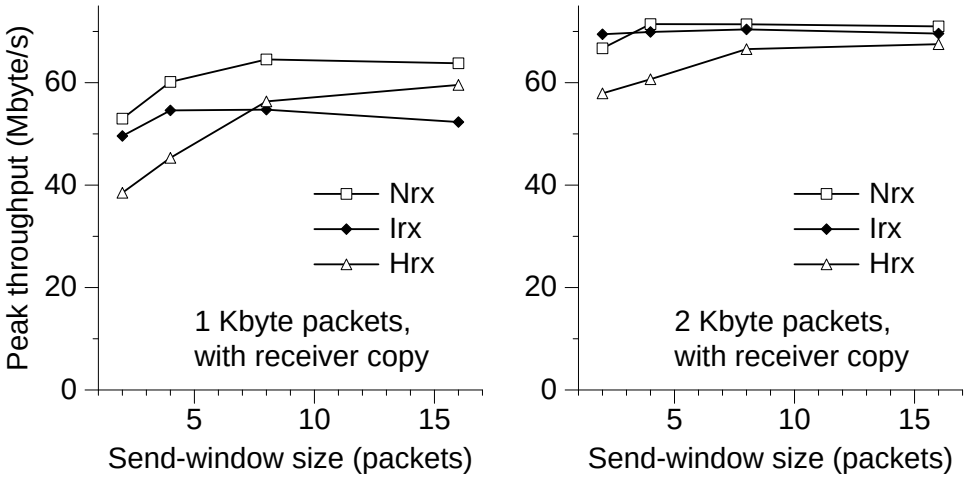


Fig. 8.2. Peak throughput for different window sizes.

need sufficiently large send windows and receive buffer pools. Figure 8.2 shows the measured peak throughput for various window sizes and for two packet sizes. To speed up sequence number computations for I_{rx}, we use only power-of-two window sizes. In this benchmark, the receiving process copies data from incoming packets to a receive buffer. This is not strictly necessary, but all LFC clients perform such a copy while processing the data they receive. Throughput without this receiver-side copy is discussed later.

For 1 Kbyte packets, N_{rx} and I_{rx} need only a small window ($W \geq 4$) to attain high throughput. H_{rx}'s throughput still increases noticeably when we grow the window size from $W = 4$ (45 Mbyte/s) to $W = 8$ (56 Mbyte/s). A small window is important for N_{rx}, because each NI allocates W receive buffers *per sender*. In the retransmitting protocols (I_{rx} and H_{rx}) NI receive buffers are shared between senders and we need only W receive buffers to allow any single sender to achieve maximum throughput.

Surprisingly, I_{rx}'s throughput *decreases* for $W > 8$. The reason is that NI-level acknowledgement processing in I_{rx} involves canceling all outstanding retransmission timers. With a larger window, more (virtual) timers must be canceled. (I_{rx} multiplexes multiple virtual timers on a single physical timer; canceling a virtual timers consists of a dequeue operation.) At some point, the slow NI can no longer

hide this work behind the current send DMA transfer. The effect disappears when we switch to 2 Kbyte packets — that is, when the send DMA transfers take more time.

Figure 8.3 shows one-way throughput for 1-Kbyte and 2-Kbyte packets, with and without receiver-side copying. For all three protocols, the window size was set to $W = 8$.

With 1 Kbyte packets and without copying, we see clear differences in throughput between the three protocols. Performing retransmission administration on the host (H_{rx}) increases the per-packet costs and reduces throughput. When the retransmission work is moved from the host to the NI (I_{rx}), throughput is reduced further. The NI performs essentially the same work as the host in H_{rx}, but does so more slowly. The LogP measurements in Table 8.4 confirm this: I_{rx} has a larger gap (g) than the other protocols. Increasing the packet size to 2 Kbyte reduces the number of times per-packet costs are incurred and increases the throughput of all protocols, but most noticeably for I_{rx}. For H_{rx} and N_{rx}, the improvement is smaller and comes mainly from faster copying at the sending side. I_{rx}, however, is now able to hide more of its per-packet overhead in the latency of larger DMA transfers.

Throughput is reduced when the receiving process copies data from incoming packets to a destination buffer, especially when the message is large and does not fit into the L2 cache. This is important, because all LFC clients do this. Since the maximum *memcpy*() speed on the Pentium Pro (52 Mbyte/s) is less than the maximum throughput achieved by our protocols (without copying), the copying stage becomes a bottleneck. In addition, memory traffic due to copying interferes with the DMA transfers of incoming packets.

With copying, the throughput differences between the three protocols are fairly small. To a large extent, this is due to the use of the NI memory as 'retransmission memory,' which eliminates a copy at the sending side.

8.2.5 Send Buffer Space

While NI receive buffer space is an issue for N_{rx}, send buffer space is just as important for I_{rx} and H_{rx}. First, however, we consider the send buffer space requirements for N_{rx}. N_{rx} can reuse send buffers as soon as their contents have been put on the wire, so only a few send buffers are needed to achieve maximum throughput between a single sender-receiver pair. With more receivers it is useful to have a larger send pool, in case some receiver does not consume incoming packets promptly. Transmissions to that receiver will be suspended and extra packets are then needed to allow transmissions to other receivers.

In the retransmitting protocols, send buffers cannot be reused until an acknowledgement confirms the receipt of their contents. (This is a consequence of

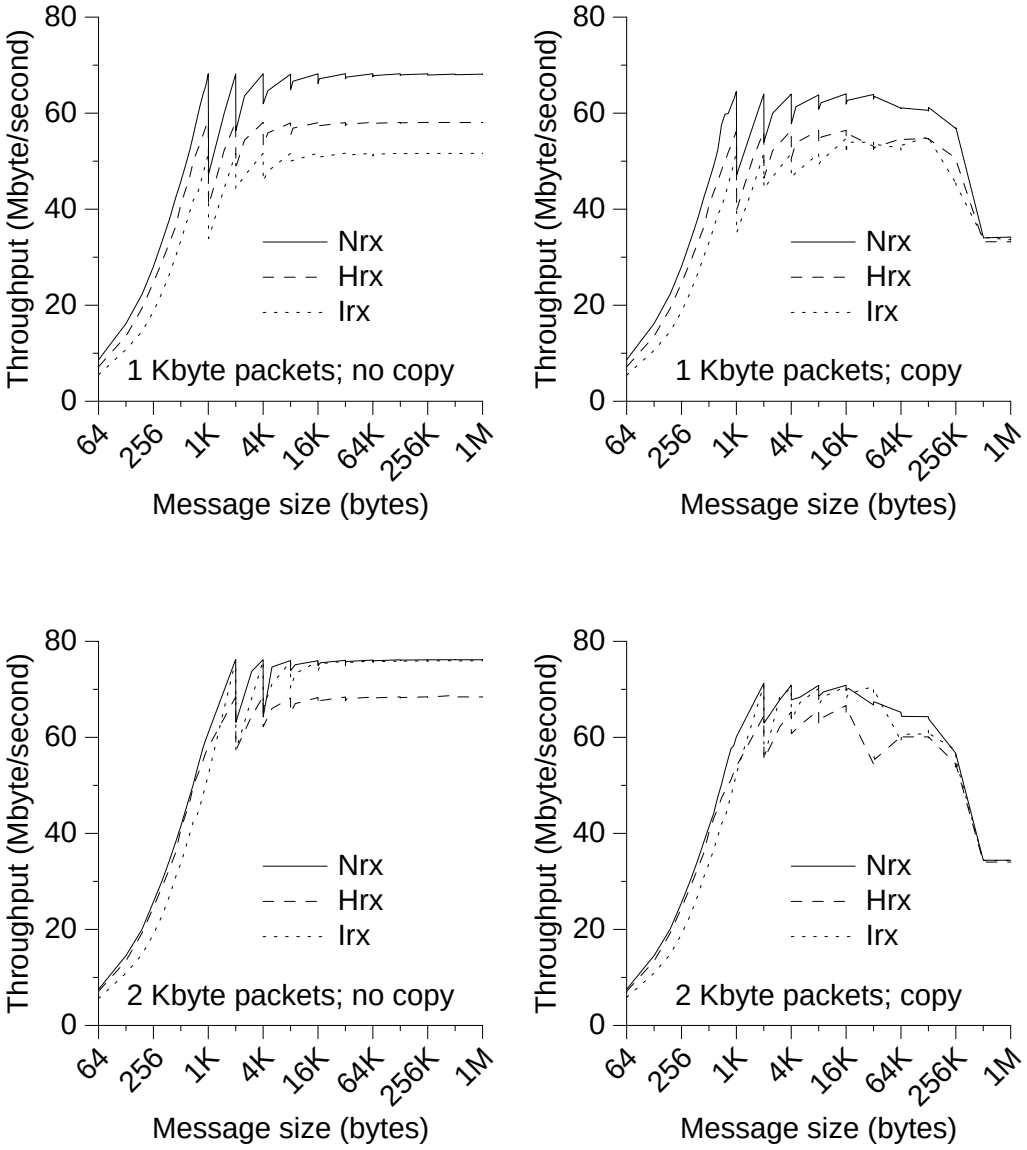


Fig. 8.3. LFC unicast throughput.

using NI memory as retransmission memory.) Since we do not acknowledge each packet individually, a sender that communicates with many receivers may run out of send buffers. One of our applications, a Fast Fourier Transform (FFT) program, illustrates this problem. In FFT, each processor sends a unique message to every other processor. With 64 processors, these messages are of medium size (2 Kbyte) and fill less than half a window. If the number of send buffers is small, a sender may run out of send buffers before any receiver's half-window acknowledgement has been triggered.

If the number of send buffers is large relative to the number of receivers, the sender will trigger half-window acknowledgements before running out of send buffers. In general, we cannot send more than $P * (W/2 - 1)$ packets without triggering at least one half-window acknowledgement, unless packets are lost.

To avoid unnecessary retransmissions when the number of send buffers is small, I_{rx} and H_{rx} could acknowledge packets individually, but this will lead to increased network occupancy, NI occupancy, and (in the case of H_{rx}) host occupancy. Instead, I_{rx} and H_{rx} start an acknowledgement timer for incoming packets. Each receiver maintains one timer per sender. The timer for sender S is started whenever a data packet from S arrives and the timer is not yet running. The timer is canceled when an acknowledgement (possibly piggybacked) travels back to S. If the timer expires, the receiver sends an explicit acknowledgement.

This scheme requires small timeout values for the acknowledgement timer, because a sender needs only a few hundred microseconds to allocate all send buffers. Using OS timer signals to implement the timer on H_{rx} does not work, because the granularity of OS timers is often too coarse (10 ms is a common value). The Myrinet NI, however, provides access to a clock with a granularity of 0.5 μs *and* is able to send signals to the user process. We therefore use the NI as an intelligent clock, as follows. We choose a clock period $TGRAN$ and let the NI generate an interrupt every $TGRAN$ μs.

To reduce the number of clock signals, we use two optimizations. First, before generating a clock interrupt, the NI reads a flag in host memory that indicates whether any timers are running. No interrupt is generated when no timers are running. Second, each time the host performs a timer operation it records the current time by reading the Pentium Pro's timestamp counter (see Section 1.6.2). This counter is incremented every clock cycle (5 ns). If necessary, the host invokes timeout handlers. Before generating a clock interrupt, the NI reads the time recorded by the host to decide whether the host recently read its clock. If so, no interrupt is generated.

8.2.6 Protocol Comparison

An important difference between the retransmitting (I_{rx} and H_{rx}) and the non-retransmitting (N_{rx}) protocols is that the retransmitting protocols do not reserve buffer space for individual senders. In I_{rx} and H_{rx}, NI receive buffers can be shared by all sending NIs, because an NI that runs out of receive buffers can drop incoming packets, knowing that the senders will eventually retransmit. In I_{rx} and H_{rx}, a single sender can fill all of an NI's receive buffers. N_{rx}, in contrast, may never drop packets and therefore allocates a fixed number of NI receive buffers to each sender. In most cases this is wasteful, but since the small bandwidth-delay product of N_{rx} allows small window sizes, this strategy does not pose problems in medium-size clusters. Nevertheless, this static buffer allocation scheme prevents N_{rx} from scaling to very large clusters, unless NI memory scales as well.

Acknowledgements in N_{rx} have a different meaning than in I_{rx} and H_{rx}. N_{rx} implements flow control (at the NI level), while I_{rx} and H_{rx} implement only reliability. In N_{rx}, an acknowledgement consists of a count of buffers released since the last acknowledgement. In I_{rx} and H_{rx}, an acknowledgement consists of a sequence number. (A count instead of a sequence number fails when an acknowledgements is lost.) The sequence number indicates which packets have arrived at the receiver (host or NI), but does not indicate which packets have been released.

An obvious disadvantage of N_{rx} is that it cannot recover from lost or corrupted packets. On Myrinet, we have observed both types of errors, caused by both hardware and software bugs. On the positive side, careful protocols are easier to implement than retransmitting protocols. There is no need to buffer packets for retransmission, to maintain sequence numbers, or to deal with timers. For the same reasons, the nonretransmitting LFC implementations achieve better latencies than the retransmitting implementations. With 1 Kbyte packets there is also a throughput advantage, but this is partly obscured by the memory-copy bottleneck.

In spite of these differences, the microbenchmarks indicate that all three protocols can be made to perform well, including protocols that perform most of their work on the NI. The keys to good performance are to use NI buffers for retransmission and to overlap DMA transfers with useful work.

8.3 Reliable Multicast

We consider two multicast schemes, which differ in where they implement the forwarding of multicast packets. In the H_{mc} protocols, multicast packets are forwarded by the host processor. In the I_{mc} protocols, the NI forwards multicast packets. In Section 8.3.1, we first compare host-level and interface-level forwarding. Section 8.3.2 compares individual implementations in more detail. Section 8.3.3

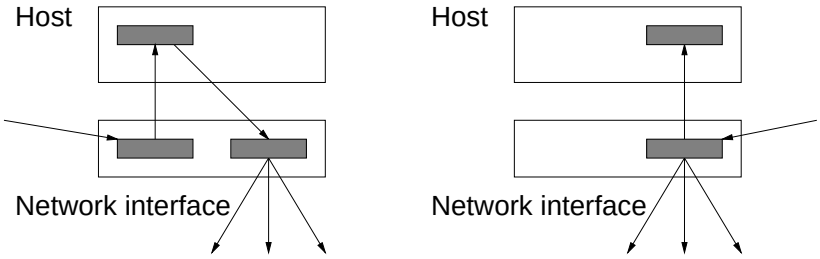


Fig. 8.4. Host-level (left) and interface-level (right) multicast forwarding.

discusses deadlock issues. Finally, Section 8.3.4 analyzes the performance of all multicast implementations.

8.3.1 Host-Level versus Interface-Level Packet Forwarding

Host-level forwarding using a spanning tree is the traditional approach for networks without hardware multicast. The sender is the root of a multicast tree and transmits a packet to each of its children. Each receiving NI passes the packet to its host, which reinjects the packet to forward it to the next level in the multicast tree (see Figure 8.4). All host-level forwarding protocols reinject each multicast packet at most once. Instead of making a separate host-to-NI copy for each forwarding destination, the host creates multiple transmission requests for the same packet.

Host-level forwarding has three disadvantages. First, since each reinjected packet was already available in the NI's memory, host-level forwarding results in an unnecessary host-to-NI data transfer at each internal tree node of the multicast tree, which wastes bus bandwidth and processor time. Second, no forwarding takes place unless the host processes incoming packets. If one host does not poll the network in a timely manner, all its children will be affected. Instead of relying on polling, the NI can raise an interrupt, but interrupts are expensive. Third, the critical sender-receiver path includes the host-NI interactions of all the nodes between the sender and the receiver. For each multicast packet, these interactions consist of copying the packet to the host, host processing, and reinjecting the packet.

The NI-level multicast implementations address all three problems. In the I_{mc} protocols, the host does not reinject multicast packets into the network (which saves a data transfer), forwarding takes place even if the host is not polling, and the host-NI interactions are not on the critical path.

Host-level and NI-level multicast forwarding can be combined with the three point-to-point reliability schemes described in the previous section. We implemented all combinations except the combination of interface-level forwarding

with host-level retransmission. In such a variant we expect the host to maintain the send and receive window administration, as in H_{rx}. However, if an NI needs to forward an incoming packet then it needs a sequence number for each of the forwarding destinations. To make this scheme work we would have to duplicate sequence number information (in a consistent way).

