

A Study on Memory-Based Communications and Synchronization in Distributed-Memory Systems

Takashi MATSUMOTO
<tm@is.s.u-tokyo.ac.jp>

October 15, 2001



**Department of Computer Science
University of Tokyo**

7-3-1 HONGO, BUNKYO-KU TOKYO, 113-0033 JAPAN

TITLE	
A Study on Memory-Based Communications and Synchronization in Distributed-Memory Systems	
AUTHORS	
Takashi MATSUMOTO <tm@is.s.u-tokyo.ac.jp>	
KEY WORDS AND PHRASES	
memory-based communication, distributed shared memory, light-weight communication, memory-based processor, MBCF, parallel processing, PC cluster, workstation cluster	
ABSTRACT	
<p>In terms of facilities for communications and synchronization in parallel programs, the descriptive power of the shared-memory model is equal to that of the message-passing-style (send-and-receive-type) model. From the viewpoint of performance, however, the situation is different. A very convenient way of improving performance or increasing the variety of functions is to have the communication/synchronization subsystems handle information related to the memory locations at which target data are stored. In other words, communication/synchronization subsystems based on the shared-memory model are superior to message-passing-style subsystems which simply relay data from one task to another. This conclusion holds whether the subsystem is implemented in hardware or software. In this thesis, the Memory-Based Communications and Synchronization (MBCS) scheme is proposed. In this scheme, recognition and exploitation of information on the locations of target data by communication/synchronization subsystems is proposed, along with brand-new mechanisms based on this scheme. Three communication/synchronization mechanisms based on the MBCS scheme are then proposed. The first mechanism, called the Memory-Based Processor (MBP), is a hardware-implemented fine-grained communication/synchronization mechanism. The MBP is also a building-block for hardware-based distributed shared memory. The second is the Memory-Based Communication Facility (MBCF) which is a software-implemented medium-grained communication/synchronization mechanism made with off-the-shelf network hardware. The third is the the Memory-Based Processor II (MBP2), a hardware-implemented medium-grained mechanism which was designed and developed on the basis of research results on the MBCF. In this thesis, (1) brand-new functions to run on these mechanisms are proposed, (2) explanations of their behaviors and of high-speed implementation methods are given, (3) comparisons are made with other existing mechanisms, qualitative discussions are presented, and (4) experimental verification is described. Through these discussions, the effectiveness of the MBCS scheme will be made clear.</p>	
REPORT DATE	WRITTEN LANGUAGE
October 15, 2001	English
TOTAL NO. OF PAGES	NO. OF REFERENCES
131	82
ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT	
DISTRIBUTION STATEMENT	
First issue 30 copies. This technical report is available via internet WEB from http://www.is.s.u-tokyo.ac.jp/tech-reports/FILES.html .	
SUPPLEMENTARY NOTES	

**A Study on Memory-Based Communications and
Synchronization in Distributed-Memory Systems**

Takashi Matsumoto

February 2001

A Dissertation Submitted to
Graduate School of Science
University of Tokyo

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science
in Information Science

Abstract

In terms of facilities for communications and synchronization in parallel programs, the descriptive power of the shared-memory model is equal to that of the message-passing-style (send-and-receive-type) model. From the viewpoint of performance, however, the situation is different. In modern architectures for distributed-memory multicomputers, memory is an important and fundamental building-block of the nodes. It is thus directly accessed by processors that use their memory management units for protection and virtualization. Therefore, a very convenient way of improving performance or increasing the variety of functions is to have the communication/synchronization subsystems handle information related to the memory locations at which target data are stored. In more concrete terms, if communication/synchronization subsystems do so, they can use the information to reduce the number of copies of data. Moreover, these subsystems have been made capable of directly accessing data in user tasks, it becomes possible to implement advanced synchronization functions (e.g., atomic operations, queue operations and so forth), as well as simple read/write operations, within the subsystems. In short, communication/synchronization subsystems based on the shared-memory model are superior in this way to message-passing-style subsystems which simply relay data from one task to another. This conclusion holds whether the subsystem is implemented in hardware or software.

In this thesis, the Memory-Based Communications and Synchronization (**MBCS**) scheme is proposed. In this scheme, recognition and exploitation of information on the locations of target data by communication/synchronization subsystems is proposed, along with brand-new mechanisms based on this scheme. Effectiveness of the overall scheme is then shown by experimental verification and by discussion and analysis. Going into more detail, subsystems based on the MBCS scheme are classified into three categories that correspond to the different grain sizes of data within operations and on implementation methodology. A communication/synchronization mechanism for each category is then proposed. The first mechanism, called the Memory-Based Processor (**MBP**), is a hardware-implemented fine-grained communication/synchronization mechanism. The MBP is also a building-block for hardware-based distributed shared memory. The second is the Memory-Based Communication Facility (**MBCF**) which is a software-implemented medium-grained communication/synchronization mechanism made with off-the-shelf network hardware. The third is the Memory-Based Processor II (**MBP2**), a hardware-implemented medium-grained mechanism which was designed and developed on the basis of research results on the MBCF. In this thesis, (1) brand-new functions to run on these mechanisms are proposed, (2) explanations of their behaviors and of high-speed implementation methods are given, (3) comparisons are made with other existing mechanisms, qualitative discussions are presented, and (4) experimental verification is described. Through these discussions, the effectiveness of the MBCS scheme will be made clear.

Acknowledgments

I would like to thank professor Kei Hiraki, my boss at the University of Tokyo, for giving me the environment to conduct research on subjects of my choice. I also wish to thank Dr. Junpei Niwa and Dr. Tatsushi Inagaki (presently with IBM Japan's Tokyo Research Laboratory), who were students at our laboratory and have developed an excellent optimizing-compiler for my software DSM scheme. I also wish to thank Dr. Kiyofumi Tanaka for designing and developing the brand-new "Casablanca" embedded processor for the MBP2. He was willing to proceed with its development and to adopt my several ideas for its architecture.

I thank Mr. Shigeru Uzuhara of AXE Corp. for carrying out the laborious work of porting the *SSS-CORE* OS to run on ULTRA workstations. My thanks are also due to Mr. Haruyasu Oyobe of Sansei Systems Co. Ltd for manufacturing the logic board for the MBP2P and developing the logic design of its central controller.

I also wish to thank Ms. Noriko Tanaka, the very able secretary of our laboratory, for relieving me of most burdensome administrative tasks. I must also thank Ms. Akiko Shintani, a super-librarian in the Department of Information Science. She always helps us in our searches for literature. I am also indebted to the system administrators at our laboratory for their steady maintenance of the research environment. In particular, I wish to thank Mr. Kanemitsu Ootsu (presently with the University of Utsunomiya), Mr. Junji Tamatsukuri, Mr. Kenji Morimoto, Mr. Masayoshi Nomura and Mr. Ryuta Niino.

I would also like to thank the Information-Technology Promotion Association, Japan (IPA) for their financial support of the development of the *SSS-CORE* and the MBP2P. I also wish to thank Japan Science and Technology Corporation (JST) for their financial support for the development of new scheduling mechanisms in the *SSS-CORE* and for providing a 24-hour office for individual research. I must also thank Sun Microsystems K. K. and Inc. for providing specifications of LSIs on their workstations. These specifications were indispensable for me to develop the MBCF system and to construct the *SSS-CORE* OS.

I am grateful to my father, who represents my idea of the perfect scholar, and to my mother who is an ideal broad-minded Japanese mother.

Contents

1	Introduction	1
1.1	Hardware DSM	1
1.2	Software DSM	4
1.3	Memory-Based Communications and Synchronization (MBCS)	5
1.4	Contributions	6
1.5	Thesis organization	8
2	Memory-Based Processor (MBP)	9
2.1	Fine-grained hardware implementation of the MBCS	9
2.2	Outline of the Memory-Based Processor	9
2.3	Caching in the MBP system	12
2.4	Address translation in the MBP system	12
2.5	Directory-based cache scheme in the MBP system	15
2.6	The MBP's protocol-switching facility	20
2.7	Related Works	20
2.7.1	Memory Channel (Reflective-Memory)	20
2.7.2	Tempest	21
2.7.3	Flash	21
2.7.4	Shrimp	21
3	Overview of Memory-Based Communication Facility (MBCF)	22
3.1	Middle-grained software implementation of the MBCS	22
3.2	Background to the MBCF	22
3.3	Basic concepts of the MBCF	24
3.3.1	From MBP to MBCF	24

3.3.2	Protection and security mechanisms of the MBCF	25
3.3.3	Virtual and global addressing in the MBCF system	26
3.4	Illustration of the MBCF's behavior	26
3.4.1	Assumptions of network interface cards (NICs)	26
3.4.2	Illustration of MBCF_WRITE operation	28
3.4.3	Illustration of the update-cache mechanism using MBCF_WRITE	35
4	Functions of the MBCF	38
4.1	The commands	38
4.2	Options for the MBCF commands	52
4.3	The supporting commands	54
4.4	The rule for combining MBCF commands	55
5	Discussions on the MBCF	57
5.1	High-speed implementation techniques of the MBCF	57
5.2	A qualitative comparison with message-passing-style communication mechanisms	61
5.3	A qualitative comparison with the Active Message	63
5.4	Related Works	64
5.4.1	Fast Message	64
5.4.2	Active Message	64
5.4.3	SparcStation Active Message	64
5.4.4	PM	64
5.4.5	U-NET	65
6	An Implementation of the MBCF using Ethernet	66
6.1	Specifications of the MBCF/Ether	66
6.1.1	Background to the MBCF/Ether	66
6.1.2	Packet format of the MBCF/Ether	66
6.1.3	User interfaces of the MBCF/Ether	69
6.1.4	Protocol of the MBCF/Ether	70
6.2	Evaluation using SPARCstation 20s and SSS-CORE Ver.1.x	72
6.2.1	Environment used for performance evaluation	72
6.2.2	Overheads of MBCF/Ether	73

6.2.3	Peak data-transfer rate of MBCF/Ether communications	77
6.2.4	Round-trip latency of the MBCF/Ether protocol	79
6.2.5	One-way latencies of high-level commands of the MBCF/Ether protocol	82
6.2.6	Evaluation of performance in executing an application program	83
6.3	Evaluation using ULTRA 60s and <i>SSS-CORE Ver.2.x</i>	88
6.3.1	Environment used for performance evaluation	88
6.3.2	Peak data-transfer rate of MBCF/Ether communications	88
6.3.3	One-way latency of the MBCF/Ether protocol	91
6.3.4	One-way latencies of high-level commands for MBCF/Ether protocol	91
6.4	A quantitative comparison with the user-level communication mechanisms of MPPs	93
7	Compiler-Assisted Distributed-Shared Memory Schemes	95
7.1	Software DSM for the medium-grained MBCS	95
7.2	Compiler-based approach	95
7.3	My approach	96
7.4	UDSM and ADSM	98
7.5	Needs of the MBCF for UDSM and ADSM	99
7.6	Optimization for the UDSM and the ADSM	99
7.7	Evaluation of my scheme and methods of optimizations	100
7.7.1	Evaluation environment	100
7.7.2	Performance in terms of single-node execution	102
7.7.3	Performance in terms of 16-node execution	103
7.7.4	Adding nodes for increased speed	104
7.8	Related Works	104
7.8.1	Shasta	104
8	Memory-Based Processor II (MBP2)	106
8.1	Middle-grained hardware implementation of the MBCS	106
8.2	System structure of the MBP2C	108
8.3	Features of the MBP2	111
8.3.1	Architecture based on the MBCF	111
8.3.2	Protected and virtualized user-level direct I/O access	112
8.3.3	Full-DMA-connection among functional units	112

8.3.4	Embedded microprocessor which can cooperate with DMA mechanisms	113
8.3.5	Embedded microprocessor with zero-cost context switching for interrupts	113
8.3.6	Encryption hardware embedded in a NIC	114
8.4	Prototype of the MBP2C	114
8.5	Discussions on the MBP2	118
8.6	Related Works	118
8.6.1	T3D	118
8.6.2	AP1000+	119
9	Concluding Remarks	120

List of Tables

6.1	Sending overheads of MBCF_WRITE packets	74
6.2	Peak data-transfer rates for the MBCF/100BASE-TX and the MBCF/10BASE-T	77
6.3	Round-trip latency of MBCF/100BASE-TX	80
6.4	One-way latencies of high-level commands of MBCF/100BASE-TX	83
6.5	Execution times of parallel ray-tracing using MBCF/100BASE-TX	86
6.6	Execution times of parallel ray-tracing using UDP100/SunOS	86
6.7	Execution times of parallel ray-tracing using MBCF/10BASE-T	87
6.8	Execution times of parallel ray-tracing using UDP10/SunOS	87
6.9	Peak data-transfer rates of MBCF/1000BASE-SX and TCP1000/Solaris	89
6.10	One-way latency (μ sec) of MBCF/1000BASE-SX	91
6.11	One-way latencies (μ sec) of high-level commands of MBCF/1000BASE-SX	93
6.12	Basic performance of user-level communication mechanisms	94
7.1	Modifications on SPLASH-2 programs	101
7.2	Problem scale and single-node execution times (exec. time: sec, (additional overhead:%)).	102
8.1	Operational environment of the MBP2P	115
8.2	Hardware specifications of the MBP2P	117

List of Figures

2.1	System architecture using MBPs	10
2.2	Structure in one node of the MBP system	11
2.3	TLBs in one node of the MBP system	13
2.4	Address translation in the MBP system	14
2.5	Hierarchical bitmap directory (1)	16
2.6	Hierarchical bitmap directory (2)	17
2.7	Hierarchical multicasting	18
2.8	Acknowledge-message combining	19
3.1	Functions of conventional NICs	27
3.2	Illustration of the MBCF_WRITE operation (1)	28
3.3	Illustration of the MBCF_WRITE operation (2)	29
3.4	Illustration of the MBCF_WRITE operation (3)	30
3.5	Illustration of the MBCF_WRITE operation (4)	31
3.6	Illustration of a secure MBCF_WRITE (1)	32
3.7	Illustration of secure MBCF_WRITE (2)	33
3.8	Illustration of cache updating with MBCF_WRITE	34
3.9	Illustration of cache updating with secure MBCF_WRITE	35
3.10	Illustration of the MBP's update process	36
4.1	Operation of the MBCF_FIFO (1) command	41
4.2	Operation of the MBCF_FIFO (2) command	42
4.3	Operation of the MBCF_FIFOe (1) command	43
4.4	Operation of the MBCF_FIFOe (2) command	44
4.5	Operation of the MBCF_FIFOe (3) command	45

4.6	Operation of the MBCF_FIFOe (4) command	46
4.7	Operation of the MBCF_SIGNAL (1) command	47
4.8	Operation of the MBCF_SIGNAL (2) command	51
4.9	Algorithm of snoopy spin-wait (SS-wait)	54
5.1	MMU and TLB of recent processors	59
6.1	Format of a single MBCF_WRITE packet	67
6.2	Two other MBCF packets	69
6.3	Zoomed signal wave forms of a single 4byte MBCF_WRITE	75
6.4	Signal wave forms for the receiving of consecutive 4byte MBCF_WRITES	76
6.5	Peak data-transfer rates for MBCF100, TCP100, and UDP100	78
6.6	Peak data-transfer rates for MBCF10, TCP10, and UDP10	79
6.7	Round-trip latency of MBCF/100BASE-TX	80
6.8	Round-trip latencies for MBCF100, TCP100, and UDP100	81
6.9	Round-trip latency of MBCF/10BASE-T	82
6.10	Round-trip latencies for MBCF10, TCP10, and UDP10	83
6.11	Signal wave forms of a single 4byte MBCF_WRITE	84
6.12	Sample picture used for measurements	85
6.13	Peak data-transfer rates for MBCF1000 and TCP1000/Solaris using 1000BASE-SX	89
6.14	One-way latencies for MBCF/1000BASE-SX and TCP1000/Solaris	92
7.1	Effects of UDSM optimization techniques on 16-node execution	103
7.2	Effects of ADSM optimization techniques on 16-node execution	104
7.3	Speed-up ratios by the UDSM and the ADSM	105
8.1	Block diagram of MBP2C	107
8.2	Block diagram of the MBP2P	115
8.3	Circuit diagram of the NIC based on the MBP2P	116
8.4	Photo of the MBP2P NIC card	117

Chapter 1

Introduction

1.1 Hardware DSM

Recent generations of high-end microprocessors have strong and hierarchical memory systems, and cacheable memory operations are much more efficiently handled than non-cacheable operations (e.g., I/O register operations). Thus when an I/O device requires a high-bandwidth, it will include cacheable DMA mechanism. For inter-processor communications and synchronization the situation is the same as for high-performance I/O, and memory-coupling is much better than register-coupling for communicating with many processors and I/O devices. Moreover, if we adopt a single and global memory space (i.e., shared-memory), the memory-coupling approach has further advantage in that multi-tasking operating systems on single processors can easily be ported to UMA (Uniform Memory Access)-type multiprocessors. In this way, each task can be allocated to a different processor, and this simple strategy is enough to improve the overall performance of the system.

For the above reasons tightly-coupled (UMA-type) multiprocessors, which we have recently come to call SMPs (symmetric multiprocessors), became fit for practical use in the latter half of the 1980s. This was earlier than the practical application of distributed-memory, that is, NUMA (non uniform memory access) architectures multiprocessors. It is, however, very difficult to make large-scale tightly-coupled multiprocessors because a mechanism that efficient access to main memory by multiple different processors is almost impossible. To attain a scale-up of the number of processors, the focus of research gradually shifted to distributed-memory multiprocessors. In order to obtain the benefits of shared-memory, even from distributed-memory multiprocessors, distributed-shared-memory (DSM) mechanisms became the subject of extensive study from the end of the 1980s.

Although snoop cache [15, 3] mechanisms are usually adopted for small-scale tightly-coupled multiproces-

sors, directory-based caching schemes are used in distributed-memory multiprocessors with hardware-based DSM mechanisms. This is because snoop cache mechanisms require some method for the broadcasting of memory transactions to all processors in the system to maintain consistency of the caches. Historically, the directory cache[73] was invented much earlier than the snoop cache, but early directory caches were used with UMA-type high-end multiprocessors (e.g., main frames). In such machines, all memory modules have their own directories, and the memory modules are symmetrically connected to the processor elements via switching networks. This type of directory cache is not very different to the snoop cache, and the only difference is that snoop caches exploit the merits of the bus connections between processors and main memory. At the end of the 1980s new types of directory caches [31, 1, 6] were proposed and developed for use with large-scale DSM machines. To ensure bandwidth between the processor (or a few processors) and main-memory module, new machines handle the processor(s) and memory module as a combined building block of the system. We call such a building block a **node** of the system. The overall main memory consists of memory of all memory modules in all nodes, and directories for the DSM mechanism are also distributed to all nodes. For main memory modules, every block, per cache-block size, is accompanied by a state tag and a directory which represents either nodes that are keeping copies of the block or the original (home) node of the block. When processors write to a shared block in main memory, some consistency-maintenance action is taken by the DSM mechanism. This represents a revival of directory-caching schemes. In most DSM systems, invalidate-type consistency protocols are adopted to maintain coherence because of the resulting reduction in network traffic. Hardware DSM mechanisms realized a logical shared-memory space with remote caching on multiprocessors with distributed memory, but the cost of remote access was too great to improve performance by parallel execution. In particular, invalidate-type protocols are very poor at such method of optimization as latency-hiding by overlapping phases for calculations and phases for communications. Moreover, when operating systems designed for tightly-coupled multiprocessors are ported to DSM machines without modification, it is not possible to improve performance by allowing a loss of access-locality within the node. DSM systems create a need for new operating systems that take account of the distribution of data and the distribution of tasks to minimize inter-node communications. Furthermore, a large quantity of high-speed tag-memory modules must be added to main memory so that the hardware DSM mechanisms can operate. These tag modules and the dedicated and special controller for the DSM raise the cost of DSM systems. For all of these reasons, hardware DSM systems have yet to become standard or popular.

A major by-product of research into hardware DSM was a deeper understanding of memory-consistency models. Hardware DSM mechanisms have been unable to attain negligible latencies for remote accesses, and the stalling of processors during these latencies dramatically worsens system-performance, processors were

allowed to perform the subsequent memory accesses before in-progress memory accesses had been completed. This mitigation of the memory-access rule results in a modification of the memory-consistency model, and is not possible without special conditions. Before the resulting research into memory models, processors had only been able to perform subsequent memory-access after all in-progress accesses had finished. In such a memory-access model, it is never possible for other processors to detect situations in which one in-progress access finishes while some previous accesses have not been completed. Even when a processor issues a subsequent access without waiting for all in-progress accesses to be completed, it is a good enough condition that other processors are not able to detect any inconsistencies of order. However, in DSM systems with remote caches, it is very difficult to ensure the order of memory-access operations from the viewpoint of external processors if a processor does not wait for all preceding accesses to be completed. Therefore, the memory models which had been expected by programmers were modified to mitigate their requirements. Memory operations on shared data are separated into those for data transfer and those for synchronization, and there are then no restrictions on the order of memory-access operations that correspond to data transfer. On the other hand, there are some restrictions (order-relations) between operations that are for data transfer and those that are for synchronization, or among those that are for synchronization. This is the basic idea, in outline form, of relaxed memory-access models (e.g., Release Consistency model[13], Weak Consistency model[9]). After the invention of relaxed memory models, large-scale parallel applications have gradually come to be written around these models.

Even if programmers use relaxed memory models to write shared-memory parallel programs, DSM machines were not superior to these distributed-memory machines that used message-passing-type communication mechanisms, in terms of their performance of on large-scale parallel applications. Therefore, at the start of the 1990s data-parallel computation or SPMD (single program multiple dataflow) computation on the simple distributed-memory machines then in existence became very popular in the field of scientific computation. The problems are cast in an explicitly and massively parallel form. Thoughtless researchers who saw this situations concluded that DSM machines are fundamentally inferior to simple distributed-memory multiprocessors. In fact, at the start of the 1990s, DSM multiprocessors were expensive but inefficient in the parallel execution of applications. In a sense, however, the comparison is not fair because fully-optimized code was used on the distributed-memory multiprocessors while the DSM multiprocessors of those days had, as yet, no facilities for code optimization. The biggest weak point of the DSM multiprocessors of those days was that each system has had a single and global protocol to maintain the consistency of shared memory. As I mentioned before, most such systems only included invalidate-type protocols, but invalidate-type protocols are very poor in terms of optimization for eager transmissions. Some multiprocessors had only

update-type protocols, but their performance was, in general, much worse than the performance of machines with invalidate-type protocols alone. This is because update-type protocols destroy the memory-access localities of the application program which are the source of the speed increase provided by the hierarchical memory/cache system. At the end of the 1980s I insisted that a form of dynamic protocol switching would be effective even in tightly-coupled multiprocessors [38, 40, 50], and, at the start of the 1990s I also claimed that dynamic protocol switching, according to the data-types, is much more effective in DSM multiprocessors than in tightly-coupled machines [34, 45]. If a DSM machine has multiple protocols (e.g., invalidate-type, update-type, simple remote-write without remote caching, and so forth) and a facility to switch protocols per page (memory block) or per access, the optimization techniques which had been applied to distributed-memory machines can also be applied to a DSM machine. I proposed such a hardware DSM machine, the Memory-Based Processors (MBP) [34] system in 1992. Aside from the protocol-switching facility, the MBP has various features to reduce overheads on the DSM and to lower the hardware cost of the system. The details are given in Chapter 2.

1.2 Software DSM

In 1988 K. Li developed a new software solution for the generation of shared virtual spaces on clusters of PCs [32] by exploiting the memory-page management mechanisms of the processors. In recent generations, high-end microprocessors have included page-management units, and illegal access to pages which are not mapped or allowed for the allocated task (process) is detected by page-traps. K. Li exploited this mechanism to detect shared-write access to shared pages and shared-read access to invalidated pages without using any dedicated hardware. The software DSM method was thus established by this invention, and this trap-based (or OS-based) software DSM was widely accepted as a way of constructing shared-memory spaces on distributed-memory multicomputers without any hardware support for DSM.

However, K. Li's software DSM, which was called "IVY", carried large software overhead in the form of trap-handling routines. This was largely because of the larger overhead of communications in those days. There was also another problem with the IVY approach in that cache-block sizes were limited to the same size as a page or larger. This was too large to avoid false-sharing or wasteful transmissions. At the start of the 1990s, the concept of relaxed memory models was introduced to software-based DSM mechanisms, to relieve the systems of such overheads [26]. Relaxed memory models were very useful in allowing overlaps between and within communications and calculations, and thus providing improved performance.

If we adopt new relaxed-memory models, application codes may have to be rewritten and will at least have to be recompiled. In other words, the exploitation of relaxed models by software-based DSM schemes

means that binary compatibility with tightly-coupled multiprocessor systems must be abandoned. Mature consideration leads us to the conclusion that binary compatibility is, in any case, useless for or irrelevant to parallel processing because of the absence of standard architectures. For this reason, I discarded the convention that trapping on load/store operations to shared pages should be exploited in software DSM schemes. In the middle of the 1990s I proposed a brand-new approach to the software-based DSM [51, 48]. In my approach, user-level codes for consistency management are inserted into the program code at compile time, and thus included in the executable files, and both the original code and the inserted codes are thoroughly optimized to reduce overheads and the need for remote communications. Details of my approach is described in Chapter 7.

1.3 Memory-Based Communications and Synchronization (MBCS)

From 1994, when the *SSS-CORE* project [36] started, I began to develop kernel-level software to emulate the MBP's functions. The result was the Memory-Based Communication Facility (MBCF) [46, 52, 47]. Through the development of the MBCF, I came to notice the following significant fact. The biggest advantage of a DSM scheme (or a global shared addressing scheme) is that information on the target locations is exploited in the communication/synchronization subsystem, whether the subsystem is implemented in hardware or software. Conversely, the use of logical global addresses is essential for the communication/synchronization subsystem. I call this methodology the Memory-Based Communications and Synchronization (MBCS) scheme. The MBCS scheme has following strong points comparing with conventional message-passing-style communication schemes.

1. The communication/synchronization subsystem can directly process the target locations and consequently reduce the number of copies of data that must be generated.
2. The communication/synchronization subsystem can provide users with a wide variety of functions because of the capability of specifying the target locations by users.
3. The communication/synchronization subsystem can furnish the facility of resource protection by exploiting conventional memory management schemes.
4. A high-speed remote-data cache mechanism can be implemented accompanied with the MBCS scheme, because a direct manipulation on cache-memory locations is suitable for cache subsystems.

From the viewpoint of the MBCS scheme, the MBP is a fine-grained hardware implementation, and the MBCF is a medium-grained software implementation. Owing to its purely software implementation, the

MBCF is able to use commodity network hardware (e.g., ethernet cards and cabling). In spite of this use of commodity network items alone, recent generations of high-performance processors have been able to attain very good performances in communications and synchronization according to the MBCS scheme. The big problem with the MBP approach is that the fine-grained communications induced by the MBP are inefficient in transmission on the buses of a node or over a network. Since off-the-shelf medium-grained network hardware is controlled by software in the MBCF, the grain-size used in communications can be enlarged to reduce overheads and to more efficiently utilize the network or buses.

Since the MBP was originally invented as a advanced hardware DSM mechanism, it naturally provided users remote-cache capability. The MBCF, however, isn't a representation of any DSM schemes but a software implementation of the MBCS scheme. In order to construct efficient remote-cache facility with the MBCF, a brand-new approach on software DSM was required. The new approach is a compiler-assisted DSM schemes which is briefly mentioned above subsection on software DSM and is described in Chapter 7.

Recently I have developed another hardware implementation of the MBCS scheme. We call it the Memory-Based Processor II (MBP2). The architecture of the MBP2 is improved by adopting the MBCF approach. The MBP2 is a medium-grained hardware implementation of the MBCS using commodity network items. The MBP2 can relieve main processors of a large part of their loads on communications and synchronization.

1.4 Contributions

The main contributions of this work are listed below.

1. Memory-Based Processor (MBP)

- An address translation system for remote memory accesses

The MBP system was the first to introduce an address-translation system for protected memory-based communications.

- The hierarchical multicasting and acknowledge-message combining method

This method of the MBP system resolves the hot-spot problem in the collection of acknowledge-messages at a home node. This method is a key technology for the construction of large-scale coherent DSM systems.

2. Memory-Based Communication Facility (MBCF)

- The MBCF method

Various functions that exploit information on target locations have been proposed in the MBCF scheme, and various implementation techniques, in terms of both software engineering and advanced processor architectures, were introduced to develop the protocol stack for the MBCF.

- Proof of the qualitative superiority of the MBCF

The MBCF is qualitatively superior to message-passing-type interfaces and to the SparcStation Active Message. In other words, the MBCF-style interface is fit for higher-speed implementation than the other interfaces.

- Design of the MBCF/Ether protocol

A protocol to provide the MBCF over ethernet connections was designed. This new packet protocol was necessary to guarantee the arrival and order of packets. A new window-based (GO-back-N) technique, in which N is dynamically decreased at a hint of a lack of buffers, was also adopted for the protocol. The technique ensures that the system is both high performance and scalable.

- Implementation of MBCF/Ether-based system

The MBCF/Ether was actually implemented and refined on clusters of workstations, in this case SPARCstation 20s and Ultra 60s, with the *SSS-CORE* operating system.

- Experimental verification of MBCF/Ether's performance

Clusters of workstations with 100BASE-TX or 1000BASE-SX were used to evaluate the MBCF/Ether system's performance. The results were a one-way latency of in $24.5\mu\text{sec}$ and data-transfer rate of 11.93Mbyte/sec for 100BASE-TX and a one-way latency of $9.6\mu\text{sec}$ and data-transfer rate of 80.92Mbyte/sec for 1000BASE-SX. Note that communications during the measurements were between two user-level tasks and the processing times and round-trip times were measured from within a user-level task.

3. The compiler-assisted software DSM approach

A brand-new approach to the software DSM is proposed in this thesis. In this approach, user-level codes for consistency management are inserted in the program code at compile time, and thus included in the executable modules, and both the original code and the inserted coded are thoroughly optimized to reduce overheads and the need for remote communications. This approach is revolutionary and abandons the trap-based method of conventional software DSMs. This approach allows workstations with off-the-shelf network-interface cards (e.g., 100BASE-TX ethernet) to efficiently support shared-memory parallel programs.

1.5 Thesis organization

The Memory-Based Processor (MBP), my first hardware implementation of the Memory-Based Communication and Synchronization (MBCS) scheme, is described in Chapter 2. Chapter 3 introduces the Memory-Based Communication Facility (MBCF), a software implementation of the MBCS scheme which requires only commodity machines (personal-computers and workstations). Chapter 3 mentions some important concepts of the MBCF, including the differences between the MBCF and the MBP, protection and security mechanism, and its virtual and global addressing system. Chapter 3 also illustrates behavior of several MBCF commands to convey a more detailed understanding of the MBCF. Chapter 4 presents variations of the MBCF functions. The wide variety of the MBCF functions is a strong point of the MBCF scheme. Chapter 5 discusses the high-speed implementation techniques and qualitative aspects of the MBCF. There are reasons why the MBCF scheme is superior to the message-passing-type communication interfaces and the Active Message scheme. Chapter 6 describes specifications of the MBCF/Ether which is an implementation of the MBCF scheme to ethernet systems. Chapter 6 also shows performance evaluations of the MBCF/Ether. Chapter 7 describes compiler-assisted distributed shared memory schemes for the medium-grained MBCS. I propose two brand-new DSM schemes: UDSM and ADSM which are supported by our optimizing compiler: RCOP. Chapter 8 presents Memory-Based Processor II (MBP2) which is a medium-grained hardware implementation of the MBCS. In this chapter I introduce a new architecture of the MBP2 and discuss its advantages and disadvantages. Chapter 9 concludes the thesis.

Chapter 2

Memory-Based Processor (MBP)

2.1 Fine-grained hardware implementation of the MBCS

The Memory-Based Processor (MBP) [41, 34, 45, 44] is first hardware mechanism for the Memory-Based Communications and Synchronization (MBCS) scheme. The MBP is a kind of co-processor for conventional high-end microprocessor and it is attached in memory modules. It realize remote memory operations (including atomic operations), memory-based synchronization, and memory consistency protocols. Since the system with MBPs was originally designed as an advanced implementation of hardware DSM system, the size of data which an MBP handles at a time is equal to the size of data which the main-processor accesses. Consequently the grain-size is the cache-block size of the main-processor. Though this size is too fine to mitigate the overhead cost for communications, there is no other way to keep characteristics of hardware DSMs.

The MBP was designed not only as a building-block of a hardware DSM system but also as a MBCS mechanism in which communications and synchronization are handled as kinds of logical memory operations. I represent outline of the MBP system and key technology of the MBP in this chapter. The MBP's functions on communications and synchronization are almost same as the MBCF's functions which is described in Chapter 4, the differences of two parties only are on grain sizes. The MBP has multiple consistency protocols [44] to construct the hardware DSM space, and they can be mixed to use without inconsistency. These consistency protocols are not described in this thesis because they are irrelevant to the MBCS scheme.

2.2 Outline of the Memory-Based Processor

The MBP is a novel device for the realization of protected and virtualized high-speed user-level communications and synchronization in NUMA systems. Figure 2.1 shows the architecture of a system in which

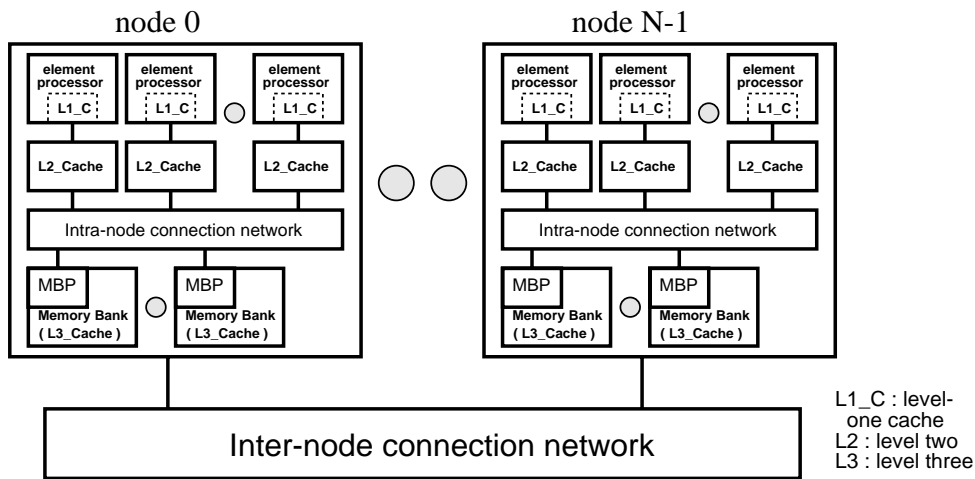


Figure 2.1: System architecture using MBPs

MBPs are used and Figure 2.2 illustrates the structure of a node of the system. In this system, each set of strongly-cooperative threads with fine-grained concurrency would be allocated to a single node, while tasks might be scheduled at more than one node for a more coarsely-grained parallelism. The MBP is attached to each memory bank on the lowest level of the memory hierarchy and it performs activities in which there are no useful localities on memory references for main processors to improve performance. Therefore, the system is equipped with multiple MBPs that can access the memory-modules quickly and directly. As you see from Figure 2.1 and Figure 2.2, the MBP architecture is a pioneering processor-in-memory (PIM) architecture. Using RISC processors as the main processing elements of the system provides the best performance in computation with access-locality on memory references. This is because the RISC processors integrate large amount of resources (registers and cache memory) which exploit access-locality to improve performance. For short-threaded fine-grained computation without access-locality the characteristics of the MBP complement those of RISC processors [41].

The MBP receives command messages that originate from the main processing elements (conventional RISC microprocessors) as main-memory operations (i.e. outgoing load/store operations). MBPs are responsible for data transmission among the memory banks, management of the consistency of the caching system, various memory-based synchronization mechanisms [34], page management of virtual shared memory, the management of context in the main processing elements, and so on. In this context, MBPs work to maintain memory consistency among nodes and to execute inter-node communications.

The MBPs in a system make up a high-performance/high-functional distributed shared memory space for a NUMA system [34, 44]. In a system with MBPs, memory-management units (including TLB mechanisms)

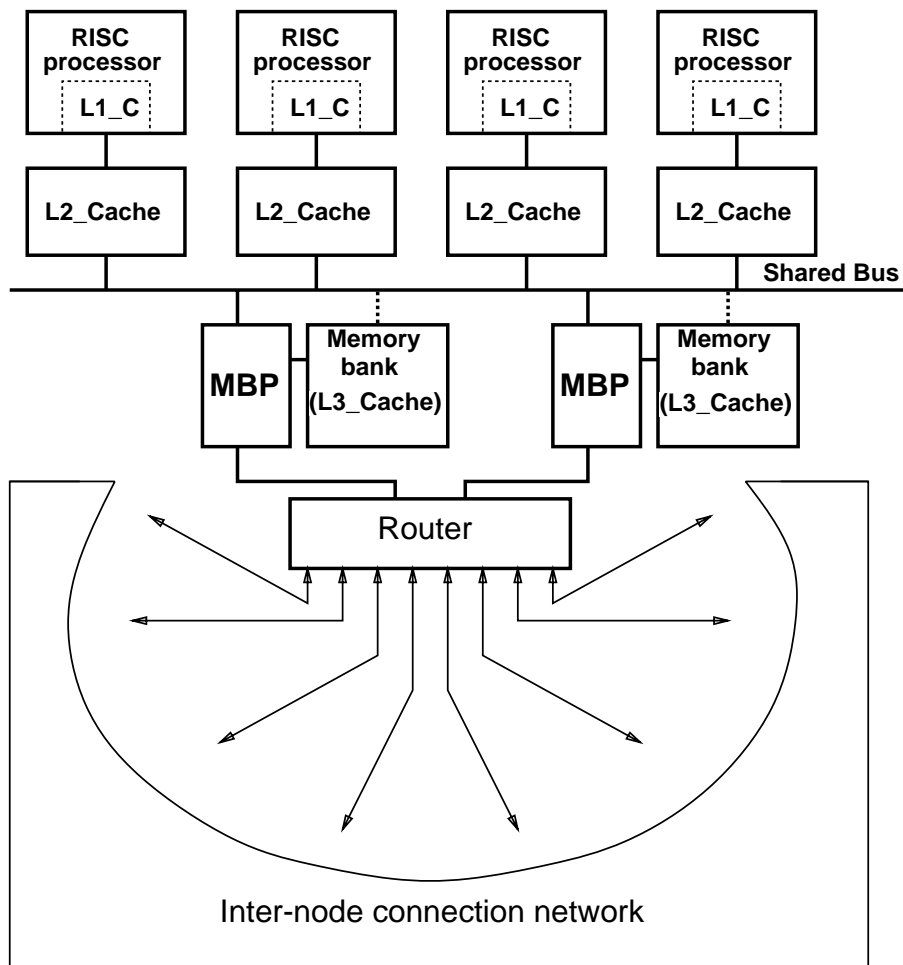


Figure 2.2: Structure in one node of the MBP system

are extended to support facilities for remote-memory access, and the system provides users with the same fine-grained memory interface as an SMP. The MBPs also support fine-grain cache-consistency protocols of both the invalidation types and the update types, and include various memory-based synchronization primitives.

