Study on Distributed Shared Memory Managed by Lightweight Hardware

軽量ハードウェアによる分散共有メモリの研究

田 中 清 史

学 位 論 文

Study on Distributed Shared Memory Managed by Lightweight Hardware

軽量ハードウェアによる分散共有メモリの研究

平成 11 年 12 月 博士 (理学) 申請

東京大学大学院理学系研究科情報科学専攻

田中 清史

Abstract

Shared memory provides a general purpose and easy programming environment in parallel/distributed computer systems. Efficient distributed shared memory in a large-scale system requires caching of remote data and management of its consistency.

In this thesis, a method for constructing an efficient system of distributed shared memory is described. Its main feature is the lightweight hardware management, supported by sharing information with the hierarchy in an interconnection network and multicasting/combining of messages. The method obviates the following subsystems and approaches which are expensive and heavyweight in terms of added hardware: directory memory in proportion to the number of processors, separate memory components for the directory, tags and state information, and protocol processors. In the method outlined in this work, the directory scheme, *hierarchical coarse directory*, requires $\log \log n$ memory, where n is the number of processors in the system. A single byte per shared memory block is thus sufficient to cover a massively parallel system and this makes access to the directory cheap in terms of processing. In addition, the directories, tags, and state information are all located in main memory, and a memory controller manages the required protocols. Neither a specialized protocol processor nor separate memory components are thus required.

A prototype parallel computer that implements this distributed shared memory and the combining of messages with lightweight hardware has been developed. When a matrix is raised to the nth power by an 8-processor system, the execution with message combining is 7.7% faster than that without combining. In LU-contig of SPLASH-2 execution using the hardware distributed shared memory, execution on an 8-processor system produced an execution time 81% of that for a single processor. The results show that the method used delivers the advantages of parallelization.

Moreover, the application of the distributed shared memory scheme to a large-scale system has been examined. The system that has been developed, with the hierarchical coarse directory, and multicasting and combining of messages used for cache coherence transactions performs such transactions much faster than existing fullmap schemes, when the number of sharing processors is large.

Acknowledgments

I would like to especially thank professor Kei Hiraki, who is my thesis adviser and gave me the chance to work on the OCHANOMIZ-5, for helpful suggestions on the study and for check and polish of the thesis. I also wish to thank Mr. Takashi Matsumoto for a lot of useful comments and discussions about the system architecture and logical circuit design.

I also thank Mrs. Akiko Shintani who is a librarian of Department of Computer Science. She made it easy to search literature and research related works. Mr. Junpei Niwa, who is a colleague of Hiraki laboratory, encouraged me many times. I wish to thank him. In addition, I am grateful to other members of Hiraki laboratory.

I also thank for financial support by JSPS Research Fellowships for Young Scientists. I would like to thank my family for their support and encouragement.

Last, but certainly not least, I am deeply grateful to OCHANOMIZ-5 for his reliable running for more than three years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Exploitation of parallelism is one way to achieve faster processing, especially in the future when the improvement of sequential processing speed on a computer will be saturated. The model of the parallel and distributed computer, and, in particular, the massively parallel and distributed computer system, is attracting a great deal of attention as a replacement for the single processor system. Moreover, introduction of a parallel and distributed computer is a good choice from the point of view of cost performance since such a system can be made from a number of identical sets of components, which can receive benefit from mass production.

Programming must take parallel execution into consideration to realize the potential high performance of a parallel computer. There are two basic approaches to parallel programming: one depends on a shared memory model, and the other depends on a message-passing model. The shared memory model has the identical view of memory address for every processor, and inter-processor communications are thus done implicitly via memory variables. On the other hand, the message passing model requires that communication functions (send, receive) be explicitly written in a program code. It is desirable that a parallel system provides both programming models in its user interface, since the two styles of parallel programming coexist at present.

When a system implements shared memory at the hardware or OS level, a trap routine on a processor or a dedicated hardware element will directly read or write to memory at the address specified in user space. In a system that implements message passing, a message

invoked by a send function is temporarily inserted into the kernel buffering space in the receiving processing node, and a receive function then takes it out of the buffering space and places the content of the message in user address space. Shared memory is thus a better communication mechanism in terms of the less frequent memory access it requires.

A system with a communication mechanism that is based on shared memory can emulate message passing communications. The send and receive functions can be provided by simply allocating space for a communication buffer in the shared memory space at the destination processing node, and then reading and writing to data and flag variables in the buffer. This thesis is thus focused, from here on the use of shared memory as the system's communication mechanism.

Shared memory can be implemented as either a symmetric multiprocessor system (SMP) or distributed shared memory (DSM). In an SMP, the processors and main memory areas are all connected by a shared bus or a multistage interconnection network (MIN) such as the Omega Network [31, 34] or Benes Network [6], and the physical distance between any processor and any main memory area is the same. A bus-based SMP requires exclusive access to the main memory areas, and the bus can easily be saturated when there are many processors in the system. Therefore, bus connection is unsuitable for large-scale systems. An MIN connection consists of multistage switching elements between processors and memory components, and thus has an inherent fixed latency for every memory access that is determined by the number of processors ($\log_2 n$ in omega network with n processors). It is thus difficult to speed the system up by optimizing the arrangement of data for parallel programming.

Distributed shared memory (DSM) is effective architecture when SMP structure is difficult to apply because of a large number of processor in the system [13]. Many different hardware and software approaches have been proposed for implementing DSM [41, 48]. In DSM, the latency of memory accesses will be not uniform because the main memories are distributed among the processing nodes. The latency will be longer when a processor accesses data in a physically more distant processing node. Reducing the number of remote memory reference in a program by optimizing the arrangement of data is desirable measure because it can reduce actual remote memory access. The measure is, however, not applicable if the program includes irregular data sharing patterns, or if the sharing pattern

cannot be statically found. The use of local caches of remote data is effective because they can minimize degradation of the overall performance by minimizing the latency for remote memory access. In this situation, a cache consistency problem arises. That is, when there is more than one copy of a given memory block in the caches at the various processing nodes and a processor writes new data to one such cache, the other processors will be reading obsolete values unless the new data is reflected in all of the caches or the copies are discarded.

Caching of remote memory means that it is necessary to create local copies of remote data, decide whether data to be accessed is in a cache or not and if it is, whether there are other copies, and in general manage cache coherence, particularly for write request access. In the design of DSM with a cache system, it is a problem to be solved how efficiently the cache management is performed.

1.1 Implementation styles of distributed shared memory

DSM may be implemented in one of three ways: in software scheme, as a hybrid of software and hardware, and in hardware. In a software scheme [15, 52], a processing element creates copies of data in local main memory (caches), decides whether data to be accessed is in the cache or not, and manages cache coherence and communications. Processing overheads for software management of caching are not small because of time-sharing execution of a processor, that is, including overheads caused by context switches, system calls and trap handling, but special hardware is not required.

Shared virtual memory originated by IVY [39, 40] uses hardware page management mechanisms (i.e., a memory management unit (MMU) and a translation lookaside buffer (TLB) in each processing element), and uses page fault trap to invoke remote memory accesses and cache coherence processing. That is, a page fault trap is triggered when initial reading of a given remote data appears, and the trap routine generates a read-only page entry in a page table and local copy of the page. When a processor is going to write data to the page, an access violation trap occurs and cache coherence is processed.

Although this scheme reduces software overheads when a processor reads data after the

generation of the page entry and the local copy, software execution overheads for maintaining cache coherence over write operations remain, and the invocation of a remote request takes up more time because the processors must execute a trap or system call routine. Moreover, shared data is cached in page units, which causes false sharing leading to performance degradation due to the excessive use of network resources. Implementation of DSM that fully or partially requires software management generally carries not only software overheads, but also bottlenecks caused by the weak performance of the network. It is thus difficult to achieve high performance in a large-scale parallel system [14]. Therefore, optimization to minimize communication is indispensable.

A hardware schemes allows all software overheads on a processing element to be reduced by providing hardware mechanisms that are dedicated to cache management [18, 32, 38]. Simultaneous processing by a processor and communication/management hardware can deliver high performance DSM. The costs of the special DSM hardware must be considered, however, since software implementation don't require such special hardware and can be implemented on standard commercial PC or workstation clusters. Under existing conditions where the performance of software schemes based on one of several release consistency models [2, 7, 10, 21, 30] or an optimizing compiler [23, 46, 51] is improving, a hardware system loses its merit if it cannot provide a performance improvement that is proportional to the cost of the special hardware.

One factor which raises hardware costs is the use of a protocol processor in existing hardware DSM systems. Although a dedicated protocol processor is useful for implementing various coherence protocols, it carries an overhead in terms of the execution of instructions to invoke processes for memory or cache management, that are implemented in hardware alone in bus-based SMP system. Moreover, a protocol processor will cost more than a processing element since few will be produced, and its sophisticated structure has the disadvantage that requires a long time to develop. When commercial processors are used in a parallel computer system, the design of the protocol processor and the other components depends largely on the interface of the processors. Hardware systems, therefore, appear after the processors. As a result, expensive hardware systems don't have a long life in terms of advanced and latest processing power. Both the hardware costs and the long development period are serious problems from the point of view of the fast evolution of a commercial processor.

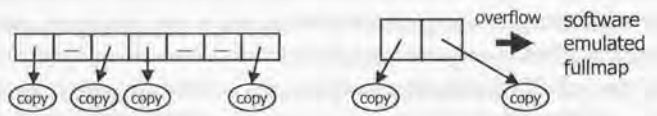
There are examples of software DSM systems that achieve a scalable performance improvement by optimizing communication/synchronization, although the scale of the systems is not large and the improvement varies according to the nature of the application [46]. A software DSM or hybrid scheme of software and hardware is, therefore, one solution for a parallel and distributed computer with a fast interconnection network that avoids the high costs of special components for hardware DSM or a complicated consistency model in hardware. A more efficient large-scale system will, however, be established with a cost-efficient lightweight hardware DSM mechanism for applications that require fine-grain communications/synchronizations [14]. This thesis aims at providing a method of constructing an efficient lightweight, low-cost hardware DSM system.

1.2 Directory schemes of hardware DSM systems

In a DSM system based on CC-NUMA (Cache Coherent Non-Uniform Memory Access) system, sharing information must be managed. Shared memory space is divided into blocks. Sharing information indicates whether or not the block is shared and whether it has been updated or not, and includes a directory that identifies the places where processors are holding a copy of the block.

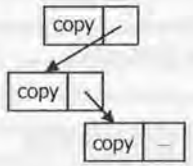
The memory space taken up by the directories increases with the scale of the system. Therefore, the structure and size of directories will affect the hardware costs and performance. Directory schemes are classified into two types: one type completely identifies the processors that hold a copy of a block, and the other identifies the processors incompletely. The former has a problem in that it requires a large amount of memory for directory storage when there are many processors in a system or a problem in that an access overhead to a directory is large when the directory size is larger than the bit width of a memory component or the structure of the directory is based on a linked list. Two general schemes for incomplete identification: one in which the number of processors that can share a block is limited and the other in which the processors that share a block are identified roughly, that is, a group that includes the processors which share a block is indicated. Both take up relatively less memory than complete identification but there are still significant overheads

cased by broadcasting or multicasting of coherence messages when many processors share a block.

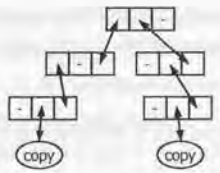


(a) fullmap directory

(b) LimitLESS directory



(c) chained directory



(d) hierarchical bit-map directory

Figure 1.1: Directory structures with complete sharing information.

The fullmap directory [11], LimitLESS directory [12], chained directory [24, 62] and hierarchical bit-map directory [18] hold complete information on sharing (Figure 1.1). The fullmap directory system assigns 1 bit to each processing node in a system to indicate whether the processing node holds a copy of the relevant memory block. Since this scheme requires memory in proportion to the number of processing nodes in the DSM systems, it is not suitable for large-scale systems. Multiple access to the directory memory is also required if the number of processing nodes exceeds the width of a single access, for example, 32 or 64 bits.

A LimitLESS directory system places a hardware limitation on the number of processors that can share a block in order to reduce the memory requirement. The directory has the limited number of pointers to point to processors with the block copies. When the number of copies held exceeds the limit, a protocol processor or processing element emulates a

Table 1.1: Size of directories with complete sharing information.

Directory	Size	Notes
fullmap	n	n processors in a system
LimitLESS	$limit \times a\ pointer\ size$	overflow invokes software exec.
chained	$a\ pointer\ size$	traces a linked list
hierarchical bit-map	$\sum_{k=1}^{m}(n+1)^k$	n-ary tree of height m

fullmap scheme. Although this directory requires less memory than fullmap when the system has many processors, execution of the software creates a large processing overhead. A chained directory is small because it is a pointer, the structure, however, results in long access latencies caused by sequential access to linked directories. Therefore, it is necessary to keep the number of sharing processors low by employing an invalidation protocol when a chained directory is used.

In the hierarchical bit-map directory used in COMA (Cache-Only Memory Architecture) [18], sharing information for a memory block is distributed, that is, fullmap information is partitioned into sub-bitmaps among hierarchical levels in the tree network as shown in Figure 1.1(d). The directory size, therefore, increases with the scale of the system. It requires $\sum_{k=1}^{m}(n+1)^k$ bits for each memory block in an n-ary tree of height m. More memory is thus required than for a fullmap directory. The scheme inherently increases communication latency, since it requires access to a part of the directory at every level of the hierarchy. Accordingly, the directory should be stored in high speed memory to prevent the high network latency from degrading system performance.

Consequently, when directories which have complete information on the locations of block copies are used in a large-scale system, the problems are that the directory is large, that access latency is high, or that protocol processing by a protocol processor/processing element induces large overheads for software execution. Table 1.1 shows the directory size of these schemes.

On the other hand, the limited directory [4], superset scheme [4], coarse vector scheme [17] and pseudo-fullmap directory [42] obtain a size that is not proportional to the number

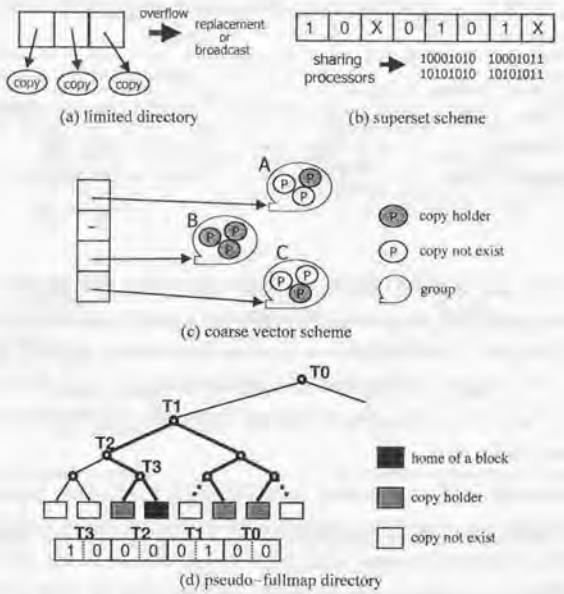


Figure 1.2: Directory structures with incomplete sharing information.

of processors by using incomplete information on sharing (Figure 1.2). The limited directory (Figure 1.2(a)) uses a limited number of pointers to processors with cached copies, and the directory size, therefore, does not increase in proportion to the system scale. However, multiple access to directory memory may be required since the bit width of a limited directory cannot be said to be small. For example, when there are 1000 processors and the limit number is four, the directory requires forty bits, which imposes two-step access to directory memory through 32-bit wide memory bus. Moreover, when the number of copies reaches the limit, a next generation of a copy forces cache replacement by victimizing an existing copy or broadcasting of a coherence message to all processors, which leads to a

lot of extra communication. It is thus important to keep the sharing number low by using an invalidation protocol.

In the superset scheme (Figure 1.2(b)), the directory is represented by a composite pointer that is made out of two pointers. Each field of the composite pointer is in one of three states: 0, 1, or X, and is thus composed of two bits. X means "both" 0 and 1. When a processor joins in sharing members, that is, generates a copy of the memory block, the processor number is compared with the pointer, and bit fields where the two patterns differ are set to X. This scheme makes a superset of processors that have a copy of the block, and can point to a maximum of 2^k copies by using two pointers with a length of k. During a coherence transaction, coherence messages are sent to all processors that correspond to all bit patterns obtained by substituting the X bits with all combinations of 0s and 1s. Hence, a processor which does not hold a copy of the block might receive the coherence messages. This leads to redundant communications. For example, when two processors whose numbers are "10101011" and "10001010" share a memory block, the composite pointer is "10X0101X" and other "10101010" and "10001011" processors receive the messages, as shown in the figure.

In a coarse vector scheme (Figure 1.2(c)), processors are grouped and a bit is assigned to each group. Groups are then identified by the same way as fullmap scheme. The number of processors that the scheme can cover is k times the bit width of the directory where k is a size of each group. Here, redundant coherence messages might be sent as in the superset scheme since all processors in any group in which one or more processors have a block copy are regarded as a copy holder. In the figure, all white processors in group A and B receive the messages.

The pseudo-fullmap directory (Figure 1.2(d)) reduces the memory required by using bit-maps that correspond to each level of the hierarchy in a tree interconnection network. There are three schemes in the pseudo-fullmap directory, LPRA (Local Precise Remote Approximate), SM (Single Map), and LARP (Local Approximate Remote Precise). LPRA scheme specifies near processors precisely and more distant processors roughly. LARP is a complementary scheme of LPRA, that is, specifies distant processors precisely and rearer processors roughly. In SM scheme, all network nodes at a level use a unique bit-map. Here, directories are only accessed at the respective home processors because each directory can

only be maintained at its home processor. On the other hand, a hierarchical bit-map directory requires that the directory be looked up at each level of the interconnection hierarchy. Figure 1.2(d) is an example of LPRA scheme. A pseudo-fullmap directory scheme takes up the amount of memory proportional to $\log n$ when the number of processors is n, and the incompleteness of sharing information leads to redundant communications in cache coherence transactions (on dotted lines in the figure).

Table 1.2 shows the size for these incomplete directory schemes.

Table 1.2: Size of directories with incomplete sharing information.

Directory	Size	Notes
limited	$limit \times a\ pointer\ size$	overflow invokes hardware broadcast or replacement
superset scheme	$2 \times \log n$	n processors in a system
coarse vector scheme	n/k	k is the size of a group
pseudo-fullmap	$n \times m$	n-ary tree of height m

1.3 Outline of lightweight hardware approach

Figure 1.3 shows an example of a typical hardware DSM system. The problem is to find a way of implementing a DSM system at low cost, with simple hardware, and over a short period.

We propose that a lightweight hardware DSM should have the following characteristics.

- Directory structure occupying a small amount of memory and requiring the low access overhead
- Elimination of the need for separate high speed memory components for the directory, tags, and state information
- No dedicated protocol processor

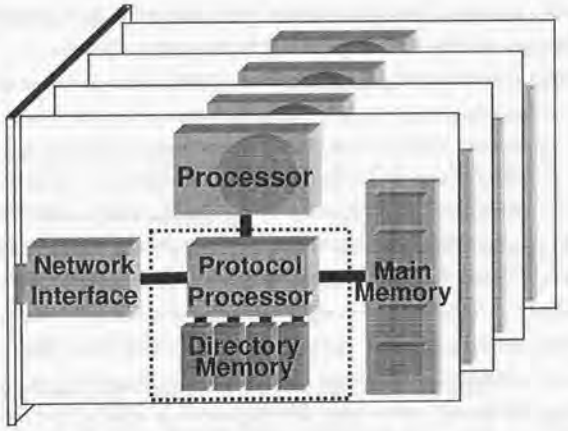


Figure 1.3: An example of a typical hardware DSM system.

