The Intelligent Cache Controller of a Massively Parallel Processor JUMP-1

Masahiro Goshima† Shin-ichiro Mori† Hiroshi Nakashima‡ Shinji Tomita†

† Division of Information Science
Kyoto University
Yoshida Hon-machi, Sakyo-ku, Kyoto, Japan
{goshima,moris,tomita}@kuis.kyoto-u.ac.jp

‡ Dept. of Information and Computer Sciences
Toyohashi University of Technology
Hibarigaoka, Tenpaku-cho, Toyohashi, Aichi, Japan
nakasima@tutics.tut.ac.jp

Abstract

This paper describes the intelligent cache controller of JUMP-1, a distributed shared memory type MPP. JUMP-1 adopts an off-the-shelf superscalar as the element processor to meet the requirement of peak performance, but such a processor lacks the ability to hide inter-processor communication latency, which may easily become too long on MPPs. Therefore JUMP-1 provides an intelligent memory system to remedy the week point. The cache controller is one of the main components of the memory system, and provides many cache-level supports for inter-processor communication; explicit cache control, high-bandwidth cache prefetching, and a few types of synchronization structures for fine-grained message communication.

1 Introduction

To give a fundamental paradigms and technologies for 10^6 scale massively parallel processing in the next century, a joint project named **JUMP**(Joint University Massively Parallel processing) project was organized. 23 laboratories of 18 universities in Japan participated in this project. The research items of the project are computation models, programming languages, operating systems, and hardware systems.

The important challenge of the project is to build a prototype system named **JUMP-1**. The aim of JUMP-1 is not only to substatiate novel ideas proposed in the hardware research of the project, but also to provide a platform for that of software. That is, it is planned to implement a prototype parallel operating system, some parallel languages, and various parallel applications on JUMP-1.

JUMP-1 is a general purpose massively parallel processor (MPP), composed of maximum of 1024 processors with distributed shared memory (DSM).

The main feature of JUMP-1 is its intelligent memory system. In this paper, we describe the intelligent cache controller of JUMP-1, which is one of the main components of the intelligent memory system.

In the following sections, Section 2 outlines the overall architecture of JUMP-1, and Section 3 describes the detail of the intelligent cache controller. Section 4 gives the conclusion and our research schedule.

2 JUMP-1

Before going into details of the intelligent cache controller in later sections, this section outlines overall architecture of JUMP-1.

2.1 Design policy

JUMP-1 is a MPP which has off-the-shelf general purpose processors and DSM. The following discussion gives our policy on designing these key components; the element processor and communication through DSM.

2.1.1 Element processor

The element processors of current MPPs can be divided into two types. One is off-the-shelf modern microprocessor used in the most of multiprocessor systems. The other is custom-made fine-grained or multithreaded processor, such as that for [7]. The former has the advantage of its high peak performance and hardware/software productivity, while the latter is superior in its low-cost context switching to hide inter-processor communication.

For the element processor of JUMP-1, we chose an off-the-shelf superscalar processor, Sun SuperSPARC+, not only because of the reason show above but also our key observation on high-performance parallel processing. That is, we claim that parallel processing consists of *local* and

global operations each of which should be performed by a component fit to the operation. Thus the off-the-shelf processor, naturally fit to the *local* processing in a parallel program, should be responsible for the *local* part.

The *global* operation such as synchronization and communication, however, is not fit to the element processor because the operation often makes the processor stalled by its long latency. Thus we remove the burden of the *global* operations from the element processor and give it to the **intelligent memory system**. The memory system of JUMP-1 is designed to proveide various functions to hide or reduce the latency of the *global* operations in order to keep the element processors from stalling.

2.1.2 Communication through DSM

The most basic inter-processor communication facility of JUMP-1 is shared memory. Shared memory is a powerful communication facility for general parallel programs. Memory protection, which is indispensable for multi-user environment, can be also realized on the shared virtual memory system.

Ordinary shared memory systems with hardware cache coherence, however, do not fit for irregular or asynchronous applications which frequently exchanges fine-grained messages. The intelligent memory system of JUMP-1 provides various high-level functions to hide or reduce fine-grained inter-processor communication latency. The functions are called in the form of the protected shared memory access.

The intelligent cache controller, which is the subject of this paper, is one of the main components of the intelligent memory system, and plays an important role in it.

