TR-767

# The Unique Features of PIM/k: A Parallel Inference Machine with Hierarchical Cache System

by

S. Asano, S. Isobe & H. Sakai (Toshiba)

April, 1992

# The Unique Features of PIM/k:A Parallel Inference Machine with Hierarchical Cache System

Shigehiro Asano* Shouzou Isobe Hiroshi Sakai
TOSHIBA R&D Center

**Abstract**

This paper presents the unique features of a multiprocessor system, its hierarchical cache system, dedicated debug supports and performance evaluation mechanisms. For the hierarchical cache system, we present general consideration to extend a well known one-level cache protocol for the two-level cache. The new feature of it is a special replacement algorithm which keeps multi-level-inclusion property. For the second feature, we present novel mechanisms by which we can debug and evaluate the multiprocessor effectively. The mechanisms guarantee the same consequence with the same input.

## 1 Introduction

Parallel processing is now considered as a major key for high-performance machine. In the Japan's fifth generation computer project(ICOT), parallel processing has been studied extensively. As one of the experimental hardware project, we have been developing a parallel inference machine called PIM/k. A parallel inference machine is a special-purpose machine which runs a parallel logic language called KL1 [Goto88].

PIM/k employs a bus-based tree-structured, multi-level parallel cache system. In this architecture, transactions between processor-processor, or processor-memory are distributed on separated buses. For this reason, this architecture has potentially higher performance compared to conventional single-level bus-structured machines[Andrew87].

An important result in developing this type of architecture is the design considerations of the hierarchical cache memory. We have enhanced Berkeley protocol[Katz85] for multi-level parallel cache. For this type of cache, Multi-Level-Inclusion(MLI) property [Baer87] is the key to implement simple and efficient protocol. We propose a special replacement algorithm which ensures the MLI property.

Our major interest is to evaluate architecture of hierarchical cache system. For this purpose, we prepare some special hardware for debugging and performance evaluation. Its significant features include the robustness of repeatability. With this feature, software debugging and performance evaluation are simplified.

This paper focuses on two topics: One is the hierarchical coherent cache, and the other is the mechanisms for multiprocessor debugging and performance evaluation.

The rest of this paper is organized as follows. Next section describes why we chose hierarchical cache architecture for PIM/k. Section 3 presents overall architecture of PIM/k. Section 4 presents the mechanisms for debugging and performance evaluation as a multiprocessor system. Section 5 describes the cache consistency protocol and our replacement algorithm which maintains the MLI property. Finally we close with summary, the current status of the machine, and future directions. Appendix presents the complete description of our hierarchical cache protocol.

## 2 Motive of the hierarchical shared-memory architecture

Parallel execution models in general are classified into two basic categories concerning interprocess communication. One is message passing and the other is data sharing. The KL1 parallel execution

*e-mail asano@isl.rdc.toshiba.co.jp

model[Goto88] belongs to the latter. The parallel execution of a typical KL1 program may be considered as a lot of concurrent processes. These processes cooperate with one another to carry out a given task by sharing data within a unique memory address space. A process which wants to pass a value to other processes, allocates a new space and set the value into it. A process which finds out that the space has already been set a value, uses the value for its own execution. Thus, the naive memory architecture required by the KL1 parallel execution would be a large uniform memory which can be shared by numerous processing elements with little latency, though it is extremely difficult.

All the parallel inference machines being developed[Goto89] have non-uniform memory architecture. Each machine consists of clusters having from one to eight processing elements and a memory shared by them. However there is a diversity of intercluster connection. All the machines except PIM/k incorporate message passing mechanisms. That is, a processor which wants to refer a different cluster's memory has to send a message to the destination cluster in order to ask a processor of the destination cluster to do the required operation. On the other hand, PIM/k adopts a global memory shared among all the processors, for the purpose of reducing the overhead of the memory reference across clusters. A processor refers the global memory by simply executing load and store instructions.

To exploit a hierarchical coherent cache memory, it is very important to reduce transactions on its memory-level bus. For the hardware, the efficient coherent protocol is required. For the KL1 software, the load balancing mechanism across clusters should be designed carefully to avoid excessive memory reference across clusters. There is one more important aspect to consider. A KL1 program execution tends to consume the global memory at a very high speed because of the single assignment property and the parallel semantics of the KL1 language. Some preceding evaluation showed that a processor allocates a new word in every reduction. Suppose a cache line size is four words and the performance of a single processor is 500,000 reductions per second. Then 20 processors would do 10,000,000 store operations every second which would cause 2,500,000 block read and 2,500,000 block write transactions. This serious memory-level bus bottle-neck comes from the premise that a new space exists only in the global memory and not in the cache memories. Therefore, when a hierarchical cache memory is applied to a parallel inference machine, we think that the second-level cache memory should be large enough so that the average working set of a cluster can be kept in it. Our motive to enlarge the second-level cache, instead of the first-level cache, comes from the current VLSI technology trend that a processor chip with only a small cache is available.

