A DSM Architecture for a Parallel Computer Cenju-4

Takeo Hosomi, Yasushi Kanoh, Masaaki Nakamura, Tetsuya Hirose
C&C Media Research Laboratories, NEC Corporation
4-1-1 Miyazaki Miyamae-ku Kawasaki Kanagawa 216-8555 JAPAN
hosomi@ccm.cl.nec.co.jp

Abstract

A parallel computer Cenju-4 is a cache-coherent non-uniform memory access (ccNUMA) multiprocessor and designed to be scalable up to 1024 nodes. For scalability, Cenju-4 adopts a bit-pattern directory. This scheme enables more precise representation than other imprecise schemes, such as a coarse vector scheme. Cenju-4 utilizes multicast and gathering functions of the network for delivering invalidation request messages and for collecting replies. This enables store access latency to be scalable, even when the block is shared among all nodes. Cenju-4 also prevents starvation and deadlock by queuing certain types of messages in the main memory. This enables a full solution to the starvation problem with centralized directory scheme, and to the deadlock problem with one physical or virtual network. The buffer sizes required for queuing messages at each node are only 32K bytes and two 64K bytes on a 1024-node system.

In this paper, we present the design of the DSM architecture and some performance results.

1. Introduction

A parallel computer Cenju-4 is a non-uniform memory access (NUMA) multiprocessor and consists of up to 1024 nodes. The goal of Cenju-4 is to combine the ease of programming of Symmetrical Multi-Processor (SMP) systems with the high performance of Massively Parallel Processor (MPP) systems. This is achieved by means of hardware support of both message passing and distributed shared memory (DSM). By using both in combination, users can write parallel programs more flexibly and attain higher performance.

In this paper, we present DSM architecture of Cenju-4. On a large system, DSM is required to be scalable in both performance and hardware cost. It is also an important requirement that a system guarantees forward progress of memory access to DSM, i.e., shared-memory access will finish in finite time.

Cenju-4 implements the DSM with the use of coherent caches and a directory-based coherence protocol.

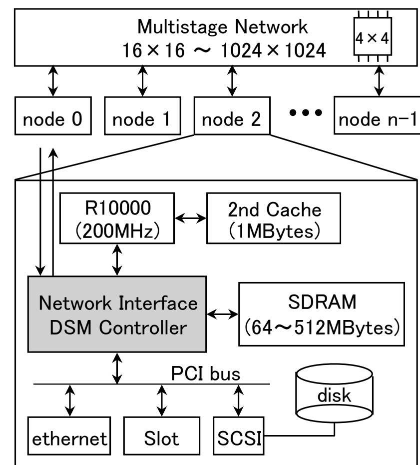

Figure 1. Cenju-4 architecture overview

In order to achieve the above-mentioned requirements for a large DSM system, Cenju-4 has the following four characteristics:

- A directory which dynamically switches its representation from a pointer structure and a bit-pattern structure
- Multicast and gathering functions of a network
- A cache coherence protocol that prevents starvation
- A deadlock-free mechanism with one network

In order to achieve hardware-cost scalability, the memory requirement for the directory that keeps a record of all nodes caching a block should not increase with the number of nodes. Cenju-4 adopts a directory scheme which dynamically switches its representation from a pointer structure to a bit-pattern structure. This scheme achieves the constant memory requirement and an efficient record of nodes.

In order to achieve performance scalability, an increase in the number of nodes must not degrade memory access latency in direct proportion to the increase. When a node issues a store access to a memory block

shared by a number of nodes, it is necessary for invalidation requests to be sent to all nodes caching the memory block and for replies to be collected from those nodes to maintain coherency. This implies that store access latency may increase with the number of nodes. Cenju-4 avoids such an increase in store access latency by utilizing multicast and gathering functions of the network to deliver requests and to collect replies. Cenju-4 also adopts a directory which can specify all nodes caching a block with one memory access.

Cenju-4 guarantees forward progress of shared-memory access by preventing starvation and deadlock. In order to prevent starvation, Cenju-4 adopts a blocking protocol for cache coherence: requests which can not be processed immediately are queued in the main memory for later processing. The size of buffer required for queuing requests is 32K bytes in a 1024-node system. In order to prevent deadlock, Cenju-4 has a mechanism which queues certain types of messages for cache coherence in the main memory. This scheme is implementable without multiple physical or virtual channels in the network. The buffer size required for queuing messages is 128K bytes in a 1024-node system.

The rest of the paper is organized as follows: the next section gives an overview of the architecture of Cenju-4. Section 3 describes the implementation of DSM on Cenju-4. Section 4 presents DSM system performance with respect to memory access latencies and parallel applications (NAS Parallel Benchmarks V2.3).

2. Cenju-4 Architecture

Figure 1 illustrates Cenju-4 in block diagram form.

Cenju-4 contains a multi-stage network that can connect up to 1024 nodes. This network is constructed by crossbar switches which have four input ports and four output ports. The network includes the following features:

- In-order message delivery between any two nodes
- Support for the multicast and gathering functions
- Freedom from deadlock

Cenju-4 is a NUMA multiprocessor. Each node consists of one R10000 [11] processor, a 1M-byte secondary cache controlled by the processor, a main memory up to 512M bytes, a PCI bus and a controller chip. The controller chip uses the network for user level message passing and for DSM access.

The operating system manages the main memories, each of which can be accessed either as a private memory by its own processor or as a DSM by all processors. The system distinguishes each attempted access by the MSB of its 40-bit physical address. Only 29 offset bits are used for access to private memory. When an access to a main memory is shared, 10 bits are used as the node number of the main memory and 29 bits are used as offset.

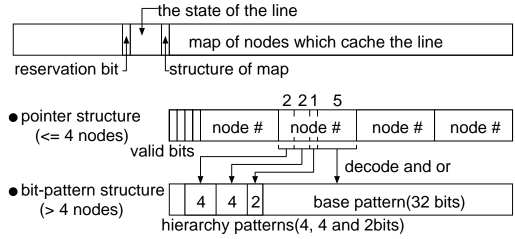

Figure 2. Directory entry

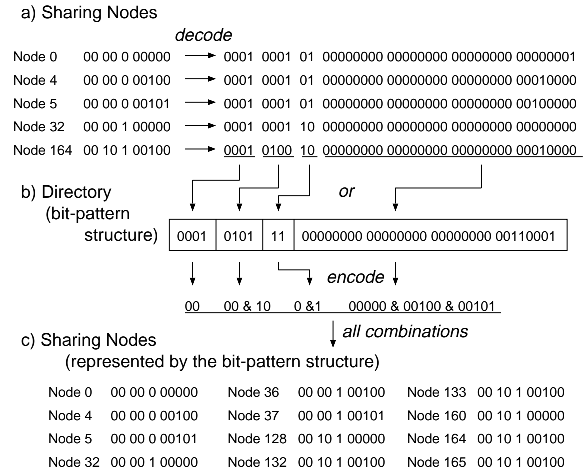

Figure 3. Bit-pattern structure

Cenju-4 supports shared memory and message passing using private memory. Details of the user-level message passing mechanism are described in [6].