Schemes with complete identification such as the fullmap directory that require memory in proportion to the number of processors are unsuitable, in terms of hardware costs, for use in a large-scale system. It is therefore necessary to find a directory scheme that uses incomplete information to occupy a small amount of memory, has a low access latency, and needs no software execution or redundant broadcasting.

The major factor in the costliness of hardware DSM systems is the use of high speed SRAM to save directories, and tags and state information. The performance improvement that results from the use of high speed memory is affected by the frequency of reference to shared variables in a program and the frequency of cache-miss operations. If the frequency is low, the use of costly SRAM is not cost-effective.

The cost of dedicated protocol processor hardware is high, too, and there is a problem in that generally its development term can only be subsequent to the appearance of the commercial general-purpose processor on which the system is based. Delays caused by the execution of its software can slow communications and increase the latency of remote

memory access. In terms of both cost and performance, therefore, it is important to find a method of replacing the protocol processor with lightweight hardware.

In this thesis, we describe a lightweight hardware DSM scheme that does, in fact, eliminate the drawbacks to hardware costs and performance outlined above. We employ the distance between processors with copies of a block as the directory information, and place the directories, tags, and state information in main memory so that separate high speed SRAMs are not needed. Moreover, cache management is implemented with a small overhead and at low cost by using simple logic hardware. Therefore, neither a dedicated protocol processor nor execution of a special program is needed. The simple directory structure and lack of a protocol processor do, however, lead to a certain amount of redundant inter-processor messages. Hierarchical multicasting and combining techniques by an interconnection network dynamically cut redundant messages and keep the system efficient. In order to construct CC-NUMA system, the only extra components that need to be added to the basic distributed memory system are a small amount of simple hardwired logic in memory controllers and the switching nodes in the interconnection network.

1.4 Contributions

This thesis makes the following contributions.

1. Proposal of a directory scheme with a size that is greatly reduced by using hierarchical distance

   This directory scheme is used in a system with a hierarchically structured interconnection network that inherently includes tree structure. The subtree which includes all processors with a copy of the block is considered to be a 'sharing group'. The benefit is that the size of the directory is smaller than those of any other existing directory scheme.

2. Achievement of lightweight and low-cost cache management

   The use of a dedicated protocol processor or high speed SRAM to hold directories, tags and state information is a primary factor in the high cost of hardware in hardware-based DSM systems. The system proposed places sharing information in main memories and makes the hardwired logic in a memory controller manage it. The method establishes a way to reduce hardware costs and to use lightweight hardware for cache protocol control.

3. Proposal of efficient communication by using dynamic multicasting and combining

   This directory scheme causes increased inter-processor communication because the sharing information is incomplete. For cache coherence transactions, dynamic multicasting and combining mechanisms of messages by the hierarchical interconnection network are used in order to decrease the number of redundant messages and to guarantee efficient communications.

4. Implementation and verification of lightweight hardware DSM

   A prototype parallel computer that incorporates these principles has been constructed. This DSM methodology has also been shown to deliver the advantages of parallelization by running several programs as tests.

5. Consideration of the method's qualities on a large-scale system

   Measured time values for the prototype machine are used as the basis for an analysis that shows its applicability to large-scale systems. The system is also compared with an other DSM system.

1.5 Thesis organization

Chapter 2 describes the new directory scheme which solves the problem with the size of directories in existing systems, and shows how the directory is generated and where the information of directories, cache tags and state is located. Chapter 3 describes the methods used to reduce redundant messages for coherence transactions, and explains generalized combining that supports the DSM system. Chapter 4 shows how the directory structure is applied to cache protocols, and outlines the network transaction used in the hardware DSM system. In Chapter 5, the architecture of the prototype parallel computer and implementation of the DSM mechanism are described. In Chapter 6 gives results for actual programs

running on the prototype, and the system's effectiveness is discussed in terms of the results. Chapter 7 is a discussion and analysis of methods that are effective for large-scale systems. Related works about hardware DSM systems, message combining schemes, and reduction schemes of coherence messages are described in Chapter 8. Chapter 9 concludes the thesis.

Chapter 2

Hierarchical Coarse Directory

2.1 Overview

The first step in constructing a DSM system with lightweight hardware is the establishment of a directory system with small directories and low access latency. A method that completely identifies all processors that have cached copies of a memory block unavoidably takes up memory space that is at least proportional to the scale of the system. It is thus necessary to set a limitation on the number of processors with cached copies such as a limited directory or to represent the sharing information in some way that does not require all such processors to be fully and exactly specified.

In this thesis, we propose a "hierarchical coarse directory" scheme. This scheme greatly reduces the size of the directories by using a hierarchical distance within the interconnection network. The size for n processing elements is $O(\log \log n)$. In order to minimize the need for extra hardware, directories, tags, and records of the state of copies in caches are located in main memories instead of in dedicated memories, and are managed by a memory controller. That is, a memory controller handles sharing information in the main memory according to memory access requests from processing elements or from outside the processing node. The lightweight hardware control obviates the need for a protocol processor or execution of software for cache management.

The simplicity of the directory structure and the method of management might increase the volume of network packets during transactions to maintain consistency. Lightweight

hardware extensions, multicasting and combining of massages by the interconnection network, dynamically decrease the number of network packets and achieve efficient transactions by preventing sequential processing of network messages. The extensions are described in Chapter 3.

2.2 Hierarchical coarse directory

2.2.1 The definition

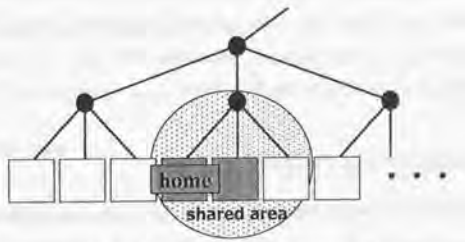
The hierarchical coarse directory is a way of keeping the directories small[57, 58, 59, 60]. We assume that a tree structure can be physically embedded in the interconnection network. For example, a hypercube [65, 67], the fat-tree [36] in CM-5 [37] and the RDT network [26] in JUMP-1 [1] include a tree structure. A home processor[1] is statically assigned to each memory block. A home processor manages the directory which records data-sharing information of the block by using the "maximum shared distance". Here, the "maximum shared distance" is half of the number of hops between the home processor and the most distant processor with a cached copy of the block. In other words, the distance is the height of the minimum subtree which includes all processors that have a copy of the block.

Figure 2.1 is an illustration of the hierarchical coarse directory structure. The gray leaves in the figure represent processors which have a copy of the block ("block holders"), and the area of the mesh indicates the "shared area" in which processors may have a copy of the block. In Figure 2.1(a), the home processor and one of its next-door neighbors in the network hierarchy hold a copy of the block. Therefore, the maximum shared distance in this example is one. The maximum shared distance depicted in Figure 2.1(b) is two since the shared area is in a subtree of the network with a height of 2. (In the actual hardware implementation, the height of the tree is decreased by one in order to simplify the calculating hardware. The distance depicted in Figure 2.1(a), for example, is 0, and the distance in Figure 2.1(b) is 1.)

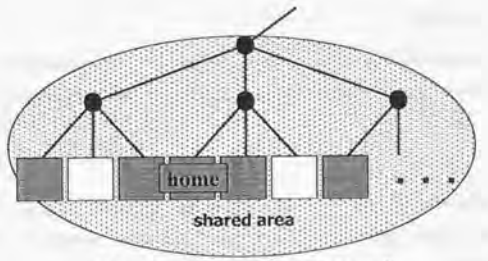
The shared area may include processors that don't have a copy of the block (in the figure, the white leaves in the shared area). To the home processor, these processors also seem to have a copy. The simplicity of the directory representation causes an inaccuracy, an overestimation of the number of block holders. The overestimation does not, however, influence cache consistency. When a processor that does not have a copy receives a coherence message from the home processor, it has only to return a dummy acknowledgement message. The procedure maintains consistency.

The hierarchical coarse directory is $\log_2 \log_k n$ wide where n is the number of processing elements and the network has a k-ary tree structure. This is smaller than any existing

[1] When a processing node is a multiprocessor cluster with a shared bus, the term "processor" is replaced by "cluster".



(a) maximum shared distance = 1



(b) maximum shared distance = 2

Figure 2.1: Hierarchical coarse directory.

directory scheme. For example, a four-bit directory for each memory block is sufficient to cover directory information for a massively parallel system that contains more than 64,000 processors connected by a binary network. Moreover, this directory system accomplishes the reduction of required memory without limitations on the number of copies.

2.2.2 Calculation of the distance

In the hierarchical coarse directory system, a home processor must calculate the hierarchical distance between itself and a requesting processor. The following is the procedure for calculating the hierarchical distance.

1. **Processor number**

   Firstly, a processor number is statically assigned to each processor. To deal with the hierarchy, the processor number A is expressed as the k-ary number:

   $$A = (A_{N-1}A_{N-2}\cdots A_1A_0)_k \qquad (A_i \in \{0, 1, ..., k-1\})$$

   where k indicates the order of branching in the tree structure, and N is the height of the network, that is, there are k^N processors in all. The above A_i is expressed as the binary number:

   $$A_i = (a_{in-1}a_{in-2}\cdots a_{i1}a_{i0})_2 \qquad (a_{ij} \in \{0, 1\}, n = \lceil \log_2 k \rceil)$$

   for convenience of use with logical operations/circuits later.

2. **Expression with exclusive OR**

   Next, it is necessary to derive the expression with exclusive OR of a pair of two processor numbers A, B. The exclusive OR value C is the binary number:

   $$C = (C_{N-1}C_{N-2}\cdots C_1C_0)_2 \qquad (C_i \in \{0, 1\}).$$

   The above C_i is the value of the disjunction of exclusive ORs:

   $$C_i = (a_{i0} \oplus b_{i0}) \vee (a_{i1} \oplus b_{i1}) \vee \cdots \vee (a_{i(n-1)} \oplus b_{i(n-1)}).$$

3. **Result**

   The final distance is calculated as:

   $$D = \lfloor \log_2 C \rfloor.$$

Software implementation

Here, we consider calculating a hierarchical distance between two processor numbers in software on a commercial processor, without special hardware support. The network is assumed to be a 4-ary with a height of eight. An example of the algorithm written in C language is shown as follows.

```
unsigned long distance(unsigned long p1,
                       unsigned long p2)
{
  unsigned long exclusive;
  int i;

  exclusive = p1 ^ p2;

  for (i=7; i>=0; i--,exclusive <<= 2) {
    if ((exclusive & 0xC000) != 0)
      return i;
  }
  return 0;
}
```

The function "distance" receives two processor numbers (p1, p2), generates the value of exclusive OR of p1 and p2, detects the hierarchy level that indicates the hierarchical distance by using a loop index, and returns the value of the index as a result.

The compilation[2] of this function generates the following SPARC [54] assembly language code.

[2]The function compiled by gcc version 2.7.2.3 with optimization "-O4".

```
_distance:
        xor %o0,%o1,%g2
        mov 7,%o0
        sethi %hi(49152),%g3
L18:
        andcc %g2,%g3,%g0
        bne L21
        nop
        addcc %o0,-1,%o0
        bpos L18
        sll %g2,2,%g2
        mov 0,%o0
L21:
        retl
        nop
```

The calculation contains at least eight instructions. In the worst case, 53 instructions needs to be executed for the calculation.

Although a faster procedure is implemented by preparing a table that contains result values in a memory, execution of three instructions at least is indispensable and enough memory space is needed.

Hardware implementation

A simple logical circuit can implement the calculation of a distance. As an example, we illustrate the circuit for an interconnection network that is a 4-ary tree with a height of eight (the same condition as the above software implementation). This network covers $4^8 = 65,536$ processors. Processors A and B are assigned the numbers:

$$A = A_7 A_6 \cdots A_0$$

$$B = B_7 B_6 \cdots B_0.$$

Here, A_i and B_i are the following two-bit binary numbers:

$$A_i = (a_{i1} a_{i0})_2 \qquad (a_{i1}, a_{i0} \in \{0, 1\})$$

$$B_i = (b_{i1} b_{i0})_2 \qquad (b_{i1}, b_{i0} \in \{0, 1\}).$$

That is, A and B can be expressed by the bit sequences:

$$A = a_{71} a_{70} a_{61} a_{60} \cdots a_{01} a_{00}$$

$$B = b_{71} b_{70} b_{61} b_{60} \cdots b_{01} b_{00}.$$

Figure 2.2 illustrates an example of the logical circuit that can generate the hierarchical distance for two processor numbers. A 3-bit binary number $(C_2 C_1 C_0)$ is generated as the result. The simple circuit shown can cope with a system containing over 64,000 processors. In the design of an integrated circuit, the few stages of logic will have little influence on the size of the circuit or the critical path in signal propagation delay. The simple hardware can generate the hierarchical distance more quickly and with a more lightweight processing load than software on a protocol processor or a processing element.

2.3 Directory storage

In order to perform cache management quickly, directories, tags and state information are stored in fast SRAMs in several directory-based hardware DSM systems. When a processor issues a memory-read request, for example, the management mechanism starts to access data in main memories(DRAMs) or caches of remote data at the same time as it is reading the sharing information in the SRAMs. This parallel processing makes a rapid response to the requester.

The use of fast SRAMs, or any other kind of dedicated memory for the caches, however, is a primary factor in increasing the hardware costs. The architecture described in this thesis holds all directories, tags, state information, and caches of remote data in the main memories. A memory controller takes charge of both the management of the sharing information and data transfer systems. The implementation costs and the number of pins on the chip are affected by equipping a main memory with an additional access bank or port

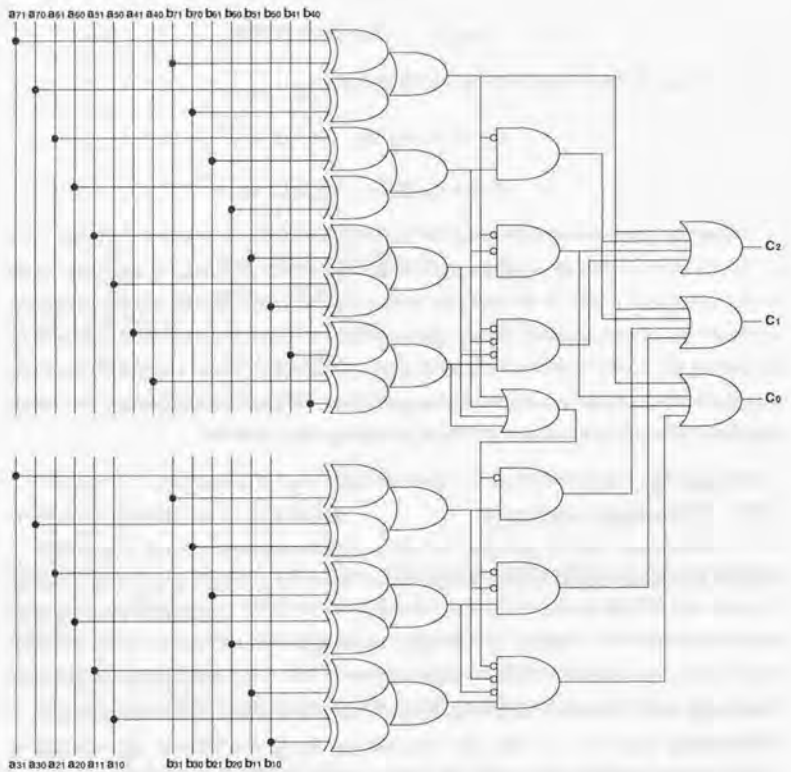


Figure 2.2: Example of the circuit that generates the hierarchical distance.

for the sharing information. Therefore, we discuss the method of having a basic memory controller successively access the sharing information and target data, and compare it with the method of dedicated memory for the directories, tags and state information.

When a processor accesses a variable in shared memory space:

1. it hits its own internal caches; or
2. it hits the caches for remote data in its processing node; or
3. it misses all caches.

In case 1, there is no overhead since the access is processed entirely in the processor. In case 3, the difference of the latency for access to the local main memory either once or twice has a little influence because of the necessity of a transaction with elements outside the processing node. In case 2, the access latency depends on whether or not there is dedicated memory, that is, whether access to the main memory is once or twice.

If there are two steps, the directory, tags and state information are read first and the target data is then accessed. The data width for the first step will take no more than a single access (64 bits, for example) to DRAM because the directory is small (three or four bits). The total latency for both steps, therefore, is no more than twice that for access to the target data, which might consist of multiple double-words[3].

It is difficult to accurately estimate whether each memory reference in a program will hit the cache or not. Here, we assume that all references to shared memory are performed as in case 2, where the influence caused by the difference between one- and two-step access is remarkable. All references to local variables are also assumed to be access to a main memory, and the latency is equivalent to that for one-step access. Let the latency for a single access[4] to the target data be t, and the ratio of references to shared variables to all memory references in a program be $\alpha \quad (0 \leq \alpha \leq 1)$. The average latencies for one- and two-step access, T_1 and T_2, are respectively:

$$T_1 = (1 - \alpha)t + \alpha t = t$$

[3] In general, the size of data equals that of a cache line.

[4] A processor immediately restarts execution after it receives the first word. The latency for the first word in the target memory block is thus taken up here.

$$T_2 = (1-\alpha)t + 2\alpha t = (1+\alpha)t.$$

If the difference $T_2 - T_1 = \alpha t$ is trivial, the overhead of the two-step access method can be said to be small. Usually, t is about four cycles of the clock frequency on the memory bus. α has been estimated by using actual programs. A compiler that detects access to shared variables [23, 46] was applied to SPLASH-2 programs[64]. Although dynamic factors such as loop counts might influence the execution, the results can be regarded as a rough indicator. Table 2.1 shows the name of each program, reference counts of local variables, reference counts of shared variables and the value of α. From the results, the difference αt is from 0.0247 to 0.1891 cycles, which is not large and does not indicate a significantly degraded performance for the two-step method. Therefore, the directories, tags and state information are located in the main memories which the memory controller accesses twice when a request is toward shared variables.

Table 2.1: The number of times local and shared memory are accessed.

Program	Local variable access	Shared variable access	α
LU-Contig	7,667	136	0.017429
Radix	3,433	106	0.029952
FFT	6,647	152	0.022356
Water-NS	15,415	255	0.016273
Water-SP	26,654	389	0.014384
Raytrace	129,323	803	0.006171
Barnes	53,518	479	0.008871
Ocean-Noncontig	16,642	826	0.047286

Chapter 3

Lightweight Extension

The simplicity of the hierarchical coarse directory structure and the absence of a protocol processor increase the volume of network communications. This chapter describes a method for reducing the number of messages by dynamic mechanisms in the interconnection network.

3.1 Hierarchical multicasting and combining

Identical messages must frequently be transported to many or all processing elements during consistency processing such as invalidation or update. The transport performance can be improved by utilizing hierarchical multicasting. For example, when an invalidation is processed for a shared memory block, the home processor assigned in advance to the memory block issues only a single invalidation message. Each switching node in the network that has received the message multicasts it in all the directions that lead to a node within the shared area. Figure 3.1 shows the multicasting operation when the maximum shared distance is 2.