Since PIM/k is the only parallel inference machine which employs a hierarchical cache memory, we have been interested in the following points from the architectural points of view.

- What kind of cache coherency protocols are possible and which one is appropriate for the KL1 program execution?
- What kind of hardware supports are useful to ease debugging and to collect accurate data for the performance evaluation?

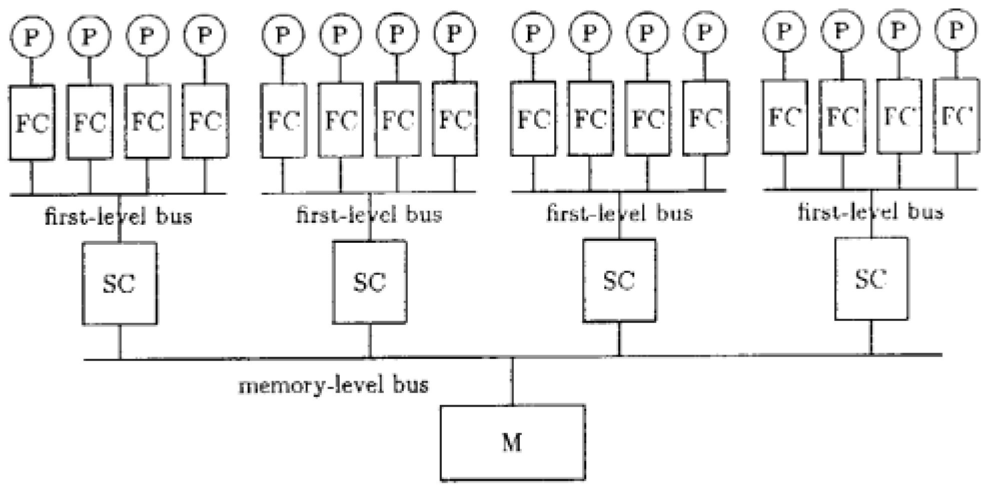
The next three sections present our decisions for these questions.

## 3 Overview of PIM/k architecture

### 3.1 Memory architecture

The memory space of each processor consists of two parts: the local space, and the shared space. The former includes the instruction local memory(128KBytes) for the code, and the data local memory(128KBytes) for the operand. The latter is shared among every processor and is accessed thorough each cache memory.

For the shared space, the memory architecture of PIM/k is a hierarchical cache system. Each processor has its own private cache, and four processors share a second-level cache through a first-level bus. Four second-level caches are connected to a shared-memory(1GBytes) by a memory-level bus. Fig.1 shows its memory architecture with 16 processors. Each private cache consists of the data cache(256KBytes)



P = Processor
FC = first-level cache
SC = second-level cache
M = memory

Figure 1: The memory architecture of PIM/k

and the instruction cache(128KBytes). The second-level cache is also divided into two parts: the data cache(4MBytes), and the instruction cache(1MBytes).

For the instruction cache, hardwared consistency mechanisms are not provided, because the cost of maintaining consistency by software is not so high. We prepare the flush operation which invalidates the instruction cache and keeps it consistent. We discuss only the data cache consistency mechanisms in the following sections.

## 3.2 Processor board

The processor board consists of a processor, the data local memory, the instruction local memory, the data cache, the instruction cache, a trace memory, two universal event counters, a slitcheck module and a diagnostic processor.

Fig.2 shows the block diagram of the processor board. The trace memory and the universal event counters are explained in the next section. The slitcheck module is for efficient interprocessor communications. The features of the processor include:

- RISC type 32bits processor;
- 4bits tag for symbol processing languages; and
- separated bus for data and instruction.

The diagnostic processor controls almost all the resources on the board. All the diagnostic processors within the system are connected to a supervisor processor(SVP). By sending requests to and receiving replies from the diagnostic processors, the SVP can control the execution of the processors and collect various information from the system.