3. DSM Architecture

The DSM architecture of Cenju-4 is provided through the use of coherent caches and a directory-based invalidation protocol. In this section, we describe the features of the DSM architecture. Section 3.1 explores the directory which dynamically switches its representation from a pointer structure to a bit-pattern structure. Section 3.2 describes multicast and gathering functions of the network. Section 3.3 presents the cache coherence protocol that prevents starvation. Section 3.4 explains the deadlock-free mechanism with one network.

3.1. Directory

Several directory schemes which are implementable with scalable hardware cost have been devised previously. We show the characteristics of these directory schemes in Table 1. Unfortunately, most of them are not scalable in performance when the directory is accessed to specify all nodes caching a block.

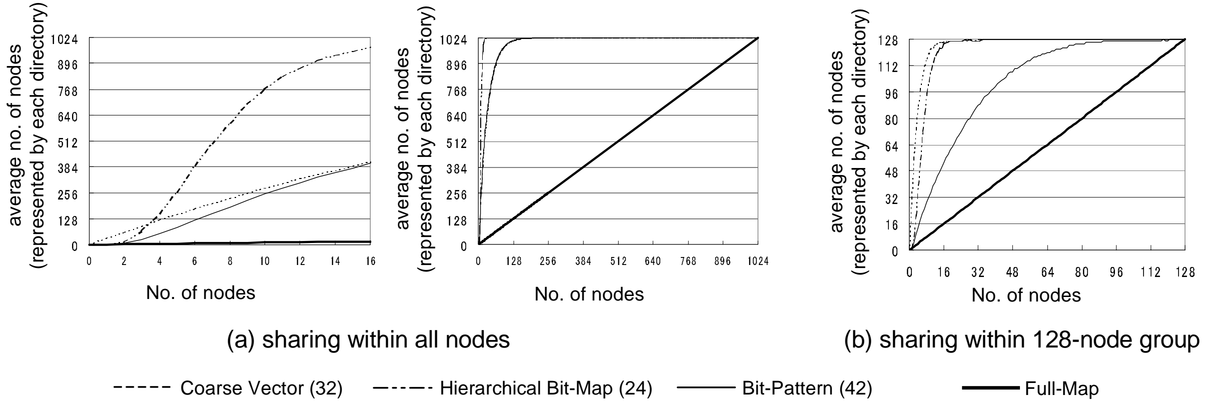

Figure 4. Behavior of imprecise node maps (in a 1024-node system)

Table 1. Characteristics of directory schemes

	scalability	
	hardware cost	access cost
Full Map[2]	×	×
Chained[5]	○	×
LimitLESS[3]	○	×
Dynamic Pointer[12]	○	×
Origin†[8]	○	○
Cenju-4‡	○	○

†: Full Map + Coarse Vector[4]
‡: Pointer + Bit Pattern

Cenju-4 adopts a directory scheme which dynamically switches its representation from a pointer structure to a bit-pattern structure. This scheme achieves scalability in both hardware and performance.

Each cache line size block (128 bytes) of a main memory is associated with a 64-bit directory entry. Figure 2 illustrates the directory entry of Cenju-4. The directory occupies 1/16 of the main memory. The size of the directory does not increase with the number of nodes.

The directory entry contains a reservation bit, the state of the block, and a record of nodes caching the block. We call this record a node map. We will explain the reservation bit and the state with the cache coherence protocol in a later section. In this section, we explain the node map.

The node map in Cenju-4 is similar to the limited pointer schemes[3][4] that dynamically switch their representations from a pointer structure to a different type of structure. In these schemes, in the most common case of a block being shared among a small number of nodes, the directory is maintained in a structure comprising several pointers (four pointers on Cenju-4), called a 'pointer structure'. When the number of nodes sharing a block exceeds the number of pointers available, the directory switches its representations to a different type of structure, such as a coarse vector structure. Cenju-4, however, adopts a 'bit-pattern structure' that is more suitable for maintaining a record of a large number of nodes than a coarse vector scheme. This bit-pattern structure is similar to a hierarchical bit-map scheme[10]. This hierarchical bit-map scheme strongly depends on the hierarchical structure of a network, and this negatively influences the preciseness of the node map.

In the bit-pattern structure, a node is represented by four, four, two and thirty-two bit fields, which are made by encoding two, two, one and five bits of a 10-bit node number. The values found by logical OR operations on all sharing nodes are stored in the node map. Figure 3 shows one example. If nodes 0, 4, 5, 32 and 164 cache the block, the values shown in Figure 3(b) are stored. This structure does not depend on the structure of the network.

With this scheme, the node map keeps a precise record of nodes in:

(a) Memory blocks which are shared by a number of nodes less than or equal to four even in a 1024-node system with a pointer structure

(b) All memory blocks in systems of 32 nodes or less with a bit-pattern structure

Preciseness is lost in other cases represented by a bit-pattern structure. In the case shown in Figure 3, even though the actual number of sharing nodes is five, the directory represents that twelve nodes share the block.

Figure 4 shows the behavior of three imprecise directory schemes: a coarse vector scheme[4], a hierarchical bit-map scheme, and a bit-pattern scheme. In the coarse vector scheme, nodes are divided into several groups. When representing 1024 nodes with a 32-bit coarse vector structure, all nodes are divided into 32 groups. Each of the 32 bits represents a group of 32 nodes. In the hierarchical bit-map scheme, the node map consists of six 4-bit fields, since the network consists of a six level quadruple-tree structure. Each of the six fields represents a level of the tree structure. And each of the four bits represents a branch of the