The directory scheme affects the multicasting method. In implementation of multicasting, a directory with complete information, such as a fullmap directory requires that a network packets for multicasting include a group of destination processor numbers or the directory itself. The former approach makes the multicasting method impractical when the number of destination processors is large. For example, when the system has 1,000

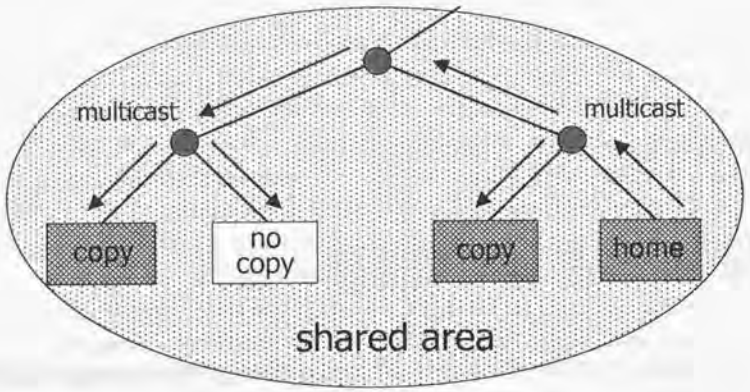


Figure 3.1: Hierarchical multicasting.

processors that require 10 bits of a processor number to identify any processor, and 200 processors share a memory block, total 2,000 bits (250 bytes) must be included in a packet header, which is not small. The latter requires the same number of bits as of all processors, which may be far from insignificant, and makes a switch perform an elaborate routing.

On the other hand, the hierarchical coarse directory system is well-suited to multicasting. It is only necessary for network packets to include the directory information, that is, the maximum shared distance. The network switching nodes multicast by comparing the maximum shared distance with their own hierarchical location.

When more than one message is sent to a single processor, the messages can be combined into a single message at any level of the network hierarchy. This reduces the need for a series of processes at the destination processor. For example, every processor that has received an invalidation message returns an acknowledgement message which indicates the completion of invalidation processing within the processing node, even if it does not have a copy of the indicated block. All the acknowledgement messages are directed to the home processor. Each switching node forwards an acknowledgement message after it confirms

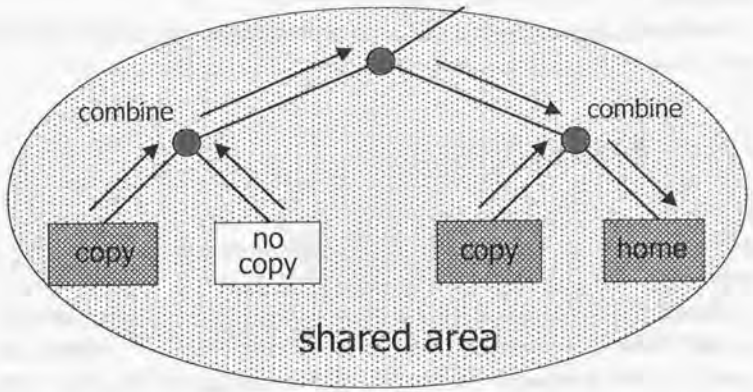


Figure 3.2: Hierarchical combining.

the arrival of all acknowledgement messages from all branches along which it sent the invalidation message. The process is shown in Figure 3.2.

The hierarchical coarse directory is also suited to the combining of messages. The number of messages to be combined at a switching node depends on whether the node is or is not a root of the shared subtree. When the switching node is the root, the number is $k-1$ where the network is k-ary. When it is not the root, the number is k. On the other hand, a switch must record the number of messages to be combined in some way for a fullmap directory system.

The hierarchical multicasting and combining schemes require no serialized processing at the home processor. It takes only one round-trip latency between the home processor and the most distant processor in the shared area for the home processor to complete the coherence transaction, regardless of the number of processors with copies of the data. The combination of the reduced size of the hierarchical coarse directory and the hierarchical multicasting and combining schemes make it possible actually to employ not only an invalidate protocol but also an update protocol even when there are a large number of sharing

processors.

This combining of acknowledgement messages is accomplished by using the generalized combining [61] as described in the next section.

3.2 Generalized combining

Factors such as the reduced size of the directory, absence of a protocol processor, and management per cache line increase the amount of inter-processor communication, and increase the network traffic and cause hotspot contentions [47]. Request combining [16] is effective for alleviating this contention. It dynamically combines more than one request destined for the same destination into a single request and thus reduces the amount of network communications. The technique in [16], however, requires a large amount of hardware within each network switching node since half the number of comparators as that of entries in a combining queue should be embedded [47]. Moreover, it cannot work sufficiently since it expects the accidental events that requests meet each other at a switching node. In other words, combining does not occur if a request arrives at the switching node after the preceding request has departed from the node.

We propose a flexible combining technique called *generalized combining* by extending the existing combining techniques. The combining of acknowledgement messages is one application. Generalized combining consists of three generalizations: generalization of arrival requirement, generalization of processing function, and generalization of matching requirement.

3.2.1 Generalization of arrival requirement

The matching points for combining are grouped into four categories according to the time a message arrives at a switching node. The four categories are in the past, at the present, during the delayed time, and in the future. Matching in the past means that matching has been already done when a message arrives at the node. Matching at the present means that matching happens at the same time as the arrival of a message. Matching during the delayed time means that a message matches with other subsequent messages during its

stay at the node; existing combinings perform only this type of matching. Matching in the future means that matching will be completed after a message leaves the node. Figure 3.3 illustrates these matching points for combining.

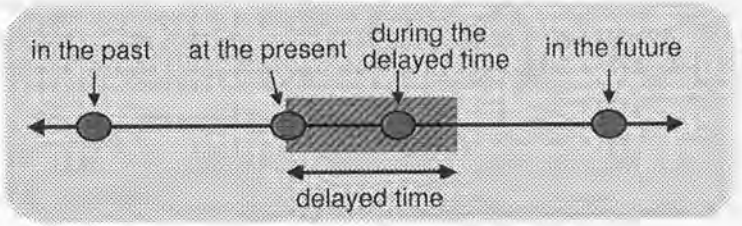


Figure 3.3: Timeframe of matching points for combining.

We enable the delayed time to be set up to any length from zero to eternity. In the case of zero, a message passes through a switching node without waiting for other subsequent messages. This message is not combined during the delayed time, although it may be combined in the past, at the present, or in the future. Setting the delayed time to eternity ensures that a message stays at the node until all of the messages that should be combined with each other arrive there. The combining of acknowledgement messages is accomplished by a delayed time of eternity. Other time values between zero and eternity make a message depart from the node after waiting there for the specified time.

This generalization eliminates the restriction that matching chances occur only while a message is staying at a node, that is, only while a message is passing through a combining queue [16]. As a result, it can cope with time gaps between the arrivals of messages and can increase the rate of successful matching.

3.2.2 Generalization of processing function

In generalized combining, a switching node supports a variety of functions with messages. This generalization makes it possible not only to combine read or Fetch&Ops requests for

the identical memory address, but also perform other operations such as the combining of acknowledgement messages, barrier synchronization, and logical/numerical reduction using the interconnection network as a systolic array like a sorting network [5] or a reduction network (bitwise logical OR/XOR, signed maximum/addition, and unsigned addition) of CM-5[37]. Although each switching node must contain all the function units to make this generalization possible, we select only the functions needed for realizing a lightweight hardware DSM. These selected functions make use of a common combining unit in a switching node.

3.2.3 Generalization of matching requirement

The matching requirement is generalized in two ways.

a) Combining of any number of messages

The number of combinable messages is any number from zero to infinity. The number is specified for every combining. A combined message is forwarded only after the specified number of messages have arrived. The departure time of a combined message thus depends on the specified number; however, when the delayed time expires, the forwarding is done even if some of the messages have not arrived. To ensure that all messages are combined, the delayed time is set to eternity. Figure 3.4 shows the situation where a combined message is forwarded (ignited) when the number of combinings is set to 3.

Figure 3.4: Ignition timing when the number of combining is three.

b) Any matching key

Any matching key can be selected as a matching requirement. In addition, we allow a message to have any number of matching keys. This generalization makes it possible to use as a key not only memory address like existing combinings, but also, for example, processor ID for barrier synchronization, some key which identifies distinctly partners of combining, or any other intentional values. In our DSM system, we use network addresses as keys for memory access requests and processor IDs as keys for barrier synchronization.

3.2.4 Features

An interconnection network which realizes generalized combining has the following features:

- **two cases of ignition**
  Ignition timing is determined by either the delayed time or the number of combining. If one of them is satisfied, a message will depart from the switching node, even though the other is not.

- **existence of incomplete matching**
  If no succeeding combinable messages have arrived while the preceding message is spending the delayed time, the combining is not completed. Incomplete combining implies a combining of smaller number of messages than the specified number. For example, occurrence of a combining between read requests for a memory block is not accurately forecasted and therefore, incomplete matching can occur.

- **overflow of messages**
  Generalized combining assumes that each switching node has buffering space or memory enough to keep arrival messages. However, it is difficult to statically estimate the number of messages. When the specified delayed time is much long, or the number of combining is much large, there is a possibility that the number of temporary messages at a switching node overflows the buffer.

The network system must deal with the buffer overflow. Since large-scale parallel systems are configured in NUMA, a processor nearest to the overflowing node exists. When a switching node detects the overflow of its own storage space, the node can charge the nearest processor with appropriate management by interrupting.

Chapter 4

Cache Coherence Design Methodology

4.1 Cache coherence protocols

Two cache consistency protocols, an invalidate protocol and an update protocol, are used in implementation of a cache coherent shared memory system. In the following, we assume that shared memory space is partitioned into blocks and a home processor is statically assigned to each block and only the home processor holds the directory of the block.

In the invalidate protocol, a processor can write to the memory block only after all other copies of the block have been invalidated when a memory block is shared. The processor that has written to the block becomes the owner of the block. It is then necessary for a home processor to be able to specify the owner. The home processor forwards a subsequent read request to the owner. The owner processor then supplies the latest values in response.

On the other hand, in the update protocol, when a processor writes to a memory block copy, all other copies of the block are updated if the block is shared. In a general DSM system based on the update protocol, a home processor that has the directory of the block always holds the latest values, and can undertake the service to read requests.

Here, we describe how to introduce the hierarchical coarse directory into the two protocols outlined above. For simplicity of cache management, the home processor holds the valid version of the block whenever the block is shared.

4.2 Application to cache system

The following descriptions are examples of the way that the hierarchical coarse directory can be applied to an invalidate or update protocols. In practice, the process differs according to the method used for the processor cache (write-through or write-back, for example) and where remote blocks are cached (in the main memory or in the special SRAM, for example). Here, we concentrate on management in processing nodes. Processing in the interconnection network, such as multicasting and combining, is not taken into consideration. The next section deals with the network communications.

Application to invalidate protocol

Preparation

A home processor of a block holds the maximum shared distance as the directory information and a valid/invalid bit, a shared/private bit and an owner identifier field as its state information. Other processors hold an address tag, a valid/invalid bit and a shared/private bit for each cached block.

Read request

When a processor needs to read a block of which it isn't the home, it locally searches a tag and state information of the block, and it then reads the block copy in the local cache if there exists the valid copy. Otherwise, it issues a read request to the home processor. When a home processor receives a read request from another processor, it returns the data if the target block is valid there, or, if the block is invalid, it returns the latest data after it gets them from the owner processor. Then it sets the state of the block to shared, calculates the hierarchical distance between itself and the requesting processor, and updates the maximum shared distance if necessary, that is, when the new hierarchical distance is greater than the existing maximum shared distance.

Write request

Before a processor writes to a copy of a block whose state is shared in its local cache, it issues an invalidate request to the block's home processor in order to acquire

ownership of the block. After the home processor receives the invalidate request, it sends an invalidate message to all processors within the subtree whose height is the maximum shared distance. A processors other than the originally requesting processor that has received an invalidate message performs the invalidation of its block copy, that is, sets the valid/invalid bit to invalid, when a valid copy exists in its caches, and sends an acknowledgement message to the block's home. A dummy acknowledgement message is returned when the processor does not have a valid copy or the processor is the original requester. When the home processor has received acknowledgement or dummy acknowledgement messages from all processors in the shared area, it updates the owner identifier, sets the valid/invalid bit to invalid, and sends an acknowledgement message that means the completion of the invalidation in the shared area to the first requesting processor.

Application to update protocol

Preparation

The home processor of a block holds a maximum shared distance as its directory information and a shared/private bit as its sate information. Note that neither a valid/invalid bit nor an owner identifier field is needed because the block of the home processor is updated whenever any copy is written to. Other processors hold an address tag and a valid/invalid bit for their cached copies.

Read request

When a processor needs to read a remote memory block, it locally searches a tag and state information of the block, and it then reads the block copy in the local cache if there is a valid copy there. Otherwise, it issues a read request to the home processor. When a home processor receives a read request from another processor, it returns the valid data, sets the state of the block to shared, calculates the hierarchical distance between itself and the requesting processor, and updates the maximum shared distance if necessary.

Write request

Before a processor other than a home processor writes to a copy of the block, it issues an update request that includes the new value to the home processor. The home processor sends an update message to all processors within the shared area after it receives the update request. A processor that has received an update message updates its copy of the block if it has the valid copy and then sends an acknowledgement message to the block's home. A dummy acknowledgement message is returned when the processor does not have a valid copy. After the home processor has received acknowledgement or dummy acknowledgement messages from all processors in the sharing area, it sends an acknowledgement message that means the completion of the update in the shared to the initiating processor.

4.3 Extended network transaction

This section outlines an example of a cache coherence transaction in a hardware DSM system based on a hierarchical coarse directory scheme and hierarchical multicasting and combining mechanisms. The Figure 4.1 and the list below explain the propagation of network messages during the invalidation or update of a shared memory block. Gray leaves in the figure are processors that have a copy of the block and white leaves are processors that don't. Now, the maximum shared distance is assumed two.

(1) A processor that will write to a copy of a remote memory block (the "requester" in the figure) issues an invalidate or update request to the home processor of the block. Each switching node in the interconnection network simply forwards the request to the block's home.

(2) The home processor issues a single invalidate or update message to the shared area which the maximum shared distance indicates. Each switching node multicasts it in all directions within the shared area.

(3) All the processors that have received the invalidate or update message send an acknowledgement message to the home to indicate the completion of the invalidation

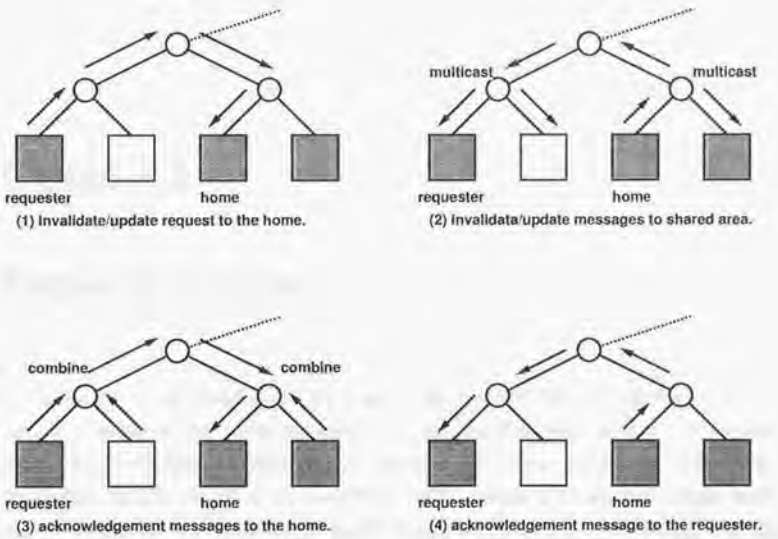


Figure 4.1: Time series of propagation of messages during a coherence transaction.

or update operation in their processing nodes. Each switching node forwards a combined message after it has received acknowledgement messages from all directions in which the invalidate or update message was multicasted. A single message arrives at the home.

(4) The home sends an acknowledgement message to the original requesting processor, in order to indicate the completion of the invalidation or update in the shared area. The transaction finishes when the requesting processor receives the acknowledgement.

Chapter 5

Implementation

When developing an architecture for a massively parallel computer of the next generation, building a real prototype machine is an effective way to verify its mechanisms. OCHANOMIZ-5 (Omnipotent Concurrency-Handling Architecture with Novel OptiMIZers-5) [57, 58, 60] is the prototype parallel computer for the lightweight hardware DSM mechanisms. This chapter explains the architecture and implementation of the prototype.

5.1 System architecture

The prototype machine consists of processing elements, main memories, memory controllers, bus arbiters, network interfaces, switching nodes in the network, and a host computer. Figure 5.1 is a block diagram of the machine. The parallel system consists of four cluster boards and a network board to interconnect those clusters. The main memories are distributed over all clusters.

Each cluster has two SuperSPARC+ processors [56] with 1Mbytes of secondary cache memories. These processors are connected by a shared bus, MBus [53]. A bus arbiter arbitrates between the three bus master, two processors and a network interface. That is, each cluster is itself a shared-bus multiprocessor system. The main memory in each cluster consists of 32 Mbytes of DRAMs. The memory controller, bus arbiter and network

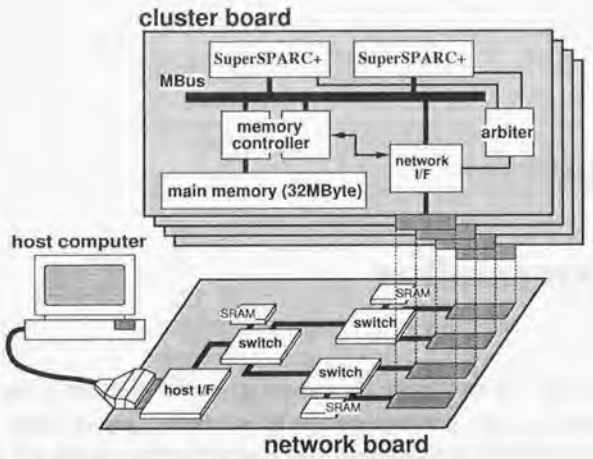


Figure 5.1: Block diagram of the prototype.

interface[1] are implemented using FPGAs (Xilinx XC4010 for the memory controller and the bus arbiter, XC4025 for the network interface) [66].

The prototype has a hierarchical binary tree interconnection network. Each internal switching node is designed on an FPGA (XC4025). There are two separate paths between each parent and child node, upward and downward. Each path is eight-bit wide. Each switching node has two SRAMs (total: 128 Kbytes) that are used as wait buffers for generalized combining.

[1] Although the memory controller and network interface are implemented separately, it is possible to incorporate them both in a current FPGA with sufficient pins.

5.2 Basic Policy

In this implementation of a hardware DSM, we have used the following policies to simplify the hardware management.

- Cluster-level cache
  When a processor accesses a remote memory block, the copy is generated in the main memory of the cluster. This is called a cluster-level cache.

- Employment of an invalidate protocol
  CC-NUMA can have invalidate and update protocols. We have only used an invalidate protocol in this implementation, because the cache coherence protocol on the MBus is write-invalidate.

- Management per a cache line
  The line (block) of cluster-level caches for shared memory is of the same size as the processor's internal caches. The management is performed per line. The management per line allows false sharing to be reduced since the size of a line is small enough (32 bytes).

- Fixed home clusters
  A home cluster is statically assigned to each cache block, by using two bits in a physical memory address.

- Concentrating the management of the directory and owner identification at the home processor
  The block's home holds its directory. When the ownership of a block is transferred from the home to another cluster by an invalidate protocol, the home retains the cluster identification of the owner.