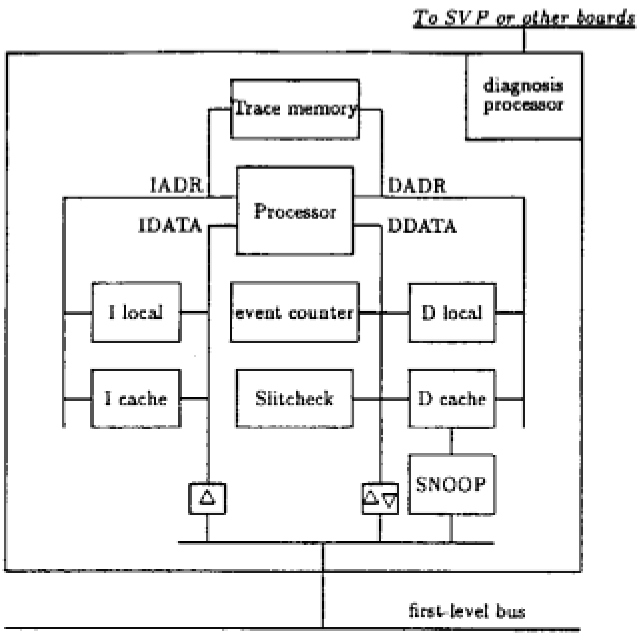


Figure 2: Block diagram of the processor board

## 4 Software debugging and performance evaluation support

Generally debugging of a parallel program is difficult compared to that of a sequential program. Some bugs are caused by an interaction between processors. The debugging requires the programmer to grasp complicated interactions between processors. Singlestep mechanisms which are useful in the case of a sequential program debugging, are less effective because they would influence interaction of the processors. This section presents our solutions for these problems.

### 4.1 Repeatability of a parallel processing

Repeated execution of a parallel program may result in different memory image even though it starts from the same initial state. This phenomenon makes debugging of an incorrect program very difficult. The phenomenon is caused by the change of accessing orders between processors which is attributed to the following asynchronous factors.

- Multiple independent clocks in the system.
- Modules which are not initialized.
- I/O operations like disk I/O whose execution time varies.
- Unexpected interrupts which notify a device error and so on.

To avoid these unfavorable factors, we have taken the following measures. For the first factor, all state-transitions of PIM/k are made synchronous to an identical clock. For the second factor, the bus arbitors are initialized when the system starts. Still, the reflesh operations of dynamic-RAM may disturb the repeatability, because a reflesh cycle is asynchronous to a program execution. To solve this problem, the start of a program execution is made suspended until the reflesh counter turns 0. The third factor requires a more sophisticated method. Our solution consists of the following steps:

1) A processor which wants to start an I/O operation interrupts all the other processors.

2) The other processors, receiving the interrupt, call a subroutine in which the processors suspend.

3) The processor does the I/O operation.

4) After finishing the I/O operation, the processor suspends itself.

5) The SVP, upon detecting all the processors are suspended, resumes their execution with their program counters set at a certain address.

By suspending the other processors during the I/O operation and resuming the execution from the same state, the order of access between processors remains unchanged.

The fourth factor, unexpected interrupt is unavoidable. But it is possible for an interrupt handler to inform a programmer of its occurrence.

## 4.2 Suspension mechanisms

In our system, suspension mechanisms play an important role in software debugging and performance evaluation because most of the resources can be examined only when the system is suspended.

PIM/k has the following hardwared means for suspension:

- *suspension address register* which suspends the system when a specified address is accessed.
- *suspension signal* which can be asserted by a processor or a diagnostic processor.
- *trace memory* which interrupts the system at a specified clock.

These mechanisms, controlled by a system software, provide a comfortable debugging environment. For example, the system can be suspended at an arbitrary clock. Here let the clock of suspension be N×32K+M cycles.

1) Set n = 0 which counts the number of interrupts from the trace memory.

2) Set the trace memory to interrupt in 32Kcycles.

3) Start or resume the program execution.

4) When an interrupt is received, if $n < N$ then increment n, else exit. The data in the trace memory or universal event counters will be saved to the SVP if required. (We will mention it later.)

5) If $n = N$ then set the trace memory to interrupt in next M cycles and goto 3), else goto 2).

Because slow-down caused by the above suspension mechanisms is only 0.3%, we use it in the normal execution by setting N very large.

## 4.3 Long trace with little distortion

Trace memory captures address traces of both data-access and instruction-fetch on every clock cycle. It enables programmers to get exact information about interactions between processors. For example, imagine a spin-lock case where processor1 has locked a variable and processor2 is spinning on the locked variable. When processor1 unlocks the variable, processor2 locks the variable. The trace memory enables a programmer to see which processor is spinning on the variable and when the processor gets the lock variable without affecting the program execution.

This trace facility is also useful as a tool for performance evaluation. Trace-driven simulation has been a popular method for evaluating cache memory systems [Smith82]. Up to now, the following methods have been used to obtain address traces: 1) using a hardware monitor, 2) modifying microcode to record memory accesses[Agarwal86], 3) causing trap on every instruction, 4) using a software simulator, and 5) instrumenting codes to record memory accesses. [Stunkel91] surveyed the above methods and pointed out the significance of the continuity and capturing speed of traces. To obtain continuous traces, a suspension mechanism of the processors is required while transferring data from the trace buffer to the I/O devices.

They gave negative remarks to the first two hardware-based methods. Their criticisms include insufficient length of continuous address traces, due to the lack of the suspension mechanism. The third method is easy to implement but tends to suffer from the execution-pattern distortion for multiprocessor system. The fourth method is accurate and can get continuous traces but the slowest. The last method is faster than the third and the fourth, but still slow-down is inevitable.