8.3.2 Acknowledgement Schemes

For reliability, multicast packets must be acknowledged by their receivers, just like unicast packets. To avoid an acknowledgement implosion, most multicast forwarding protocols use a reverse acknowledgement scheme in which acknowledgements flow back along the multicast tree.

Our protocols use two reverse acknowledgement schemes: ACK-forward and ACK-receive. ACK-forward is used by $N_{rx}I_{mc}$(the default implementation), $I_{rx}I_{mc}$, and $H_{rx}H_{mc}$. In these protocols, acknowledgements are sent at the level (NI or host) at which multicast forwarding takes place. The main characteristic of ACK-forward is that it does not allow a receiver to acknowledge a multicast packet to its parent in the multicast tree until that packet has been forwarded to the receiver's children.

ACK-forward cannot easily be used by $N_{rx}H_{mc}$ or $I_{rx}H_{mc}$, because in these protocols acknowledgements are sent by the NI and forwarding is performed by the host. $N_{rx}H_{mc}$ and $I_{rx}H_{mc}$ therefore use a simpler scheme, ACK-receive, which acknowledges multicast packets in the same way as unicast packets (i.e., without waiting for forwarding). In the case of $N_{rx}H_{mc}$, multicast packets can be acknowledged as soon as they have been delivered to the local host. In the case of $I_{rx}H_{mc}$, multicast packets can be acknowledged as soon as they have been received by the NI.

A potential problem with ACK-receive is that it creates a flow-control loophole that cannot easily be closed by an LFC client. In $N_{rx}H_{mc}$ and $I_{rx}H_{mc}$, a host receive buffer containing a multicast packet is not released until the packet has been delivered to the client *and* has been forwarded to all children. Consequently, after processing a packet, the client cannot be sure that the packet is available for reuse, because it may still have to be forwarded. This makes it difficult for the client to implement a fool-proof flow-control scheme (when needed).

8.3.3 Deadlock Issues

With a spanning-tree multicast protocol, it is easy to create a buffer deadlock cycle in which all NIs (or hosts) have filled their receive pools with packets that need to be forwarded to the next NI (or host) in the cycle which also has a full receive pool. Such deadlocks can be prevented or recovered from in several ways. The

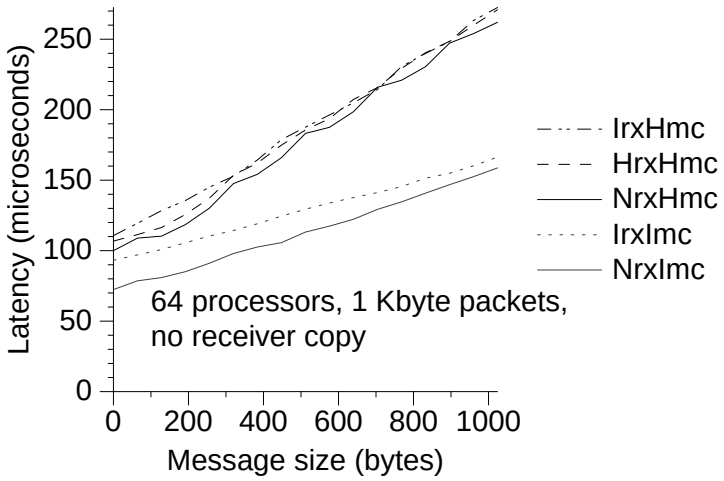


Fig. 8.5. LFC multicast latency.

NI-level flow control protocol of $N_{rx}I_{mc}$(the default implementation), for example, allows deadlock-free multicasting for a certain class of multicast trees (including binary trees).

Our other protocols do not implement true flow control at the multicast forwarding level and may therefore suffer from deadlocks. For certain types of multicast trees, including binary trees, the ACK-forward scheme prevents buffer deadlocks *if* sufficient buffer space is available at the forwarding level. Buffer space is sufficient if each receiver can store a full send window from each sender (see Appendix A). $N_{rx}I_{mc}$'s NI-level flow control scheme satisfies this requirement. Unfortunately, reserving buffer space for each sender destroys one of the advantages of retransmission: better utilization of receive buffers (see Section 8.2.4).

$N_{rx}H_{mc}$ and $I_{rx}H_{mc}$ do not provide any flow control at the forwarding (i.e., host) level and are therefore susceptible to deadlock. With our benchmarks and applications, however, the number of host receive buffers is sufficiently large that deadlocks do not occur.

8.3.4 Latency and Throughput Measurements

Multicast latency for 64 processors is shown in Figure 8.5. We define multicast latency as the time it takes to reach the last receiver. The top three curves in the graph correspond to the H_{mc} protocols, which perform multicast forwarding on the host processor. These protocols clearly perform worse than those that perform multicast forwarding on the NI. The reason is simple: with host-level forwarding two extra data copies occur on the critical path at each internal node of the multicast tree.

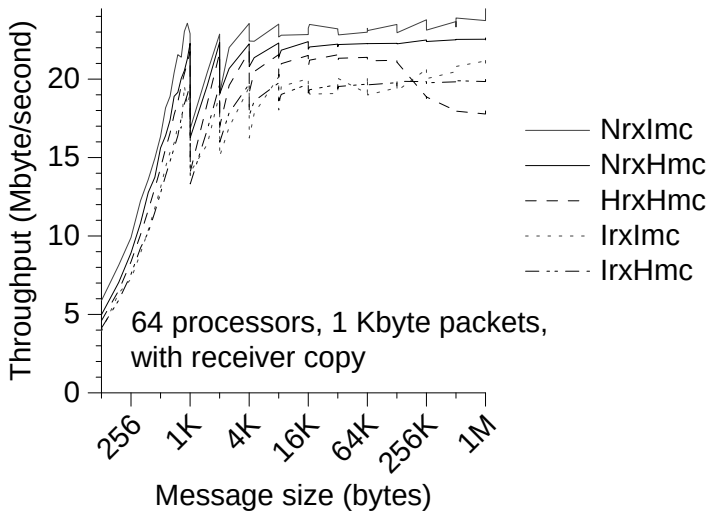


Fig. 8.6. LFC multicast throughput.

Figure 8.6 shows multicast throughput on 64 processors, using 1 Kbyte packets and copying at the receiver side. Throughput in this graph refers to the sender's outgoing data rate, not to the summed receiver-side data rate. In contrast with the latency graph (Figure 8.5), this graph shows that interface-level forwarding does not always yield the best results: for message sizes up to 128 Kbyte, for example, $H_{rx}H_{mc}$ achieves higher throughput than $I_{rx}I_{mc}$.

The dip in the $H_{rx}H_{mc}$ curve is the result of $H_{rx}H_{mc}$'s higher receive overhead and a cache effect. For larger messages, the receive buffer no longer fits in the L2 cache, and so the copying of incoming packets becomes more expensive, because these copies now generate extra memory traffic. In $H_{rx}H_{mc}$, these increased copying costs turn the receiving host into the bottleneck: due to its higher receive overhead, $H_{rx}H_{mc}$ does not manage to copy a packet completely before the next packet arrives. In the other protocols, the receive overhead is lower, so there is enough time to copy an incoming packet before the next packet arrives.

8.4 Parallel-Programming Systems

All applications discussed in this chapter run on one of the following PPSs: Orca, CRL, MPI, or Multigame. Orca, CRL, and MPI were discussed in Chapter 7; Multigame is introduced below. This section describes the communication style of all PPSs and discusses PPS-level performance.

8.4.1 Communication Style

Each of our PPSs implements a small number of communication patterns, which are executed in response to actions in the application. For each PPS, we summarize its main communication patterns and discuss the interactions between these patterns and the communication system.

CRL

In CRL, most communication results from read and write misses. The resulting communication patterns were described in Section 7.3.2 (see Figure 7.17).

CRL's invalidation-based coherence protocol does not allow applications to multicast data. (CRL uses LFC's multicast primitive only during initialization and barrier synchronization). Also, since CRL's single-threaded implementation does not allow a requesting process to continue while a miss is outstanding, applications are sensitive to roundtrip performance. Roundtrip latency is more important than roundtrip throughput, because CRL mainly sends small messages. Control messages are always small and data messages are small because most applications use fairly small regions.

The roundtrip nature of CRL's communication patterns and the small region sizes used in most CRL applications lead to low acknowledgement rates for the N_{rx} protocols. The roundtrips make piggybacking effective and due to the small data messages few half-window acknowledgements are needed. Finally, $N_{rx}I_{mc}$ and $N_{rx}H_{mc}$ never send delayed acknowledgements. This is important: for one of our applications (Barnes, see Section 8.5), for example, $H_{rx}H_{mc}$ sends 31 times as many acknowledgements as $N_{rx}I_{mc}$, the default LFC implementation.

MPI

Since MPI is a message-passing sysytem, programmers can express many different communication patterns. MPI's main restriction is that incoming messages cannot be processed asynchronously, which occasionally forces programmers to insert polling statements into their programs. The sensitivity of an MPI program to the parameters of the communication architecture depends largely on the application's communication rate and communication pattern.

All our MPI applications (QR, ASP, and SOR) use MPI's collective-communication operations. Internally, these operations (broadcast and reduce-to-all) all use broadcasting.

All MPI measurements in this chapter were performed using the same MPICH-based MPI implementation as described in Section 7.4. Recall that MPICH provides a default spanning-tree broadcast implementation built on top of unicast

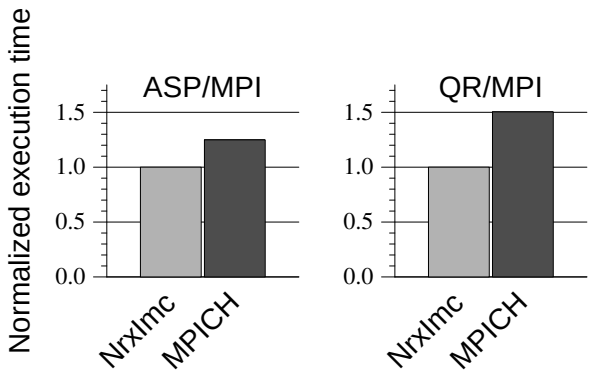


Fig. 8.7. Application-level impact of efficient broadcast support.

primitives. This default implementation can be replaced with a more efficient 'native' implementation. All MPI measurements in Section 8.5 were performed with a native implementation that uses Panda's broadcast which, in turn, uses one of our implementations of LFC's broadcast. In Section 7.4, we used microbenchmarks to show that this native implementation is more efficient than MPICH's default implementation. Figure 8.7 shows that this difference is also visible at the application level. The figure shows the relative performance of two broadcasting MPI applications, ASP and QR, which we will discuss in more detail in Section 8.5. The MPICH-default measurements were performed using unicast primitives on top of the default LFC implementation ($N_{rx}I_{mc}$). For both ASP and QR, this default broadcast is significantly slower than the implementation that uses the broadcast of $N_{rx}I_{mc}$: ASP runs 25% slower and QR runs 50% slower.

MPICH's default broadcast implementation suffers from two problems. First, MPICH forwards entire messages rather than individual packets. For large messages, this eliminates pipelining and reduces throughput. ASP pipelines multiple broadcast messages, each of which consists of multiple packets. In QR, there is no such pipelining of messages; only the packets within a single message can be pipelined. Second, the default implementation cannot reuse NI packet buffers. At each internal node of the multicast tree, the message will be copied to the network interface once for each child. The LFC implementations reuse NI packet buffers to avoid this repeated copying.

Orca

Orca programs can perform two types of communication patterns: RPC and totally-ordered broadcast. In their communication behavior, Orca programs resemble message-passing programs, except in that there is no asynchronous one-way message send primitive. Such a primitive can be simulated by means of multithread-

ing, but none of our applications does this. (Our Orca applications are dominated by multicast traffic.) In Orca, messages carry the parameters and results of user-defined operations, so programmers have control over message size and message rate.

Multigame

Multigame (MG) is a parallel game-playing system developed by Romein [122]. Given the rules of a board game and a board evaluation function, Multigame automatically searches for good moves, using one of several search strategies (e.g., IDA*). During a search, processors push search jobs to each other. A job consists of (recursively) evaluating a board position. To avoid re-searching positions, positions are cached in a distributed hash table. When a process needs to read a table entry, it sends the job to the owner of the table entry [122]. The owner looks up the entry and continues to work on the job until it needs to access a remote table entry.

Multigame's job descriptors are small (32 bytes). To avoid the overhead of sending and receiving many small messages, Multigame aggregates job messages. The communication granularity depends on the maximum number of jobs per message.

Multigame runs on Panda-ST and receives all messages through polling. Almost all communication consists of small to medium-size one-way messages, which are sent to random destinations. The maximum message size depends on the message aggregation limit. The message rate also depends on this limit and, additionally, on the application-specific cost of evaluating a board position. Senders need not wait for replies, so as long as each process has a sufficient amount of work to do, latency (in the LogP sense) is unimportant. Communication overhead is dominated by send and receive overhead.

8.4.2 Performance Issues

Table 7.2 summarizes the minimum latency and the maximum throughput of characteristic operations for Orca, CRL, and MPI. Rules in Multigame specifications are not easily tied to communication patterns. The most important pattern, however, consists of sending a number of jobs in a single message to a remote processor. All communication consists of Panda-ST point-to-point messages.

Table 7.2 shows that our PPSs add significant overhead to an LFC-level implementation of the same communication pattern, so we expect that these overheads will reduce the application-level performance differences between different LFC implementations.

Client-level optimizations form another dampening factor. Two optimizations are worth mentioning: message combining and latency hiding.

As described above, Multigame performs message combining by aggregating search jobs in per-processor buffers. Instead of sending out a search job as soon as it has been generated, the Multigame runtime system stores jobs in an aggregation queue for the destination processor. The contents of this queue is not transmitted until a sufficient number of jobs has been placed into the queue. The difference in packet rate between Puzzle-4 and Puzzle-64 in Figure 8.8 shows that message combining significantly reduces the packet rate for all protocols. Of course, the resulting performance gain (see Figure 8.9) is larger for protocols with higher per-packet costs.

CRL and Orca implement latency hiding. Both systems perform RPC-style transactions: a processor sends a request to another machine and waits for the reply. Both systems then start polling the network and process incoming packets while waiting for the reply. Consequently, protocols can compensate high latencies by processing other packets in their idle periods. (This holds only if the increased latency is not the result of increased host processor overhead.)

8.5 Application Performance

This section analyzes the performance of several applications on all five LFC implementations. Section 8.5.1 summarizes the performance results. Sections 8.5.2 to 8.5.10 discuss and analyze the performance of individual applications.

8.5.1 Performance Results

Table 8.6 lists, for each application, the PPS it runs on, its input parameters, sequential execution time, speedup, and parallel efficiency. (Parallel efficiency is defined as speedup divided by the number of processors: $E_{64} = S_{64}/64$.) The sequential execution time and parallel efficiency were measured using the default implementation ($N_{rx}I_{mc}$). Sequential execution times range from a few seconds to several minutes; speedups on 64 processors range from poor (Radix) to superlinear (ASP). While we use small problems that easily fit into the memories of 64 processors, 7 out of 10 applications achieve an efficiency of at least 50%. Superlinear speedup occurs because we use fixed-size problems and because 64 processors have more cache memory at their disposal than a single processor.

With the exception of Awari and the Puzzle programs, all programs implement well-known parallel algorithms that are frequently used to benchmark PPSs. All CRL applications (Barnes, FFT, and Radix), for example, are adaptations of programs from the SPLASH-2 suite of parallel benchmark programs [153].

Application	PPS	Problem size	T_1	S_{64}	E_{64}
ASP	MPI	1024 nodes	49.32	74.73	1.17
Awari	Orca	13 stones	448.90	31.81	0.50
Barnes	CRL	16,384 bodies	123.24	23.25	0.36
FFT	CRL	1,048,576 complex floats	4.46	49.56	0.77
LEQ	Orca	1000 equations	610.90	30.12	0.47
Puzzle-4	MG	15-puzzle, $\leq$ 4 jobs/message	281.00	45.32	0.71
Puzzle-64	MG	15-puzzle, $\leq$ 64 jobs/message	281.00	53.63	0.84
QR	MPI	1024×1024 doubles	54.16	45.51	0.71
Radix	CRL	3,000,000 ints, radix 128	3.20	10.32	0.16
SOR	MPI	1536×1536 doubles	30.91	51.52	0.80

Table 8.6. Application characteristics and timings. T_1 is the execution time (in seconds) on one processor; S_{64} is the speedup on 64 processors; E_{64} is the efficiency on 64 processors.