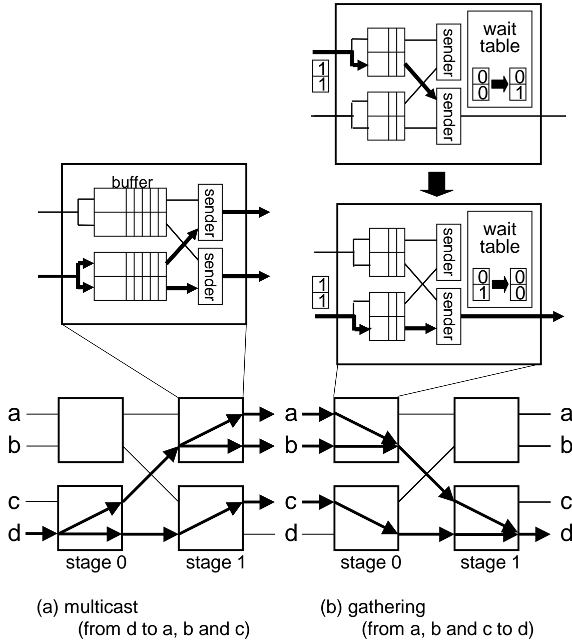

Figure 5. Multicast and gathering functions of the network

tree structure. The same 4-bit field is used at switches of the same level. In Figure 4, we compare the 32-bit coarse vector structure, a 24-bit hierarchical bit-map structure, and a 42-bit bit-pattern structure. Though the numbers of bits used for these schemes are different from each other, they are under the equal condition; the maximum number of bits that can be used for the node map is 59 bits on Cenju-4.

The figure shows that the average number of nodes represented by each directory scheme varies as the number of nodes sharing a block varies. In Figure 4(a), the sharers were chosen from 1024 nodes. Even though there is no difference between three schemes with a large number of nodes, the bit-pattern structure performs well with a small number of nodes. In Figure 4(b), the sharers were chosen from a 128-node group. The average number of nodes represented by the bit-pattern scheme is much smaller than that of a coarse vector scheme or that of a hierarchical bit-map scheme. This implies that the bit-pattern structure is advantageous in a multi-user environment, where a large system might be divided among several programs.

3.2. Multicast and Gathering Functions

Cenju-4 utilizes the multicast and gathering functions of the network to deliver invalidation request messages and to collect their reply messages. If there is no multicast function, a node sends invalidation request messages to nodes represented by the node map through the use of many singlecast messages. If there is no gathering function, a node receives all messages sent from the nodes. In both cases, the latency of the transaction of invalidating cache copies becomes unscalable. Moreover, sending invalidation messages and receiving reply messages make hot spots, and this negatively influences the latencies of other memory accesses.

Figure 5(a) shows the concept of the multicast function of the network. The multicast function is used to transfer invalidation request messages to the nodes represented by the node map of the directory. A pointer structure and a bit-pattern structure to specify the multicast destination similar to those of the node map are employed. Coinciding the specifications of the multicast destination with the directory structures prevents messages from being delivered to any nodes not represented by the node map. The switches in the network find out which ports to output the messages to by their own position information in the network, the system size, and the multicast destination specified by either the pointer structure or the bit-pattern structure in the message. Calculation in the switch makes it possible to support a multicast pattern which is not dependent on the network structure.

To realize a multicast function, we had to avoid the deadlock that occurs in arbitration when two switches try to send multicast messages into the same switch simultaneously. In order to avoid such deadlock, we adopt a crosspoint buffer scheme with which it is unnecessary to arbitrate among switches when sending messages, and a virtual cut through flow control. As shown in Figure 5(a), four buffers are needed for a 2 $\times$ 2 switch with a crosspoint buffer scheme, thus sixteen buffers are needed for the 4 $\times$ 4 switch in Cenju-4.

Figure 5(b) shows the concept of the gathering function of the network. The gathering function is used to gather reply messages corresponding to the multicasted invalidation messages. Each node specifies a wait pattern at each switch that a message passes through in the gathered message. The wait pattern is calculated from the destination node numbers of the multicasted invalidation message, the destination node number of the gathered reply message, the own node number, and the system size. Each gathered message has a 10-bit identifier to enable different gathering to be distinguished from each other. The switch has a 1024-entry table to record a wait pattern for each gathering. The table occupies only 3.6% of the switch chip in gates.

When a gathered message arrives, the switch checks the entry indexed by the identifier in the message. If it is an initially gathered message with its identifier, the switch resets the bit of the input port from the wait pattern and sets it to the entry. The message is removed from the buffer and is not sent to the next switch. When the following gathered messages arrive, the switch resets the bit of the input port from the wait pattern in the entry, and all the messages are removed except the last gathered message. Only the last gath-

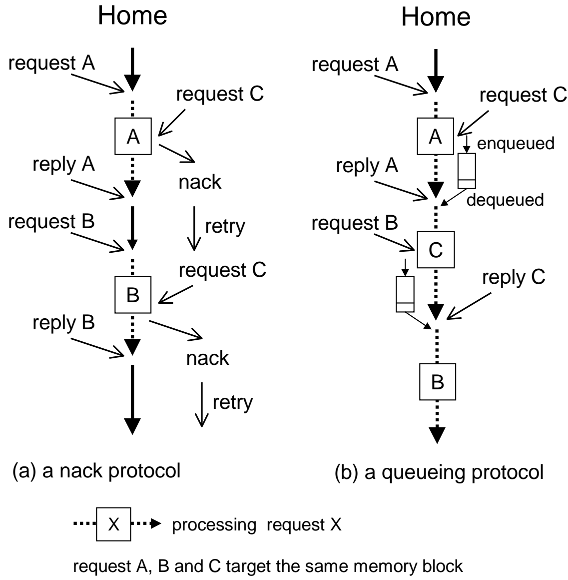

Figure 6. A nack protocol and a queuing protocol

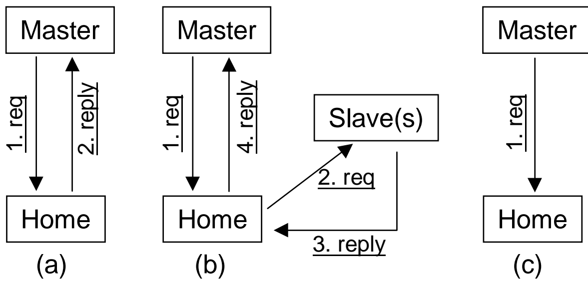

Figure 7. Coherency control sequences with the Cenju-4 protocol

ered message is sent to the next switch. Thus, the network collects all gathered messages, but outputs only one message to the destination node.

3.3. Cache Coherence Protocol

In our discussion of the coherence protocol, we use the following naming conventions. A '*master*' is a node that contains the processor originating a memory access. A '*home*' is a node that contains the main memory and the directory for a given physical memory address. A '*slave*' is a node that caches the data, unless the node is a *master*.

