ICOT Technical Report: TR-768

TR-768

PIM/k: a Parallel Inference Machine with a Cache Hierarchy

by
H. Sakai, S.Asano, A. Nakase
S. Isobe, H. Muratani & T. Takewaki (Toshiba)

April, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

PIM/k: a Parallel Inference Machine with a Cache Hierarchy

Hiroshi Sakai, Shigehiro Asano, Akihiko Nakase
Shouzou Isobe, Hirofumi Muratani, Toshiaki Takewaki

TOSHIBA Corporation

Abstract

This paper presents the features of PIM/k, one of the Parallel Inference Machines being developed under the Japan's Fifth Generation Computer Project. One of the primary goals of the project is to realize a fast inference machine by means of parallel processing. Towards this goal, PIM/k employs a cache hierarchy to reduce the memory access latency, while the other PIMs employ inter-cluster networks to obtain high band-width of the whole systems. This paper describes the design considerations of a hierarchical cache consistency protocol with an efficient replacement algorithm. The KL1 system software which was initially designed for the inter-cluster networks of the other PIMs should be tuned for the cache hierarchy. This paper shows that the cache hierarchy is advantageous from the software points of view and that stop-and-copy GC employed by the PIMs requires adapting in order to avoid an elficiency problem. Though PIM/k is still under development, this paper also reports a KL1 program evaluation on the actual machine with 4 processing elements (PEs).

1 Introduction

Parallel processing is one of the most important keys for faster execution in the knowledge engineering domain. Under the Japan's Fifth Generation Computer Project, several types of Parallel Inference Machines (PIMs) are being developed. All PIMs employ a common parallel logic programming language KL1 which was derived from Flat GHC [Ueda87].

To achieve high performance with hundreds of processing elements (PEs), most of the machines have employed inter-cluster networks to obtain high band-width of the whole systems. Each cluster has from one to eight PEs and a shared memory. While a PE can have access to its memory directly by load and store instructions, an inter-cluster memory access requires a request and a reply message exchange. Though this approach has the advantage of scalability, a major disadvantage may be the inter-cluster memory access latency caused by the message handling software overhead. For example, a KL1 program which contains a database referred frequently by the rest part of the program may suffer from a serious performance bottle-neck at the cluster where the database resides.

To reduce memory access latency, a Parallel Inference Machine PIM/k employs a cache hierarchy [Wilson87]. It enables a PE to have access to a different cluster's shared memory by load and store instructions. We have designed a consistency protocol for a cache

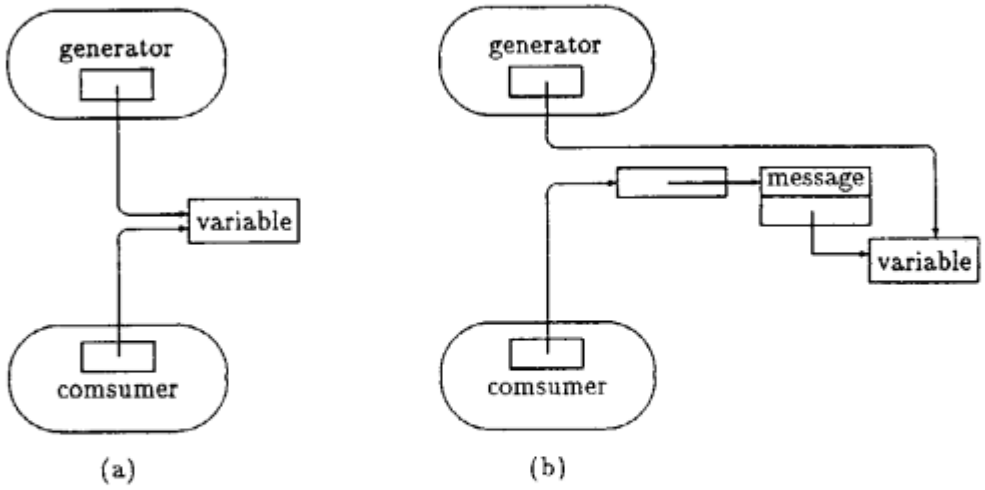

Figure 1: The shared memory based communication among KL1 processes

hierarchy and have tuned the KL1 system software to exploit our memory architecture. This paper discusses the features of PIM/k, especially the application of a cache hierarchy.