We performed the application measurements using the following parameter settings: 64 processors (P = 64), a packet size of 1 Kbyte (PKTSZ = 1), 4096 host receive buffers (HRB = 4096), a send window of 8 packet buffers (W = 8), an interrupt delay of 70 μs (INTD = 70), and a timer granularity of 5000 μs (TGRAN = 5000). For the retransmitting protocols ($I_{rx}I_{mc}$, $I_{rx}H_{mc}$, and $H_{rx}H_{mc}$), we use 256 NI send buffers and 386 NI receive buffers (ISB + IRB = 256 + 384 = 640). For the careful protocols ($N_{rx}I_{mc}$ and $N_{rx}H_{mc}$), we use ISB + IRB = 128 + 512 = 640. With these settings, the control program occupies approximately 900 Kbyte of the NI's memory (1 Mbyte). The remaining space is used by the control program's runtime stack.

Communication statistics for all applications are summarized in Figure 8.8, which gives per-processor data and packet rates, broken down according to packet type. The figure distinguishes unicast data packets, multicast data packets, acknowledgements, and synchronization packets. Data and packet rates refer to *incoming* traffic, so a broadcast is counted as many times as the number of destinations (63).

The goal of the figure is to show that the applications exhibit diverse communication patterns. Compare, for example, Barnes, Radix, and SOR. Barnes has high packet rates and low data rates (i.e., packets are small). Radix, in contrast, has both high packet and data rates. SOR has a low packet rate and a high data rate (i.e., packets are large). These rates were all measured using $H_{rx}H_{mc}$; the rates on other implementations are different, of course, but show similar differences between applications.

Figure 8.9 shows application performance of the alternative implementations

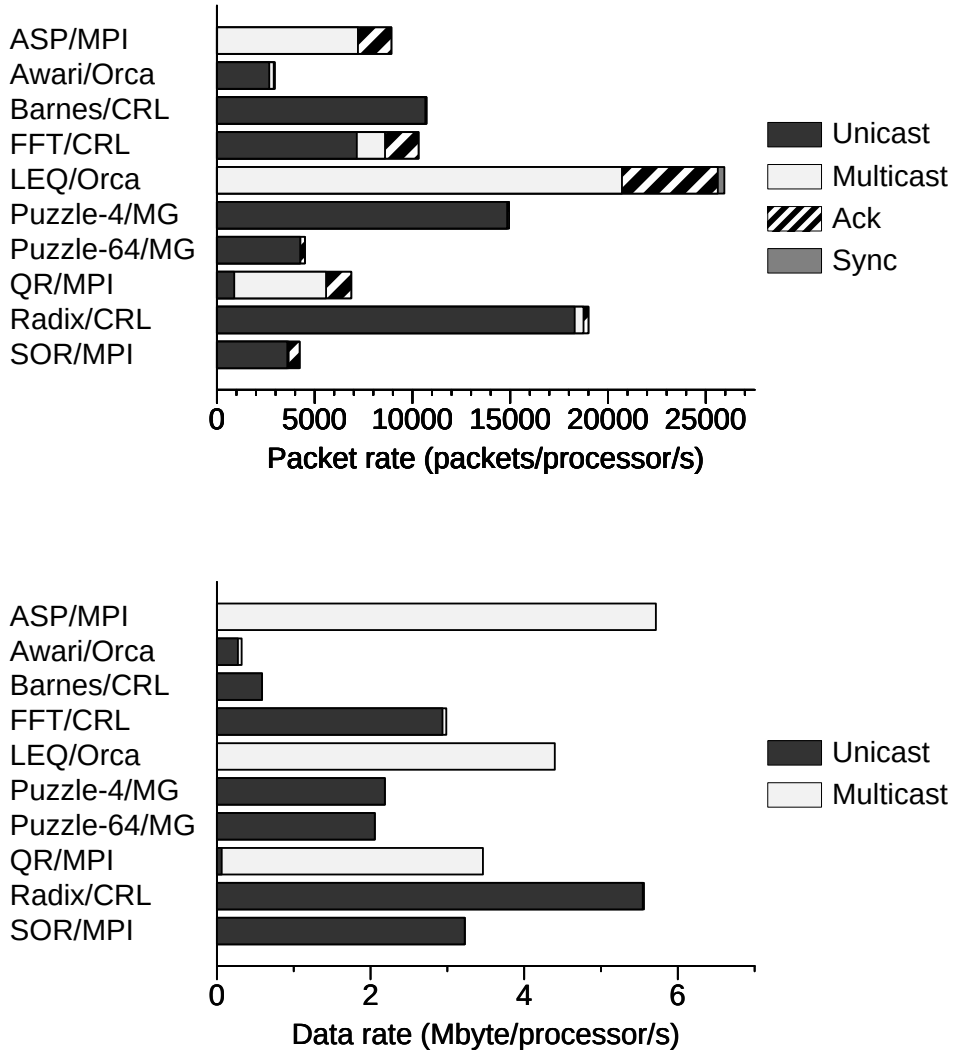


Fig. 8.8. Data and packet rates of $N_{rx}I_{mc}$ on 64 nodes.

relative to the performance of the default implementation ($N_{rx}I_{mc}$). None of the alternative implementations is faster than the default implementation. In several cases, however, the alternatives are slower. Below, these slowdowns are discussed in more detail.

8.5.2 All-pairs Shortest Paths (ASP)

ASP solves the All-pairs Shortest Path (ASP) problem using an iterative algorithm (Floyd-Warshall). ASP finds the shortest path between all nodes in a graph. The graph is represented as a distance matrix which is row-wise distributed across all processors. In each iteration, one processor broadcasts one of its 4 Kbyte rows. The algorithm iterates over all of this processor's rows before switching to another processor's rows. Consequently, the current sender can pipeline the broadcasts of its rows.

Figure 8.9 shows that both protocols that use interface-level multicast forwarding perform better than the protocols that use host-level forwarding. This is surprising, because Figure 8.6 showed that $I_{rx}I_{mc}$ achieves the worst multicast throughput. Indeed, ASP revealed several problems with host-level forwarding that do not show up in microbenchmarks. First, in ASP, it is essential that the sender can pipeline its broadcasts. Receivers, however, are often still working on one iteration when broadcast packets for the next iteration arrive. These packets are not processed until the receivers invoke a receive primitive. (This is a property of our MPI implementation, which uses only polling; see Section 7.4.) Consequently, the sender is stalled, because acknowledgements do not flow back in time. In the interface-level forwarding protocols, acknowledgements are sent by the NI, not by the host processor. As long as the host supplies a sufficiently large number of free host receive buffers, NIs can continue to deliver and forward broadcast packets.

We augmented the H_{mc} versions of ASP with application-level polling statements (*MPI_probe*()), which improved the performance of the host-level protocols. Figure 8.9 shows the improved numbers. Nevertheless, the H_{mc} protocols do not attain the same performance as the I_{mc} protocols. The remaining difference is due to the processor overhead caused by failed polls and host-level forwarding. The effect of forwarding-related processor overhead is visible in the multicast latency benchmark (see Figure 8.5).

8.5.3 Awari

Awari creates an endgame database for Awari, a two-player board game, by means of parallel retrograde analysis [8]. In contrast with top-down search techniques like α-β search, retrograde analysis proceeds bottom-up by making *unmoves*,

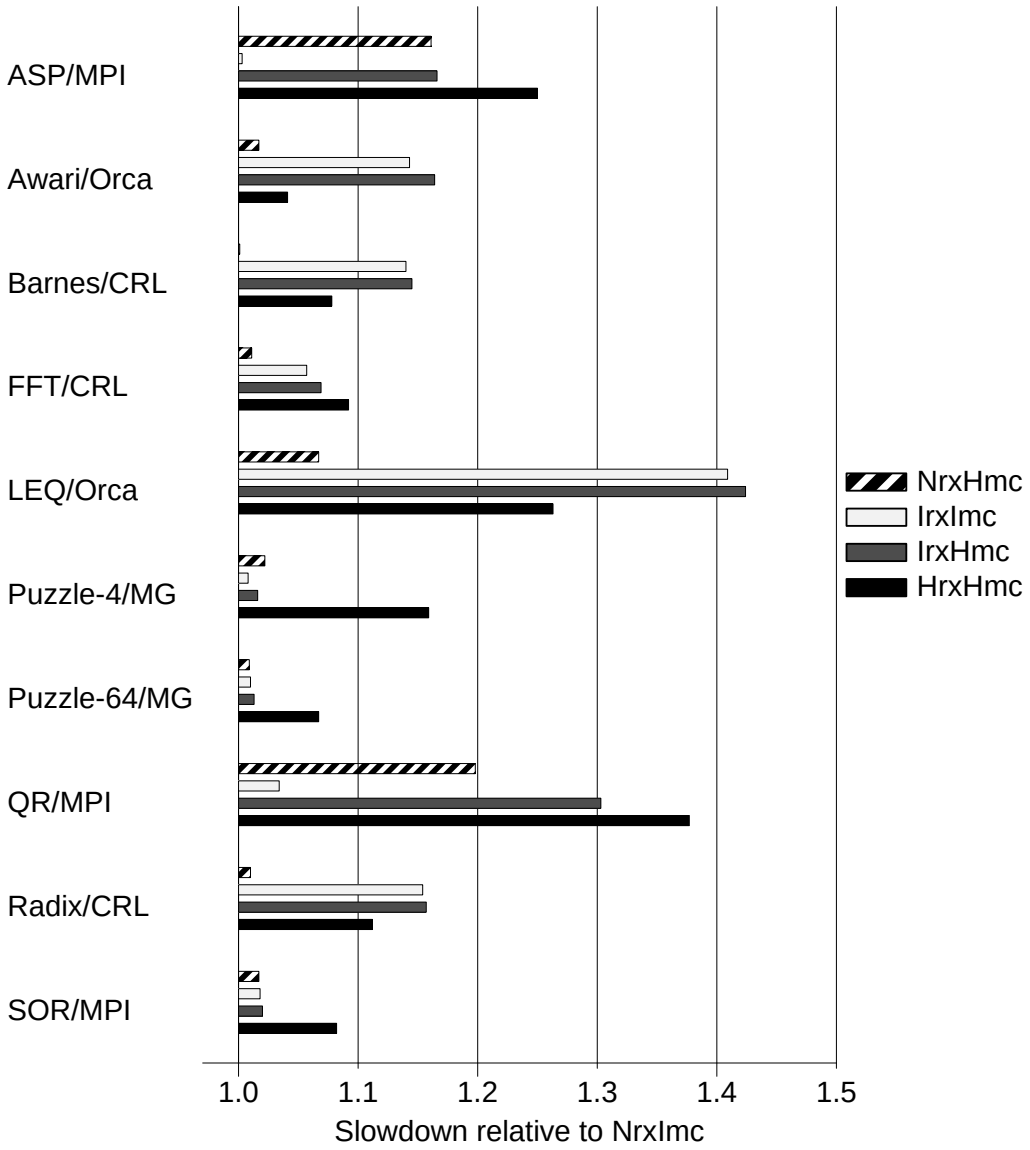


Fig. 8.9. Normalized application execution times.

starting with the end positions of Awari. The Orca program creates an endgame database DB_n, which can be used by a game-playing program. Each entry in the database represents a board position's hash and the board's game-theoretical value. The game-theoretical value represents the outcome of the game in the case that both players make best moves only. DB_n contains game-theoretical values for all boards that have at most n stones left on the board. The game-theoretical value is a number between $-n$ and n that indicates the number of pieces that can be won (or lost). We ran Awari with $n = 13$.

Parallel Awari operates as follows. Each processor stores part of the database. The *parents* of a board are the boards that can be reached by applying a legal unmove to the board. When a processor updates a board's game-theoretical value it must also update the board's parents. Since the boards are randomly distributed across all processors, it is likely that a parent is located on another processor. A single update may thus result in several remote update operations that are executed by RPCs.

To avoid excessive communication overhead, remote updates are delayed and stored in a queue for their destination processor. As soon as a reasonable number of updates has accumulated, they are transferred to their destination with a single RPC. The performance of Awari is determined by these RPCs; broadcasts do not play a major role in this application.

The updates are transferred by a single communication thread, which can run in parallel with the computation thread. Application-level statistics, however, indicate that most updates are transmitted when there is no work for the computation thread. As a result, performance is dominated by the speed at which work can be distributed. Queued updates that must be sent to *different* destination processors cannot be sent out at a rate that is higher than the rate at which the communication thread can perform RPCs.

Although Awari's data rate appears low (see Figure 8.8), communication takes place in specific program phases. In these phases, communication is bursty and most messages are small despite message combining. Performance is therefore dominated by occupancy rather than roundtrip latency. The LogP parameters in Table 8.4 show that I_{rx} has the highest gap ($g = 9.7$ μs), followed by H_{rx} ($g = 8.3$ μs) and then N_{rx} ($g = 6.7$ μs). This ranking is reflected in the application's performance.

8.5.4 Barnes-Hut

Barnes simulates a galaxy using the hierarchical Barnes-Hut N-body algorithm. The program organizes all simulated bodies in a shared oct-tree. Each node in this tree and each body (stored in a leaf) is represented as a small CRL region (88–108 bytes). Each processor owns part of the bodies. Most communication takes place

during the phase that computes the gravitational forces that bodies exert on one another. In this phase, each processor traverses the tree to compute for each of its bodies the interaction with other bodies or subtrees. Barnes has a relatively high packet rate, but due to the small regions the data rate is low (see Figure 8.8).

Even though Barnes runs on a different PPS, its performance profile in Figure 8.9 is similar to that of Awari. Since CRL makes little use of LFC's multicast, there is no large difference between interface-level and host-level forwarding. Retransmission support, however, leads to decreased performance. This effect is strongest for I_{rx}, which has the highest interpacket gap ($g = 9.7\,\mu$s) and is therefore more likely to suffer from NI occupancy under high loads. These high loads occur because Barnes has a high packet rate (see Figure 8.8). Due to CRL's roundtrip communication style (see Section 7.3), Barnes is sensitive to this increase in occupancy.

8.5.5 Fast Fourier Transform (FFT)

FFT performs a Fast Fourier transform on an array of complex numbers. These numbers are stored in a matrix which is partitioned into P^2 square blocks (where P is the number of processors). Each processor owns a vertical stripe of P blocks and each block is stored in a 2 Kbyte CRL region. The main communication phases consist of matrix transposes. During a transpose a personalized all-to-all exchange of all blocks takes place: each processor exchanges each of its blocks (except the block on the diagonal) with a block owned by another processor. Each block transfer is the result of a write miss. The node that generates the write miss sends a small request message to the block's home node, which replies with a data message that contains the block.

As explained in Section 8.2.5, this communication pattern puts pressure on the number of send buffers for the retransmitting implementations. Since we have configured these implementations with a fairly large number of send buffers (ISB = 256), however, this poses no problems.

The performance results in Figure 8.9 show no large differences between different LFC implementations. The retransmitting protocols perform slightly worse than the careful protocols. Occasionally, these protocols send delayed acknowledgements. If we increase the delayed acknowledgement timeout, no delayed acknowledgements are sent, and the differences become even smaller.

8.5.6 The Linear Equation Solver (LEQ)

LEQ is an iterative solver for linear systems of the form $Ax = b$. Each iteration refines a candidate solution vector x_i into a better solution x_{i+1}. This is repeated until the difference between x_{i+1} and x_i becomes smaller than a specified bound.

In each iteration, each processor produces a part of x_{i+1}, but needs all of x_i as its input. Therefore, all processors exchange their 128-byte partial solution vectors at the end of each iteration (6906 total). Each processor broadcasts its part of x_{i+1} to all other processors. After this exchange, all processors synchronize to determine whether convergence has occurred.

In LEQ, $H_{rx}H_{mc}$ suffers from its host-level fetch-and-add implementation. Processing F&A requests on the host rather than on the NI increases the response time (see Table 8.5) and the occupancy of the processor that stores the F&A variable (processor 0).

Although LEQ is dominated by broadcast traffic, the cost of executing a retransmission protocol appears to be the main performance factor: increased host and NI occupancy lead to decreased performance. This is due to LEQ's communication behavior. All processes broadcast at the same time, congesting the network and the NIs. In ASP and QR, processes broadcast one at a time.

8.5.7 Puzzle

Puzzle performs a parallel IDA* search to solve the 15-puzzle, a well-know single-player sliding-tile puzzle. We experimented with two versions of Puzzle: in Puzzle-4, Multigame aggregates at most 4 jobs before pushing these jobs to another processor, while in Puzzle-64 up to 64 jobs are accumulated. Both programs solve the same problem, but Puzzle-4 sends many more messages than Puzzle-64.