A cache coherence protocol consists of a sequence of operations applied to memories and caches to achieve data coherency. Operations are mediated by request and reply messages sent among *masters*, *homes* and *slaves*. Such a coherence protocol must prevent star-

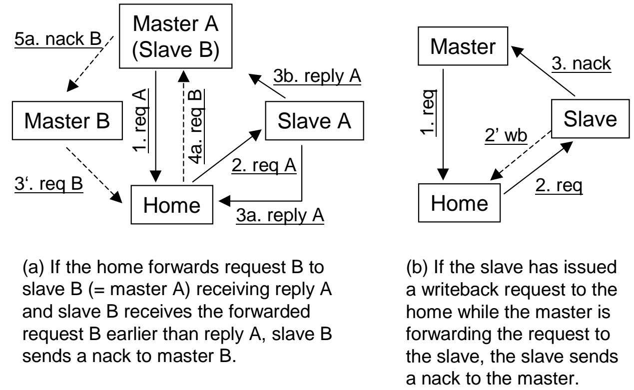

Figure 8. Sequences which cause nack at slave with the DASH protocol

vation and deadlock in order to guarantee forward progress of memory access. We describe starvation in this section, and deadlock in the next section.

Several previously devised coherence protocols have a negative-acknowledge (nack) sequence. When either a *home* or a *slave* can not process a request, it sends a nack message to a *master* and forces the *master* to retry the request. A nack sequence occurs when several nodes access the same memory block. Figure 6(a) shows that request C is disturbed by requests A and B, which target the same memory block as request C, and a nack sequence occurs repeatedly. This implies that the nack sequence causes starvation.

In order to prevent starvation, Cenju-4 adopts a cache coherence protocol, which queues requests that can not be processed immediately without a nack for later processing at a *home*. This cache coherence protocol is based on the Alewife protocol [1] [3]. Modifications were made mostly to enable queuing and to make some performance improvements. For performance improvements, the Cenju-4 protocol supports an exclusive cache state (known as one of the states in MESI) and a write request which enables a store access to a shared block to be satisfied without a data transfer. Details of the cache coherence protocol are shown in the Appendix.

Figure 7 shows all the sequences taken by the cache coherence protocol in Cenju-4. As shown in Figure 7(a), a *home* replies to a *master* if the *home* can satisfy a request. As shown in Figure 7(b), if the *home* can not satisfy a request, it forwards the request to a *slave*. In order to remove two nack sequences of the DASH[9] protocol shown in Figure 8, a *slave* sends a reply to the *home*, and the *home* forwards the reply to the *master*. As shown in Figure 7(c), if a request issued by a *master* is a writeback request, the 'no-reply' sequence is taken. A *home* processes a writeback request even while the *home* is processing another request that targets the same memory block and is waiting for its reply.

This writeback sequence reduces buffer sizes required for preventing starvation and deadlock.

Figure 6(b) shows the behavior of the queuing protocol. A *home* does not reply to any request with a nack. A *home* queues the request B and C which the *home* receives while the *home* is processing another request that targets the same memory block and is waiting for its reply. After receiving the reply, the *home* dequeues the requests and processes them.

A *home* queues requests in the same buffer, even though these requests target different memory blocks. The buffer is placed in the main memory and is controlled as a FIFO queue. The *home* uses the reservation bit in the directory for checking whether a request is waiting at the top of the queue. When the *home* saves the request at the top of the queue, the *home* sets the reservation bit of the target directory. If the *home* processes a reply and the reservation bit of the target directory has been set, the *home* resets the bit and reads the request at the top of the queue (does not dequeue yet). If the request is processed, the *home* dequeues the request from the queue and reads the next request in the queue. If there is a request that the *home* can not process yet, the *home* sets the reservation bit of the target directory, stops processing the requests in the queue, and waits for a reply. This process continues until the queue becomes empty.

It is possible for all requests except writeback requests to be saved in the queue. The maximum number of requests (excluding writeback requests) issued from one *master* is four in Cenju-4. And these requests are represented with 64 bits. Therefore, the buffer size prepared for queuing requests at each node is 32K bytes [1] on a 1024-node system.

It has been previously reported that a Scalable Coherent Interface (SCI)[5] also prevents starvation. The coherence protocol is formed on SCI by sharing chains through the caches in order to form a distributed coherence directory. This form is also used to link requests from many nodes. The requests are satisfied one by one in the same order in which they are linked. This scheme fully depends on a distributed chained directory and is not applicable to systems that adopt a centralized directory, such as Cenju-4.

Both the DASH project and the Alewife[7] project also addressed the starvation problem, but no implementable solution that prevented starvation was given in either work.

3.4. Deadlock Prevention

The previous section shows that the coherence protocol consists of transmitting request messages and reply messages among a *master*, a *home* and *slaves*. All physical nodes contain the three modules which act as *master*, *home* and *slave*. The messages between the

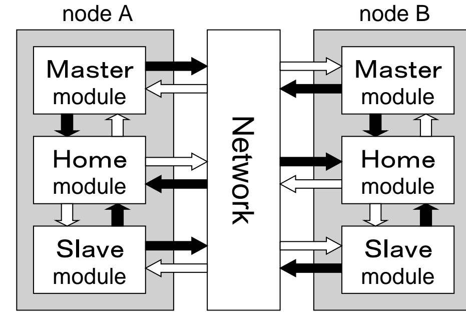

Figure 9. Resource dependency graph

modules in the different nodes are transmitted through the network. Each module starts a service by receiving a message, and does not start another service while processing a message.

The network of Cenju-4 is deadlock-free by itself. The path between two nodes is unique and the messages through the same path are delivered in the same order they are transmitted.

Figure 9 shows a resource dependency graph. In this graph, three modules in each node and network are resources, and the arrows represent dependencies between resources caused by the message transmission. There are many loops in the graph, and this suggests that deadlock will occur.

One solution to this deadlock problem, employed by the DASH project, is to prepare two network channels. Request messages are assigned to one channel and reply messages to the other. This solution is not suitable for most cache coherence protocols, since their coherence protocols contain dependencies that are more complicated than simple request-reply ones. There are request-request dependencies that occur when a *home* forwards a request to a *slave*. To eliminate such dependencies, a *home* or a *slave* sends a nack to a *master* under certain situations in which the buffer for outputting to the request network channel is full. Unfortunately, this solution transforms the deadlock problem into a starvation problem. Moreover, in order to have two network channels, the network switch must have double buffers for each physical path. In Cenju-4, since buffers already occupy 57% of the switch chip in gates with one network channel, it is hard to have two network channels.

On the other hand, the Alewife project employed a deadlock prevention scheme with one network. When a node detects deadlock, the node queues all messages sent from the network in the main memory under software control. However, the buffer size required for queuing all messages was not specified.

