Distributed Pool and its Implementation

Masaki SATO Masahiko YAMAUCHI Takashi CHIKAYAMA
masaki@icot.or.jp yamauchi@icot.or.jp chikayama@icot.or.jp
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

Abstract

PIMOS, an operating system for the parallel inference machine PIM, provides a utility, called *pool* , that manages tables of arbitrary KL1 data. This pool stores any KL1 data and enables it to be extracted and/or referenced when needed according to a search key. This pool is used with many application programs, reducing the cost of program development.

However, its performance is not satisfactory for highly parallel applications, because stored data is located in the memory of the node where the pool was initially created. When its data is accessed from another node, the ensuing data transfer increases network traffic, and simultaneous access causes centralization of the computing load. To overcome this problem, we have developed a distributed pool. The pool is distributed to many nodes and, if a process accesses data existing in another node, the pool caches data in its node. It provides the same functionality as the original pool by maintaining cache coherence. This paper describes the mechanism of this distributed pool.

1 Introduction

PIMOS [Chikayama 1988], the operating system of the parallel inference machine PIM [Goto 1989], provides a comfortable software environment for programming in concurrent logic programming language KL1 [Ueda 1990].

PIMOS provides a utility named *pool*. Pool provides basic in-memory database features, where arbitrary KL1 data are temporarily stored when a program is run, with the necessary data being extracted and referenced according to a search key. The pool feature has been extensively used by most experimental parallel application software systems running on PIM. It has been quite effective in reducing the programming effort required of application programmers.

However, for applications that process a large amounts of data, managing all the data as a unit, pool has performed poorly. Irrespective of the processing node accessing data, they are located in the memory of the node where the pool was initially created. They are then copied between nodes, thus increasing the amount of network traffic. As the PIM has a distributed memory architecture, such centralization of data results in longer latency for data access and a concentration of communication load on specific processing nodes.

There are two other problems. The first is in the nature of the distributed implementation [Rokusawa 1994] of KLIC [Chikayama 1994], for many computers connected through a network. In the current implementation, since interprocessor communication is handled by the operating system, communication cost are very high, making the situation more serious.

The second is that when a program using a large amount of data and centralized data management is executed, the problem size is restricted by the memory limitations of one node. This problem was encountered when running MGTP [Hasegawa 1992] where each node contains the same data.

Given this situation, we have developed a new implementation of the feature where data are distributed to many processing nodes.

Two methods of data distribution may be used.

1. Management of Distributed Data Using Hashing

   In this method, data is assigned a key such that the data is distributed according to the hash value of its search key. While the load concentration can be relieved, the communication frequency and quantity can not be decreased.

2. Management of Distributed Data Using Caches

   In this method, data is managed by keeping a copy of the data made wherever that data is to be used. The load concentration can be relieved and the communication frequency and quantity can also be minimized. However, delicate management of consistency between the copies is required.

We used the latter method with software caching for an efficient distributed pool. The contents of a pool are cached on each node to reduce the cost of access on each node. This distributes memory consumption to many processing nodes, making the maximum capacity of the

pool much larger. Section 2 gives an overview of the distributed pool. Section 3 describes the implementation in detail. Section 4 presents an evaluation of our cache protocol.

2 Overview of Distributed Pool

2.1 Cache Protocol

Our caching mechanism is similar to that of the coherent cache memory systems of shared-memory multiprocessor systems [Archibald 1986]. However, some subtle differences exist between the systems. The main features of our cache protocol are as follows.

1. Asynchronous Communication

   A shared-memory multiprocessor system uses a single serialization point as its shared bus, allowing memory transactions to be kept in order[1]. On the other hand, in general, communication between KL1 processes is asynchronous. The implementation of a shared bus as a process can keep messages in order, but it performs poorly from the viewpoint of throughput and prevents natural concurrency. Therefore, we define temporary states between sending a message and receiving a reply. We have to design a state transition scheme containing temporary states. In this point, our system is similar to the DASH [Lenoski 1990] directory-based cache coherence protocol.