On the other hand, our method gets rid of the above disadvantages. Hardware-based mechanism of PIM/k can obtain very long, accurate, and continuous address traces by getting 32Kcycles-long trace at every step 4 of section4.2.

Our address trace does contain a distortion; If a processor being stalled received the interrupt, it does not accept the interrupt immediately. But, even in this case, the discrepancy is only several clocks between processors. Therefore the distortion is negligible compared to other methods which slows down the execution by 10 times or more.

### 4.4 Universal event counters

Each processor board has two 16bits counters which count various types of events on the board. For example, we can obtain read-hit ratio by setting the counters to count the number of read-hit and the number of read-access respectively.

The value of the counter is examined at step 4 of section4.2. The interrupt handler accumulates these values, or sends them to the SVP through the diagnostic processor. The SVP can show the dynamic behavior of the system on its screen. The typical resolution is 32 Kcyles because the trace memory raises an interrupt signal in every that interval.

For the sake of the repeatability, we need not measure all types of events at a single execution. Instead, we can obtain parameters separately by repeating the program execution with different setting of the counters.

## 5 Cache consistency protocol

A multiprocessor system with private caches introduces the cache consistency problem: Existence of multiple copies in private caches causes data inconsistency when a processor writes to its private cache. This section focuses on how we designed and implemented a cache consistency protocol for PIM/k.

### 5.1 Write invalidate versus write broadcast

In a coherent cache system, each cache memory maintains the state of each block appropriately on the arrival of the request from its own processor and the snooping bus. Hardware-based cache consistency mechanisms are basically classified into two types, namely write invalidate and write broadcast. The former keeps the consistency by invalidating the copies kept in other cache memories when a processor does a write operation. The latter broadcasts the new data to update each copy in other cache memories.

Some simulation studies[Eggers88] compares the performance of these two mechanisms. The results depend heavily on the type of applications. [Goto89] shows that write invalidate, copybacked protocol is preferred for KL1 execution. So we decided to follow the recommendation.

### 5.2 Design considerations of a hierarchical cache protocol

For the single-level cache system, several consistency protocols belonging to write invalidate type have been proposed. Each of them can be considered as a reduced set of the MOESI model[Sweazey86]. In the design of PIM/k hierarchical cache, we began to extend the MOESI model for a hierarchical cache and reached an extension of the Berkeley protocol[Katz85] though there are some other possible alternatives. This subsection describes the outline of our design considerations.

The MOESI model introduced the following three properties to describe a cache block state.

- valid or invalid
- shared or exclusive
- owned or unowned

The MOESI model has the set of five states described below which is the maximum set characterized by the above properties.

- M: Modified; valid, exclusive and owned
- O: Owned; valid, shared and owned
- E: Exclusive; valid, exclusive and not owned
- S: Shared; valid, shared and not owned
- I: Invalid; invalid

The other protocols can be described as reduced versions of this model. For example, in the Berkeley protocol, E and S are mapped to the same state UNO.

Now we try to extend the MOESI model for the hierarchical cache. The state of a first-level cache block can be described fully by the above three properties. However it is not true of the second-level cache. In order to implement an efficient protocol, it is necessary to manage the inherent data difference between the first-level and the second-level caused by the copybacked scheme. Therefore we decided to introduce the following fourth property.

- same or different

Suppose that a read request from the memory-level bus hits a second-level cache block with "owned same" state. The second-level cache can make a reply without consulting a first-level cache because the fourth property tells both the contents are the same. The fourth property reduces the traffic on the first-level bus.

Now we get back to applying the MOESI model to the first-level cache and show the split of E and S state is less effective than that in a single-level cache. For a single-level cache adopting the MOESI model, when a processor does a write operation to a data block with state E (exclusive notowned), the cache need not issue an invalidation command. Let us consider the same situation in the case of a hierarchical cache. Before the write operation, the first-level cache block has state E and the second-cache block has a state of "exclusive, same". The write operation causes difference between the first-level and the second-level cache and thus the fourth property of the second-level cache block must be maintained. Generally the first-level cache must send a request to change the fourth property of the second-level cache block whenever the state of its block changes from state E to state O. Therefore we decided to merge the states E and S of the first-level cache into one, namely the UNO state of the Berkeley protocol. As a result, we adopted the Berkeley protocol for the first-level cache.

Now we present our design considerations of the second-level cache protocol. The possible combination of the above four properties would identify at most 9 states[1]. However the following three states are impractical.

- 'notowned, different and shared'
- 'notowned, different and exclusive'
- 'owned, shared and different'

Now, we have six meaningful states.