Cenju-4 also prevents deadlock with one network by queuing not all but certain types of messages in the main memory. This scheme enables the buffer size to be reduced to 128K bytes, thus finite, in a 1024-node system. The rest of this section describes this scheme

[1] $1024 \times 4 \times 64 bits = 32 Kbytes$

in detail.

Cenju-4 prevents deadlock by removing the dependencies represented by white arrows in Figure 9 by introducing buffers which can queue all messages that pass through the arrows. Thus all loops are removed from the graph. We select these arrows for minimizing the memory requirements, though there are many alternative ways to remove loops. Each node prepares three buffers: a buffer that queues messages received by the *master* module, a buffer that queues messages received by the *slave* module, and a buffer that queues messages to be transmitted from the *home* module to the network.

The *master* module receives reply messages that may contain data. The maximum number of reply messages received by one *master* module is the maximum number of a processor's outstanding requests: four in our system. The *master* module has a buffer sufficiently large to receive all reply messages.

A *slave* module receives request messages without data. The maximum number of request messages received by one *slave* module is found by the maximum number of a processor's outstanding requests multiplied by the number of nodes. In 1024-node systems, each node allocates a 64K-byte[2] region in the main memory for queuing messages. A *slave* module also has a buffer for receiving several request messages, and uses the buffer in the main memory only when the buffer in the module is full.

A *home* module outputs request and reply messages that may contain data. However, data is always in the memory block and does not need to be saved in the buffer. A *home* module also outputs invalidation request messages, which are generated by processing one request message. In our implementation, one invalidation request message and a node map that indicates the destination nodes are saved instead of invalidation messages. Invalidation request messages are generated from this information. By this means, the maximum number of messages and node maps queued in the buffer is the same as that of messages one *slave* module receives. In a 1024-node system, each node allocates another 64K-byte[3] region in the main memory for queuing messages. A *home* module also has a buffer in the module and uses the buffer in the main memory only when the buffer in the module is full.

4. Performance

In this section, we discuss the performance of the DSM system of Cenju-4 using latencies of load accesses and store accesses. We also measured the performance by using four applications in the NAS Parallel Benchmarks V2.3.

[2] 1024 (nodes) × 4 (outstanding requests) × 128 bits (one entry)

[3] 1024 (nodes) × 4 (outstanding requests) × 128 bits (one entry)

Table 2. Load access latencies (ns)

network stages (no. of nodes)	2 (~16)	4 (~128)	6 (~1024)
a) private	470	470	470
b) shared local(clean)	610	610	610
c) shared remote(clean)	1690	2210	2730
d) shared local(dirty)	1900	2480	3060
e) shared remote(dirty)	3120	4170	5220

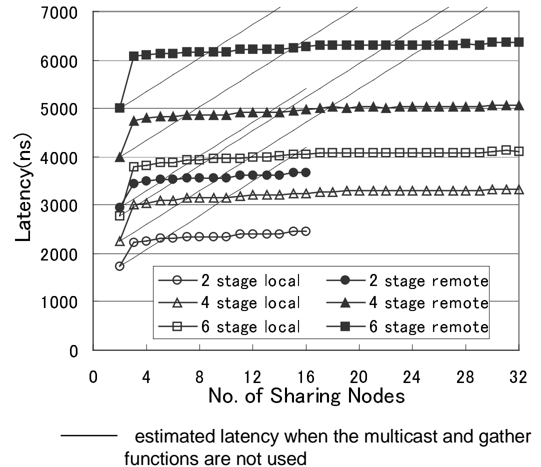

Figure 10. Store access latencies

4.1. Memory Access Latency

In our discussion of the results, we use the following naming conventions. A 'private' memory is a main memory accessed without the DSM system, while a 'shared' memory is a main memory accessed through the DSM system. A 'local' memory is a shared memory of its own node, while a 'remote' memory is a shared memory of another node.

Table 2 and Figure 10 show load and store access latencies, respectively. This is the time from when a memory access instruction is issued until the memory access graduates. Table 2 shows latencies of load accesses that miss the secondary cache. Figure 10 shows latencies of store accesses that hit a cache block shared with nodes. The Cenju-4 multistage network changes the number of stages according to the system size. In this evaluation, we use a 16-node system that contains a two-stage network, and a 128-node system that contains a four-stage network. Latencies with a six-stage network shown in Table 2 and Figure 10 are estimated from the difference between latencies with a two-stage network and a four-stage network. Latencies when the multicast and gathering functions of the network are not used are also estimated by using a logic level simulator of Cenju-4.

In Table 2, though both a) and b) are accesses to the

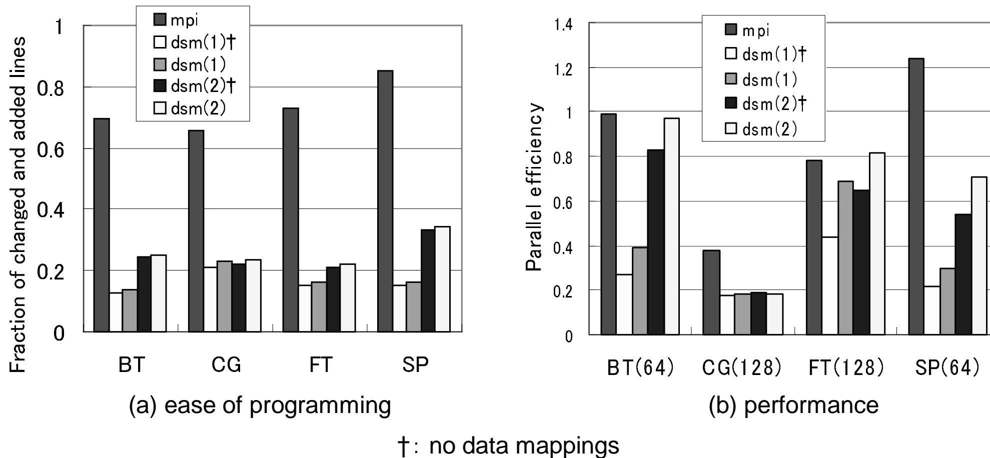

Figure 11. DSM vs Message Passing

local memory, the costs of accessing directory are added to the latencies of b). The difference among b), c), d) and e) arises from the difference in cache-coherence sequences and from how many times the network is used to transfer messages in the sequences. The sequence taken by b) and c) is (a) shown in Figure 7, and the sequence taken by d) and e) is (b) shown in Figure 7. The number of times the network is used to transfer messages in b), c), d), and e) are zero,two,two, and four, respectively. In c), d) and e), the latency increases not with the system size but with the number of network stages.