2.2 System configuration

In order to map the physical connectivity of system components onto the logical architecture in a natural form, JUMP-1 adopts a clustered architecture. A cluster consists mainly of four elemet processors and cluster main memory. The system has 256 clusters connected by inter-cluster networks.

Inter-cluster Networks JUMP-1 has three different networks; the main network, the I/O network, and the maintenance network. All of these networks are newly proposed and designed for JUMP-1.

The main network is Recursive Diagonal Torus (**RDT**) network[8]. RDT has a hierarchical structure of recursively coarsened 2-dimensional tori. Higher/coarser tori are placed diagonally by 45 degrees on a lower/finer torus. This structure achieves both small degree and diameter as well as $O(\log N)$ multicasting and combining.

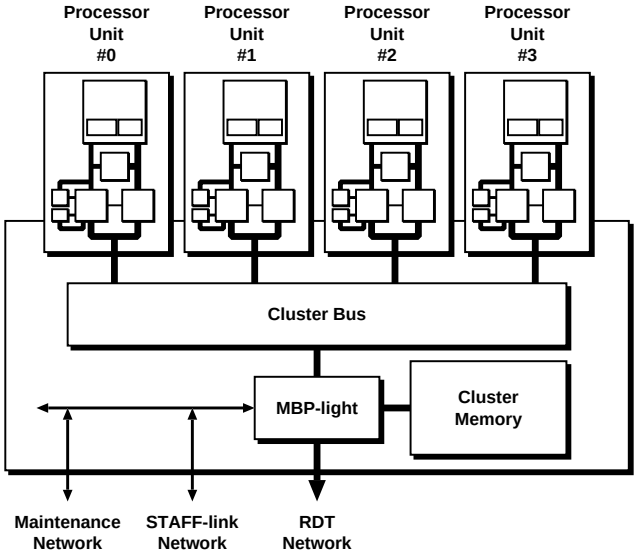

Figure 1. Block diagram of a JUMP-1 cluster

The I/O network interconnects clusters with various I/O devices such as disks and high-definition video devices. The I/O network is a bundle of point-to-point high-speed serial links named STAFF-link [6] with a concentrator which dynamically changes the assignment of I/O devices to clusters.

The maintenance network, which is a tree of low-speed bust robust buses, is used for booting and instrumentation.

Cluster Figure 1 shows the block diagram of the JUMP-1 cluster. A cluster has four processor units and a memory unit connected by a shared bus. All of these components are mounted on a printed circuit board.

The detail of the processor unit is described later in this section.

The memory unit consists of a cluster memory and a memory control coprocessor called **MBP-light**. The cluster memory forms a potion of DSM.

MBP-light MBP-light is a *lighter* version of Memory Based Processor proposed in [3]. MBP-light is responsible for all the memory access packet coming from inside and outside of the cluster. While *local* transactions that simply read and write(back) the cluster memory are handled by a fully hardwired mechanism, *global* transactions that require system-wide coherence maintenance and/or high-level memory operations are manipulated by the microcode of MBP-light. Thus MBP-light has a *message driven* architecture that efficiently creates, suspends, resumes and terminates a thread invoked at the arrival of a memory access packet.

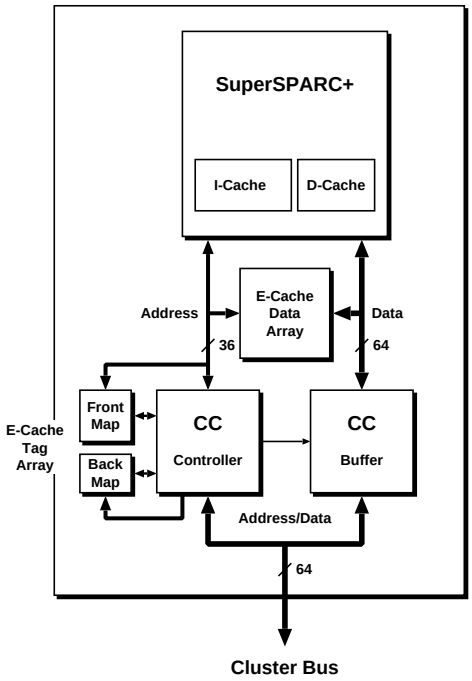

Figure 2. Block diagram of a processor card

type		way	size	block size
primary	I	5	20KB	32B
	D	4	16KB	
secondary	E	1	1MB	

Table 1. Cache parameters

Processor unit Figure 2 shows the detailed block diagram of the processor unit. The processor unit consists of an element processor, Sun SuperSPARC+, and its private caches. All of the components are mounted on a printed circuit card.