2.3 Caching in the MBP system

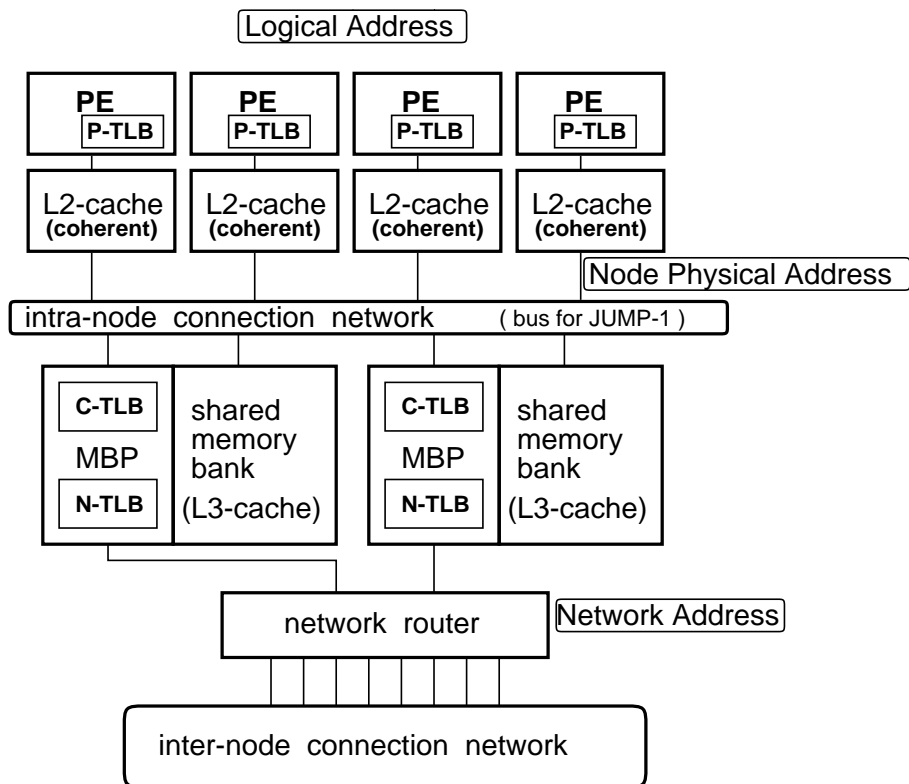
The memory banks are used as caches of the data from remote nodes that is required by inter-node memory references and as main memory for local data. The system thus has hierarchical cache system: the on-chip caches of the processors (L1 caches) are at level 1, ‘snoop caches’ (L2 caches) are at level 2 and the memory banks of the nodes are at level 3. Page-level directory schemes such as IVY[32] were used in the MBP systems instead of cache-block-level directory schemes [31], in order to reduce the amount of directory memory and to use translation look-aside buffers (TLBs) to accelerate consistency-preserving operations. The unit of data transmission, however, is the size of an L1 cache block instead of that of a memory-page corresponding to a TLB entry. This is to reduce the amount of wasteful data traffic that arises in the case where the unit of transmission is the page. Each block in memory banks has state tags that are used to maintain consistency and reduce the amount of bus traffic in each node. A **pseudo-fullmap** [34, 44] method was adopted for the inter-node directory scheme. This method assumes that the inter-node network has a hierarchical structure and reduces not only the cost (in terms of time) of maintaining cache consistency but also the amount of space taken up by directory entries. I will describe this directory scheme later.

2.4 Address translation in the MBP system

Fully virtual shared memory is implemented with a two-stage address translation scheme. In a clustered system, there are three classifications for address spaces. The first is the logical space in a main processing element, and this is called a ‘logical address’ space. The second is the physical address space of a node, and is called a ‘node-physical address’ space. The third is the network-wide logical space, and is called a ‘network address’ space. The network address space is the unique virtual shared-memory space of the system.

Memory protection for the global network address space is much the same as in the conventional MMU/TLB system. In other words, page-fault traps are used to detect access violations by processing elements.

The TLB in a main processing element is used for the high-speed translation of logical addresses to node physical addresses. A cluster-translation look-aside buffer (**C-TLB**) and a network translation look-aside buffer (**N-TLB**) have been implemented in the MBP to accelerate translation of addresses and management



MBP : Memory-Based Processor
C-TLB : Cluster TLB P-TLB : Processor TLB
N-TLB : Network TLB

Figure 2.3: TLBs in one node of the MBP system

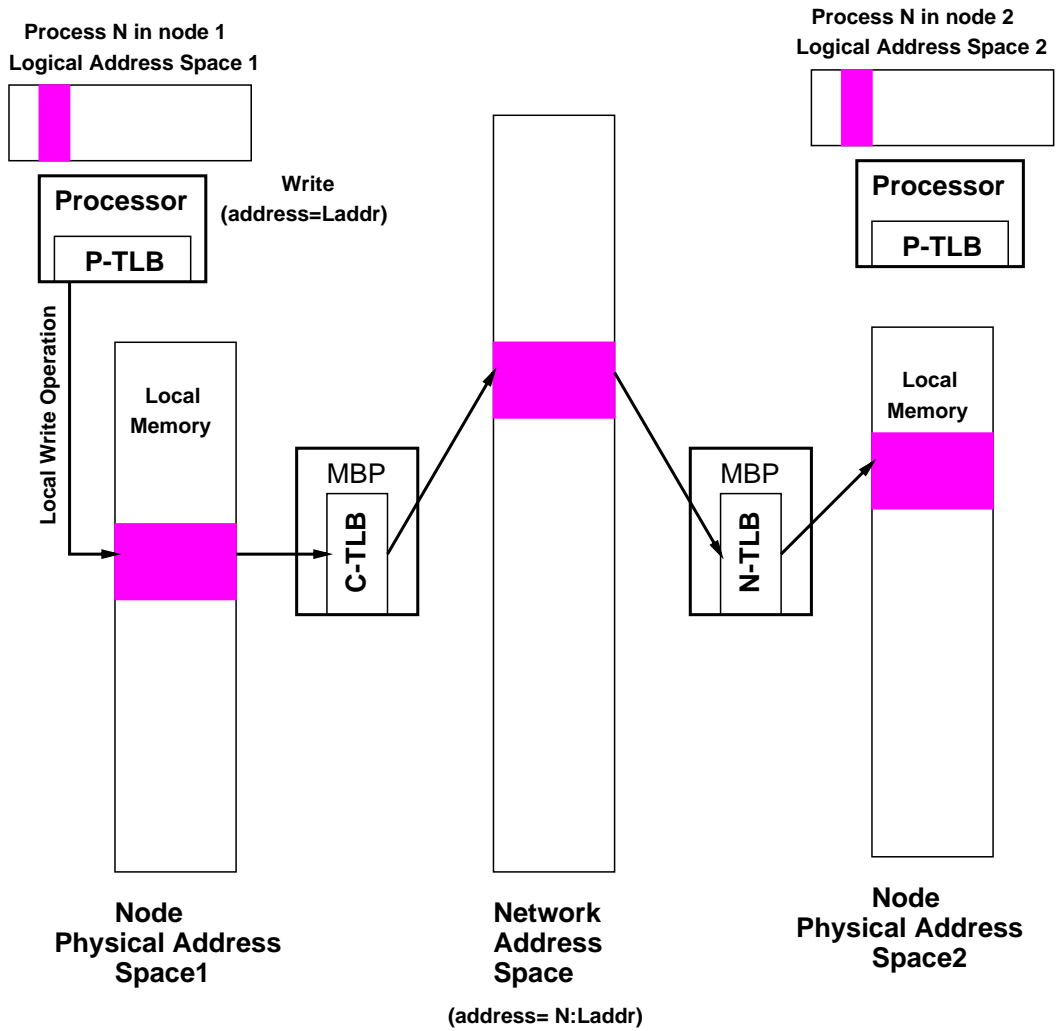


Figure 2.4: Address translation in the MBP system

of consistency (refer to Figure 2.3). The C-TLB translates the node-physical address in a cluster to a network address and has a room in its entry for caching the associated directory of the L3 cache system, and thus raising the system's performance. C-TLBs always monitor the bus transactions in a node and check operations in terms of requirements for inter-node consistency. If it is needed, the MBP is invoked as needed to maintain consistency. The N-TLB checks messages from external nodes, translates network addresses to node-physical addresses, and invokes the MBP for processing of operations at the target node-physical address. Each time a location in memory is modified or fetched by external nodes, the state tags of the target node-physical address are checked. Bus transactions are then performed as required to maintain consistency in the node.

My two-stage address-translation scheme enables update-type protocols and makes the use of main memory for L3 caching feasible (refer to Figure 2.4). This is why it is possible to use the different node-physical addresses in the respective nodes for one shared memory location (one network address) and thus to have translation between network addresses and physical addresses take place locally. The MBP is the earliest mechanism to have a two-stage address-translation scheme that is extended to cover remote-memory access.

2.5 Directory-based cache scheme in the MBP system

The MBP also introduces two brand-new techniques on directory-based cache schemes. One is the "hierarchical bitmap directory" which reduces the amount of memory required to store directory entries, and another is the "hierarchical multicasting and acknowledge-message combining" which resolves bottlenecks that arise in the transmitting of multicasting-packets and processing of acknowledge-packets at source nodes.

The directory system, the MBP's "pseudo-fullmap directory", has three type formats which can be selected according to the situations of individual applications. The hierarchical bitmap is one of the three types and is suitable for large-scale shared-memory systems.

My directory scheme exploits the hierarchical structure of the system and the hardware routing technology, and is therefore based on two assumptions. One is that tree-type network can be efficiently emulated on the interconnection network of the system. The other is that the system has intelligent hardware routers which can support the "hierarchical multicasting and acknowledge-message combining" method.

The basic idea of the hierarchical bitmap directory is explained with the aid of Figure 2.5 and Figure 2.6. We assume that there is a quad-tree-type inter-node connection network in the system. Figure 2.5 shows a network with 16 nodes and the patterns of directory bitmaps at the node "S". In this figure, an alphabetical character represents a node of the system and the node indicated by "S" is the node I will focus in describing the bitmap patterns. The quad-tree network has a two-layered structure and five routing points. The highest-

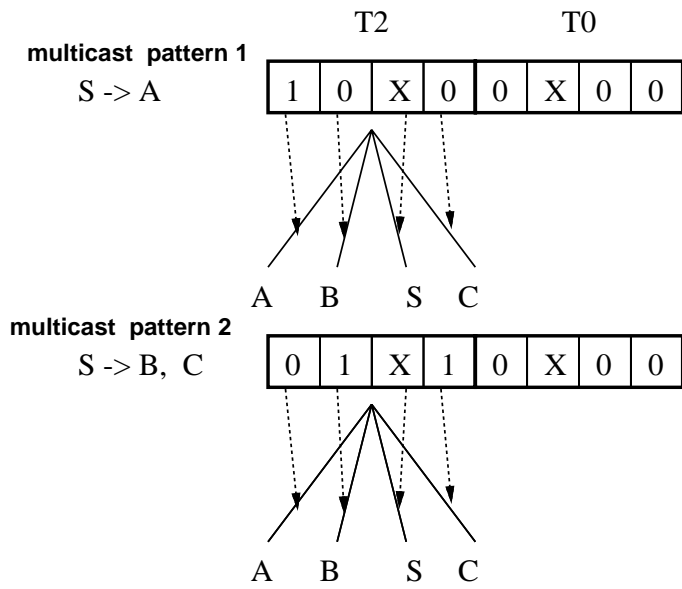
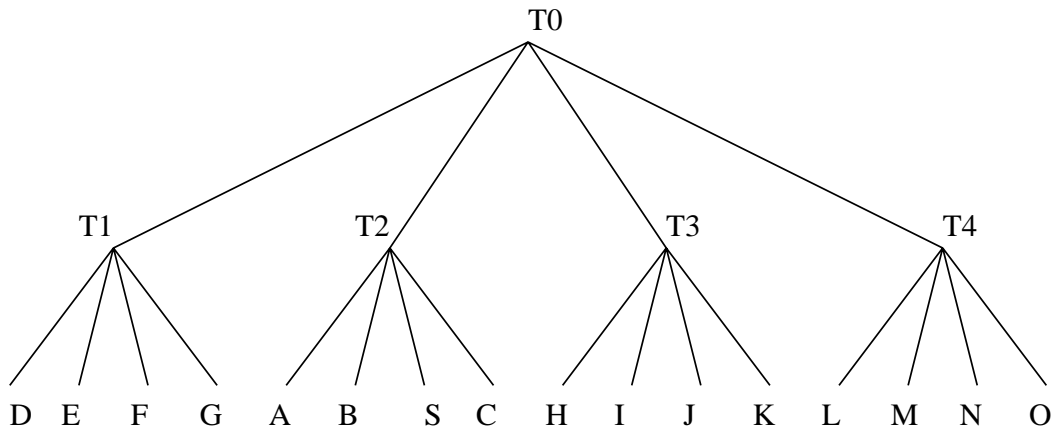


Figure 2.5: Hierarchical bitmap directory (1)

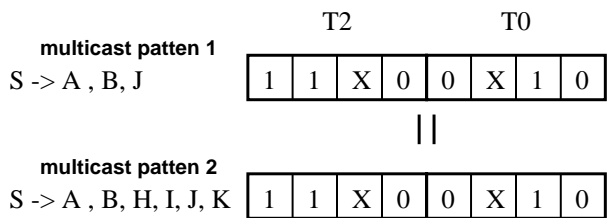
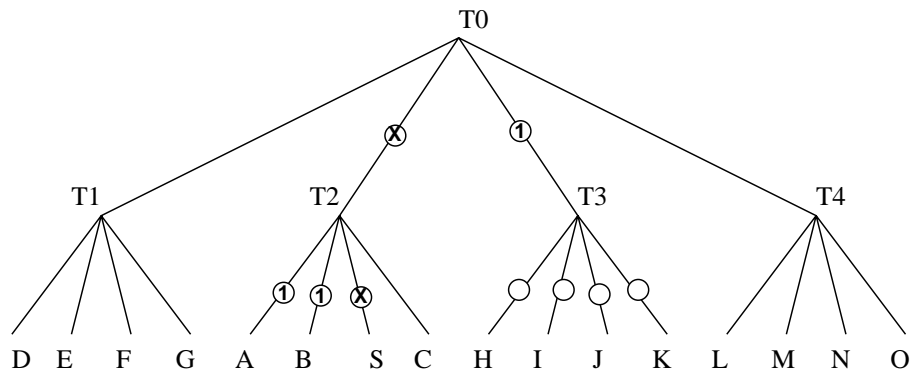


Figure 2.6: Hierarchical bitmap directory (2)

level routing point is the global root “T0” and I name the other 4 points as “T1”, “T2”, “T3”, and “T4” respectively. There are two routing points: ”T2” and “T0” between the node “S” and the global root. The hierarchical bitmap at the node “S” consists of the bitmap that indicates selections at “T2” and the bitmap that indicates selections at “T0”. Each bit of the bitmap for “T2” or “T0” corresponds to a link (branch) of a subtree of the network. A “1” in the bitmap means that child node(s) of the corresponding link is enrolled in the map. Figure 2.5 shows two bitmap patterns which represent simple cases in which each selected links lead to only single nodes. Note that the “X” bit corresponds to links which are routes to the global root from the node “S” and the values in these bits are irrelevant to the hierarchical bitmap scheme, and thus these bits can be omitted in actual implementations. Figure 2.6 shows a bitmap pattern which includes a selected link corresponding to multiple nodes. In this figure the third link of “T0” is activated and this means that the map includes 4 nodes: “H”, “I”, ”J”, and “K”. The bitmap does not indicate which of these 4 nodes are really intended for inclusion in the map. This lack of information enables us to reduce memory sizes required for directories. I allow unnecessary communications because of lack of information in the hierarchical bitmap scheme, and each node dynamically decides whether a reaction is really required of the system as each packet is received. The actual decisions are based on whether or not there are cached copies of some relevant data at the respective nodes.

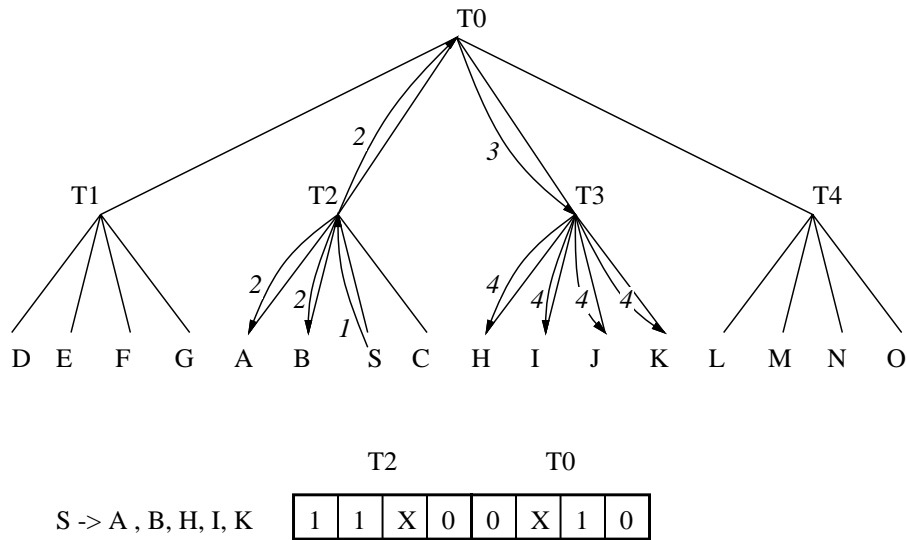


Figure 2.7: Hierarchical multicasting

Figure 2.7 illustrates the action of hierarchical multicasting using the hierarchical bitmap directory. An update-message or an invalidate-message is transmitted from the node “S” to the nodes “A,””B,””H,””I”, and “K”, all of which have cached copies of data from “S”. The numerical characters in the figure represent the order of actions.

1. A message-packet from “S” reaches the routing point “T2”.
2. The router at “T2” multicasts the packets to three directions: to the node “A”, to the node “B” and to the parent routing point “T0”. Two of these directions (to “A” and to “B”) correspond to the bitmap for “T2”. The uplink of the packet to “T0” is required because of the non-zero bitmap for “T0”.
3. The router at “T0” multicasts the packet in only one direction (to “T3”), in which lies the nodes “H”, “I”, “J”, and “K”. This direction corresponds to the “1” in the third bit of the bitmap for “T0”.
4. The router at “T3” multicasts the packet in all directions (i.e. it broadcasts the packet). The hierarchical bitmap does not contain information for selecting directions at “T3”.

After these actions 6 nodes “A”, “B”, “H”, “I”, “J”, and “K” have received packets from “S”. The packet received at the node “J” is unnecessary and is ignored because there is no cached copy of the data to be updated at this node. The large-scale multicasting mechanism can thus be efficiently implemented using the hierarchical multicasting.

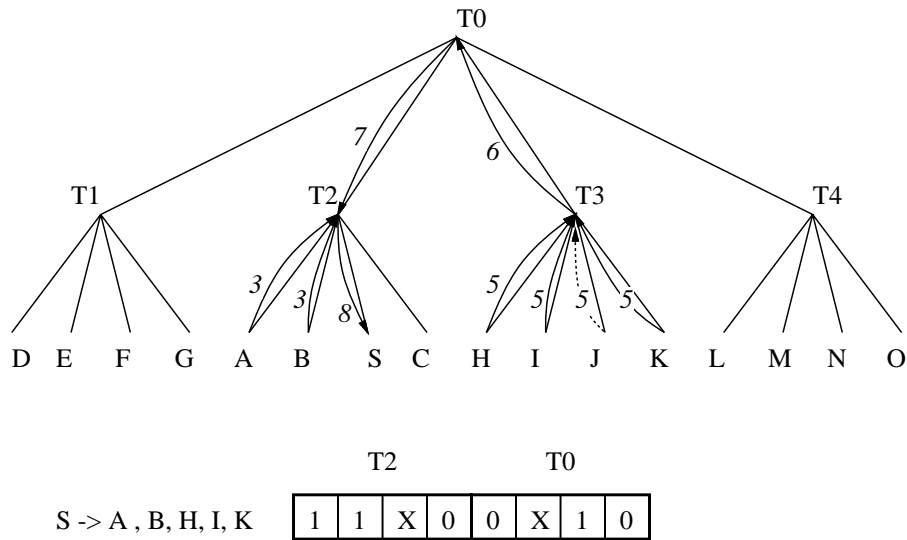


Figure 2.8: Acknowledge-message combining

Figure 2.8 illustrates action of an “acknowledge-message combining” operation. The “acknowledge-message combining” method complements the “hierarchical multicasting” method, and it is used for the acknowledge-reply phase of operation in response to a multicast-message. In the figure I assume the same situation as in Figure 2.7. The numerals in Figure 2.8 represent the order of actions and in sequence from the numerals in Figure 2.7.

1. “A” and “B” reply acknowledge-messages to the routing point “T2”.
2. No action. The router at “T2” waits for the arrival of the acknowledge-message from the parent nodes direction.
3. 4 nodes “H”, “I”, “J”, and “K” reply acknowledge-messages to the routing point “T3”. More accurately, “J” replies with a dummy acknowledge-message, as no action is taken at that node.
4. The router at “T3” has now received acknowledge-messages from all of its child-node directions. It then combines the four messages, and sends this acknowledge-message in the direction of its parent node.
5. The router at “T0” has received the single acknowledge-message that corresponds to the single message sent from “T0”, and replies with an acknowledge-message to “T2”.
6. The router at “T2” has finally received the acknowledge-message from “T0”, and it now combines the

three messages (from “A”, “B”, and “T0”) , and replies, with a single acknowledge-message, to the requesting node “S”.

The node “S” only receives a single message to acknowledge the successful multicasting of the original invalidate-message or update-message to multiple target nodes. The “acknowledge-message combining” technique can thus dramatically reduce overheads in terms of the processing of acknowledge-messages at a requesting node.

2.6 The MBP’s protocol-switching facility

There are several consistency protocols in the MBP system: invalidate-type protocols, update-type protocols and non-cacheable remote-memory operations. They can be dynamically switched at any time without carrying inconsistencies. Using these protocols, an MBP switches between consistency management strategies for directory-based L3-caches on the basis of the attributes that specify data-access types and are attached to each entry of the TLB and/or the C-TLB [45]. Maintaining perfect inter-node consistency for all access patterns is a costly operation and dedicated protocols for maintaining limited access patterns are feasible under the protocol-switching mechanism. Therefore, to improve system performance, I have defined and use new protocols which do not generally produce consistency but are consistent with some restrictions on application code [34]. The variant protocols in the MBP clustered system are illustrated in the reference[44].

2.7 Related Works

2.7.1 Memory Channel (Reflective-Memory)

The Memory Channel [14] is a hardware mechanism for fine-grained communications and is based on the Encore’s Reflective-Memory Multiprocessor [12, 33]. DEC’s sever machines adopted this mechanism for inter-node communications. This mechanism is very similar to the basic mechanism of the first MBP. In particular, the address-translation scheme and message-invocation scheme are identical. The MBP was, however, earlier than either the Memory Channel or Reflective-Memory. Moreover, the MBP is greatly superior to these mechanisms, because the Memory Channel or Reflective-Memory has neither an efficient scheme for collecting acknowledge-messages to match the acknowledge-message combining of the MBP nor any invalidate-type protocols to maintain consistency.

2.7.2 Tempest

The Tempest [63] is a hardware mechanism for detecting access by processors to actual shared blocks in a shared memory space. A similar mechanism is included in the MBP, and the MBP was both designed and reported on in published work much earlier than was the Tempest. The difference between the Tempest and the MBP is the mechanism for sending messages. The Tempest generates interrupts to the main processors when some communications is needed to maintain consistency, and the main processors send the required messages from the corresponding interrupt routines. The MBP performs fine-grained communications by using a dedicated hard-wired circuit. The Tempest detects fine-grained access by the main processors, so interrupts occur too frequently to allow an improved performance for the system as a whole.

2.7.3 Flash

The Flash [29] is the successor to the DASH [31, 30] multiprocessor and includes an integrated protocol processor which detects fine-grained access by the main processors to shared blocks and handles the communications required to maintain consistency. Its basic functions are thus similar to those of the MBP, but the Flash has no virtual network-address space and is not able to exploit the main memory as a level-3 cache. The Flash only has an invalidate-type shared-memory protocol, so calculations and communications cannot be allowed to overlap in its shared-memory scheme. The Flash has a facility, therefore, for message-passing-type communications. With the MBP or the MBCF, however, this problem was successfully overcome by using update-type protocols or remote-writes within the frame work of the shared-memory.

2.7.4 Shrimp

The Shrimp [4] is for the hardware support of fine-grained update-type shared-memory traffic, such as that which occurs in Reflective-Memory (i.e., the Memory Channel). The Shrimp is very simple hardware and can only generate and transfer one remote packet on each shared-access of main processors. The Shrimp mechanism is thus a subset of the Memory Channel and so in turn is a subset of a subset of the MBP. The Shrimp is the latest of these devices but has the lowest level of functionality. The Shrimp is not an interesting device in comparison with the others, but the team which developed the Shrimp developed some interesting shared-memory protocols (e.g., AURC [22]) which can be applied to the Shrimp or to compiler-based DSM schemes.

Chapter 3

Overview of Memory-Based Communication Facility (MBCF)

3.1 Middle-grained software implementation of the MBCS

The Memory-Based Communication Facility (MBCF)[46, 52, 47] is second implementation of the Memory-Based Communications and Synchronization (MBCS) scheme and first software one of the MBCS. Since conventional off-the-shelf network interface cards (NIC) are exploited in the MBCF system, grain size of communications is larger than that in the MBP system. The size of data which the MBCF system handles at a time is equal to the size of data-packet which the NIC transfers. Consequently the MBCF is a medium-grained (or coarse-grained) software implementation of the MBCS. Though this grain-size is suitable for mitigation of the overhead cost for communications, it is too large to realize a remote-cache subsystem. Therefore I needed to invent a brand-new software DSM approach for the MBCF. This new software DSM approach will be shown in Chapter 7.

In this chapter I will describe basic concepts and basic mechanisms on the MBCF. Next I will represent a wide variety of the MBCF functions in Chapter 4, and give discussions on qualitative points of the MBCF in Chapter 5. In Chapter 6 I will explain a concrete implementation of the MBCF using ethernet and give the result of its performance evaluations.

3.2 Background to the MBCF

The MBCF is a software-only solution for the realization of protected and virtualized high-speed user-level communications and synchronization. The MBCF was invented as part of Network of Workstations (NOW) project and is for distributed-memory multiprocessor systems that do not have hardware DSM mechanisms.

So far, many hardware DSM mechanisms (including MBP) are proposed and no DSM mechanism has been proved to be sufficiently cost-effective and general for wide range of applications. The WSs/PCs of recent times are very powerful but, because of mass-productions, they are inexpensive. The performance of their network interfaces has also improved rapidly. They can therefore be used as the nodes of parallel computers. Hardware for LAN communications is usually available for such machines, and the performance of such hardware is also subject to rapid and continued improvement. Hardware DSM mechanisms are not able to exploit LAN hardware and require additional special hardware for communications. This represents a double investment in communication functions and makes the systems expensive. Another big drawback of hardware DSM mechanisms is that their granularity of communications is too fine to bring out the maximum performance of either network hardware or DMA hardware. The granularity of communications is comparable to the amount of data accessible in a single processor operation, and is less than or equal to the size of a cache line. Since, for microprocessors of recent generations, the outgoing memory-access operation which is required to invoke some function of the hardware DSM mechanism for a transmission is more expensive than the software processing of hundreds of instructions within the on-chip cache, a hardware DSM mechanism that is based on fine-grained transmission will be poor at high-bandwidth applications. Therefore, if we assume that our system has fast communication hardware for LAN (e.g., gigabit ethernet) with a bandwidth that is comparable to that of a hardware DSM mechanism, it is probably true that applying light-weight software-based protocols to the communication hardware will provide a performance that surpasses that of hardware DSM mechanisms in terms of maximum throughput. We consequently need to develop efficient methods for user-level communications and synchronization that only require commodity-hardware mechanisms.

I assume that the MBCF system is equipped with off-the-shelf network interface cards: NICs (e.g., ethernet cards). These cards have no functionality for protection or security, transmitting memory images as packets to other nodes, or receiving packets from other nodes into a specified ring buffer in the system (kernel) space.

Two factors produce most of the overheads resulting from user-level communications and synchronization. The first is the methodological factor (i.e., functionalities, protocols, packet formats, etc.). Conventional user-level communication interfaces (for examples, TCP/IP [62], UDP/IP [61] and MPI [17]) are of the message-passing type, and their functions are limited, in remote-write operations, to specific message-buffer addresses of the kernel-space. To break out of this limitation, I adopted a memory-based form of operation, in which arbitrary target addresses and a wide variety of functions are available. By using memory-based operations, protection and virtualization in communications and synchronization can be replaced with those in memory access operations that are induced by original memory-based communications and synchronization.

This replacement makes high-speed implementation of the scheme feasible, since the advanced architectural mechanisms of processors in terms of memory access can then be exploited.

The other is the software-engineering factor (implementation methodology). In a conventional OS, communications between and synchronization of nodes (machines) is regarded as a set of typical I/O events, like disk operations, and device-drivers for communications and synchronization exploit service routines in common with other I/O devices. Consequently, the device-drivers suffer rather large processing overheads that are superfluous in terms of the functions of communications and synchronization. To realize a high-performance implementation, MBCF-dedicated system-calls and MBCF-dedicated interrupt routines have been developed and used, so that no operations are irrelevant to the functions of the MBCF. The MBCF thus implemented is free of the problems arising from the second factor (the software-engineering factor).

3.3 Basic concepts of the MBCF

3.3.1 From MBP to MBCF

The MBCF uses pure software routines to emulate the MBP's functions. The three types of software mechanisms are user-level-request codes (packet-sending codes), MBCF-dedicated interrupt handlers and MBCF-dedicated system-calls. In the MBP system, a remote memory access is invoked by a processor's operation on memory. When an MBP detects access, by a processor, to memory at a target address that belongs to a remote node (the target node), the MBP translates the access information into a format for inter-node communications, and makes a packet, then transmits it to the target node. When the MBP in the target node receives the packet, it executes the remote-access specified in the packet and returns a reply if necessary. In the MBCF system, on the other hand, remote memory access is invoked by an explicit system-call for an MBCF function. Firstly, a user-program prepares an MBCF packet in user-mode and executes the MBCF-requesting system-call. Secondly, the kernel-level routine of the MBCF-dedicated requesting system-call makes a packet for inter-node communications and transmits it via conventional NICs. Finally, the MBCF-dedicated interrupt routine at the target node receives the packet, immediately executes the remote-access operation specified by the packet, and returns a reply if necessary. In comparison with an MBP system, an MBCF system carries some additional software overhead, but an MBCF packet need not correspond to a memory access operation by a processor. This flexibility means that the MBCF provides opportunities in terms of optimization of the number of packets and amount of data being communicated. More specifically, large amount of data can be handled in a single MBCF operation and/or multiple MBCF operations can be merged into a single packet. I call such a merged packet a "combined packet" and refer to this technique for optimization as "combining". If a system has rather poor communication hardware in comparison with its processor power, these opportunities

for optimization must be seized to obtain efficient execution.

3.3.2 Protection and security mechanisms of the MBCF

In the MBCF scheme, an MBCF-dedicated kernel-level interrupt routine makes the actual access to the target location of an operation requiring access to remote memory. If strong protection and security are required for the system, powerful capability-checking or authentication procedures can be added to this interrupt routine. However, such additions, of course increase the overhead of the MBCF interrupt routine. To avoid increasing the overhead in this way, I developed a novel method that exploits page-aliases and simple access-keys.

In parallel processing, when some error occurs during an activity, the further execution of related activities is probably meaningless. Owing to this characteristic a simple protection mechanism which merely separates a task from other unrelated tasks is sufficient. To prevent the spread of the bad effects of an error to other tasks, the MBCF uses logical address spaces with a memory-management mechanism. Only those memory areas which are mapped to the target task can be accessed through the MBCF. To protect memory from attacks by other tasks, I adopted a unique access-key to represent the right to access a target memory-space.

On the other hand, in distributed processing (e.g., the client-server model) the server's activities must be protected from errors during and attacks by client activities. In this case, a strict protection mechanism which distinguishes the working area on each client from the working areas on the server and the other clients is required. I solved this issue by using a combination of unique access-key and page-aliasing. In Section 3.4, I will include this protection scheme in an illustration of the behavior of the MBCF, so I only describe the scheme in outline from here.

When the server requests MBCF communications with a distrusted client, the server creates an agent (an activity with its own independent memory-space) within the server's node, and delegates the job of communicating with the client to the agent. The server then allocates a working-memory area for use in communication with the client, and this area is also mapped to the agent. In other words, the area is intra-node shared-memory shared between the server and the agent by using a page-alias mechanism. After these preparations on the server's node, the agent informs the client of the agent's access-key (but not the server's access-key), and the client uses the agent's memory-space to communicate with the server. Even if the client has an evil intention, such as to destroy the server, it can only damage the agent's space and can NOT stop the execution of other server activities.

In both the MBCF scheme and MBP schemes, protection and virtualization in communications and synchronization are replaced with protection and virtualization of memory accesses. This replacement makes a high-speed implementation of the mechanisms involved feasible. Especially in the MBCF scheme, since

the TLBs and the MMUs of node processors (conventional microprocessors) are exploited for the translation of actual accesses in target nodes, no additional hardware mechanisms are required to improve performance.

3.3.3 Virtual and global addressing in the MBCF system

In the MBCF scheme, communications and synchronization are performed via virtual inter-node memory locations. An address at some location is specified by the combination of a logical task-ID “Ltask” and a logical address “Laddr”. I write the combination as “(Ltask:Laddr)”. A **task** is an abstraction of a processor’s activity and has its own memory-space. Laddr is an address within this memory space. A task in the MBCF system belongs to a specific node, and is specified in this physical-node by the combination of a physical node-ID “Pnode” and a physical task-ID “Ptask”. I write this combination as “(Pnode:Ptask)”. In user-level application programs, Ltask is used to specify a task but the combination (Pnode:Ptask) is not used directly. This degree of virtualization allows the MBCF system to migrate tasks from node to node. The OS for the MBCF system maintains one task-translation table per task, and that table represents the correspondences between Ltasks and (Pnode:Ptask)s. When a task is migrated from its original node to some other node, the OS updates all task-translation tables which have the entries related on the moved task.

The Ltask notation for an MBCF applications is the local and virtual identifier for that task, while the (Pnode:Ptask) notation is used in MBCF inter-node packets. Therefore, in each MBCF packet, the notation for a global address is “(Pnode:Ptask:Laddr)”.

3.4 Illustration of the MBCF’s behavior

In this section I give detailed descriptions of the sequences of actions for an MBCF operation and for the application of that operation to an essential shared-memory mechanism. The MBCF has a variety of commands but the basic procedure is the same for the commands. The remote-memory-write operation “MBCF_WRITE” and cache system with a update-type protocol are my examples.

3.4.1 Assumptions of network interface cards (NICs)

I assume that off-the-shelf NICs are used to implement the MBCF system. Most of these NICs are only able to transmit the memory image of a packet to other nodes and to receive packets into a kernel-specified ring buffer in the system (kernel) space. This is shown in Figure 3.1. The processor on a sender node creates a packet image, which includes its header (routing) information, in the NIC-DMA area, which NIC can directly access for sending and/or receiving. The processor then kicks its NIC to start sending the data, using DMA to retrieve it from memory. The receiving node has a ring buffer for incoming packets in its NIC-DMA area.

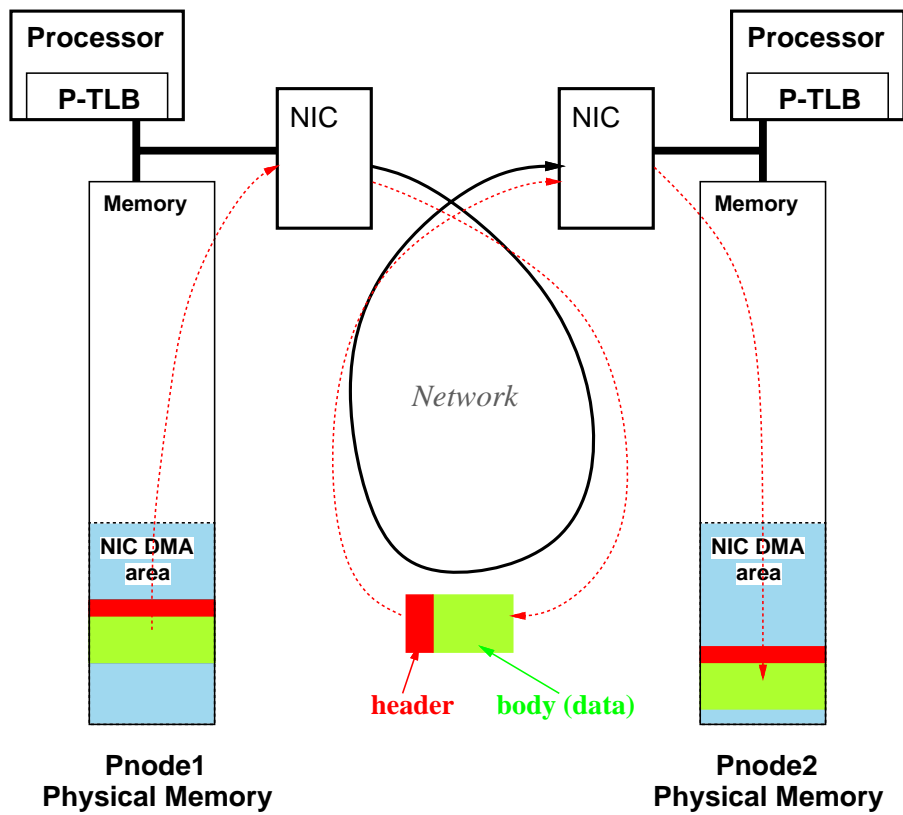


Figure 3.1: Functions of conventional NICs

The NIC selects packets that are bound for its node, and copies the images of these packets to its ring buffer. The NIC then generates an interrupt to notify its processor of the arrival of the packets.