The rest of this paper is organized as follows. First, we present the motive for applying a cache hierarchy to PIM/k. Secondly, the design considerations of the PIM/k memory architecture, are discussed. Then we describe the KL1 system software tuned for the cache hierarchy and discuss the advantages of the cache hierarchy and a possible efficiency problem. Finally, we describe our current status and a KL1 program evaluation on the actual hardware with 4 PEs.

2 Motive for applying a cache hierarchy to PIM/k

2.1 KL1 parallel execution on a shared memory

The parallel execution of a KL1 program [Goto88] and a shared memory model are well matched in terms of inter-process communication and automatic load balancing.

Inter-process communication is classified into two basic categories, namely message passing and data sharing. KL1, belonging to the latter, may be considered a set of concurrent processes which share data within a unique memory address space.

Consider the communications between generator and consumer processes: When there are no messages, both of processes have pointers to the same variable as illustrated in Figure 1(a). In this case, the consumer which finds out no message would suspend its execution until the generator sends a message. When the generator gets ready to send a message, it creates a list structure with the message and processes its pointer as illustrated in Figure 1(b). The consumer can then process this message cell and attain the pointer to the new variable, for further communication. Then the inter-process communication returns to the state illustrated in Figure 1(a). This shared memory based communication

is suitable for knowledge processing because a message containing pointers to structured data can be passed efficiently so that processes may share the data.

Automatic load balancing during a program execution is inevitable for a PIM to achieve high performance. Generally, it is essentially impossible to predict how many and what kind of subtasks are created during a program execution within knowledge processing domain. Even if possible, it would be heavily time dependent. A PIM realizes the automatic load balancing by migrating goals between from heavy loaded PEs to light loaded PEs. Since goals may have pointers, a goal migration within a uniform address space is more efficient than that across different address spaces.

2.2 Applying a cache hierarchy to PIM/k

Based on the above consideration, PIM/k has adopted a cache hierarchy as illustrated in Figure 2. PEs of a cluster share a second-level cache through a first-level bus, which means a second-level cache may be considered as a cluster's shared memory. The second-level caches are connected to a global memory through a memory-level bus. Therefore the memory-level bus may correspond to the inter-cluster networks of the other PIMs.

While the shared memory architectures [Lenoski91] [Warren88] have been proposed for multiprocessors, we adopted a cache hierarchy for two reasons. Firstly, the allocation of memory to each cluster should change, because the amount of memory is application and time dependent. Secondly, the evolution of VLSI will deliver inference processors with built-in first-level cache. This will bring the cache hierarchy a reality.

Exploiting a cache hierarchy with multiple processors necessitates the reduction of transactions on its memory-level bus. For the hardware, an efficient cache protocol is required. For the KL1 system software, the load balancing mechanism across clusters should be designed carefully to avoid excessive inter-cluster memory access. There is one more important aspect to consider; A KL1 program execution tends to consume the heap space at a very high speed because of the single assignment property and the parallel semantics of the KL1 language. It is known that a goal reduction of practical programs allocates two words in average from the free heap space. Suppose a cache line size is four words and the performance of a processor is 400,000 reductions per second. Then 20 processors would do 8,000,000 store operations every second which would cause 2,000,000 block read and 2,000,000 block copyback transactions. To avoid this amount of transactions on the memory-level bus, the second-level cache should be large enough so that it keeps the free heap space. Then, the memory access transactions described above would be enclosed within a first-level bus.