Caches A processor has three types of private caches; on-chip primary **I-cache** and **D-cache**, and the external secondary **E-cache**. The I-cache and D-cache are set-associative and phyically addressed. The D-cache is write-through and adopts no write allocate policy. The E-cache is unified, direct-mapped and phyisically addressed. Table 1 shows the other parameters of these caches.

The coherence of these caches is maintained by the intelligent cache controller, **CC**, which is the subject of this paper. Each block of the primary caches can be invalidated by an external agent, CC in this case. While the E-cache is completely under the control of the CC.

The CC consists of twin chips; one is the controller core and the other is for data buffers. Both of the chips are fabricated using 0.5μm CMOS double-metal gate array technology, and sealed into a 304-pin plastic QFP package.

2.3 Intelligent memory system

The intelligent memory system provides various functions to hide and reduce the inter-processor communication latency. These functions are implemented as hardware logic of CC and microcode on MBP-light.

In the rest of this section, we explain the addressing system and how to invoke the functions of the intelligent memory system, before going into the detail of CC.

2.3.1 Addressing system

JUMP-1 memory system adopts a variant type of shared virtual memory[4]. A cluster memory is used not only as a main memory but also as a cache for cluster memories of remote clusters. Remember that each processor has a private secondary cache. Thus the cluster memory also functions as a shared third-level cache.

The cache block size of the third-level cache is 4KB, which is the minimum page size of SuperSPARC+. Thus the cached block of the third-level cache is called a copy page, whereas a ordinary non-copy page is called an original page. Original pages and copy pages are mingled on a cluster memory.

The copy page is accessed by the processors in the cluster just the same way as the original page. The physical address used to access these pages on a cluster memory is called **cluster address**. The mapping from virtual pages to original or copy pages are managed by the ordinary page table with which SuperSPARC+ MMU generates the cluster address.

On the other hand, another address is required for inter-cluster memory access, because the cluster address is individual for each cluster. This address is called **network address**. The network address is a virtual address which uniquely identifies system-wide memory objects. Although a network address is obtained by simply concatenating the virtual address per process and the system-wide process ID in usual cases, more complicated mapping such as that for a page shared among processes is possible because the mapping is completely under the control of MBP-light micorcode.

2.3.2 Memory command

The access to the intelligent memory system of JUMP-1 is performed by loads and stores just in the same way as ordinary systems, but the system produces various beneficial side effects on the addressed object. For example, cache coherence protocol for an object can be dynamically and

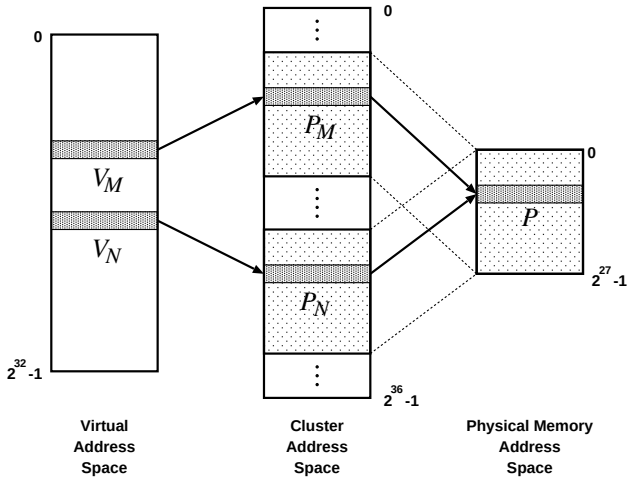

Figure 3. Address and memory command

explicitly specified. Another example is an operation on a synchronization structure. These effects are specified by **memory commands**.

A memory command is specified by a part of the cluster address. SuperSPARC+ provides 36-bit of physical address PA[35:0] to the external agents. Since the maximum size of a cluster memory is 128MB, only PA[27:0] is used as the physical memory address. The rest, PA[35:28] is used to transmit additional information from the processor to the external agents including the CC. PA[35:32] is used to specify memory command to the data addressed by PA[27:0] [1]. Thus 16(= 2^4) types of memory commands are defined to each address.