3.4.2 Illustration of MBCF_WRITE operation

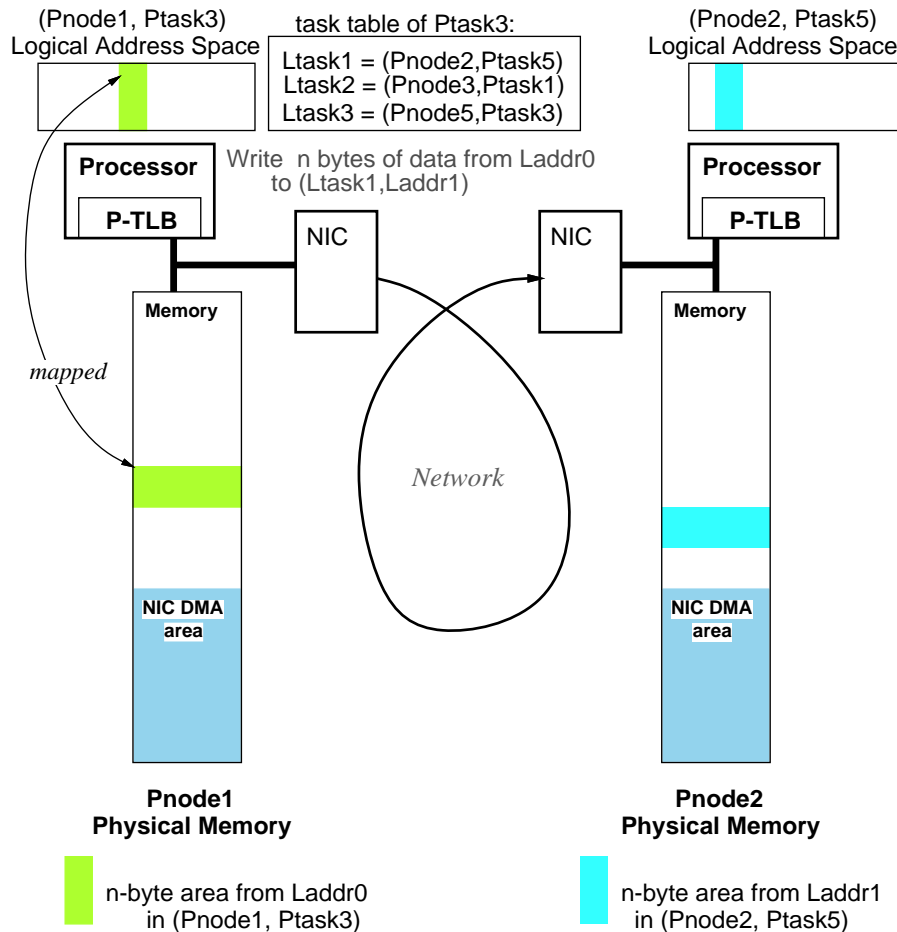


Figure 3.2: Illustration of the MBCF_WRITE operation (1)

Figure 3.2 shows the use of MBCF_WRITE operations by two communicating tasks in a parallel application. The left task (Pnode1:Ptask3) in the figure is the requestor of the remote-write operation. Figure 3.3 represents the procedures involved in making a packet at the requestor node. The requestor sets up the mbcf request parameters, these consist of the target task-ID (Ltask1), a target address (Laddr1), an access-key for the target task, the MBCF command (MBCF_WRITE), the amount of data to be written and the pointer to the data. After preparing the MBCF parameters, the requestor makes an MBCF-requesting system-call to the OS, and the

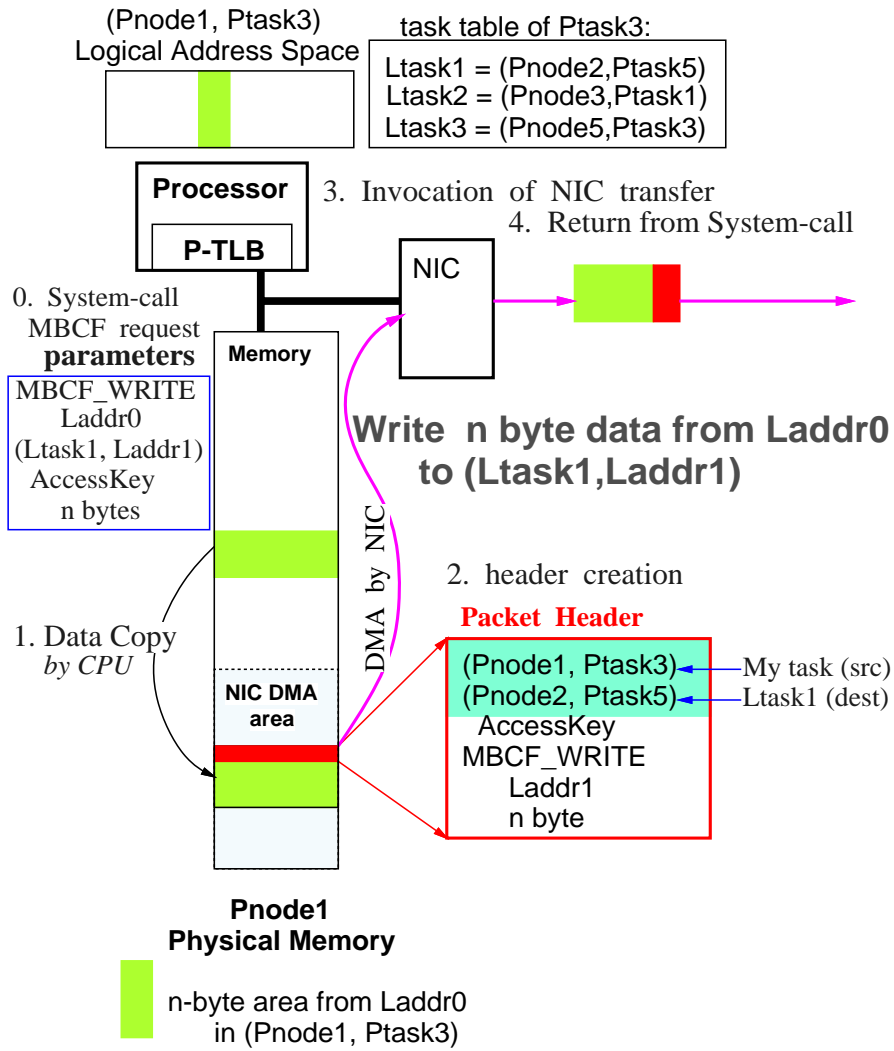


Figure 3.3: Illustration of the MBCF_WRITE operation (2)

OS translates the logical task-ID to obtain paired physical task-ID (Pnode2:Ptask5) and makes up the routing information so that the NIC can send the inter-node MBCF packet. The OS then makes the NIC send the packet. Figure 3.4 shows the arrival of the packet at the target node. The packet is carried to the target node

task table of the requester task:

```
Ltask1 = (Pnode2,Ptask5)
Ltask2 = (Pnode3,Ptask1)
Ltask3 = (Pnode5,Ptask3)
```

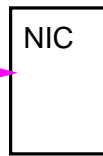
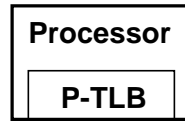
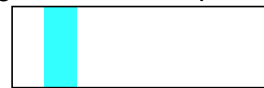
Original Request

```
MBCF_WRITE
from Laddr0
to
(Ltask1, Laddr1)
n byte
```

```
(Pnode1, Ptask3)
(Pnode2, Ptask5)
AccessKey
MBCF_WRITE
Laddr1
n bytes
```

Packet Header

(Pnode2, Ptask5)
Logical Address Space



DMA by NIC

**Pnode2
Physical Memory**


 n-byte area from Laddr1 in (Pnode2, Ptask5)

Figure 3.4: Illustration of the MBCF_WRITE operation (3)

(Pnode2) via the network of this system. When the packet arrives, the target node's NIC uses DMA to copy the

packet image to the ring buffer, and generates an interrupt signal to inform the node's processor of the arrival of one MBCF packet. Figure 3.5 represents the procedures of the MBCF-dedicated interrupt routine in the

task table of the requester task:

```
Ltask1 = (Pnode2,Ptask5)
Ltask2 = (Pnode3,Ptask1)
Ltask3 = (Pnode5,Ptask3)
```

Original Request

```
MBCF_WRITE
from Laddr0
to
(Ltask1, Laddr1)
n bytes
```

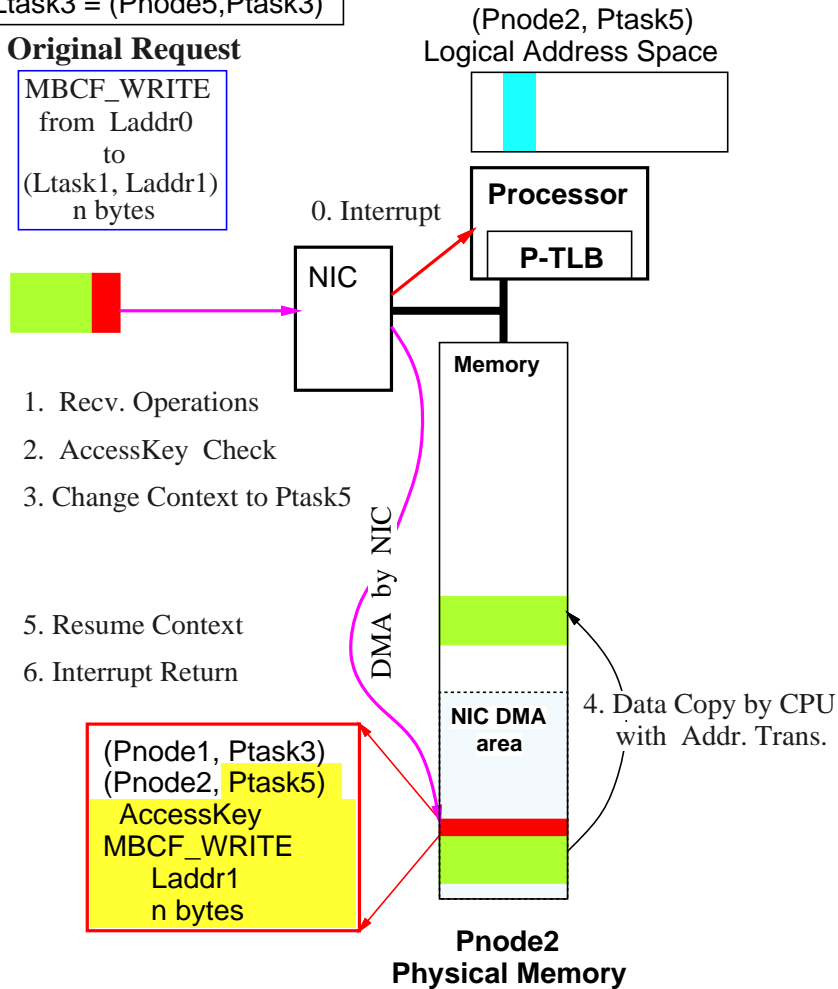


Figure 3.5: Illustration of the MBCF_WRITE operation (4)

target node. The interrupt causes the MBCF-dedicated interrupt routine to take control of the processor. The routine then begins to process the MBCF command. Firstly, the routine performs the lowest-level protocol required by the NIC, if any. Secondly, it uses the physical task-ID (Ptask5) to find the target task, checks the access-key, and if the key fits, changes the context (address space) to that of the target task, then writes

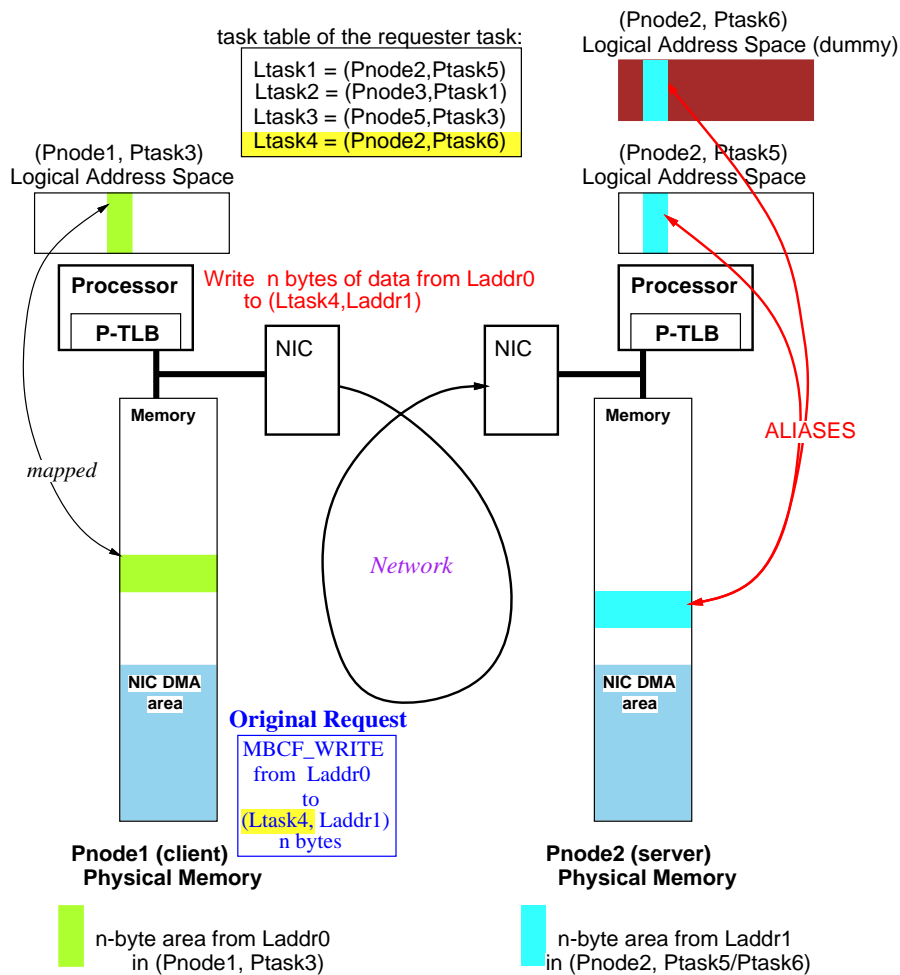


Figure 3.6: Illustration of a secure MBCF_WRITE (1)

the data to the target address (Laddr1) under the user-level privilege instead of supervisor-level one. The interrupt routine then ends. If error occurs or a check detects some kind of violation, the interrupt routine stops following the above procedure so that it can deal with the situation. Optional parameters are available for the MBCF_WRITE command (e.g., an address for status-return, a flag for elastic memory barrier and so forth), and, if the optional action requires that a reply be sent from the target to the requester, the reply packet is returned.

If the system has a page-swapping mechanism and the area which includes the target address is swapped out from main memory to a secondary storage device, a page-fault trap occurs in the MBCF-dedicated interrupt routine. The page-fault handler saves the MBCF packet to the buffer in the kernel space, and set the flag which

indicates that the target task awaits an MBCF operation, then invokes the operation that will swap the target area back into memory. MBCF requesting packets that then arrive for other tasks are processed normally, but those for the same target task are buffered in the save area until the waiting flag has been cleared. After completion of the page-in operation, all saved MBCF packets are processed and the waiting flag for the target task is then cleared.

Figure 3.6 shows the operation of MBCF_WRITE in a distributed application which requires secure operations. In this case, the client task requests an MBCF_WRITE operation to the server task. In the figure

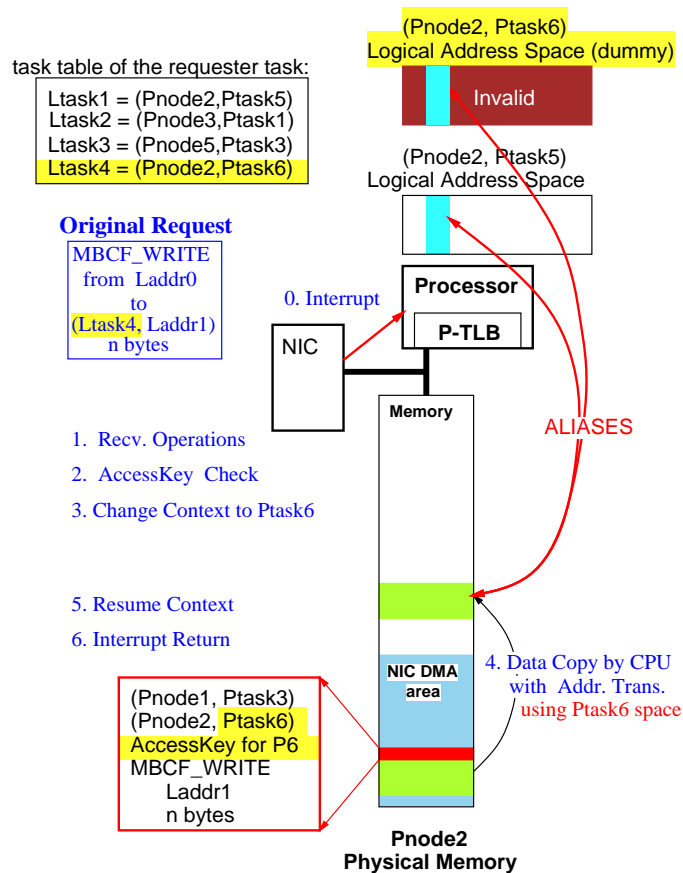


Figure 3.7: Illustration of secure MBCF_WRITE (2)

the left-hand task (Pnode1:Ptask3) is the requestor and a client task, and the right-hand task (Pnode2:Ptask5) is the server and the task (Pnode2:Ptask6) is a dummy task used to secure communications between the server task and the task (Pnode1:Ptask5). The client sets up the MBCF parameters. These consist of a target task (**Ltask4**), a target address (Laddr1), the access-key to the target task, an MBCF command (MBCF_WRITE),

the amount of data to be written and the pointer to the data. It is important that the target task is the dummy task of the server and not the actual server task. After preparing the MBCF packet, the client makes the MBCF-requesting system-call to the OS, and the OS then sends the packet. When the packet arrives, the target node's NIC generates an interrupt signal. Figure 3.7 represents the procedures of the MBCF-dedicated interrupt routine in the server node. The MBCF-dedicated interrupt routine uses the physical task-ID (Ptask6) to find the target task, then writes the data to the target address (Laddr1). Though the target task is the dummy task but not the actual server task, the memory area of the target location is shared among these two tasks. The server can thus read the data without requiring additional procedures. If the target address is wrong and beyond the limits of the working area for the client, the effect of this wrong address is NOT propagated to the server's memory areas to which the dummy has no mapping.

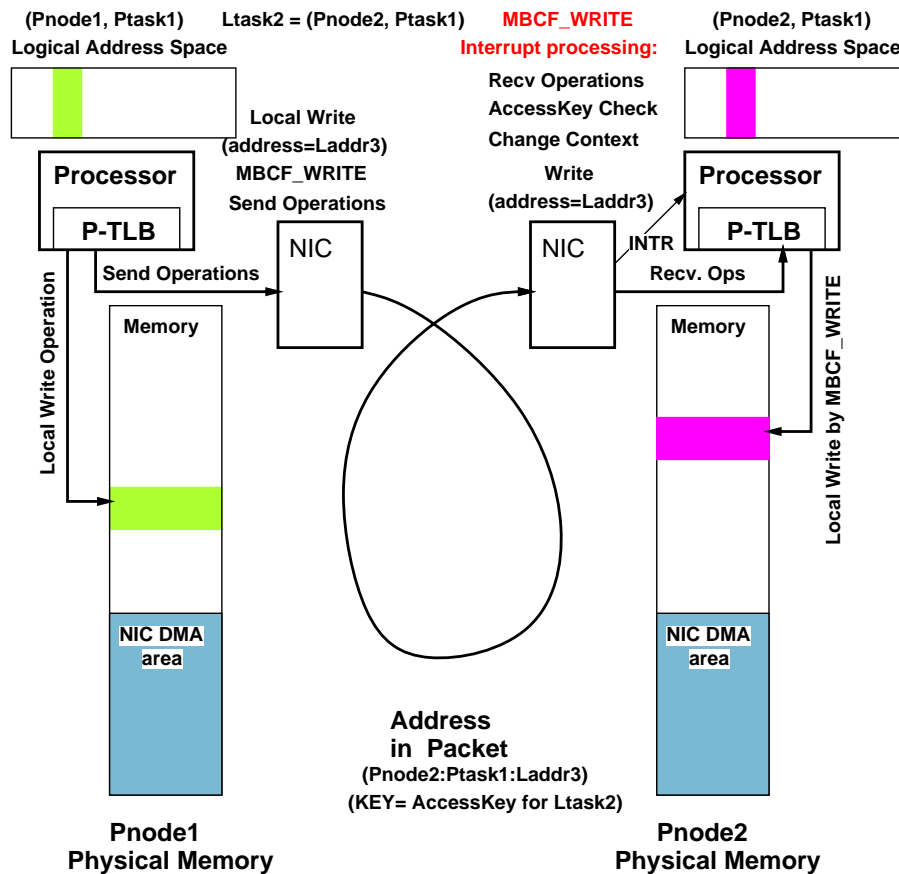


Figure 3.8: Illustration of cache updating with MBCF_WRITE

3.4.3 Illustration of the update-cache mechanism using MBCF_WRITE

Figure 3.8 shows the update operation in a cache of a system that is using the MBCF. In the figure, two tasks communicate through a page-based cache with an update-type protocol, and the left-hand task is the writer to a shared page. The MBCF itself does not directly support any form of caching, so codes other than the store instruction are required for a shared-write. These codes are added to the program object code of a program by the optimizing compiler for the MBCF. Firstly, the writer stores the data at a local location (Pnode1:Ptask1:Laddr3). Secondly, the writer sends an MBCF_WRITE message to the cooperative task (Ltask2). The global address of the target location is (Pnode2:Ptask1:Laddr3). If there are more than two cooperative tasks, I use the MBCF multicast method that is described in Chapter 4.

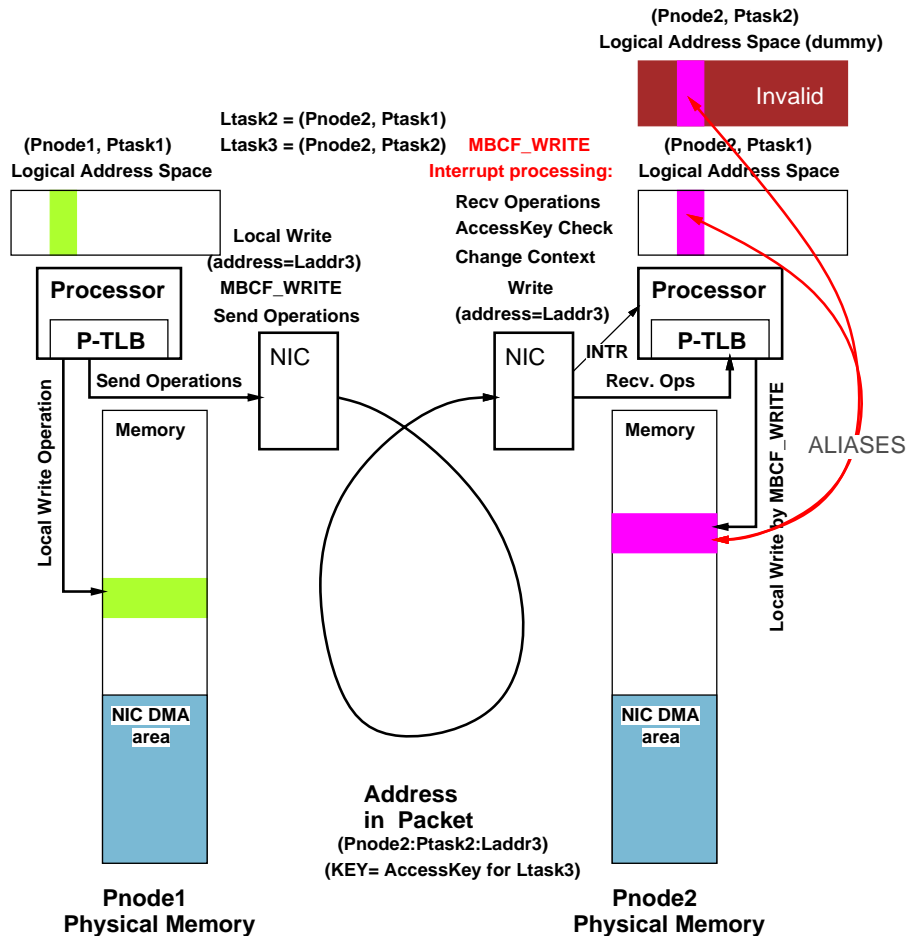


Figure 3.9: Illustration of cache updating with secure MBCF_WRITE

Figure 3.9 shows the update operation in a cache system which requires secure operation. In this case, the

client task requests an update operation of the memory area which is shared with the server task. Firstly, the client stores the data at a local location (Pnode1:Ptask1:Laddr3). Secondly, the writer sends an MBCF.WRITE message to the dummy task (Ltask3) of the cooperative task (Ltask2). The global address of the target location is (Pnode2:Ptask2:Laddr3).

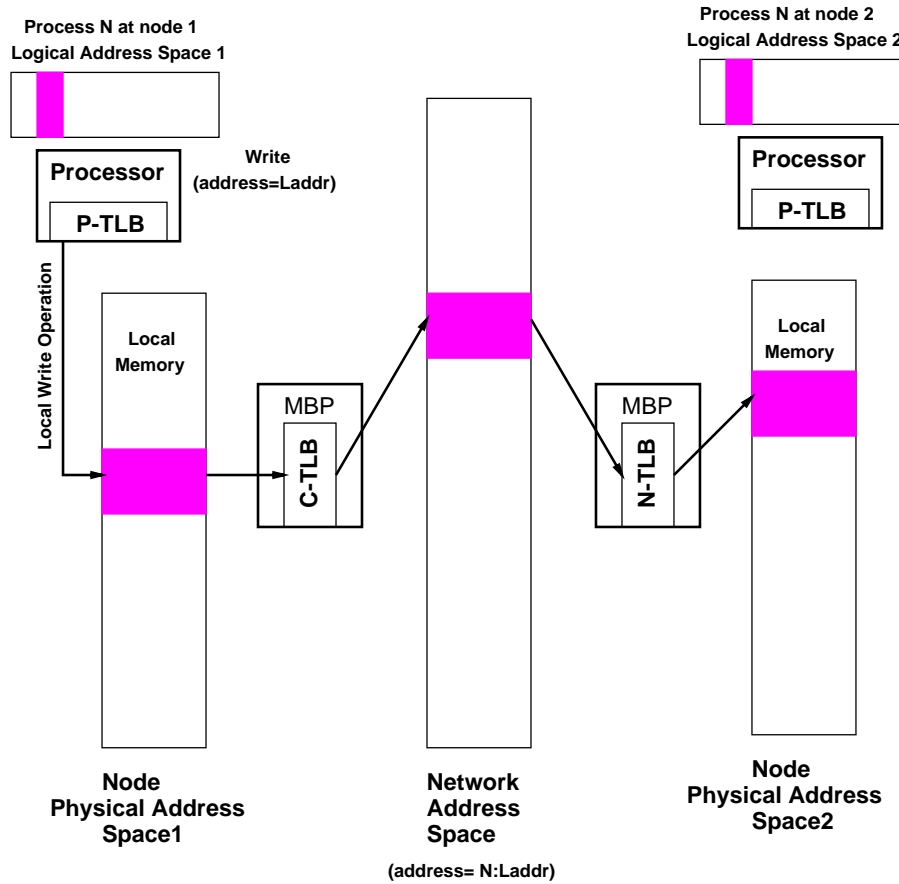


Figure 3.10: Illustration of the MBP's update process

The MBCF's execution model as represented in Figure 3.9 is the closest to that of the MBP. Comparing the figure with Figure 3.10, we can see that the MBCF uses the processors' TLBs (P-TLBs) for the functions of both the N-TLBs and the P-TLBs of the MBP system. In the MBP system, an MBP which is attached to the processor's memory bus is required to detect remote-memory access by a processor so three types of TLBs (P-TLB, C-TLB and N-TLB) are needed. The C-TLB is used to translate physical addresses in the node to logical global addresses within the system. In the MBCF system, however, users explicitly specify the logical global addresses (Ltask:Laddr) and so a TLB that corresponds to the C-TLB is not required. From

the viewpoint of protection and security, the three-TLB-type system of the MBP is an over-investment, and a single TLB for each remote-access is enough. Therefore, in the secure execution model of the MBCF, the processor uses P-TLB's page entries on the dummy task in the target node to translate and check addresses for remote-access, and, in this situation, the P-TLB works in the same way as the N-TLB of the MBP system.

Chapter 4

Functions of the MBCF

In this chapter I list the commands and supporting commands of the MBCF, along with their options. A command specifies a main function at a target location in a target node. The commands are common to all implementations of the Memory-Based Communications and Synchronization (MBCS) schemes, and the MBP also has a fine-grained version of a set of the commands. Options are attached to the command and give it some additional meanings. Several supporting commands have been implemented to complement the MBCF system, and they are used in preparation for MBCF communications, modifications to the task-translation table, and user operations on those MBCF structures which require mutual exclusion or write-protection against external access.

Although there is no limitation on the variety of MBCF functions, heavy/complicated commands are unsuitable for a multi-tasking system from the viewpoint of fairness in task scheduling. Operations in a single call of an MBCF-interrupt routine are regarded as part of its target task. An interrupted task is robbed of execution time by invocations of the MBCF-interrupt routine for other tasks. Operations triggered by the receiving of interrupt for communications in the conventional system are also asynchronous and disturb the execution of the interrupted task. If the cost of the MBCF-receiving routine is comparable to that of the conventional one, the degree of fairness of the MBCF system can be considered similar to that of the conventional system.

4.1 The commands

There is a wide variety of MBCF commands. I list them in the pages that follow.

- **MBCF_WRITE**

MBCF_WRITE is a remote-write operation and is a basic and simple command of the MBCF. The

amount of data carried by one command is specified arbitrarily but the maximum amount is restricted to the maximum capacity of a single packet on the given network. This restriction applies to the other MBCF commands.

- **MBCF_WRITE_F**

MBCF_WRITE_F is an ordinary remote-write operation plus a flag-setting operation. After the remote write, a specified value is written to the flag location which is indicated by the command packet. This function was first seen in the remote-access hardware of the AP-1000+ system [18]. In the MBCF, this function can be also realized in the form of a combined packet in which two MBCF_WRITE commands are merged, but an MBCF_WRITE_F packet is a little smaller and a little lighter than the combined packet.

- **MBCF_READ**

MBCF_READ is an ordinary remote-read command. This command is followed by a reply packet which writes the read-data to the pre-specified location in the requesting node. The address for storage of the read-data is carried by both the request-packet and the reply-packet, and this policy (the MBCF-requesting routine has as little states as possible) makes the MBCF procedures simple and fast.

- **MBCF_SWAP**

MBCF_SWAP is an atomic-swap command. When this command packet arrives at its target node, the MBCF-interrupt routine atomically exchanges local data at the target location for the data in the packet and returns a packet with the local data. The returned data are written at the location which is specified by the requestor when it makes the request. In memory-based (MBCF and MBP) schemes, since all atomic operations are executed in the target node without any inter-node interlocks, it is possible to simultaneously invoke multiple atomic operations on the same location at the same node from multiple nodes, and these operations are efficiently executed in a pipeline-like manner within the target node.

- **MBCF_FETCH_ADD**

MBCF_FETCH_ADD is an atomic fetch&add [16] command. When this command packet arrives at the target node, the MBCF-interrupt routine reads the local data at the target location, then adds the local data to the data in the packet and writes the sum back to the target location, and returns a packet with the original locally read data (i.e., before the addition). The returned data are written at the location which is specified by the requestor when it makes the request. The series of operations at the target node is atomic. The amount of data carried by this command is also variable like other commands, and the addition is executed like as a vector additions if there are multiple data in a single command.

- **MBCF_COMP_SWAP**

MBCF_COMP_SWAP is an atomic compare&swap command. When this command packet arrives at the target node, the MBCF-interrupt routine reads one word from the reference address specified in the packet. If the word is equal to the reference data in the packet the following operations are executed. If not, the command ends. If operation continues, the interrupt routine exchanges local data at the target location with the swap data in the packet and returns a packet containing the local data. The returned data are written at the location which is specified by the requestor when it makes the request. The series of the operations at the target node is atomic.

- **MBCF_UPDATE**

MBCF_UPDATE is a multicasting remote-write command. Each memory-page which is allocated for the targets of an MBCF_UPDATE has a pattern for multicasting. The delivered pattern consists of inter-node link-pointers which form a tree-like route that indicates the target pages for the multicast. When this command arrives at a node, the MBCF-interrupt routine updates the specified location on the page and the MBCF_UPDATE packet is then copied and transmitted to the external pages which are specified in the multicast pattern. I call this multicasting technique hierarchical multicasting [34, 44], and in Chapter 2 I have already explained a version of this technique for the MBP. The hierarchical multicasting technique applies to the other page-based multicasting commands of the MBCF, as well as to MBCF_UPDATE.

When a multicasting command (including MBCF_UPDATE) is used in implementing the protocols of a DSM system, write operations to locations in copy-pages (i.e., other than those at tree-roots) are first forwarded to the home-pages, which are pages at tree-roots of the multicast patterns. The multicasting of packets with the command is then invoked from the home-pages.

When a status-report option or acknowledge option for memory barriers accompanies a multicast command, the “acknowledge-message combining” technique [34, 44] is used to collect the reports or acknowledge-messages. As I explained in Chapter 2 “Memory-Based Processor”, acknowledges-messages (or reports) are returned from leaf nodes (leaf pages) to the direct parent node in the pattern of multicasting but not immediately to the original requesting node. The parent node waits for the acknowledges-messages from all of its direct children, then the parent merges these messages into a combined acknowledge-message. This acknowledge-message is then transmitted to the direct parent of the parent node itself. This combining technique eliminates the hot-spots in terms of the concentration and collection of acknowledge-messages at the original requesting node (or home node).

- **MBCF_INVALIDATE**

MBCF_INVALIDATE is a multicast command for page-based invalidation of remote memory. When this command arrives at a node, the node's processor executes the MBCF-interrupt routine and invalidates the specified page, and the MBCF_INVALIDATE packet is then copied and transmitted to the outside pages that correspond to the pattern for multicasting as described under MBCF_UPDATE. This command is usually used to realize an invalidate protocol for a page-based DSM system. Non-page-based invalidate commands can also be implemented in the MBCF scheme but are not listed in this section.

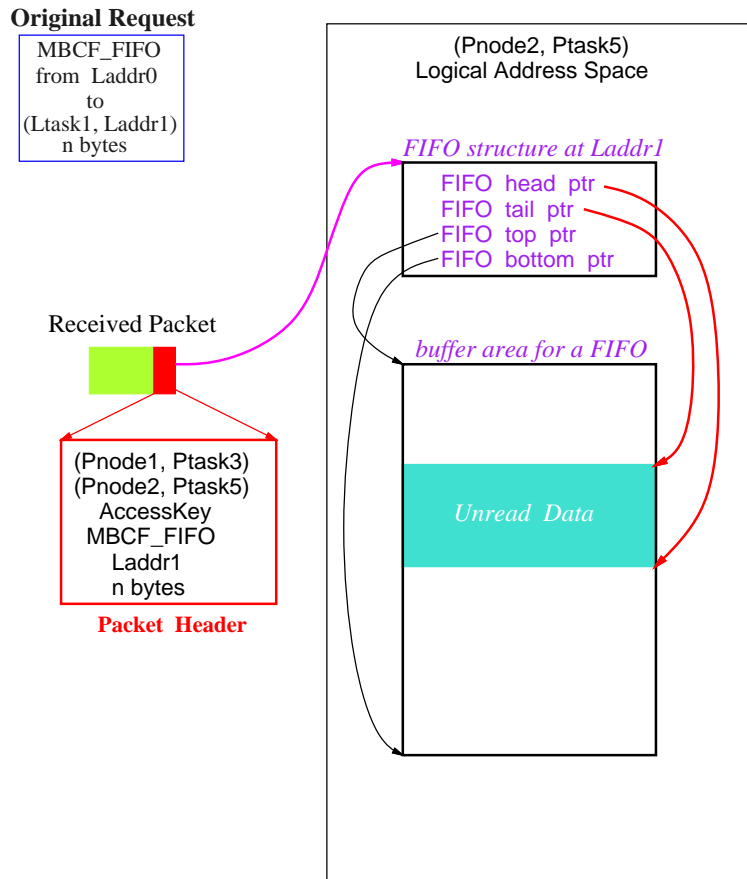


Figure 4.1: Operation of the MBCF_FIFO (1) command

- **MBCF_FIFO**

MBCF_FIFO is a command for storing data in fifo structures (fifos). In the MBCF scheme, users can create any number of fifos at any locations. The MBCF_FIFO command packet specifies a location of

the structures, which define fifos, rather than that of the buffers in which data will be stored. Figure 4.1 shows a fifo structure and buffer in a target node. Each fifo structure includes four pointers, to the top, head, tail, and end of the data buffer, and status flags. In the MBCF-interrupt routine, the processor reads the pointers and flags of the target fifo structure first, stores data in the buffer specified by the structure, then updates the structure. These operations are performed locally and atomically. Figure 4.2

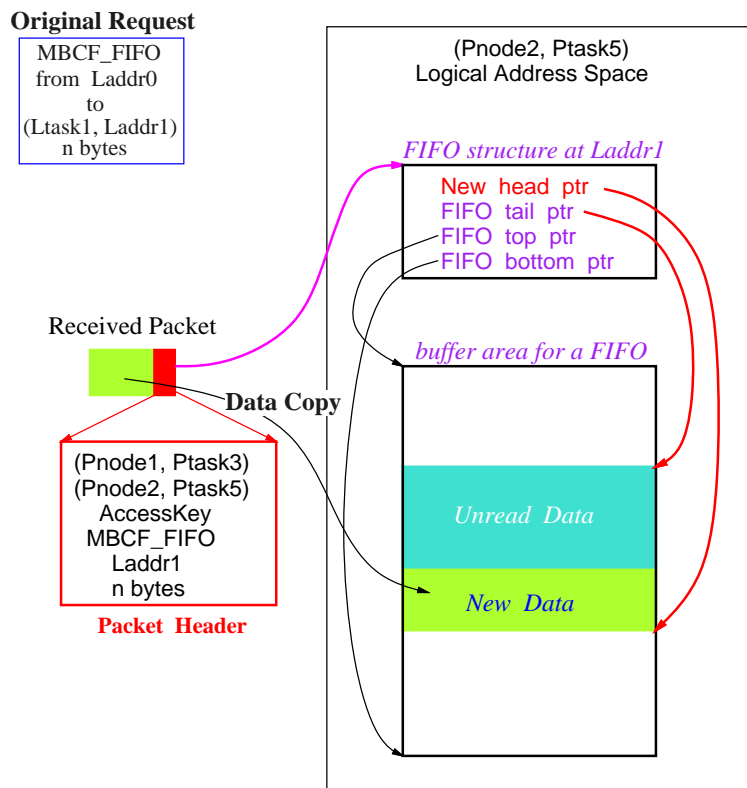


Figure 4.2: Operation of the MBCF_FIFO (2) command

represents the series of these procedures at the target node.

If the buffer of the specified structure is full, processing of the data in the MBCF_FIFO packet is cancelled. The status-report option should be used to inform the requesting task (node) of this cancellation of the command. This possibility of cancellation means that this command cannot guarantee that the order of MBCF_FIFO commands from a given task will remain intact. If fifoness (i.e., the characteristic of keeping the order intact) is required for the fifo-data-entry commands from a given requestor, users use the status report option to implement a handshake protocol, or use the MBCF_FIFOe commands described below.

If protection of an fifo is required, memory pages used for fifo structures and fifo buffers are set as read-only in user mode (but both read and write are permitted in kernel mode) and users are needed to use a light-weight system-call to get an entry from a memory-based fifo.

- **MBCF_FIFOe**

MBCF_FIFOe consists of two commands MBCF_FIFOe_NOMAL and MBCF_FIFOe_RETRY. These commands implement the **eager** usage of a memory-based fifo while maintaining fifoness for the commands from a given requestor. The MBCF_FIFOe commands exploit the same structures as the MBCF_FIFO command. The only differences between the MBCF_FIFOe and MBCF_FIFO mechanisms are in the method of handling the buffer-full situation. After a fifo-entry command has been cancelled in the MBCF_FIFOe scheme, the use of the target fifo is prohibited until a special fifo-entry command has arrived.

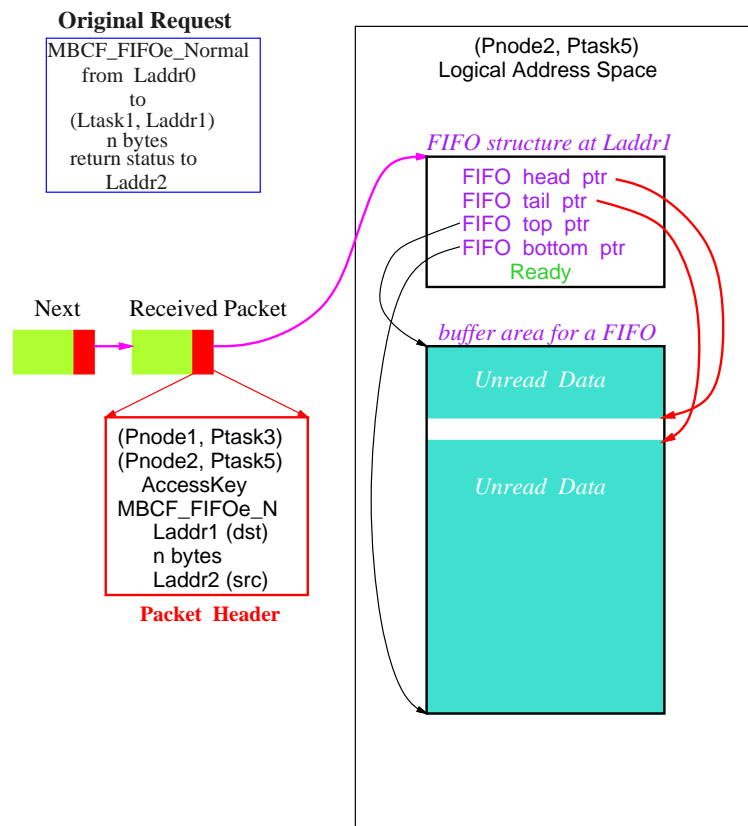


Figure 4.3: Operation of the MBCF_FIFOe (1) command

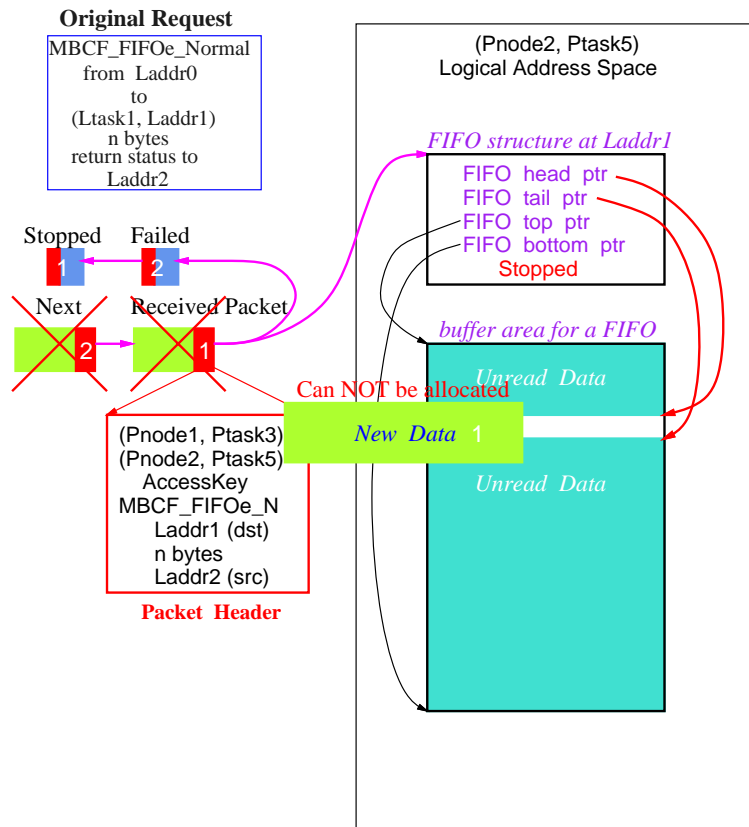


Figure 4.4: Operation of the MBCF_FIFOe (2) command

In my implementation, the MBCF_FIFOe scheme assumes that requestors record the order in which packets are sent and keep copies of the packets. When status reports are returned from the target to tell the requestor that packets have been cancelled, the requestor uses this information to resend the cancelled packets in the proper order. The first status-report in the series of packets returned to indicate cancellations carries the information that the fifo has entered its cancellation state. The requestor uses a special MBCF_FIFOe.RETRY command to resend the original packet which corresponds to the status-report that indicate a fifo-state transition, and only this MBCF_FIFOe.RETRY command is able to restart the fifo structure at the target.