3 Memory architecture

3.1 Overview

As is illustrated in Figure 2, the actual PIM/k has four clusters, each of which has four PEs. A PE is a RISC-type 32 bit micro-processor with tag architecture. A PE has a

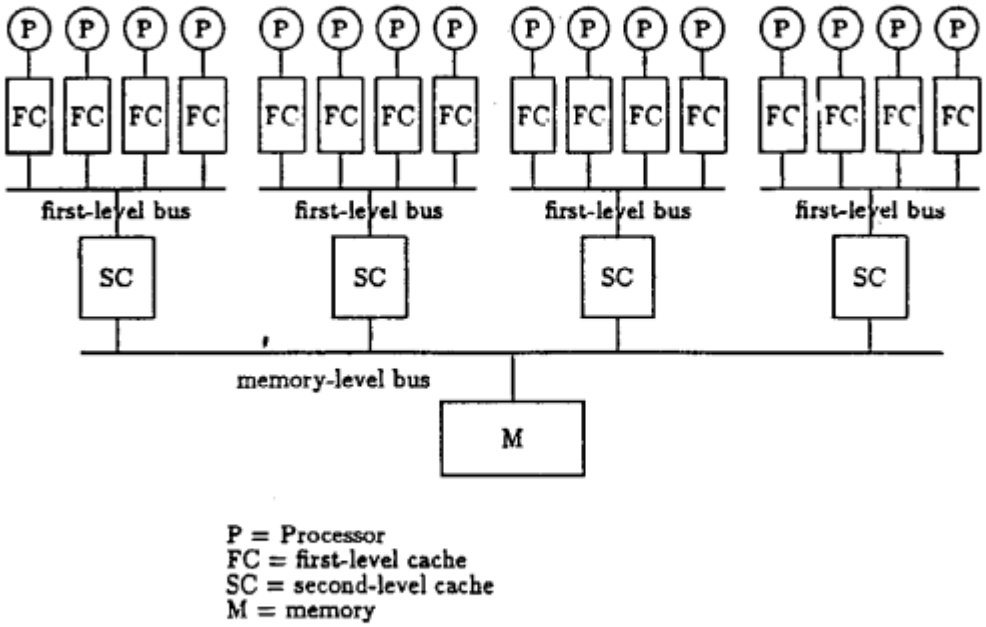

Figure 2: The memory architecture of PIM/k

private memory as well as a first-level cache, since the KL1 system software uses run-time routines and small fixed amount of memory frequently. Following the Harvard architecture of the processor, the private memory consists of an instruction memory (128KBytes) and a data memory (128KBytes). The first-level cache consists of an instruction cache (128KBytes) and a data cache (256KBytes). The second-level cache also consists of an instruction cache (1MB) and a data cache (4MB). The global memory size is 1 GByte. To reduce hardware, no consistency mechanism is provided for the instruction cache however a flush mechanism which invalidates the tag memory is adopted.

3.2 Design considerations of a hierarchical cache protocol

[Wilson87] proposed a hierarchical cache memory and reported its performance evaluation by a simulation using an analytical model. However, no results, as far as we know, have shed light on the design considerations of the hierarchical cache consistency protocol. Our hierarchical cache employs a write invalidate, copybacked protocol following [Goto89] which reports its superiority for KL1 execution. We think that our design procedure can also be applied to other types of hierarchical cache memories.

3.2.1 Enhancement of the MOESI protocol to a cache hierarchy

There are several consistency protocols for a single-level cache, and most of them can be considered as a reduced version of the MOESI model [Sweazey86]. In the design of PIM/k cache protocol, we began to enhance the MOESI model for the cache hierarchy and finally reached an extension of the Berkeley protocol [Katz85] though there are some other possible alternatives.

The MOESI model introduced the following three properties.

valid or invalid:
indicates whether the block has valid data or not.

shared or exclusive:
indicates whether the block data is shared with other cache memories or not.

owned or unowned:
indicates whether the block has ownership or not. The ownership means that the block is to reply to a data request on the bus, and to do a copyback to the memory. The block with ownership has always valid data.

These properties characterize the five states of a cache block listed in Table 1. Some other protocols for a single-level cache can be obtained by merging some of the above five states. For example, the Berkeley protocol merges states E and S into state UNO.

Table 1: Five states of the MOESI model

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

Now let us consider applying the MOESI model to a cache hierarchy. It can be applied to the first-level cache by simply changing the word "memory" to the word "second-level cache". However it is not true of the second-level cache. Consider the case where a processor does a write operation. The relevant first-level cache block turns its state to state M and holds the new data. For the second-level cache, the corresponding block also turns its state to state M, but the block data remains unchanged. This suggests that the ownership definition should be changed so that either the second-level cache block or some of its first-level cache blocks has valid data.