2. Nonexistent Back Storage

   In a distributed pool, a cache is not used for small and fast local buffers that mediate fast processors from slow main memory as does hardware cache memory. It is used for caching data, which may be created in the memory of another node, on its own node according to demand for quicker data access, not through the slow network, on the second access. Therefore, the storage structure does not have back storage like main memory; instead a distributed cache on each node constitutes whole storage like in cache only memory architectures(COMA) [Hagersten 1991]. In the event of a cache miss, it is necessary for another cache to supply data instead of back storage. If some caches have the same data, the data in only one of these is set as an *owner* and a cache that has owned data responds to a requirement for data from another cache. When a user tries to store an amount of data exceeding user-defined cache capacity, data has to be removed from the cache to reclaim memory space. Data, except for owned data, has to be selected for replacement because at least one of the data with the same key must be left in some cache.

3. User Message Interfaces

   Besides messages enabling easy handling or storing data, we have a variety of ways to access data stored in the pool. For example, some messages replace data only when data with the same key exists. Others place new data regardless of whether or not data with the same key exists. This complicates our cache coherent mechanism further.

4. Cache Line

   In hardware cache memory systems, selection of the cache line size is one of the central design issues. Cache memories are accessed using data addresses as keys and data with adjacent addresses are likely to be accessed in a short period of time. Thus, we can create a prefetching effect by making the cache line size greater than one. On the other hand, data in pools are accessed by keys arbitrarily chosen by the user: integer, atom, string or structured data. In general, we cannot expect any access localities associated with similarities of keys. Thus, we made the cache line size to be one; that is, we abandoned the idea of grouping data with similar keys.

5. Write-Invalidate Type Cache Consistency

   There are two ways to keep the cache consistency: write-invalidate and write-update. Our coherent protocol introduces the write-invalidate type to decrease the amount of data sharing as in most coherent cache memory systems.

2.2 Components of Distributed Pool

Figure 1 shows two principal kinds of processes that constitute a distributed pool.

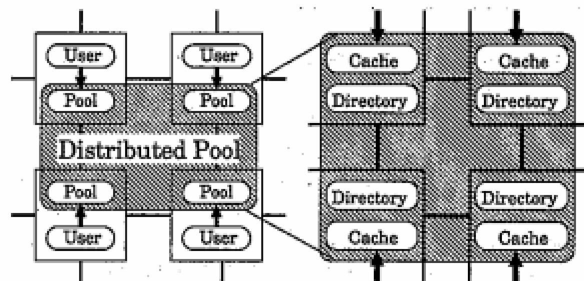

Figure 1: Process Constitution

[1] The weak consistency model used on split bus implementations is not acceptable for ease of programming in our case.

- Cache Processes

  Cache processes keep data in a hash table and manage their four permanent states and seven temporary states. The permanent states are classified according to exclusivity and the responsibility for data supply. The temporary states take care of crossed messages.

- Directory Processes

  Directory processes keep no data but manage cached data locations with lists of caching nodes. Such a list is called a *cache list* . The head element of a cache list shows the owner. The owner is responsible for providing data for cache misses, and is not selected for replacement. According to the cache list, directory processes transfer messages between caches. The directory processes have two permanent states classified according to the state of data validity, and six temporary states. To avoid access centralization, directory processes are located in each node unless otherwise designated. The managed set of data is decided according to search keys.

2.3 Information Located in a Cache

The currently available KL1 implementations employ lazy data transfer policies. The actual data is not moved only by moving pointers. Thus, we cannot be sure where the data exists. In fact, a cache may not keep real data, only a pointer. Nevertheless, managing the pointer locations can be effective because a user process that obtains a pointer from a pool usually reads this data that causes data migration.

3 Implementation

3.1 User Interfaces

Principal messages to pools for users are as follows.

carbon_copy(Key, O) : copies an element.
put(Key, X, Status) : adds an element.
get_if_any_and_put(Key, X, Y) : extracts an element, if any, and then stores a new element with the same key.
get_and_put_if_any(Key, X, Y, Y1) : extracts an element, if any, and then stores a new element with the same key. If there is no corresponding element, the new data is not stored in the pool either.
get_if_any(Key, X) : extracts an element, if any, with the given key.
get_all(Key, O) : extracts all the elements with the key.
remove(Key, Status) : deletes an element.