If the communication system is reliable (i.e., there is no packet-loss) and sender nodes do not keep copies of transmitted packets, the MBCF_FIFOe-method can still be implemented by adding the contents of the received packets to replying packets for status-reports which indicate cancellations. The method, in which a replying packet includes the content of the received packet with a fifo-entry command when

it has been cancelled, is called the “return-to-sender” method, and is widely used [2, 8, 60]. In the MBCF_FIFOe scheme, the “return-to-sender” method is extended to keep the order of point-to-point packets intact.

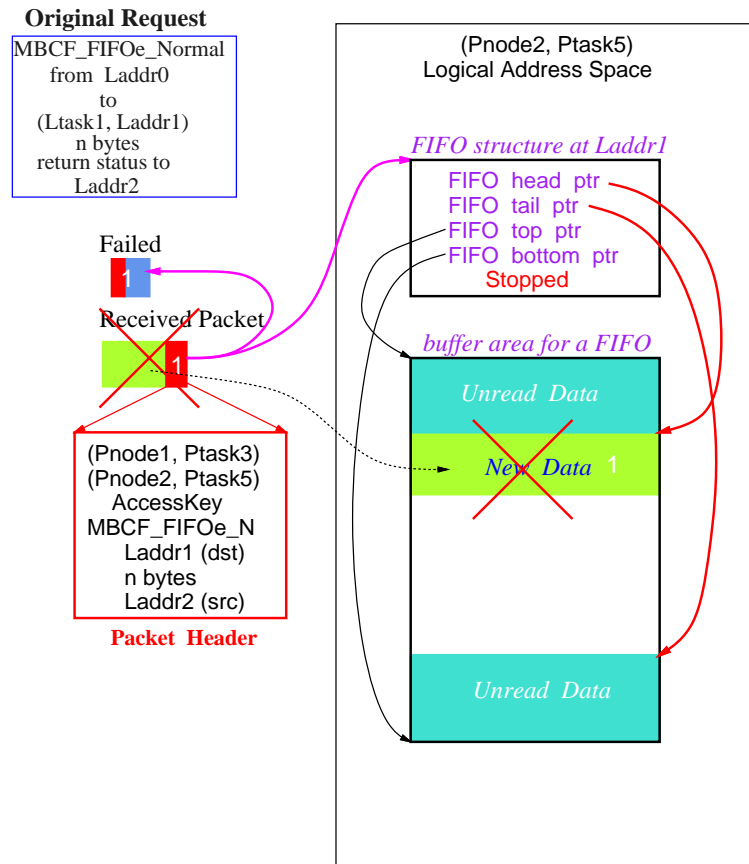


Figure 4.5: Operation of the MBCF_FIFOe (3) command

– **MBCF_FIFOe_NORMAL**

MBCF_FIFOe_NORMAL is an ordinary MBCF_FIFOe data-entry command (Figure 4.3). Once an MBCF_FIFOe_NORMAL command is cancelled (Figure 4.4), all subsequent MBCF_FIFOe_NORMAL commands will also be cancelled, even if a room to receive data again becomes available (Figure 4.5). A MBCF_FIFOe_RETRY command which has the stopped fifo as a target resumes use of the fifo if the buffer has room to receive data. Users detect transitions to the cancellation state by use of the status reports. Three status values may be returned for an MBCF_FIFOe command:

1. the command successfully stored data in the buffer;
2. the command failed to store the data and the fifo entered its cancellation (stopped) state; or
3. the command failed to store data because of a cancellation state.

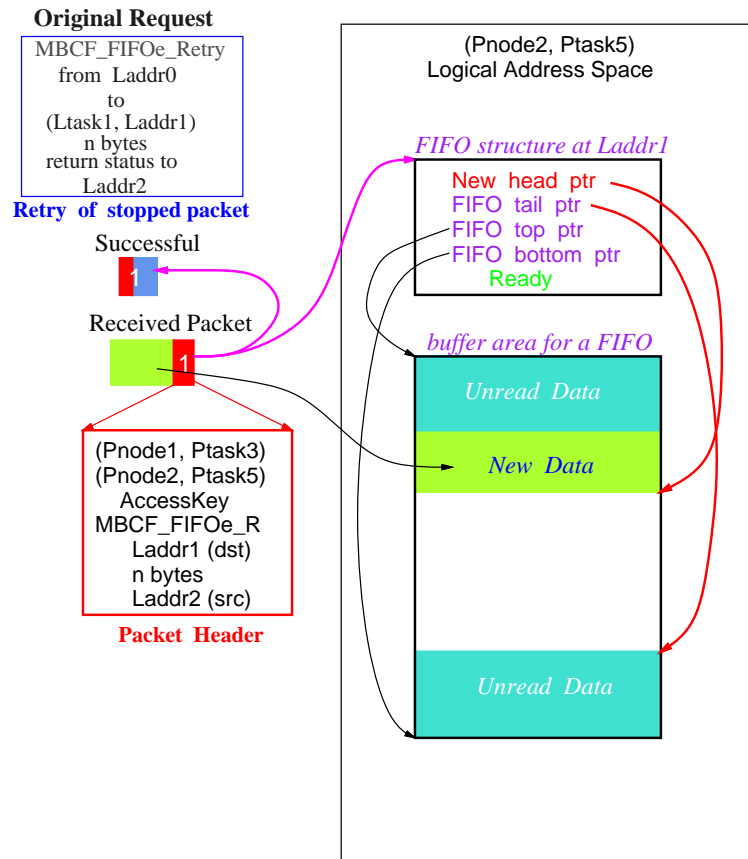


Figure 4.6: Operation of the MBCF_FIFOe (4) command

– **MBCF_FIFOe_RETRY**

MBCF_FIFOe_RETRY is the special MBCF_FIFOe data-entry command that resumes operation with a stopped fifo if there is room to receive data (Figure 4.6). If the target fifo is running (i.e., the fifo is in its normal state), this command acts just like an MBCF_FIFOe_NORMAL command.

• **MBCF_SIGNAL**

MBCF_SIGNAL command is accompanied by a remote invocation of the user-specified program with the privileges of the target task. The mechanism by which the MBCF_SIGNAL commands invoke user

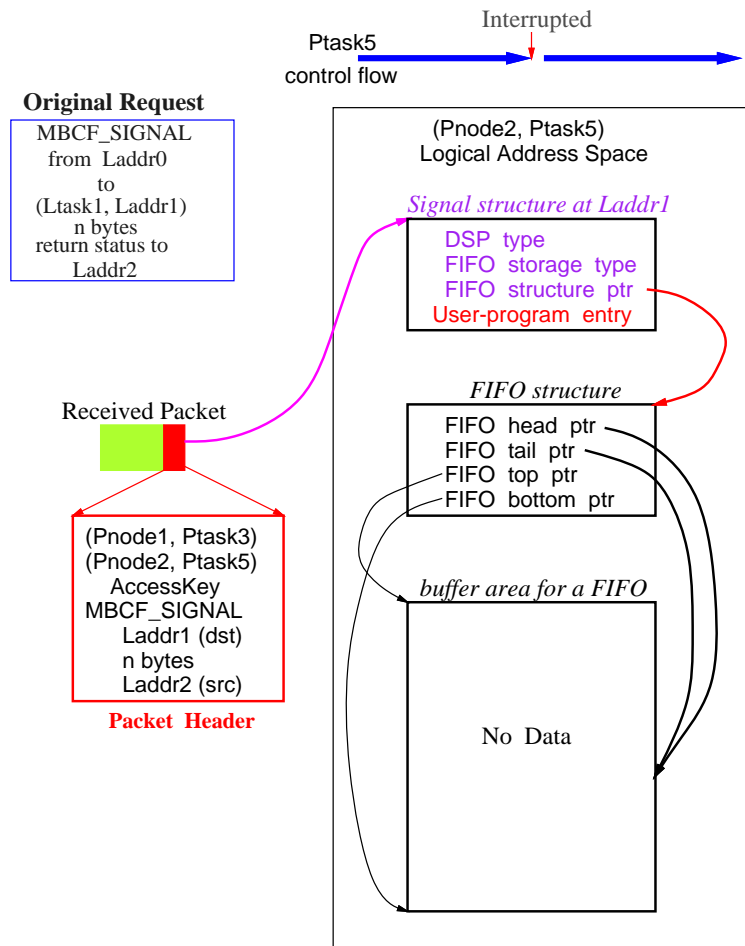


Figure 4.7: Operation of the MBCF_SIGNAL (1) command

programs is similar to that used by the signals of UNIX, and invoked programs are executed only in the scheduling periods of the target task.

An MBCF_SIGNAL command is thrown to a memory-based signal structure in the target task. The structure has a set of subcommands (invocation type, parameter-storage type, and invocation conditions), an invocation-program-counter and pointers to a data-buffer. These parameters of the signal structure allow a wide variety of MBCF_SIGNAL functions. Figure 4.7 gives a rough illustration of the operation of the signal structure for a fifo-type data buffer.

An invocation-type subcommand specifies the type of programs to be invoked. MBCF_SIGNAL commands can deliver data by invoking commands such as MBCF_WRITE or MBCF_FIFO, and can

also invoke user programs. In almost all configurations of the signal structure the information (Ltask, Pnode and Ptask of the requestor) on the received MBCF_SIGNAL packet is also stored in the buffer or the fifo where the carried data is stored.

For protection and security, the target task sets up the structure of the MBCF_SIGNAL and is able to restrict the capabilities of the MBCF_SIGNAL command. Since the information in the signal structure is important and critical, memory-pages for signal structures should be set as read-only in user mode.

The following list represents the variety of invocation type subcommands.

– MB_SIGNAL_NONE

MB_SIGNAL_NONE is used to temporarily stop functioning of the signal structure.

– MB_SIGNAL_POLLING

MBCF_SIGNAL commands to the structures with This subcommand does not invoke any user-programs at the target structure, but is able to store data and information from the received packet to the buffer or the fifo specified in the signal structure. In this case, the user checks the buffer (or fifo) by polling, and explicitly invokes programs from the resident routines to handle the data in the buffer (or fifo).

– MB_SIGNAL_ULS_INVOKE

ULS is an abbreviation of “User-Level Scheduler”, the special program for scheduling activities (threads) in the target task. At most, a single ULS is enrolled in the task structure of the OS by the target task itself. MBCF_SIGNAL commands to a signal structure which includes this subcommand invoke the ULS of the target task.

– MB_SIGNAL_DSP_BY_ULS

DSP is an abbreviation of “Destination-Specified Program” and is an ordinary user-level routine of the target task. The entry-point (i.e., the instruction pointer to the entry) of a DSP is included in the signal structure. MBCF_SIGNAL commands thrown to a signal structure which includes this subcommand invoke the ULS of the target task, the ULS then invokes the DSP, which is specified in the signal structure, with checks by the ULS for protection and security.

– MB_SIGNAL_DSP_INVOKE

The entry-point of the DSP is included in the signal structure. MBCF_SIGNAL commands thrown to the signal structure which includes this subcommand directly invoke the DSP of the target task. With this subcommand, secure active messages [80] can be efficiently emulated within the general MBCF scheme.

– MB_SIGNAL_SSP_INVOKE

SSP is an abbreviation of “Source-Specified Program” and is an ordinary user-level routine of the target task. The entry-point of the SSP is specified by the requestor and is included in the MBCF_SIGNAL packet. MBCF_SIGNAL commands thrown to a signal structure which includes this subcommand directly invoke the SSP of the target task. This configuration of the signal structure is rather risky, because the requestor can specify an arbitrary instruction pointer in the target program space.

The issues of MBCF_SIGNAL commands to a signal structure which includes this subcommand are conceptually very similar to SparcStation Active Messages (SSAM) [78], but in the MBCF scheme an arbitrary number of invocation mechanisms can be created in a single task.

– MB_SIGNAL_MEMW_SSP

This subcommand is an extension of MB_SIGNAL_SSP_INVOKE, and MBCF_SIGNAL commands to a signal structure with MB_SIGNAL_MEMW_SSP omit to save information about the MBCF_SIGNAL packets to the buffer. Moreover, in this situation, the requestor specifies an arbitrary offset for the buffer in the MBCF_SIGNAL packet. Only the pay-load data is stored at the location which is the sum of this offset and the base pointer in the signal structure. This is the most risky configuration of the signal structure, because the requestor is able to specify an arbitrary instruction pointer and an arbitrary pointer for data storage in the target space while no information about the requestor remains at the target.

The MBCF_SIGNAL with this MB_SIGNAL_MEMW_SSP subcommand allows Active Messages to be more efficiently emulated than the SSAM under general-purpose OSs, because the number of copies of data in invoked programs can be reduced by specifying an arbitrary offset.

The variety of parameter-storage-type subcommands are listed below.

– MB_SIGNAL_NONW

MBCF_SIGNAL commands thrown to a signal structure with this subcommand store neither data nor information to the memory of the target task.

– MB_SIGNAL_QUEW

This subcommand means that a fifo-queue is used as the data-storage for data and information of MBCF_SIGNAL packets. The pointers to the fifo-queue are specified in the signal structure and are maintained locally by the MBCF-interrupt routine. MBCF_SIGNAL commands thrown to a signal structure with this subcommand store carried data and information about the packet

to the fifo. If the fifo is full, the `MBCF_SIGNAL` command is cancelled just as the `MBCF_FIFO` command.

– `MB_SIGNAL_MEMW`

This subcommand means that a fixed buffer is used as the parameter-store for `MBCF_SIGNAL` packets. One fixed pointer to the buffer is specified in the signal structure. `MBCF_SIGNAL` commands thrown to a signal structure with this subcommand store carried data and information about the packet in the buffer. Since a single pointer is used for multiple `MBCF_SIGNAL` commands to a single signal structure, new data and information supersede old ones.

– `MB_SIGNAL_MEMW_OFFSET`

This subcommand means that a working area is used as the data-storage for data and information of `MBCF_SIGNAL` packets. One base pointer of the area is specified in the signal structure. One offset value is included in the `MBCF_SIGNAL` packet, and it is used to calculate the location of the data-storage. `MBCF_SIGNAL` commands to a signal structure with this subcommand store carried data in the area indicated by the base pointer plus the offset. If the invocation-type is `MB_SIGNAL_MEMW_SSP`, no information about the packet is written to the working area. Since the offset is specified by the requestor, there is some risk that old but still necessary data will be superseded by new data.

The last subcommand specifies the conditions of user-program invocation. The invocation-condition subcommands control the exclusion property of the programs that are invoked at a given signal structure. The following is a list of the subcommands for invocation conditions.

– `MB_SIGNAL EVERY_TIME_BLK` and `MB_SIGNAL_FIRST_ONLY_BLK`

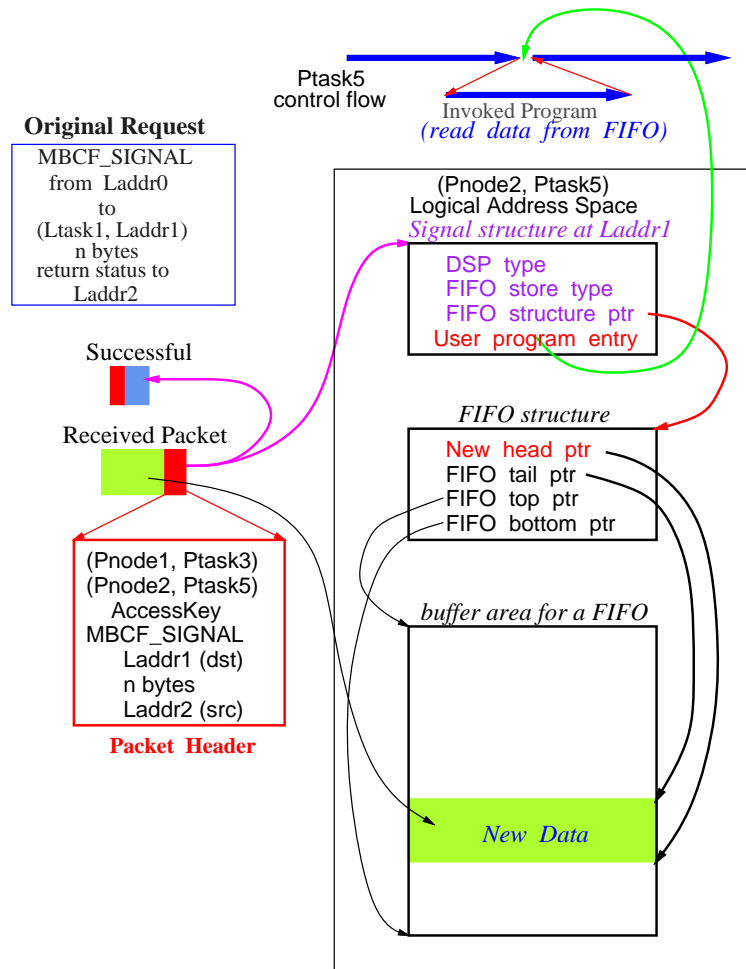
These subcommands mean that the invoked program is atomically executed. When an `MBCF_SIGNAL` arrives at the target node, the command is cancelled if the last program to be invoked is still running.

– `MB_SIGNAL EVERY_TIME_NONBLK`

This subcommand means that the specified program is invoked every time the `MBCF_SIGNAL` packet arrives. Since there is no atomicity, there is some possibility that the invoked program is recursively interrupted by itself.

– `MB_SIGNAL_FIRST_ONLY_NONBLK`

This subcommand means that the specified program is invoked only if no program that was earlier invoked from the signal structure is running. If such a program is running, the `MBCF_SIGNAL`



First_only attribute can prohibit multiple invocations at a time from the same signal structure

Figure 4.8: Operation of the MBCF_SIGNAL (2) command

packet only stores data and information in the buffer or the fifo. Note that invocations of the programs corresponding to the MBCF_SIGNAL packets are not delayed and the signal commands are completed successfully. In order to guarantee arrival of signal packets is recognized, the flag which represents that a program that was invoked earlier is still running is set by the MBCF-interrupt routine and reset by a light-weight system-call. The system-call checks the fifo-queue of the signal structure, and resets the flag if the fifo is empty. If there are some entries in the fifo, the system-call returns a pointer for the new entry. These operations of the system-call are executed atomically. This system-call is usually used at the end of the invoked program. If the

value returned by the system-call is the pointer to the new entry, the invoked program continues because of the need to process the remaining entries.

The function of a memory-based signal is specified by the combination of the three types of subcommands in the signal structure. Figure 4.8 represents actions at the target node for an `MBCF_SIGNAL` command thrown to a signal structure which includes `MB_SIGNAL_DSP_INVOKE`, `MB_SIGNAL_QUE`, and `MB_SIGNAL_FIRST_ONLY_NONBLK`.

- **Specialized commands**

Since there is no limit on the variety of functions in the `MBCF` scheme, special functions can be implemented. I list two examples of special commands for the system.

- **`MBCF_ZBUFFER`**

`MBCF_ZBUFFER` is a conditional remote-write command. The command packet has a target reference address and a reference value, along with the target address and data to be stored. In the `MBCF`-interrupt routine, the processor first compares the value at the target reference address with the value in the packet. If the target value is greater than the reference value, the processor writes the data to the target address and updates the reference value, otherwise it finishes processing received packets with no side-effects. This command is convenient for implementing z-buffer methods for computer graphics [64].

- **`MBCF_ZBUF_PACKED`**

`MBCF_ZBUF_PACKED` is a compressed version of multiple `MBCF_ZBUFFER` commands to the consecutive target locations. Data for this command are sets of structures. Each structure consists of a reference value and a datum. In the `MBCF`-interrupt routine at the target node, the processor decides, per location, whether the datum corresponding to the location should be written or not.

4.2 Options for the `MBCF` commands

Four types of options can be specified for each `MBCF` command. I list these options below.

- **Reply with a status report** This option indicates whether or not a reply on the status of the command's execution is returned to the requestor. If a reply is to be returned, it is stored at the location which has been specified by the requestor and is included in both the requesting packet and the replying packet. The requestor uses the stored status to recognize the completion of the `MBCF` command or the occurrence of some failure at the target task. This option takes one of the following values.

- **NO_REPLY_STATUS** The MBCF command makes no reply on status.
 - **REPLY_STATUS** The MBCF command makes a status reply after the operations of the command have finished. However, when the command or some other command in the same packet commits an access violation with respect to the target task, a status reply is not returned to the requestor. A special message indicating a handling violation is returned instead.
 - **REPLY_ONLY_FAILURES** The MBCF command only sends a failure-status reply when some failure has occurred in the target task during the requested MBCF process. The failure is one of the following:
 - * a fifo buffer is full;
 - * a fifo or a signal has stopped; or
 - * the conditions of a signal do not match.
- **No flow control** The MBCF scheme basically guarantees the arrival of packets and the order of their arrival. In distributed computing, however, there are many cases where requestors of communications cannot recognize whether their target nodes are alive or not. In these cases, it is not useful for the system to guarantee the arrival of packets and fifo properties, because a resending mechanism is wasteful when nodes are inactive. This no-flow-control option inhibits the functionality of guaranteeing packet-arrival and order of arrival, and there is no resending of packets for which this option is selected. The relation between MBCF commands with and without this option selected is similar to the relation between TCP/IP and UDP/IP in the IP.
 - **Elastic memory barrier** This option allows users to perform elastic memory-barrier (EMB) [35] synchronization with MBCF commands. The EMB is an extended form of memory-barrier synchronization. If the requestor specifies the use of EMB and the EMB color of the MBCF command, the request-counter for that color is incremented, and the reply-counter for that color is incremented after arrival of the reply packet. Users compare these counters to determine whether or not processing of the preceding MBCF commands of the same color has been completed.
 - **Task scheduling** This option is for blocking-type synchronization. If this option is set and the target task is not in its ready-state or active-state, the processor wakes the task up during the MBCF-interrupt routine. When requestor specifies this option, the target task may block its execution without setting any hooks for the re-scheduling. Hence, programmers and compilers are able to easily realize snoopy spin-wait (SS-wait) algorithm [39], a kind of spin-wait that includes scheduling options (self-blocking, turning over, continuing to poll) when the synchronization is not completed (refer to Figure 4.9).

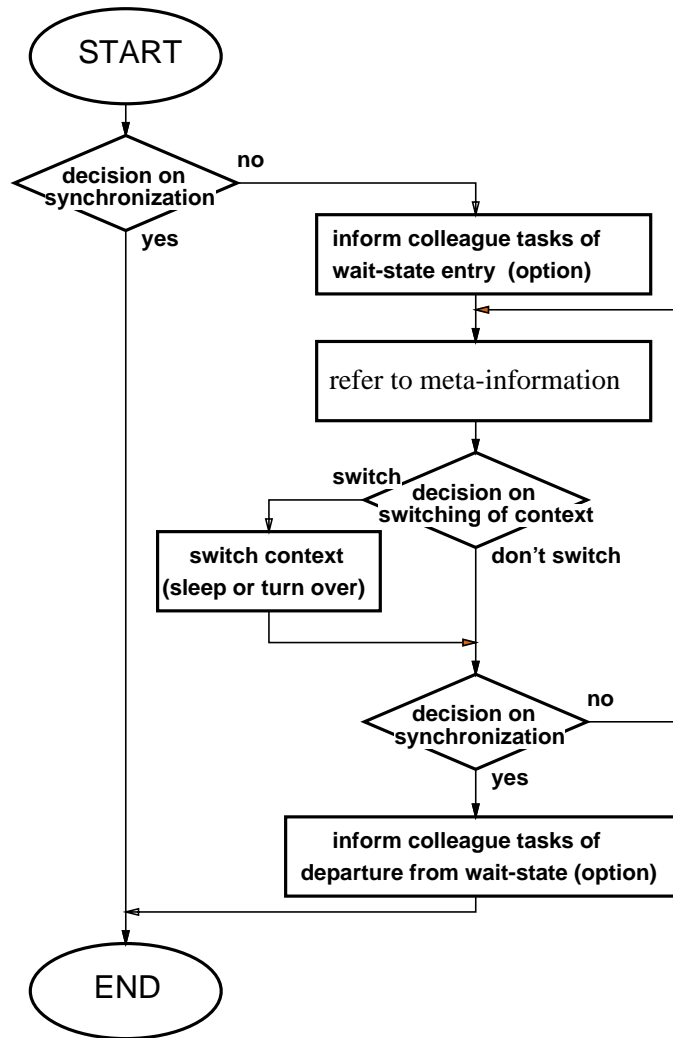


Figure 4.9: Algorithm of snoopy spin-wait (SS-wait)

4.3 The supporting commands

Here I only list two of the supporting commands of the MBCF. These are for initiating MBCF remote-message communications. Other supporting commands (system-calls for the MBCF) have already been mentioned in the explanations of the MBCF commands.

In a typical distributed-processing scheme, a client task knows neither the access-key of the server nor the locations of the working-area to be accessed. Therefore, the initial message to the server cannot be in the MBCF style, and some method for sending a special message is needed. Conventional communication

methods (e.g., TCP/IP) are available for the initial communication, but a special remote-write-message, which is called **WRITE_TASKQUE**, was prepared for the system to strengthen the security of the MBCF. A task which wants to start MBCF communications should use the **WRITE_TASKQUE** command first. If some violation of memory access occurs in the MBCF processings after channels for communication have been established, the responsibility for the violation rests with the task which issued the **WRITE_TASKQUE** (or an earlier task if other tasks have already issued the same command). Furthermore, a task which receives a **WRITE_TASKQUE** packet from some requestor acquires a strong privilege with which it can cancel the execution of the requestor task.

Each task can create, at most, one special fifo-queue whose name is “TASKQUE” in the kernel-space. When a **WRITE_TASKQUE** command arrives, the contents of the packet is stored in the **TASKQUE** of the target task, if the task has created such a queue. The target task can read the data from the **TASKQUE** by issuing the **READ_TASKQUE** light-weight system-call. This system-call is another supporting command of the MBCF.

4.4 The rule for combining MBCF commands

I described the “combined packet” and the “combining” optimization technique in the Subsection 3.3.1 “From MBP to MBCF” of Chapter 3. Since there are many types of MBCF command function, certain combinations of commands are prohibited to users. I now list these restrictions on the combining of MBCF commands.

- Restriction on the number of target tasks in a packet

I adopted the restriction that the target task of all commands in a given packet must be the same, although, technically or theoretically, multiple targets in the same physical node can be specified in a single packet. The MBCF scheme intends users to directly apply combining techniques to multiple commands, and users use logical task IDs as the target parameter. This means that users cannot easily check the coincidence of physical nodes. Moreover, this restriction allows the omission of target task parameters for all commands in a packet other than the first. This shortens the combined packet, and reduces the overhead incurred in handling it.

- Restriction on multicast commands

Those commands which circulate through multicasting routes may only be combined with other commands which have the same multicasting pattern.

- Restriction to do with the sizes of reply packets

If multiple commands in a combined packet need to return replies, these replies are always combined

in a single packet. Therefore, the reply packet cannot exceed the maximum size of a packet. This restriction means that those commands (e.g., `MBCF_READ`, `MBCF_SWAP`) which follow replies with data should not be combined with other such commands. Otherwise a requestor must carefully calculate the amount of data in the combined reply before sending a combined packet, and must then adjust the combinations of commands in the packet for sending when the result requires this.

- Restriction on the granularity of processing

The processor at a target node processes a combined packet as an atomic lump. If a page-fault trap takes place in the middle of the processing of a packet, the packet is reprocessed from its top after the page has been swapped back in. Therefore, atomic commands (e.g. `MBCF_SWAP`, `MBCF_FETCH_ADD`, `MBCF_COMP_SWAP`) should not be combined with other commands and their target areas should not cross page-boundaries. Otherwise, there is a risk that multiple atomic commands in a packet will be repeatedly executed at the target node.

Chapter 5

Discussions on the MBCF

5.1 High-speed implementation techniques of the MBCF

To make the implementations of the MBCF as high-performance as possible, I have been applying many software-engineering techniques and exploiting the advanced architectural features of latest commercial processors. In this section I list and explain these techniques. Some of the techniques have been briefly mentioned in the preceding sections.

- **Direct accesses to target logical spaces**

The most distinctive feature of the MBCF is that a requestor (sender) specifies the logical address of the target location to which some specified operation is to be applied. Since there are no intermediate queue structures such as those involved in message-passing-style communications, there is no cost arising from queue operations. If queue structures are intrinsically required by some application, the memory-based FIFO, which is a more general communication structure than conventional queues, can be used in the MBCF scheme. The memory-based FIFO will be explained in Chapter 4 “Functions of the MBCF”.

- **On-memory synchronization**

The basic scheme of synchronization in the MBCF is to use synchronization variables in the memory spaces of tasks. These are updated in the MBCF-interrupt routine so that no other synchronization with tasks is required. The tasks can check synchronization by simple direct accesses to the local variables. In the best case, where MBCF operations for the synchronization are completed before the variables are accessed for checking, on-memory synchronization by the MBCF can reduce the overhead of the system-calls that are conventionally required for inter-node synchronization. If there is nothing for a running task to do but wait for synchronization, and a blocking-type synchronization is required so that

wasteful spin-wait can be avoided, either the memory-based signal or `MBCF.WRITE` with scheduling option can be used. Both were explained in the previous chapter. The memory-based signal also provides users of the MBCF with a method of asynchronous communication.

- **Cache-conscious programming**

On the latest processors, cache-miss penalties are very large in comparison with the average number of cycles of an instruction. The programs for the MBCF-dedicated system-calls and the MBCF-dedicated interrupt routines have been developed with the aim of having as few cache-misses as possible.

- **MBCF-dedicated requesting system-call**

The system-calls of such conventional OSs as UNIX carry large overheads, because there are many copy operations which are only there for ease of programming; other unnecessary operations are only in place for portability or system-level maintenance (e.g., signal-checking and accounting). The MBCF-dedicated system-call does not include any such wasteful operations. Since the functions of the system-calls are restricted to non-blocking-type operation alone, the kernel can limit the number of save-registers and restore-registers required in the processor to as small number as possible.

- **MBCF-dedicated receiving-interrupt routine**

This MBCF-dedicated interrupt routine, which is invoked by a NIC, is similar to the MBCF-dedicated system-calls. The receiving-interrupt routine does not include unnecessary operations. It utilizes only dedicated-subroutines which are optimized for MBCF operations, and most of its subroutines are inline.

- **Wide but fixed variety of MBCF functions**

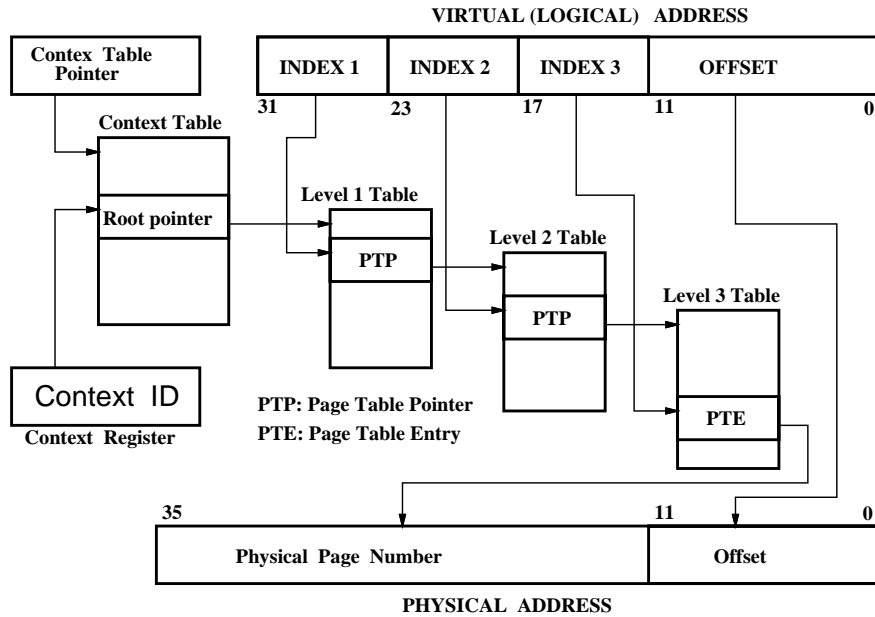
Since the user-customization of MBCF functions is prohibited, it is possible for the MBCF interrupt routine to operate directly on the target address within the kernel mode. There is a wide variety of MBCF commands, so user-level combination of the system-provided commands allows for complicated and heavy functions that do not result in performance degradation.

Special mechanisms in the processors are not assumed for the above techniques, and they are thus applicable to all computers. The following mechanisms are useful for the high-speed implementation of the MBCF, and take advantage of special features of the latest, most advanced processors (e.g., SuperSPARC [24, 67] and UltraSPARC [70, 81]).

- **TLB corresponding to coexistence of multiple contexts**

Each entry in the translation look-aside buffer (TLB) of a recent processor has a field for a context identifier (context-ID) and the OS for the processor can switch contexts without clearing entries from the

TLB. Figure 5.1(a) shows the address translation mechanism of the memory management unit (MMU)



(a) Address translation in the MMU

Context ID	Virtual (INDEX1)	Page Number (INDEX2)	(INDEX3)	Physical Page Number	ACC
------------	------------------	----------------------	----------	----------------------	-----

ACC: Access Permission Codes

(b) An entry of TLB

Figure 5.1: MMU and TLB of recent processors

of a recent processor, and Figure 5.1(b) shows an entry of the TLB of the processor. Each live task has its own unique context-ID in the node. When the OS switches the contexts of tasks, it also updates the context-ID in the context register. When the processor accesses memory or memory-mapped I/O devices, logical addresses have to be translated to physical addresses by using the TLB. If the content of the context register does not match the context-ID field of a TLB entry, the entry is regarded as not being available for current address translation. Therefore the OS does not need to invalidate all entries in the TLB when the context of tasks is switched.

During the MBCF-interrupt routine, each MBCF packet requires a few page-entries to be used for direct memory-accesses to the target space. If we use processors with TLBs of the type described, the MBCF-interrupt routine replaces, at most, only a few entries of the TLB (if the MBCF operations

frequently access the same locations, it is likely that the page entries corresponding to the locations will be resident in the TLB, and the interrupt routine can then avoid paying the penalties of a TLB-miss). The cost of replacing an entry in the TLB is low and the effect of the MBCF operations is also minimal after the MBCF interrupt has been processed.

- **Physical-address-tagged cache**

This item and the advanced TLB make for similar situations. Recent processors have internal caches with physical-address tags attached to cache entries. The OS for such processors can switch contexts without clearing entries from the caches. Since facilities for snooping on caches are implemented on most of the processors produced for use in multiprocessor configurations today, and such facilities require physical-address tags for external invalidations or updates, these physical-address-tagged caches are very popular.

- **Light-weight context switching**

This item is strongly related to the preceding two items. With advanced processors, there are no costly operations involved in switching contexts or address spaces. Therefore, the cost of access to a different context (space) is almost the same as the cost of access within the current space.

- **User-privileged memory-access capability in kernel mode**

Some smart processors allow user-privileged memory access in the kernel (supervisor) mode. If a processor has this facility, the OS as well as MBCF-interrupt routine is able to directly access user-spaces through user-specified pointers without checking access ranges. This facility is easily implemented without additional hardware, because most of the hardware resources for it are already in place for ordinary access in the user mode.

- **Page aliasing**

Since mechanisms for page aliasing among multiple memory-spaces are used to realize copy-on-write facilities and intra-node shared-memory facilities, most processors have page-aliasing facilities. As described in Section 3.3, an MBCF-requestor uses, in secure operations, a dummy space to which the true target task has allocated the required pages within the target space as aliased pages.

- **Registers dedicated to system-calls or interrupts**

In some advanced processors (e.g., the UltraSPARC) there are spare registers for use in interrupt or exception processing. Some of the ordinary registers are superseded by these spare registers at the beginning of interrupt and other exception routines. The ordinary registers are restored when the

routine ends. If we develop routines that use these extra registers alone, overheads of context switching is dramatically reduced. Moreover, if there are many sets of such spare registers so that they can correspond to various types of events, important pointers and parameters can be kept resident in the registers. We can then quickly access such data without referring to memory.

If some of the architectural mechanisms described above are not available on a specific processor, there is some loss of performance, but the MBCF can still be implemented with software emulations of these mechanisms.

5.2 A qualitative comparison with message-passing-style communication mechanisms

In most conventional systems, message-passing-style interfaces are popular as well as for communications in user-programming interfaces for communications at system interfaces (in other words, the kernel-user interfaces or system-call interfaces). The MBCF is primarily a system-interface but can be directly used as a programming interface. By using additional user-level codes, any message-passing-style interface can be realized in the MBCF scheme. Conversely, all functions of the MBCF may be carried out with any system-interface of the message-passing-type and some additional user-level codes. The selection of programming interfaces is purely an issue of taste and must be left for programmers and language-designers to decide. Therefore, the problem is to determine the types of system-interface that are suitable for parallel and distributed systems.

In this section I abbreviate message-passing-style system-interfaces as “MPSI”. The definition of an MPSI is as follows. The MPSI has at least these two system-calls:

- `send(destination-task, &message);`
- `receive(source-task, &message);`

. To attain high-speed execution, both system-calls should be implemented as non-blocking-type calls and some other methods of synchronization that can detect the end of execution of such calls are available. To increase expressive power, the source-task parameter in the “receive” call can be a wild-card.

The methods of implementation available in an MPSI are less flexible than those of the MBCF. I explain this difference in flexibility using MBCF-style terminology.

- The MPSI limits target logical addresses to only one (or a few) implicit message-buffer (i.e., fifo-queue) address(es). Though users can add some logical target addresses to a packet even in the MPSI system, the communication mechanism of the system cannot directly handle these addresses because they are

only part of a user-level payload. In the MBCF system, however, the communication mechanism directly processes the target addresses in a packet and consequently reduces the number of copies of data that must be generated.

- The MPSI also limits functions to a single type “MBCF_FIFO” (a simple remote fifo write) for one (or a few) implicit fifo(s). Since the MPSI system uses implicit target addresses, variations of the functions are meaningless. In the MBCF system users are able to specify not only arbitrary target locations but also to select operations (MBCF-commands) from among a wide variety of functions on each request.
- In the MPSI system a correspondence among between the messages sent and received is essential. Hence the order of execution of the messages is inflexible. In the MBCF scheme, users are able to directly exploit a number of user-space buffers for communications and a number of user-space flags for synchronization, and the MBCF commands for different target locations are independent of each other. Therefore the order with which MBCF requests are issued is more flexible than the order of sending and receiving in an MPSI.

These differences in terms of flexibility often cause a big difference in performance. For example, because the implicit message-buffer of the MPSI is implemented in the kernel space, data must be copied into another buffer in a user-space when the user wants to use them. The inflexibility in terms of target-address specification means that more copies of data must be generated in an MPSI than in an MBCF.