- Sending requests to the home cluster
  Read requests caused by cache miss or invalid state in the cluster-level cache and invalidate requests caused by shared state are issued to the home cluster. The home cluster replies to a read request if the block is valid, or forwards the request to the

owner cluster if it is invalid. On receiving an invalidate request, the home cluster performs the invalidation of the block and copies in the shared area, then replies to the requester with an acknowledgement message ("Ack").

5.3 State transition

A block copy in a cluster-level cache is in one of the following four states[2].

- **Private**
  This is the only valid copy. That is, no valid copy is cached in any other cluster.
  - The initial state of the block in the home is private.
  - The sate of the copy held by the owner after an invalidation is private.
- **Shared**
  More than one cluster has a valid copy.
  - After the home or an owner has processed a read request from a remote cluster, or
  - after a cluster has read the valid copy from the home or an owner cluster,

  the resulting copy is shared.
- **Invalid**
  This is not valid state.
  - The initial state of cache blocks other than the home block is invalid.
  - Copies held by clusters that have received an invalidate message are invalid if the cluster is not requesting processor of the invalidation.

  When the block in the home is itself invalid, the owner has the only valid copy, which is thus private.

[2]In this description of states, the memory block itself is considered a "copy". The original block is not distinguished from copies unless it is specifically referred to.

- **Pending**
  Pending is the state of a copy in the interval
  - between issue of a read request and receipt of the reply, or
  - between issue of an invalidate request/message and receipt of the Ack.

  When a home cluster receives a request for a block that is in pending state, it returns a Nack(not acknowledged). If a cluster other than the home receives a request for a copy that is in pending state and the request is for an invalidation, it is forced to invalidate its copy and return an Ack. Otherwise, it returns a Nack.

Figure 5.2 illustrates the state transitions for copies in cluster-level caches (the pending state is omitted). The shared & dirty state[3] is avoided by writing back the latest copy from the owner to the home when the home receives a read request and the block is invalid in the home.

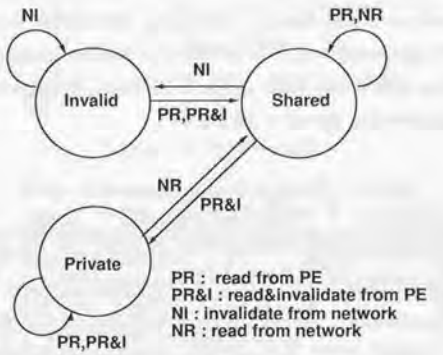


Figure 5.2: State transition diagram for cluster-level cache.

[3]In general, a copy is shared & dirty state when the cluster is not the home, other copies of the block exists, and the block in the home is invalid.

5.4 Memory map and address translation

Of the 32 Mbytes main memory in each cluster, 8 Mbytes are the cluster-level caches, and 2 Mbytes hold the states, owner information, and directories[4]. Figure 5.3 shows the memory map.

The 32-bit virtual address for shared memory includes a two-bit cluster address and a 21-bit offset within a cluster. Translating a virtual shared address into a physical address is straightforward (Figure 5.4). If the most significant bit of the virtual address is one, the address is that for a shared variable. The upper four-bit field of a physical address is an address space identifier. A memory controller uses the address space identifier to distinguish between shared address and local addresses.

5.5 Function of memory controller

The memory controller in the hardware distributed shared memory manages data in main memory in the usual ways, and manages the sharing information in main memory. The sharing information consists of state tags for each block in the cluster-level cache, and the directories and owner information if the cluster is the home. This information is refined to suit the four-cluster system as shown in Table 5.1.

Table 5.1: Fields in sharing information record.

Field	Size (bits)	Usage
Maximum shared distance	1	0: Height=one, 1: Height=two
Valid/Invalid	1	0: Invalid, 1: Valid
Shared/Private	1	0: Private, 1: Shared
Owner	2	Cluster address of the owner
Pending	1	1 during state transition

[4]Although 256 Kbytes of state, owner information and directories is adequate enough for 8 Mbytes of cluster-level cache, each information field for a cache line is aligned to an 8-byte boundary in order to simplify the wiring of the hardware logic.

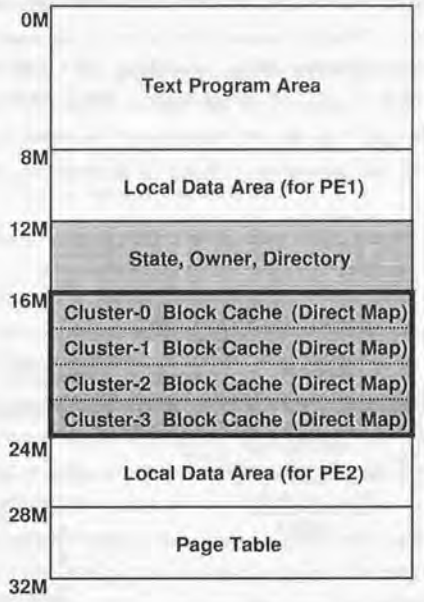


Figure 5.3: Memory map.

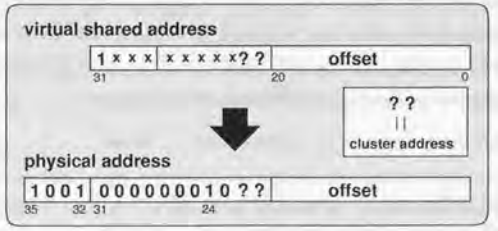


Figure 5.4: Address translation.

The memory controller receives a cluster bus request, then checks the upper four bits of the physical address. When the request is for access to the shared address space, the controller reads the corresponding sharing information and returns valid data or a retry message to the requester, according to the information. When a retry message is returned, the memory controller makes the network interface start a network transaction as the need arises and modifies the state appropriately. Figure 5.5 illustrates the control flow.

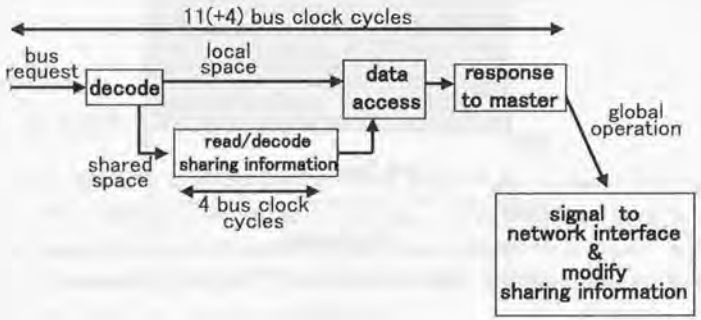


Figure 5.5: Control sequence for the memory controller.

It takes 11 bus-clock cycles to complete a memory transaction that requires access to a 32-byte block in the local address space. Access to the shared address space takes four extra cycles to read and decode the sharing information. However, references to shared variables in a program don't always bring about the extra cycles. After the initial cache miss, cache hits within the processor's internal cache can decrease overheads caused by the extra cycles.

5.6 Function of network switching node

A network switching node takes charge of hierarchical multicasting and generalized combining. The network of the prototype implements the following subset of generalized combining.

- Combining of acknowledgement messages [42]

  Acknowledgement messages are combined by setting the delayed time to eternity and dynamically setting the number for combining when the corresponding multicast occurs. The matching key is a memory address.

- Hierarchical hardware barrier[5]

  Each switching node in the network combines barrier request signals from its child nodes and sends one signal to its parent node, and the root node sends completion signals to the child nodes after it has received the barrier request signals from them. Each switching node then multicasts the completion signal downward. The name space of the hardware barrier is the processor space, so the matching key is the processor group ID. The delayed time is eternal because of the nature of barrier synchronization, that is, because barrier synchronization must be completed.

- Combining of atomic requests

  The generalization of processing function allows atomic requests such as Fetch&Add [16] or Test&Set to be combined. Doing so can reduce the occurrence of tree saturations caused by hot-spot contentions. The generalization of arrival requirement increases the rate of successful matching. The matching key is the memory address.

- Combining of read requests

  Combining of read requests that are sent to one memory address reduces the total number of messages. Future matching increases the success rate. The matching key is the memory address.

[5]Really, the interconnection network of OCHANOMIZ-5 implements the Hierarchical Elastic Barrier[43].

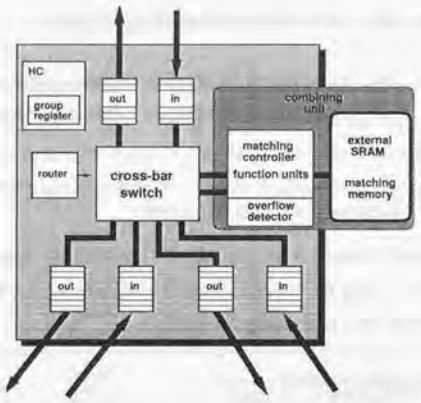


Figure 5.6: Block diagram of a switching node.

- Combining of invalidate requests

  Invalidate requests that are sent to one memory block are combined. Future matching can be applied to this combining. As a result, the invalidation caused by only one request is performed and the other requests are not served. The matching key is the memory address.

Each internal switching node in the network has the following elements for the generalized combining operations (Figure 5.6).

- Switching elements and controller (router)

  The switching mechanism is a cross-bar with four inputs and four outputs connecting a parent node, two child nodes and a combining unit.

- Memory for matching (wait buffer)

  Each switching node has SRAM that acts as its wait buffer. The control logic uses a hash table to cut down the overhead on searching for entries.

- Function unit and controller

  The function unit places the address and data in a message to a register, and then performs an appropriate operation between the register and the corresponding entry in the wait buffer.

- Detector of space overflow

  A detector checks for overflow in the space assigned to each hash key. If it detects overflow of messages, it generates an interrupting packet and sends it to the nearest processor.

- Hierarchy controller (HC) and group registers

  Each switching node has a hierarchy controller that is separate from its other elements. The controller is used to build hierarchical hardware barriers.

5.7 Memory and network transactions

5.7.1 Types of transactions

For cacheable shared memory access, the following four memory requests are issued from processing elements to a local bus.

- Coherent Read
- Coherent Invalidate
- Coherent Read & Invalidate
- Write

The following seven inter-cluster messages are used in network transactions for cacheable shared memory access.

- Read
- Reply

- Invalidate request to home (InvReq)
- Acknowledgement to InvReq requester (InvReqAck)
- Invalidate multicast (InvMul)
- Acknowledgement to home (InvMulAck)
- Not acknowledged (Nack)

Other inter-cluster messages/signals are as follows:

- Write
- Test&Set
- Barrier request signal (BRS)
- Barrier complete signal (BCS)

The messages/signals used in network transactions are divided into four types for the generalization of arrival requirement.

- **Type-0**

  Indicates that the delayed time is zero and the message does not wait at switching nodes. Type-0 messages do not pass through the combining unit, and are not combined.

- **Type-1**

  Indicates that the delayed time is zero. An arriving message does, however pass through the combining unit in switching nodes. If the message is a first request for access to an address, the message is forwarded immediately and an entry is made in the wait buffer. If the message is not such a first request, and can be combined with a preceding message, it is not forwarded. At the same time, the corresponding entry in the wait buffer is modified for the reply message to find that combining has occurred. When the reply message arrives at the switching node, it is multicasted if the entry has been modified. At the same time, the entry is cleared.

- **Type-2**

  Indicates that a message waits at switching nodes for a specified time. One of four time values specified in advance can be selected. This type is applied to requests such as Fetch&Add requests that includes operand data.

- **Type-3**

  Indicates that the delayed time is eternal. A combined message is forwarded only after all related messages have arrived. That is, the corresponding entry in the wait buffer is modified when each message arrives at the switching node, and the entry is cleared and the message is forwarded only when the last message arrives. This allows combining to be ensured.

Type-1 messages are used for matching in the future, and all combinable messages can be combined as long as the entry is still in the wait buffer. It has the lowest overhead and is the most tolerant of arrival time lags of any method of combining messages for which arrival cannot be predicted.

Table 5.2 shows the network messages that correspond to the above four types[6]. Test&Set and Fetch&Add are requests for non-cacheable shared memory space, and Barrier signals (BRS, BCS) is the hardware barrier signal. Messages other than those listed in the table are Type-0.

Table 5.2: Types of waiting time.

Type	Delayed time	Purpose	Remark
0	zero	InvMul Write BCS	No combining
1	zero	Read Reply InvReq InvReqAck Nack Test&Set	Future matching
2	specified time	Fetch&Add	Four different preset times
3	eternity	InvMulAck BRS	Predictable combining

[6]Fetch&Add in the table is not implemented in the prototype.

5.7.2 Operation of the memory controller

The relationship between local bus requests and network messages that are invoked by the requests, and the processing of a memory controller in terms of the states, the owner field and the directory are described in this section.

"Ready", "Retry" and "Inhibit" are signals on the local bus. Ready indicates that a transaction has been finished normally. Retry makes the requester try a bus request again. Inhibit is a signal asserted by a processor that has valid data and will supply it.

- Coherent Read from a processor
  When the Pending bit is one, a memory controller completes a transaction by asserting a Retry signal and making the processor relinquish the bus. When the Pending bit is zero and the block is valid, it completes the transaction by supplying data. If the block is not valid, the controller issues a Read request to the home (or the owner if the subject cluster is itself the home) and completes the transaction by asserting a Retry signal. At the same time, it sets the Pending bit. When another processor in the same cluster asserts an Inhibit signal, memory controller processing is interrupted.

- Coherent Invalidate from a processor
  When a Pending bit is one, a memory controller completes a transaction by asserting a Retry signal and making the processor relinquish the bus. When the Pending bit is zero and the block is private, it completes the transaction by asserting a Ready signal. When the block is shared, it issues InvReq to the home cluster if the subject cluster is not itself the home, and asserts a Retry signal. If the subject cluster is the home cluster, the memory controller sets its cluster number in the Owner field, issues InvMul with the directory value, and finishes with a Retry. At the same time, it sets the Pending bit.

- Coherent Read & Invalidate from a processor
  When the Pending bit is one, a memory controller completes the transaction by asserting a Retry signal and making the processor relinquish the bus. When the Pending bit is zero and the block is invalid, it issues a Read request to home (or the owner if the cluster is itself the home) and finishes the transaction by asserting a Retry signal.

  At the same time, it sets the Pending bit. If the block is valid and private, it completes the transaction by supplying data. The assertion of an Inhibit signal, however, can interrupt processing by the memory controller. If the block is valid and shared, the controller issues InvReq to the home cluster if the subject cluster is not itself the home, and asserts a Retry signal. If the cluster is the home, it sets its cluster number in the Owner field, issues InvMul with the directory value, finishes with a Retry. At the same time, it sets the Pending bit.

- Write from a processor
  A memory controller performs a normal write operation. A processor issues this request when it replaces a dirty block in the processor's internal cache. The write requests from processors are issued only when a block in a cluster-level cache is private.

- Read from a remote cluster
  The network interface issues a Coherent Read&Invalidate request[7] on the bus when it receives a Read request from another cluster. The memory controller returns a Nack when the Pending bit is one. When the block is invalid, it issues a Read to the cluster indicated by the Owner field, returns Nack to the requester, and sets the Pending bit. When the block in valid and Inhibit is not asserted, the controller returns a Reply and sets the Shared bit. When the cluster is the home, it also simultaneously updates the directory if necessary. When Inhibit is asserted, the memory controller updates the cached block in main memory by using data from the bus and sets the Shared bit and new directory.

- Reply from a remote cluster
  The memory controller updates the cached block in its main memory and sets the Valid bit and the Shared bit. It also updates the directory when it is the controller in

[7] When a block is shared in terms of processor's internal caches and the processor owns the block, the processor finishes the transaction at its discretion even if the transaction should invoke an invalidate message to remote clusters. To prevent this situation from happening, the network interface converts Read requests from remote clusters into Coherent Read & Invalidate requests. That is, no ownership in terms of processor's internal caches is allowed when the block is shared among clusters.

the home cluster.

- InvReq from a remote cluster
  The memory controller returns Nack when the Pending bit is one. If the Pending is zero, it issues InvMul with the directory value, sets the Pending bit and writes the cluster ID of the requester into the Owner field.

- InvReqAck from a remote cluster
  The memory controller sets the Valid and Private bits, and resets the Pending bit.

- InvMul from a remote cluster
  The memory controller returns an InvMulAck. When the copy exists in the cluster-level cache, it makes the copy invalid (Valid = 0) regardless of the value of the Pending bit. The Pending bit is not changed. When the copy does not exist, no operation is done for the cluster-level cache.

- InvMulAck from a remote cluster
  The home cluster only receives this message. The memory controller looks up the Owner field. It makes the copy valid and private (Valid = 1, Shared = 0) and resets the Pending bit, when the owner value is the controller's cluster number. Otherwise, it returns an InvReqAck to the original requester indicated by the Owner field, makes the copy invalid (Valid = 0), and resets the Pending bit.

- Nack from a remote cluster
  The memory controller resets the Pending bit. All other state bits are unchanged.

5.7.3 Operation of the switching node

Details of switching node's processes in terms of network messages in an invalidation transaction are given in this section.

Each switching node has a total of seven bits per cache block; two 2-bit ID fields for the source cluster number of InvReq in each direction, two bits (Arrival bits) to indicate the arrival of InvReq from each direction, and one bit (Ack bit) to indicate the arrival of

InvMulAck. The seven bits form an entry in the wait buffer. The processes in a switching node when InvReq, InvMul, InvMulAck or InvReqAck arrive at the node are outlined below.

1. InvReq
   The switching node sets an ID field and an Arrival bit for the requested block. It then forwards the InvReq as long as the other Arrival bit is zero. If the other Arrival bit is set, it does not forward anything.

2. InvMul
   The switch multicasts the message in all directions within the shared area except the direction from which the InvMul itself comes. It sets the Ack bit in the entry if and only if it is the root node for the shared area.

3. InvMulAck
   The switch sets the Ack bit if the bit is currently zero. Otherwise, it resets the Ack bit and forwards the InvMulAck to the home.

4. InvReqAck
   The switch looks up the Arrival bits in the entry. It forwards the message in the direction of the source, i.e. for which the Arrival bit is one, or multicasts it in both directions when both Arrival bits are one, then resets the Arrival bits.

5.7.4 Example of a coherence transaction

The operations required for a coherent invalidation of cached blocks are used as an example of a network transaction. Suppose that two clusters issue an InvReq to the same memory block. The steps involved in the transaction are outlined. Figure 5.7 shows the operations of the transaction in sequence.