Figure 10 shows that store access latency increases not with the system size but with the number of stages of the network. Moreover, the store access latencies do not increase linearly with the number of sharing nodes. It is estimated that the store access latency will take 6.3 microseconds when the data is shared with 1024 nodes. On the other hand, the estimated results show that the latency linearly increases with the number of nodes when the multicast and gathering functions are not used. It is also estimated that the store access latency will take 184 microseconds when the data is shared with 1024 nodes. This result shows that multicasting invalidation requests and gathering replies to them is an effective approach.

The store access latency greatly increases when the number of nodes sharing a block exceeds two. This is because the multicast and gathering functions are used when the number of target nodes exceeds one, i.e., the number of nodes sharing a block exceeds two. If the number of target nodes is one, a singlecast message is used for sending an invalidation request messages and collecting a reply message. It is possible to use singlecast messages in order to improve store access latency up to a certain number of nodes, though it was not implemented in Cenju-4.

4.2. Applications

4.2.1. Workload

We use four applications in the NPB(NAS Parallel Benchmarks V2.3 Class A) as workloads. The applications we use are BT, CG, FT and SP. We measured the performance of four programs for each application:

seq – a given sequential program

mpi – a given parallel program written with MPI (Message Passing Interface) library

dsm(1) – a parallel program written with shared memory library

dsm(2) – an optimized parallel program written with shared memory library

We made *dsm(1)* and *dsm(2)* programs from a *seq* program. The *dsm(1)* programs are parallelized only on the outermost loop. The *dsm(2)* programs are tuned to optimize memory accesses by loop translations, dividing shared arrays and mapping the arrays to a private memory. In both the *dsm(1)* programs and the *dsm(2)* programs, data mappings are specified for shared data to localize memory accesses.

The shared memory library has several functions: to allocate shared memory, to specify a data mapping, and so on. We use MPI library for performing synchronization and reduction operations in the shared memory programs.

With MPI library, the message passing mechanism achieves 9.1 microsecond latencies on a 128-node system, and 169M byte/s throughput.

4.2.2 DSM vs Message Passing

Figure 11 shows the ease of programming and the performance of the *mpi*, *dsm(1)* and *dsm(2)* programs.

Table 3. Secondary cache miss characteristics of applications

64/128 nodes		miss ratio	private	shared local	shared remote
BT	dsm(1)†	1.49%	2.4%	1.7%	95.9%
(64)	dsm(1)	1.47%	2.2%	63.7%	34.1%
	dsm(2)†	0.84%	76.3%	0.6%	23.0%
	dsm(2)	0.85%	76.1%	12.7%	11.2%
CG	dsm(1)†	1.48%	27.8%	0.6%	71.6%
(128)	dsm(1)	1.48%	26.7%	0.7%	72.6%
	dsm(2)†	1.48%	28.2%	0.6%	71.1%
	dsm(2)	1.44%	25.9%	0.7%	73.4%
FT	dsm(1)†	0.84%	30.2%	0.6%	69.2%
(128)	dsm(1)	0.81%	30.8%	50.9%	18.3%
	dsm(2)†	0.69%	57.2%	0.4%	42.4%
	dsm(2)	0.77%	59.2%	23.0%	17.9%
SP	dsm(1)†	1.77%	4.5%	1.5%	93.9%
(64)	dsm(1)	1.84%	4.3%	36.0%	59.7%
	dsm(2)†	1.04%	24.7%	1.9%	73.3%
	dsm(2)	1.02%	24.5%	36.9%	38.6%

†: no data mappings

In order to evaluate the ease of programming, we measured the program rewriting ratios of the *mpi*, *dsm(1)* and *dsm(2)* programs. The rewriting ratio is calculated by dividing the number of lines changed from and added to the lines of the sequential programs by the number of lines of the sequential program. We also measured the performance of four applications using a 128-node system. We run BT and SP using 64 nodes, and run CG and FT using 128 nodes. The performance is shown by parallel efficiency (speedup divided by the number of nodes). Table 3 shows secondary cache miss characteristics of *dsm(1)* and *dsm(2)* programs running with 64 or 128 nodes. For each program, we give secondary cache miss ratios and breakdowns of the cache misses. Cache misses include store accesses to shared cache blocks. We also evaluate the effect of specifying shared data mappings. The '*no data mappings*' in Figure 11 and Table 3 indicates that mapping codes are removed from programs.

Figure 11(a) shows the program rewriting ratio of each program. The difference in the program rewriting ratio between *mpi* and *dsm(1)* arises from the following differences. To parallelize sequential programs, we have to change initial values and end values of induction variables of loops and insert synchronizations in both shared memory programs and message passing programs. Almost all the changed lines of *dsm(1)* programs account for these changes. On the other hand, explicit inter-node communications are added to *mpi* programs. Moreover, arrays are divided in a complicated way to minimize the amount of communications in the given *mpi* programs. The ratios of *dsm(2)* programs increase from those of *dsm(1)* programs because of the optimization. However, the ratios of *dsm(2)* programs are less than half of those of *mpi* programs.

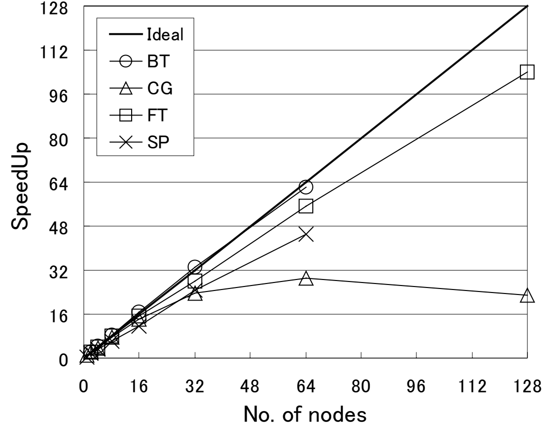

Figure 12. Speedups of applications

On both *dsm(1)* and *dsm(2)*, specifying data mappings causes less increase in the program rewriting ratio.

Figure 11(b) shows the performance of each program. The *dsm(1)* programs, that are simply parallelized from the *seq* programs, do not show high parallel efficiency. The efficiency is only about 20% on BT, CG and SP, and 40% on FT. However, specifying data mappings and optimizing memory access patterns greatly improve parallel efficiency to 97% on BT, 81% on FT, and 71% on SP. Table 3 shows that the optimization decreases the cache miss ratios and the shared miss ratios, and that specifying data mappings decreases the remote miss ratios. On CG, optimizing memory access patterns and specifying data mappings has no effect on secondary cache miss characteristics. This is the reason that the performance of the *dsm(2)* program does not improve. On BT and FT, *dsm(2)* programs achieve high performance comparable to that of *mpi* programs.