Figure 3 shows the relationship between the address spaces and the memory command. When a user process requires a new memory command for a memory object in a virtual page V_M, it tells the OS following items:

- virtual page V_M
- new memory command $N(N = 0, 1, \cdots, 15)$
- free virtual page V_N for the new memory command N

If the request is legal, the OS first finds the physical memory page P from the virtual page V_M. Then the OS allocates the specified virtual page V_N, and maps it to the cluster physical page P_N, of which PA[35:32] equals to the specified memory command N and PA[27:0] equals to the physical memory page P. Now the process has 2 virtual pages for the same object, V_M and V_N to invoke different functions by choosing these pages. Since a user process invokes the functions through the virtual memory, the functions are protected from illegal invocations by the ordinary memory protection mechanism. For example, V_N can be read-only even though read and write are allowed to V_M, if a write access V_N is illegal.

Accesses through different memory commands to the same data are distinguishable by their physical addresses. This is important because SuperSPARC+ and its primary data cache regard accesses through different memory commands to the same object as those to distinct objects. Theoretically maximum of four different memory commands to an object can be cached to the data cache at the same time (remember the data cache is 4-way set-associative). In actuality, CC controls so that at most two blocks of different memory commands are cached to the data cache.

[1]The rest, PA[31:29] represents the fixed attributes of the physical page.

3 Cache Controller

CC plays an important role in the JUMP-1 intelligent memory system, because CC functions as the interface between the element processor and the memory system.

The functions of the intelligent memory system are implemented as a co-operation of CC and MBP-light. While MBP-light is fully programmable, CC is completely hardwired because its speed for usual secondary cache operations is critical to keep the total system performance high. Thus we must have carefully chosen the functions to be implemented on CC. The complicated functions of the memory system are implemented mainly by MBP-light microcode, while CC acts just as a relay. On the other hand, the functions classified into the following two categories are directly supported by CC; that is, explicit control of cache coherence and cache-level supports for synchronization structures.

In the rest of this section, Section 3.1 describes the cache coherence control and related mechanism of CC, and Section 3.2 presents the high-level functions emphasizing how CC assists their efficient execution.

3.1 Cache coherence

The aim of the cache system of JUMP-1 is not only to provide coherent shared memory for programmability, but also to achieve high-performance by giving applications the chance and means to control the cache system explicitly. For example, an application may choose a coherence protocol appropriate to each of its data structures per access. The explict control of caching and decaching is also available by means of prefetch and invalidation. In addition to those functions, CC has various sophisticated mechanisms to reduce both traffics and latency on the conference management.

3.1.1 Outline of cache coherence

Cacheability In order to minimize the involvement of the primary caches in snooping operations, CC keeps multi-level inclusion property of the primary caches and E-cache. Thus a page marked cacheable in its page table entry, notifying SuperSPARC+ that its contents may be in the primary caches, is also cacheable to E-cache. A non-cachable page, however, may be also made cacheable to the E-cache by specifying appropriate memory commands. This E-cache-only cacheablility makes it possible that a synchronization structure is accessed in a special way keeping logical correctness as well as efficiency, as described in Section 3.2. The cacheablity of the third-level cache is fully under the control of MBP-light, so that an access with complicated side effects is performed correctly and efficiently.

Cache state E-cache state is similar to that of the MOESI protocol, except that it represents inter-cluster sharing status, as described in Section 3.1.3.

Coherence protocol CC supports both of two major coherence protocols, that is write-invalidate and write-update. In addition, CC also supports a few types of their combinations, for example, a protocol which invalidates other copies on read miss.

A programmer can specify the protocols through memory command as mentioned before, and can change it without any restrictions, even on-the-fly. Thus a data structure may have its own most appropriate protocol that may even varies in progress of the program execution. CC also provides some explicit control of a cache block such as invalidation of own copy or other copies, which is performed by a simple store to the block with the correspondent memory command.