(1) When a processor issues a coherent invalidate request for a shared block, the network interface in the cluster issues an InvReq by sending a **Type-1** message towards the home cluster of the block (Figure 5.7 (a)). When this InvReq arrives at a network switching node, the node's wait buffer is searched. If there is no entry indicating a

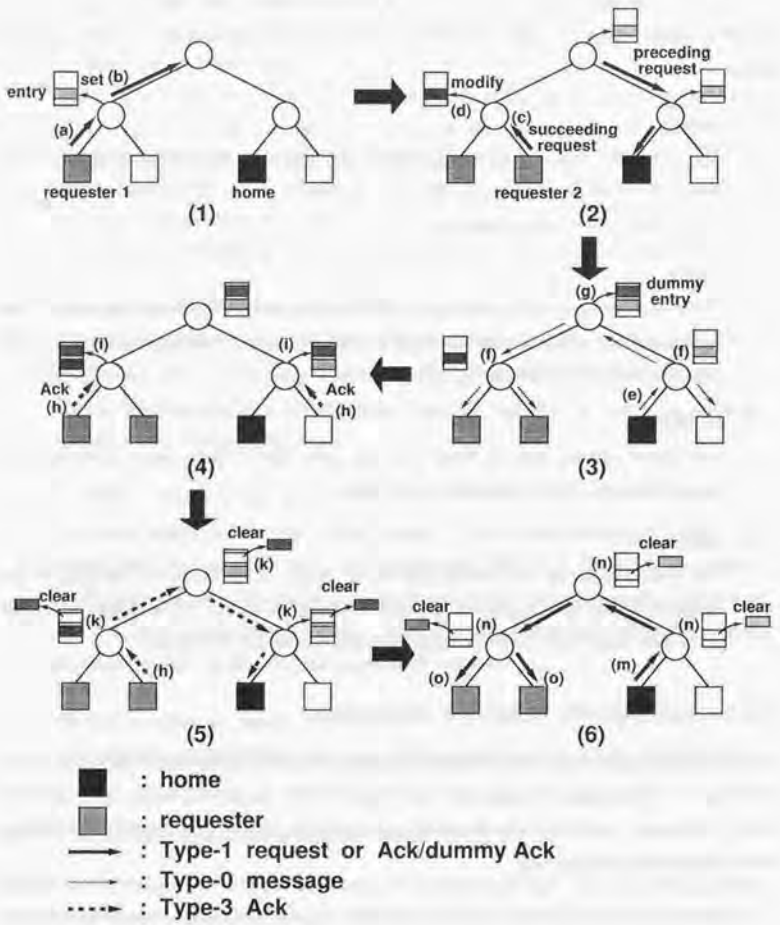


Figure 5.7: The series of operations during an invalidate transaction.

preceding request sent to the same address, such an entry is created and InvReq is forwarded to the next node because it is Type-1 (Figure 5.7 (b)).

(2) If an entry is found, the message is not forwarded but combined (Figure 5.7 (c)). The entry is then modified to indicate that a final reply acknowledgement message (InvReqAck) should later be multicasted (Figure 5.7 (d)).

(3) When the InvReq reaches the home cluster, the network interface issues a single InvMul by sending a **Type-0** message towards the shared area determined by the maximum shared distance (Figure 5.7 (e)). When this Type-0 InvMul arrives at an internal switching node, it is multicasted in all directions within the shared area (Figure 5.7 (f)). The root node of the shared area only forwards the InvMul to the other non-originating child node, and a dummy Ack bit is set in the wait buffer (Figure 5.7 (g)). This Ack bit is used later for the collection of acknowledgement messages.

(4) Each cluster that receives the InvMul performs invalidation of the block and returns a **Type-3** InvMulAck message (Figure 5.7 (h)). If the cluster does not have a copy of the block or is the original requester (requester 1), it returns a dummy InvMulAck[8]. When an InvMulAck arrives at an internal switching node, the wait buffer is searched. If the Ack bit is not currently set, it is set (Figure 5.7 (i)).

(5) If the Ack bit is one, InvMulAck is forwarded and the Ack bit is cleared (Figure 5.7 (k)). This process can be regarded as combining of acknowledgement messages. The home results in receiving a single InvMulAck.

(6) When the combined InvMulAck reaches the home cluster, the network interface returns an InvReqAck by sending a **Type-1** message to the original requester (Figure 5.7 (m)). When this InvReqAck arrives at an internal node, it is forwarded (multicasted if necessary. For example, the Nack is sent to requester 2 in the figure), and the corresponding entry is cleared (Figure 5.7 (n)). The transaction finishes when all requesters receive an InvReqAck or Nack (Figure 5.7 (o)).

[8] An InvMulAck and dummy InvMulAck don't differ in terms of actual network packets.

5.8 Scale of required hardware logic

5.8.1 Memory controller

Our memory controller includes management logic for the cluster-level cache. It is, therefore, possible that the controller will be larger in terms of hardware than a normal memory controller. We designed two memory controllers: one to manage normal data operations and cluster-level cache coherence, and the other without the management logic for the cluster-level cache. Each is embedded in a FPGA, Xilinx XC4010 with a capacity equivalent to 10,000 gates. Table 5.3 shows the device utilization. "Non-cache" in the table is for the memory controller without cache management, and "Cache DSM" is for the memory controller with cache management. "Max available" shows the capacity of an XC4010. "CLB" is a configurable logic block that contains F, G, and H function generators, and two flip-flops. "F&G FG" is an F or G function generator that can generate a combination circuit with four inputs and one output, or a 16-bit RAM element. "H FG" is a H function generator that can generate a combination circuit with three inputs and one output. "FF" is a flip-flop.

The memory controller with cache management logic requires about 30-40% more gates than the memory controller without such management logic. The increase is not really great because the non-cache controller only requires about 4,000 logic gates.

Table 5.3: Device utilization for the memory controller.

	Non-cache	Cache DSM	Max. available
CLB	211	274	400
F&G FG	217	305	800
H FG	42	82	400
FF	135	157	800

5.8.2 Switching node

Our mechanism for combining messages does not require a combining queue [16]. Our switching node consists of simple wired logic circuits, and memory for the wait buffer. The logic circuits are built within the capacity of a single FPGA, XC4025, of which contents are equivalent to about 25,000 gates. Table 5.4 shows the device utilization. "No-combine" is for the switch without a combining unit. "Combine" is for the switch with a combining unit. The switch without a combining unit requires the equivalent of about 7,000 gates. The switch with a combining unit requires about 50% more gates. Pfister estimated that the scale of hardware for a switch with a combining queue would be 6–32 times that for a non-combining switch [47]. From this point, our method is cost-effective.

Table 5.4: Device utilization for the switching node.

	No-combine	Combine	Max. available
CLB	400	670	1024
F&G FG	682	963	2048
H FG	74	159	1024
CLB FF	184	294	2048

Chapter 6

Performance Evaluation

6.1 Network communication

The processors in the prototype machine run at 60 MHz, and the other components run at 22 MHz. The interconnection network is a cut-through network, and network paths are eight-bit wide. The latency for remote memory transactions is shown in Table 6.1. "Distance" denotes the height of the minimum subtree which includes the source and destination clusters, or the shared area. "Type" is one of three request types; "Write", "Read", or "Inv". "Inv" denotes the time needed for a processor in a home cluster to complete the invalidation of the shared area after the issue of an invalidate request to a local bus. "Size" is the amount of data that a request is for. Latency is shown for both types of communication; non-combinable communication and combinable communication, "Combining" ON and OFF respectively. "Cycles" is the number of system clock cycles that it takes to complete the communication after the initial request is issued by a processor. In the last column, the latencies are shown by microseconds.

Table 6.2 shows the bandwidth for remote memory access. Bandwidth was measured by having the specified number of processing elements (PEs) repeatedly issue 32-byte write or read requests to the next-door (Distance = 1) cluster. When the number of PEs is four, one PE in each cluster issues the write or read requests. Four PEs gave a better result than eight PEs for write execution because of bus contention within each cluster. That is, in the four-PE case, one PE and the network interface in each cluster are bus masters. In the

Table 6.1: Latency for remote memory access.

Distance	Type	Size (bytes)	Combining (ON/OFF)	Cycles	μsec
1	Write	1–8	OFF	33	1.50
			ON	43	1.95
	Write	32	OFF	60	2.73
			ON	70	3.18
	Read	1–8	OFF	45	2.05
			ON	65	2.95
	Read	32	OFF	74	3.27
			ON	94	4.27
	Inv	32	ON	76	3.45
2	Write	1–8	OFF	41	1.86
			ON	71	3.23
	Write	32	OFF	68	3.09
		32	ON	98	4.45
	Read	1–8	OFF	61	2.77
			ON	121	5.50
	Read	32	OFF	88	4.00
		32	ON	148	6.73
	Inv	32	ON	138	6.27

eight-PE case, two PEs and the network interface are bus masters.

Table 6.2: Bandwidth for remote memory access.

Number of PEs	Type	Bandwidth (Mbyte/sec)
1	Write	15.2
	Read	7.3
4	Write	58.3
	Read	24.5
8	Write	47.8
	Read	30.8

Table 6.3 shows the improvement in communication performance gained by combining of read requests in the network. The results are for all eight PEs repeatedly issuing read requests to the same memory block, i.e. to the same address in a cluster. The result under "OFF" shows the case without combinable requests. The result under "ON" in the table shows the situation when the maximum possible amount of combining is made. That is, any contention in a switching node is processed as combining. The result implies the limit of the performance improvement that can be obtained by combining. The bandwidth in "ON" is as high as for the equivalent case in Table 6.2, where there is no contention in the network.

Table 6.3: The effect of combining read requests.

Combining ON/OFF	Bandwidth (Mbyte/sec)
OFF	19.3
ON	30.0

Combining of Test&Set requests can reduce hot-spot contention at a destination, although at most only one request is satisfied in terms of receipt of the value "0". We used

a Test&Set request for lock acquisition of a critical section. Each processor repeatedly acquires a lock and increments a counter variable in the critical section. The following shows the program model that was used.

```
for (i=0; i<LOOP_COUNT; i++) {
  while(Test&Set(LockVar) != 0); // Lock Acquire
  counter += 1;                  // Critical Section
  LockVar = 0;                   // Lock Release
}
```

"LockVar" is a shared variable for the lock and the variable is non-cacheable. "counter" is also a shared variable and non-cacheable. The execution of the function "Test&Set" on a processor generates a Test&Set request to a local bus. Table 6.4 shows the result for eight PEs with LOOP_COUNT = 2^{20}. Combining reduced the execution time by 28%.

Table 6.4: Execution time for the lock acquisition loop.

Combining ON/OFF	Total time (seconds)
OFF	119.1
ON	85.7

6.2 Effect of generalized combining

We will now illustrate the impact of several types of combining by computing the nth power of the matrix A[N][N] with $N = 128$ and $n = 32$.

Each PE gets an element number (eNO) for each execution unit. The element number is the value of a shared variable which indicates the value of $i \times N + j$ for the element $A[i][j]$ ($0 \leq eNO < N \times N$). The shared variable is protected by a critical section. Each PE computes $\sum_{i=0}^{N} A[eNO/N][i] \times A[i][eNO\%N]$ in a unit. Barrier synchronization occurs at the beginning and end of each increment in the power. We used a software barrier or a

hierarchical hardware barrier. The former is implemented by using a critical section and a shared counter variable. When the counter value reaches the number of PEs, the barrier is completed. The program is as follows:

```
shared double A[N][N];
shared long   eNO, LockVar, counter;
double AA[N][N]; /* copy of matrix A */
int    k;
for(k=0; k<n; k++) {
  counter = 0;
  copy(A,AA);            /* copy from A to AA */
  barrier();
  while(1) {
    while(Test&Set(LockVar) != 0);
    eNO = counter;
    counter = eNO + 1;
    LockVar = 0;
    if (eNO < N*N) {
      int    i;
      double d = 0.0;
      for (i=0; i<N; i++)
        d += AA[eNO/N][i]*AA[i][eNO%N];
      A[eNO/N][eNO%N] = d;
    }
    else {
      barrier();
      break;
    }
  }
}
```

Figure 6.1 shows the flowchart of the algorithm.

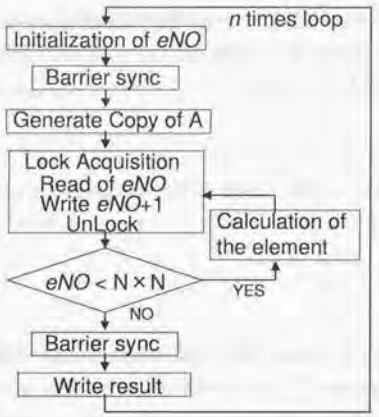


Figure 6.1: Flowchart of the algorithm.

Shared variables (A, $counter$, and $LockVar$) are non-cacheable. Other variables are local and cacheable for each PE. The instances of the matrix A reside in the cluster-0. Each PE creates a copy of A in its local memory ($copy(A, AA)$) (Figure 6.2). It is in this operation that the effect of combining read requests is gained. (Read request for a block was used. A block is of 32-byte data.)

The times for execution by 1, 2, 4, and 8 PEs are listed in Table 6.5. 2 indicates computation by two PEs in one cluster (cluster-0). 4 indicates computation by four PEs in two clusters (cluster-0 and cluster-1). "NC" means computation without combining. The "HC" column is for computation with a hierarchical hardware barrier. "HLC" indicates that Test&Set requests can be also combined, and "HLRC" indicates that read requests can be combined too.

For 4-PE execution, combining of Test&Set requests and combining of read requests do not contribute to the improvement of the execution. Inter-cluster communications are one to one only, since the matrix and other shared variables are all located in the memory of one cluster (cluster-0). This is the reason why any combining of requests from different

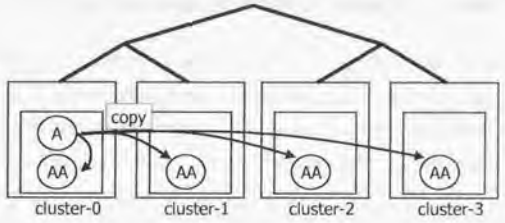


Figure 6.2: Creation of the copy AA.

directions does not occur at switching nodes[1]. In more detail, network packets for the requests must pass through a combining unit in every switching node. The extra latency increases the execution time. In 8-PE execution, HLRC provides the best result. The gain derived from combining exceeds the losses due to the increased latency. HLRC is 7.7% faster than NC. Figure 6.3 is the graph of the execution when the number of PEs is between 2 and 8.

Table 6.5: Execution time (seconds) for the power of a matrix.

Number of PEs	NC	HC	HLC	HLRC
1	19.9	19.9	19.9	19.9
2	10.1	9.9	9.9	9.9
4	7.3	6.5	6.6	6.7
8	6.5	6.4	6.3	6.0

[1] Combining of requests from two processors in a cluster is done locally in the cluster

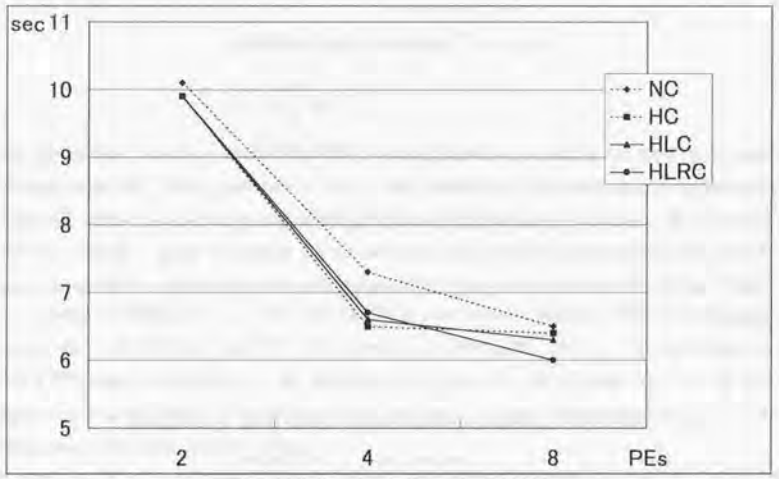


Figure 6.3: Execution time (seconds) for the power of a matrix.

6.3 Effect of lightweight hardware cache coherence DSM

We employed the LU-Contig program from SPLASH-2 benchmark suit [64], by using 1, 2, 4, and 8 PEs, to evaluate the performance of the lightweight hardware cache coherence DSM system[2]. LU-Contig performs blocked LU factorization of a dense matrix. The size of a matrix used is 512×512, and it has 16×16 sub-blocks. When the number of PEs is 2, two PEs in one cluster execute. When the number of PEs is 4, four PEs in two clusters execute.

Table 6.6 shows the results of LU-Contig. We used a hierarchical hardware barrier as barrier synchronization for the program. "Cache OFF" means that caches are not used (neither processor's internal cache nor cluster cache) for shared variables. (Local variables and instructions are cached.) "Cache ON" means computation with the full hierarchy of caches. Execution with two PEs is more than twice as fast as execution with a single PE. This is the effect of the superlinear when the total amount of data exceeds the capacity of a single PE's internal cache. Cache ON is much faster than Cache OFF, that is, execution with eight PEs takes only 19% of the time for a single PE, and demonstrates the advantages of parallelization.

Table 6.6: Execution time (seconds) for LU-Contig.

Number of PEs	Cache OFF	Cache ON
1	149.36	10.57
2	64.39	4.96
4	72.32	2.71
8	47.18	1.97

Figure 6.4 shows the executions with Cache ON and Cache OFF, and Figure 6.5 shows results for execution with Cache ON.

[2]In the execution, combining of read requests are not used.

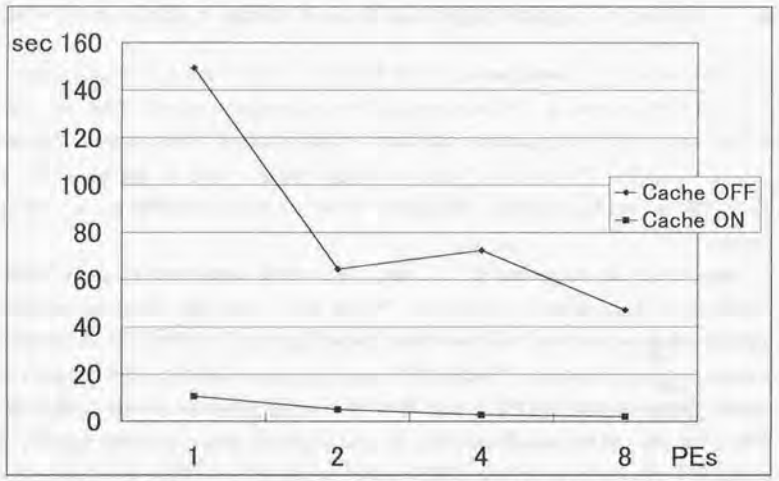


Figure 6.4: Execution time for LU-Contig (Cache ON and Cache OFF).

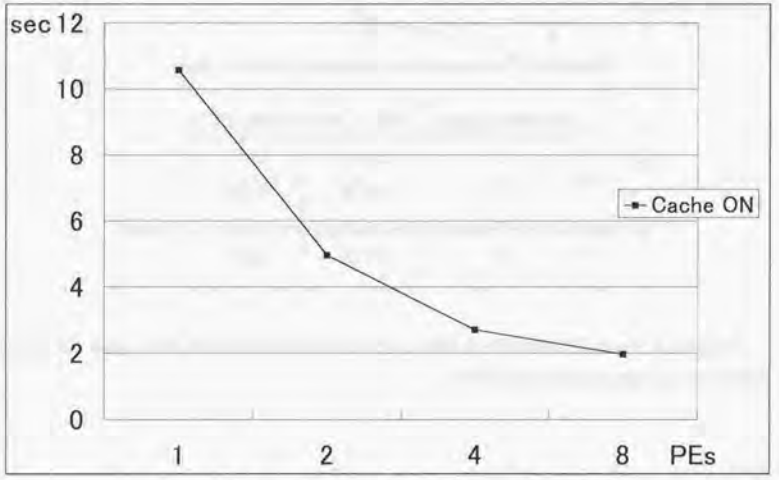


Figure 6.5: Execution time for LU-Contig (Cache ON).