In Puzzle, all communication is one-way and processes do not wait for incoming messages. As a result, send and receive overhead are more important than NI-level latency and occupancy. $H_{rx}H_{mc}$ has the worst send and receive overheads of our LFC implementations (see Table 8.4) and therefore performs worse than the other implementations. In Puzzle-4, the difference between $H_{rx}H_{mc}$ and the other protocols is larger than in Puzzle-64, because Puzzle-4 needs to send more messages to transfer the same amount of data. Consequently, the higher send and receive overheads are incurred more often during the program's execution.

8.5.8 QR Factorization

QR is a parallel implementation of QR factorization [59] with column distribution. In each iteration, one column, the *Householder vector* H, is broadcast to all processors, which update their columns using H. The current upper row and H are then deleted from the data set so that the size of H decreases by 1 in each of the 1024 iterations. The vector with maximum norm becomes the Householder vector for the next iteration. This is decided with a Reduce-To-All collective operation to which each processor contributes two integers and two doubles.

The performance characteristics of QR are similar to those of ASP (see Figure 8.9): the implementations based on host-level forwarding perform worse than those based on interface-level forwarding. The performance of QR is dominated by the broadcasts of column H. As in ASP, host-level forwarding leads to increased processor overhead. In addition, the implementations based on host-level forwarding are sensitive to their higher broadcast latency. At the start of each iteration, each receiving processor must wait for an incoming broadcast. In contrast with ASP, the broadcasting processor cannot get ahead of the receivers by pipelining multiple broadcasts, because the Reduce-to-All synchronizes all processors in each iteration. (Also, due to pivoting, the identity of the broadcasting processor changes in almost every iteration.)

8.5.9 Radix Sort

Radix sorts a large array of (random) integers using a parallel version of the radix sort algorithm. Each processor owns a contiguous part of the array and each part is further subdivided into CRL regions, which act as software cache lines. Communication is dominated by the algorithm's permutation phase, in which each processor moves integers in its partition to some other partition of the array. If the keys are distributed uniformly, each processor accesses all other processors' partitions. We chose a region size (1 Kbyte) that minimizes write sharing in this phase. After the permutation phase each processor reads the new values in its partition and starts the next sorting iteration. Radix has the highest unicast data rate of all our applications (see Figure 8.8). In each of the three permutation phases, most of the array to be sorted is transferred across the network.

Although Radix sends larger messages than Awari, Barnes, and LEQ, it has a similar performance profile as these applications. Radix's messages are still relatively small (at most 1 Kbyte of region data) and each message requires at most two LFC packets, a full packet and a nearly empty packet. Consequently, Radix remains sensitive to the small-message bottleneck, so that NI-level retransmission ($I_{rx}I_{mc}$ and $I_{rx}H_{mc}$) performs worst, followed by host-level retransmission ($H_{rx}H_{mc}$).

8.5.10 Successive Overrelaxation (SOR)

SOR is used to solve discretized Laplace equations. The SOR program uses red-black iteration to update a 1536×1536 matrix. Each processor owns an equal number of contiguous rows of the matrix. In each of the 42 iterations, processors exchange neighboring border rows (12 Kbyte) and perform a single-element (one double) reduction to determine whether convergence has occurred.

Class	Applications	$N_{rx}I_{mc}$	$N_{rx}H_{mc}$	$I_{rx}I_{mc}$	$I_{rx}H_{mc}$	$H_{rx}H_{mc}$
Roundtrip	Barnes, Radix, Awari	+	+	−−	−−	−
Multicast	ASP, QR	+	−	+	−	−−
One-way	Puzzle-4, Puzzle-64	+	+	+	+	−

Table 8.7. Classification of applications.

For SOR, $H_{rx}H_{mc}$ performs worse than the other implementations. With the $H_{rx}H_{mc}$ implementation, the host processor suffers from sliding-window stalls. Each 12 Kbyte message that is sent to a neighbor requires 13 LFC packets, while the window size is only 8 packets. If the destination process is actively polling when the sender's packets arrive, it will send a half-window acknowledgement before the sender is stalled. In SOR, however, all processes send out their messages at approximately the same time, first to their higher-numbered neighbor, then to their lower-numbered neighbor. The destination process will therefore not poll until one of its own sends blocks (due to a closed send window). At that point, the sender has already been stalled.

The implementations that implement NI-level reliability do not suffer from this problem, because they implement the sliding window on the NI. Even if the NI's send window to another NI closes, the host processor can continue to post packets until the supply of free send descriptors is exhausted. Also, if the NI's send window to one NI fills up, it can still send packets to other NIs if the host supplies packets for other destinations.

The performance of $H_{rx}H_{mc}$ improves significantly if the boundary row exchange code is modified so that not all processes send in the same direction at the same time. In practice, this improves the probability that acknowledgements can be piggybacked and reduces the number of window stalls.

8.6 Classification

Based on the performance analysis of the individual applications, we have identified three classes of applications with distinct behavior. These classes and an indication of the relative performance of each LFC implementation for each class are shown in Table 8.7. In this table, a '+' indicates that an implementation performs well for a class of applications; a '−' indicates a modest decrease in performance; and '−−' indicates a significant decrease in performance.

The performance of *roundtrip* applications is dominated by roundtrip latency. Since multicast plays no important role in these applications, performance differences between the LFC implementations are determined by differences in the reliability schemes. This class of applications shows that the robustness of re-

transmission has a price. Both the host-level retransmission schemes and the interface-level retransmission schemes perform worse than the no-retransmission scheme, but for different reasons. Host-level retransmission ($H_{rx}H_{mc}$) suffers from its higher send and receive overhead and is up to 11% slower than the default implementation ($N_{rx}I_{mc}$). NI-level retransmission ($I_{rx}I_{mc}$ and $I_{rx}H_{mc}$) suffers from increased (NI) occupancy and latency. These implementations are up to 16% slower than the default implementation. For this class of applications, the LogP measurements presented earlier in this chapter give a good indication of the causes of differences in application-level performance.

The performance of the *multicast* applications is determined by the efficiency of the multicast implementation. The performance results show the advantages of NI-level multicast forwarding. $H_{rx}H_{mc}$, $I_{rx}H_{mc}$, and $N_{rx}H_{mc}$ all suffer from host-level forwarding overhead which takes away time from the application. Host-level forwarding is up to 38% slower than interface-level forwarding in the default implementation.

The 'class' of *one-way* applications contains both variants of the Puzzle application. In contrast with roundtrip applications, one-way applications do not wait for reply messages, so latency and NI occupancy can be tolerated fairly well. Send and receive overhead, on the other hand, cannot be hidden. As a result, $H_{rx}H_{mc}$ suffers from its increased send and receive overhead; this leads to a performance loss of up to 16% relative to the default implementation.

The remaining applications (FFT, LEQ, and SOR) do not fit into any of these categories.

8.7 Related Work

This chapter has concentrated on NI protocol issues and their relationship with properties of PPSs and applications. Figure 8.10 classifies several existing Myrinet communication systems along two protocol design axes: reliability and multicast. Most systems do not implement retransmission. We assume that this is partly due to the prototype nature of research systems. The figure also shows that few systems provide multicast or broadcast support. Below, we first discuss related work in these two areas. Next, we discuss other NI-related application studies.

8.7.1 Reliability

Only a few studies have compared careful and retransmitting protocols. The earliest study that we are aware of is by Mosberger and Peterson [106], which compares a careful and a retransmitting protocol implementation for FiberChannel. They note the scalability problems of static buffer reservation (as in N_{rx}). This

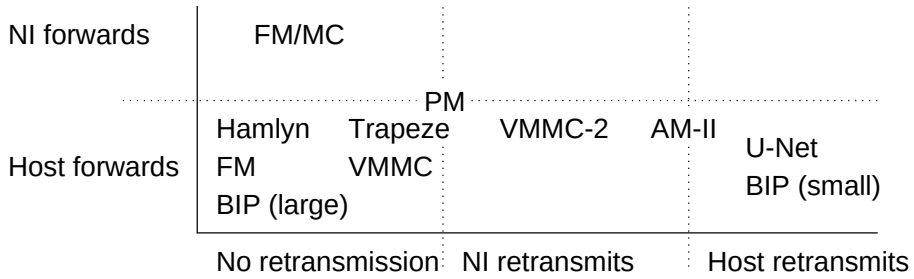


Fig. 8.10. Classification of communication systems for Myrinet based on the multicast and reliability design issues.

problem exists, but the low bandwidth-delay product of modern networks (and protocols) allows the use of static reservation in medium-size clusters. (Another, dynamic solution, used in PM's NI protocol, is described below.) Mosberger and Peterson used only one application, Jacobi. That application shows much larger benefits for careful protocols than we find with our application suite. This may be due to extra copying in their retransmitting protocol. As explained in Section 8.2.2, our retransmitting protocols do not make more copies than our careful protocols.

Some of the protocols in Figure 8.10 combine the strategies used by N_{rx}, H_{rx}, and I_{rx}. Like N_{rx}, for example, PM assumes that the hardware never drops or corrupts packets. Like I_{rx}, though, PM lets senders share NI receive buffers and drops incoming *data* packets when all buffers are full. When a data packet is dropped, a negative acknowledgement is sent to its sender. PM never drops acknowledgements (negative or positive). Since it is assumed that the hardware delivers all packets correctly, senders always know when one of their packets has been dropped and can retransmit that packet without having to set a retransmission timer.

Active Messages II combines an NI-level (alternating-bit) reliability protocol with a host-level sliding window protocol which is used both for reliability and flow control [37].

Our NI-supported fine-grain timer implementation is similar to the *soft timer* scheme described by Aron and Druschel [6] —developed independently— who also use polling to implement a fine-grain timer. They optimize kernel-level timers and poll the host's timestamp counter on system calls, exceptions, and interrupts to amortize the state-saving overhead of these events over multiple actions (e.g., network interrupt processing and timer processing). Soft timers use the kernel's clock interrupt as a backup mechanism. This interrupt often has a granularity of at least several milliseconds. H_{rx} uses the NI's timer as a backup. This timer has a 0.5 μs granularity and is polled in each iteration of the NI's main loop. Our backup

mechanism is therefore more precise and can generate an interrupt at a time close to the desired timer expiration time in case the host's timestamp counter is not polled frequently enough. This is only a modest advantage, because it does not address the interrupt overhead problem that can result from infrequent polling of the timestamp counter.

8.7.2 Multicast

To the best of our knowledge, this is the first study that compares a high-performance NI-level and high-performance host-level multicast *and* their respective impact on application performance. Several papers describe multicast protocols that use NI-level multicast forwarding [16, 56, 68, 80, 146]. This work was described in Section 4.5.3. Some of these papers (e.g., [146]) compare host-level forwarding and NI-level forwarding, but most of these comparisons use a host-level forwarding scheme that copies data to the NI multiple times. We are not aware of any previous study that studies the *application*-level impact of both forwarding strategies.

8.7.3 NI Protocol Studies

Araki et al. used microbenchmarks and the LogP performance model to compare the performance of Generic Active Messages, Illinois Fast Messages (version 2), BIP, PM, and VMMC-2 [5]. Their study compares communication systems with programming interfaces that are sometimes fundamentally different (e.g., memory-mapped communication in VMMC-2 versus message-based communication in PM and rendezvous-style communication in BIP versus asynchronous message passing in Fast Messages). In our study, we compare five different implementations of the same interface. The most important difference with the work described in this chapter, however, is that Araki et al. do not consider the application-level impact of their results.

In another study [102], Martin et al. do focus on applications, but evaluate only a single communication architecture (Active Messages) and a single programming system (Split-C). Where Martin et al. vary the performance characteristics (the LogGP parameters) of a single communication system, we use five different systems which have different performance characteristics due to the way they divide protocol work between the host processor and the NI. An important contribution of our work is the evaluation of different network interface protocols using a wide variety of parallel applications (fine-grain to medium-grain, unicast and multicast) and PPSs (distributed search, message passing, update-based DSM, and invalidation-based DSM). Each system has its own, fairly large and complex runtime system, which imposes significant communication overheads

(see Table 7.2). In addition, three out of four systems do not run immediately on top of one NI protocol layer (LFC), but on an intermediate message-passing layer (Panda).

Bilas et al. discuss NI support for shared virtual memory systems (SVMs) [21]. They study NI support for fetching data from remote memories, for depositing data into remote memories, and for distributed locks. They find that these mechanisms significantly improve the performance of SVM implementations and claim that they are more widely applicable. Using these mechanisms, and by restructuring their SVM, they eliminated all asynchronous host-level protocol processing in their SVM. As a result, interrupts are not needed and polling is used only when a message is expected. This result, however, depends on the ability to push asynchronous protocol processing to the NI. Bilas et al. use a PPS in which asynchronous processing consists of accessing data at a known address or accessing a lock. These actions are relatively simple and can be performed by the NI. PPSs such as Orca and Manta, however, must execute incoming user-defined operations, which cannot easily be handled by the NI.

In a simulation study, Bilas et al. identify bottlenecks in software shared-memory systems [20]. This study is structured around the same three layers as we used in this chapter: low-level communication software and hardware, PPS, and application. Both the communication layer and the PPSs are different from the ones studied in this thesis. Bilas et al. assume a communication layer based on virtual memory-mapped communication (see Chapter 2) and analyze the performance of page-based and fine-grained DSMs. Our work uses a communication layer based on low-level message passing and PPSs that implement message passing or object-based sharing. (CRL uses a similar cache coherence protocol as fine-grained DSMs, but relies on the programmer to define 'cache lines' and to trigger coherence actions. Fine-grained DSMs such as Tempest [119] and Shasta [125] require little or no programmer intervention.)

8.8 Summary

In this chapter, we have studied the performance of five implementations of LFC's communication interface. Each implementation is a combination of one reliability scheme (no retransmission, host-level retransmission, or NI-level retransmission) and one multicast forwarding scheme (host-level forwarding or NI-level forwarding). These five implementations represent different assumptions about the capabilities of network interfaces (availability of a programmable NI processor and its speed) and the operating environment (reliability of the network hardware). We compared the performance of all five implementations at multiple levels. We used microbenchmarks for direct comparisons between the different implementations

and for comparisons at the PPS level. We used four different PPSs, which represent different programming paradigms and which exhibit different communication patterns. Most importantly, we also performed an application-level comparison.

These performance comparisons reveal several interesting facts. First, although performance differences are visible, all five LFC implementations can be made to perform well on most LFC-level microbenchmarks. Multicast latency forms an exception: NI-level multicast forwarding yields better multicast latency than host-level forwarding.

Second, all PPSs add large overheads to LFC implementations. Measuring essentially the same communication pattern at multiple levels shows large increases in latency and large reductions in throughput. While this is a logical consequence of the layered structure of the PPSs, it is important to make this observation, because one would expect that these overheads, and not the relative small performance differences between LFC implementations, will dominate application performance.

Third, in spite of the previous observation, running applications on the different LFC implementations yields significant differences in application execution time. Most of our applications exhibited one of three communication patterns: roundtrip, one-way, or multicast. For each of these patterns, the relative performance of the LFC implementations is similar:

- The roundtrip applications perform best on the nonretransmitting LFC implementations. Retransmission support adds sufficient overhead that it cannot be hidden by applications in this class. The overheads have a larger impact when retransmission is implemented on the NI.

- The multicast applications perform best on the implementations that perform NI-level forwarding. NI-level forwarding yields lower multicast latency and does not waste host-processor cycles on packet forwarding.

- The one-way applications —really two variants of the same applications— perform well on all implementations, except on the implementations that implement host-level retransmission. This implementation suffers from its larger send and receive overheads.

Our original implementation of LFC, $N_{rx}I_{mc}$, performs best for all patterns. This implementation, however, optimistically assumes the presence of reliable network hardware and an intelligent NI.

Chapter 9

Summary and Conclusions

The implementation of a high-level programming model consists of multiple layers: a runtime system or parallel-programming system that implements the programming model and one or more communication layers that have no knowledge of the parallel-programming model, but that provide efficient communication mechanisms. This thesis has concentrated on the lower communication layers, in particular on the network interface (NI) protocol layer. A key contribution of this thesis, however, is that it also addresses the interactions of the NI protocol layer with higher-level communication layers and applications. The following sections summarize our work as it relates to each layer and draw conclusions.

9.1 LFC and NI Protocols

We have implemented our ideas in and on top of LFC, a new user-level communication system. In several ways, LFC resembles a hardware device: communication is packet-based and all packets are delivered to a single upcall. However, LFC adds two services that are usually not provided by network hardware: reliable point-to-point and multicast communication. The efficient implementation of both services has been the key to LFC's success.