Then, suppose that a read request from the memory-level bus hits a second-level cache block with "owned" state. If it were not for any mechanism to determine whether the data is valid or not, the second-level cache would always have to consult the first-level cache with ownership. In order to reduce this kind of traffic on the first-level bus, we decided to introduce a fourth property associated with each second-level cache block.

same or different:

indicates whether the block has valid data or not.

Now we show the split of states E and S in the first-level cache is less effective than in the case of a single-level cache. For a single-level cache based on the MOESI model, a write operation to a block with state S has to issue an invalidation command, while one to a block with state E need not issue any command. Let us consider the same situation in the case of a cache hierarchy. Just 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 from a processor causes difference between the first-level and the second-level caches and thus to change the fourth property of the second-level cache block into "different", an additional command from the first-level cache is required. Therefore we decided to merge states E and S of the first-level cache into one, namely state UNO of the Berkeley protocol. As a result, we decided to adopt the Berkeley protocol for the first-level cache.

Getting back to the second-level cache protocol, the possible combination of the above four properties would be 9 states[1]. However the following three states:

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

are impractical and can be removed with no penalty. The final version of state definition and the bus commands are summarized in the next section.

Thus, we have six meaningful states. In our actual implementation, however, we decided to merge the six states into four because of the following reasons.

- It requires less hardware, 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 hardware debugging.

3.2.2 Definition of states and bus commands

The state definition for the first-level cache is the same as the Berkeley protocol.

INV: Invalid, does not contain valid data.

UNO: UNOwned, has valid data, possibly be shared among other processors' caches, cannot be written before obtaining right to update, does not have ownership.

NON: Not exclusive OwNed, has valid data, possibly be shared among other processors' caches, cannot be written before obtaining right to update, has ownership.

EXC: EXClusive, has valid data exclusively, not be shared, can be written locally, has ownership.

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

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

EXC: EXClusive, does not have valid data, but one of the first-level caches above has valid data, has ownership.

The first-level bus commands are listed in Table 2. The first four commands come from the first-level cache memories, while the other two are used by the second-level cache to realize a snooping mechanism. The memory-level bus commands are the first four commands in Table 2 since the shared memory does not have a snooping mechanism.

Table 2: Bus commands

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

3.3 Replacement algorithm for the second-level cache

3.3.1 Multi-level-inclusion property

While the most important role of the second-level cache is to provide the first-level caches with required data, it also works as a filter of the bus traffic between the memory-level and the first-level buses. For example, if an invalidation command from a first-level cache hits a second-level cache block with exclusive property, the second-level cache does not have to transfer the command to the memory-level bus. Likewise, if a memory-level bus command does not hit the block of a second-level cache, the second-level cache does not have to transfer the command to its first-level bus.

In the above discussion, we have assumed implicitly that the memory location which exists in a first-level cache must be held by its second-level cache. This constraint is called the Multi-Level-Inclusion (MLI) property [Baer87]. This property puts the following constraint to the second-level cache memory.

constraint A: The second-level cache should be at least M×N-way set associative, where M is the number of its first-level cache memories and N is the number of ways of each first-level cache memory.

3.3.2 Replacement algorithm

Consider the case where a memory access request by a processor causes a cache miss both in the first-level and the second-level caches and an appropriate cache block has to be selected to hold the required block data in each cache. The second-level cache block also has to be selected to keep the MLI property. It should be noted that some replacement algorithms for a single-level cache, including the LRU algorithm, violate the MLI property.

[Baer87] presented a replacement algorithm which keeps this property. The idea was that when an block is to be replaced from the second-level cache, all the corresponding blocks of the first-level cache should be copybacked beforehand. This method has two disadvantages, namely increase in bus transaction and decrease in hit-ratio of the first-level cache caused by invalidating valid copies still used by other processors.

We introduce an efficient algorithm which consists of the following three rules.

rule 1) If there is a cache block with state INV, select it.

rule 2) Otherwise, if there is a cache block which does not have any corresponding blocks in the upper first-level caches, select it.

rule 3) Otherwise select the cache block which corresponds to the first-level cache block to be replaced.