Chapter 7

Consideration for a Large-Scale System

From the actual implementation of the hardware DSM system on the prototype machine, various values were obtained, such as the time required for a message to pass through a switch. In this chapter, coherence processing (invalidation) on a larger scale system is considered in terms of the obtained values, and the hierarchical coarse directory with multicasting and combining of messages is compared with the fullmap directory system.

7.1 Parameters and time required

The following parameters are used. We assume that there is one processing element in each cluster.

- *NPROC* is the number of processors that share a memory block. The value is between 2 to 256.
- *COM* indicates the use of combining. The value is 0 for the fullmap directory scheme and 1 for the hierarchical coarse directory scheme.
- *TREE* indicates the degree of the network tree structure, a binary or a 4-ary, and is 2 or 4 respectively.
- *WIDTH* indicates the width of a network path. The width is between 1 and 4 bytes.

- *PLENGTH* is the length of a network packet. We use a fixed 8-byte packet for invalidate and acknowledgement messages.

The values shown in Table 7.1 are used as the times required by the various system elements. These values are based on measurements of the operation of the prototype machine. The items are defined below.

Table 7.1: Times required at elements.

Required time	Cycles
H_MCTL_REQ	7
H_NETIF_OUT	3
$SWITCH$	$4 + COM \times (PLENGTH/WIDTH + 4)$
D_NETIF_IN	$3 + PLENGTH/WIDTH$
D_MEMCTL_IN	7
D_NETIF_OUT	3
H_NETIF_IN	$3 + PLENGTH/WIDTH$
H_MEMCTL_ACK	9

- *H_MCTL_REQ* is the time required between the issue of an invalidate request by a processor in a home cluster and the notification to a network interface by a memory controller in the cluster.
- *H_NETIF_OUT* is the time required for a network interface in a home cluster to dispatch an invalidate message to a network. (For a fullmap system, it is necessary to send as many messages as there are target processors. Hence, the time $(PLENGTH/WIDTH) \times (i-1)$ is added to a message for the ith target processor.)
- *SWITCH* is the time required for a message to pass through a switching node. The value is 4 in a fullmap system. The time required to pass through a combining unit in a switching node (PLENGTH/WIDTH + 4) is added for the hierarchical coarse directory scheme.

- *D_NETIF_IN* is the time required for a network interface in a destination cluster to issue a request to a local bus after it has received a message that causes the bus request from another cluster.
- *D_MEMCTL_IN* is the time required for a memory controller in a destination cluster to decode a bus request and reply to a network interface in the cluster.
- *D_NETIF_OUT* is the time required for a network interface in a destination cluster to dispatch an acknowledgement message to a home cluster.
- *H_NETIF_IN* is the time required for a network interface in a home cluster to transfer an acknowledgement message to the local bus after it has received the message from another cluster.
- *H_MEMCTL_ACK* is the time required for a memory controller in a home cluster to decode an acknowledgement message and reply a network interface in the cluster.

7.2 Methodology

The number of cycles required for an invalidate transaction to be completed were calculated for various numbers of sharing processors. From the point of view of the optimization for sharing of data, sharing processors are arranged as described below.

- A processor number (Pno) is assigned to each processor in order from left to right (Figure 7.1).
- A processor with Pno = 0 is a home processor.
- Processors with Pno = $0, 1, ..., n-1$ are the sharing members when the sharing number is n.

When a home processor invalidates sharing processors in the fullmap scheme, it first sends an invalidate message to the processor with the largest processor number, then sends to the one with the next largest processor number, and so on, in order to reduce the total time. This is because the larger processor number is, the higher the latency is. Similarly in

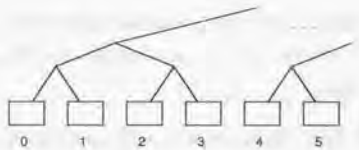


Figure 7.1: Processor numbers.

the hierarchical coarse directory scheme, a switching node sends a message in the direction of the processor with the largest processor number of all directions when it starts the multicasting of a message. At that point, a delay ($PLENGTH/WIDTH$) is added according to the order of transfer.

From the above assumptions, the time required for an invalidation on the round trip between a home and a destination processor ($RTL_{fullmap}$) is represented for the fullmap scheme by using the following expression.

$$\begin{aligned} RTL_{fullmap} &= H\_MCTL\_REQ + H\_NETIF\_OUT \\ &+ (PLENGTH/WIDTH) \times (Pno_{max} - Pno) \\ &+ (2 \times height(Pno, home) + 1) \times SWITCH \\ &+ D\_NETIF\_IN + D\_MEMCTL\_IN + D\_NETIF\_OUT \\ &+ (2 \times height(Pno, home) + 1) \times SWITCH \\ &+ H\_NETIF\_IN + H\_MEMCTL\_ACK \end{aligned}$$

Here, $home$ is the processor number of the home processor, that is 0. Pno is the processor number of the destination processor. $height(Pno, home)$ is the height of the minimum subtree that includes a home and destination processors decremented by 1. For example, $height(10, 0)$ equals 3 when the network is a binary tree structure. Pno_{max} is the largest processor number among the set of sharing processors. The expression is evaluated for each sharing processor and the maximum value for the results is regarded as a temporary value to represent the total latency. To consider hot-spot contention at a home, any cycles during processing of an acknowledgement message at a home

($H\_NETIF\_IN + H\_MEMCTL\_ACK$), that overlap on different processors, are added the latency. The final result is the total latency.

The latency RTL_{msd} is calculated by using the following expression for the hierarchical coarse directory scheme.

$$\begin{aligned} RTL_{msd} &= H\_MCTL\_REQ + H\_NETIF\_OUT \\ &+ \sum_{i=0}^{i<2\times height(Pno,home)+1} (SWITCH + order \times (PLENGTH/WIDTH)) \\ &+ D\_NETIF\_IN + D\_MEMCTL\_IN + D\_NETIF\_OUT \\ &+ (2 \times height(Pno, home) + 1) \times SWITCH \\ &+ H\_NETIF\_IN + H\_MEMCTL\_ACK \end{aligned}$$

Here, $order$ ($0 \leq order < TREE$) in the expression is the order of multicasting at each switching node. The expression is evaluated for each processor in the shared area and the maximum value among the results is regarded as the total latency.

7.3 Results

The results for a fullmap directory system when the interconnection network is either a binary or 4-ary tree are shown in Figures 7.2 and 7.3 respectively. The horizontal axis indicates the number of sharing processors ($NPROC$). The vertical axis indicates the total number of cycles required for invalidating the shared area. "fullmap2x1", "fullmap2x2" or "fullmap2x4" indicates that network paths are one-, two- and four-byte wide respectively, and the same representation is used for the 4-ary network. The results show that the total time is in proportion to the number of sharing processors. This is caused by sequential processing on acknowledgement messages at the home cluster. In addition, comparison of Figures 7.2 and 7.3 shows that the topology of the network has little effect.

Figures 7.4 and 7.5 show the results for the hierarchical coarse directory system. The representation is the same as in the previous two figures. The figures show that the total time depends on the height of the subtree which indicates a shared area. In addition, the topology of the network does influence the total time.

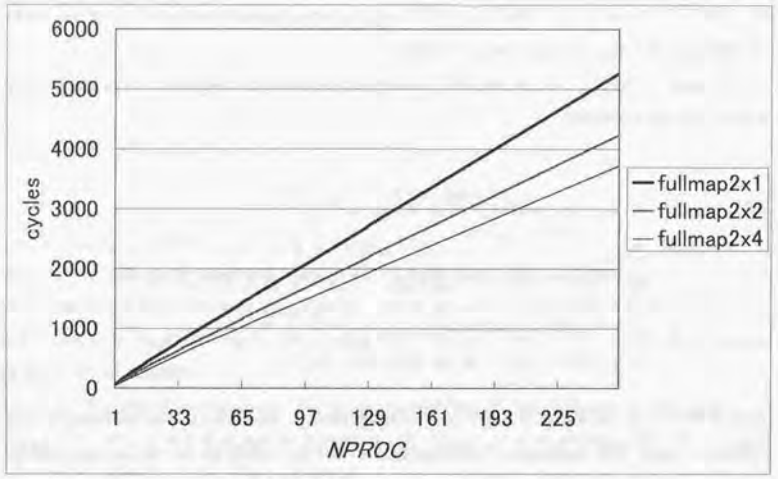


Figure 7.2: Fullmap directory, binary tree.

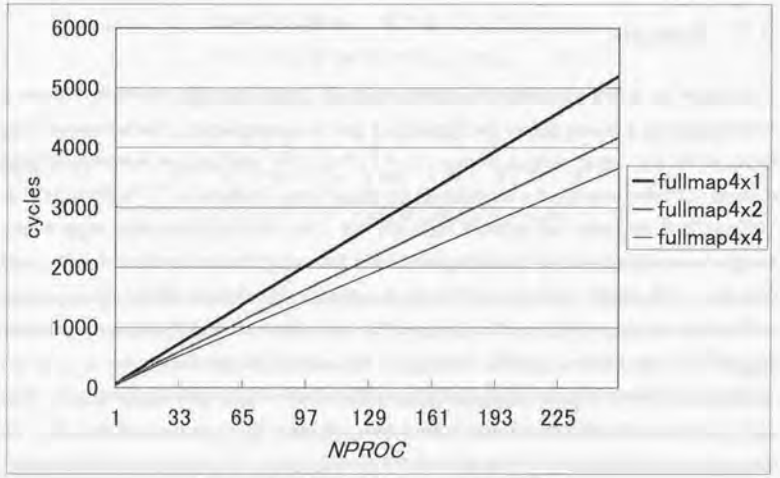


Figure 7.3: Fullmap directory, 4-ary tree.

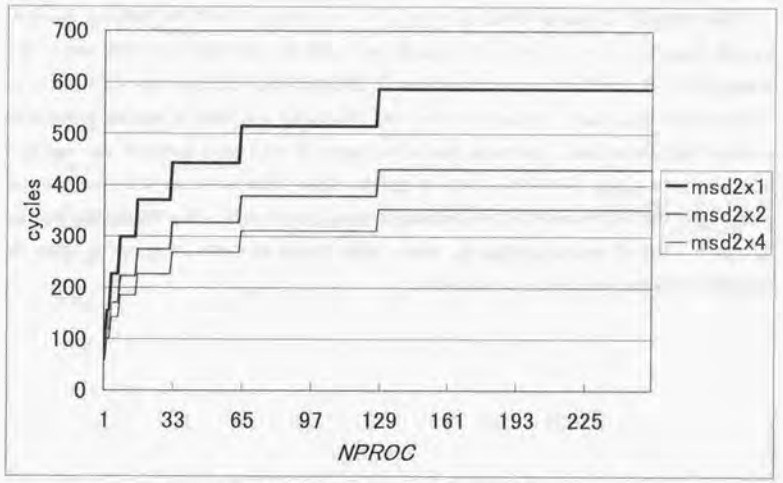


Figure 7.4: Hierarchical coarse directory, binary tree.

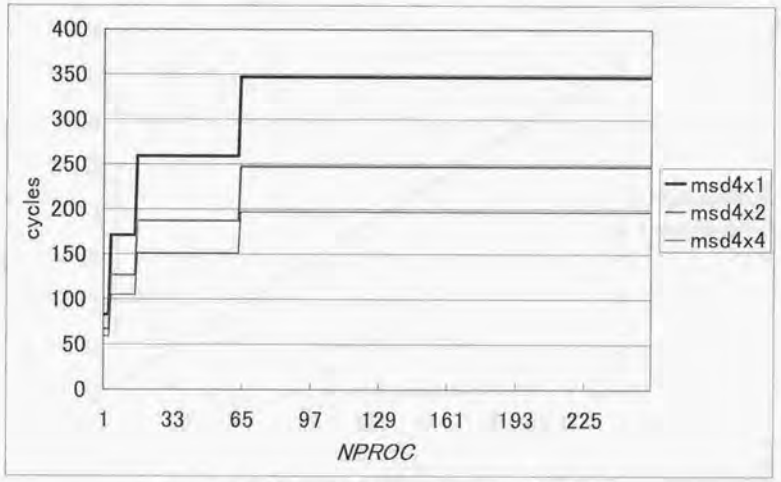


Figure 7.5: Hierarchical coarse directory, 4-ary tree.

The hierarchical coarse directory system is now compared with the fullmap directory system. Figures 7.6 and 7.7 show the results for a network with four-byte wide paths. As a whole, the total time for the hierarchical coarse directory system is less than for the fullmap directory system. Figures 7.8 and 7.9 show the results for less than 16 sharing processors, and are magnifications of part of the previous figures. For a binary network, the total time taken by the fullmap directory system is greater when there are more than nine sharing processors. For a 4-ary network, the fullmap system takes longer when the sharing number is over 5. The results show that the hierarchical coarse directory is effective when the number of sharing processors is not small.

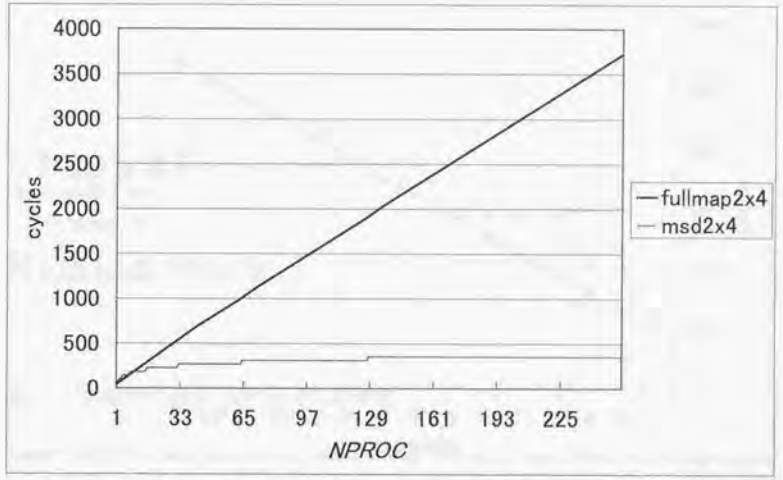


Figure 7.6: Fullmap vs. Hierarchical coarse directory, binary tree (WIDTH=4).

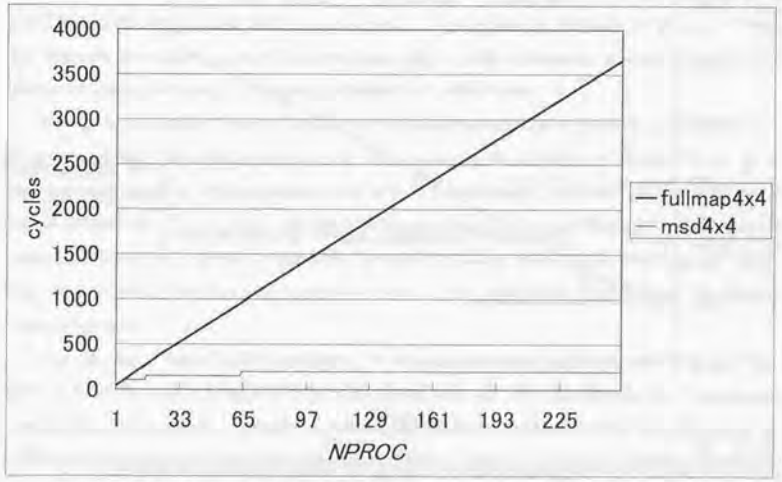


Figure 7.7: Fullmap vs. Hierarchical coarse directory, 4-ary tree (WIDTH=4).

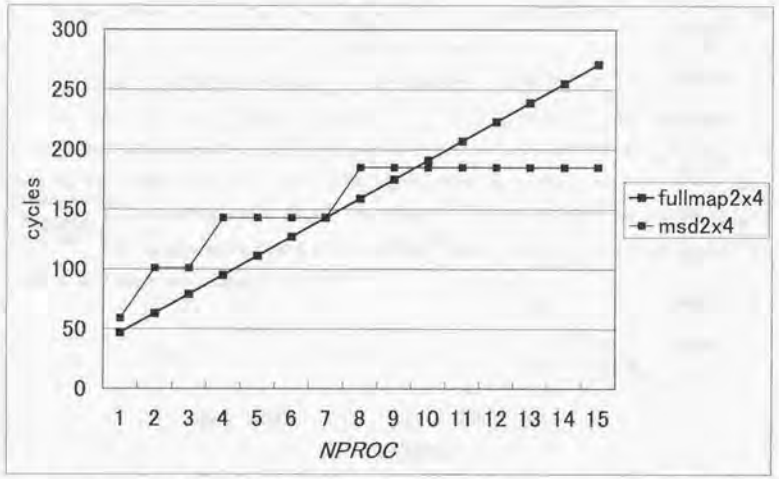


Figure 7.8: Magnification of Fullmap vs. Hierarchical coarse directory.

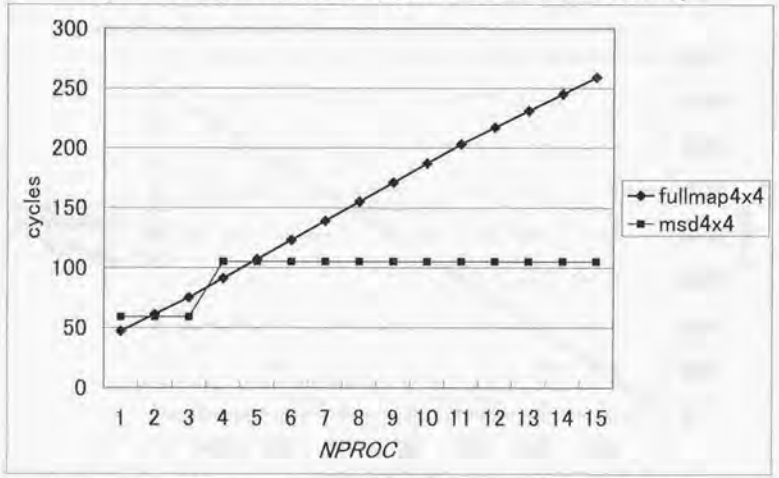


Figure 7.9: Magnification of Fullmap vs. Hierarchical coarse directory.

Chapter 8

Related Work

8.1 Hardware DSM systems

Hardware CC-NUMA systems are classified into two categories: those which have a protocol processor that is dedicated to cache management and those which don't.

For example, Stanford DASH [38], SGI Origin 2000 [33], MIT Alewife [3], KSR1 [22] and DDM [18] don't have protocol processors. DASH has a fullmap directory scheme and separate memories are used for the directory. Cache coherence is managed by a DC (directory controller) and RC (reply controller) in each cluster.

The DSM in Origin 2000 is almost the same as that in DASH, that is, uses a fullmap directory scheme. In configurations up to 16 processors, directories are located in the main memory, whereas, in configurations with over 16 processors, separate DRAMs are used for the directories. Each cluster includes two processors but is not bus-based SMP, that is, cache coherence in a cluster is managed by the directories, not by bus snooping. A "Hub" chip in a cluster is the directory controller and at the same time takes charge of normal memory access.

Alewife has a LimitLESS directory. A Communications and Memory Management Unit (CMMU) serves normal memory requests and controls directories and coherence transactions in hardware. Sparcle, a processing element, emulates a fullmap directory in software when there are more than five shared copies, which is a limited number. The paper [3] reports that about ten times as many cycles are required for software emulation as for

hardware-only processing. The system has separate DRAMs for the directories.

KSR1 provides hardware DSM by a COMA scheme. Directories are distributed in the fat-tree network hierarchy and managed by ALLCACHE Engines. The scheme requires directory access at every level in the hierarchy because of the hierarchical location of directories and therefore carries a large access cost.