3.2 Cache States

Cached data can take one of the following states.

Invalid(I) : the cache does not contain this data.
Exclusive(E) : no other cache contains this data.
Shared-Owned(SO) : other caches possibly contain this data, but this cache is responsible for supplying the data and continues to keep it.
Shared-Unowned(SU) : at least one other cache contains this data and this cache does not take any responsibility for keeping it.
Waiting-Exclusion(WE) : the cache contains this data and is waiting for invalidation of the same data on all other caches, after which it enters the E state.
Waiting-Exclusion-If-any(WEI) : the cache contains this data and is waiting for invalidation of the same data on all other caches, after which it enters the E state. If the data is invalidated, because of crossed messsages, and no directory sends back the data, later it enters the I state.
Waiting-Shared-Data(WSD) : the cache does not contain this data and is waiting for the data to be transferred. If other caches contain this data, it will be shared after the data arrives.
Waiting-Purge(WP) : the cache contains this data and is waiting for invalidation of the same data on all other caches, after which it enters the I state.
Waiting-Exclusion-without-Data(WED) : the cache does not contain this data and is waiting for invalidation of the same data on all other caches, after which it enters the E state.
Waiting-Exclusion-If-any-without-Data(WEID) : the cache does not contain this data and is waiting for invalidation of the same data on all other caches, after which it enters the E state. If no directory sends back the data, later it enters the I state.
Waiting-Purge-without-Data(WPD) : the cache does not contain this data and is waiting for invalidation of the same data on all other caches, after which it enters the I state.

3.3 Directory States

The directory manages two permanent and six temporary data states.

Invalid(I) : no cache contains this data.
Valid(V) : some caches contain this data.
Waiting-Data(WD) : waiting for the data to be sent from a cache.
Waiting-Exclusive-Data(WED) : waiting for the data for exclusive access. Later it enters the V state.
Waiting-Exclusive-Data-If-any(WEDI) : waiting for the data for exclusive access. If no cache sends back the data, because of crossed messages, later it enters the

I state.
Waiting-Exclusion(WE) : waiting for exclusive access to the data. Later it enters the V state.
Waiting-Removed-Data(WRD) : waiting for the data and requires exclusive access to it. Later it enters the I state.
Waiting-Purge(WP) : waiting for exclusive access to the data. Later it enters the I state.

3.4 Network Transactions

Network transactions can be divided into five groups: requests from a cache to a directory, requests from a directory to a cache, replies from a cache to a directory, replies from a directory to a cache, and report from a cache to a directory. Requests from a cache to a directory can be further classified depending on needs of exclusive access. Network transactions are as follows.

1. Requests from a Cache to a Directory
   (1) Not Requiring Exclusive Access
   get_shared_data
   (2) Requiring Exclusive Access
   (a) Requiring Data
   get_exclusive_data : to go to the E state
   get_exclusive_data_if_any : to go to the E or I state
   get_removed_data : to go to the I state
   (b) Not Requiring Data
   i. Already Keeping it
   exclude : to go to the E state
   exclude_if_any : to go to the E or I state
   purge_others : to go to the I state
   ii. Not Needing it
   purge_and_exclude : to go to the E state
   purge_all : to go to the I state
2. Requests from a Directory to a Cache
   send_data
   send_data_and_invalidate
   invalidate
3. Replies from a Cache to a Directory
   data
   purged
4. Replies from a Directory to a Cache
   data_found
   no_data_found
   exclusion_made
5. Report from a Cache to a Directory
   purged

3.5 State Transition

Table 1 shows cache state transition by user messages, and table 2 shows cache state transition by requests and replies from directories.

3.6 Coherence Protocol

This section explains the coherence protocol.

3.6.1 Usual Behavior

1. Copy Requests

   When a cache receives a copy request (*carbon_copy* message) for data in an Exclusive state, a Shared-Owned state, or a Shared-Unowned state, the data is simply copied without a network transaction and the cache stays in the same state.