3.1.2 Update protocol support

In a large scale distributed shared memory system like JUMP-1, write-update should be superior to write-invalidate that brings long read latency hardly to be hidden. Thus it is expected that applications mainly use write-update to reduce the read latency. However an easygoing use of write-update might cause traffic congestion by update requests to small pieces of data and/or those to no longer referred data. One of our solution is the dynamic choice of coherence protocol and the explict invalidation, which will solve the latter problem with small efforts of application programmers.

As the solution of the former problem, CC has a strong write-merge mechanism for update protocol. CC has 8-entry write-buffer of 32-byte on which write requests for write-update type store is merged, while those for write-invalidate are send as soon as possible. Thus even if a 32-byte cache block is modified by the sequence of 32 store-byte instructions, only one update requests is delivered from the CCunless the following exceptional cases.

Since a packet for the coherence maintenance targets a 32-byte cache block, an update request packet would have to carry a 32-bit indicator of byte modification if we coped with any types of write-merge. This scheme would cause significant (20 %, in fact) bus performance degradation and have to be said too earnest to the rare cases of partial byte modification. Thus we reduced the modification indicator to 8-bit to show the validity of each 32-bit word carried by the update request.

This reduction, of course, causes a problem that some sequence of byte or 16-bit halfword write-update type stores has no correspondent update request packet. The solution is to use write-invalidate in such an exceptional case. CC tries to merge a write request to its predecessors even if the size of it is smaller than 32-bit word. When CC decides to send the write request, if all the write requests are fortunately merged into a set of words, CC successfully sends the update request for the corresponding block. If unfortunately not, CC sends an invalidate request as the substitute for the update request. For example, a N-byte string copy to a shared region with write-update protocol will cause at most only two unexpected invalidate requests while at least $\lfloor N/32 \rfloor$ update requests will be successfully delivered.

3.1.3 Inter-cluster sharing status

If a transaction needs inter-cluster communication, the MBP-light creates and dispatches a thread for the request packet on the cluster bus. Although MBP-light is designed to handle this process efficiently, it is still too heavy to create a thread for every request packet on the bus which often initiates a cluster-local transaction.

Therefore CC manages inter-cluster sharing status of each block to reduce the possiblity to involve the MBP-light to a transaction. A cache block of which any copy can exist out of the cluster is marked as *globally-shared*. CC sends a write request packet on the bus with a bit that represents this status. MBP-light can immediately ignores a packet with the bit reset to not globally-shared. Thus MBP-light is free from a write transaction for a block private to the cluster permanently, such as that of local variables, or temporarily such as that of shared variables modified by write-invalidate type stores.

3.1.4 Cache prefetch

Since the D-cache of SuperSPARC+ is a blocking cache and the pipeline of SuperSPARC+ completely stalls when a D-cache read miss occurs, cache prefetching is one of the most important techniques to hide the load latency on JUMP-1.

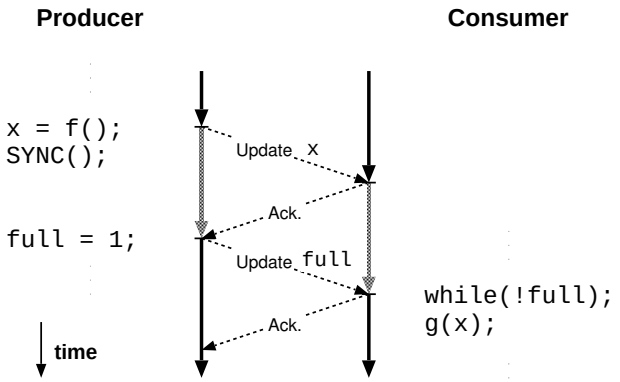

Figure 4. Producer-consumer communication on an ordinary shared memory system

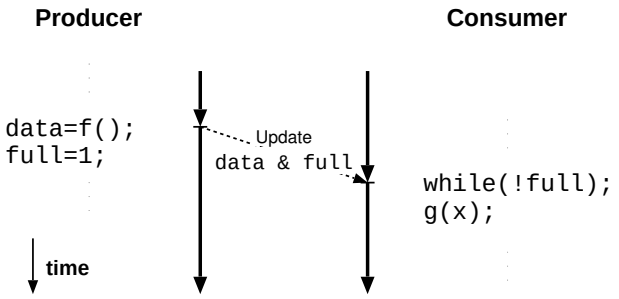

Figure 5. Producer-consumer communication using I-structure

Although SuperSPARC+ does not have instructions for D-cache prefetch, it is still available to prefetch a block to the E-cache. The prefetch function is invoked through a store-type memory command, which never stalls the pipeline because of cache miss.