DDM is another COMA system and its interconnection network is a hierarchical bus structure. Directories reside at each level of the hierarchy of the network.

Stanford FLASH [32] and Typhoon [49] have dedicated protocol processors. FLASH uses a fullmap scheme in which a home has linked lists of sharing processors locally. Directory information is stored in main memories, and therefore separate memory components don't exist. A protocol processor manages its protocols by executing handler routines. The MAGIC chip that includes the protocol processor is large scale since it includes internal cache memories. Typhoon has a protocol processor which includes a SPARC-core, and the protocol processor implements a LimitLESS directory in software.

Table 8.1 shows the structure of the above systems. The DSM scheme in this thesis is different from all of the above systems in that it stores the directory information in main memories, and simple hardwired logic in the memory controller manages caches in shared memory.

Table 8.1: Structure of DSM systems.

System	Directory scheme	Dedicated memory	Protocol processor
DASH	fullmap	yes	no
Origin 2000	fullmap	depend on configuration	no
Alewife	LimitLESS	yes	no
KSR1	hierarchical bit-map	yes	no
DDM	hierarchical bit-map	yes	no
FLASH	fullmap with linked lists	no	yes
Typhoon	LimitLESS	no	yes
Our system	hierarchical coarse directory	no	no

8.2 Combining schemes

Generalized combining is used in our hardware distributed shared memory system. Combining technique was first set up on the CHoPP [55] project. This design uses "repetition filter memories" (RFMs) which operate like a cache at each switching node. The reply to the first read request sets an entry in each RFM on the path between the source processor and target memory module, and subsequent read requests for the same address can be satisfied by the closest RFM. This design assumes that combining is done with requests for read-only data, and that more than one read request for a shared data would not traverse the network simultaneously. Eager Combining [8] uses a scheme that replicates a memory block in "server" nodes other than a home. Client nodes send requests for a block to the most nearest server selected statically. The server sends then sends the block to the requesters. Since the number of server nodes is fixed and server nodes are statically assigned, useless copies might be generated.

The NYU Ultracomputer [16] proposed a combining scheme based on a combining queue in a switch. In that design, messages can only be combined while they are passing through a combining queue (Figure 8.1). Since the NYU technique only combines 'messagess pairwise', that is, only two messages can be combined, the effect of combinings has a limit [35]. Moreove, each switching element in the interconnection network must have half as many comparators as there are entries in the combining queue, which is complicated, and thus requires a large amount of logic gates. The IBM RP3 [47] was designed with combining switches similar to those of NYU. The system, however, has two networks; one for low-latency non-combining messages, and the other for messages which are candidates of combining. The cost and size in the hardware of the combining switch were shown to be 6 to 32 times that for a non-combining switch.

SSS-MIN [50] has achieved low-cost combining of read requests and test and set, by using its bit-serial synchronized address tracing. The increase in hardware due to combining is 20%. This combining, however, can not improve the execution time in all applications because of its strict requirement for the arrival simultaneity.

To raise the rate of successful combining, Philip Bitar [9] proposes the "combining window". He claims that if each combinable message temporarily stayed in a buffering space

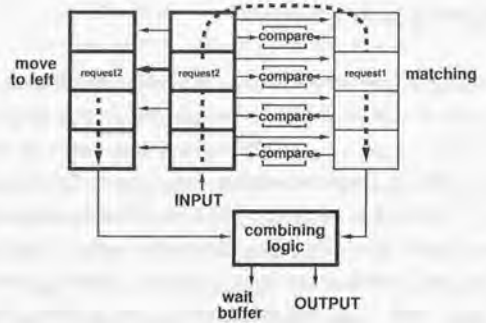


Figure 8.1: Combining queue.

at a switching node, the occurrence of combinings increases. The concept for implementing this waiting time at the switching node is the combining window. This approach might increase the rate of successful combining, but there is a danger that if enough combinings do not occur, each combinable message will have the overhead of an extra latency because of the waiting time.

In contrast with the above combining techniques, the technique outlined in this thesis only requires a small amount of extra logic and a matching memory, and switching nodes can combine any number of succeeding messages with preceding messages even after preceding messages have left the node. This can improve the success rate of combining.

8.3 Reduction of coherence messages

The STEM [25, 27] and GLOW kiloprocessor extensions [28, 29] on the Scalable Coherent Interface (SCI) [20] are examples of a system that uses a cache protocol with a combining scheme. Although the coherence management in STEM kiloprocessor extension is basically the same as the method in this thesis from the point of view of the use of multicasting and combining, the overheads of processing are not small since the tree structure is formed dynamically, a writing processor has to move to the root of the tree before it does writes,

and the directories are a hierarchical linked list because of the system's dependence on the SCI protocol, which leads to inefficiency of chained access to the directories. Similarly, GLOW kiloprocessor extension requires references to linked pointers in order to identify sharing processors and therefore has a large overhead. A request combining scheme is implemented in these extensions as a combining queue with comparators.

The bit-map pruning cache [44] is a scheme that dynamically cuts messages at network intermediate nodes. A hierarchical multicasting and combining schemes used in the system were originally proposed in [42], and are almost the same as the schemes described in this thesis. When acknowledgement or dummy acknowledgement messages for a memory address arrive at a node, a supplementary bit-map entry is generated or updated in the bit-map pruning cache, that is, an entry's field for the source direction in the message's arrival is set to one if the message is acknowledgement type, or the field is set to zero if the message is dummy acknowledgement type. Succeeding multicasting operations for the same memory address check the entry and multicast messages only in the directions with one in the entry. This scheme requires special memory for the bit-map pruning cache. If there are not an enough amount of memory at nodes, entries are easily discarded because of overflow and this scheme cannot have an effect.

Chapter 9

Conclusion

In large-scale parallel computer systems, distributed shared memory mechanisms are necessary for generality and convenience in the parallel programming. Efficient distributed shared memory requires caching of remote data, which introduces a cache consistency problem. A hardware scheme is effective in reducing the overheads incurred by the cache management of remote data. In this thesis, a lightweight hardware solution supported by hierarchical cache coherence management and generalized combining techniques has been described. The hierarchical management and generalized combining work in unison. Hierarchical coarse directory used for sharing information in our DSM system is smaller than any other directory scheme, since it uses hierarchical distance between processors as directory information and thus requires bits in proportion to $\log \log N$ when there are N processors in the system with a tree interconnection network. This allows reduction of the amount of memory used for the directories and the small overhead of the directory access because single access is adequate.

Overheads are raised if software execution on a dedicated protocol processor or a processing element is required for the communication processing and cache management. A dedicated protocol processor is eliminated and low latency of communication is realized by having a memory controller manage communication and coherence processing. We analyzed the frequency of memory reference in actual programs. The results show that the rate of reference to shared data is low relative to the total rate of memory reference. From the point of view of the results and the small size of hierarchical coarse directory,

low hardware costs can be achieved without large degradation of performance, by locating directories, tags and state information in DRAM main memory, instead of using separate dedicated fast memory (SRAM). As a result, only a small amount of hardwired logic circuits needs to be added to memory controllers and switching nodes in the interconnection network in order to provide distributed shared memory on a conventional distributed memory system. Although the simplicity of the cache management might increase the number of communication packets, dynamic multicasting and combining mechanisms on the interconnection network decrease the number of network massages, eliminates the necessity for sequential processing at a home node, and achieve fast execution. Even if there are many sharing processors, it is really feasible to employ an update protocol as well as an invalidate protocol due to the hierarchical coarse directory and message multicasting/combining mechanisms.

A prototype parallel computer OCHANOMIZ-5 implements the lightweight hardware distributed shared memory and generalized combining. The prototype machine is based on commercial processors and other controllers that are build in FPGAs. The mechanisms for the DSM are designed with a small amount of simple logic circuits. Performance of the prototype machine was evaluated by running several programs. In parallel calculation of the nth power of a matrix by eight processors, the generalized combining techniques reduced the calculation time by 7.7%. The results of the execution of the SPLASH-2 LU-Contig program on the lightweight hardware distributed shared memory system proposed in this thesis show that execution on eight processors reduces the execution time to 19% of that for a single processor. The lightweight hardware DSM has been shown to be effective and realizable by the development of a real hardware system.

Processing time required for coherence transactions on a large-scale system with hardware DSM mechanisms has been measured by using times required at each components derived from operation of the prototype machine. In a fullmap directory system, the time required for completion of a coherence transaction is proportional to the number of processors that have a copy of a block. On the other hand, in a hierarchical coarse directory system, a linear increase in the number of processors with a copy produces a logarithmic increase in processing time. The results show that coherence transactions in the hierarchical coarse directory system are much faster than in the fullmap system when there are

many sharing processors. From the results, the method is a good prospect as the basis for hardware DSM on a large-scale system.

Existing hardware distributed shared memory systems have complicated mechanisms and is thus expensive. Although the systems indeed deliver high performance when they appear, fast evolution of commercial processors makes the life short in terms of advanced and latest processing power. Recently, workstation or PC clusters are attracting a great deal of attention for parallel and distributed computing from the point of view of cost performance. These systems are constructed by using commercial workstations or PCs that are connected by commercial networks such as Ethernet. These systems don't have special mechanisms for distributed shared memory, and communications between processing nodes are performed through not only cables and switches but also IO buses (for example, a PCI bus) in source and destination processing nodes. The use of IO bus processing generally prevents the network packets of being quickly dispatched at the source or reflected in the destination memory, since the IO bus protocol is specialized for IO devices, it is separated from memory bus, and two bus transactions therefore must be processed to transfer data.

The feature that simple hardware can manage cache coherence in DSM has a possibility that cost-effective DSM systems can be constructed when the mechanisms are added to commercial PC, workstations and network switches.

Appendix A

Parallel Computer Prototype OCHANOMIZ-5

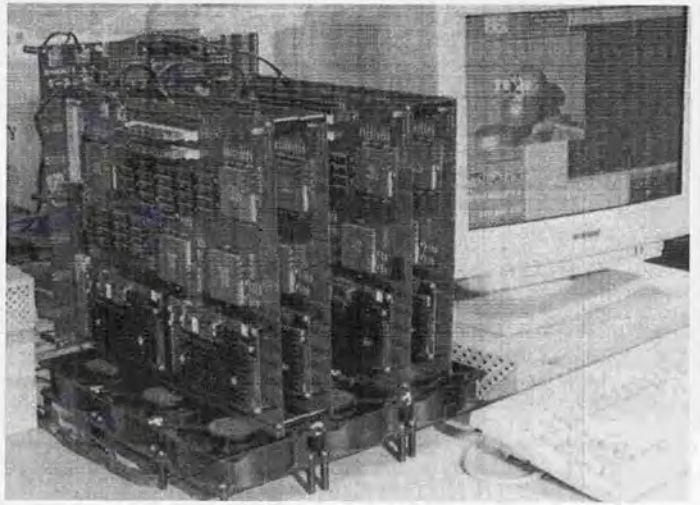


Figure A.1: OCHANOMIZ5.

OCHANOMIZ-5 (Omnipotent Concurrency-Handling Architecture with Novel OptiMIZers-5) is a prototype parallel computer that has developed to verify new mechanisms for large-scale systems. Hierarchical structure for scalability, hardware distributed shared memory, processor-based synchronization mechanisms, and a highly functional interconnection network are designed in the machine. Flexible tests and evaluation can be done by using reconfigurable FPGAs. Figure A.1 is the picture of OCHANOMIZ-5.

A.1 Print circuit boards

PWS/CR3000/ of ZUKEN inc.[68] was used when designing the OCHANOMIZ-5 boards. CR3000 is a Computer Aided Design (CAD) tool for print circuit boards. Each board consists of six layers; four for signal lines and two for power and ground. 96 pin standard DIN connectors are used to connect cluster boards to a network board. Figure A.2 and A.3 show the cluster board and the network board, respectively. Figure A.4 and A.5 are their pictures. Table A.1 and A.2 show the photo data lists about signal lines in the cluster board and the network board, respectively. Of six layers, layer 3 and 4 are for power and ground.

Table A.1: Photo data list of cluster board.

Layer No.	Number of lines	Total length of all lines (mm)
1	3,449	35,298
2	3,263	34,834
5	2,913	26,727
6	3,124	39,738

Table A.2: Photo data list of network board.

Layer No.	Number of lines	Total length of all lines (mm)
1	10,018	17,452
2	5,244	14,546
5	4,836	10,463
6	5,831	19,216

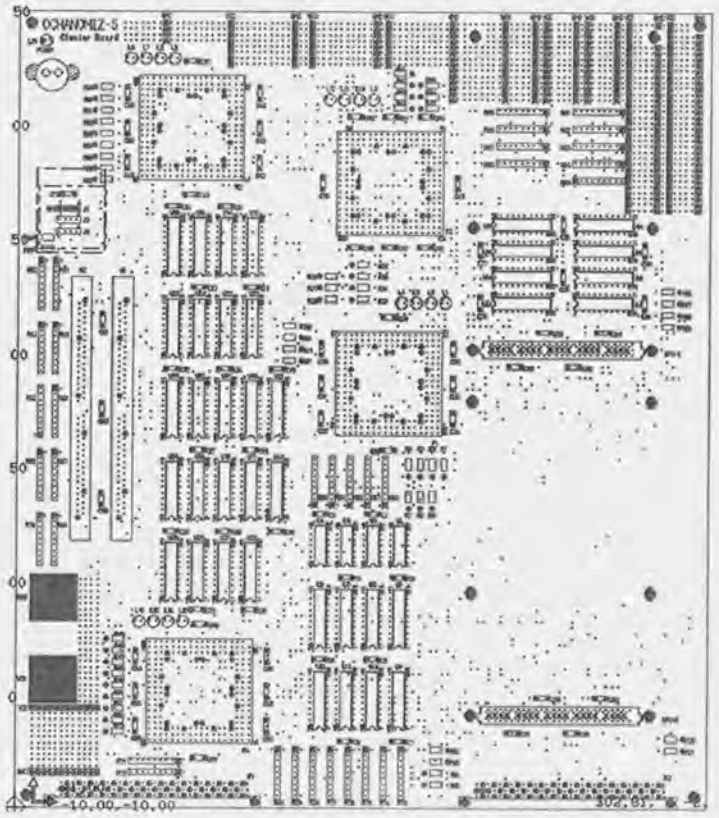


Figure A.2: Cluster board (29.972 cm × 34.798 cm).

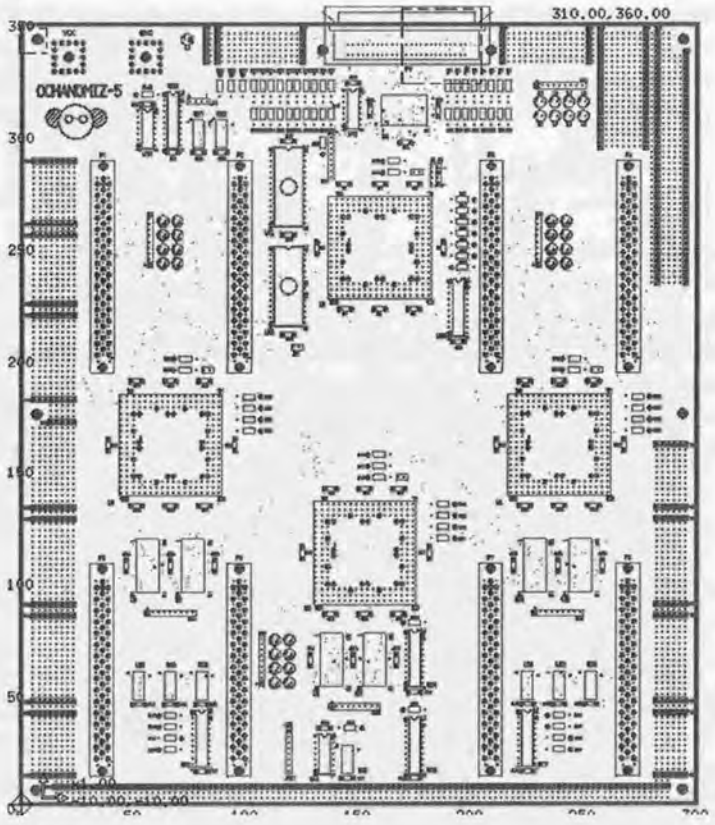


Figure A.3: Network board (29.972 cm × 34.798 cm).

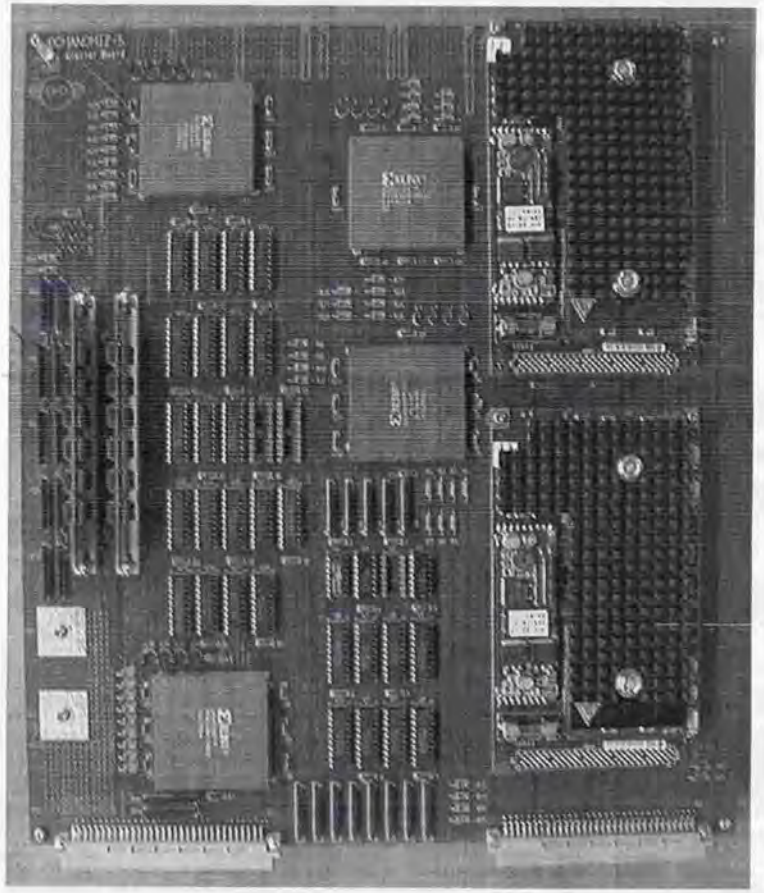


Figure A.4: Picture of cluster board.