   In an Invalid state[2], as shown in figure 2, the cache sends a *get_shared_data* message to a directory and changes its state to Waiting-Shared-Data. We call the cache that sends a request message to a directory the *requesting cache* . The directory checks the cache list and issues a *send_data* message to the owning cache. The owning cache returns the pointer to the data with a *data* message, and changes its state to Shared-Unowned. The directory sends the pointer from the owning cache to the requesting cache with a *data_found* message, and registers the requesting cache in the cache list as the owning cache. The requesting cache returns the pointer to the user, and changes its state to Shared-Owned. If no other cache has the data then a *no_data_found* message is sent from the directory, which changes the cache state to Invalid.

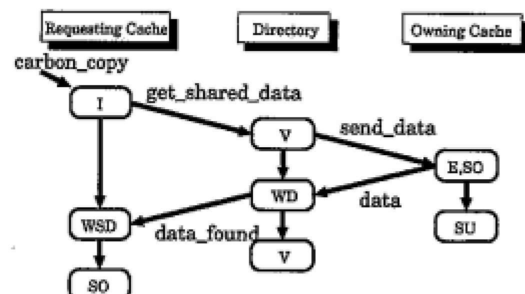

Figure 2: State Transition Diagram for Receiving a *carbon_copy* Message

[2] In fact, in the Invalid state, there is no data in the cache. This state is termed Invalid from a logical viewpoint.

Table 1: Cache State Transition by User Messages

Messages / States	carbon_copy	put	get_if_any_and_put	get_and_put_if_any	get_if_any get_all	remove
1.I	get_shared_data/WSD	purge_and_exclude/WED	get_exclusive_data/WED	get_exclusive_data_if_any/WEID	get_removed_data/WPD	purge_all/WPD
2.E	-/E	-/E	-/E	-/E	-/I	-/I
3.SO	-/SO	purge_and_exclude/WE	exclude/WE	exclude_if_any/WEI	purge_others/WP	purge_all/WP
4.SU	-/SU	purge_and_exclude/WE	exclude/WE	exclude_if_any/WEI	purge_others/WP	purge_all/WP
5.WE	->queue	->queue	->queue	->queue	->queue	->queue
6.WEI	->queue	->queue	->queue	->queue	->queue	->queue
7.WSD	->queue	->queue	->queue	->queue	->queue	->queue
8.WP	->queue	->queue	->queue	->queue	->queue	->queue
9.WED	->queue	->queue	->queue	->queue	->queue	->queue
10.WEID	->queue	->queue	->queue	->queue	->queue	->queue
11.WPD	->queue	->queue	->queue	->queue	->queue	->queue

notations: 1.request message to a directory/next state
2.->queue : this message is queueing

2. Copy-and-Update Requests

When a cache receives a copy-and-update request (*get_if_any_and_put* message) for data in an Exclusive state, it returns the old data to the user, stores the new data and stays in the same state without sending a message to a directory.

As figure 3 shows, in an Invalid state, the cache sends a *get_exclusive_data* message to a directory, and changes its state to Waiting-Exclusion-without-Data. The directory checks the cache list, sends a *send_data_and_invalidate* message to the owning cache and sends *invalidate* messages to the other caches in the list. The owning cache returns the pointer to the data with a *data* message, and changes its state to Invalid. The other caches send *purged* messages and change their state to Invalid. The directory, after receiving all the replies, sends a *data_found* message to the requesting cache, and updates the cache list to contain the requesting cache only. The requesting cache returns the pointer to the user, stores the new data and changes its state to Exclusive. As figure 4 shows, if no other cache has the data, then a *no_data_found* message is sent from the directory, and the cache stores the new data and changes its state to Exclusive.

As figure 5 shows, in a Shared-Owned state or a Shared-Unowned state, the cache sends an *exclude* message to a directory, and changes its state to Waiting-Exclude. The directory sends *invalidate* messages to caches that have a copy of the data. After receiving the replies from all the caches, it sends an *exclusion_made* message to the requesting cache. The requesting cache returns the pointer to the user, stores the new data and changes its state to Exclusive.