In our actual implementation, however, we decided to merge these states into four states because:

[1] 2(owned or notowned)×2(shared or exclusive)×2(same or different)+1(invalid)=9.

- It requires less hardware resources, because four states are represented by only two bits.
- It enables us to make the first-level and the memory-level buses compatible, which eases the hardware debugging.

The final version of the state definition and the bus commands are summarized in section5.5.

## 5.3 Multi-level-inclusion property

While the most important role of the second-level cache is to provide the small first-level caches with their required data, the second-level cache also works as a filter of the bus traffic between the memory-level and the first-level buses. For example, suppose a second-level cache receives an invalidation command from its first-level bus. If the second-level cache has an appropriate block with exclusive property, it does not have to transfer the command to the memory-level bus. Moreover, a memory-level bus command for the data which a second-level cache doesn't hold, will not incur another command on its first-level bus.

Through this section, we will assume the following condition is satisfied.

- The memory location which exists in a first-level cache, must be held by the second-level cache directly connected to the first-level cache.

This constraint is called the Multi-Level-Inclusion(MLI) property and was first proposed by [Baer87].

## 5.4 Enhancement of the ownership

Here, we extend the ownership concept for the multi-level cache. The definition of ownership is given differently for each level.

- For the first-level cache, the ownership means that the cache block is to reply to a data request on the first-level bus, and to do a copyback to the second-level cache. The cache block with ownership has always valid data.
- For the second-level cache, the ownership means that the cache block is to reply to a data request on the memory-level bus, and to do a copyback to the memory. The main difference is that the cache block with ownership doesn't always have valid data.

Because the number of cache blocks with the ownership is at most only one for each level, there is at most only one module which replies to a data request for each bus. If no module replies to the data request, the lower module is responsible for the reply.

## 5.5 Definition of states and bus commands

Definition of the states for the first-level cache is the same as the Berkeley protocol.

- INV: Invalid, doesn't contain valid data.
- UNO: UNOwned, has a valid data copy, possibly is shared with other processors' cache memories, cannot be written before obtaining right to update, doesn't have ownership.
- NON: Not exclusive OwNed, has a valid data copy, possibly be shared among other processors, cannot be written before obtaining right to update, has ownership.
- EXC: EXClusive, has only one valid data, not be shared, can be written locally, has ownership.

Definition of the states for the second-level cache is the same as above except that of the state EXC. For the fourth property, UNO and NON have the property of "same", while EXC has "different".

- EXC: has not a valid data copy, but one of the first-level cache above has a valid data copy, has ownership.

Table 1: Bus commands

| | |
|---|---|
| RSH: | Read SHared,<br>Read data-block. |
| RFO: | Read For Ownership,<br>Read data-block, and obtain ownership. |
| WFI: | Write For Invalidation,<br>Invalidate other data-block. |
| WWI: | Write Without Invalidation,<br>Copyback data-block to second-level. |
| FAI: | Flush And Invalidation,<br>Force first-level cache which has ownership to copyback,<br>and Invalidate first-level cache. |
| FWI: | Flush Without Invalidation,<br>Force first-level cache which has ownership to copyback. |

The first-level bus commands are listed in Table 1. There are six commands, the first four from the first-level cache, and the rest from the second-level cache.

The memory-level bus commands are the same as Table1 except they don't have FWI or FAI.

The detail of the protocol behavior will be presented in the appendix.

## 5.6 Replacement algorithm

This section describes our replacement algorithm which keeps multi-level-inclusion property.

### 5.6.1 Previous replacement algorithm

It should be noted that every replacement algorithm in the second-level cache doesn't keep the MLI property. Here we prove that the LRU algorithm violates the MLI property. Assume that there are two processors, each of which has its own private cache (first-level direct-mapped cache) and a second-level cache which is two-way-set-associative.

Table 2: Violation of MLI

| sequence | P1 | P2 | WAY1 | WAY2 |
|---|---|---|---|---|
| At first P1 access location 1 | 1 | - | 1 | - |
| P2 access location 2 | 1 | 2 | 1 | 2 |
| P2 access location 3 | 1 | 3 | 3 | 2 |

When P2 has an access to location 3, the LRU algorithm replaces the oldest entry location 1 which exists in the WAY1 of the second-level cache. As a result, the data of location1 exists in P1's first-level cache, but not in the second-level cache. Thus the MLI property is violated.

To keep the MLI property, [Baer87] shows a solution that when an entry is to be replaced from the second-level cache, all the corresponding entries of the first-level cache should be copybacked beforehand.

This method has two disadvantages of increase in bus transaction and decrease in hit-ratio of the first-level cache caused by invalidation of valid copies used by other processors.

### 5.6.2 U-bit mechanism

We introduce a U-bit mechanism which is used in our new replacement algorithm. The basic idea of our replacement strategy is to select a replacement block which is not occupied by other processors. A U-bit which is attached to each set of a second-level cache, keeps track of which processor uses which way. A second-level cache selects the replacement block according to the U-bit information.