To make the best use of its flexibility, the MBCF system makes direct accesses to user-spaces from the interrupt routine (in kernel-mode). However, as described in Section 5.1, high-end processors in recent times (such as SuperSPARC [24] or UltraSPARC [70]) include the following mechanisms for realizing memory-access to user-spaces from the kernel-space without incurring penalties.

- The processor is able to change the current task-space (context) without purging all TLB-entries.
- Many pages from various task-spaces can exist at the same time in a processor’s TLB.
- Changing the current context is inexpensive and only costs one instruction and a few clock cycles.

Therefore, without paying any penalties in its interrupt routine the MBCF has many more areas of flexibility than the MPSI. Moreover, the MPSI needs an additional system-call to receive a packet while in the MBCF users can only receive a packet by polling locations in the user-space memory. From the viewpoint of performance the MBCF interface is much better than the MPSI. As described at the beginning of this section, from the point of view of expressive power (semantics) the two are equal. As a consequence, the MBCF interface is superior to the MPSI.

5.3 A qualitative comparison with the Active Message

The Active Message [80] (AM) ¹ was originally invented for the direct and high-speed execution of dataflow-style programs on the bare hardware of parallel computers. There are thus no mechanisms for protection, security or virtualization. The primary feature of the AM is that a user-level receive-routine is selected for each message and the receive-routine for a message is specified in the message itself (the pointer to the entry point of the receive-routine is included in the message).

The SparcStation Active Message [78] (SSAM) is an extension of the AM for workstation clusters with general-purpose OSs. In the SSAM scheme, a message (packet) is explicitly prepared by users and transmitted through the SSAM-dedicated system-call. The light-weight implementation of the packet-sending procedure of the SSAM is almost the same as that of the MBCF but there is no access-key for security and no virtualization for task-migration. The receiving procedure of the SSAM is very different from that of the MBCF. Since the receiving-interrupt routine is executed in kernel-mode and directly operates the hardware registers of the NIC, the user-level receive-routine which is specified in the SSAM message must be invoked indirectly through some kernel-level interrupt routine. The kernel-level temporary receive-routine is invoked on every interrupt of the NIC, and the routine receives the packet in the target-task's buffer in the kernel-user-shared space. After the packet has been received in the buffer, a mechanism like the signal mechanism of UNIX is used to invoke the specified user-level receive-routine, and the user-level routine uses data in the temporary buffer to perform customized functions.

On the other hand, while it guarantees protection and virtualization, the MBCF's receive-routine is directly invoked in kernel-mode on every packet-arrival interrupt. Since the user-customization of functions are prohibited in the MBCF system, the receive-routine can be kept safe and fair, and the upper limit of cost for an operation requested by an MBCF packet is known before its execution by kernel.

Copying to the temporary buffers of the SSAM represents an additional overheads in comparison with the MBCF. Moreover, in the SSAM system there is a probability that the invocation of the use-level receive-routine and the reply of the message will be delayed because the routine is only invoked and executed when the target task is scheduled in core. In the case of simple remote-memory access, it's very likely, that the invoking of user-level receive-routines will carry other overheads. The SSAM is qualitatively inferior to the MBCF on these three points. On the other hand, its flexibility in that it is possible to customize receiving functions perfectly is the strong point of the SSAM. In the MBCF system, however, there is enough flexibility on the variety of MBCF commands to prepare critical primitives for the functions which are performed in user-level receive-routine of the SSAM. Therefore, the perfect customizability of receiving functions is less

¹The light-weight communication mechanism of ABCL/AP-1000 [74, 75] is almost the same as the AM.

significant than the number of data copies that must be generated.

5.4 Related Works

5.4.1 Fast Message

The Fast Message (FM) [60] is a software solution for fast user-level communications and is one of the earliest mechanisms to have been called a “zero-copy” protocol. In the FM scheme, communication devices (i.e., NICs) are directly mapped to the user-task space so that user can directly manipulate the NICs’ registers. There is thus no protection and no virtualization, so this method is not applicable to general-purpose systems or in situations where it has to share NICs with conventional communication protocols (e.g., TCP/IP, UDP/IP). I insist that the FM and similar mechanisms should be called “zero-safety” protocols and that they are only useful as the topics of papers.

5.4.2 Active Message

The Active Message (AM) [80] was originally invented for the direct high-speed execution of dataflow-type programs on the bare hardware of parallel computers, and there are thus no mechanisms for protection, security, or virtualization. The SparcStation Active Message (SSAM) mechanism, described in the next subsection, is based on the AM and supports protection. The SSAM is comparable with the MBCF, but not the AM.

5.4.3 SparcStation Active Message

The SparcStation Active Message (SSAM) [78] was developed for the efficient implementation of the AM on general-purpose workstations. The inventor(s) of the SSAM noticed that the overhead cost of system-calls is not intrinsically high. Dedicated system-calls were thus implemented for the SSAM. The MBCF independently adopted the same approach of using dedicated system-calls but publication of a report on the MBCF was a year later than the corresponding date for the SSAM. There are, however, many differences between the MBCF and the SSAM. For a detailed discussion, please see Section 5.3 “A qualitative comparison with the Active Message” of Chapter 5 “Discussions on the MBCF”.

5.4.4 PM

The PM [76, 77] is a software solution for fast communication on machines using Myricom’s NIC (Myrinet [5]) cards. The PM originally had only message-passing-type primitives for communications and synchronization. There is an embedded programmable processor in the Myrinet card, and the PM exploits the processor to

invoke the card's DMA mechanism and to transfer data from the card to main memory. The PM with the Myrinet cards attained a remarkable low latency because the main processor directly accesses and polls the embedded memory in the card. This technique is not applicable to typical conventional NICs, and the MBCF/Ether attains a much better latency than the PM with ethernet NICs [69, 68]. Of course, this comparison is not rigorous since the hardware environment for measurement of the PM is different from the equivalent environment for the MBCF, though 1000BASE-SX is used in both cases. The PM has a significant weak point in that the PM requires a buffer area **in the Myrinet card for each user-application task**. However, the card cannot allocate enough buffers to support multiple tasks at one time. This requirement for embedded memory imposes a strong limitation in terms of the number of simultaneous communication tasks. Therefore, to construct a multi-tasking system for multiple parallel tasks, the PM must apply a gang-scheduling policy that simultaneously switches tasks in the hosts and the buffers aboard the Myrinet cards across the whole system. The overhead cost of the switching required by the fine-grained gang-scheduling is not low, so the PM is not suitable for use on the general-purpose systems with short time slices for context switching. In passing, it should be mentioned that there is no limitation on task switching in the MBCF system.

In the PM scheme every task requires its own buffers and control structures for communications. In other words, a node of the PM requires a buffer area which is proportional to the total number of target **tasks** of various active applications. A node of the MBCF, however, only needs a buffer area which is proportional to the number of target **nodes**. In short, the MBCF is much more scalable than the PM.

A new version of the PM specification supports a few remote memory operations, but, in the PM, target locations must be pinned down to physical main memory. Of course message-passing-type operations of the PM also requires buffers for receiving to be pinned down, since embedded processors in the Myrinet cards cannot handle virtual memory facilities. As for the MBCF, pin-down actions for target locations are unnecessary because of exploiting MMU/TLB mechanisms of the host processors.

5.4.5 U-NET

The U-NET [79] is a framework for making direct user-level communication handlers that use conventional NICs. Even if we use the U-NET framework, with ordinary and conventional ethernet NICs it is not possible to avoid system-calls to maintain protection of the packet-header. In the U-NET framework, receiving addresses are specified in the control structures of the receiving tasks, and this creates a potential overhead in comparison with the MBCF in which the sending tasks can specify receiving addresses for the receiving tasks. In the U-NET scheme, as in the PM every task requires its own buffers and control structures for communications. The MBCF is thus, again, much more scalable than the U-NET.

Chapter 6

An Implementation of the MBCF using Ethernet

6.1 Specifications of the MBCF/Ether

6.1.1 Background to the MBCF/Ether

I adopted the ethernet as the transportation layer of the first sample implementation of the MBCF. I call it the “MBCF/Ether”.

The ethernet is the most popular device in local-area networks (LANs) and is also promising as a candidate for high-speed communications within and between clusters of workstations and of personal-computers. The current ethernet system includes three speed-variants: 10BASE (so-called “Ether”) [20], 100BASE (“Fast Ether”) [19], and 1000BASE (“Gigabit Ether”) [21]. The latter became an international standard in November 1998. The next standard for “10 Gigabit Ether” is now being drawn up. Each of these variants includes several variants for operation on several media. Cheap and thin twisted-pair cable is the medium for 10BASE-T, 100BASE-TX and 1000BASE-T. Ethernet in this form is thus very popular for use in LANs and the required NICs have already become inexpensive commodities.

The ethernet system cannot guarantee the arrival of transmitted packets because of the CSMA/CD [20] protocol and the cancellations of packets that is the basis of its dead-lock-free characteristic. The MBCF/Ether includes an original protocol to guarantee the arrival of packets and the fifo-ness of point-to-point communications without degrading its performance in terms of communications and synchronization.

6.1.2 Packet format of the MBCF/Ether

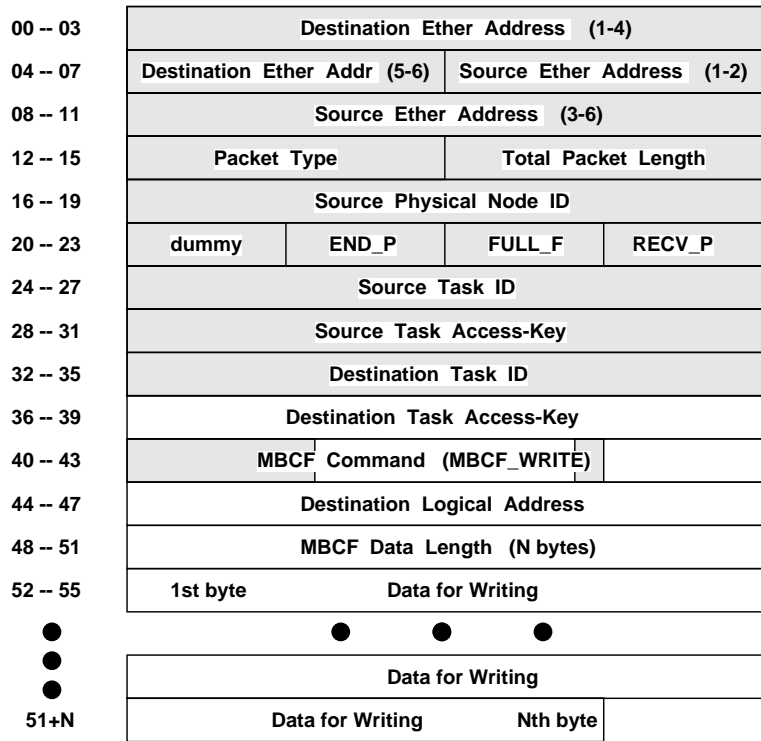


Figure 6.1: Format of a single MBCF.WRITE packet

I show the MBCF/Ether packet that carries an MBCF_WRITE command in Figure 6.1¹. In this subsection I use a byte-addressing (octet-addressing) scheme which represents locations from the top of an MBCF/Ether packet. The area from address:0 to address:13 is the header of the Ether packet. The value of “Packet Type”(address:12) is a specific value for the MBCF/Ether packets, and is different from the values that indicate conventional packet types. The field “Total Packet Length” (address:14) shows the total length of the Ether packet, and this is expressed in bytes. The 4byte field at address:16 represents the “Source Physical Node ID” (Pnode) of the MBCF system. I could have adopted the same value as the IP address of the node for the value of Pnode, but to keep implementation of the MBCF simple I have adopted independent ID numbers within the MBCF system. The byte at address:20 is reserved for future use or for debugging of the MBCF/Ether system. The Three bytes of data from address:21 to address 23 are used by the MBCF/Ether protocol to guarantee performance, fineness, and the arrivals of packets. The details of the protocol are described in Subsection 6.1.4 “Protocol of the MBCF/Ether”. The 4byte field at address:24 is for a physical “Source Task ID” (the requesting Ptask) and the field at address:28 is used to store a “Source Task Access-Key”, which

¹This packet format is for 32bit address-spaces, so a little modification is required to use it with 64bit address-spaces.

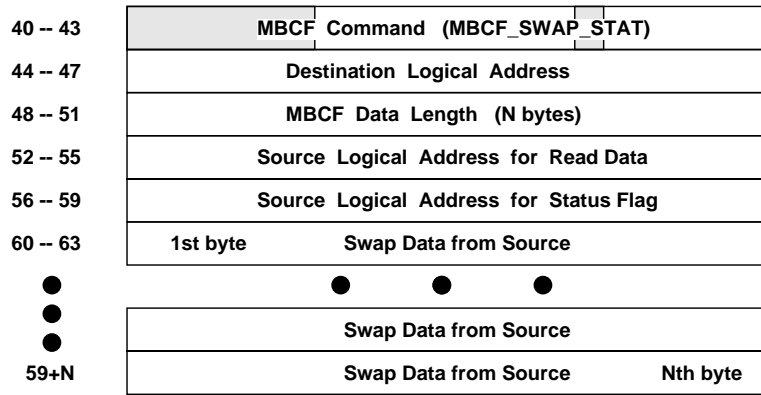
enables the reply-packet to access the source-task space. The 4byte value at address:32 represents a physical “Destination Task ID” (the target Ptask).

For protection and security the data from address:0 to address:35 are stored to the packet by the kernel-level routine of the MBCF-requesting system-call. The final values for “Destination Ether Address” and “Destination Task ID” are calculated by referring to tables in the system-call. The fields from address:36 to address:51+N take the user-specified data without any modification or checking by the system-call. The only exception: some bit fields, which are used to control the commands of the MBCF/Ether protocol, in the “MBCF Command” word at address:40 are properly set up by the system-call routine. The hatched parts of the packet in Figure 6.1 are set up by the kernel, and the non-hatched parts are arbitrarily specified by users.

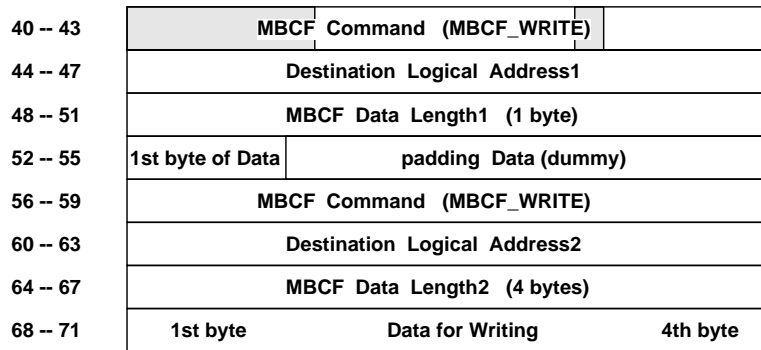
A “Destination Task Access-Key” (access-key for the target task) is stored at address:36, and an “MBCF Command” is written at address:40. The command illustrated in the figure is MBCF.WRITE and MBCF.WRITE takes three parameters: “Destination Logical Address” (target Laddr) at address:44, “MBCF Data Length” (amount of data to be stored) at address:48, and data to be stored from address:52 to address:51+N. Here, N is the length of the data.

In Figure 6.2 I give examples of two other MBCF/Ether packets. Since the fields from address:0 to address:39 are the same as those in Figure 6.1, they are omitted in Figure 6.2. Figure 6.2(a) represents an MBCF_SWAP_STAT (MBCF_SWAP with a status report option) command packet. The MBCF_SWAP_STAT command takes five parameters: “Destination Logical Address” at address:44, “MBCF Data Length” at address:48, “Source Logical Address for Read Data ” at address:52, “Source Logical Address for Status Flag” at address:56, and “Swap Data from Source” from address:60 to address:59+N. After completion of the swap operation at the target task, the N bytes of data which were stored at “Destination Logical Address” before the operation, a requestor-side pointer for the N bytes of consecutive data for reading, a status value “SUCCEEDED” and a requestor-side pointer for the status value, are returned to the requester through a reply packet. When the reply packet arrives, the N bytes of data that have been read and the status value are stored by using the respective pointers.

The Figure 6.2(b) is an example of a combined packet. The example contains two MBCF.WRITE commands (a 1byte remote-write and a 4byte remote-write). The destination (target) tasks, nodes, and access-keys of these two commands are common and such fields of the packet are shared. The combined packet is thus compact and inexpensive to transmit. For quick access by node processors, the second and later commands are aligned on a 4byte boundary of the packet by padding with dummy data as necessary.



(a) MBCF_SWAP_STAT Packet



(b) Combined MBCF Packet (two MBCF_WRITE)

Figure 6.2: Two other MBCF packets

6.1.3 User interfaces of the MBCF/Ether

The MBCF-requesting system-calls are light-weight system-calls in that they present no possibilities of context switching within a call and thus never require many save-and-restore operations for management of register contexts. The MBCF-requesting system-call interfaces in the MBCF/Ether system are of two types. I list them below.

1. **User-level construction** of request-packets:

- user applications make the bodies of MBCF-request packets
- combined (merged) packets are easily and directly constructed by user applications

2. **Kernel-level construction** of request-packets:

- pointers to the data to be sent are handed over to the system-call
- it is unnecessary in the user-mode to copy data into packet buffers

User-level construction is suitable for the MBCF policy according to which operations should be performed in user-mode as much as possible. In the MBCF/Ether implementation, however, copying of data by the kernel is necessary at least once, in order to guarantee the arrival and fineness of packets and to use packet-cancellation methods to maintain the dead-lock-free characteristic. Therefore, kernel-level construction takes advantage of the reductions in the copying of data, and is less expensive than user-level construction in those cases where the amount of data in the packet is relatively large. Kernel-level construction is the primary interface in the MBCF/Ether.

6.1.4 Protocol of the MBCF/Ether

In this subsection I describe the protocol of the MBCF/Ether. The protocol guarantees the arrival of packets and point-to-point fineness without degrading system performance. As I mentioned in the subsection “Background to the MBCF/Ether”, there is a small possibility of the loss of transmitted packets on the way of ethernet connection line. Although this characteristic “packet-loss” may seem to be a weak point or a defect of the ethernet system, we cannot make such a sweeping generalization. Communication hardware that allows a possibility of “packet-loss” is simpler than that which doesn’t and the hardware can thus be faster. Therefore, if we can develop an efficient protocol for use with hardware that allows a possibility of “packet-loss”, we can take advantage of an inexpensive but fast communication system. In this sense, the MBCF/Ether protocol is significant enough to be explained in this thesis.

In the MBCF/Ether system I assume that all nodes exist in an ethernet segment and that they are connected via switching hubs or non-switching hubs. Algorithms for congestion avoidance and slow-start, such as those of TCP/IP, in which the sizes of packets are modified when they are resent time, are useless because either the CSMA/CD protocol or the full-duplex flow-control of the ethernet is efficient and sufficient to avoid contentions in the circumstances of the MBCF/Ether. Moreover, users can avoid congestions with SS-waits that use information on ethernet congestion. These conditions of the circumstances of the MBCF/Ether suggest that the possibility of “packet-loss” is rather low and we need a mechanism which resends the same packets while maintaining the point-to-point order of packets when some packets are lost.

We can imagine another approach in which user-level routines maintain and guarantee both the fineness and the arrival of packets with the assistance of the invocation of user-level routines by timer interrupts. Some systems (e.g., FM [60], U-NET [79], VIA [7]) actually adopt this approach. In systems with the majority of conventional NIC types, however, this approach is not only very complex for users but also presents a big

drawback in terms of performance. The NICs require a kernel-level buffer area so that DMA transfers can be used to transmit packets. In a kernel-level approach, this buffer can be exploited as the buffer for use in resending, but in a user-level approach it is invisible and an increase in the amount of data-copying cannot be avoided. Furthermore, each user-level task in a user-level approach has to prepare control structures and buffers for flow-control, but in the kernel-level MBCF/Ether approach kernel-level control structures and buffers for flow-control can be shared by all user-tasks of a node. This characteristic of the MBCF/Ether is very significant in terms of the scalability of the system. I decided to guarantee the fineness and the arrival of packets in the kernel-level routines.

My MBCF/Ether protocol is an extension of the packet-level window-based (GO-back-N) technique for flow control. I explain the protocol in outline below. Every node maintains sequence numbers of two kinds for each remote node: one is the number of packets sent and the other is the number of packets that arrive. Whenever a packet is transmitted to a node, the packets-sent number for that node is incremented. The number sent is attached to the "END_P" field (address:21) of the requesting packet (See Figure 6.1). Whenever a packet is received from a node, the packets-arrived number for that node is checked against the sent number in the packet. If the packets-arrived number is one lower than the packets-sent number, the packet has been properly received and is then processed by the MBCF-interrupt routine. The packets-sent number of the packet that has arrived is stored as the packets-arrived number (or rather, the packets-arrived number is incremented). If not, the difference between the packets-sent number of the received packet and the expected number is the number of possibly lost packets. A resend-requesting packet is in this case transmitted to the original requestor with a copy of the packets-arrived number. When a resend-requesting packet arrives, the number of lost packets is calculated from the packets-arrived number in the packet, and the lost packets are then re-transmitted, in turn, with a handshake algorithm. I use the handshaking during the resending procedure, because the loss of some packets suggests congestion of the ethernet system and an eager transmission would appear to be possibly harmful.

To collect free buffers for use in resending, the packets-arrived number is also sent in the field "RECV_P" (address:23) of every packet. The number of buffers that corresponds to the numbers required as calculated from the packets-arrived number in a received packet are freed and collected by the system. If uni-directional communication to a node is continuous and the number of uncollected buffers for the node reaches a threshold (TH1), the MBCF/Ether system attaches an ack-request flag and requests that an acknowledge-packet be returned with the packets-arrived number of that node. This TH1 corresponds to the N of the GO-back-N algorithm. After reaching the TH1, the MBCF/Ether system permits users to send more packets to the node until the number of uncollected buffers reaches another larger threshold (TH2). The loss of acknowledge

packets, resend-requesting packets or handshaked packets during resend procedures is detected by timeouts, and additional packets that correspond to such lost-packets are transmitted.

The actual MBCF/Ether kernel-level routines can also handle the global lack of buffers. In the usual GO-back-N algorithm TH1 (N) is a fixed number, but in the MBCF/Ether protocol TH1 is decreased at a hint of a lack of buffers. Uncollected buffers which have been used for previous communications are also withdrawn by sending ack-requesting packets when a shortage of buffers arises. The sudden shutdowns of nodes and the sudden supply of power to inactive nodes are also supported in the MBCF/Ether protocol. The details of the actual algorithm are too lengthy to give here, but the buffering mechanisms described above can avoid declines in performance in a typical unsaturated communications. Cache-conscious programming techniques and trace-preference programming techniques mean that the complicated MBCF/Ether protocol does not carry a large overhead.

Of course, the MBCF/Ether protocol is consistent with other protocols on ethernet because it has its own protocol-type. Actually my operating system can simultaneously handle MBCF/Ether, TCP/IP, UDP/IP, ICMP and so forth over an ethernet link.

6.2 Evaluation using SPARCstation 20s and SSS-CORE Ver.1.x

The *SSS-CORE* [36, 53, 55] is my original scalable operating system for workstation clusters and the *SSS-CORE Ver.1.x* is the name of a series of editions of this OS for SPARCstation 10 and SPARCstation 20 from Sun Microsystems. Both machines are powered by SuperSPARC processors [24].

6.2.1 Environment used for performance evaluation

I used the following workstation cluster to implement and evaluate the MBCF/Ether protocol.

- Node of the network of workstations (NOW)
 - Axil 320 Model 8.1.1
(Sun SPARCstation 20 compatible, 85MHz SuperSPARC x 1, 32bit 25MHz SBus)
 - Sun Fast Ethernet SBus Adapter 2.0
- Network
 - non-switching hub connection for 100BASE-TX
 - non-switching hub connection for 10BASE-T
 - switching hubs were also used for 100BASE-TX and 10BASE-T

- non-switching hubs were used in the following measurements unless otherwise specified.
- Operating system of the NOW
 - SSS-CORE Ver.1.0
 - * general purpose (multitasking with parallel task execution)
 - * scalable
 - * parallel and distributed processing
 - * high degree of optimizability for user-code
 - * test bed for the MBCF and the compiler-assisted DSM system
 - * supports the SuperSPARC processor (SPARC V8 [67] architecture)
 - * development from scratch

This section is a report on various aspects of the performance of the MBCF/Ether protocol on a real system, including the overheads incurred by using MBCF/Ether. Most of the results fluctuate to a greater or lesser degree according to the situation of the processor caches and of memory-allocation. When thrashing of the caches arises, the results become rather bad. The occurrence of cache-thrashing is unusual for advanced processors with multi-way set-associative L1 caches and direct-mapped large L2 cache(s). Therefore, in this thesis results are only reported for in the non-thrashing cases.

6.2.2 Overheads of MBCF/Ether

In this subsection I measure overheads carried by the MBCF/Ether protocol. I define the sending overhead and the receiving overhead as follows.

- Sending overhead: cost in time of an MBCF-request system-call
- Receiving overhead: cost in time of the interrupt routine for packet arrivals

I used MBCF.WRITE commands and the kernel-construction-type interface to measure these overheads.

I started by referring to the clock-counter LSI in the node to determine the sending overhead. The base clock of the LSI runs at 2.0MHz ($0.5\mu s$ cycle). The cost in time of referring to the LSI is about $1.2\mu s$. This value is rather high, and this is because the LSI is attached to the I/O subsystem and the node processor must pass through two external buses to access it. Note that results of measurement using the clock-counter are overestimated owing to this cost.

Table 6.1: Sending overheads of MBCF_WRITE packets

amount of data (byte)	4	16	64	256	1024
cost (μ s)	5.0	5.5	6.0	8.5	20.0

Table 6.1 shows the results of these measurements. The columns represent the amount of data included in the MBCF_WRITE command.

I then used a logic analyzer to time the sending overhead and thus acquire accurate values of the sending overhead. In this measurement I used an MBCF_WRITE_STAT, which is an MBCF_WRITE command with a status-report option, with a fixed 4 bytes of data, because an effect of different amount of data was indicated by the previous measurements. Figure 6.3 shows a screens of results from the logic analyzer, displaying the signal waves at the requestor node. Signals from A0_03 to A0_06 correspond to data transmitted over the 100BASE-TX connection, and signals from A1_01 to A1_04 are the received status-reply data. Signal C_03 represents the state of the supervisor-access pin of the SuperSPARC-II. The pin's low state means the processor is in its supervisor-mode (kernel-mode).

In Figure 6.3, the first low state on C_03 corresponds to the period of the MBCF-dedicated receiving-interrupt routine for a status reply. The second low state of C_03 is the period of the MBCF-dedicated requesting system-call for an MBCF_WRITE_STAT, and the third low state is the period of the procedure for completion of a transmission. The length of the second low state of C_03 is thus the sending overhead of the MBCF/Ether protocol, and the value in the figure is **3.8 μ sec**. I used the above method to measure the length of this signal many times and the best result for this sending overhead is **3.4 μ sec**.

These measurements are only accurate provided that the 100BASE-TX interface card (Sun FastEthernet Adapter2) is used as the NIC for MBCF/Ether communications. However, I can estimate that the overhead corresponding to the 10BASE-T NIC (built-in interface) is the almost as same as that for the 100BASE-TX NIC because almost all procedures for both NICs are common.

It is difficult to measure the receiving overhead by referring to the clock-counter LSI, because the receiving routine is invoked by an interrupt of the NIC. Therefore, I used the logic analyzer to time the receiving overhead. The first low state of C_03 in Figure 6.3 corresponds to the period of the procedures for receiving a status reply. I can use this value as the receiving overhead but I should more properly measure the signals at the target-side node. Figure 6.4 shows consecutive MBCF_WRITE commands with 4 bytes of data and without status reporting options. Every low state of C_03 in the figure corresponds to the period of the MBCF receiving-interrupt routine at the target node. The first low state lasts for 13.85 μ sec and is rather long in comparison with the other low states. During this period, the node receives two MBCF packets. This

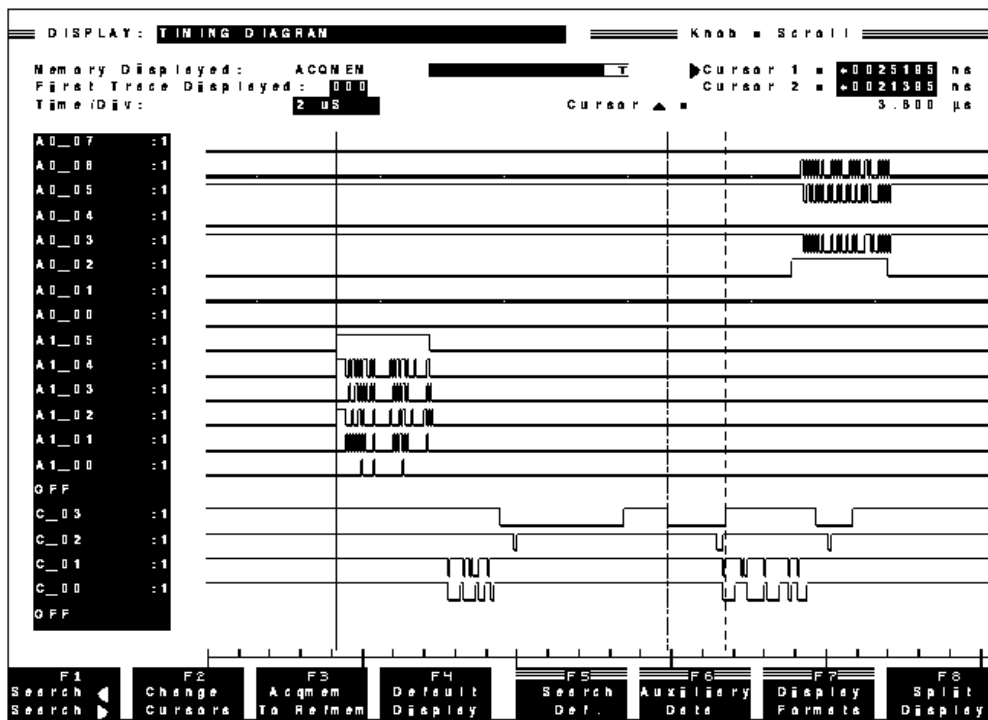


Figure 6.3: Zoomed signal wave forms of a single 4byte MBCF.WRITE

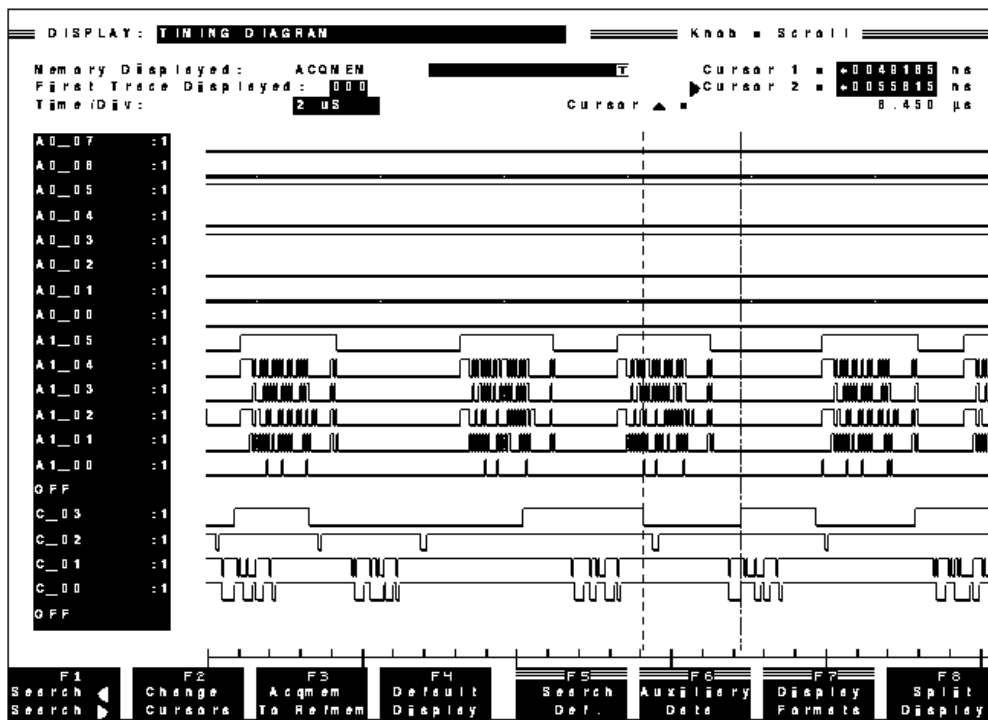


Figure 6.4: Signal wave forms for the receiving of consecutive 4byte MBCF.WRITES

Table 6.2: Peak data-transfer rates for the MBCF/100BASE-TX and the MBCF/10BASE-T

data size (bytes)	4	16	64	256	1024	1408
100BASE-TX (Mbyte/s)	0.34	1.27	4.82	9.63	11.64	11.93
100BASE-T (Mbyte/s)	0.31	1.15	4.31	8.56	11.13	11.48
10BASE-T (Mbyte/s)	0.04	0.17	0.48	0.89	1.13	1.17

fact is indicated by the two activations of the C_02 signal, because each activation represents an access to the NIC in order to check and reset the NIC's interrupt. The second low state of C_03 lasts for **6.45 μ sec** and the third lasts for **6.35 μ sec**. These results indicate that the overhead of the MBCF receiving-interrupt routine is about **6.4 μ sec**.

6.2.3 Peak data-transfer rate of MBCF/Ether communications

Table 6.2 shows peak data-transfer rates of communications using the MBCF/100BASE-TX and the MBCF/10BASE-T. I used MBCF_WRITE commands with various amount of data to measure the data-transfer rates. The figures in the table are net quantities which correspond only to the payload of data and do not include the header of the MBCF/Ether packet or additional data for the ethernet protocol. The method of measurement was to have a requestor repeatedly send MBCF_WRITE commands to a fixed target and only checking for acknowledge-messages once every 16 transmissions. This checking every 16 transmissions is used to avoid saturation of the ethernet connection. The ideal values of peak data-transfer rates are 12.5Mbyte/s for the 100BASE-TX line and 1.25Mbyte/s for the 10BASE-T line, and these ideal values correspond to all transmitted data, including all headers and all additional signals for the protocols. Therefore, the highest result, 11.93Mbyte/sec in Table 6.2 indicates that the performance limits of the NICs and/or the ethernet systems are bottlenecks on the peak data-transfer rates for MBCF/Ether communications. In the table, 100BASE-TX indicates full-duplex transmission method and 100BASE-T indicates half-duplex transmission.

To compare the MBCF/Ether with ordinary communication methods in a conventional OS (SunOS 4.1.4), I measured the peak data-transfer rates of the TCP/IP [62] and the UDP/IP [61] in the same hardware environment. In these measurements I used the socket libraries of the SunOS 4.1.4. To cope with measurement for fine-grained communications, I added the TCP_NODELAY option to the TCP/IP sockets. Because of The same protocol as guarantees packet-arrival and fineness for the MBCF/Ether was added to the UDP/IP transmissions, in terms of user-level routines. I refer to TCP/IP communications via the 10BASE-T and 100BASE-TX connections as "TCP10/SunOS" and "TCP100/SunOS", respectively. The corresponding UDP/IP communications are referred to as "UDP10/SunOS" and "UDP100/SunOS".

Figure 6.5 shows the peak data-transfer rates for MBCF100 (MBCF/100BASE-TX), TCP100/SunOS,

and UDP100/SunOS. The x-axis in the figures represents the amount of data in a single ethernet-packet,

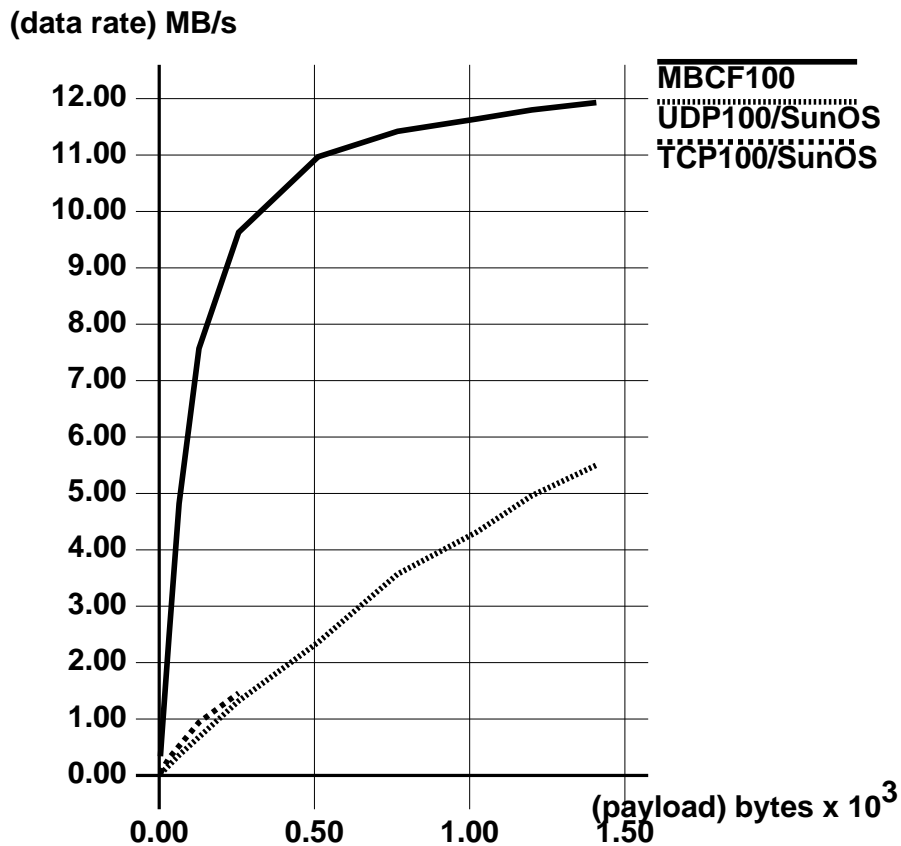


Figure 6.5: Peak data-transfer rates for MBCF100, TCP100, and UDP100

and the y-axis is the rate of data-transfer. With TCP100, a dramatic drop in performance was observed at a 512byte packet size. This is due to the TCP-specific protocols (congestion avoidance and/or slow start). These protocols are unsuited to eager and repeated transfers. The start of the slopes for UDP100/SunOS and TCP100/SunOS are much easier than for MBCF100. Although the curve for MBCF100 is almost saturated for amounts of data above 1024 bytes, owing to the limitations of the hardware, UDP100/SunOS is not able to reach half of the maximum performance of the MBCF100.

For your information, I show the peak data-transfer rates for MBCF10 (MBCF/10BASE-TX), TCP10/SunOS, and UDP10/SunOS in Figure 6.6. The 10BASE-T system is too slow for MBCF/10BASE-T to take advantage of its low-overhead routines. We only find a small difference between the performance of MBCF/10BASE-T and UDP10/SunOS.