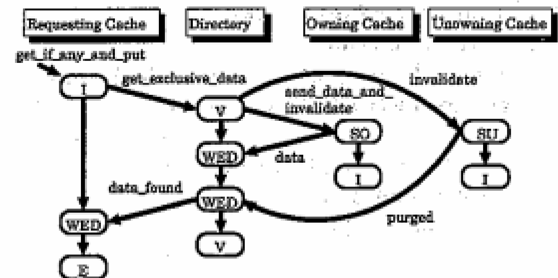

Figure 3: State Transition Diagram for Receiving a *get_if_any_and_put* message(1)

3. Extract Requests

When a cache receives an extract request (*get_if_any* message) for data in an Exclusive state, it sends a *purged* message to a directory, returns the extracted data and changes its state to Invalid.

As figure 6 shows, in an Invalid state, the cache sends a *get_removed_data* message to a directory,

Table 2: Cache State Transition by Requests and Replies from Directories

Messages States	data_found	exclusion_made	no_data_found	send_data	send_data_and_invalidate	invalidate
1.I	error	error	error	ignore	ignore	ignore
2.E	error	error	error	data/SU	data/I	purged/I
3.SO	error	error	error	data/SU	data/I	purged/I
4.SU	error	error	error	error	error	purged/I
5.WE	-/E	-/E	-/E	data/WE	data/WE	purged/WE
6.WEI	-/E	-/E	-/I	data/WE	data/WE	purged/WE
7.WSD	-/SO	error	-/I	ignore	ignore	ignore
8.WP	-/I	-/I	-/I	data/WP	data/WP	data/WP
9.WED	-/E	-/E	-/E	ignore	ignore	ignore
10.WEID	-/E	-/E	-/I	ignore	ignore	ignore
11.WPD	-/I	-/I	-/I	ignore	ignore	ignore

notation: reply message to a directory/next state

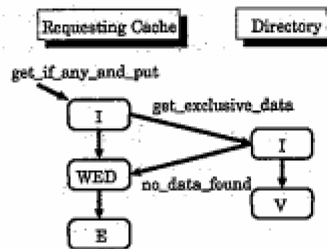

Figure 4: State Transition Diagram for Receiving a *get_if_any_and_put* Message(2)

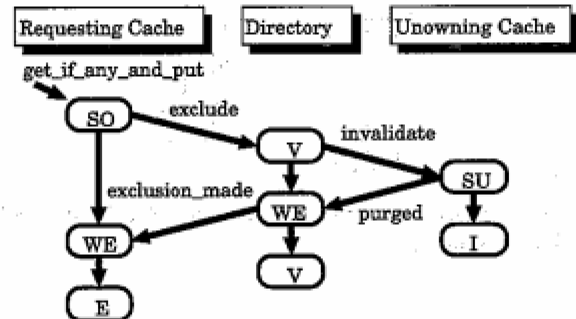

Figure 5: State Transition Diagram for Receiving a *get_if_any_and_put* Message(3)

and changes its state to Waiting-Purge-without-Data. The directory checks the cache list, sends a *send_data_and_invalidate* message to the owning cache and sends *invalidate* messages to the other caches. The owning cache returns the pointer to the data with a *data* message, and changes its state to Invalid. The other caches return *purged* messages, and change their states to Invalid. After receiving all the replies from the caches, the directory sends a *data_found* message to the requesting cache. The requesting cache returns the pointer to the user, and changes its state to Invalid. As figure 7 shows, if no other cache has the data, then the directory sends a *no_data_found* message to the requesting cache which in turn returns nil data to the user and changes its state to Invalid.

As figure 8 shows, in a Shared-Owned state or a Shared-Unowned state, the cache sends a *purge_others* message to a directory, and changes its state to Waiting-Purge. The directory sends *invalidate* messages to all of the caches that have a copy of the data except the requesting cache. After receiving the replies from all the caches, it sends an *exclusion_made* message to the requesting cache. The requesting cache returns the pointer to the user, and changes its state to Invalid.