These results show that we can write shared memory programs with less rewriting than that of message passing programs. In addition, some shared memory programs achieve high performance comparable to that of message passing programs. Specifying data mappings and optimizing memory access patterns produces high performance on many shared memory programs.

4.2.3 Performance scalability of dsm programs

Figure 12 shows the performance of four applications up to 64 nodes on BT and SP, and up to 128 nodes on CG and FT. We use *dsm(2)* programs and specify data mappings. Though BT, FT and SP show good speedups, the speedup of CG is saturated. In this section, we discuss performance scalability of these shared memory programs.

Table 4 gives characteristics of four applications running with 16 and 64 nodes on BT and SP, and 16 and 128 nodes on CG and FT. For each application, we give the number of executed instructions, the number of memory access instructions, and the breakdowns of

Table 4. characteristics of applications

	no. of nodes	execution time			executed instructions‡					secondary cache misses‡			
		total (sec)	system	sync.	total†	mem. access†	private	shared		ratio	private	shared	
								local	remote			local	remote
BT	16	203.722	3.26%	3.84%	25542	12027	82.7%	13.8%	3.60%	0.86%	71.9%	22.5%	5.59%
	64	56.324	2.94%	7.72%	6386	3007	82.7%	13.0%	4.35%	0.82%	75.0%	13.1%	11.9%
CG	16	5.348	1.93%	7.04%	369.0	141.0	66.4%	2.52%	31.1%	2.73%	90.0%	0.66%	9.31%
	128	4.182	0.88%	25.1%	46.61	17.85	66.8%	0.28%	32.9%	2.39%	18.4%	0.73%	80.9%
FT	16	9.222	5.04%	1.67%	1205	419.3	92.5%	4.69%	2.81%	0.77%	47.0%	37.6%	15.4%
	128	1.346	4.37%	8.92%	150.8	52.43	92.5%	4.52%	2.98%	0.79%	45.0%	35.7%	19.3%
SP	16	214.763	7.34%	5.42%	17420	6184	49.9%	19.9%	30.2%	1.24%	21.4%	59.3%	19.4%
	64	68.064	5.89%	12.8%	4356	1547	50.0%	17.3%	32.8%	1.03%	13.8%	39.8%	46.4%

†: $\times 10^6$ ‡: system and synchronization phases are not included in measurements

memory accesses as a fraction of all memory access instructions. Table 4 also shows the total execution times, the system execution times and the synchronization execution times as a fraction of total execution times, the secondary cache miss ratios, and the breakdowns as a fraction of all secondary cache misses. Except for execution times, system and synchronization phases were not considered when measuring these characteristics. The values given were obtained by taking the average of all nodes.

The numbers of total executed instructions and memory access instructions decrease with an increase in the number of nodes. This implies that all programs are scalable by themselves and synchronization cost and average memory access latency will harm performance scalability. Even the synchronization time subtracted from the execution time does not show linear speedups. This implies that average memory access latency degrades with an increase in the number of nodes.

There is little difference in the memory access breakdowns between 16 nodes and 64 or 128 nodes; only a slight decrease in local access ratios and a slight increase in remote access ratios are seen. The remote access ratios increase by 0.75% on BT, 1.82% on CG, 0.17% on FT, and 2.56% on SP. However, there are significant differences in the breakdowns of secondary cache misses between 16 nodes and 64 or 128 nodes. In this case, the remote miss ratios increase by 6.34% on BT, 71.5% on CG, 3.90% on FT, and 27.1% on SP. These increases degrade the average memory access latency and harm performance scalability. The notably high increase in the remote miss ratio of CG is the cause of CG performance saturation.

On CG, shared data is distributed to all nodes, and each node calculates the data assigned to it. At some phase of the program, each node accesses all shared data that is modified at the previous phase for calculating one data, and repeats accessing all shared data for all assigned data. This access pattern of shared data increases the remote memory miss ratio, since the time that shared data is reused decreases with the increase in the number of nodes.

To improve the performance scalability of CG, the system must enable this kind of access pattern to be scalable. To achieve this, it is not enough for the system to make store access latency scalable. It is also required for the system to make the load access latency to be scalable, even though all nodes are accessing the same memory block. Moreover, these load accesses must be satisfied at the local memory. One solution to this problem is for the system to use the main memory as third-level cache and to use an update-type protocol for this type of data. When data are modified, data in third-level caches of all nodes are updated. The load access at each node is satisfied by its third-level cache in the main memory.

5. Conclusion

This paper has described the DSM mechanism of Cenju-4. This mechanism is designed to be highly scalable up to 1024 nodes and has the following features:

- A directory which dynamically switches its representation from a pointer structure to a bit-pattern structure
- The multicast and gathering functions of a network
- A cache coherence protocol that prevents starvation with a centralized directory
- A deadlock-free mechanism with one network

The evaluation result shows that the bit-pattern structure enables more precise representation than other imprecise directory schemes, such as a coarse vector scheme. And by using the multicast and gathering functions of a network, the system enables the store access latency to be scalable. Starvation is prevented by using a 32K-byte buffer in the main memory for queuing certain types of messages. Deadlock is also prevented by using two 64K-byte buffers for queuing certain types of messages.

The performance results we obtained using four applications show the ease with which shared memory programming can be performed, and that high-level performance can be achieved by adjusting the program so that it can fit a distributed memory system.

The performance results also show that making store access latency scalable is not enough for some applications. It is also required that load access latencies are scalable, even though all nodes are accessing the same memory block. Moreover, these load accesses must be satisfied by the local memory.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. "The MIT Alewife Machine: Architecture and Performance." In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 1112–1118, December 1978.

[2] L. M. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems." In *IEEE Transactions on Computers*, volume C27, pages 1112–1118, December 1978.

[3] D. Chaiken, J. Kubiatowicz, and A. Agarwal. "LimitLESS Directories: A Scalable Cache Coherence Scheme." In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[4] A. Gupta, W.-D. Weber, and T. Mowry. "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes." In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 312–321, May 1990.

[5] Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Scalable Coherent Interface*, August 1993.

[6] Y. Kanoh, M. Nakamura, T. Hirose, T. Hosomi, H. Takayama, and T. Nakata. "Message Passing Communication in a Parallel Computer Cenju-4." In *Proceedings of the 2nd International Symposium on High Performance Computing*, volume 1615 of *LNCS*, pages 55–70. Springer-Verlag, May 1999.

[7] J. D. Kubiatowicz. *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*. PhD thesis, Massachusetts Institute of Technology, 1998.