Efficient communication is of obvious importance to parallel-programming systems (PPSs). While many communication systems provide low-latency, high-throughput point-to-point primitives, very few systems provide efficient multicast implementations. In fact, many do not provide multicast at all. This is unfortunate, because PPSs such as Orca and MPI rely on multicasting to update replicated objects and to implement collective-communication operations, respectively. Moreover, multicast is not an add-on feature: multicasts layered on top of point-to-point primitives perform considerably worse than multicasts that are supported by the bottom-most communication layer (see Chapters 7 and 8).

The presence of *reliable* communication greatly reduces the effort needed to implement a PPS. Compared to fragmentation and demultiplexing —services not provided by LFC— reliability protocols are difficult to implement correctly and efficiently. Nevertheless, all PPSs must deliver data reliably.

For efficiency, the default implementation of LFC uses NI support. The NI performs four tasks: flow control for reliability, multicast forwarding, interrupt management, and fetch-and-add processing. The following paragraphs discuss these tasks in more detail.

The default implementation assumes reliable network hardware and implements reliable communication channels by means of a simple, NI-level flow control protocol. This protocol, UCAST, is also the basis of MCAST and RECOV, LFC's NI-supported multicast protocols. Since these protocols assume reliable hardware, however, they can operate only in a controlled environment such as a cluster of computers dedicated to running high-performance parallel jobs.

MCAST is simple and efficient, but does not work for all multicast tree topologies. RECOV works for all topologies, but is more complex and requires additional NI buffer space. Both MCAST and RECOV perform fewer data transfers than host-level store-and-forward multicast protocols, which reduces the number of host cycles spent on multicast processing. The performance measurements in Chapter 8 show that host-level multicasting reduces application performance by up to 38%. These measurements compared the performance of an NI-supported multicast to an *agressive* host-level multicast. In practice, host-level multicast implementations are frequently implemented on top of unicast primitives, which further increases the number of redundant data transfers.

Network interrupts form an important source of overheads in communication systems. To reduce the number of unnecessary interrupts, LFC implements a software polling watchdog on the NI. This mechanism delays network interrupts for incoming packets for a certain amount of time. In addition, LFC's polling watchdog monitors host-level packet processing progress to determine whether a network interrupt should be generated.

We have implemented an NI-supported fetch-and-add operation. Panda combines LFC's fetch-and-add with LFC's broadcast to implement a totally-ordered broadcast, which, in turn, is used by Orca. Fetch-and-add also has other applications. Karamcheti and Chien, for example, have used fetch-and-add to implement pull-based messaging [76].

Other types of synchronization primitives can als benefit from NI support. Bilas et al. use the NI to implement distributed locking in a shared virtual memory system [21]. These examples indicate that NI support for synchronization primitives is a good idea. Without such support, synchronization requests generate expensive interrupts. It is not clear yet, however, *which* primitives exactly must be supported. Different systems require different primitives and, in spite of the

generality claimed by the implementors of some mechanisms [21], it is unlikely that one primitive will satisfy every system. Another problem is the virtualization of NI-supported primitives. LFC, for example, supports only one fetch-and-add variable per NI. This constraint was introduced to bound the space occupied by NI variables and to avoid introducing an extra demultiplexing step. Ideally, this type of constraint would be hidden from users. Active Messages II virtualizes NI-level communication endpoints by using host memory as backing storage for NI memory [37]. This strategy is general, but increases the complexity and decreases the efficiency of the implementation.

It is frequently claimed that programmable NIs are 'too slow'; such claims are often accompanied with references to the failure of I/O channels. LFC performs four different tasks on the NI: flow control for reliability, multicast forwarding, interrupt management, and fetch-and-add processing. Nevertheless, LFC is an efficient communication system. The main issue is not whether NIs should be programmable, but which mechanisms the lowest communication layer should supply and where they should be implemented. Programmable NIs can be considered one tool in a spectrum of tools that help answer this type of questions; others include formal analysis and simulation. In this thesis, we studied reliability, multicast, synchronization and interrupt management. Others have investigated NI support for address translation in zero-copy protocols [13, 49, 142], distributed locking [21], and remote memory access [51, 83]). Some of these mechanisms (e.g., remote memory access) have recently found their way into industry standards (the Virtual Interface Architecture) and commercial products.

9.2 Panda

Many parallel-programming systems are sufficiently complex that they can benefit from an intermediate communication layer that provides higher-level communication services than a system like LFC. Panda provides threads, messages, message passing, RPC, and group communication. These services are used in the implementation of three of the four PPSs described in Chapter 7.

Panda's thread package, OpenThreads, dynamically switches between polling and interrupts. By default, interrupts are enabled, but when all threads are idle, OpenThreads disables interrupts and polls the network. This strategy is simple but effective. When a message is expected, it will often be received through polling, which is more efficient than receiving it by means of an interrupt. Unexpected messages generate an interrupt, unless the receiving process polls before LFC's polling watchdog generates the interrupt.

Panda's stream messages allow Panda clients to transmit and receive a message in a pipelined manner: the receiver can start processing an incoming mes-

sage before it has been fully received. Stream messages are implemented on top of LFC's packet-based interface without introducing extra data copies.

Blocking in message handlers is a difficult problem. Creating a (popup) thread per message allows message handlers to block whenever convenient, but it also removes the ordering between messages, introduces scheduling anomalies, and wastes (stack) space. Disallowing all handlers to block (as in active messages) forces programmers to use asynchronous primitives and continuations whenever any form of blocking (e.g., acquiring a lock) is required. Panda uses single-threaded upcalls to process incoming messages. Message handlers are allowed to block on locks, but cannot wait for the arrival of other messages. If the receiver of a message can decide early that waiting for another message will not be necessary, then the message can be processed without creating a new thread. In other cases, blocking can be avoided by using nonblocking primitives instead of blocking primitives or by using continuations.

Panda and LFC work well together. Panda's thread scheduler uses LFC's interrupt support to switch dynamically between polling and interrupts. Stream messages pipeline the transmission and consumption of LFC packets. Finally, Panda's totally-ordered broadcast combines LFC's fetch-and-add and broadcast primitives.

9.3 Parallel-Programming Systems

Different PPSs have different communication styles and are therefore sensitive to different aspects of the underlying communication system. This is illustrated by the four PPSs studied in Chapter 7.

Orca uses two communication primitives to implement operations on shared objects: remote procedure call and asynchronous, totally-ordered broadcast. The performance of the Orca applications that we studied in Chapter 8 depends on one of the two. Orca operations may block on guards. Since guards occur only at the beginning of an operation, it is easy to suspend the operation without blocking. Without blocking and thread switching, the Orca runtime system creates a small continuation that represents the blocked operation.

Like Orca, the Java-based Manta system provides multithreading and shared objects. Since Manta does not replicate objects, it requires only RPC to implement operations on shared objects. In contrast with Orca, Manta operations can block halfway through. This makes it difficult for the RTS to use small continuations to represent the blocked operation. The RTS therefore creates a popup thread for each operation, unless the compiler can prove that the operation will never block. Another complication in Manta is the presence of a garbage collector. The space occupied by unmarshaled parameters usually cannot be reused until

the parameters have been garbage collected. Consequently, unmarshaling leaves a large memory footprint and pollutes large parts of the data cache.

In CRL, most communication patterns are roundtrips. When invalidations are needed, nested RPCs occur. On average, CRL's RPCs are smaller than those of Manta and Orca, because CRL frequently sends small control messages (region requests, invalidations, and acknowledgements). Manta and Orca generate some control traffic, but far less than CRL.

MPI has the simplest programming model of the four PPSs studied in this thesis. In most cases, communication at the MPI level corresponds in a straightforward way with communication at lower levels. MPI's collective-communication operations form an important exception. These high-level primitives are usually implemented by a combination of point-to-point and multicast messages.

While these PPSs generate different communication patterns, they often rely on the same communication mechanisms. All PPSs require reliable communication. Several low-level communication systems (e.g., U-Net and BIP) do not provide reliable communication primitives.

Orca, Manta, and CRL all benefit from LFC's and Panda's mechanisms to reduce the number of network interrupts. Orca and Manta rely on Panda's thread package, OpenThreads, to poll when all threads are idle. In, CRL the same optimization has been hardcoded.

Orca and MPI benefit from an efficient multicast implementation. Orca's totally-ordered broadcast is layered (in Panda) on top of LFC's broadcast and fetch-and-add. MPI's collective-communication operations use (through Panda) LFC's broadcast.

Orca, Manta, and MPI all use Panda's stream messages. Stream messages provide a convenient and efficient message interface without introducing intermediate data copies. The Orca and Manta compilers generate operation-specific code to marshal to and from stream messages.

9.4 Performance Impact of Design Decisions

Chapter 8 studied the impact of different reliability and multicast protocols on application performance and showed that the LFC implementation described in Chapters 3 and 4 ($N_{rx}I_{mc}$) performs better than alternative implementations. This implementation, however, is agressive in two ways. First, it assumes reliable hardware, which means it can be used only in dedicated environments. Second, it delegates some tasks (flow control, multicast, fetch-and-add, and interrupt management) to the NI. These tasks are performed by the NI-level protocols described in Chapter 4. Additional robustness can be obtained by means of retransmission. We investigated both host-level and NI-level retransmission. The unicast-dominated

applications of Chapter 8 show that both types of retransmission have a modest application-level cost. The largest overhead (up to 16%) is incurred by applications in the 'roundtrip' class. The performance cost of moving NI-level components to the host can be large. Performing multicast forwarding on the host rather than on the NI slows down applications by up to 38%.

Chapter 8 also revealed interesting interactions between the structure of applications and the implementation of the underlying communication system(s). In SOR, the exact location (host or NI) of the sliding window protocol had a noticeable impact on performance. In ASP, we added polling statements to reduce the delays that resulted from synchronous host-level multicast forwarding. In some cases, applications can be restructured to work around the weaknesses of a particular communication system. The Puzzle application, for example, can be configured to use a larger aggregation buffer on top of a host-level reliability protocol to reduce the impact of increased send and receive overhead. The performance of SOR was improved by changing the boundary exchange phase.

9.5 Conclusions

Based on our experiences with LFC and the systems layered on top of LFC, we draw the following conclusions:

1. Low-level communication systems should support both polling and interrupt-driven message delivery. We studied four PPSs. All of them use polling and three of them use interrupts (Orca, Manta, and CRL). Transparent emulation of interrupt-driven message delivery is nontrivial —several systems use binary rewriting— or relies on interrupts as a backup mechanism.

2. Low-level communication systems should support multicast. Multicast plays an important role in two of the four PPSs studied (Orca and MPI). Multicast implementations are often layered on top of point-to-point primitives. This always introduces unnecessary data transfers that consume cycles that could have been used by the application. Moreover, if multicast forwarding is implemented at the message rather than the packet level, then this strategy also reduces throughput by removing intramessage pipelining.

3. A variety of PPSs can be implemented efficiently on a simple communication system *if* all layers cooperate to avoid unnecessary interrupts, thread switches, and data copies. LFC provides a simple interface: reliable, packet-based point-to-point and multicast communication and interrupt management. These mechanisms are used to implement abstractions that preserve efficiency. In Panda, they are used to implement OpenThreads's automatic

switching between polling and interrupts, message streams, and totally-ordered multicast. Orca, Manta, and MPI are all built on top of Panda. In addition, Orca and Manta use operation-specific marshaling in combination with stream messages to avoid making extra data copies.

4. An aggressive, NI-supported communication system can yield better performance than a traditional communication system without NI support. We have experimented with a reliability protocol (UCAST), two multicast protocols (MCAST and RECOV), a polling watchdog (INTR), and a fetch-and-add primitive. Additional robustness and simplicity can be obtained by systems that implement all functionality on the host, but these systems perform worse, both at the level of microbenchmarks and at the level of applications.

9.6 Limitations and Open Issues

Although this thesis has a broad scope, several important issues have not been addressed and some issues that have been addressed require further research.

First, LFC lacks zero-copy support. Zero-copy mechanisms are based on DMA rather than PIO and, for sufficiently large transfers, give better throughput than PIO (see Chapter 2). Sender-side zero-copy support is relatively easy to implement and use. For optimal performance, however, data should also be moved directly to its final destination at the receiver side. Some systems (e.g., Hamlyn [32]) achieve this by having the sender specify a destination address in the receiver's address space. PPSs need to be restructured in nontrivial ways to determine this address without introducing extra messages [21, 35]. While zero-copy communication support for PPSs warrants further investigation, we are not aware of any study that shows significant *application-level* performance advantages of zero-copy communication mechanisms over message-based mechanisms. There *are* studies that show that parallel applications are relatively insensitive to throughput [102].

Second, more consideration needs to be given to different buffer and timer configurations for the various LFC implementations studied in Chapter 8. We selected buffer and timer settings that yield good performance for each implementation. It is not completely clear how sensitive the different implementations are to different settings.

Finally, the evaluation in Chapter 8 focused on the impact of alternative reliability and multicast schemes. Additional work is needed to determine the impact of NI-supported synchronization and interrupt delivery (i.e., the polling watchdog). Some of our results (not presented in this thesis), however, show a good qualitative match with those presented by Maquelin et al. [101].

Bibliography

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. MacKenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proc. of the 22nd Int. Symp. on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 1995.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[3] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – One Step Closer Towards a Realistic Model for Parallel Computation. In *Proc. of the 1995 Symp. on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, CA, July 1995.

[4] K.V. Anjan and T.M. Pinkston. An Efficient, Fully Adaptive Deadlock Recovery Scheme: DISHA. In *Proc. of the 22nd Int. Symp. on Computer Architecture*, pages 201–210, Santa Margherita Ligure, Italy, June 1995.

[5] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-Space Communication: A Quantitative Study. In *Supercomputing'98*, Orlando, FL, November 1998.

[6] M. Aron and P. Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. In *Proc. of the 17th Symp. on Operating Systems Principles*, pages 232–246, Kiawah Island Resort, SC, December 1999.

[7] H.E. Bal. *Programming Distributed Systems*. Prentice Hall International Ltd., Hemel Hempstead, UK, 1991.

[8] H.E. Bal and L.V. Allis. Parallel Retrograde Analysis on a Distributed System. In *Supercomputing '95*, San Diego, CA, December 1995.

[9] H.E. Bal, R.A.F. Bhoedjang, R.F.H. Hofman, C. Jacobs, K.G. Langendoen, T. Rühl, and M.F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Trans. on Computer Systems*, 16(1):1–40, February 1998.

[10] H.E. Bal, R.A.F. Bhoedjang, R.F.H. Hofman, C. Jacobs, K.G. Langendoen, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. *Journal of Parallel and Distributed Computing*, 40(1):49–64, February 1997.

[11] H.E. Bal and M.F. Kaashoek. Object Distribution in Orca using Compile-time and Run-Time Techniques. In *Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 162–177, Washington, DC, September 1993.

[12] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Trans. on Software Engineering*, 18(3):190–205, March 1992.

[13] A. Basu, M. Welsh, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Hot Interconnects'97*, Stanford, CA, April 1997.

[14] R.A.F. Bhoedjang and K.G. Langendoen. Friendly and Efficient Message Handling. In *Proc. of the 29th Annual Hawaii Int. Conf. on System Sciences*, pages 121–130, Maui, HI, January 1996.

[15] R.A.F. Bhoedjang, J.W. Romein, and H.E. Bal. Optimizing Distributed Data Structures Using Application-Specific Network Interface Software. In *Proc. of the 1998 Int. Conf. on Parallel Processing*, pages 485–492, Minneapolis, MN, August 1998.

[16] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Proc. of the 1998 Int. Conf. on Parallel Processing*, pages 381–390, Minneapolis, MN, August 1998.

[17] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. LFC: A Communication Substrate for Myrinet. In *4th Annual Conf. of the Advanced School for Computing and Imaging*, pages 31–37, Lommel, Belgium, June 1998.

[18] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[19] R.A.F. Bhoedjang, T. Rühl, R.F.H. Hofman, K.G. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A Portable Platform to Support Parallel Programming Languages. In *Proc. of the USENIX Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 213–226, San Diego, CA, September 1993.

[20] A. Bilas, D. Jiang, Y. Zhou, and J.P. Singh. Limits to the Performance of Software Shared Memory: A Layered Approach. In *Proc. of the 5th Int. Symp. on High-Performance Computer Architecture*, pages 193–202, Orlando, FL, January 1999.

[21] A. Bilas, C. Liao, and J.P. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proc. of the 26th Int. Symp. on Computer Architecture*, pages 282–293, Atlanta, GA, May 1999.

[22] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[23] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, 2(1):39–59, February 1984.