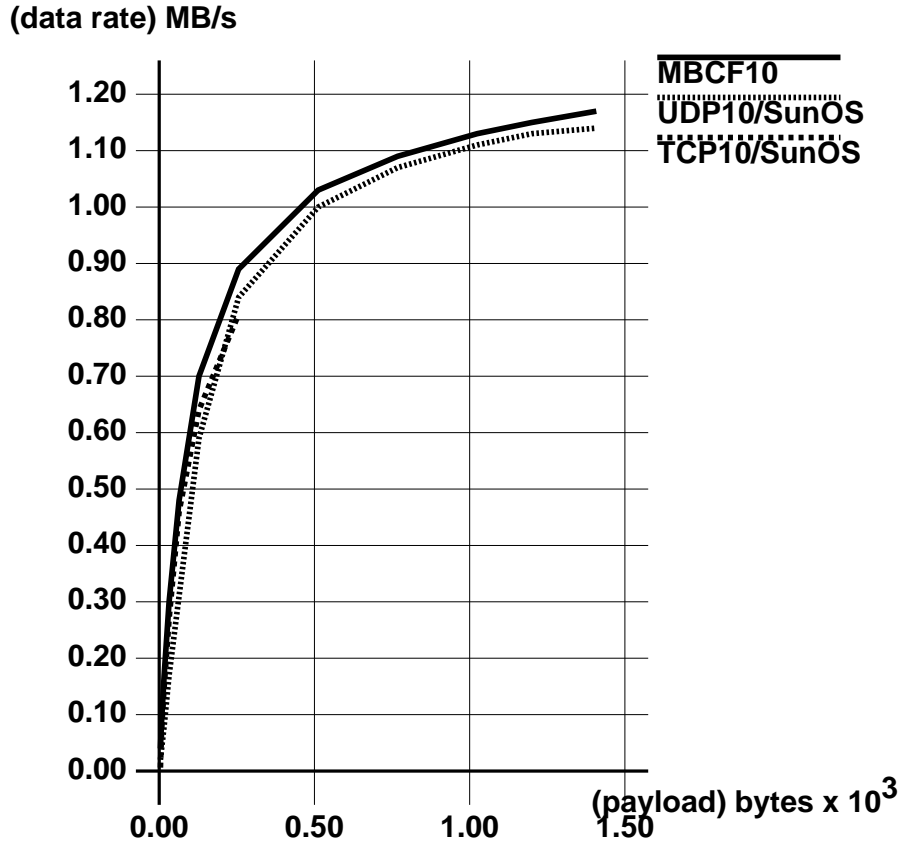


Figure 6.6: Peak data-transfer rates for MBCF10, TCP10, and UDP10

6.2.4 Round-trip latency of the MBCF/Ether protocol

I show the round-trip latencies of MBCF/100BASE-TX in Table 6.3. The latencies in the tables are measured by referring to the clock-counter LSI for three commands: MBCF_WRITE, MBCF_READ and MBCF_SWAP. All three commands are accompanied by their status-report options. I read the start time Just before requesting the command, and I check the end time just after recognizing the return of the status reply by polling. Only the reply-packet for the MBCF_SWAP command is loaded with the same amount of data as the requesting-packet. The figures in the table include the overheads incurred in referring to the clock-timer.

For simplicity of analysis, I changed the format of the results of these measurements into the graph shown in Figure 6.7. The x-axis represents the amount of data in one MBCF/100BASE-TX packet, and the y-axis is the latency. The curves for MBCF_WRITE and MBCF_READ lie on top of one another in the figure. Every

Table 6.3: Round-trip latency of MBCF/100BASE-TX

amount of data (bytes) command	4	16	64	256	1024
MBCF_WRITE_STAT (μs)	51	54	60.5	88	200
MBCF_READ_STAT (μs)	51	54.5	61	88	200.5
MBCF_SWAP_STAT (μs)	51.5	58	71.5	125.5	351

curve for latency has a good degree of linearity, and this fact tells us that there are few or no factors in the MBCF/100BASE-TX to make the measurements fluctuate.

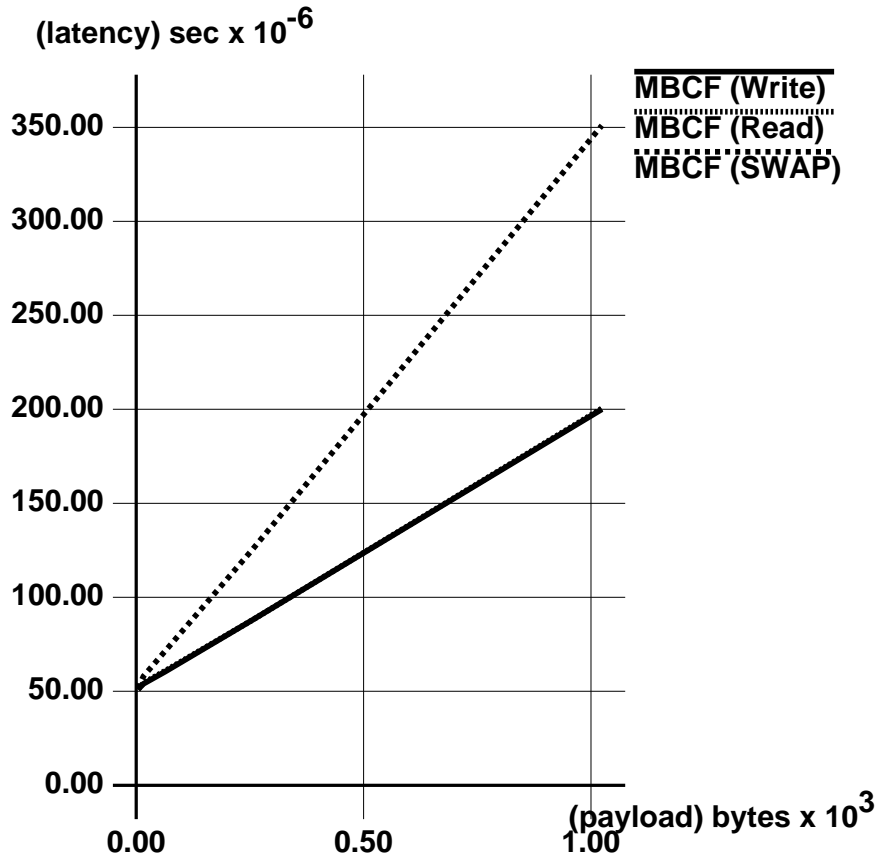


Figure 6.7: Round-trip latency of MBCF/100BASE-TX

In order to compare the latencies of MBCF/100BASE-TX and of typical communication methods under a conventional OS, I used TCP100/SunOS and UDP100/SunOS as defined in the previous subsection to obtain the curves for comparison given in Figure 6.2.4. In this comparison, I only considered MBCF_WRITE. This

was to avoid making the figure too busy. The notation in the figure is the same as in Figure 6.7. The latencies

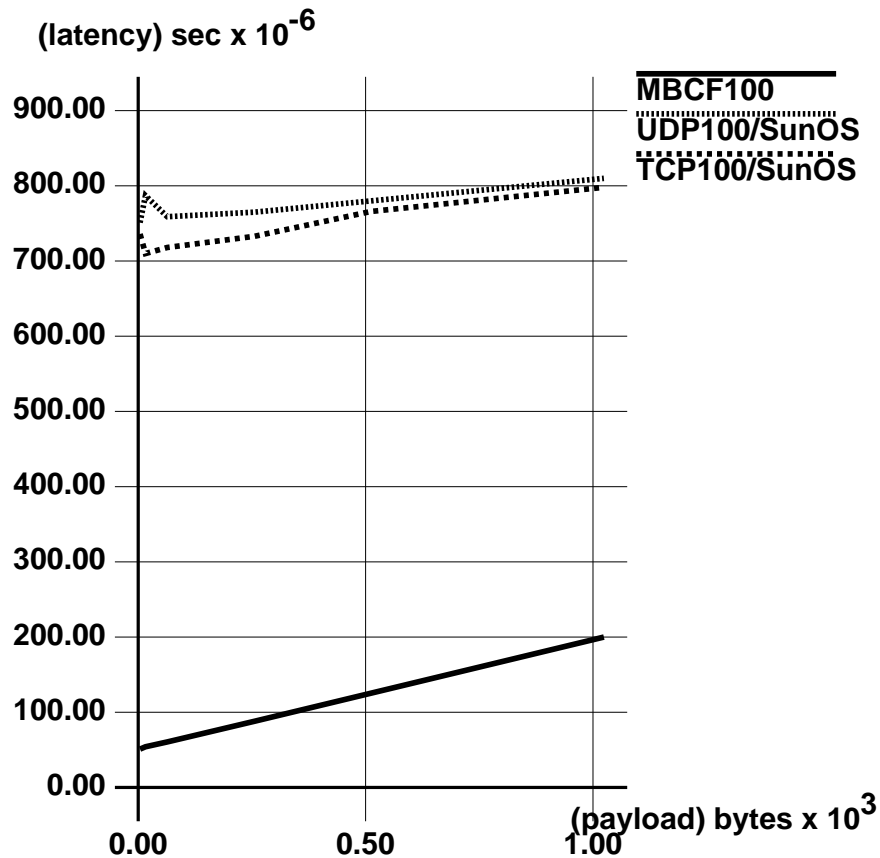


Figure 6.8: Round-trip latencies for MBCF100, TCP100, and UDP100

for both TCP100/SunOS and UDP100/SunOS are greater than $700\mu\text{sec}$ and are more than 10 times those for MBCF/10BASE-TX when the amount of data are below 64 bytes. In the curves for TCP100/SunOS and UDP100/SunOS I can hardly detect linearity according to the amount of data, because, in both cases, it is hidden in the fluctuations of the measurements according to non-deterministic factors arising from the conventional OS.

For your information, the latencies of the three MBCF commands on a 10BASE-T connection are also shown in Figure 6.9. The latencies for MBCF/10BASE-T, TCP10/SunOS, and UDP10/SunOS are also shown for comparison in Figure 6.10.

Next, I used a logic analyzer to acquire accurate values of the round-trip latencies. In this measurement I used the MBCF_WRITE_STAT with a fixed 4 bytes of data, because an effect of the different amount of

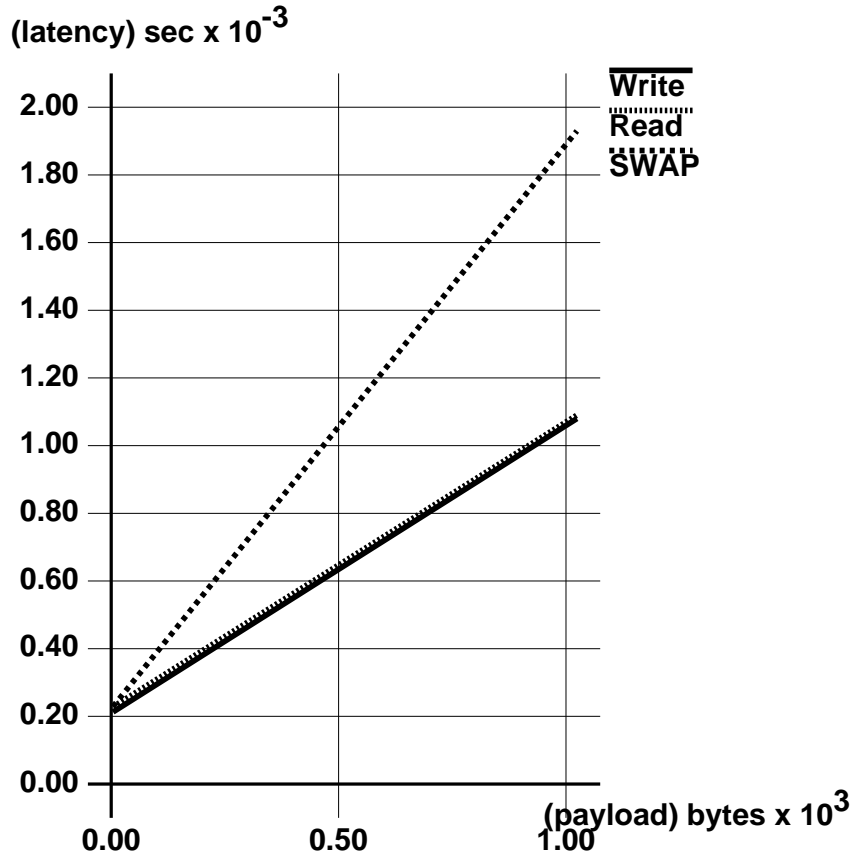


Figure 6.9: Round-trip latency of MBCF/10BASE-T

data was found in the previous measurement. Figure 6.11 shows a screen of results from the logic analyzer. The notations on the screen is the same as in Figure 6.3. The period between the two dotted-vertical-line cursors on the screen corresponds to the round-trip latency. The results fluctuate from $48.4\mu\text{sec}$ to $52.0\mu\text{sec}$ according to the cache conditions. The value measured most frequently was about $49.0\mu\text{sec}$.

6.2.5 One-way latencies of high-level commands of the MBCF/Ether protocol

I give the one-way latencies of high-level functions of the MBCF/100BASE-TX in Table 6.4. I used three commands in the measurements of latency in the tables: MBCF.WRITE (for reference), MBCF_FIFO and MBCF_SIGNAL. I started by measuring the round-trip latencies then used the results to calculate the one-way latencies. MBCF_FIFO latencies include the cost of one light-weight system-call for reading fifo data. MBCF_SIGNAL latencies include the cost of one invocation of the user-level subroutine and one light-weight

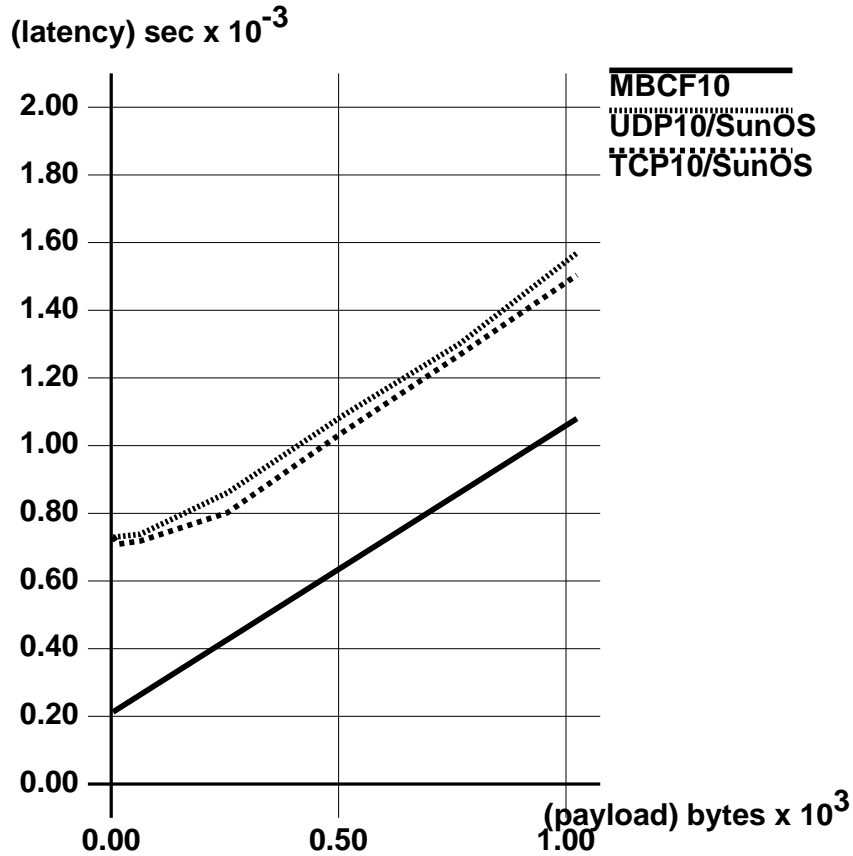


Figure 6.10: Round-trip latencies for MBCF10, TCP10, and UDP10

Table 6.4: One-way latencies of high-level commands of MBCF/100BASE-TX

amount of data (bytes) command	4	16	64	256	1024
MBCF_WRITE (μ s)	24.5	27.5	34	60.5	172
MBCF_FIFO (μ s)	32	32	40.5	73	210.5
MBCF_SIGNAL (μ s)	49	52.5	60.5	93	227.5

system-call for reading the fifo data.

6.2.6 Evaluation of performance in executing an application program

In this subsection I give some results of an evaluation of MBCF/Ether on a sample application: a parallel ray-tracing program. In the execution of this program N nodes calculate pixel values then use MBCF.WRITE commands to send them to a displaying node.

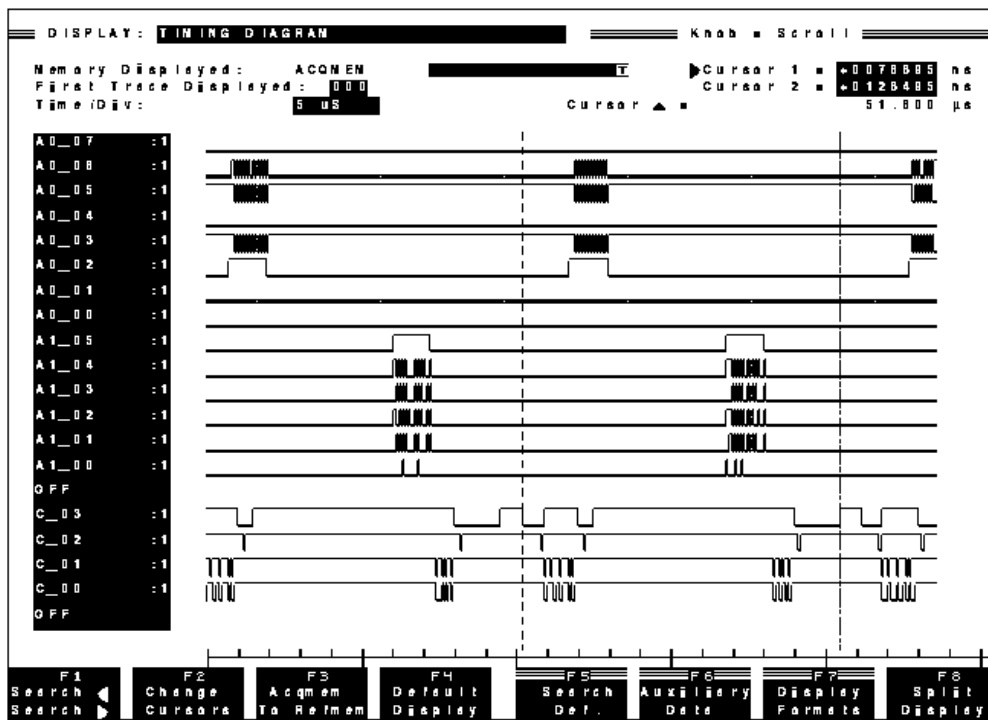


Figure 6.11: Signal wave forms of a single 4byte MBCF_WRITE

The ray-tracing program has 576 x 450 iterations in its main calculation loop. Each calculator node tries to execute all of the iterations, but checks whether or not it is responsible at the beginning of each iteration, and skips the iterations for which it is not responsible. The picture calculated in this evaluation procedure is shown in Figure 6.12.

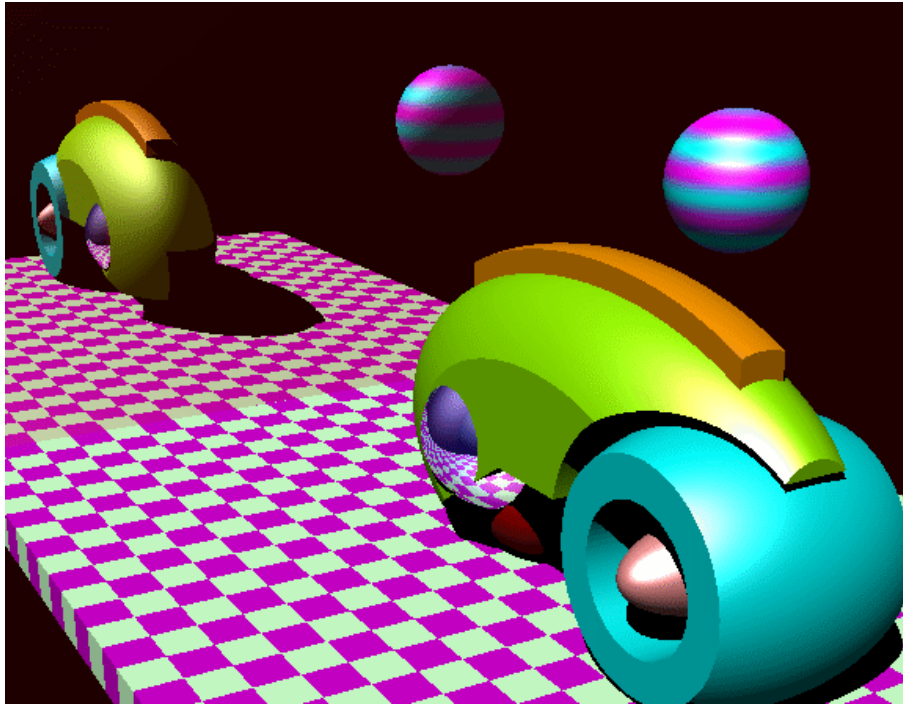


Figure 6.12: Sample picture used for measurements

The detailed specifications of the ray-tracing program and its execution are as follows.

- 576 x 450 pixels, 8bit dithered color (from 24bit RGB)
- One node is exclusively used to display results of the calculation, and the other nodes use `MBCF_WRITE` to directly write data to the frame-buffer of the displaying node.
- The distributions of load to calculator nodes is static and cyclic, with load assigned every N pixels.
- The unit for time measurement is 10msec.
- For calculations alone, the average costs per pixel in this evaluation is $70\mu\text{sec}/\text{pixel}$.

- By changing parameters (the number of processors and/or the unit of load distribution) the parallel characteristics of the ray-tracing program may be altered from computation intensive to communication intensive.

I show the execution time of the program using MBCF/100BASE-TX in Table 6.5.

Table 6.5: Execution times of parallel ray-tracing using MBCF/100BASE-TX

payload size (byte)	64	32	16	8	4	2	1
1node (sec)	16.70	16.74	16.84	17.00	17.37	18.10	19.56
2node (sec)	8.55	8.63	8.67	8.72	8.90	9.26	9.98
3node (sec)	6.32	5.97	5.98	5.94	6.07	6.32	6.81
4node (sec)	4.49	4.53	4.55	4.60	4.69	4.85	5.25
5node (sec)	3.68	3.69	3.70	3.73	3.81	3.96	4.47
6node (sec)	3.36	3.38	3.25	3.22	3.25	3.37	4.18
7node (sec)	2.74	2.76	2.77	2.80	2.85	2.95	4.12

In Table 6.5, can attain a speed-up with an increase in node numbers (i.e., processor numbers) is obtained even with very fine-grained communications (payload-size: 1, 2, or 4), since the overhead of the MBCF/100BASE-TX is low and its performance is fast enough for several processors to efficiently run the parallel ray-tracing program. Since the minimum packet-size of the ethernet is 64 bytes and the amount of data for ethernet-level padding and/or synchronization is 12 bytes per a packet, a total of 76 bytes of data is transmitted through the ethernet a minimally fine-grain of communications. At the entry (7node, 1byte) in Table 6.5, the transmission rate is about 4.78Mbyte/sec, and the performance in handling packets at the displaying node reaches 63,000packet/sec.

In order to compare the MBCF with typical communication mechanisms, I again used the UDP100/SunOS as was described earlier. I used the same hardware environment as for the UDP100/SunOS experiments. The X11R6 window system was used at the displaying node without “XFlush” so as not to impose additional overheads. The same compiler (gcc) and the same optimizing option (O4) were applied to both the MBCF/100BASE-TX and UDP100/SunOS cases. Table 6.6 shows the results for the UDP100/SunOS.

Table 6.6: Execution times of parallel ray-tracing using UDP100/SunOS

payload size (byte)	64	32	16	8	4	2	1
1node (sec)	18.19	20.69	22.12	27.36	38.68	59.31	99.45
2node (sec)	10.78	12.04	14.16	19.07	28.33	48.13	77.10
3node (sec)	11.81	11.85	13.88	18.30	26.54	46.26	80.41
4node (sec)	10.81	11.45	13.60	18.37	27.31	45.79	75.96

For small payload-sizes and with more than two processors (nodes) the performances for UDP100/SunOS

are much worse than for MBCF/10BASE-T. The overheads of UDP100/SunOS are too great to attain a speed-up when there are more than two processors (nodes).

I also list the execution times of the program using MBCF/10BASE-T in Table 6.7 and that using UDP10/SunOS in Table 6.8, for reference.

Table 6.7: Execution times of parallel ray-tracing using MBCF/10BASE-T

payload size (byte)	64	32	16	8	4	2	1
1node (sec)	16.90	16.97	17.12	17.39	17.96	19.26	24.63
2node (sec)	8.67	8.75	8.83	8.94	9.26	12.96	*23.43
3node (sec)	6.41	6.07	6.07	6.10	7.85	*12.93	*23.16
4node (sec)	4.55	4.60	4.65	4.83	*7.48	*12.95	*23.54
5node (sec)	3.73	3.74	3.77	*4.28	*7.34	*12.86	*23.46
6node (sec)	3.43	3.44	3.32	*4.26	*7.27	*13.04	*23.59
7node (sec)	2.79	2.80	2.88	*4.16	*7.44	*13.06	*23.65

Table 6.8: Execution times of parallel ray-tracing using UDP10/SunOS

payload size (byte)	64	32	16	8	4	2	1
1node (sec)	18.04	19.25	21.50	26.94	36.91	56.74	96.33
2node (sec)	10.34	11.61	13.35	19.13	28.11	45.67	76.14
3node (sec)	10.90	11.61	13.15	19.18	28.42	47.39	78.79
4node (sec)	9.80	11.30	14.44	19.04	30.62	46.08	75.18

The asterisks in Table 6.7 indicate that the corresponding measurement of execution time tended to fluctuate because of contention between ethernet packets, and the measured values actually varied within ± 0.3 sec. The figure written in these entries is an average of measurements over three trials. As explained earlier in this subsection, 76 bytes of data is the finest grain transmitted through the ethernet. At the entry (6node, 4byte) in Table 6.7, The transmission rate is about 677Kbyte/sec, and this value means that the medium has almost reached saturation for a CSMA/CD algorithm. Similarly, the other entries with asterisks reflect saturation of the 10BASE-T ethernet.

Measurements for the UDP10/SunOS are presented for your information. At small payload-sizes and with more than two processors (nodes) the performance of UDP10/SunOS is much worse than MBCF/10BASE-T but is a little better than UDP100/SunOS. This is because UDP/IP suffers from a lower overhead on the 10BASE-T than on the 100BASE-TX. The 100BASE-TX is an additional equipment to the SPARCstation 20s which are the test-beds of this evaluation, but the 10BASE-T is a basic and primary facility and is implemented on their mother-boards.

6.3 Evaluation using ULTRA 60s and SSS-CORE Ver.2.x

The SSS-CORE [36, 53, 55] is my original scalable operating system for workstation clusters and the SSS-CORE Ver.2.x is the name of a series of editions of this OS for Sun Microsystems' ULTRA workstations, which are powered by UltraSPARC processors [70].

6.3.1 Environment used for performance evaluation

I used the following workstation cluster to implement and evaluate the MBCF/Ether protocol.

- Node of the network of workstations (NOW)
 - Ultra 60 (450MHz UltraSPARC x 1, 64bit 66MHz PCI bus)
 - Sun GigabitEthernet/P 2.0 Adapter
- Network
 - direct connection between two 1000BASE-SX cards
- Operating system of the NOW
 - SSS-CORE Ver.2.2
 - * general purpose (multitasking with parallel task execution)
 - * scalable
 - * parallel and distributed processing
 - * high degree of optimizability for user-code
 - * test bed for the MBCF and the compiler-assisted DSM system
 - * task-migration facility
 - * supports the UltraSPARC processor (SPARC V9[81] architecture)
 - * development from scratch

6.3.2 Peak data-transfer rate of MBCF/Ether communications

Table 6.9 shows peak data-transfer rates of the MBCF/1000BASE-SX and the TCP/IP (Solaris2.6, 1000BASE-SX).

As for the MBCF measurements, I use MBCF_WRITE commands with various amount of data to measure the data rate. The figures in the table are net quantities which correspond to payload data alone and do not

Table 6.9: Peak data-transfer rates of MBCF/1000BASE-SX and TCP1000/Solaris

amount of data (bytes)	4	16	64	256	1024	1408
MBCF/1000BASE-SX	2.29	5.67	22.30	55.41	78.22	80.92
TCP/Solaris (1000BASE-SX)	0.09	0.43	1.67	5.56	12.79	20.21
MBCF/100BASE-TX	0.34	1.27	4.82	9.63	11.64	11.93

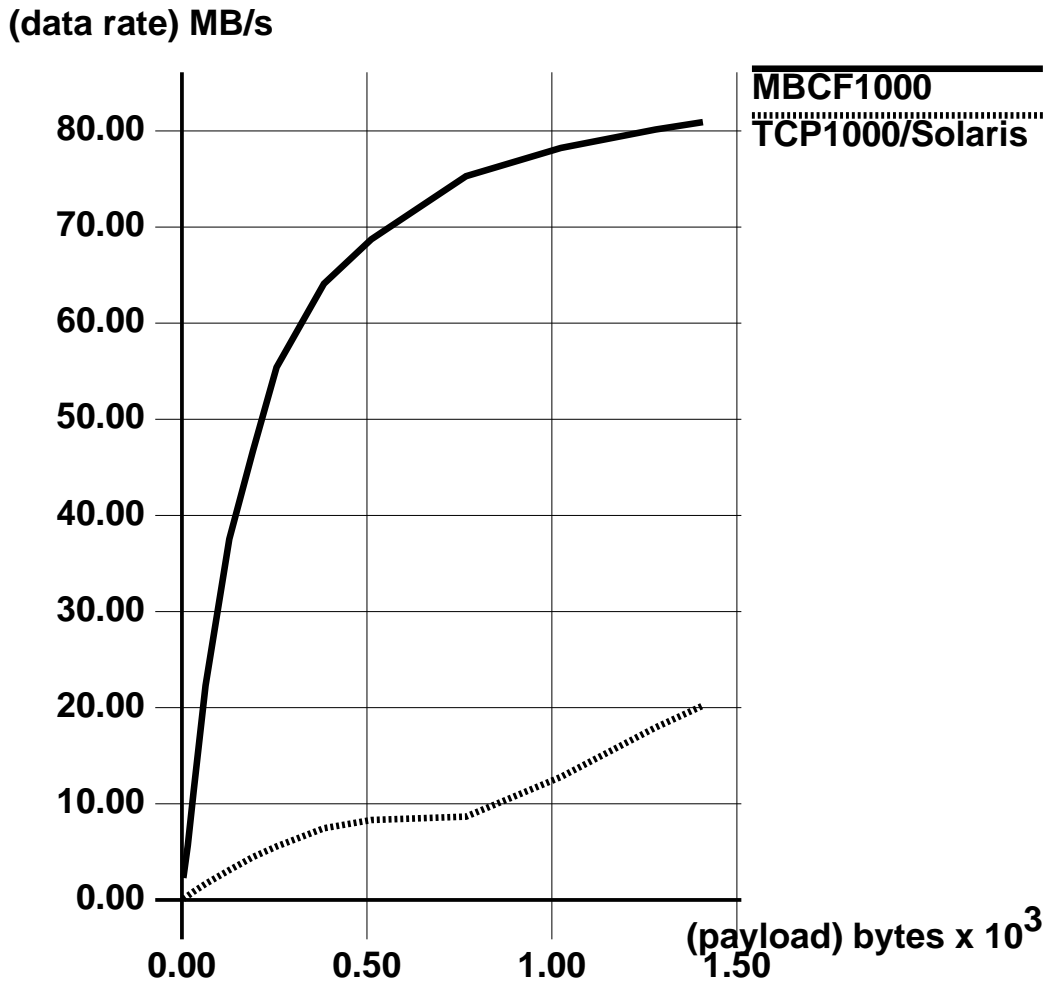


Figure 6.13: Peak data-transfer rates for MBCF1000 and TCP1000/Solaris using 1000BASE-SX

include the packet header of the MBCF/Ether or additional data for ethernet protocol. The method of measurement is for a requester to repeatedly send MBCF_WRITE commands to a fixed target and only checking acknowledgment every 16 transmissions. This checking every 16 transmissions is used to avoid saturation of the ethernet. The ideal value of the peak data-transfer rate is 125Mbyte/s for the 1000BASE-SX, and this ideal value corresponds to all transmitted data, including all headers and all additional signals for the protocols. Even if we take these overheads into account, the actual result, a maximum of 80.92Mbyte/sec as shown in Table 6.9, is rather low in comparison with the ideal maximum data rate. In the latency measurements on MBCF/1000BASE-SX I got the result that the software overhead for sending a transmission ranges from 1.6 μ sec with a 4byte payload to 3.2 μ sec with a 1408byte payload. Although I have no measured values of accurate software-overhead cost for receiving, we can estimate the cost by exploiting the results of measurement on the cluster of SPARCstation 20s. The value is about twice the sending cost and is smaller than the cost of transfer in the fiber cable that is the medium of 1000BASE-SX (about 11.2 μ sec with a 1408byte payload). Note that the costs of sending, transferring, and receiving all overlap with each other. Therefore, I conclude that software overhead of the MBCF/Ether is NOT the bottle neck of the peak data rates and that there must be some limitations imposed by the hardware of the ULTRA 60 workstations. This conclusion is also supported by the shape of the curve that corresponds to MBCF/1000BASE-SX in the peak data-transfer rate graph (Figure 6.13). The saturation in the shape means that there some limitation appears at the performance line of 80Mbyte/sec (12.5 μ sec in time). However, the software overhead of sending or receiving is much smaller than this 12.5 μ sec period.

As for the TCP/IP measurements given in the Table 6.9, I measured the peak transfer rates for the TCP/IP in the same hardware environment in order to compare MBCF/Ether with ordinary communication methods under a conventional OS (Solaris 2.6). I used the socket libraries of Solaris 2.6 for these measurements. For measurements of peak TCP/IP throughput (peak transfer rate) I did not attach the TCP_NODELAY option to the TCP/IP sockets and measured the rates with various amount of data which correspond to the parameters of the “send” function in the socket library. I refer to this TCP/IP communication on the 1000BASE-SX line as “TCP1000/Solaris”.

Figure 6.13 shows the peak rates for MBCF1000 (MBCF/1000BASE-SX) and TCP1000/Solaris. The x-axis of the figure represents the amount of data in one ethernet-packet, and the y-axis is the data-transfer rate. The TCP1000 only attains 20.21Mbyte/sec and this is due to the overheads involved in processing TCP system-calls. If we use a larger amount-of-data parameter for the send function than the maximum payload-size of an ethernet packet, we get better performance values than in the table and the figure. The maximum performance value was 53.06Mbyte/sec with a 64Kbyte amount-of-data parameter, but this is still

Table 6.10: One-way latency (μsec) of MBCF/1000BASE-SX

amount of data (bytes)	4	16	64	256	1024
MBCF 1000BASE-SX	9.6	11.0	11.5	16.2	35.9
Solaris TCP/IP(1000BASE-SX)	95.08	95.22	95.39	99.45	114.15
MBCF 100BASE-TX	24.5	27.5	34	60.5	172

much lower than that for the MBCF/1000BASE-SX. The initial slope for TCP1000/Solaris in Figure 6.13 is much easier than the initial slope for the MBCF1000. However, the curve for the MBCF1000 almost goes into saturation at payload-sizes above 1024 bytes, owing to the limitations of the hardware.

6.3.3 One-way latency of the MBCF/Ether protocol

I show the one-way latencies of MBCF/1000BASE-SX and TCP/IP (Solaris2.6, 1000BASE-SX) in Table 6.10.

For the MBCF measurements, I measured the MBCF_WRITE latencies in the tables by referring to the clock-counter in the UltraSPARC processor. I initially measured the round-trip latencies and then calculated the one-way latencies. I read the start time just before requesting the MBCF_WRITE command, and I checked the end time just after recognizing the return of the status reply by polling. The figures in the table include the overhead of referring the clock-timer, but the values of these overheads are negligible in the case of the measurements with ULTRA 60 workstations.

I measured latencies for TCP1000/Solaris as described in the previous subsection to obtain, for comparison, the TCP/IP figures in Table 6.10. I used the Solaris 2.6 socket libraries. To cope with latency measurements of fine-grained communications, I added the TCP_NODELAY option to the TCP/IP sockets. For simplicity of analysis I present the same results of the measurements in graphical form as well (Figure 6.14). The x-axis represents the amount of data in one MBCF/1000BASE-SX packet, and the y-axis is the latency. The other notation in the figure is the same as in Figure 6.13. Latencies for TCP1000/Solaris are about $100\mu\text{sec}$ and are thus about 10 times those for the MBCF/1000BASE-SX when payload size is below 64 bytes. In the curve for TCP1000/Solaris we can barely determine linearity according to payload size. This is because it is hidden in the fluctuations that appear in the measurements because of non-deterministic factors in the conventional OS.

6.3.4 One-way latencies of high-level commands for MBCF/Ether protocol

I show the one-way latencies of high-level functions for MBCF/1000BASE-TX in Table 6.11. I used three commands: MBCF_WRITE (for reference), MBCF_FIFO, and MBCF_SIGNAL to measure the latencies in

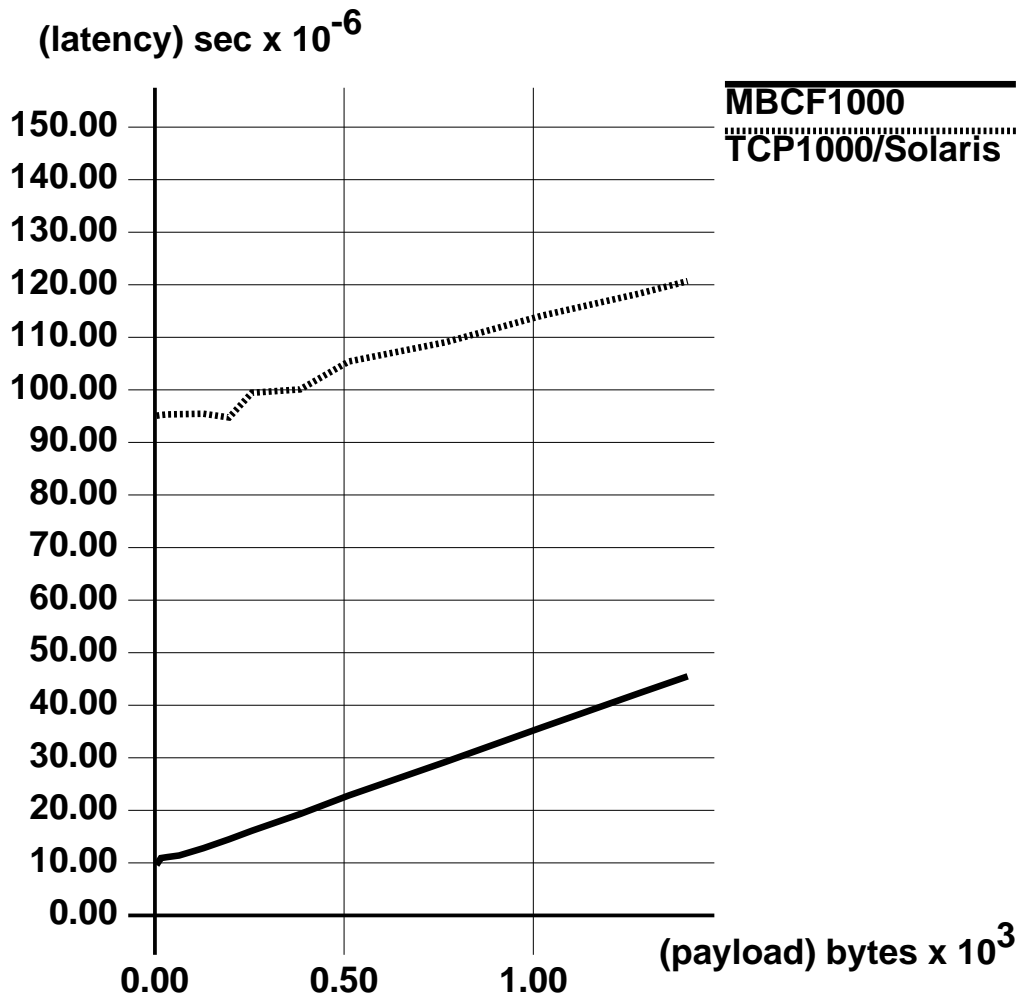


Figure 6.14: One-way latencies for MBCF/1000BASE-SX and TCP1000/Solaris

the tables. I initially measured the round-trip latencies then calculated one-way latencies from those results. MBCF_FIFO latencies include the cost of one light-weight system-call to read the fifo data. MBCF_SIGNAL latencies include cost of one invocation of a user-level subroutine and of one light-weight system-call to read the fifo data.

Table 6.11: One-way latencies (μ sec) of high-level commands of MBCF/1000BASE-SX

amount of data (bytes) command	4	16	64	256	1024
MBCF_WRITE	9.6	11.0	11.5	16.2	35.9
MBCF_FIFO	14.7	15.3	16.4	23.5	54.8
MBCF_SIGNAL	22.5	24.0	24.9	31.4	57.8

6.4 A quantitative comparison with the user-level communication mechanisms of MPPs

In this section I quantitatively compare the MBCF with several user-level communication mechanisms of commercial massively parallel processors (MPPs). The mechanisms of the MPPs all have limitations to a greater or a less degree. For examples, some mechanisms force an application to use these mechanisms exclusively, or other mechanisms force the OS to allocate tasks only in a gang-scheduling manner. Except for the Active-Message-type mechanisms, all of the mechanisms are message-passing style, and the only variety of function is message-sending. Therefore, the MBCF is qualitatively much better protected, virtualized, and varied than the mechanisms of the MPPs, and is thus superior.