Prefetching to E-cache, however, degrades effective bandwidth of the bus between the processor and the CC. On a normal load, E-cache fill is performed at the same time CC replies the block to the processor (see Figure 2). On a prefetch, however, E-cache fill and the transmission of the prefetched block from the E-cache to the processor are performed at different times. Therefore the effective bandwidth of the bus between the processor and the CC decreases by half compared with normal load.

CC has a *delayed* prefetch mechanism to avoid this problem. It has an 8-entry receive buffer that usually act as the queue for incoming packets from the cluster bus. This buffer is also used to delay E-cache fill of prefetched blocks. Each time the processor requires a block on its D-cache read miss, the buffer is looked up as well as E-cache tag. If a prefetched block is found in the buffer, CC provides the block just like E-cache hit, and at the same time performs E-cache fill.

The period the prefetched block stays in the receive buffer is specified by a control register. A 12-bit down counter is attached to each entry of the buffer to measure the period, and the prefetched block of which counter runs out to 0 is swept out to the E-cache. The control register specifies the initial value of the counters.

3.2 Synchronization structure

Ordinary distributed shared memory sytems inherently has drawbacks in handling message communication. In this subsection, we give the detailed explanation of the problems and JUMP-1's solution for them, under the following heads; one-producer communication and multiple-producers communication.

One-producer communication In ordinary distributed shared memory systems based on a relaxed memory model, each store requires an acknowledgment to ensure the completion of it. This means that the store operation is executed in a round-trip manner makeing the traffic double. Moreover, the processor must wait for the arrival of the acknowledgements at a synchronization point. This stall should be intolerable in fine-grained applications.

Figure 4 shows the simplest producer-consumer communication on a ordinary shared memory system based on a weak memory model, using shared variables `data` and `full`. The producer writes `data`, waits for the acknowledgment for the write, then turns on `full`. The consumer has to confirm that `full` is turned on, before it read `data`. This figure shows the best situation; `data` and `full` are already cached on each processor's cache, and write-update protocol is used, so the write to `data` and `full` is transmitted as fast as possible. Even in this situation, however, wasteful time appears as represented by gray arrows in the figure.

Our solution to this problem is **I-structure**[1] that has the capabilitiy of indicating its validity by itself. The basic idea utilizing I-structure is to execute the communication and synchronization in a one-way manner.

Each word of the memory of JUMP-1 has a tag that represents full/empty status of the word. A producer uses the store-type memory command for the I-structure, which sets the tag full at the same time it changes the value of the word. The consumer successfully load the new value after confirming the tag indicates full. The method to check the full/empty status of the word is described later. Figure 5 shows a communication using an I-structure. Since the store and the synchronization for it are executed atomically, the store does not need an acknowledgements, nor the producer needs to wait the completion of the store.

Multiple-producers communication Althogh the I-structure gives a good solution for the one-producer type communication, it cannot solely cope with the communication in which two or more producers are involved. That is, another mechanism for mutual exclusion is required.

The well known primitive for mutual exclusion is the atomic read and write, which JUMP-1 provides by means of SPARC's swap instruction as ordinary systems do. Although we made every effort to implement the swap efficiently, it is essentially hard to hide or reduce its latency because of the following reasons:

- All the atomic operations on an address in the system must be serialized at a certain point corresponding to the address.
- The atomic operation contains the read operation which essentially stalls the processor until its completion.

Thus we introduced a better alternative for mutually excluded message communication through FIFO queues, called **Q-structure**. A Q-structure has a queue of 64-bit word data and an address to which a set of memory commands is applied to access the queue. The enqueue of a data is performed by simply storing the data to the address with a store-type command for Q-structure. Load-type counterparts for the address obtain and remove the contents of the queue top. The mutual exclusion of the multiple producers sharing a Q-structure is solely performed by the MBP-light of a cluster where the queue body and usually the consumer reside. This means that only the MBP-light accesses the queue body with the data given by a producer. Thus, the producer does not need to lock the queue body, nor wait the completion of the enqueuing.