Here, we explain our algorithm under the following conditions. The one is to simplify the explanation, and the algorithm still works in the case of a set-associative cache. The second one is a requirement which comes from the above strategy. The last one is a general requirement of the MLI property.

1) Each first-level cache is direct-mapped.

2) Each second-level cache is N-way set-associative. N is the number of processors connected to a second-level cache.

3) The number of sets in a second-level cache is larger than or equal to that of first-level cache.

We denote U-bit as U[m,n], a boolean function, where m is the way number, and n is the processor number. U[m,n] is set to one when processor n use the entry of way m of second-level cache, otherwise set to zero.

Example: Imagine three processors are connected to a second-level cache.

- P1's location 2000 occupies way_1 of the second-level cache.
- P2's location 1000 occupies way_2 of the second-level cache.
- P3's location 1000 occupies way_2 of the second-level cache.
- Both location 1000, 2000 and 3000 occupies same set of the second-level cache.

In this situation, U-bit has following values:

- U[1,1]=1, U[1,2]=0, U[1,3]=0.
- U[2,1]=0, U[2,2]=1, U[2,3]=1.
- U[3,1]=0, U[3,2]=0, U[3,3]=0.

Fig.3 illustrates this example.

When a second-level cache receives a command and hit to way_i U-bit must be changed in the following manner.

Let p be the processor which issued the command.

1) For a RSH command, U[i, p]=1, U[j, p]=0 for all j≠i,

2) for a RFO command, U[i, p]=1, U[j, p]=0 for all j≠i, U[i, q]=0 for all q≠p,

3) for a WFI command, U[i, q]=0 for all q≠p,

4) for a WWI command, U[i, p]=0,

### 5.6.3 Selection of the replacement block

When a command issued by a first-level cache does not hit any way of the second-level cache (in this case, because of the MLI property, no first-level caches reply to the command), the second-level cache should allocate an appropriate way so as to hold the required block data. Our replacement algorithm selects the replacement block according to the following rules.

**rule 1)** if there is a way which is INV, use it.

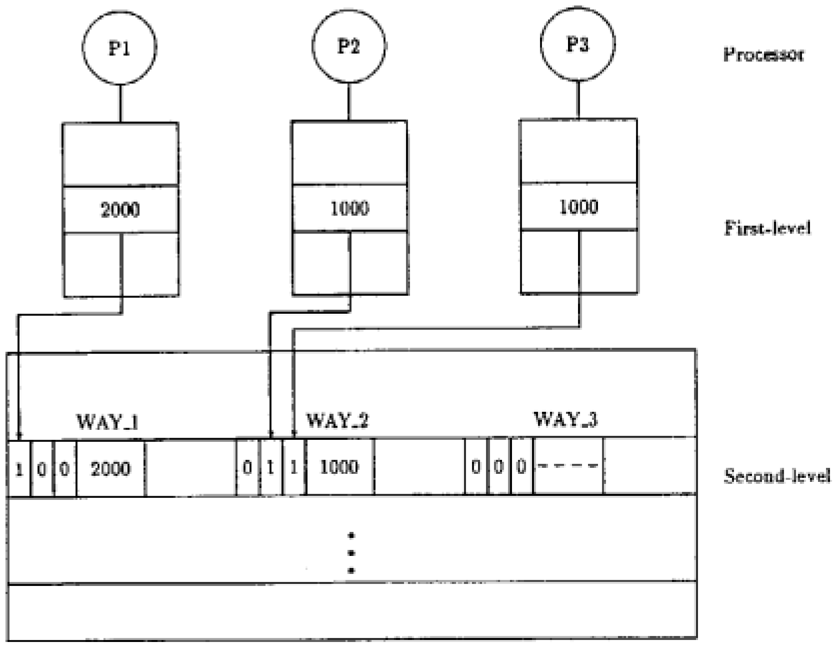


Figure 3: Example of the U-bit

**rule 2)** else if there is a s such that U[s,**p**]=0 for all **p**. use the s-th way.

**rule 3)** else use the s-th way such that U[s,**p**]=1. Here **p** is the processor which issued the command.

The first two rules try to find a block which does not have a valid copy in any upper first-level caches. The last rule guarantees that a block which has a valid entry in any upper first-level cache except the requestor's is not selected. The requirements described in section 5.6.2 ensures the last rule is always applied successfully. Whenever the last rule is applied, the requestor's first-level cache always purges the block corresponding to the second-level cache block. Therefore our replacement algorithm manages to maintain the MLI property.

## 5.7 Lock operation

Lock and synchronization operations, like SWAP and CAS are essential for multiprocessing. In an ownership protocol, these operations are implemented as follows.