Table 6.12 shows quantitative comparisons of peak data-transfer rates and round-trip latencies. All machines in the table except for the workstation clusters (SSAM[78] and MBCF) have system-specific high-speed communication hardware which is much faster than the 100BASE-TX system. The SSAM system uses a 156Mbps ATM NIC. This, too, is faster than the NIC of the MBCF/100BASE-TX.

All figures in the table except those in the MBCF row are quoted from the following papers. Figures without any marks are from von Eicken et al. [78], and figures on SP-1/SP-2 (with ‡ mark) are from Snir et al. [66].

The implementations of the SSAM in the table did not have any mechanisms to guarantee arrival and/or an fifo property for packets. The practical SSAM would suffer a larger overheads than the SSAM in the table. The SP-2 has two entries in the table: “MPL/p” is a method in which an application exclusively uses the SP-2’s high-speed communication hardware, while “MPL/udp” is a method in which the communication library uses the UDP interface of the SP-2’s OS. The former is not worth calling a “virtualized” interface,

while the latter is comparable with the MBCF.

Table 6.12: Basic performance of user-level communication mechanisms

Machine + software	Peak rate (Mbytes/s)	Round-trip latency(μ s)
SP-1 + MPL/p	8.3 / 8.7 [‡]	56 / 75 [‡]
Paragon + NX	7.3	44
CM-5 + Active Message	10.0	12
SP-2 + MPL/udp	10.8 [‡]	554.0 [‡]
SP-2 + MPL/p	35.5 [‡]	78.0 [‡]
SS-20 cluster + SSAM@!J@!(JATM@!K@!(J	7.5	52
SS-20 cluster + MBCF/100BASE-TX	11.9	49
ULTRA 60 cluster + MBCF/1000BASE-SX	80.9	19.2

Considering these three points:

- the MBCF has the highest level of protection and virtualization;
- MBCF/Ether only requires commodity ethernet NICs and requires no dedicated hardware mechanisms;
and
- as for the performance of the raw communication hardware, the MBCF/100BASE-TX is the lowest of all;

the figures in the table 6.12 tell us that the methodology and implementation techniques of the MBCF are excellent and remarkable.

Chapter 7

Compiler-Assisted Distributed-Shared Memory Schemes

7.1 Software DSM for the medium-grained MBCS

As I mentioned in Section 1.3, it is one of strong points on the Memory-Based Communications and Synchronization (MBCS) scheme that efficient remote-cache mechanisms are suitable for MBCS systems. In the MBP system, which is a fine-grained hardware implementation of the MBCS, a hardware DSM facility is carried from the beginning of its architecture design. The DSM subsystem of the MBP system realizes very wide cache-memory space in each node by exploiting main memory as caching space. While, in the MBCF system, which is a medium-grained software implementation, conventional hardware and/or software DSM approaches is not suitable owing to its rather large grain-size and per-packet overhead. Conventional DSM approaches causes a lot of fine-grained communications or a lot of fine-grained page-traps to maintain memory coherence. Any implementation of the MBCS scheme can efficiently operate remote-memory including cache-area in a remote node. Therefore, in the MBCF system some new software DSM approach where grain-size of communications and/or synchronization can be merged into the larger size is strongly required. I have invented such a new DSM approach in 1996 and will describe it in this chapter.

7.2 Compiler-based approach

It is difficult to efficiently execute shared-memory-based parallel programs (e.g., SPLASH-2 [82]) on distributed-memory parallel systems without hardware-based remote-caching mechanisms. We typically need to modify and rewrite such programs to make them fit for the distributed-memory system. One way to avoid this rewriting is to utilize software-based distributed-shared memory (DSM) mechanisms that are

supported by the operating system (OS). We can then obtain the benefits of caching from the conventional software DSM, but the cache-miss penalty and the overhead for maintenance of coherence are still large. The penalty and overhead mean that we cannot obtain good results through an approach which uses a conventional software DSM.

In this chapter I introduce a brand-new approach to solving this problem. My approach is a combination of user-level cache emulation and an optimizing compiler. My goal is to realize the efficient execution of shared-memory-based parallel programs with automatic optimization for remote communications under a general-purpose operating system on a stock network of workstations. If we use any conventional OS-based (page-trap-based) software DSM, the procedures for cache maintenance are hidden from the user level and it is thus difficult to apply optimizing compilers to codes for the procedures together with the user-level application code. On the other hand, applying user-level codes to cache maintenance increases the opportunities for code optimization. Therefore, I adopted user-level cache emulation (i.e., user-level software DSMs).

We have developed an optimizing compiler (**RCOP**: the Remote Communication Optimizer) [57, 58, 59] that applies this approach and includes run-time libraries for user-level cache emulation. The RCOP analyzes memory access by a target application program, finds requests for remote memory access, inserts code for the checking and maintenance of caches of remote data, removes redundant cache maintenance codes, and reduces the number of communications by merging packets.

I have already introduced a novel protected and virtualized high-speed user-level communication and synchronization scheme, the “Memory-Based Communication Facility (MBCF)”, for use with a general-purpose distributed-memory system with off-the-shelf communication hardware. The MBCF is suited to my approach to the efficient execution of shared-memory-based parallel programs. For programmers and compilers, the MBCF provides light-weight methods for direct access to remote memory in user-task spaces.

7.3 My approach

My basic strategies for the efficient execution of shared-memory-based parallel programs are as follows.

1. automatic translation/compilation from source programs
2. shared-memory-based data allocation
3. inserting user-level cache-emulation code
4. using code that explicitly contains communication procedures for remote access
5. optimizing remote communications

- reducing the amount of cache-maintenance code
- reducing the number of communications
- tuning payload sizes of packets for communication

Our automatic optimizer exploits information in the source programs. For elegant handling of memory pointers to the shared objects, I assume a shared-memory-based data allocation. This supposes that the local virtual addresses of a shared data object are identical in every node of the system, so that the system and applications can easily identify shared data objects. However, I assume that the system has neither hardware for remote-memory access nor hardware DSM. Therefore, I must use code which invokes explicit operations to perform remote-memory-access. If I use an executable object in which every remote access is replaced with code for an explicit remote communication, there are too many fine-grains in the communications that result, and the very large overhead thus accumulated degrades the speed of execution. I thus need to reduce the number of grains by communications as much as possible. The first step in this reduction is to introduce a kind of cache system to our execution environment. I adopt the user-level (application-level) implementation of software DSM, since user-level code for cache maintenance increases the opportunities for code optimization. I need to insert explicit cache-emulation code at the candidate points, i.e. the points where remote access occurs. If our analyzer cannot determine whether the memory access at a candidate point is remote or local, our optimizer inserts code for address-range checking before the cache-emulation code in order to determine whether there is a necessity for a remote access. Cache-emulation code then explicitly performs remote communications if this is necessary.

The simple replacement of remote-accesses with cache-emulation codes will still be redundant, and leaves room for the optimization of remote communications. I will describe our optimizing techniques in a later section.

I assume that commodities (e.g., ethernet links) are used as the communication hardware of the system. Such mechanisms carry a non-negligible overhead of communication and are hence poor at fine-grained communications. Since the size of the sending buffer in the communication hardware is a limiting factor, it is possible to efficiently send amounts of data below that limit at a time. While keeping the meaning of a parallel program, it is advantageous in terms of performance to statically or dynamically adjust the amount of data in packets for communication.

7.4 UDSM and ADSM

In terms of the optimization of code for inter-node communications, the full user-level cache scheme (user-level DSM:UDSM), in which application programs only use user-level codes to maintain software-based remote-caches, is better than an OS-based software DSM. In other words, the UDSM scheme is more suited to the exploitation of opportunities for the optimization of communications and execution than OS-based DSMs.

In the case of a UDSM system, however, the user-level executable code must explicitly maintain, check and modify software-controlled-cache tags. It is a little difficult to develop optimizing compilers that are sophisticated enough to hide and/or reduce the overhead involved in handling cache tags. Inter-node communications are only required in shared-write or cache-miss situations. If we use a large area of memory as a software-controlled cache we can maintain a low rate of cache misses. Also, in typical applications there are many more shared-read than shared-write operations. These characteristics suggested a brand-new remote-caching scheme “asymmetric distributed-shared memory (ADSM)” [51, 48].

In conventional page-based (i.e., OS-based) DSMs, shared-writes as well as read-cache-misses are supported by the TLB/MMU mechanisms of node processors, with the use of write-protection traps and page-fault traps. Though the ADSM is a page-based caching scheme, only read-cache misses are supported by the TLB/MMU mechanisms. For each shared-write in the ADSM scheme, a sequence of instructions which maintains the consistency of caches in the system is inserted into the user-level executable code by the optimizing compiler. Sequences in the user-level code include explicit communication procedures and invalidation (or updating) of remote caches while modifying the states of local caches.

There is less opportunity for the optimization of communications and execution in the ADSM than the UDSM. This is because the code for shared-reads are implicitly supported by the OS and the cache-block size of the ADSM should be equal to the size of a memory-page. In the ADSM, however, there is still a lot of room for a variety of optimizations using inserted cache maintenance codes. The strategies for handling shared-reads (read-cache misses) and shared-writes are different. Therefore I call this scheme the “asymmetric” DSM.

Which should we use, the UDSM or the ADSM? The answer depends on system parameters: the optimization level of the compiler, memory-access pattern of application programs, software cost of checking cache tags, and cost of page-fault trapping.

7.5 Needs of the MBCF for UDSM and ADSM

Conventional user-level communication interfaces (e.g., the socket library or MPI) of existing operating systems are useless for realizing my goal because they carry large overheads. Therefore, I adopted the MBCF for use with the UDSM and/or the ADSM. Factors of two types produce the major part of overheads carried by conventional interfaces. The first type are methodological factors (including degrees of functionality, protocols, and packet formats). Conventional communication interfaces are of message-passing type, and their functions are limited to remote-write operations into a few specific message-buffer addresses in the kernel-space. To break out of this limitation, I have adopted a memory-based form of operation in which arbitrary target addresses and a wide variety of functions can be used. Moreover, the provision of direct remote-memory-access capability with medium-grained or coarse-grained transmissions by the MBCF is suitable for the optimization of communications for the UDSM and the ADSM. Middle-sized or large amounts of data can be handled in one MBCF operation and multiple MBCF operations can be merged into a single communication packet by the optimization of communications. The other set of factors that influence the overhead are software-engineering aspects (implementation methodology). In conventional OSs, communications and synchronization among nodes (machines) are regarded as typical I/O events handled in much the same way as, for example, disk operations. Device-drivers for communications and synchronization thus have the same data and control structures as device-drivers for other I/O devices. Consequently, the device-drivers suffer from large overheads that are not necessary for the functions of communication and synchronization. To realize a high-performance implementation, the MBCF-dedicated system-calls and the MBCF-dedicated interrupt routine were developed and used. These ensure that no operation is irrelevant to the functions of the MBCF.

7.6 Optimization for the UDSM and the ADSM

In this section I list the code-optimization techniques which are suitable for the UDSM and ADSM schemes. Firstly I explain the common techniques used in shared-writes operations on UDSM and/or ADSM.

- When shared-writes to contiguous locations take place and there is no synchronization point among them, the corresponding consistency-management code can be coalesced [57, 56]. This **coalescing optimization** (for write operations) reduces the runtime overhead of the maintenance of local consistency and the number of communications.
- When there are many fine-grained packets with the same task of the same node as destinations, they can be combined into a large packet in compilation and/or at runtime. This **combining optimization**

reduces the number of packets communicated.

- When multiple shared-writes by a node modify the same location between two contiguous synchronization points, all except the last one can be ignored. This **last-write optimization** reduces the runtime overhead, the number of packets communicated, and the amount of data transferred.
- By changing the instructions for consistency maintenance, the compiler can specify various consistency protocols according to the characteristics of the target shared variables of an application program. This **protocol-switching optimization** reduces the number of packets communicated and the amount of data transferred.

Next, I will describe the techniques that apply specifically to shared-read operations in UDSM.

- When shared-reads to the contiguous locations take place and there is no synchronization point among them, the corresponding cache-hit checking code can be coalesced [57, 56]. This **coalescing optimization** (for read operations) reduces the runtime overhead of the maintenances of local consistency and the number of packets communicated.
- When multiple shared-reads from a node fetch the same location between two contiguous synchronization points, all except the first can be ignored. This **first-read optimization** reduces the runtime overhead, the number of packets communicated, and the amount of data transferred.
- By changing the size of the consistency maintenance block, application programmers or the compiler can adapt the block size according to the characteristics of the target shared variables of an application program, in order to avoid worsened performance because of false-sharing. This **block-size optimization** reduces the number of packets communicated and the amount of data transferred.

7.7 Evaluation of my scheme and methods of optimizations

My colleagues (Dr. Niwa and Dr. Inagaki) have researched and implemented a prototype of the optimizing compiler [57] and the runtime system [57] for the UDSM and the ADSM on the *SSS-CORE* with MBCF/Ether. In this section I introduce the results of their evaluations of the effects of the various types of optimizations for the UDSM/ADSM.

7.7.1 Evaluation environment

The evaluation environment is as follows.

- Application programs

We used nine programs from the SPLASH-2 program suite [82] to evaluate my approach and our optimization methods. The programs are explicitly written as parallel programs and were intended as application programs for hardware DSMs. The nine programs are LU decomposition (LU-Contig), Radix, FFT, Barnes, Raytrace, Water-Nsquared (Water-NS), Water-Spatial (Water-SP), Ocean (Ocean-RW), and Volume rendering (Volrend). 5 of these programs were slightly modified to make them execute more efficiently, and these modifications are usually made when the programs are executed on software DSM systems [11, 23, 25]. The actual modifications are shown in Table 7.1.

Table 7.1: Modifications on SPLASH-2 programs

Program	Modification
LU-Contig	owner of block $(i, j) \leftrightarrow$ owner of block (j, i)
FFT	sender-initiated Transposition [23]
Raytrace	elimination of unused lock-operation for ray ID [23]
Ocean	rowwise partition (Ocean-RW) [25]
Barnes	sequential tree-construction [11]

- Compiler

We used our optimizing compiler RCOP [37, 59], in which optimizing techniques for the UDSM/ADSM are implemented. The RCOP analyzes the source code of shared-memory parallel programs which are based on the LRC (lazy release consistency) model, solves dataflow equations for redundancy eliminations, then generates efficient C-language programs which explicitly include code for managing cache coherence in the DSM. We used gcc-2.7.2 as the backend compiler.

- Node machine

We used a cluster of 16 machines of Axil 320 model 8.1.1 workstations. These machines are compatible with Sun Microsystems' SPARCstation 20 (85MHz SuperSPARC-II x 1). The Fast Ethernet SBus Adapter 2.0 from Sun Microsystems was installed on all of the machines.

- Network

We used a switching hub (3Com's SuperStackII 3900) for a fast ethernet connection (100BASE-TX).

- Operating system

We used Ver. 1.2 of the *SSS-CORE* general-purpose scalable operating system. This is my original

cluster OS and is able to handle the MBCF/Ether protocol. MBCF/Ether under *SSS-CORE* using 100BASE-TX attains a peak data-transfer rate of 11.93Mbyte/sec and round-trip latency of 49 μ sec.

- Runtime system

The RCOP inserted explicit communication code into parallel programs. The code is based on the SAURC protocol [52] for updating of home pages. Fine-grained packets for communications are combined into larger packets and are then transferred to the home. For each cache block, our cache-maintenance code does not keep information on the identity of the processor which updated the cache-block, but only on the existence of modifications to the block. Remote requests such as page-invalidations are detected by MBCF.SIGNALs which invoke a specified user-level handler to process the requests. In the ADSM scheme cache-blocks and pages are of equal size, at 4K bytes, while in the UDSM we use 1Kbyte blocks, with reference to the results of the earlier research [59].

7.7.2 Performance in terms of single-node execution

Table 7.2: Problem scale and single-node execution times (exec. time: sec, (additional overhead:%)).

Program	Problem scale	Sequential exec.	ADSM parallel-single exec.		UDSM parallel-single exec.	
			FOW	w/o FOW	FOW	w/o FOW
LU-Contig	2048 ² doubles	435.62	436.34(0.16)	436.34(0.16)	457.73(5.1)	439.55(0.99)
Radix	4M integer keys	6.42	6.44(0.30)	7.80(21)	8.55 (33)	6.75(5.1)
FFT	1M complex doubles	18.27	18.29(0.11)	22.03(20)	18.49(1.2)	18.48(1.2)
Barnes	2 ¹⁵ bodies	66.63	67.14(0.77)	67.86(1.84)	75.10(13)	74.83(12)
Raytrace	balls4, 128 ² pixels	171.41	171.44(0.017)	174.44(1.76)	194.82(14)	200.0(17)
Water-NS	4096 molecules	464.11	471.84(1.7)	472.59(1.82)	477.74(2.9)	479.36(3.2)
Water-SP	4096 molecules	53.23	54.01(1.4)	54.01(1.4)	55.21(3.7)	54.54(2.4)
Ocean-RW	258 ² ocean	21.00	21.67(3.2)	23.93(14)	23.38(11)	23.34(11)
Volrend	head	4.11	4.12(0.43)	4.14(0.73)	5.36(30)	5.00(21)

Table 7.2 shows problem scales, sequential execution times, and single-node execution times for parallel programs which are optimized and generated by the RCOP. In the table “FOW” represents the use of a conventional fetch-on-write method in the cache-management codes, while “w/o FOW” means that the RCOP avoids generating the code for a fetch-on-write if this is possible. The fetch-on-write codes, in some cases, will cause false-sharing or wasteful fetching of data and thus increase the amount of traffic for cache maintenance and worsen performance.

The single-node execution times for the parallel programs are of course slower than sequential execution because of the additional code required for coherence maintenance, but the difference is kept low owing to the

optimization applied by the RCOP. If we use the “w/o FOW” method in the UDSM scheme, we can reduce the amount of code for checking and improve performance. In the Water-NS and Raytrace programs, there are several points at which the RCOP is not able to merge maintenance code because of the synchronization aspect of shared-read operations. In such cases, therefore, the “w/o FOW” will, conversely, worsens the performance.

The overheads for maintaining coherence in the ADSM scheme are smaller than in the UDSM scheme. This is because maintenance code is inserted for both shared-writes and shared-reads with UDSM, but only for shared-writes with ADSM. If we use the conventional “FOW” method, the overheads are very low and the maximum contribution is 3%. If we use the “w/o FOW” method, Radix, FFT, and Ocean-RW suffer rather large overheads that approach 20%. This is because of the additional copy-overhead required to avoid fetch-on-writes operations. However, these overheads can be distributed among multiple nodes, and the overall performance in actual parallel execution can be improved.

7.7.3 Performance in terms of 16-node execution

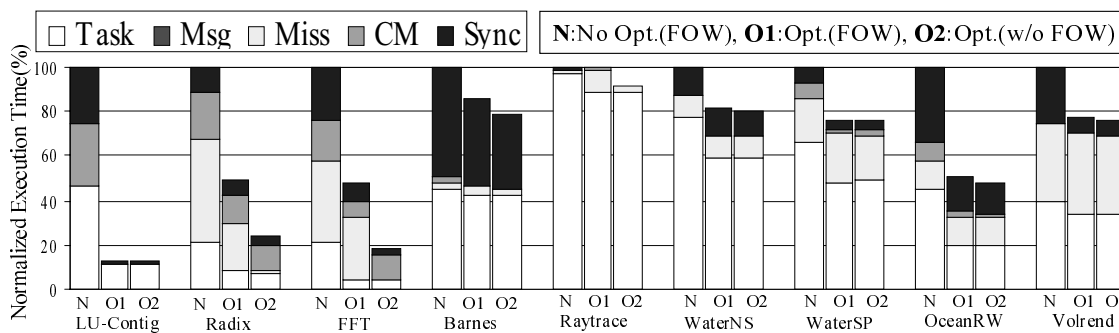


Figure 7.1: Effects of UDSM optimization techniques on 16-node execution

Figure 7.1 shows the effects of our optimization techniques on 16-node parallel execution using the UDSM scheme. Three bars correspond to each program of SPLASH-2, and the left-hand bar corresponds to the execution time with no optimization, the middle bar represents the execution time with optimization and the “FOW” method, and the right-hand shows the execution time with optimization and the “w/o FOW” method. In the figure “Sync” means the total time spent waiting for synchronizations, “CM” is time spent processing maintenance codes for cache consistency, “Miss” is time spent waiting for cache-fills on cache-misses, “Msg” is time spent processing requested messages, and “Task” is total time spent on calculations intrinsic to the problem.

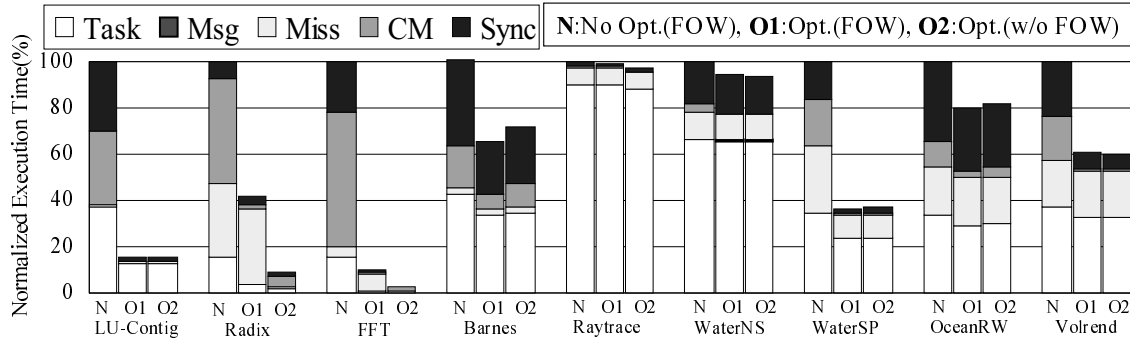


Figure 7.2: Effects of ADSM optimization techniques on 16-node execution

Figure 7.2 shows the effects of our optimization techniques on 16-node parallel execution using the ADSM scheme. In the Radix and FFT programs, there are critical parts where it is possible to avoid fetch-on-write and thus dramatically reduce the amount of communications. The “w/o FOW” method is thus very effective for these programs. For the Barnes and Ocean-RW programs, the “w/o FOW” method leads to an increased overhead because of the additional copies that must be generated for this method, and this performance is thus slightly worsened.

7.7.4 Adding nodes for increased speed

Figure 7.7.4 shows the speed increases that result from adding nodes in the UDSM and ADSM schemes. The ratios in the figure are relative to the execution time of the sequential version of the program. There are four bars for each of the SPLASH-2 programs, and the left most bar corresponds to the ratio with 8-nodes and ADSM, the bar at center-left represents the ratio with 8-nodes and UDSM, the bar at center-right represents the ratio with 16-nodes and ADSM, and the right most bar tells us the ratio in 16-nodes and UDSM. Using our optimization methods and the MBCF, we can obtain an increase in speed-up even for the Radix and FFT programs, for which improved performance on workstation clusters is difficult to obtain [25].

7.8 Related Works

7.8.1 Shasta

The Shasta [65] binary translator supports fine-grained coherence by rewriting application binary-codes to intercept shared loads and shared stores. This mechanism of Shasta is an emulation of the invalidate-type hardware DSM mechanism. The Shasta also carries out certain peep-hole optimizations to improve the

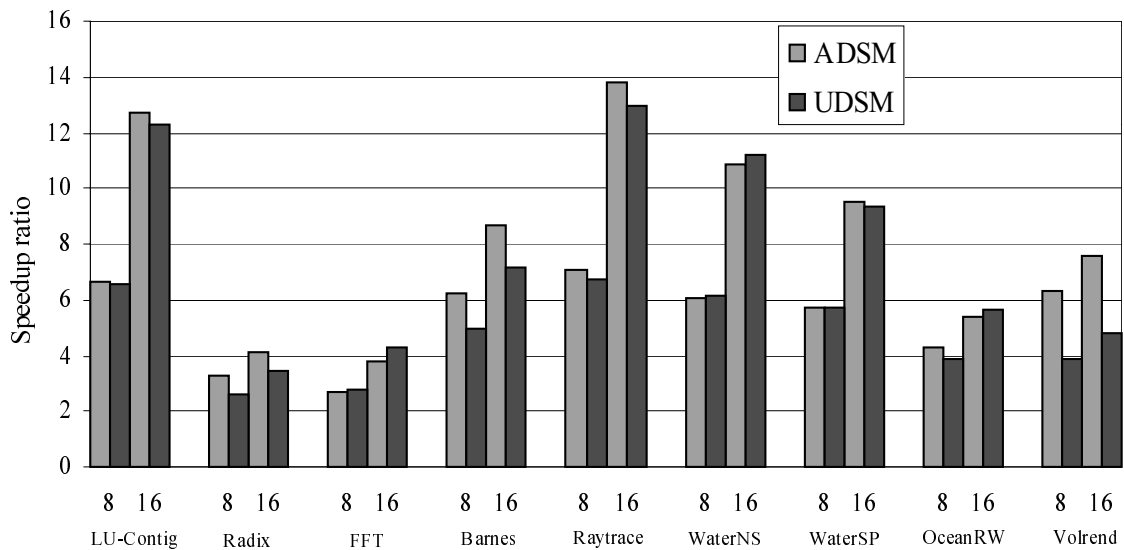


Figure 7.3: Speed-up ratios by the UDSM and the ADSM

performance of code in execution. The Shasta does not use loop-level, interprocedural optimization, so it requires a large-bandwidth and low-latency network. Shasta's goal is to reduce processing overheads to maintain consistency while preserving fine-grained shared-accesses. In other words, Shasta is developed for Memory Channel [14] machines with their large-bandwidth and low-latency network. Furthermore, the Shasta must create codes in which an invalidate-type protocol is emulated, because the Memory Channel machines do not have any invalidate-type consistency protocols. My approach changes fine-grained accesses to shared memory into medium-grained or coarse-grained accesses while retaining the meanings of parallel programs to reduce overheads resulting from both communications and processing to maintain consistency. Our optimizing compiler RCOP, which has been developed for my approach to realization of compiler-assisted DSM, issues the cache-coherence management routines for shared-memory accesses in coarse-grains by using source-program information. My approach, therefore, is more suitable for computer clusters in which commodity communication hardware with a non-negligible overhead is used. My compiler-assisted DSM approach (UDSM/ADSM) with the MBCF allows computer clusters to support practical shared-memory parallel programs.

Chapter 8

Memory-Based Processor II (MBP2)

8.1 Middle-grained hardware implementation of the MBCS

The Memory-Based Processor II (MBP2) [54, 49] is third implementation of the Memory-Based Communications and Synchronization (MBCS) scheme and second hardware one of the MBCS. Since a conventional network standard (i.e. gigabit ethernet) is exploited in the MBP2 system, grain size of communications is equal to the size of data-packet of ethernet. Consequently the MBP2 is a medium-grained hardware implementation of the MBCS. This grain-size is suitable for mitigation of the overhead cost for communications, and our compiler-assisted software DSM approach in Chapter 7 can efficiently support the MBP2 system as well as the MBCF system which has the same level of grain-size.

The MBP2 architecture was designed as a hardware support mechanism for the MBCF scheme. We have already described the MBCF's superiority over message-passing-style mechanisms when conventional NIC hardware is implemented in the system. However, this condition is only required for the exclusion of dedicated message-passing hardware. The architectural aspects of the MBCF are superior to those of message-passing-style mechanisms because the strong points of the MBCF do not depend on specific hardware structures of the NICs.

In this chapter I will discuss hardware-support mechanisms for the MBCF scheme. As mentioned earlier, the DMA mechanisms of conventional NICs cannot securely handle user-space of memory. Owing to this lack of NIC functionality, the main processor must perform additional copies of data at both the sender and receiver sides. If the NIC were to have an address-translation mechanism (TLB/MMU) like that of the MBP, no making of copies by the main processors would be required in the MBCF scheme. On the other hand, in message-passing-style schemes we can never completely eliminate the copying of data since the NIC does not recognize target locations in user-space.

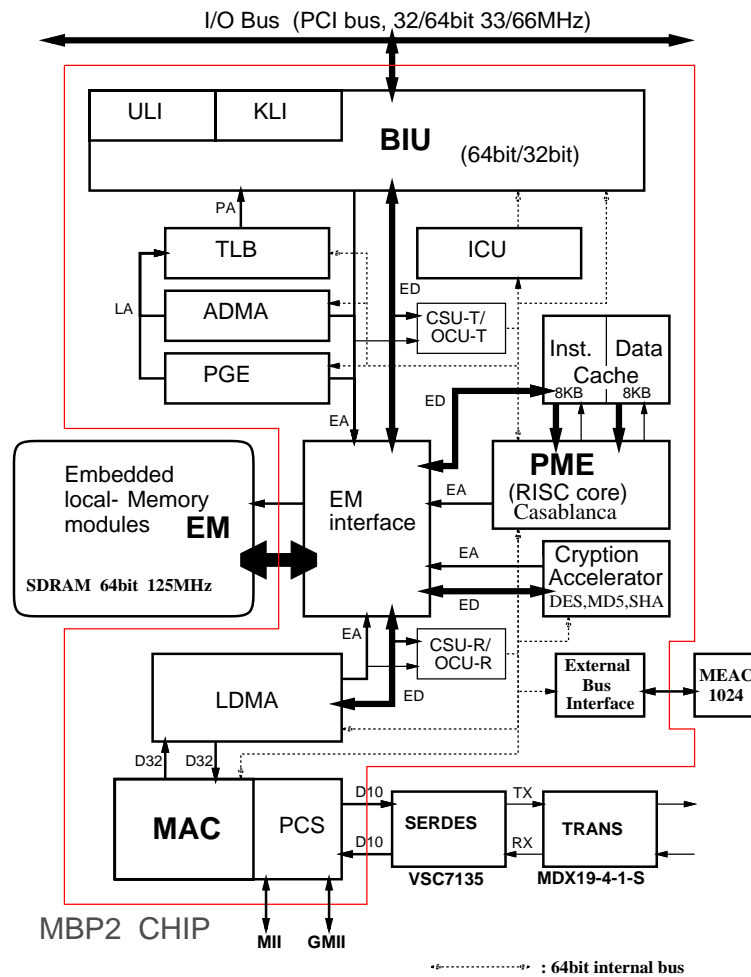


Figure 8.1: Block diagram of MBP2C

The MBP2 is a new network interface architecture and it is based on the MBCF scheme, with the weak points of the MBP eliminated. Figure 8.1 shows an implementation of the MBP2, the “MBP2C”. This is a system-on-chip LSI for high-performance communications with gigabit ethernet connections. The MBP2C uses the general-purpose PCI I/O extension bus as its connection-interface with the host system. The local memory bus or system bus of the main processor will be greatly superior to the PCI bus in terms of bandwidth, but the PCI bus is very popular and has a high-level of compatibility among various systems. Since I hope that the MBP2C will itself become to be a commodity item, I adopted the PCI as the host-connection interface.

8.2 System structure of the MBP2C

Before explaining the features of the MBP2 or the MBP2C, I use Figure 8.1 to introduce and describe their components.

- Card-Embedded Memory (**EM**)

The EM is the on-board high-speed memory of the NIC that is equipped with the MBP2C. It is accessed independently via four routes: to and from the I/O bus, MAC, PME and Cryption Accelerator. The EM is primarily intended as a temporary buffer for data transfer. To attain this purpose without degradation of performance over gigabit ethernet connection, its bandwidth must be four times as large as that for gigabit-ethernet communications. To get this high bandwidth, I adopted a wide data-path (more than 64bit wide) between memory modules and the MBP2C chip. Synchronous DRAMs are used as the MBP2C's memory modules to obtain good cost-performance ratio. "EA" in figure8.1 is added to the data-paths for EM addresses, and "ED" indicates buses for EM data.

- Protocol Management Engine (**PME**)

The PME is an embedded RISC-type microprocessor which utilizes the EM as its program and data storage area. It has no MMU or floating-point processor, and is a so-called RISC core. Firmware programs on the PME realize control of the MAC, protocols for assurance of packet arrival and order, management of communication buffers in the EM, page management for the TLB in the MBP2, IP protocol, TCP protocol, and so forth. Functionalities of the MBCF are obtained by the PME's control of the ADMA.

The MBP2C uses the "Casablanca" embedded processor [71]. This processor was developed expressly for the MBP2C and has special features that make it suitable for the MBP2 architecture. The Casablanca has interrupt-dedicated register sets for high-speed context switching and cache-control instructions for cooperation with DMA mechanisms.

- Local Direct-Memory-Access Unit (**LDMA**)

The LDMA transfers packets between the EM and the MAC by DMA. It has two channels of DMA corresponding to the two directions, and handles the speed-negotiations and handshakes for transfers to and from the MAC.

- Advanced Direct-Memory-Access Unit (**ADMA**)

The ADMA is a one-channel DMA unit for transferring data between the main memory of the host computer and the EM of the MBP2. Locations in the main memory are specified by combinations of context IDs and logical addresses in the MBP2, and these combinations are transformed to physical addresses (“PA”) in the host system by the TLB in the MBP2. The MBP2 uses physical addresses on the host’s I/O extension bus to access the host’s main memory. The ADMA of the MBP2C can exactly transfer memory data that is not aligned without performance degradations.

- Packet-Generation Engine (PGE)

The PGE traverses data structures in the host’s main memory, and generates initial packet images in the EM. The control logic of the PGE is as complicated as that of the MAC, programmable engine is a candidate for an implementation of the PGE. The PGE of the MBP2C is, however, implemented as hardwired logic for high-performance. The PGE is also used as the host-memory interface unit for conventional message-passing-style mechanism which traverses the receive structures in the host’s main memory and transfers the payloads of packets from the EM to the host’s main memory. Like the ADMA, the PGE handles logical addressing of the host’s main memory and can exactly transfer memory data that is not aligned without performance degradations. In other words, the PGE is another logical-address-based DMA unit with special traversal functions. The data-path and TLB for the PGE are common with those for the ADMA.

- Translation Look-aside Buffer (TLB)

The TLB translates logical address information into physical address information for the host’s main memory. The ADMA and the PGE send enquires regarding address information to the TLB in order to access the host’s main memory via the I/O extension bus. Each entry of the TLB has a task ID (context ID) field, so the TLB can simultaneously hold entries for different address-spaces. When a TLB-miss occurs, the PME is interrupted and sets up a new entry which is fetched from the page table entries in the EM. The ADMA and the PGE are also able to access the host’s main memory without transformation by the TLB (bypass mode). This ability is, however, only available in the kernel (supervisor) mode.

- Checksum Unit (CSU) and Order Checking Unit (OCU)

The CSU is the unit which calculates checksums for TCP and/or UDP. The OCU is an acceleration unit to assure packet-order for the MCBF and/or TCP. In actual implementations, each unit consists of two sub-components: a sending part (CSU-T/OCU-T) and a receiving part (CSU-R/OCU-R). The receiving part uses data which are transferred from the MAC to the EM by the LDMA in its calculations, while

the sending part handles data which are transferred from the host's main memory to the EM by the PGE. The PME uses results from both units to accelerate its protocol processings.

- I/O Bus Interface Unit (**BIU**)

The BIU is the interface between the MBP2C and the I/O extension bus (PCI bus). Since the ADMA and the PGE of the MBP2C access the host's main memory, the bus-master function must be implemented in this unit. The EM in the MBP2 card can be mapped to some location in the host's memory space, and can then be accessed, via the BIU, by the main processor in the host.

- Cryption Accelerator Unit (**CAU**)

The CAU is a cryptographic function unit with DMA engines, and has enough calculation ability to support gigabit networks. It applies the transformations involved in DES or triple DES to data in the EM, and also calculates hash functions for authentication. The data are fetched by the DMA engines in the CAU, and the results of transformation are stored by the DMA engines.

- MEAC1024

The MEAC1024 is a commercial LSI chip from IBM Corp. for accelerating the Internet Key Exchange algorithm and Public Key Encryption algorithms. It has facilities for fast 1024-bit modular multiplication and fast 1024-bit modular reduction. The MEAC1024 is included on the card of the MBP2C to support the PME.

- Interrupt Control Unit (**ICU**)

The ICU is for asynchronous communications, by interrupt, between the main processor and the PME. It can generate interrupt signals for the main processor and/or the PME, and these interrupts can be accompanied by 64bit messages.

- Kernel-Level-Interface Unit (**KLI**)

Kernel-level programs on the host machine are able to access the EM directly, so the KLI provides an address area for the EM which is mapped to part of the host's physical address space. The kernel programs can thus communicate with the PME via shared variables on the EM.

- User-Level-Interface Unit (**ULI**)

The ULI is for the direct invocation of the PGE by user-level programs. In spite of user-level direct access to this ULI, it allows the PME to recognize which user process has accessed the ULI of the MBP2. This mechanism is based on Matsumoto's device [42].

- Media Access Controller (**MAC**)

The MAC is a controller unit for processing ethernet protocols and communications.

- Physical Coding Sublayer (**PCS**)

The PCS is the unit for the physical layer of ethernet communications. It performs autonegotiation, flow control for full-duplex communications, pausing functions and encoding/decoding of the 8B10B transformation.

- Serializer-Deserializer Unit (**SERDES**)

The SERDES transforms the parallel output of the PCS into the serial input for the TRANS, and vice versa.

- Gigabit Ethernet Transceiver (**TRANS**)

The TRANS is transceiver for fiber-based gigabit ethernet link. MBP2C has two interfaces for gigabit ethernet, one is for the TRANS (the SERDES), the other is for the GMII (MII), a standard media-independent interface.

8.3 Features of the MBP2

In this section I describe the unique features of the MBP2.

8.3.1 Architecture based on the MBCF

As I mentioned before, the biggest feature of the MBP2 is its architecture that is based on the MBCF scheme. An MBP2 can handle the logical addresses of tasks at its node and transmit data directly among tasks without requiring system-calls or copying of data by the main processors. The MBP2 can also send medium-grained data in a packet and its limit is much greater than the transmission size of the MBP. This solves the problem of the utilization of the network and/or bus in an MBP system being limited to a low bandwidth. Since I hope that the MBP2C will be a commodity in world-wide use, I adopted the PCI bus as the interface with the host, and the MBP2 card supports the TCP/IP [62] and IPSec [27, 28] protocols. In our TCP/IP and/or IPSec implementation, neither system-calls nor data-copies-by-processor are required for communications, exploiting the MBCF technology, and no CPU (main processor) power is thus needed for the processing of protocol stacks.

8.3.2 Protected and virtualized user-level direct I/O access

In the MBP2 architecture users can communicate without system-calls for protected and virtualized sending or receiving. On the receiving side, data in communication packets are directly stored in the locations in user-task spaces, and so no system-calls for receiving are required. As for the sending side, there is a DMA mechanism for sending packets, but some mechanism is also needed to inform the MBP2 of the data-ready-for-sending state so that the DMA can be started. To realize this mechanism without system-calls, I invented a brand-new I/O access method, the “Memory-based Virtual Interface (MVI)” [42] and this is implemented in the ULI of the MBP2. The MVI allows I/O devices to identify which task is accessing their registers. This information on task identity is used by firmware programs in the I/O devices for protection and virtualization. In the MBP2C case, the ULI has a register for informing the MBP2C of the data-ready-for-sending state of a task but the register has multiple entry addresses (aliases) in pages of the host processors. The OS of the MBP2 system allocates only one alias address to each user-task. When any user-task accesses the register through its unique alias-address, the MBP2C recognizes the alias which is used for the access and then recognizes the accessed task. This is because the OS informed the MBP2C of the correspondence between the alias and the task when it allocated the alias to the task.

8.3.3 Full-DMA-connection among functional units

All functional units which process the payloads of packets have their own DMA mechanisms and data communications among those units is thus solely by DMA.

On sending a packet, the PGE is firstly invoked by the MVI in the ULI and uses its DMA function to transfer data from the main memory in the host to the EM. The transferred image in the EM is a model for the packet to be sent. The PME then generates the header for the packet to be sent by using the header of the image and information on the requestor-user-task which has previously been stored in the EM by the OS. During the generation of the packet header, the PME checks for the necessity of encryption, and starts the CAU’s DMA to process the data in the EM if encryption is necessary. After completion of making of the final image for sending in the EM, the PME starts the LDMA’s DMA to transfer the image into the MAC and send it over the network.