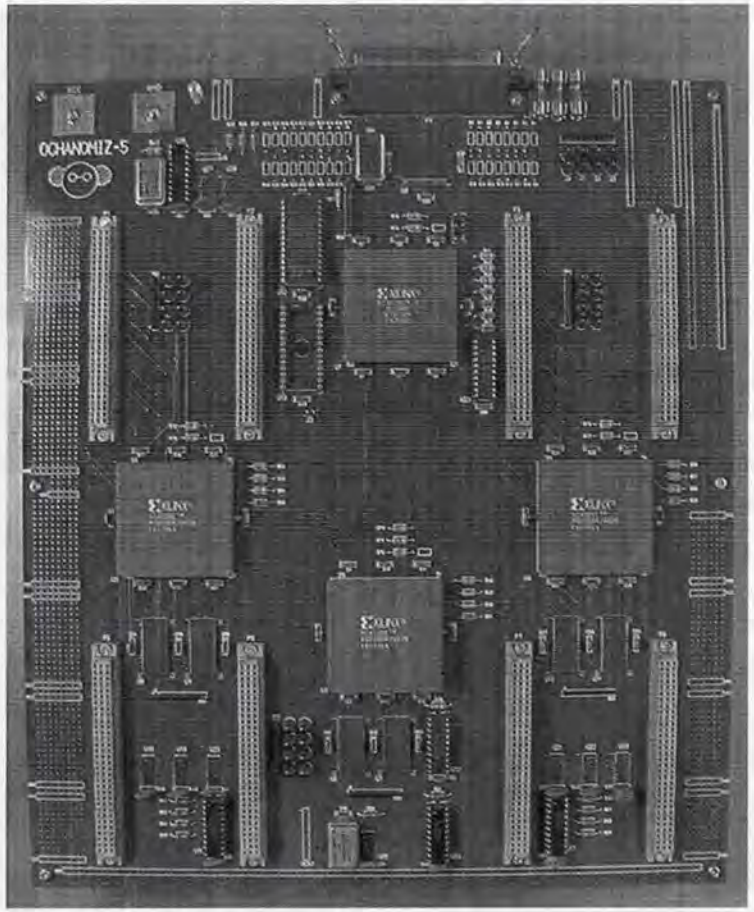


Figure A.5: Picture of network board.

A.2 Architecture of cluster board

In each cluster, there are two processing elements (SuperSPARC+ [56]), main memories, a memory controller, a bus arbiter, and a network interface. The processing element is a module which includes a CPU, secondary cache, and a cache controller, and running at 60 MHz. The module is designed to be connected directly to the SPARC standard memory bus, MBus [53].

MBus is used for the shared bus in each cluster. Each cluster, therefore, is a multiprocessor with the shared bus. MBus is a bus with 64-bit lines used for address and data by time sharing. The cache consistency protocol is based on write invalidation. Two processing elements and a network interface can be a bus master. The MBus arbitration scheme assumes a central arbiter. The protocol used is based on "bus parking" [53].

Each secondary cache in the processing element is 1Mbyte, and main memory in a cluster is 32Mbyte. Main memory is composed of DRAM modules. Other components, memory controller, bus arbiter and network interface, are built in FPGAs from Xilinx Inc. The memory controller and bus arbiter are on XC4010s, and the network interface is on a XC4025.

A.3 Architecture of network board

The interconnection network of OCHANOMIZ-5 is a hierarchical binary tree. Each internal switching node includes a $4 \times h4$ cross-bar switch connecting a parent node, two child nodes and a combining unit. It is built on a FPGA (XC4025).

There are two separate paths between each parent and child node, that is, one upward, and the other downward. Each path is 8 bits wide. Each switching node has two SRAMs (total: 128Kbyte) at its side, which is used as the wait buffer when generalized combining is valid.

In addition, the network board includes a controller ($\mu PD72611$ by NEC [45]) based on the SCSI-2 standard as an interface between OCHANOMIZ-5 and a host computer. An eight-bit processor, COSC (Controller Of SCSI Controller) in a FPGA, operates the controller.

A.4 Basic performance evaluation

A.4.1 Ray tracing

In order to measure the basic performance of OCHANOMIZ-5, we used a ray tracing program as an application. The program is a C language program transplanted from the original program written in Basic language [63]. As a target model, a 450 by 576 pixels picture was computed. The picture is shown in Figure A.6. Table A.3 shows the number of times each type of floating point operation appears in the total execution.

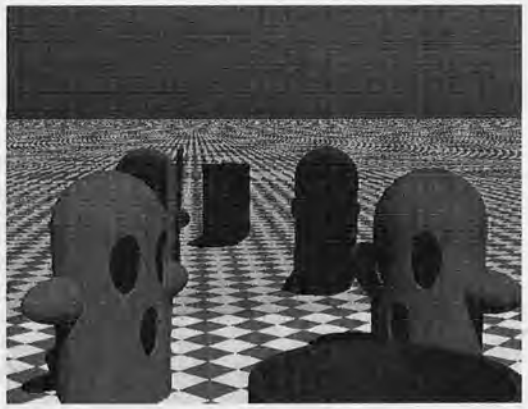


Figure A.6: Ray tracing model.

Figure A.7 shows the total execution time. The vertical axis indicates execution time, and the horizontal axis indicates the number of processors which participated in the parallel execution. Moreover, two dots (SS-10 and SS-20) in this graph show the results when the same program was executed on two different workstations, SPARCstation10 (50MHz) and SPARCstation20 (75MHz) respectively. From this graph, an almost perfect effect of parallelization is seen, from one to eight processors.

Table A.3: The number of floating point operations.

Operation	Times
multiply	240,112,818
division	9,489,750
addition	96,541,114
subtraction	66,809,162
square root	2,167,476
total	415,120,300

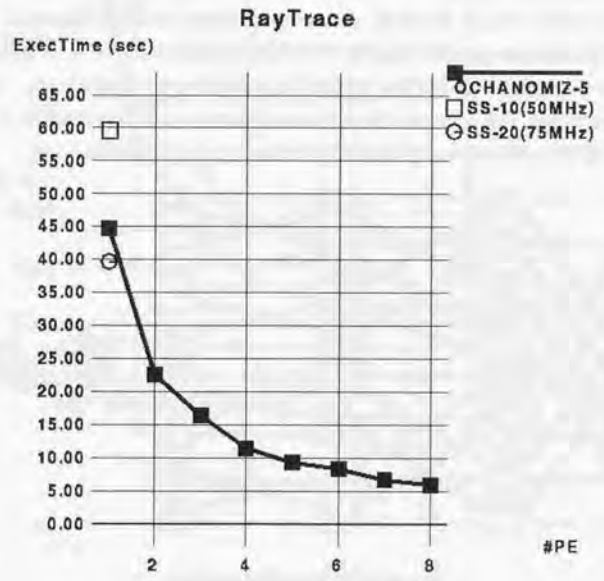


Figure A.7: Execution time for ray tracing.

A.4.2 FFT

As the next example, we show the execution time of a fast Fourier transform (FFT) program. A butterfly algorithm (Figure A.8) is used in the execution.

In advance, N/M data are allocated to each processor. N is the total size of data and M is the number of processors. The first $\log M$ steps of butterfly operations require communications between processors. Here, we used remote write requests to main memory in the cluster in which the partner exists, and flag variables to inform the partner of completion of the data sending. In the latter $\log N - \log M$ steps, each processor executes only with local data.

The program was executed for $N = 2^{14}$. The result is shown in Figure A.9. The two axes have the same meanings as for ray tracing. For two processor execution, we used two processors in one cluster. Similarly, we used two clusters in four's execution. Here, the results of the same program's execution on AP1000+ [19] is included in the graph.

When comparing two's and four's execution, the latter takes a little longer than half the time of the former. This is because there is inter-cluster communication in four's execution. We can say that sufficiently good results were got from parallelization.

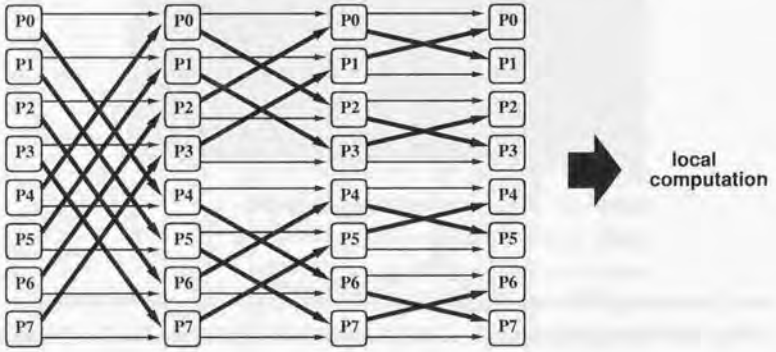


Figure A.8: Butterfly execution.

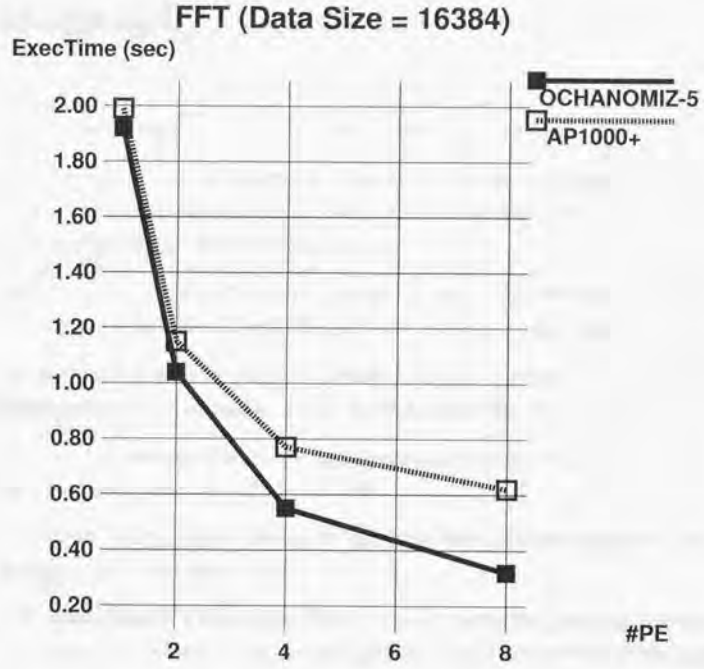


Figure A.9: Execution time for FFT.

Bibliography

[1] *The Massively Parallel Processing System JUMP-1*. Ohmsha, Lmt., 1996.

[2] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proc. of the 2nd Int'l Symp. HPCA*, February 1996.

[3] A. Agarwal, R. Bianchini, D. Chaiken, and K. L. Johnson. The MIT Alewife Machine: Architecture and Performance. In *Proc. of ISCA*, pages 2–13, June 1995.

[4] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proc. of ISCA*, pages 280–289, June 1988.

[5] K. E. Batcher. Sorting networks and their applications. *AFIPS Spring Joint Computing Conference*, pages 307–314, April 1968.

[6] V. E. Benes. Mathematical Theory of Connecting Networks and Telephone Traffic. *Academic Press, New Yourk*, 1965.

[7] B. N. Bershad and M. J. Zekauskas. Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. In *Carnegie Mellon University Technical Report CMU-CS-91-170*, September 1991.

[8] R. Bianchini and T. J. LeBlanc. Eager Combining: A Coherency Protocol for Increasing Effective Network and Memory Bandwidth in Shared-Memory. *Proc. of 6th IEEE Symp. Parallel and Distributed Processing*, October 1994.

[9] P. Bitar. Combining Window: The Key to Managing MIMD Combining Trees. *The Workshop on Scalable Shared Memory Multiprocessors*, May 1990.

[10] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of 13th ACM Symp. Operating System Principles*, October 1991.

[11] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Trans. Computers*, pages 1112–1118, December 1978.

[12] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proc. of Int'l Conf. ASPLOS*, pages 224–234, April 1991.

[13] A. Charlesworth, N. Aneshansley, M. Haakemeetster, D. Drogichen, G. Gilbert, R. Williams, and A. Phelps. The Starfire SMP Interconnect. In *Proc. of Int'l Conf. Supercomputing*, page CDROM, October 1997.

[14] A. L. Cox, S. Dwarkadas, and P. Keleher. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proc. of the 21st Symp. Computer Architecture*, pages 106–117, April 1994.

[15] R. Cytron, S. Karlovsky, and K. P. McAuliffe. Automatic Management of Programmable Caches. In *Proc. of ICPP*, pages 229–238, August 1988.

[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] A. Gupta, W. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proc. of ICPP*, pages I–312–321, August 1990.

[18] E. Hagersten, A. Landin, and S. Haridi. DDM – A Cache-Only Memory Architecture. *Computer*, pages 44–54, September 1992.

[19] 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.

[20] IEEE. *IEEE Standard for Scalable Coherent Interface (SCI) 1596–1992*, 1993.

[21] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proc. of the 2nd Int'l Symp. HPCA*, February 1996.

[22] H. Burkhardt III, S. Frank, and J. Rothnie. Overview of the KSR1 Computer System. *Technical Report KSR-TR-9202001, Kendall Square Research*, February 1992.

[23] T. Inagaki, J. Niwa, T. Matsumoto, and K. Hiraki. Supporting Software Distributed Shared Memory with an Optimizing Compiler. In *Proc. of ICPP*, pages 225–234, August 1998.

[24] D. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. Distibuted–Directory Scheme: Scalable Coherent Interface. *Computer*, pages 74–77, June 1990.

[25] R. E. Johnson. Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors. *PhD Thesis*, 1993.

[26] K. Katoh, H. Nishi, Y. Yang, and H. Amano. On Design of a Router for the Interconnection network RDT. *IEICE Technical Report*, 93(320):49–56, November 1993. (In Japanese).

[27] S. Kaxiras. Kiloprocessor Extensions to SCI. In *Proc. of 10th Int'l Parallel Processing Symposium*, April 1996.

[28] S. Kaxiras and J. R. Goodman. The GLOW Cache Coherence Protocol Extension for Widely Shared Data. In *Proc. of Int'l Conf. Supercomputing*, pages 35–43, May 1996.

[29] S. Kaxiras and J. R. Goodman. A Study of Three Dynamic Approaches to Handle Widely Shared Data in Shared-Memory Multiprocessors. In *Proc. of Int'l Conf. Supercomputing*, pages 457–464, July 1998.

[30] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of ISCA*, pages 13–21, May 1992.

[31] J. Konicek, T. Tilton, A. Veidenbaum, C. Q. Zhu, E. S. Davidson, R. Downing, M. Haney, M. Sharma, P. C. Yew, P. M Farmwald, D. Kuch, D. lavery, R. Lindsey, D. Pointer, J. Andrews, T. Beck, T. Murphy, S. Turner, and N. Warter. The Organization of the Ceder System. In *Proc. of ICPP*, pages I–49–56, August 1991.

[32] 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 ISCA*, pages 302–313, April 1994.

[33] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. of ISCA*, pages 241–251, June 1997.

[34] D. H. Lawrie. Access and Alignment of Data in an Array Processor. In *IEEE Trans. Computers*, pages 1145–1155, December 1975.

[35] G. Lee, C. P. Kruskal, and D. J. Kuck. On the Effectiveness of Combining in Resolving 'Hot Spot' Contention. *Journal of Parallel and Distributed Computing*, 20(2):136–144, February 1994.

[36] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Trans. Computers*, C-34(10):892–901, October 1985.

[37] C. E. Leiserson. The Network Architecture of Connection Machine CM-5. In *Proc. of 4th Ann. ACM Symp. Parallel Algorithms and Architectures*, pages 272–285, May 1992.

[38] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for DASH Multiprocessor. In *Proc. of ISCA*, pages 148–159, May 1990.

[39] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of ICPP*, pages 94–101, August 1988.

[40] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Computer Systems*, 7(4):321–359, November 1989.

[41] D. J. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys*, 3(25):303–338, September 1993.

[42] T. Matsumoto and K. Hiraki. A Shared Memory Architecture for Massively Parallel Computer Systems. *IEICE Japan SIG Reports*, 92(173):47–55, August 1992. (In Japanese).

[43] T. Matsumoto and K. Hiraki. Extended Snoopy Spin Wait and Hierarchical Elastic Barrier. *Ann. Convention Record of IPSJ*, pages 43–44, October 1993. (In Japanese).

[44] T. Matsumoto, T. Kudoh, E. Nishimura, K. Hiraki, H. Amano, and H. Tanaka. Distributed Shared Memory Architecture for JUMP-1 a General-Purpose MPP Prototype. In *Proc. of I-SPAN*, pages 131–137, June 1996.

[45] NEC Corp. *Data Sheet: MOS Integrated Ciruit μPD72611*, 1992. (in Japanese).

[46] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki. Evaluation of Compiler-Assisted Software DSM Schemes for a Workstation Cluster. In *Proc. of the 1999 IWIA*, November 2000. Accepted for publication.

[47] G. F. Pfister and V. A. Norton. Hot Spot Contention and Combining in Multistage Interconnection Networks. *IEEE Trans. Computers*, pages 943–948, October 1985.

[48] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technology*, 4(2):63–79, summer 1996.

[49] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. of ISCA*, April 1994.

[50] M. Sasahara, J. Terada, L. Zhou, K. Gaye, J. Yamato, S. Ogura, and H. Amano. SNAIL: A Multiprocessor Based on the Simple Serial Synchronized Multistage Interconnection Network Architecture. In *Proc. of ICPP*, volume 1, pages I–117–1120, August 1994.

[51] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software–Only Approach for Supporting Fine–Grain Shared Memory. In *Proc. of the 7th Int'l Conf. ASPLOS*, pages 174–185, October 1996.

[52] A. J. Smith. CPU Cache Consistency with Software Support and Using One Time Identifiers. In *Proc. of the Pacific Computer Communications Symposium*, October 1985.

[53] SPARC International. *SPARC Mbus Interface Specification*, April 1991.

[54] SPARC International, Inc. *The SPARC Architecture Manual Version 8*, 1992.

[55] H. Sullivan and T. R. Bashkow. A Large Scale, Homogenous, Fully Distributed Parallel Machine, I. In *Proc. of 4th Symp. Computer Architecture*, pages 105–117, March 1977.

[56] Sun Microsystems, Inc. *SuperSPARC & MultiCache Controller User's Manual*, 1994.

[57] K. Tanaka, T. Matsumoto, and K. Hiraki. Distributed Shared Memory with Lightweight Hardware. *Trans. IPSJ*, 40(5):2025–2036, May 1999. (In Japanese).

[58] K. Tanaka, T. Matsumoto, and K. Hiraki. Lightweight Hardware Distributed Shared Memory Supported by Generalized Combining. In *Proc. of the 5th Int'l Symp. HPCA*, pages 90–99, January 1999.

[59] K. Tanaka, T. Matsumoto, J. Tsuiki, and K. Hiraki. Low Cost Hardware Distributed Shared Memory. *IPSJ SIG, ARC*, 97(102):79–84, October 1997. (In Japanese).

[60] K. Tanaka, J. Tsuiki, T. Matsumoto, and K. Hiraki. Parallel Computer Prototype OCHANOMIZ-5. In *The 3rd FPGA/PLD Design Conference & Exhibit*, pages 505–514, July 1995. (In Japanese).

[61] K. Tanaka, J. Tsuiki, T. Matsumoto, and K. Hiraki. Generalized Combining. *IPSJ SIG, ARC*, 96(13):31–36, January 1996. (In Japanese).

[62] M. Thapar and B. Delagi. Distributed–Directory Scheme: Stanford Distributed Directory Protocol. *Computer*, pages 78–80, June 1990.

[63] T.Yamamoto. *The 3 Dimensional Computer Graphics*. CQ publication, 1983.

[64] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of ISCA*, pages 24–36, June 1995.

[65] A. Y. Wu. Embedding of Tree Networks into Hypercubes. *J. Parallel and Distributed computing*, 2:238–249, 1985.

[66] XILINX, Inc. *The Programmable Logic Data Book*, 1994.

[67] A. S. Youssef and B. Narahari. The Banyan-Hypercube Networks. *IEEE Trans. Parallel and Distributed Systems*, 1(2):160–169, April 1990.

[68] ZUKEN. *REFERENCE MANUAL vol.1-10*. (in Japanese).

超並ハードウェアによる分散共有メモリの研究