Although the entry of the Q-structure is 64-bit word, the combination with I-structure gives a general scheme of the communication with multiple producers. For example, a message of an arbitrary length is transmitted in the one-way manner by using I-structures for the message body and a Q-structure to send its base address.

The following shows the detail of the memmory commands for I-structure and Q-structure enphasizing how the CC contributes to their efficiency.

3.2.1 I-structure

As mentioned above, each 64-bit word of the main memory has a tag to represent the full/empty status of the word. E-cache also has an array for the tag so that an I-structure is cached with its tag. The D-cache, however, does not have any space for the tag, nor SuperSPARC+ has any means to check the full/empty status of the word directly. Therefore CC has a sophisticated mechanism to give the processor efficient ways to check the full/empty status.

Memory commands The store-type memory command for I-structure is called S-write. The command sets the tag full at the same time it changes the value of the addressed data.

As for read, the following three types of memory commands are available to check the full/empty status of the word:

F-read The most basic command to check the full/empty status of the word is the F-read command. This command returns the value $-1/0$ corresponding to the full/empty status of the addressed word.

Non-cacheable S-read The behavior of the S-read command depends on the cacheability of the target page. The S-read command to a non-cacheable page returns the value of the addressed word if the word is full, while causes a trap otherwise. The behavior of the S-read command to a cacheable page is described below.

This command can not be cached to the D-cache as its name shows, but is most efficient in the following case. Suppose an inter-processor communication with an I-structure in which the consumer reads the value only once per communication. If the spatial locality cannot be exploited, there is no merit to put the word onto D-cache because it is always invalidated by the producer's S-write. Under these modest assumptions, the merit of the command, implicit status check without conditional brances, is highlighted providing the possibility of the fullness is high enough.

Cacheable S-read The S-read command to a cache-able page returns the value of the addressed word when the word is full. Otherwise it returns the value 0.

If an application program regards the value 0 of the word as *meaningless* pattern such as a null pointer, this type is useful because one load instruction gives both the value and full/empty status.

The return values of these commands are summarized in the upper half of Table 2.

Cache support I-structure can be fully cached to the E-cache, and some types of read commands can be cached to the D-cache as well:

F-read This command is fully cacheable to the D-cache. When an F-read hits the E-cache, the CC replies a block containing $-1/0$ values corresponding to the full/empty status of the words, rather than the contents of the data array.

The $-1/0$ block can reside on the D-cache together with its counterpart for the values because of the set-associativity and the difference of phisical addresses.

Type		full	empty
I	F-read	-1	0
	Non-cacheable S-read	data	(cause trap)
	Cacheable S-read	data	0
Q	QC-read	-1	0
	Q-read	data	(cause trap)
	P-read	data	0

Table 2. The return value of I/Q-structure read commands

This makes it possible that both F-reads and ordinary loads to an I-structure block hit the D-cache even when they are performed alternately.

When an F-read misses the E-cache, the CC obtains not only the full/empty tags but also the 64-bit words of the missed block from the memory. Then the CC replies the $-1/0$ values to the SuperSPARC+ to minimize the latency, and finally updates the E-cache with the words. To read the whole block in addition to the full/empty tags on the E-cache miss is considered to be a kind of prefetch.

Non-cacheable S-read This command can not be cached to the D-cache because there is no way to cause a trap when an S-read to an empty word hits the D-cache. It is also impossible to delay loading a block containg empty words onto the D-cache until they becames full, because this may cause deadlock.

Cacheable S-read When the command misses the D-cache but hits the E-cache, the values of the full words from the E-cache data array and 0 for empty words are mixed to make the block to be loaded. The residence of an empty word in D-cache is allowed because hitting S-read correctly gives its result, that is 0.

The coherece protocol for the S-write is fixed to write-update because I-structure is for single-producer communication. Optional no write allocate policy is available for a producer which just writes data to I-structure without reading it. This option avoids the cache pollution and/or unnecessary remote block loading, while spatial locality is fully exploited by the write merge mechanism of CC.

Each command for I-structure has the polarity that defines whether the value 0 or 1 of full/empty tag represents the full status. After a communication through a set of I-structures is completed, for example, the producer and consumer can reuse it by simply switching the polarity in order to make the full words empty without accessing the words. In addition, these words are kept cached in most cases so that subsequent S-writes update the consumer's cache successfully.