[24] M.A. Blumrich, R.D. Alpert, Y. Chen, D.W. Clark, S.N. Damianakis, E.W. Felten, L. Iftode, K. Li, M. Martonosi, and R.A. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In *Proc. of the 25th Int. Symp. on Computer Architecture*, pages 330–341, Barcelona, Spain, June 1998.

[25] M.A. Blumrich, C. Dubnicki, E.W. Felten, and K. Li. Protected, User-level DMA for the SHRIMP Network Interface. In *Proc. of the 2nd Int. Symp. on High-Performance Computer Architecture*, pages 154–165, San Jose, CA, February 1996.

[26] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP multicomputer. In *Proc. of the 21st Int. Symp. on Computer Architecture*, pages 142–153, Chicago, IL, April 1994.

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

[28] E.A. Brewer, F.T. Chong, L.T. Liu, S.D. Sharma, and J.D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proc. of the 1995 Symp. on Parallel Algorithms and Architectures*, pages 42–53, Santa Barbara, CA, July 1995.

[29] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proc. of the 1997 Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 214–224, Seattle, WA, June 1997.

[30] J. Bruck, L. De Coster, N. Dewulf, C.-T. Ho, and R. Lauwereins. On the Design and Implementation of Broadcast and Global Combine Operations Using the Postal Model. *IEEE Trans. on Parallel and Distributed Systems*, 7(3):256–265, March 1996.

[31] M. Buchanan. Private communication, 1997.

[32] G. Buzzard, D. Jacobson, M. MacKey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-managed Interface Architecture. In *2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 245–259, Seattle, WA, October 1996.

[33] J. Carreira, J. Gabriel Silva, K.G. Langendoen, and H.E. Bal. Implementing Tuple Space with Threads. In *1st Int. Conf. on Parallel and Distributed Systems*, pages 259–264, Barcelona, Spain, June 1997.

[34] C.-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low Latency Communication on the IBM RS/6000 SP. In *Supercomputing '96*, Pitssburgh, PA, November 1996.

[35] C.-C. Chang and T. von Eicken. A Software Architecture for Zero-Copy RPC in Java. Technical Report 98-1708, Dept. of Computer Science, Cornell University, Ithaca, NY, September 1998.

[36] J.-D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 1–19, Denver, CO, November 1999.

[37] B. Chun, A. Mainwaring, and D.E. Culler. Virtual Network Transport Protocols for Myrinet. In *Hot Interconnects'97*, Stanford, CA, April 1997.

[38] D.D. Clark. The Structuring of Systems Using Upcalls. In *Proc. of the 10th Symp. on Operating Systems Principles*, pages 171–180, Orcas Island, WA, December 1985.

[39] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurty, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Supercomputing '93*, pages 262–273, Portland, OR, November 1993.

[40] D.E. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *4th Symp. on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, CA, May 1993.

[41] D.E. Culler, L.T. Liu, R.P. Martin, and C.O. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[42] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming . *IEEE Computational Science and Engineering*, 5(1):46–55, January/March 1998.

[43] W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. on Computers*, 36(5):547–553, May 1987.

[44] E.W. Dijkstra. Guarded Commands. *Communications of the ACM*, 18(8):453–457, August 1975.

[45] J.J. Dongarra, S.W. Otto, M. Snir, and D.W. Walker. A Message Passing Standard for MPP and Workstations. *Communications of the ACM*, 39(7):84–90, July 1996.

[46] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proc. of the 13th Symp. on Operating Systems Principles*, pages 122–136, Asilomar, Pacific Grove, CA, October 1991.

[47] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, October 1996.

[48] P. Druschel, L.L. Peterson, and B.S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proc. of the 1994 Conf. on Communications Architectures, Protocols, and Applications (SIGCOMM)*, pages 2–13, London, UK, September 1994.

[49] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects'97*, Stanford, CA, April 1997.

[50] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *11th Int. Parallel Processing Symp.*, pages 388–396, Geneva, Switzerland, April 1997.

[51] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, March–April 1998.

[52] A. Fekete, M. F. Kaashoek, and N. Lynch. Implementing Sequentially Consistent Shared Objects Using Broadcast and Point-to-Point Communication. In *Proc. of the 15th Int. Conf. on Distributed Computing Systems*, pages 439–449, Vancouver, Canada, May 1995.

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

[54] I. Foster, C. Kesselman, and S. Tuecke. The NEXUS Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(2):70–82, February 1996.

[55] M. Frigo, C.E. Leiserson, and K.H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 212–223, Montreal, Canada, June 1998.

[56] M. Gerla, P. Palnati, and S. Walton. Multicasting Protocols for High-Speed, Wormhole-Routing Local Area Networks. In *Proc. of the 1996 Conf. on Communications Architectures, Protocols, and Applications (SIGCOMM)*, pages 184–193, Stanford University, CA, August 1996.

[57] R.B. Gillett. Memory Channel for PCI. *IEEE Micro*, 15(1):12–18, February 1996.

[58] S.C. Goldstein, K.E. Schauser, and D.E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[59] G.H. Golub and C. F. van Loan. *Matrix Computations*. The John Hopkins University Press, 3rd edition, 1996.

[60] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[61] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[62] The Panda group. *The Panda 4.0 Interface Document*. Dept. of Mathematics and Computer Science, Vrije Universiteit, April 1998. On-line at http://www.cs.vu.nl/panda/panda4/panda.html.

[63] M.D. Haines. An Open Implementation Analysis and Design of Lightweight Threads. In *Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 229–242, Atlanta, GA, October 1997.

[64] M.D. Haines and K.G. Langendoen. Platform-Independent Runtime Optimizations Using OpenThreads. In *11th Int. Parallel Processing Symp.*, pages 460–466, Geneva, Switzerland, April 1997.

[65] H.P. Heinzle, H.E. Bal, and K.G. Langendoen. Implementing Object-Based Distributed Shared Memory on Transputers. In *Transputer Applications and Systems '94*, pages 390–405, Villa Erba, Cernobbio, Como, Italy, September 1994.

[66] W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, D.A. Wallach, and W.E. Weihl. Efficient Implementation of High-Level Languages on User-Level Communication Architectures. Technical Report MIT/LCS/TR-616, MIT Laboratory for Computer Science, May 1994.

[67] W.C. Hsieh, M.F. Kaashoek, and W.E. Weihl. Dynamic Computation Migration in DSM Systems. In *Supercomputing '96*, Pitssburgh, PA, November 1996.

[68] Y. Huang and P.M. McKinley. Efficient Collective Operations with ATM Network Interface Support. In *Proc. of the 1996 Int. Conf. on Parallel Processing*, volume I, pages 34–43, Bloomingdale, IL, August 1996.

[69] G. Iannello, M. Lauria, and S. Mercolino. Cross-Platform Analysis of Fast Messages for Myrinet. In *Proc. of the Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (LNCS 1362)*, pages 217–231, Las Vegas, NV, January 1998.

[70] Intel Corporation. *Pentium Pro Family Developer's Manual*, 1995.

[71] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Guide*, 1995.

[72] SPARC International. *The SPARC Architecture Manual: Version 8*, 1992.

[73] K.L. Johnson. *High-Performance All-Software Distributed Shared Memory*. PhD thesis, MIT Laboratory for Computer Science, Cambridge, USA, December 1995. Technical Report MIT/LCS/TR-674.

[74] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the 15th Symp. on Operating Systems Principles*, pages 213–226, Copper Mountain, CO, December 1995.

[75] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit Amsterdam, 1992.

[76] V. Karamcheti and A.A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *Proc. of the 22nd Int. Symp. on Computer Architecture*, pages 298–307, Santa Margherita Ligure, Italy, June 1995.

[77] R.M. Karp, A. Sahay, E.E. Santos, and K.E. Schauser. Optimal Broadcast and Summation in the LogP Model. In *Proc. of the 1993 Symp. on Parallel Algorithms and Architectures*, pages 142–153, Velen, Germany, June 1993.

[78] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 Usenix Conf.*, pages 115–131, San Francisco, CA, January 1994.

[79] D. Keppel. Register Windows and User Space Threads on the SPARC. Technical Report UWCSE 91-08-01, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, August 1991.

[80] R. Kesavan and D.K. Panda. Optimal Multicast with Packetization and Network Interface Support. In *Proc. of the 1997 Int. Conf. on Parallel Processing*, pages 370–377, Bloomingdale, IL, August 1997.

[81] C.H. Koelbel, D.B. Loveman, R. Schreiber, Jr G.L. Steele, and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[82] L.I. Kontothanassis and M.L. Scott. Using Memory-Mapped Network Interface to Improve the Performance of Distributed Shared Memory. In *Proc. of the 2nd Int. Symp. on High-Performance Computer Architecture*, pages 166–177, San Jose, CA, February 1996.

[83] A. Krishnamurthy, K.E. Schauser, C.J. Scheiman, R.Y. Wang, D.E. Culler, and K. Yelick. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. In *Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, Cambridge, MA, October 1996.

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

[85] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. of the 21st Int. Symp. on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[86] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[87] K. Langendoen, R.A.F. Bhoedjang, and H.E. Bal. Automatic Distribution of Shared Data Objects. In *Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, Troy, NY, May 1995.

[88] K.G. Langendoen, R.A.F. Bhoedjang, and H.E. Bal. Models for Asynchronous Message Handling. *IEEE Concurrency*, 5(2):28–37, April–June 1997.

[89] K.G. Langendoen, J. Romein, R.A.F. Bhoedjang, and H.E. Bal. Integrating Polling, Interrupts, and Thread Management. In *The 6th Symp. on the Frontiers of Massively Parallel Computation*, pages 13–22, Annapolis, MD, October 1996.

[90] M. Lauria and A.A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.

[91] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong-Chan, S.-W. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Proc. of the 1992 Symp. on Parallel Algorithms and Architectures*, pages 272–285, San Diego, CA, June 1992.

[92] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[93] C. Liao, D. Jiang, L. Iftode, M. Martonosi, and D.W. Clark. Monitoring Shared Virtual Memory Performance on a Myrinet-based PC Cluster. In *Supercomputing'98*, Orlando, FL, November 1998.

[94] J. Liedtke. On Micro-Kernel Construction. In *Proc. of the 15th Symp. on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, December 1995.

[95] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[96] P. López, J.M. Martínez, and J. Duato. A Very Efficient Distributed Deadlock Detection Mechanism for Wormhole Networks. In *Proc. of the 4th Int. Symp. on High-Performance Computer Architecture*, pages 57–66, Las Vegas, NV, February 1998.

[97] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Quantifying the Performance Differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):65–78, June 1997.

[98] J. Maassen and R. van Nieuwpoort. Fast Parallel Java. Master's thesis, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, August 1998.

[99] J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *7th Symp. on Principles and Practice of Parallel Programming*, pages 173–182, Atlanta, GA, May 1999.

[100] A.M. Mainwaring, B.N. Chun, S. Schleimer, and D.S. Wilkerson. System Area Network Mapping. In *Proc. of the 1997 Symp. on Parallel Algorithms and Architectures*, pages 116–126, Newport, RI, June 1997.

[101] O. Maquelin, G.R. Gao, H.H.J. Hum, K.B. Theobald, and X. Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proc. of the 23rd Int. Symp. on Computer Architecture*, pages 179–188, Philadelphia, PA, May 1996.

[102] R.P. Martin, A.M. Vahdat, D.E. Culler, and T.E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proc. of the 24th Int. Symp. on Computer Architecture*, pages 85–97, Denver, CO, June 1997.

[103] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers. In *Proc. of the 1996 Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 138–147, Philadelphia, PA, May 1996.

[104] H. McGhan and M. O'Connor. PicoJava: A Direct Execution Engine for Java Bytecode. *IEEE Computer*, 31(10):22–30, October 1998.

[105] E. Mohr, D.A. Kranz, and R.H. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[106] D. Mosberger and L.L. Peterson. Careful Protocols or How to Use Highly Reliable Networks. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, pages 80–84, Napa, CA, October 1993.

[107] S.S. Mukherjee, B. Falsafi, M.D. Hill, and D.A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proc. of the 23rd Int. Symp. on Computer Architecture*, pages 247–258, Philadelphia, PA, May 1996.

[108] S.S. Mukherjee and M.D. Hill. The Impact of Data Transfer and Buffering Alternatives on Network Interface Design. In *Proc. of the 4th Int. Symp. on High-Performance Computer Architecture*, pages 207–218, Las Vegas, NV, February 1998.

[109] G. Muller, R. Marlet, E.-N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, Optimized Sun RPC Using Automatic Program Specialization. In *Proc. of the 18th Int. Conf. on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998.

[110] B. Nichols, B. Buttlar, and J. Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Newton, MA, 1996.

[111] M. Oey, K. Langendoen, and H.E. Bal. Comparing Kernel-Space and User-Space Communication Protocols on Amoeba. In *Proc. of the 15th Int. Conf. on Distributed Computing Systems*, pages 238–245, Vancouver, Canada, May 1995.

[112] S. Pakin, V. Karamcheti, and A.A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively Parallel Processors. *IEEE Concurrency*, 5(2):60–73, April–June 1997.

[113] S. Pakin, M. Lauria, and A.A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, December 1995.

[114] D. Perkovic and P.J. Keleher. Responsiveness without Interrupts. In *Proc. of the Int. Conf. on Supercomputing*, pages 101–108, Rhodes, Greece, June 1999.

[115] L.L. Peterson and B.S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[116] L.L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao. The x-kernel: A Platform for Accessing Internet Resources. *IEEE Computer*, 23(5):23–33, May 1990.

[117] M. Philippsen and M. Zenger. JavaParty — Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, pages 1225–1242, November 1997.

[118] L. Prylli and B. Tourancheau. Protocol Design for High Performance Networking: A Myrinet Experience. Technical Report 97-22, LIP-ENS Lyon, July 1997.

[119] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. of the 21st Int. Symp. on Computer Architecture*, pages 325–337, Chicago, IL, April 1994.

[120] S.H. Rodrigues, T.E. Anderson, and D.E. Culler. High-Performance Local-Area Communication With Fast Sockets. In *USENIX Technical Conf.*, pages 257–274, Anaheim, CA, January 1997.

[121] J.W. Romein, H.E Bal, and D. Grune. An Application Domain Specific Language for Describing Board Games. In *Parallel and Distributed Processing Techniques and Applications*, volume I, pages 305–314, Las Vegas, NV, July 1997.

[122] J.W. Romein, A. Plaat, H.E. Bal, and J. Schaeffer. Transposition Driven Work Scheduling in Distributed Search. In *AAAI National Conference*, pages 725–731, Orlando, FL, July 1999.

[123] T. Rühl and H.E. Bal. A Portable Collective Communication Library using Communication Schedules. In *Proc. of the 5th Euromicro Workshop on Parallel and Distributed Processing*, pages 297–306, London, United Kingdom, January 1997.

[124] T. Rühl, H.E. Bal, R. Bhoedjang, K.G. Langendoen, and G. Benson. Experience with a Portability Layer for Implementing Parallel Programming Systems. In *The 1996 Int. Conf. on Parallel and Distributed Processing*

Techniques and Applications, pages 1477–1488, Sunnyvale, CA, August 1996.

[125] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, October 1996.

[126] I. Schoinas and M.D. Hill. Address Translation Mechanisms in Network Interfaces. In *Proc. of the 4th Int. Symp. on High-Performance Computer Architecture*, pages 219–230, Las Vegas, NV, February 1998.

[127] J.T. Schwartz. Ultracomputers. *ACM Trans. on Programming Languages and Systems*, 2(4):484–521, October 1980.

[128] R. Sivaram, R. Kesavan, D.K. Panda, and C.B. Stunkel. Where to Provide Support for Efficient Multicasting in Irregular Networks: Network Interface or Switch? In *Proc. of the 1998 Int. Conf. on Parallel Processing*, pages 452–459, Minneapolis, MN, August 1998.

[129] M. Snir, P. Hochschild, D.D. Frye, and K.J. Gildea. The Communication Software and Parallel Environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[130] E. Speight, H. Abdel-Shafi, and J.K. Bennett. Realizing the Performance Potential of the Virtual Interface Architecture. In *Proc. of the Int. Conf. on Supercomputing*, pages 184–192, Rhodes, Greece, June 1999.

[131] E. Speight and J.K. Bennett. Using Multicast and Multithreading to Reduce Communication in Software DSM Systems. In *Proc. of the 4th Int. Symp. on High-Performance Computer Architecture*, pages 312–323, Las Vegas, NV, February 1998.