The process for receiving is the reverse of the process for sending. The packet which is received by the MAC is transferred into the EM by the LDMA’s DMA, and the PME reads and checks the header part of the packet. If the PME decides that the packet should be decrypted, it starts the CAU’s DMA and the decrypted data are consequently stored in the EM by the DMA. If the packet is an MBCF packet, the PME starts the DMA of the ADMA using the parameters in the header of the packet, and realizes the specified operation

(remote-write, remote-read, and so forth). If the packet is a TCP/IP or UDP/IP packet, the PME start the DMA of the PGE to transfer the packet image into the buffer area in the user-space/kernel-space which is specified by the information on the TCP connection or port number.

As described above, the functional blocks of the MBP2 are connected by the shared memory, EM, and all blocks which process the data parts of packets have DMA capabilities. Therefore, the PME and/or the main processor in the host do not need to copy or scan the data.

8.3.4 Embedded microprocessor which can cooperate with DMA mechanisms

All DMA mechanisms of the PGE, the ADMA, the LDMA and the CAU have their own descriptor rings which are ring buffers in the EM, and multiple entries in each ring can buffer speed differences among the functional blocks. The information that a block has completed a function or that a block has failed to perform a function is obtained by updating the ring-entry of its DMA. Furthermore, the PGE must read the header of the packet image in the EM to interpret and change the header. In other words, the areas of descriptor rings and of packet buffers are shared-memory between the PME and the DMAs, and are modified by all of them. An embedded processor like the PME has a data-cache memory to reduce memory-access latency, but it does not have a snooping or coherency mechanism for the cache. A conventional embedded processor must thus allocate the area that is shared with the DMAs to non-cacheable pages because of the need to avoid cache-inconsistencies, and access to the shared area will then have a high cost in terms of large access latencies. As for the MBP2C, we designed a new embedded processor “Casablanca” which overcomes the weak points of conventional embedded processors. To improve the cost of access to the shared area, we implemented two new cache-control instructions in the new embedded processor [43]. One is an explicit cache-fill instruction and the other is an explicit writeback instruction. The former causes an external load operation and cache-fill by new data even if the cache is hit. The latter causes an external writeback operation for a dirty cache entry even if cache replacement does not occur. We use these instructions in firmware programs to maintain coherence for the shared-area.

8.3.5 Embedded microprocessor with zero-cost context switching for interrupts

In the MBP2 the PME must process some procedure on every completion of DMA processing. On completion of DMA processing, the PME must collect the used entry of the descriptor and/or start the next round of DMA processing. If we use the interrupt mechanism of the PME to detect the completion of DMA processing, the frequency of the interrupts will be very high because of the high-throughput of the gigabit ethernet link. Even if the data-length of the packets is assumed to be 1Kbyte, there are 2 interrupts for receiving and 3 interrupts

for sending during an $8\mu\text{sec}$ interval in a full-speed (1Gbit/sec) transmission. The number of interruptions increases if we use encryption facilities. Conventional embedded microprocessors cost a few μsec per interrupt for preparation of the interrupt routine, and cannot handle high-frequency interrupts. We implemented a zero-cost context switching mechanism in the Casablanca to solve this problem. The Casablanca has multiple register sets for different contexts and switches the current register set on an interrupt, without any pipeline stall. Owing to this context switching mechanism the Casablanca is better at using interrupts for events than at using polling, which costs in terms of time taken up by wasteful access. Moreover, programs based on interrupts are suitable for priority-sensitive routines, as long as the interrupt mechanism supports multiple interrupt-levels.

8.3.6 Encryption hardware embedded in a NIC

Encryption/Decryption functions are intrinsically related to communications, but the acceleration hardware for these functions would normally be implemented as an additional card, independent of the NICs. The conventional architecture causes a wasteful traffic of data on the extension bus and the system bus for cryptographic functions. In the MBP2 architecture, the hardware for cryptographic functions is implemented in the NIC and data traffic among functional units does not use any external buses. The MBP2 is the first architecture to solve this problem of data traffic for encryption/decryption.

8.4 Prototype of the MBP2C

The MBP2C is a system-on-chip LSI and required a huge budget for development. I and Mr. Haruyasu Oyobe of Sansei Systems Co. Ltd have developed a prototype of the MBP2C. We call it the MBP2P. It has the same features as the MBP2C. Figure 8.2 is a block diagram of the MBP2P, which is a multi-chip implementation of the MBP2C. The MBP2P adopts a large-scale field programmable gate array (XCV1000-6FG680) as the main controller, a commercial embedded microprocessor (SPARC*lite*:MB86832), and a commercial media access controller (XQ11800) for gigabit ethernet in order to make rapid development possible. In the MBP2P design I also changed the PCI bus to an SBus, because I was very familiar with the Sun Microsystems' workstations built around the SBus. The bandwidth of the SBus is lower than that of the PCI bus. The biggest functional difference between the MBP2P and MBP2C is the lack of circuitry for cryptographic functions in the MBP2P. This circuit is too large and complicated to be integrated in a single field programmable gate array. Therefore, in the MBP2P, cryptographic functions are performed by the firmware programs of the PME.

Figure 8.3 is a schematic circuit diagram of the NIC based on the MBP2P, and Figure 8.4 represents is a photograph of the MBP2P NIC card.

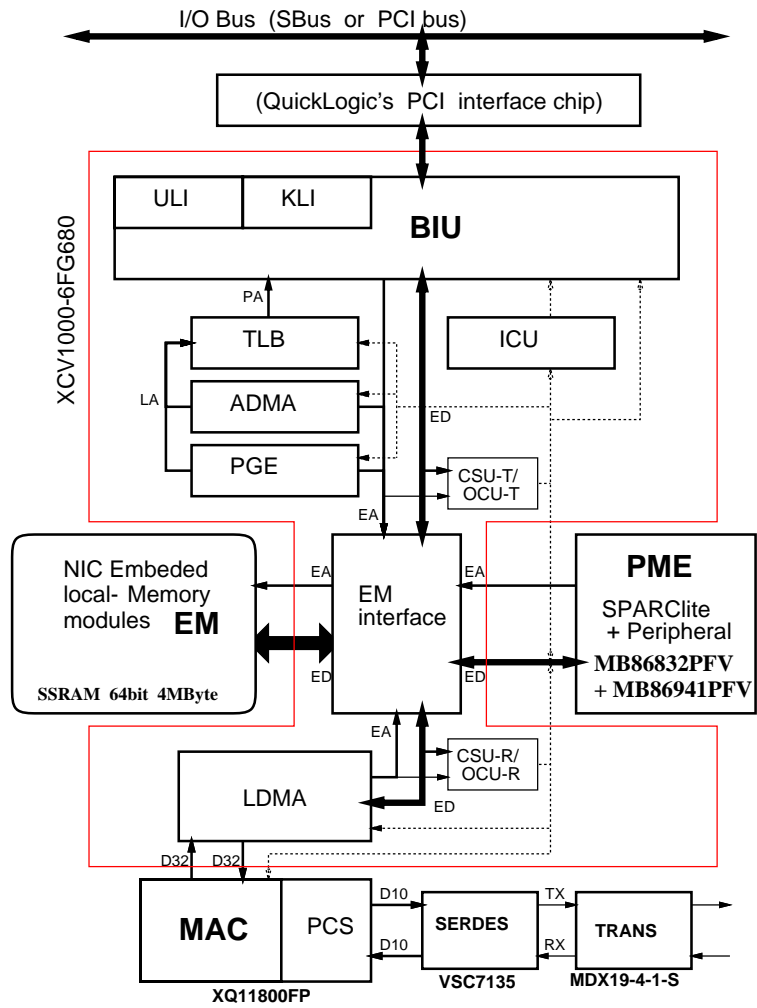


Figure 8.2: Block diagram of the MBP2P

Table 8.1 explains the operational environment of the MBP2P.

Table 8.1: Operational environment of the MBP2P

Host computer system	Sun SPARCstation 20
Operating system	SSS-CORE Ver.1.2-p
SPARClike monitoring interface	RS232 (9600bps)
FPGA download interface	Xilinx JTAG cable (printer port)
Ethernet input/output	1000BASE-SX (850nm)

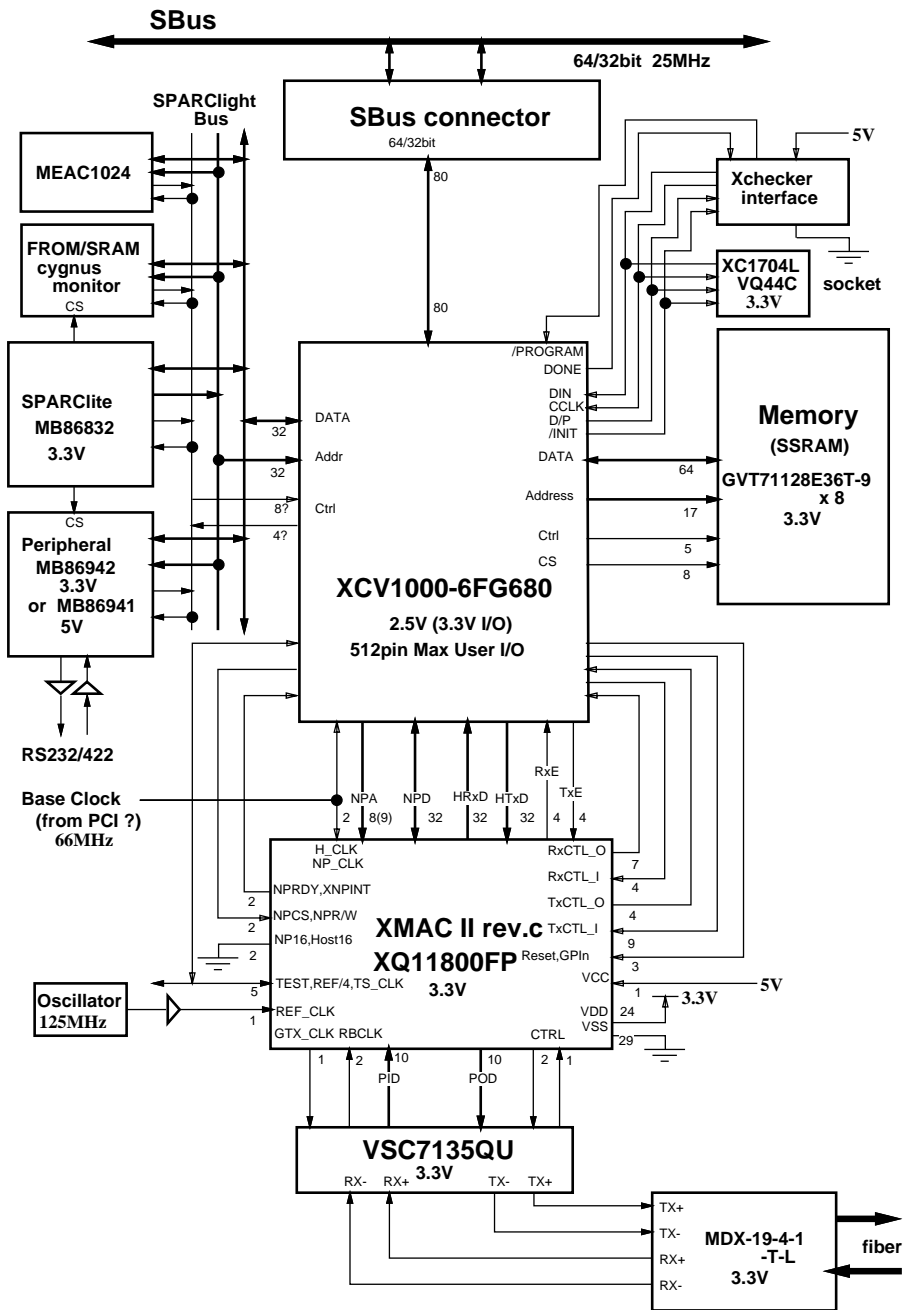


Figure 8.3: Circuit diagram of the NIC based on the MBP2P

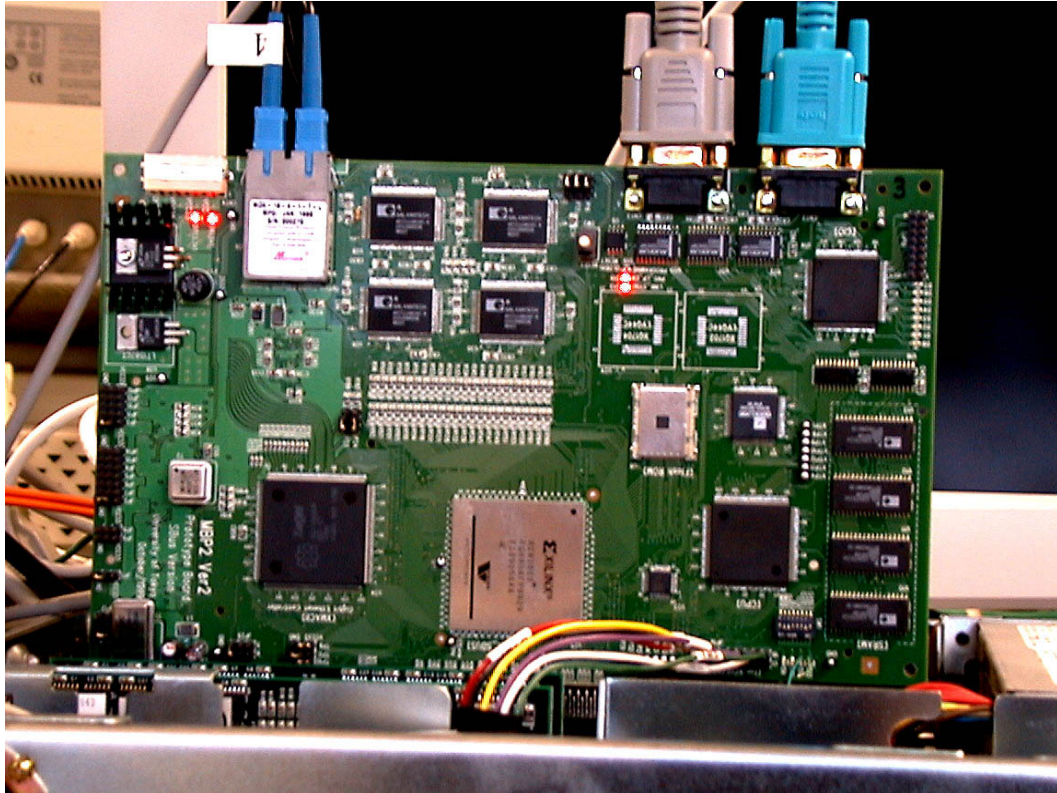


Figure 8.4: Photo of the MBP2P NIC card

Table 8.2 represents the important hardware specifications of the MBP2P. The figures in the table are real operational values.

Table 8.2: Hardware specifications of the MBP2P

SPARClite internal clock frequency	100MHz
SPARClite bus frequency	20MHz
FPGA occupied resources	about 600,000gate
FPGA operational frequency	40MHz
SBus frequency	25MHz
SBus bus bandwidth	4byte
SBus max burst-transfer size	32byte

8.5 Discussions on the MBP2

Sending by the MBP2P only takes about $3\mu\text{sec}$ of main-processor power, because no protocol-stack processing and no system-calls are required. This value is not affected by the amount of payload, and MBP2P's overhead cost of sending a packet with 1024byte payload is about one-fifth of the value for MBCF/Ether on a SPARCstation 20 with 100BASE-TX. However, the round-trip latency of the MBP2P is more than $140\mu\text{sec}$, and this value is about three times the latency of the MBCF/Ether on a SPARCstation 20 with 100BASE-TX. The biggest factor in the latency is the performance of the microprocessor which processes protocol stacks. I roughly estimated that the performance of a SPARClite (100MHz) is almost equal to that of a SuperSPARC (85MHz) which is the processor in the SPARCstation 20. This estimate was greatly mistaken. Though the frequencies of the two processors are almost the same, the SPARClite has many weak points: lack of a secondary cache, write-through policy of its cache, very small amount of instruction cache, and lack of snooping mechanism. Conventional embedded processors are greatly inferior to commercial main processors. Since the performance of the high-end commercial processors is continually increasing, software implementations like MBCF/Ether have the advantage being able to exploit this high-performance and will attain lower-latencies of communication than hardware-support implementations with embedded processors, such as MBP2P. The hardware-support solution is still effective in terms of relieving main processors of the load of communications processing.

8.6 Related Works

8.6.1 T3D

The T3D [8] is one of earliest distributed-memory parallel computers with a logical-address-based remote-DMA mechanism. The T3D has a dedicated network in which there is no packet-loss. The first MBP was also able to perform similar logical-address-based remote-memory accesses, and the MBP is an earlier design than the T3D's DMA. Published work on the MBP also appeared earlier. The address-translation mechanism for remote access originates in the MBP. The difference between the MBP and the T3D's DMA is the grain-size of memory transfers. In other words, the MBP's transfers are fine-grained while the T3D's are medium-grained or large-grained. From the point of view of physical data transmission, fine-grained remote-memory access is inefficient because of the overheads contributed by packet-headers and parameter-setups. The more capable MBP2 is current and handles medium-grained transmission, the approach is more similar to the T3D's DMA. The T3D's DMA cannot perform a variety of memory-based commands but is limited to remote-write and remote-read operations. Though MBP2 can be directly accessed by user tasks under protection, the T3D's

DMA requires that system-calls be used to set parameters up.

8.6.2 AP1000+

The AP1000+ [18] is one of earliest parallel computers to feature distributed-memory with a logical-address-based remote-DMA mechanism, but the DMA mechanism of the T3D in the previous subsection was designed earlier than that of the AP1000+. Like the T3D, the AP1000+ has a dedicated network in which there is no packet-loss. The original point of the DMA mechanism of the AP1000+ is that it has such special operations as MBCF_WRITE_F (remote write with flag-set). Like the DMA mechanism of the T3D, the DMA of the AP1000+ requires that system-calls be used to set up parameters for protection. This creates an additional overhead cost in comparison with the MBP2.

The communication mechanism of the AP1000+ is not only able to perform remote DMA but is also capable of message-passing-type communications. A “return-to-sender” policy was not adopted for full fifo buffers. Instead, interrupts are generated to the main processors. Even if we spent the time to develop no-packet-loss network such as the AP1000+, such a buffer-full policy would spoil the merits of the network. In other words, packet copies of packets must be kept at the requestor nodes in case they must be resent. I think this policy is a kind of bug in the system design.

A more advanced and unique policy than the simple “return-to-sender” policy was adopted for the MBCF and the MBP2. Please see the description under “MBCF_FIFOe” in Section 4.1 of “The commands” in Chapter 4 of “Functions of the MBCF”.

Chapter 9

Concluding Remarks

The descriptive power of the shared-memory model are equal to those of the message-passing-style (send-recv-type) model in terms of the writing of parallel programs. In other words, the one is able to emulate the other. From the viewpoint of performance, however, the situation is different. It goes without saying that memory is an important and fundamental building-block of the nodes of distributed-memory multicomputers. It is very significant, in terms of improving communications performance and of increasing the variety of functions that accompany the communications, that it is possible to exploit information on buffer locations and data locations in the communication/synchronization subsystem, whether this is implemented in hardware or software. Moreover, by applying memory-management methods, such as MMU/TLB mechanisms for processors, to remote communications, protected-communication subsystems can be constructed. On the basis of the above principals, I invented and proposed three Memory-Based Communications and Synchronization (MBCS) mechanisms, and I actually developed two of them for experimental verification and for research into implementation techniques. The earliest one was the Memory-Based Processor (MBP), a fine-grained co-processor built in the main memory of the nodes to handle operations related to communications and synchronization. The second was the Memory-Based Communication Facility (MBCF), which is a software-only solution for the realization of protected and virtualized high-speed user-level communications and synchronization. I developed the MBCF communication subsystem to run under the *SSS-CORE* operating system for clusters of workstations with fast ethernet and/or gigabit ethernet. The third was the Memory-Based Processor II (MBP2), a new network interface architecture for the support of the MBCF over gigabit ethernet links by taking the work-load of communications and synchronization from the host processors. I developed a prototype of the MBP2 to evaluate its actual performance and for a comparison of the MBCF and MBP2 approaches.

Several key technologies for the construction of efficient hardware DSMs were also invented and proposed

as part of the research associated with the MBP. The MBP was the first device to have an address-translation system for remote-memory access to realize protected memory-based communications. The hierarchical bitmap directory was introduced to the MBP system to reduce the amount of directory-tag memory. The directory has only insufficient information about nodes with shared copies, but the area for the multicasting of consistency-maintenance-operations can be limited within the tree-branching layers of the system. A new hardware DSM scheme which has page-level directories and cache-block-level state tags in main memory was adopted for the MBP. This scheme allows hardware DSMs to exploit main memory as remote-cache memory while reducing the amount of directory-tag memory required, and by cache-block-level coherence operations avoid false-sharings. The “hierarchical multicasting and acknowledge-message combining” method was proposed in the MBP project. This method resolves the hot-spot problem in the collection of acknowledge messages at home nodes. This method is a key technology in the construction of large-scale coherent DSM systems.

In the research into the MBCF, I developed the *SSS-CORE* general-purpose scalable operating system. The MBCF is a basic and essential primitive for communications among the tasks running under this OS. Owing to the purely software implementation of the MBCF, the MBCF is able to use commodity network hardware (e.g., ethernet cards). In spite of only using commodity network hardware, recent generations of high-performance processors are able to attain very good levels of performance in communications and synchronization with the MBCF scheme. In the MBP approach, there was a big problem in that fine-grained communications induced by the MBP are inefficient in transmissions on the buses of nodes and across the network. Since the MBCF uses software to control conventional medium-grained network hardware by software, the grain-size of communications can be enlarged to reduce overheads and to more efficiently utilize the network or buses. The MBCF is qualitatively superior to the message-passing-type interface or the SparcStation Active Message. In other words, the MBCF-style interface is fit for higher-speed implementations than other methods. I gave the proof of this superiority of the MBCF in this thesis.

Various functions for exploiting information on target locations were proposed in the MBCF scheme, and various implementation techniques to do with software engineering and advanced processor architectures were introduced to make the light-weight protocol stack of the MBCF. For the MBCF_FIFOe, which is one of a variety of MBCF commands, a new modified “return-to-sender” method was adopted. Even if senders transmit MBCF_FIFOe packets in bursts and some fifo-queue thus overflows, this modified method can keep the order of point-to-point packets intact without requiring any additional synchronization.

A protocol for providing the MBCF over ethernet was designed to guarantee the arrival and order of packets. A new window-based (GO-back-N) technique, where N is dynamically decreased at a hint of a

lack of buffers, was adopted for the protocol. The technique assures both high-performance and scalability. MBCF/Ether was actually developed on clusters of workstations, SPARCstation 20s and Ultra 60s, running the *SSS-CORE* operating system. Performance evaluation using 100BASE-TX and 1000BASE-SX were carried out, and the results were a one-way latency of in $24.5\mu\text{sec}$ and data-transfer rate of 11.93Mbyte/sec for 100BASE-TX and a one-way latency of $9.6\mu\text{sec}$ and data-transfer rate of 80.92Mbyte/sec for 1000BASE-SX. Note that communications for these measurements were among two user-level tasks and the processing times or round-trip times were measured from within a user-level task.

A brand-new approach to software DSM for the medium-grained MBCS mechanisms including the MBCF was proposed in this thesis. In my approach, user-level codes for consistency management are inserted in the program code at compile time for inclusion in the executable modules. Both original code and the inserted codes are thoroughly optimized to reduce overheads and the amount of remote communications. This approach was revolutionary and the conventional trap-based methods of software DSMs were abandoned. This approach allows workstations with commodity network-interface cards (e.g., 100BASE-TX ethernet) to efficiently support shared-memory parallel programs. Two new software DSM methods were also proposed, and are based on the above approach. The UDSM is a fully user-level DSM where all codes for consistency maintenance are inserted in the user-application code, and the ADSM partly accepts the use of trap-based methods for detecting read-misses of shared-pages. Both DSMs are implemented under the *SSS-CORE* operating system with the MBCF/Ether protocol. Using these software DSMs on clusters of commodity workstation, fine-grained shared-memory parallel applications (e.g., the SPLASH-2 Radix program and SPLASH-2 FFT program) can be speed-up, though it had been very difficult to improve the performance of these programs in workstation clusters until invention of my new approach.

A new network-interface architecture, the MBP2, based on the MBCF scheme, was proposed to remove heavy-weighted communications processing from the main processors. A prototype of the MBP2, called the MBP2P, was developed to evaluate the effectiveness of the MBP2 and for comparison with software implementation of the MBCF. The MBP2P succeeded in removing the work-load from the main processors but the round-trip latency of the MBP2P was worse than that of the software MBCF. The biggest factor in this latency was the performance of the microprocessor which processes protocol stacks. The embedded processor of the MBP2P was much poorer at this than the main processor of the host. Conventionally, embedded processors are very much inferior to commercial high-end microprocessors. Even in the future, software implementations of the MBCF will attain lower-latency communications than hardware-supported implementations with embedded processors, such as MBP2P. The hardware-supported solution will still, however, be effective in relieving main processors of the load of communication processing.

Through this research on the MBCS schemes, I have learned that when operating systems and optimizing techniques were immature, fine-grained memory-coupling methods (e.g., tightly-coupled multiprocessor, hardware DSM, first MBP) had to be adopted for communications and/or synchronization among processors carried out by simple load/store instructions. At the recent level of technology, however, medium-grained or large-grained memory-coupling methods such as the MBCF are superior to the fine-grained methods from the point of view of efficiency of data transmission. Moreover, for the communication interface, the MBCF scheme is better than a message-passing-type scheme as has been described in this thesis. Therefore, the MBCF approach is the ultimate, or close to the ultimate, in terms of the construction of communications/synchronization subsystems on clusters of workstations/personal-computers with off-the-shelf network interface cards.

On the Linux operating system, which is the very popular POSIX-based OS, we have already developed a prototype of the MBCF/Ether subsystem [10]. The prototype has a subset of the MBCF functions and is running on UltraSPARC workstations. The success of this porting verified the portability of the MBCF. We are now porting the full MBCF run under Linux on IBM-compatible personal-computers.

As for future plans to do with this work, we will develop an efficient multi-trunked communication method for the MBCF scheme using multiple NICs per node and parallel networks in order to scale-up the bandwidths of nodes. If we can obtain high-performance off-the-shelf NICs, i.e., faster-than-gigabit-rate ethernet (and their datasheets), I will port the MBCF to these devices. We are also currently developing an embedded microprocessor called "Casablanca" [72]. The Casablanca processor is good at cooperation with DMA units and external interrupts owing to my brand-new mechanisms[43]. We will build another prototype of the MBP2 architecture around the Casablanca processor. In that prototype, the encryption/decryption support hardware will also be implemented. The adoption of the Casablanca processor will dramatically improve the latencies of the MBCF in comparison with those of the current prototype with its rather poor embedded microprocessor. On simple communication functions the prototype will still result in worse (greater) latencies than the software implementation of the MBCF/Ether. It will, however, be able to relieve main processors of a large part of their loads in terms of encrypted communications.

Bibliography

- [1] A. Agarwal et al. An Evaluation of Directory Schemes for cache Coherence. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 280–289, June 1988.
- [2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. 1990 Int. Conf. on Supercomputing*, June 1990.
- [3] J. Archibald and J. L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. Computer Systems*, 4(4):273–298, 1986.
- [4] M. Blumrich, K. Li, R. Alpert, C. Dubniki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for SHRIMP Multiprocessor. In *Proc. 21st Int. Symp. on Computer Architecture*, pages 142–153, April 1994.
- [5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE MICRO*, 15(1):29–36, February 1995.
- [6] D. Chaiken et al. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):49–58, 1990.
- [7] Compaq Computer Corp., Intel Corp., and Microsoft Corp. *Virtual Interface Architecture Specification Draft Revision 1.0*. <http://www.viarch.org/>, December 1997.
- [8] Cray Research, Inc. *Cray T3D System Architecture Overview*. Cray Research, Inc., March 1993.
- [9] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of 13th Int. Symp. on Computer Architecture*, pages 432–442, June 1986.
- [10] Marc Durantez. Towards a fast distributed shared memory paradigm for Linux. Master’s thesis, Department of Information Science, University of Tokyo, February 2000.

- [11] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. Comparative evaluation of fine- and coarse-grain approaches for software distributed shared memory. In *Proc. of the 5th Int. Symp. on High-Performance Computer Architecture (HPCA)*, pages 260–269, January 1999.
- [12] Izidor Gertner and Stephen Lucci. Infinity-Encore’s Shared-Memory Multiprocessors. In *Encore Computer Corp. Technical Report*, 1993.
- [13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th Int. Symp. on Computer Architecture*, pages 15–26, May 1990.
- [14] R. B. Gillet. Memory Channel Network for PCI. *IEEE Micro*, pages 12–18, February 1996.
- [15] J. R. Goodman. Using Cache memory to reduce Processor-Memory Traffic. In *Proc. of the 10th Int. Symp. on Computer Architecture*, pages 124–131, May 1983.
- [16] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuleffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing and MIMD Shared Memory Parallel Computer. *IEEE Trans. Computers*, pages 175–189, February 1983.
- [17] 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.
- [18] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihata, and T. Shindo. AP1000+: Architectural support for parallelizing compilers and parallel programs. In *Third Parallel Computing Workshop*, pages P1–F1–P1–F9, November 1994.
- [19] IEEE. *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Std. 802.3u-1995 & 802.3x-1997 edition, 1995.
- [20] IEEE. *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Std. 802.3 edition, 1996.
- [21] IEEE. *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Std. 802.3z-1998 edition, 1998.
- [22] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proc. the 2nd Int. Symp. on High-Performance Computer Architecture*, pages 14–25, February 1996.

- [23] T. Inagaki, J. Niwa, T. Matsumoto, and K. Hiraki. Supporting Software Distributed Shared Memory with a Optimizing Compiler. In *Proc. of the 1998 International Conference on Parallel Processing*, pages 225–234, August 1998.
- [24] Texas Instruments. *SuperSPARC User's Guide*, volume SPKU005. Texas Instruments, October 1992.
- [25] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Proc. of the 6th ACM SIGPLAN Symp. on PPOPP*, pages 217–229, June 1997.
- [26] 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, January 1994.
- [27] S. Kent and R. Atkinson. IP Authentication Header. Technical Report RFC 2402, Internet Society Network Working Group, November 1998.
- [28] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). Technical Report RFC 2406, Internet Society Network Working Group, November 1998.
- [29] 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 21st Int. Symp. on Computer Architecture*, pages 302–313, April 1994.
- [30] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Int. Symp. on Computer Architecture*, pages 148–159, May 1990.
- [31] D. E. Lenoski et al. Design of Stanford DASH Multiprocessor. In *Technical Report CSL-TR-89-403*. Stanford Univ., December 1989.
- [32] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 ICPP*, pages 94–101, August 1988.
- [33] Stephen Lucci and Izidor Gertner. Reflective-Memory Multiprocessor. In *Proc. of the 28th Annual Hawaii International Conference on System Science*, pages 85–94, 1995.
- [34] T. Matsumoto and K. Hiraki. A shared-memory architecture for massively parallel computer systems. *IEICE Japan SIG Reports CPSY*, 92(173):47–55, August 1992. (in Japanese).

- [35] T. Matsumoto and K. Hiraki. Elastic Memory Consistency Models. In *Proc. of 49th Annual Convention of IPS Japan*, pages 5–6, September 1994. (in Japanese).
- [36] T. Matsumoto and K. Hiraki. Resource management methods of the general-purpose massively-parallel operating system: SSS-CORE. In *Proc. of 11th Conf. of JSSST*, pages 13–16, October 1994. (in Japanese).
- [37] T. Matsumoto, J. Niwa, and K. Hiraki. Compiler-Assisted Distributed Shared Memory Schemes Using Memory-Based Communication Facilities. In *Proc. of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA-98)*, volume 2, pages 875–882, July 1998.
- [38] Takashi Matsumoto. Fine Grain Support Mechanisms. *IPS Japan SIG Reports*, 89(60):91–98, July 1989. (in Japanese).
- [39] Takashi Matsumoto. Synchronization and Processor Scheduling Mechanisms for Multiprocessors. *IPS Japan SIG Reports*, 89(99):1–8, August 1989. (in Japanese).
- [40] Takashi Matsumoto. A Study of FGMP: Fine Grain multi-Processor. *Trans. of IPS Japan*, 31(12):1840–1851, December 1990. (in Japanese).
- [41] Takashi Matsumoto. A Multiprocessor System with Memory-Based Processors and Register-Based Processors. In *Proc. of 43th Annual Convention of IPS Japan*, volume 6, pages 115–116, October 1991. (in Japanese).
- [42] Takashi Matsumoto. *User-Level I/O Access Methods*. Japan Science and Technology Corporation, Japanese Patent Application No.255272, 1999.
- [43] Takashi Matsumoto. Mechanisms for High-Performance Embedded Processors. *IEICE Japan SIG Reports*, 100(248):17–24, August 2000. (in Japanese).
- [44] Takashi Matsumoto and Kei Hiraki. Distributed Shared-Memory Architecture Using Memory-Based Processors. In *Proc. of Joint Symp. on Parallel Processing '93*, pages 245–252, May 1993. (in Japanese).
- [45] Takashi Matsumoto and Kei Hiraki. Dynamic Switching of Coherent Cache Protocols and its Effects on Doacross Loops. In *Proc. of the 1993 ACM Int. Conf. on Supercomputing*, pages 328–337, July 1993.

- [46] Takashi Matsumoto and Kei Hiraki. Memory-Based Communication Facilities of the General-Purpose Massively Parallel Operating System: SSS-CORE. In *Proc. of 53rd Annual Convention of IPS Japan*, volume 1, pages 37–38, September 1996. (in Japanese).
- [47] Takashi Matsumoto and Kei Hiraki. MBCF: A Protected and Virtualized High-Speed User-Level Memory-Based Communication Facility. In *Proc. of 1998 ACM International Conference on Supercomputing*, pages 259–266, July 1998.
- [48] Takashi Matsumoto and Kei Hiraki. Memory-Based Communication Facilities and Asymmetric Distributed Shared Memory. In *Proc. of the 1997 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 30–39, Los Alamitos, CA, 1998. IEEE Computer Society.
- [49] Takashi Matsumoto and Kei Hiraki. Research and development of the next generation network architecture based on a memory-based communication model. In *Proc. of Next Generation Digital Application Technology Development Projects*, pages 117–124. Information-Technology Promotion Agency (IPA), Japan, June 2000.
- [50] Takashi Matsumoto et al. MISC: a Mechanism for Integrated Synchronization and Communication using Snoop Caches. In *Proc. of the 1991 Int. Conf. on Parallel Processing*, volume 1, pages 161–170, August 1991.
- [51] Takashi Matsumoto et al. A General-Purpose Massively-Parallel Operating System: SSS-CORE – Implementation Methods for Network of Workstations –. *IPS Japan SIG Reports*, 96(79):115–120, August 1996. (in Japanese).
- [52] Takashi Matsumoto et al. The Asymmetric Distributed Shared Memory using Memory-Based Communication Facilities. In *Proc. of the IPSJ Computer System Symposium*, pages 37–44, November 1996. (in Japanese).
- [53] Takashi Matsumoto et al. A general-purpose scalable operating system: SSS-CORE. In *Proc. of The 20th Int. Conf. on Software Engineering*, volume 2, pages 147–152, April 1998.
- [54] Takashi Matsumoto et al. Memory-Based Processor II – A commodity supporting middle-grained memory-based communication –. *IPS Japan SIG Reports*, 98(70):103–108, August 1998. (in Japanese).

- [55] Takashi Matsumoto et al. The general-purpose scalable operating system: *SSS-CORE*. In *Proc. of The 17th Technical Conference*, pages 175–188. Information-Technology Promotion Agency (IPA), Japan, October 1998. (in Japanese).
- [56] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki. Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory. In *Proc. of the 1997 Int. Symp. on Parallel Architectures, Algorithms and Networks (ISPAN-97)*, pages 198–201, December 1997.
- [57] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki. Performance Evaluation of Compiling Techniques on Asymmetric Distributed Shared Memory. *IPS Japan SIG Reports*, 97(102):91–96, October 1997. (in Japanese).
- [58] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki. Performance Evaluation of Compiling Techniques on Asymmetric Distributed Shared Memory. *IPSJ Transactions on Parallel Processing*, 39(6):1729–1737, June 1998. (in Japanese).
- [59] J. Niwa, T. Matsumoto, and K. Hiraki. Comparative Study of Page-based and Segment-based Software DSM through Compiler Optimization. In *Proc. of 2000 International Conference on Supercomputing*, pages 284–295, May 2000.
- [60] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Message (FM) for Myrinet. In *Proc. of Supercomputing '95*, December 1995.
- [61] Jon Postel. User Datagram Protocol. Technical Report RFC 768, Internet Society Network Working Group, August 1980.
- [62] Jon Postel. Transmission control protocol darpa internet program protocol specification. Technical Report RFC 793, Internet Society Network Working Group, September 1981.
- [63] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, April 1994.
- [64] David F. Rogers. *Procedural Elements For Computer Graphics*. McGraw-Hill Book Company, New York, 1985.
- [65] 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 Int. Conf. on Architectural Support for Programming Language and Operating System (ASPLOS-VII)*, pages 174–185, October 1996.

- [66] M. Snir et al. The communication software and parallel environment of IBM SP2. *IBM Systems Journal*, 34(2), 1995.
- [67] SPARC International, Inc. *The SPARC Architecture Manual Version 8*, 1992.
- [68] S. Sumimoto, A. Hori, H. Tezuka, H. Harada, T. Takahashi, and Y. Ishikawa. High Performance Communication for Cluster Systems Using an Existing Operating System Framework. *Trans. of IPS Japan*, 41(6):1688–1696, 2000. (in Japanese).
- [69] S. Sumimoto, H. Tezuka, A. Hori, H. Harada, T. Takahashi, and Y. Ishikawa. The Design and Evaluation of High Performance Communication Using a Gigabit Ethernet. In *Proc. of 1999 ACM International Conference on Supercomputing*, pages 243–250, July 1999.
- [70] Sun Microsystems, Inc. *UltraSPARC-I User's Manual Revision 1.4*, volume STP1030-UG. Sun Microsystems, Inc., Mountain View, CA, January 1996.
- [71] K. Tanaka, T. Matsumoto, and K. Hiraki. *Casablanca*: Design and Implementation of Realtime RISC Core. *IPS Japan SIG Reports*, 99(100):51–56, November 1999. (in Japanese).
- [72] Kiyofumi Tanaka and Takashi Matsumoto. Evaluation of Realtime RISC Core *Casablanca*. *IEICE Japan SIG Reports*, 100(248):25–32, August 2000. (in Japanese).
- [73] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proc. of National Computer Conference*, pages 749–753, 1976.
- [74] K. Taura, S. Matsuoka, and A. Yonezawa. An Efficient Implementation of Object-Oriented Concurrent Language on Multicomputer. In *Proc. of Joint Symposium on Parallel Processing 1992*, pages 163–170, June 1992. (in Japanese).
- [75] K. Taura, S. Matsuoka, and A. Yonezawa. An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multicomputers. In *Proc. of Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 218–228, May 1993.
- [76] H. Tezuka, A. Hori, and Y. Ishikawa. Design and Implementation of PM: A Communication Library for Workstation Cluster. In *Proc. of Joint Symposium on Parallel Processing 1996*, pages 41–48, June 1996. (in Japanese).

- [77] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High Performance Communication Library. In *Proc. of High-Performance Computing and Networking*, pages 708–717. LNCS1225 Springer-Verlag, 1997.
- [78] T. von Eicken, A. Basu, and V. Buch. Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, pages 46–53, February 1995.
- [79] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Symp. on Operating Systems Principle*, pages 40–53, December 1995.
- [80] T. von Eicken and D. E. Culler et al. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. 19th Int. Symp. on Computer Architecture*, pages 256–266, May 1992.
- [81] D. L. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. SPARC International, Inc., 1994.
- [82] 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 Aarchitecture*, pages 24–36, June 1995.