3.6.2 Crossed Message Behavior

(1) Usually a cache in an Invalid state doesn't receive requests from a directory, because a directory only sends request messages to caches that are registered in the cache list. However, when messages get crossed, a cache in an Invalid state receives a request message from a directory. This happens because a *purged* message, that is, a report message from a cache, is issued independent of directory behavior. In this case, the cache which issued the report message changes its state to Invalid without confirming that the cache list in the directory has been updated. When a cache in an Invalid state receives a request message from a directory, it doesn't need to send a reply message to the directory because it already sent a purged message to the directory. The directory recognizes the report message from the cache as the reply message to the request mes-

Figure 6: State Transition Diagram for Receiving a *get_if_any* Message(1)

Figure 7: State Transition Diagram for Receiving a *get_if_any* Message(2)

sage to the cache. Figure 9 shows an example of this case. Cache B in the Invalid state receives a *send_data_and_invalidate* message from the directory.

(2) Usually a directory in an I state doesn't receive a request message for invalidation to all other caches keeping the data, like *exclude* , *exclude_if_any* , and *purge_others* messages. However, when messages get crossed, a directory in an I state receives such messages. Figure 10 illustrates this case. An *exclude*

Figure 8: State Transition Diagram for Receiving a *get_if_any* Message(3)

Figure 9: State Transition Diagram for Crossed Messages(1)

message which is sent by Cache A receiving a *get_if_any_and_put* message in an SU state, and a *purge_others* message which is sent by Cache B receiving a *get_if_any* message in an SO state, are sent to a directory synchronously. If the *purge_others* message is accepted first, the directory sends an *invalidate* message to the cache A in a Waiting-Exclusion state, and after receiving a *purged* message from the cache A, it changes its state to I. In such a case, the next *exclude* message is accepted by the directory which is in the I state. The directory concludes that the cache A has no data, and it performs as if it had received a *get_exclusive_data* message. As a result, it sends a *no_data_found* message to the cache A.

Figure 10: State Transition Diagram for Crossed Messages(2)

(3) Usually a cache in a Waiting-Shared-Data state, a Waiting-Exclusion-without-Data state, a Waiting-Exclusion-If-any-without-Data state, or a Waiting-Purge-without-Data state doesn't receive any request message from a directory. This is because their states have changed from Invalid and the directory recognizes that a cache in an Invalid state has no data. However, this can happen.

When a cache receives a *get_if_any* message or other messages from a user, it sends a *purged* message and changes its state to Invalid. After that, it may receive a message from a user that changes its state to a Waiting-Shared-Data state, a Waiting-Exclusion-without-Data state, a Waiting-Exclusion-If-any-without-Data state, or a Waiting-Purge-without-Data state and the directory accepts a request message from another cache, for example, a *get_shared_data message* , a *get_exclusive_data* message, or an *exclude* message, in advance of the previous *purged* message. In this state, the cache receives a request message from the directory, such as a *send_data* message, a *send_data_and_invalid* message or an *invalidate* message. In this state the cache must not respond to it. The directory recognizes the *purged* message as a reply message to the request message it sent to the cache. Figure 11 shows an example of this case. Cache A in the Waiting-Exclusion-without-Data state receives an *invalidate* message from the directory.

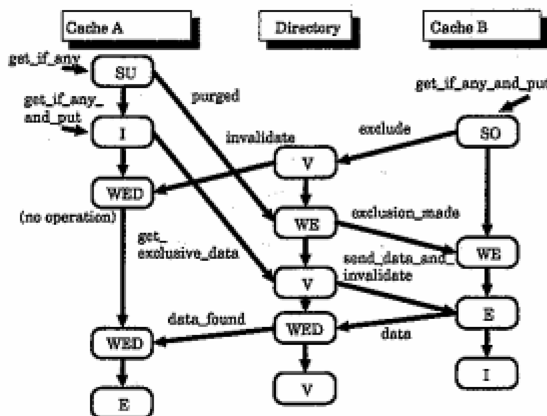

Figure 11: State Transition Diagram for Crossed Messages(3)