3.2.2 Q-structure

A Q-structure is associated with a 32-byte block, whose first 64-bit word is the target of the memory commands for the Q-structure. The rest of the block is private to the MBP-light for the management of the queue, and is protected from accidental modification by processors.

Memory commands The Q-structure write command, Q-write enqueues the data at the bottom of the queue on the addressed block.

The Q-structure read commands have following variations:

QC-read Like the F-read for I-structure, the QC-read command returns the value $-1/0$ corresponding to the non-empty/empty status of the Q-structure.

Q-read Like the non-cacheable S-read for I-structure, the Q-read command dequeues and returns the queue top when the Q-structure is not empty, while causes a trap otherwise.

P-read Like the cacheable S-read for I-structure, the P-read command dequeues and returns the queue top when the Q-structure is non-empty, or the value 0 otherwise.

The return values of these commands are summarized in the lower half of Table 2.

Cache support The management of the queue body is performed by the MBP-light microcode. The CC assists it with a mechanism of prefetching the queue top.

All the commands to Q-structure are not cacheable to the D-cache, because each read and write must be observed by the CC and MBP-light to manipulate the queue properly. As for the E-cache, however, caching is allowed and beneficial to reduce the access latency by the following mechanism. The CC tries to keep the queue top is resident on the E-cache so that it immediately replies the value to a dequeueing request from the processor. After the reply, the CC forwards the request to the MBP-light to obtain the new top preparing for the following dequeue. Thus if the throughput of the data production and consumption and the queue management by the MBP-light are balanced, the consumer will always find the queue top in the D-cache.

4 Conclusion

We have described the functions of the intelligent cache controller CC, one of the main components of the JUMP-1 intelligent memory system. Its unique feature is the

sophisticated mechanism to support efficient message communication through I-structure and Q-structure as explained in detail.

As of February, 1997, the fabrication and test of the RDT chips has been completed. The chips for CC is being fabricated and the MBP-light will soon follow them. The first prototype is expected to be completed at the end of the next second quarter.

Acknowledgments

The basic idea of JUMP-1 and MBP owes much to Mr. T. Matsumoto [3].

We would like to Prof. H. Tanaka, Prof. Y. Kaneda, Prof. T. Sueyoshi, Prof. K. Hiraki, Dr. H. Amano, Dr. T. Kudoh, Dr. H. Nakajo, Dr. M. Kuga, and all the research members of the JUMP project for their support.

We would also like to thank Mentor Graphics Japan Corp. for providing their products and services as a part of Higher Education Program.

A part of this research was supported by the Grant-in-Aid for Scientific Research (C) #09680334 and Grant-in-Aid for Encouragement of Young Scientist #09780268 from the Ministry of Education, Science, Sports and Culture, and by the Parallel and Distributed Processing Consortium.

References

[1] Arvind and R. A. Iannucci. A critique of multiprocessing von neumann style. In *10th Int'l. Symp. on Computer Architecture*, pages 426–436, 1983.

[2] K. Hiraki et al. Overview of the JUMP-1, a MPP prototype for general-purpose parallel computations. In *ISPAN '94*, 1994.

[3] K. Hiraki and T. Matsumoto. Composite parallel processing architecture with two different processing element with two different grain size. *IEICE Technical Reports*, 90(144):25–30, 1993.

[4] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. ICPP '88*, pages 94–101, St. Charles, 1988.

[5] T. Matsumoto et al. MISC: A mechanism for integrated synchronization and communication using snoop caches. In *the 1991 Int'l. Conf. on Parallel Processing*, volume 1, pages 161–170, 1991.

[6] H. Nakajo and Y. Kaneda. A scalable I/O subsystem of a distributed shared-memory massively parallel computer JUMP-1. In *Trans. Information Processing Society Japan*, volume 37, pages 1429–1439, 1997.

[7] S. Sakai, K. Okamoto, H. Matsuoka, H. Hirano, Y. Kodama, and M. Sato. Super-threading: Architecture and software mechanisms for optimizing parallel computation. In *Int'l Conf. on Supercomputing 93*, 1993.

[8] H. S. Y. Yang, H. Amano and T. Sueyoshi. Recursive diagonal torus: An interconnection network for massively parallel computers. In *the 5th IEEE Symp. on Parallel and Distributed Processing*. IEEE, 1993.