**1)** If the data block has a EXC state, do an atomic read-write operation immediately.

**2)** Otherwise, get ownership and invalidate other copies before doing an atomic read-write operation.

Here "atomic" means blocking any snoop requests between the read and write operation.

## 5.8 Deadlock

Generally, a system which allows multiple requests to multiple resources simultaneously faces the deadlock problem. In hierarchical cache, there is a possibility of deadlock, which is illustrated in Fig.4

**1)** A request from processor P1 causes read miss in both first-level cache F1 and second-level cache S1, then S1 tries to send a RSH through memory-level bus M1,

**2)** At the same time, an WFI arrives at S1 as the consequence of a write request from another processor P2.

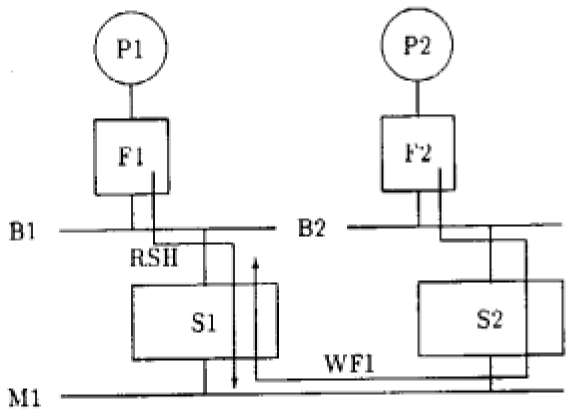


Figure 4: Example of deadlock

In our implementation, a second-level cache detects the deadlock condition and resolves it as follows:

1) Sending NAK signal to the first-level cache to cancel the command.

2) Transferring the request from the memory-level bus to the first-level bus.

3) Releasing NAK signal to let the first-level cache reexecute the suspended operation from the beginning.

Since our implementation employs non-split-bus, the second-level cache has only to detect above condition. However, in the case of a split-bus, the system would have to memorize outstanding requests in order to detect address collision between requests.

### 5.9 Possible enhancements

[Goto89] proposed some protocol optimizations for KL1. They introduced a Direct Write (DW) which avoids a fetch operation on the write attempt to an uninitialized area. They reported DW reduce bus traffic considerably.

In the case of two-level cache, however, DW is less effective because an additional command is required to allocate an entry in the second-level cache to ensure the MLI property. If DW operation allocated a cache entry in the first-level cache without notifying to other caches, it would not allocate the cache entry in the second-level cache, which would violate the MLI property Therefore, the first-level cache must notify second-level cache whenever DW operation takes place.

## 6 Conclusion

In this paper, we have presented the unique features of PIM/k, a hierarchical cache memory based multiprocessor. In the design of its cache consistency protocol, we began to enhance the MOESI model and finally decided to employ a Berkeley-like protocol. We also proposed a new replacement algorithm which ensures the multi-level-inclusion property.

PIM/k employs new mechanisms which ensure the deterministic execution of the processors. These mechanisms are useful for multiprocessor software debugging and performance evaluation. Our recent experience have proved that the deterministic execution is also useful in hardware trouble shooting because some bugs appear after the machine has been released to the software people. The mechanisms also captures continuous address traces with small distortion.

The current status is as follows. We have already finished building a prototype hardware and confirmed that the machine executes simple parallel programs such as *12 queen* or *prime-number-generation* with four processors. We are planning to complete a new version with 16 processors by the end of next March. For the basic software, we are developing an efficient KL1 system which is designed to exploit the hierarchical memory architecture. On top of it, a parallel operating system and some application programs written in KL1 will be running. For the evaluation, we are also developing a software simulator in order to compare various cache-memory architectures extensively. We are now checking the simulator's validity by comparing the result with that of the real hardware under the same system parameters.

## Acknowledgement

We are grateful to Dr. Taki and Dr. Hirata of ICOT who have been leading the PIM project.

## References

[Agarwal86] A.Agarwal, R.L.Sites, and M.Horowitz, 'ATUM:A New Technique for Capturing Address Traces Using Microcode', Proceedings of the 13th Annual International Symposium on Computer Architecture, June 1986, pp. 119-127.

[Andrew87] Andrew W. Wilson Jr., 'Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors', Proceedings of the 14th Annual International Symposium on Computer Architecture, June 1987, pp. 244-252.

[Baer87] Jean-Loup Baer and Wen-Hann Wang, 'Architectural Choices for Multilevel Cache Hierarchies', Proceedings of the 1987 International Conference on Parallel Processing, 1987, pp. 258-261.

[Eggers88] Susan J.Eggers and Randy H.Katz, 'A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation', Proceedings of the 15th Annual International Symposium on Computer Architecture, May 1988, pp. 373-382.