[132] E. Spertus, S.C. Goldstein, K.E. Schauser, T. von Eicken, D.E. Culler, and W.J. Dally. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proc. of the 20th Int. Symp. on Computer Architecture*, pages 302–313, San Diego, CA, May 1993.

[133] P.A. Steenkiste. A Systematic Approach to Host Interface Design for High-Speed Networks. *IEEE Computer*, 27(3):47–57, March 1994.

[134] D. Stodolsky, J.B. Chen, and B. Bershad. Fast Interrupt Priority Management in Operating Systems. In *Proc. of the USENIX Symp. on Microkernels*

and Other Kernel Architectures, pages 105–110, San Diego, September 1993.

[135] C.B. Stunkel, J. Herring, B. Abali, and R. Sivaram. A New Switch Chip for IBM RS/6000 SP Systems. In *Supercomputing'99*, Portland, OR, November 1999.

[136] C.B. Stunkel, R. Sivaram, and D.K. Panda. Implementing Multidestination Worms in Switch-based Parallel Systems: Architectural Alternatives and their Impact. In *Proc. of the 24th Int. Symp. on Computer Architecture*, pages 50–61, Denver, CO, June 1997.

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

[138] A.S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, NJ, 3rd edition, 1996.

[139] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(2):46–63, December 1990.

[140] H. Tang, K. Shen, and T. Yang. Compile/Run-Time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. In *7th Symp. on Principles and Practice of Parallel Programming*, pages 107–118, Atlanta, GA, May 1999.

[141] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High-Performance Communication Library. In *High-Performance Computing and Networking (LNCS 1225)*, pages 708–717, Vienna, Austria, April 1997.

[142] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *12th Int. Parallel Processing Symp.*, pages 308–314, Orlando, FL, March 1998.

[143] C.A. Thekkath and H.M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, CA, October 1994.

[144] R. van Renesse, K.P. Birman, and S. Maffeis. Horus, a Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[145] R. Veldema. Jcc, a Native Java Compiler. Master's thesis, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, August 1998.

[146] K. Verstoep, K.G. Langendoen, and H.E. Bal. Efficient Reliable Multicast on Myrinet. In *Proc. of the 1996 Int. Conf. on Parallel Processing*, volume III, pages 156–165, Bloomingdale, IL, August 1996.

[147] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th Symp. on Operating Systems Principles*, pages 303–316, Copper Mountain, CO, December 1995.

[148] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int. Symp. on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[149] T. von Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, 31(11):61–68, November 1998.

[150] D.A. Wallach, W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, and W.E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *5th Symp. on Principles and Practice of Parallel Programming*, pages 217–226, Santa Barbara, CA, July 1995.

[151] H. Wasserman, O.M. Lubeck, Y. Luo, and F. Bassetti. Performance Evaluation of the SGI Origin2000: A Memory-Centric Characterization of LANL ASCI Applications. In *Supercomputing'97*, San Jose, CA, November 1997.

[152] A. Wollrath, J. Waldo, and R. Riggs. Java-Centric Distributed Computing. *IEEE Micro*, 17(3):44–53, May/June 1997.

[153] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Int. Symp. on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[154] K. Yocum, J. Chase, A. Gallatin, and A. Lebeck. Cut-Through Delivery in Trapeze: An Exercise in Low-Latency Messaging. In *The 6th Int. Symp. on*

High Performance Distributed Computing, pages 243–252, Portland, OR, August 1997.

[155] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release-Consistency Protocols for Shared Virtual Memory Systems. In *2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 75–88, Seattle, WA, October 1996.

Appendix A

Deadlock issues

As discussed in Section 4.2, LFC's basic multicast protocol cannot use arbitrary multicast trees. With chains, for example, we can create a buffer deadlock (see Figure 4.8). Below, in Section A.1, we derive — informally — a sufficient condition for deadlock-free multicasting in LFC. The binary trees used by LFC satisfy this condition. The condition applies only to multicasting within a single multicast group. The example in Section A.2 shows that the condition does *not* guarantee deadlock-free multicasting in overlapping multicast groups.

A.1 Deadlock-Free Multicasting in LFC

Before deriving a condition for deadlock-free multicasting, we first consider the nature of deadlocks in LFC. Recall that LFC partitions each NI's receive buffer space among all possible senders (see Section 4.1). That is, each NI has a separate receive buffer pool for each sender.

A deadlock in LFC consists of a cycle C of NIs such that for each NI S with successor R (both in C)

1. S has a nonempty blocked-sends queue $BSQ_{S \to R}$ for R

2. R has a full NI receive buffer pool $RBP_{S \to R}$ for S

We assume that hosts are not involved in the deadlock cycle. This is true only if all hosts regularly drain the network by supplying free host receive buffers to their NI. This requirement, however, is part of the contract between LFC and its clients (see Section 3.6). If all hosts drain the network, the full receive pools in the deadlock cycle do not contain packets that are waiting (only) for a free host receive buffer. Consequently, each packet p in a full receive pool $RBP_{S \to R}$ is waiting to be forwarded. That is, p is a multicast packet that has travelled from S to R and is

enqueued on at least one blocked-sends queue involved in a deadlock cycle (not necessarily C). Finally, observe that if $RBP_{S\to R}$ is part of deadlock cycle C, then at least one of the packets in $RBP_{S\to R}$ has to be enqueued on the blocked-sends queue to the next NI in C (i.e., to the successor of R).

We now formulate a sufficient condition for deadlock-free multicasting in LFC. We assume the system consists of P NIs, numbered $0,\ldots,P-1$. The 'distance' from NI m to NI n is defined as $(n+P-m) \bmod P$: the number of hops needed to reach n from m if all NIs are organized in a unidirectional ring. In a multicast group G the set of multicast trees T_G (one per group member) cannot yield deadlock if the following condition holds:

> For each member $g \in G$, it holds that, for each tree $t \in T_G$, the distance from g's parent to g in t is smaller than each of the distances from g to its children in t.

To see why this is true, assume that the condition is satisfied and that we can construct a deadlock cycle C of length k. According to the nature of deadlocks in LFC, each NI n_i in C holds at least one multicast packet p_i from its predecessor in C that needs to be forwarded to its successor in C. According to the condition, each NI in the cycle must therefore have a larger distance to its successor than to its predecessor. That is, if we denote the deadlock cycle

$$n_0 \xrightarrow{d_0} n_1 \xrightarrow{d_1} \ldots \xrightarrow{d_{k-1}} n_0$$

where d_i is the distance from n_{i-1} to n_i, then we must have

$$d_0 < d_1 < \ldots < d_{k-1} < d_0$$

which is impossible.

Using this condition, we can see why LFC's binary trees are deadlock-free. In the binary tree for node 0, it is clear that each node's distances to its children are larger than the distance to the node's parent (see for example Figure 4.9). The multicast tree for a node $p \neq 0$ is obtained by renumbering each tree node n in the multicast tree of node 0 to $(n+p) \bmod P$, where P is the number of processors. This renumbering preserves the distances between nodes, so the condition applies for all trees. Consequently, LFC's binary trees are deadlock-free.

Previous versions of LFC used *binomial* instead of binary trees [16].[1] Figure A.1 shows a 16-node binomial tree. In binomial trees the distance (as defined

[1]This paper ([16]) contains two errors. First, it incorrectly states that forwarding in binomial trees is equivalent to e-cube routing, which is deadlock-free [43]. This is true for some senders (e.g., node 0), but not for all. Second, the paper assumes that binary trees are not deadlock-free.

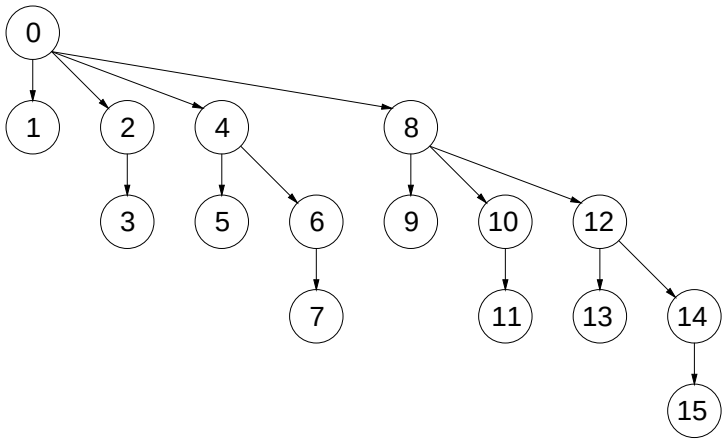


Fig. A.1. A biomial spanning tree.

above) between any pair of nodes is always a power of two. Moreover, and in contrast with binary trees, binomial trees have the property that the distance between a node and its parent is always *greater* than the distances between a node and its children. The condition for deadlock-free multicasting in LFC can easily be adapted to apply to trees that have the latter property. Using this adapted version, we can show that binomial multicast trees are also deadlock-free.

A.2 Overlapping Multicast Groups

Without deadlock recovery, overlapping multicast groups are unsafe. Consider a 9-node system with three multicast groups $G_1 = \{0,3,4,5,6\}$, $G_2 = \{0,3,6,7,8\}$, and $G_3 = \{0,1,2,3,6\}$. Within each group, we use binary multicast trees. These trees are constructed as if the nodes within each group were numbered $0,\ldots,4$. Figure A.2 shows the trees for node 0 in G_1, for node 3 in G_2, and for node 6 in G_3. The bold arrows in each tree indicate a forwarding path that overlaps with a forwarding path in another tree. By concatenating these paths we can create a deadlock cycle.

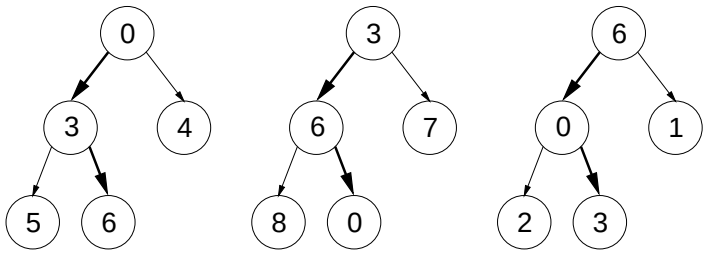


Fig. A.2. Multicast trees for overlapping multicast groups.

Appendix B

Abbreviations

ADC	Application Device Channel
ADI	Application Device Interface
AM	Active Messages
API	Application Programming Interface
ASCI	Advanced School for Computing and Imaging
ATM	Asynchronous Transfer Mode
BB	Broadcast/Broadcast
BIP	Basic Interface for Parallelism
CRL	C Region Library
DAS	Distributed ASCI Supercomputer
DMA	Direct Memory Access
DSM	Distributed Shared Memory
F&A	Fetch and Add
FM	Fast Messages
GSB	Get Sequence number, then Broadcast
JIT	Just In Time
JVM	Java Virtual Machine
LFC	Link-level Flow Control
MG	Multigame
MPI	Message Passing Interface
MPICH	MPI Chameleon
MPL	Message-Passing Library
MPP	Massively Parallel Processor
NI	Network Interface
NIC	Network Interface Card
OT	OpenThreads

PB	Point-to-point/Broadcast
PIO	Programmed I/O
PPS	Parallel Programming System
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RSR	Remote Service Request
RTS	RunTime System
SAP	Service Access Point
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
UDP	User Datagram Protocol
UTLB	User-managed TLB
VI	Virtual Interface
VMMC	Virtual Memory-Mapped Communication
WWW	World Wide Web

Samenvatting

Parallelle programma's laten een aantal computers tegelijk aan één rekenintensief probleem werken met als doel dat probleem sneller op te lossen dan met één computer mogelijk is. Daar het moeilijk is om computers efficiënt en correct samen te laten werken, zijn er systemen ontwikkeld om het schrijven en uitvoeren van parallelle programma's te vereenvoudigen. Dergelijke systemen noemen we parallelle programmeersystemen.

Verschillende parallelle programmeersystemen bieden verschillende programmeerparadigmata aan en worden in verschillende hardware-omgevingen gebruikt. Dit proefschrift concentreert zich op programmeersystemen voor *computerclusters*. Zo'n cluster bestaat uit een door een netwerk verbonden verzameling computers. De meeste clusters bestaan uit tientallen computers, maar er bestaan ook clusters van honderden en zelfs duizenden computers. Kenmerkend voor een computercluster is dat zowel de computers als het netwerk door massaproductie relatief goedkoop zijn en dat de computers met elkaar communiceren door via het netwerk, en niet via een gemeenschappelijk geheugen, berichten met elkaar uit te wisselen.

De uitwisseling van berichten wordt ondersteund door communicatiesoftware. Deze software bepaalt in belangrijke mate de prestaties van de parallelle programma's die met een parallel programmeersysteem ontwikkeld worden. Dit proefschrift, getiteld "Communicatie-architecturen voor parallelle programmeersystemen", richt zich op deze communicatiesoftware en behandelt de volgende vragen:

1. Welke mechanismen dient een communicatiesysteem aan te bieden?

2. Hoe moeten parallelle programmeersystemen de aangeboden mechanismen gebruiken?

3. Hoe moeten deze mechanismen in het communicatiesysteem geïmplementeerd worden?

Met betrekking tot deze vragen levert het proefschrift de volgende bijdragen:

1. Het toont aan dat een kleine verzameling eenvoudige communicatiemechanismen effectief aangewend kan worden om verschillende parallelle programmeersystemen efficiënt te implementeren. Deze communicatiemechanismen zijn geïmplementeerd in een nieuw communicatiesysteem, LFC, dat later in deze samenvatting preciezer beschreven wordt. Een aantal van deze mechanismen veronderstelt ondersteuning van de netwerkadapter. De netwerkadapter is het computeronderdeel dat de koppeling tussen netwerk en computer verzorgt. Moderne netwerkadapters zijn vaak programmeerbaar en kunnen daarom een aantal communicatietaken uitvoeren die traditioneel door een computer uitgevoerd worden.

2. Het beschrijft een efficiënt en betrouwbaar multicast-algoritme. Multicast is één van de bovengenoemde communicatiemechanismen en speelt in verschillende parallelle programmeersystemen een belangrijke rol. Het is een vorm van communicatie waarbij één computer een bericht naar verscheidene andere computers verstuurt.

   Op netwerken die multicast niet in hardware ondersteunen, wordt multicast geïmplementeerd door middel van *doorsturen*. Een multicastbericht wordt door de zender eerst sequentieel naar een (meestal) klein aantal computers verstuurd; ieder van die computers ontvangt het bericht, verwerkt het en stuurt het door naar andere computers, enz. Deze wijze van doorsturen is redelijk efficiënt, omdat het bericht door verscheidene computers tegelijk verwerkt en doorgestuurd wordt. Zoals beschreven is de methode echter niet optimaal, omdat iedere doorsturende computer het bericht terugkopieert naar de netwerkadapter. Dit proefschrift beschrijft een algoritme dat gebruik maakt van de netwerkadapter om multicastberichten op efficiëntere wijze door te sturen: zodra de netwerkadapter een multicastbericht ontvangt, stuurt het dit bericht zelfstandig door naar andere netwerkadapters en kopieert het tevens naar de computer. De ontvangende computer is niet meer betrokken bij het doorsturen van het bericht.

3. Het onderzoekt op systematische wijze de invloed van verschillende verdelingen van protocoltaken tussen computer en netwerkadapter op de prestaties van parallelle programma's. De evaluatie concentreert zich op het wel of niet gebruiken van de netwerkadapter bij het implementeren van betrouwbare communicatie en bij het doorsturen van multicastberichten. Een interessant resultaat is dat bepaalde werkverdelingen de prestaties van sommige parallelle programma's verbeteren, maar de prestaties van andere programma's doen verslechteren. De invloed van een werkverdeling blijkt deels af te hangen van de communicatiepatronen die een parallel programmeersysteem gebruikt.

Hoofdstuk 1 van dit proefschrift beschrijft bovengenoemde onderzoeksvragen en -bijdragen in detail.

Hoofdstuk 2 geeft een overzicht van bestaande, efficiënte communicatiesystemen. Dit zijn bijna zonder uitzondering systemen die rechtstreeks door gebruikersprocessen aangesproken kunnen worden, dat wil zeggen, zonder tussenkomst van het besturingssysteem. Hoewel de eliminatie van het besturingssysteem leidt tot goede prestaties in eenvoudige tests, voldoet geen van deze communicatiesystemen volledig aan de eisen die parallelle programmeersystemen stellen. In het bijzonder blijkt dat veel van deze communicatiesystemen geen asynchrone afhandeling van berichten of geen multicast ondersteunen; beide zijn belangrijk in parallelle programmeersystemen. Een andere belangrijke observatie is dat de bestaande communicatiesystemen sterk verschillen in de manier waarop ze werk verdelen tussen computer en netwerkadapter. Sommige systemen beperken het werk op de netwerkadapter tot een minimum, omdat de processor van de netwerkadapter in het algemeen traag is. Andere systemen daarentegen, draaien volledige communicatieprotocollen op de netwerkadapter. Hoofdstuk 8 van dit proefschrift onderzoekt op systematische wijze de invloed van verschillende werkverdelingen.