[8] J. Laudon and D. Lenoski. "The SGI Origin: A cc-NUMA Highly Scalable Server." In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[9] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennesy. "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor." In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[10] T. Matsumoto, K. Nishimura, T. Kudoh, K. Hiraki, H. Amano, and H. Tanaka. "Distributed Shared Memory Architecture for JUMP-1 a general-purpose MPP prototype." In *Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms and Networks*, pages 131–137, June 1996.

[11] MIPS Technologies, Inc. *MIPS R10000 Microprocessor User's Manual Version 2.0*, October 1996.

[12] R. Simoni and M. Horowitz. "Dyanmic Pointer Allocation for Scalable Cache Coherence Directories." In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 72–81, April 1991.

APPENDIX

Requests issued by a master

- A read-shared request which is initiated by a processor's load access to an invalid cache block
- A read-exclusive request which is initiated by a processor's store access to an invalid cache block
- An ownership request which is initiated by a processor's store access to a shared cache block
- A writeback request when a modified cache block is replaced

States of a cache block

A cache block in the cache memory may be in one of four states: modified, exclusive, shared, invalid, known as MESI. We abbreviate them as M^c, E^c, S^c and I^c.

States of a memory block

As with cache blocks, a memory block in the main memory may also be in one of five states: clean, dirty, pending-shared, pending-exclusive, and pending-invalidate. We abbreviate them as C^m, D^m, Ps^m, Pe^m and Pi^m, respectively. The block state C^m indicates that there are many nodes which cache the data, and that the data in the main memory is valid. The block state D^m indicates that only one node caches the data, and the data in the memory block may be invalid. The block state Ps^m, Pe^m and Pi^m are pending states, while C^m and D^m are stable states.

Read-shared request sequence

The following sequence is initiated by a processor's load access to a I^c state cache block.

(1) The *master* sends the read-shared request to the *home*.

(2) Upon receiving the request, the *home* checks the directory.

  - If the state is C^m or D^m and no node or only the *master* is registered in the node map, the *home* changes the state to D^m and sets the node map that indicates only the *master* holds the copy. The *home* sends the data in the memory block to the *home*. Go to (3).
  - If the state is C^m and nodes other than the *master* are registered in the node map, the *home* adds the *master* to the node map. The *home* sends the data in the memory block to the *home*. Go to (4).
  - If the state is D^m and a node other than the *master* is registered in the node map, the *home* changes the state to Ps^m and forwards the request to the *slave* registered in the node map. Go to (5).
  - If the state is a pending state, the *home* queues the request in the main memory until the state becomes a stable state.

(3) Upon receiving the data, the *master* puts the given data into the cache block and changes the state to E^c. END.

(4) Upon receiving the data, the *master* puts the given data into the cache block and changes the state to S^c. END.

(5) Upon receiving the request, the *slave* checks the cache block state.

- If the state is M^c, the *slave* changes the state to S^c and sends the data in the cache block to the *home*. Go to (7).
- If the state is E^c, the *slave* changes the state to S^c and sends the reply to the *home*. Go to (6).
- Otherwise, the *slave* sends the reply to the *home*. Go to (6).

(6) Upon receiving the reply, the *home* changes the state to C^m, adds the *master* to the node map, and sends the data in the memory block to the *master*. Go to (4).

(7) Upon receiving the data, the *home* puts the given data into the memory block, changes the state to C^m, adds the *master* to the node map, and forwards the data to the *master*. Go to (4).

Read-exclusive request sequence

The following sequence is initiated by a processor's store access to an I^c state cache block.

(1) The *master* sends the read-exclusive request to the *home*.

(2) Upon receiving the request, the *home* checks the directory.

- If the state is C^m or D^m and no node or only the *master* is registered in the node map, the *home* changes the state to D^m and sets the node map that indicates only the *master* holds the copy. The *home* sends the data in the memory block to the *home*. Go to (3).
- If the state is C^m and nodes other than the *master* are registered in the node map, the *home* changes the state to Pe^m, and sends invalidation requests to all *slave*s registered in the node map (the invalidation requests are multicasted in the network). Go to (4).
- If the state is D^m and a node other than the *master* is registered in the node map, the *home* changes the state to Pe^m and forwards the request to the *slave* registered in the node map. Go to (6).
- If the state is a pending state, the *home* queues the request in the main memory until the state becomes a stable state.

(3) Upon receiving the data, the *master* puts the given data into the cache block and changes the state to M^c. END.

(4) Upon receiving the invalidation requests, all *slave*s change their cache block states to I^c if they hold copies and send replies to the *home*. All replies are gathered to one reply in the network.

(5) Upon receiving the reply, the *home* changes the state to D^m, sets the node map that indicates only the *master* holds the copy, and sends the data in the memory block to the *home*. Go to (3).

(6) Upon receiving the request, the *slave* checks the cache block state.

- If the state is M^c, the *slave* changes the state to I^c and sends the data in the cache block to the *home*. Go to (7).
- Otherwise, the *slave* changes the state to I^c if it holds the copy, and sends the reply to the *home*. Go to (5).

(7) Upon receiving the data, the *home* puts the given data into the memory block, changes the state to D^m, sets the node map that indicates only the *master* holds the copy, and forwards the data to the *master*. Go to (3).

Ownership request sequence

The following sequence is initiated by a processor's store access to a S^c state cache block.

(1) The *master* sends the ownership request to the *home*.

(2) Upon receiving the request, the *home* checks the directory.

- If the state is C^m and nodes other than the *master* are registered in the node map, the *home* changes the state to Pi^m and sends invalidation requests to all *slave* nodes registered in the node map. Go to (3).
- If the state is a pending state, the *home* changes the ownership request to the read-exclusive request and queues the changed request in the main memory until the state becomes a stable state. (Refer to the read-exclusive request sequence.)

(3) Upon receiving the invalidation requests, all *slave*s change their cache block states to I^c if they hold copies and send replies to the *home*. All replies are gathered to one reply in the network.

(4) Upon receiving the reply, the *home* changes the state to D^m, sets the node map that indicates only the *master* holds the copy, and forwards the reply to the *master*.

(5) Upon receiving the reply, the *master* changes the state to M^c. END.

WriteBack request sequence

When an M^c state cache block is replaced, the following sequence starts:

(1) The *master* changes the cache block state to I^c, and sends the data in the cache block to the *home*.

(2) Upon receiving the data, the *home* writes back the given data into the memory block and checks the directory.

- If the state is D^m, the *home* changes the state to C^m and sets the node map that indicates no node holds a copy.
- Otherwise, the *home* does not change the directory.