[Goto88] Atsuhiro Goto, Masatoshi Sato, Katsuto Nakajima, Kazuo Taki, and Akira Matsumoto, 'Overview of The Parallel Inference Machine Architecture (PIM)', Proceedings of the International Conference on Fifth Generation Computer Systems, Nov. 1988, pp. 208-229.

[Goto89] Atsuhiro Goto, Akira Matsumoto and Evan Tick, 'Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures', Proceedings of the 16th Annual International Symposium on Computer Architecture, May 1989, pp. 25-33.

[Katz85] R.H.Katz, S.J.Eggers, D.A. Wood., C.L.Perkins, R.G.Sheldon, 'Implementing A Cache Consistency Protocol', Proceedings of the 12th Annual International Symposium on Computer Architecture, June 1985, pp. 276-283.

[Smith82] Alan Jay Smith, 'Cache Memories', ACM Computing Surveys, Vol.14, No.3, Sept. 1982, pp. 473-530.

[Stunkel91] Craig B.Stunkel, Bob Janssends and W.Kent Fuchs, 'Address Tracing for Parallel Machines', IEEE COMPUTER, Jan. 1991, pp. 31-38.

[Sweazey86] Paul Sweazey and Alan Jay Smith, 'A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus', Proceedings of the 13the Annual International Symposium on Computer Architecture, June 1986, pp. 414-423.

## Appendix: Protocol behavior

The complete behavior of our protocol is described by the following independent tables.

1) Table A.1 shows the first-level cache behavior in response to the processor.

2) Table A.2 shows the first-level cache behavior in response to the first-level bus commands.

3) Table A.3 shows the second-level cache behavior in response to the first-level bus commands.

4) Table A.4 shows the second-level cache behavior in response to the memory-level bus commands.

5) Table A.5 shows the memory behavior which is responded to memory-level bus commands.

The notations commonly used in the tables are listed below.

- "*action/state*" means " after *action* finished, change to *state*".
- "no" means no action, no state change is taken.
- "never" means this case is never occur.
- "-" means no action is taken.

Table A.1: processor access vs state of first-level cache

| | Read | Write |
|---|---|---|
| INV miss | RSH/UNO | RFO/EXC |
| EXC | no | no |
| NON | no | WFI/EXC |
| UNO | no | WFI/EXC |

- if read or write miss and state is EXC or NON then issue WWI then treat as INV.

Table A.2: first-level bus access vs state of first-level cache

| | RSH | RFO | WFI | WWI | FAI | FWI |
|---|---|---|---|---|---|---|
| INV miss | no | no | no | no | no | no |
| EXC | datatoC/NON | datatoC/INV | never | never | datatoC/INV | datatoC/UNO |
| NON | datatoC/NON | datatoC/INV | -/INV | never | datatoC/INV | datatoC/UNO |
| UNO | no | -/INV | -/INV | no | -/INV | no |

- action "datatoC" means return data to first-level bus.

Table A.3: First-level bus access vs state of second-level cache

| | RSH | RFO | WFI | WWI | FAI | FWI |
|---|---|---|---|---|---|---|
| INV miss | RSHtoM datatoC/UNO | RFOtoM datatoC/EXC | never | never | never | never |
| EXC | no | no | no | -/NON | never | never |
| NON | datatoC/NON | WFItoM datatoC/EXC | WFItoM/EXC | never | never | never |
| UNO | datatoC/UNO | WFItoM datatoC/EXC | WFItoM/EXC | never | never | never |

- "RSHtoM" means "RSH command is issued to memory-level bus".
- "RFOtoM" means "RFO command is issued to memory-level bus".
- "WFItoM" means "WFI command is issued to memory-level bus".

Table A.4: Memory-level bus access vs state of second-level cache

| | RSH | RFO | WFI | WWI |
|---|---|---|---|---|
| INV miss | no | no | no | no |
| EXC | FWItoC datatoM/NON | FAItoC datatoM/INV | never | never |
| NON | datatoM/NON | if not(ubza) then (WFItoC datatoM)/INV | if not(ubza) then (WFItoC)/INV | never |
| UNO | no | if not(ubza) then (WFItoC)/INV | if not(ubza) then (WFItoC)/INV | no |

- "FWItoC" means "FWI command is issued to first-level bus".
- "FAItoC" means "FAI command is issued to first-level bus".
- "datatoM" means "return data to memory-level bus, and assert cache signal".
- "if not(ubza)" means "if all of the U-bits of the entry is zero, following action is taken".

Table A.5: memory-level bus access of memory

| RSH | RFO | WFI | WWI |
|---|---|---|---|
| if not(cache) then (datatoM) | if not(cache) then (datatoM) | no | write data to memory |

- "if not(cache)" means "if cache signal* is asserted, following action is taken".
  * cache signal is asserted when EXC or NON block reply to request.