Hoofdstuk 3 beschrijft het ontwerp en de implementatie van LFC, een nieuw communicatiesysteem voor parallelle programmeersystemen. LFC ondersteunt vijf parallelle programmeersystemen. Hoewel deze programmeersystemen sterk verschillende communicatie-eisen stellen, zijn ze alle op efficiënte wijze bovenop LFC geïmplementeerd.

LFC biedt de gebruiker een eenvoudig en laag-niveau communicatie-interface aan. Dit interface bestaat uit functies om netwerkpakketten betrouwbaar naar één of meer andere computers te versturen. Deze pakketten kunnen zowel synchroon als asynchroon ontvangen worden. Het interface bevat ook een eenvoudige, maar nuttige synchronisatiefunctie (fetch-and-add).

De implementatie van het interface maakt agressief gebruik van een programmeerbare netwerkadapter. Hoofdstuk 4 beschrijft in detail de protocollen en taken die LFC op de netwerkadapter uitvoert:

- het voorkomen van bufferoverloop ten behoeve van betrouwbare communicatie (flow control)
- het doorsturen van multicastberichten
- het vertraagd genereren van interrupts
- het afhandelen van synchronisatieberichten

LFC veronderstelt dat de netwerkhardware pakketten corrumpeert noch verliest, maar dat is niet voldoende om betrouwbare communicatie tussen processen

te garanderen. Het is ook noodzakelijk dat ontvangers van berichten voldoende bufferruimte hebben om binnenkomende berichten op te slaan. Om dat te garanderen, implementeert LFC een eenvoudig flow-controlprotocol op de netwerkadapter.

LFC laat de netwerkadapter multicastberichten zonder tussenkomst van de computer doorsturen. Het blijkt mogelijk om dit te implementeren als een eenvoudige uitbreiding van bovengenoemd flow-controlprotocol.

LFC staat gebruikers toe om berichten synchroon te ontvangen, door een computer actief te laten testen of een bericht gearriveerd is (polling), of asynchroon, door de netwerkadapter een interrupt te laten genereren. Een computer kan snel testen of een bericht is binnengekomen, maar het afhandelen van een interrupt verloopt meestal traag. Als een bericht verwacht wordt, dan is polling dus efficiënter dan het gebruik van interrupts. Als niet bekend is wanneer het volgende bericht binnenkomt, dan is polling in het algemeen duur, omdat de meeste polls geen berichten zullen vinden. Omdat interrupts duur zijn, laat LFC de netwerkadapter pas na een korte vertraging een interrupt genereren. Op deze wijze krijgt de ontvangende computer de kans een bericht door middel van polling weg te lezen en de interrupt te voorkomen.

LFC biedt een eenvoudige synchronisatiefunctie aan: fetch-and-add. Deze functie leest en verhoogt op ondeelbare wijze de waarde van een gedeelde integervariabele. LFC slaat fetch-and-add-variabelen in het geheugen van de netwerkadapter op en laat de netwerkadapter zelfstandig binnenkomende fetch-and-add-verzoeken afhandelen.

Hoofdstuk 5 evalueert de prestaties van LFC. De taken die LFC op de netwerkadapter uitvoert zijn relatief eenvoudig, maar omdat de processor van een netwerkadapter in het algemeen traag is, is niet a priori duidelijk of het verstandig is al deze taken op de netwerkadapter uit te voeren. De prestatiemetingen in hoofdstuk 5 laten echter zien dat LFC uitstekend presteert, ondanks het agressieve gebruik van de relatief trage netwerkadapter.

Hoofdstuk 6 beschrijft Panda, een communicatiesysteem dat een hoger-niveau interface aanbiedt dan LFC. Panda is bovenop LFC geïmplementeerd en biedt de gebruiker threads, message passing, Remote Procedure Call (RPC) en groepscommunicatie. Deze abstracties vereenvoudigen vaak de implementatie van een parallel programmeersysteem.

Panda gebruikt in het threadsysteem aanwezige kennis om automatisch te besluiten of binnenkomende berichten door middel van polling dan wel interrupts ontvangen moeten worden. Gewoonlijk gebruikt Panda interrupts, maar zodra alle threads geblokkeerd zijn stapt Panda over op polling. Panda beschouwt ieder binnenkomend bericht als een bytestroom en staat een ontvangend proces toe om te beginnen met het lezen van deze bytestroom voordat het bericht in zijn geheel ontvangen is. Tenslotte implementeert Panda totaal-geordende groeps-

communicatie met behulp van een eenvoudige, maar effectieve combinatie van de synchronisatie- en broadcastfuncties van LFC.

Hoofdstuk 7 beschrijft vier parallelle programmeersystemen die met behulp van LFC en Panda geïmplementeerd zijn. Orca, CRL en Manta zijn object-gebaseerde systemen die processen op verschillende computers toestaan om objecten met elkaar te delen en om via die gedeelde objecten te communiceren. MPI daarentegen, is gebaseerd op het expliciet versturen van berichten tussen processen op verschillende computers (message passing). Orca en Manta zijn op programmeertalen gebaseerde systemen (respectievelijk Orca en Java). CRL en MPI worden niet door een compiler ondersteund.

Hoewel Orca, CRL en Manta vergelijkbare programmeerabstracties aanbieden, implementeren ze die abstracties op verschillende manieren. Orca repliceert sommige objecten en transporteert via RPC en groepscommunicatie de parameters van operaties op gedeelde objecten. Manta repliceert objecten niet en gebruikt alleen RPC om de parameters van operaties te transporteren. CRL repliceert objecten, maar maakt gebruik van invalidatie om de waarde van gerepliceerde objecten consistent te houden. Bovendien verstuurt CRL geen operatieparameters, maar objectdata. Orca, Manta en MPI zijn alle met behulp van Panda geïmplementeerd; CRL is direct bovenop LFC geïmplementeerd.

Het hoofdstuk laat zien dat al deze systemen efficiënt op LFC en Panda geïmplementeerd kunnen worden. In het geval van Orca, een relatief complex systeem, worden daartoe twee optimalisaties geïntroduceerd. Ten eerste genereert de Orca-compiler gespecialiseerde code voor het in- en uitpakken van berichten die operatieparameters bevatten. Dit voorkomt onnodig kopiëren van data en onnodige interpretatie van type-beschrijvingen. Ten tweede maakt Orca gebruik van voortzettingen (continuations) in plaats van threads om geblokkeerde operaties op objecten compact te representeren. Voorgaande Orca-implementaties gebruikten threads in plaats van voortzettingen. Threads nemen echter meer ruimte in beslag en kunnen in onvoorspelbare volgordes geactiveerd worden.

Hoofdstuk 8 beschrijft en evalueert andere implementaties van het communicatie-interface van LFC. In totaal worden vijf implementaties bestudeerd, inclusief de in hoofdstukken 3 t/m 5 beschreven LFC-implementatie. Deze implementaties verschillen in de manier waarop ze protocoltaken tussen computer en netwerkadapter verdelen, met name taken die verband houden met betrouwbaarheid en multicast. Het hoofdstuk concentreert zich op de invloed van deze verschillende werkverdelingen op de prestaties van parallelle programma's. Omdat alle LFC-implementaties hetzelfde communicatie-interface implementeren, kan deze invloed bestudeerd worden zonder wijziging van de parallelle programmeersystemen.

Hoofdstuk 8 bestudeert het gedrag van negen parallelle programma's. Deze programma's maken gebruik van Orca, CRL, MPI, en Multigame. Orca, CRL en

MPI zijn boven reeds besproken. Multigame is een declaratief parallel programmeersysteem dat automatisch en parallel spelbomen doorzoekt. Prestatiemetingen laten zien dat de in hoofdstukken 3 t/m 5 beschreven LFC-implementatie altijd de beste prestaties van parallelle programma's oplevert. Deze LFC-implementatie veronderstelt dat de netwerkhardware betrouwbaar is en gebruikt de netwerkadapter om multicastberichten door te sturen. De andere implementaties veronderstellen dat de netwerk-hardware onbetrouwbaar is en implementeren daarom een duurder betrouwbaarheidsprotocol. Sommige implementaties voeren dit protocol op de netwerkadapter uit, andere op de computer.

Een interessante uitkomst van de metingen is dat het uitvoeren van meer werk op de netwerkadapter de prestaties van sommige parallelle programma's verbetert, maar de prestaties van andere programma's doet verslechteren. Met name het gebruik van de netwerkadapter om betrouwbaarheid te implementeren kan zowel positief als negatief uitwerken. Het lijkt daarentegen altijd nuttig om multicastberichten door de netwerkadapter (en niet door de computer) door te laten sturen. De invloed van een werkverdeling blijkt deels af te hangen van de communicatiepatronen die een parallel programmeersysteem gebruikt. Voor communicatiepatronen die veel relatief kleine RPC-transacties bevatten is het verhogen van de werklast van de netwerkadapter ongunstig. Voor asynchrone communicatiepatronen daarentegen, is het nuttig om de computer te ontlasten en werk naar de netwerkadapter te verschuiven.

Hoofdstuk 9 besluit dit proefschrift. Uit het gepresenteerde werk blijkt dat een communicatiesysteem met een eenvoudig interface een verscheidenheid aan parallelle programmeersystemen efficiënt kan ondersteunen. Zo'n communicatiesysteem moet interrupts, polling en multicast aanbieden. De netwerkadapter speelt een belangrijke rol bij de implementatie van het communicatiesysteem. Met name multicast kan met behulp van de netwerkadapter efficiënt geïmplementeerd worden. De prestaties van parallelle applicaties worden echter niet alleen door het communicatiesysteem bepaald, maar ook door communicatiebibliotheken zoals Panda en door parallelle programmeersystemen. Dit proefschrift beschrijft verschillende technieken om ook deze lagen goed te laten presteren en legt verbanden tussen eigenschappen van communicatiesystemen, eigenschappen van parallelle programmeersystemen en de prestaties van parallelle programma's.

Curriculum Vitae

Personal data

Name:	Raoul A.F. Bhoedjang
Born:	14 December 1970, The Hague, The Netherlands
Nationality:	Dutch
Address:	Dept. of Computer Science Upson Hall, Cornell University Ithaca, NY 14853–7501 USA
Email:	raoul@cs.cornell.edu
WWW:	http://www.cs.cornell.edu/raoul/

Education

November 1992:	Master's degree in Computer Science (cum laude) Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
August 1988:	Athenaeum degree Han Fortmann College, Heerhugowaard, The Netherlands

Professional Experience

Aug. 1999 – present:	Researcher Dept. of Computer Science, Cornell University, Ithaca, NY, USA
July 1998 – July 1999:	Researcher Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
July 1994 – July 1998:	Graduate student Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
Feb. 1994 – June 1994:	Programmer Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
Dec. 1993 – Jan. 1994:	Teaching assistant (Compiler Construction) Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
Nov. 1993:	Visiting student Dept. of Computer Science, Cornell University, Ithaca, NY, USA
Sept. 1993 – Oct. 1993:	Visiting student Laboratory for Computer Science, MIT, Cambridge, MA, USA
Feb. 1993 – Sept. 1993:	Teaching assistant (Operating Systems) Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
July 1992 – Sept. 1992:	Summer Scholarship student Edinburgh Parallel Computing Centre, Scotland, UK

Jan. 1992 – June 1992:	Exchange student Computing Dept., Lancaster University, Lancaster, UK
Sept. 1991 – Dec. 1991:	Teaching assistant (Introduction to Programming) Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

Publications

Journal publications

1. R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

2. H.E. Bal, R.A.F. Bhoedjang, R. Hofman, C. Jacobs, K.G. Langendoen, T. Rühl, and M.F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Trans. on Computer Systems*, 16(1):1–40, February 1998.

3. H.E. Bal, K.G. Langendoen, R.A.F. Bhoedjang, and F. Breg. Experience with Parallel Symbolic Applications in Orca. *J. of Programming Languages*, 1998.

4. K.G. Langendoen, R.A.F. Bhoedjang, and H.E. Bal. Models for Asynchronous Message Handling. *IEEE Concurrency*, 5(2):28–37, April–June 1997.

5. H.E. Bal, R.A.F. Bhoedjang, R. Hofman, C. Jacobs, K.G. Langendoen, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. *J. of Parallel and Distributed Computing*, 40(1):49–64, February 1997.

Conference publications

1. T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, and R.A.F. Bhoedjang. MAGPIE: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Conf. on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–40, Atlanta, GA, May 1999.

2. T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, and R.A.F. Bhoedjang. MPI's Reduction Operations in Clustered Wide Area Systems. In *Message Passing Interface Developer's and User's Conference (MPIDC'99)* pages 43–52, Atlanta, GA, March 1999.

3. R.A.F. Bhoedjang, J.W. Romein, and H.E. Bal. Optimizing Distributed Data Structures Using Application-Specific Network Interface Software. In *Int. Conf. on Parallel Processing*, pages 485–492, Minneapolis, MN, August 1998.

4. R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Int. Conf. on Parallel Processing*, pages 381–390, Minneapolis, MN, August 1998. (Best paper award).

5. K.G. Langendoen, J. Romein, R.A.F. Bhoedjang, and H.E. Bal. Integrating Polling, Interrupts, and Thread Management. In *The 6th Symp. on the Frontiers of Massively Parallel Computation (Frontiers '96)*, pages 13–22, Annapolis, MD, October 1996.

6. T. Rühl, H.E. Bal, R. Bhoedjang, K.G. Langendoen, and G. Benson. Experience with a Portability Layer for Implementing Parallel Programming Systems. In *The 1996 Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Sunnyvale, CA, August 1996.

7. R.A.F. Bhoedjang and K.G. Langendoen. Friendly and Efficient Message Handling. In *Proc. of the 29th Annual Hawaii Int. Conf. on System Sciences*, pages 121–130, Maui, HI, January 1996.

8. K.G. Langendoen, R.A.F. Bhoedjang, and H.E. Bal. Automatic Distribution of Shared Data Objects. In B. Szymanski and B. Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 287–290, Troy, NY, May 1995. Kluwer Academic Publishers.

9. H.E. Bal, K.G. Langendoen, and R.A.F. Bhoedjang. Experience with Parallel Symbolic Applications in Orca. In T. Ito, R.H. Halstead, Jr, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (PSLS'95)*, number 1068 in Lecture Notes in Computer Science, pages 266–285, Beaune, France, October 1995.

10. R.A.F. Bhoedjang, T. Rühl, R. Hofman, K.G. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A Portable Platform to Support Parallel Programming Languages. In *Proc. of the USENIX Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 213–226, San Diego, CA, September 1993.

Technical reports

1. R. Veldema, R.A.F. Bhoedjang, and H.E. Bal. Distributed Shared Memory Management for Java. In *6th Annual Conf. of the Advanced School for Computing and Imaging*, Lommel, Belgium, June 2000.

2. R. Veldema, R.A.F. Bhoedjang, and H.E. Bal. Jackal: A Compiler-Based Implementation of Java for Clusters of Workstations. In *5th Annual Conf. of the Advanced School for Computing and Imaging*, pages 348–355, Heijen, The Netherlands, June 1999.

3. R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. LFC: A Communication Substrate for Myrinet. In *4th Annual Conf. of the Advanced School for Computing and Imaging*, pages 31–37, Lommel, Belgium, June 1998.

4. H.E. Bal, R.A.F. Bhoedjang, R. Hofman, C. Jacobs, K.G. Langendoen, T. Rühl, and M.F. Kaashoek. Orca: a Portable User-Level Shared Object System. Technical Report IR-408, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, July 1996.

5. H.E. Bal, R.A.F. Bhoedjang, R. Hofman, C. Jacobs, K.G. Langendoen, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. Technical Report IR-400, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, 1996.

6. K.G. Langendoen, J. Romein, R.A.F. Bhoedjang, and H.E. Bal. Integrating Polling, Interrupts, and Thread Management. In *2nd Annual Conf. of the Advanced School for Computing and Imaging*, pages 168–173, Lommel, Belgium, June 1996.

7. R.A.F. Bhoedjang and K.G. Langendoen. User-Space Solutions to Thread Switching Overhead. In *1st Annual Conf. of the Advanced School for Computing and Imaging*, pages 397–406, Heijen, The Netherlands, May 1995.