We now prove that the above rules always keep the MLI property. When either the rule 1 or 2 is applied, a block to be replaced does not have any valid data in its corresponding first level cache blocks. When the last rule is applied, the constraint A ensures that the replaced block does not have any valid copies other than the block to be replaced in the first-level cache. This replacement algorithm needs no additional commands to keep the MLI property and can be efficiently implemented by a small amount of hardware [Asano92].

3.4 Debugging support

A powerful debugging environment is necessary especially in the development of a PIM. Generally, parallel execution of a KL1 program almost always results in different memory image, yet they are all correct. Goal reductions which create new KL1 objects compete with one another for allocating heap area for the objects. Therefore some asynchronous factors of hardware would affect the address of each KL1 object, which, in turn, affects cache hit or miss and increases the asynchronousness among PEs.

To ease the debugging, PIM/k was designed to achieve repeatable parallel execution [Asano92]. Although it is disturbed by the arrival of an asynchronous I/O request from a front-end processor or disk devices, we have appreciated it in debugging hardware and software errors which appear after several minute parallel execution. It is realized by the following mechanisms:

- adopting a system-level synchronous clock;
- initializing the bus arbitors at the beginning of a program execution; and
- starting the parallel execution at a certain DRAM reflesh cycle.

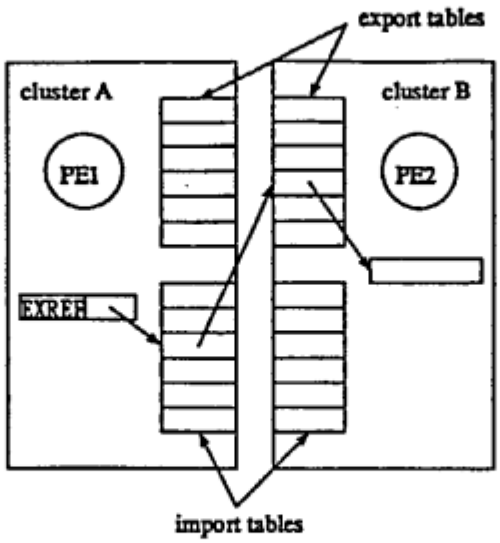

Figure 3: External Data Reference Mechanisms

4 The effects on the KL1 system software

The standard KL1 system software was designed for PIMs with a message passing network should be tuned for the cache hierarchy. This section reviews the standard software and then discusses how it should be tuned.

4.1 Overview of the standard KL1 system

4.1.1 External data reference mechanisms

The standard KL1 system handles internal pointers and external ones as different data types. They are given different tags, REF and EXREF. While an internal reference to a KL1 object requires only a load instruction, an external reference requires consulting import and export tables as illustrated in figure 3. Suppose PE_1 of $cluster_A$ tries to get the KL1 object pointed by an EXREF pointer. This read operation consists of the following steps.

- PE_1 looks up the proper entry of its import table and sends a read request to $cluster_B$.
- PE_2 of $cluster_B$ receives the request, looks up the proper entry of its export table, reads the required object, and sends back a reply message which holds the object.
- PE_1 receives the reply message.

[Ohnishi90] reported that a passive unification which includes one external read takes a Multi-PSI system 23 append logical instructions, while an internal one takes less than one.

4.1.2 Garbage collection

The standard KL1 system employs both a real-time GC and a stop-and-copy GC. For the real-time GC, the KL1 system employs a multiple reference bit (MRB) which identifies single and multiple references [Nishida88]. It is based on the observation that most objects are single referenced. The MRB mechanism also ensures a constant time update operation of a single referenced vector. Without this mechanism, the update operation would take time proportional to its size because KL1 avoids destructive assignment operations.

Since the real-time GC does not reclaim all the reusable area, a stop-and-copy GC is invoked when the entire heap space of a cluster has been exhausted. The external reference mechanism enables a cluster to do a stop-and-copy GC without suspending other clusters because all the external memory access requests are served by the cluster which is doing the GC. However, if a cluster-level GC fails to reclaim any area, clusters have to participate in a global GC which may reclaim some more area by reclaiming external references.

4.2 Tuned KL1 system software

4.2.1 Direct access mechanism and its dvantages

In order to exploit the low latency of the remote memory access, we decided to integrate the internal and external pointers. An external memory access is estimated to take PIM/k less than one append logical instruction in a typical case. Avoiding the external pointers does not lead to any difficulties, because every operation requiring remote memory access results in a state equivalent to the case of the standard KL1 system. The integration of the pointer types has three more advantages. One is that the tuned KL1 system is reduced half in code size, which, in turn, leads to higher hit ratio of the instruction cache. Another advantage is much easier debugging of the inter-cluster operations of the KL1 system software. The other is that various kinds of inter-cluster load distribution schemes get available, because the resource management tables of other clusters can be examined more flexibly and more efficiently than the standard KL1 system.

4.2.2 Garbage collection

In PIM/k, the real-time GC does not have to be amended. The MRB mechanism reclaims unreferenced area without regard to internal or external reference and the reclaimed area can be put into the free list of the cluster where it belongs.

The stop-and-copy GC, however, is affected. A cluster should not start a stop-and-copy GC without notifying the other clusters, because it might move objects which are still referred to from other clusters without adjusting the external pointers. Therefore all the clusters should participate in the GC, which is likely to cause an efficiency problem. The required time for a stop-and-copy GC may be thought proportional to the amount of live objects and independent of the GC interval. Thus frequent GCs would lose the overall efficiency of the system.

We have three possible solutions to reduce the GC frequency though we have implemented only the first one.

- Reserve some memory as a system-level free space and allocate it to the heap exhausted clusters.
- Suspend the goal reduction of the heap exhausted clusters. Since other clusters can continue the goal reduction, the overall efficiency would be acceptable if the number of suspended clusters remains small.
- Substitute the MRB-based real-time GC for a more powerful one like an ordinary reference count mechanism. It would reclaim almost all garbage and reduce the GC frequency.

The stop-and-copy GC creates import and export tables temporarily to avoid excessive inter-cluster memory access. Each reclaimed area is put into the free list of the cluster which the area belongs to.

5 Current status and preliminary evaluation

The PIM/k hardware is still under development and the KL1 system software has room to be improved. So far, the hardware has one cluster with four PEs which runs a 12 queen program coupled with the KL1 system software. The program execution consumes about 600 MB of the heap memory without any stop-and-copy GC. The performance with four PEs is about 3.5 times faster than the single PE's case.

Each PE board has counters and a trace memory. The former can count the number of events such as all the cache accesses and those that hit the first-level cache which calculate the first-level cache hit ratio. The latter can record the PE's memory access request and part of the first-level bus status on every clock. All the trace memories can also interrupt their PEs in every 32K clocks which suspend the parallel execution almost synchronously[1]. By reading the event counters during suspension and then resuming the parallel execution, we can observe the dynamic features of a parallel execution without affecting the execution much.

Figure 4 shows the first-cache hit ratio of the 12 queen program executed by one PE. In this figure, the hit ratio is averaged in every 0.8 second. At the beginning of the execution, the PE has 40K byte free heap space within the first-level cache, which gets exhausted in the first two seconds. Then, the hit ratio gets stationary around 94%.

Though the actual PIM/k implementation has cut some design alternatives, we are still interested in various types of shared memory model. For this purpose, we are also developing a clock level software simulator which models the behavior of PE's instruction fetch and decode, cache, memory, and bus arbitor. A KL1 program runs on the simulator as if it runs on the actual hardware though the execution speed is about 7,500 times slower.

[1]Interrupt is taken only at the end of an instruction which varies depending on the cache miss and the like.

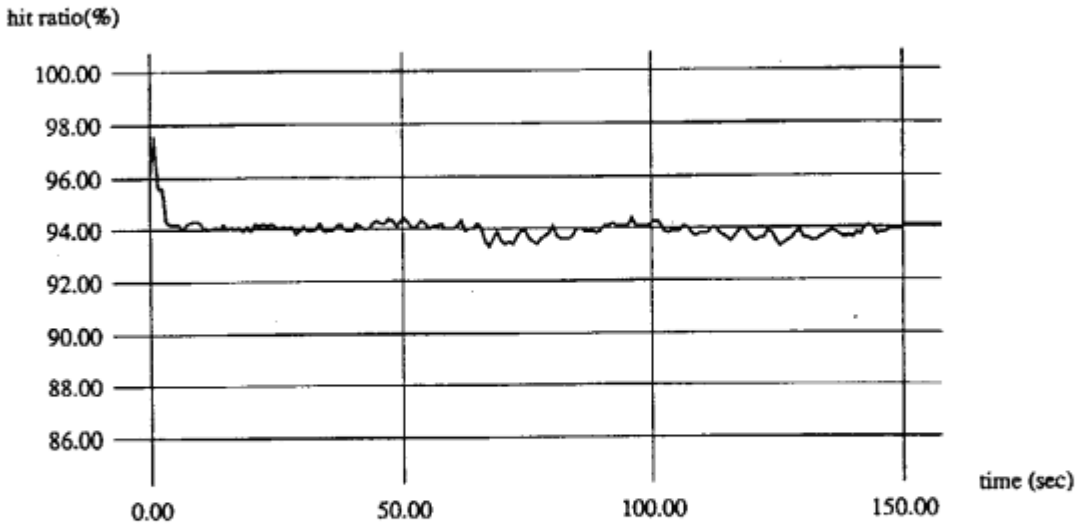

Figure 4: First-level cache hit ratio of PIM/k

It allows us to get the system's behavior in detail and to change the architectural configuration and/or cache protocol design easily. We are expecting that these approaches will bring us extensive experience about parallel processing in knowledge processing domain.

6 Conclusion

In this paper, we present the features of PIM/k with a cache hierarchy. It was motivated by a favorable relationship between the KL1 parallel execution and shared memory model. The design of our hierarchical cache protocol started at enhancing the MOESI model and has resulted in an extension of the Berkeley protocol. To keep the MLI property, a new replacement algorithm for the second-level cache was proposed. The cache hierarchy generally affects the KL1 system software in a favorable way. However the stop-and-copy GC requires a more sophisticated treatment which we should evaluate by experimentation. The actual machine has mechanisms which enables us to observe the dynamic features of a parallel execution, yet we are developing a clock-level software simulator to evaluate KL1 parallel execution on various types of shared memory architecture.

Acknowledgement

We are grateful to Dr. Taki and Dr. Hirata of ICOT who have been leading the PIM project. We also express thanks to the other members of PIM/k group and Dr. Trehan who gave us fruitful suggestions.

References

[Asano92] S.Asano, et al., 'The Unique Features of PIM/k: A Parallel Inference Machine with Hierarchical Cache System', Submitted to the International Parallel Processing Symposium.

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

[Eggers88] S.J.Eggers and R.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] A.Goto, et al., 'Overview of The Parallel Inference Machine Architecture (PIM)', Proceedings of the International Conference on Fifth Generation Computer Systems, Nov. 1988, pp. 208-229.

[Goto89] A.Goto, et al., '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] Ra.H.Katz, et al., 'Implementing A Cache Consistency Protocol', Proceedings of the 12th Annual International Symposium on Computer Architecture, June 1985, pp. 276-283.

[Lenoski91] D.Lenoski, et al., 'Overview and Status of the Stanford DASH Multiprocessor', Proceedings of the International Symposium on Shared Memory Multiprocessing, pp. 102-108, 1991.

[Nishida88] K.Nishida, et al., 'Evaluation of the Effect of Incremental Garbage Collection by MRB on FGHC Parallel Execution Performance', Technical Report 394, ICOT, 1988.

[Ohnishi90] S.Ohnishi, et al., 'Evaluation of the KL1 Language System on the Multi-PSI', In Workshop on Parallel Implementation of Languages for Symbolic Computation, University of Oregon, 1990.

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

[Sweazey86] P.Sweazey and A.J.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.

[Ueda87] K.Ueda, 'Guarded Horn Clauses', Concurrent Prolog: Collected Papers, E.Shapiro, Ed., MIT Press, 1987.

[Warren88] D.H.D.Warren, et al., 'Data Diffusion Machine - a Scalable Shared Virtual Memory Multiprocessor', Proceedings of International Conference on FGCS'88, 1988.

[Wilson87] A.W.Wilson, 'Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors', Proceedings of the 14th Annual International Symposium on Computer Architecture, June 1987, pp. 244-252.