3.7 Replacement

When a cache is full and new data cannot be stored, some data must be replaced. The data which is replaced is kept in another cache as copied, not owned data, that is in a Shared-Unowned state. We don't use the LRU algorithm for managing data in a Shared-Unowned state because of the high management costs. We manage data in a Shared-Unowned state with the FIFO algorithm. Replacing data may or may not reported to a directory. If it is reported to a directory, one interprocessor communication has to be made. If it is not reported, if the data is modified by data from another cache, an *invalidate* message from the directory to the cache and a reply message from the cache to the directory have to be sent because the cache list hasn't been updated. When the cache requires the data again, the directory has to make sure not to register the cache twice in the cache list. Which is better depends on replacement frequency, data sharing ratio, and update frequency in the application program. We chose the former for simplicity.

3.8 Management of Ownership

Whether it is better or not to transfer ownership on data copying depends on the application. As the owned data is not selected for replacement, it is desirable to distribute ownership among caches to distribute memory usage. For example, in a single-writer multiple-reader type program, it is better to change ownership when the data is copied from the writer side to the reader side because a lot of nodes on which the reader processes are allocated can have owned data. On the other hand, in a multiple-writer single-reader type program, it is better not to change ownership because a lot of nodes on which the writer processes are allocated can have owned data. Further analysis of the problem is necessary. The protocol described in this paper transfer ownership along with data.

4 Evaluation

The purpose of this evaluation is to investigate the effectiveness of the distributed pool in comparison with the conventional centralized pool. Our evaluation is on the following points.

- Reduction of interprocessor communication by caching.
- Distribution of computing load.

4.1 Reduction of Interprocessor Communication by Caching

To see whether caching efficiently reduces interprocessor communication, we measured the execution time (response time) of carbon_copy messages from a user using a centralized pool and a distributed pool. For the centralized pool, times of local access in which the pool process and the user process are located on same node, and that of remote access in which they are located on different nodes were measured. The hit ratio is used as the parameter for the distributed pool. Since the complexity of managed data changes the traffic between nodes, we measured with integer data and list data containing ten integer elements. We measured them on PIM/m and the distributed version of KLIC. The distributed KLIC system was run on a Sparc Center 2000 using PVM. The Sparc Center 2000 is a shared memory multiprocessor

Table 3: Comparison of the Number of Messages Transferred

Transfer mode	The number of messages transferred	
	Remote access to the centralized pool	The distributed pool with no cache hits
lazy	7060	18784
eager	4060	10753

machine, and the PVM is a message-passing library. In the distributed KLIC system, there are two modes for transferring structured data, *lazy transfer mode* and *eager transfer mode* . Lazy transfer mode sends only one level of the structured data. Eager transfer mode sends the whole data structure based on pointers to the structure. We measured with both modes.

Figure 12 shows the execution time for copying integer data on PIM/m, figure 13 shows the result for list data on PIM/m, and Figure 14 shows the result for integer data on the distributed KLIC system. The x axis indicates the hit ratio of the distributed pool and the y axis indicates the execution time for copying data per one access.

Figure 12 shows that, with perfect cache hits, the access for the distributed pool is slower than that for local access to the centralized pool, but that it is much faster than that for remote access. The access for the distributed pool with no cache hits is slower than that for both accesses of the centralized pool. As a result, when the hit ratio is about 85%, the performance of the distributed pool and remote access to the centralized pool are equal.

Figure 13 shows that the performance of the distributed pool is equal to that of remote access to the centralized pool at a 40% hit ratio because the communication costs increase for complex data.

Figure 14 shows that the execution time on the distributed KLIC system is about two orders of magnitude slower than on PIM/m. Since the distributed KLIC system has slower communication, the hit ratio, where the performance of the distributed pool is equal to that of remote access to the centralized pool, is lower than that on PIM/m. In the lazy transfer mode, the hit ratio is about 35%. In the eager transfer mode, the hit ratio is about 50%. The eager transfer mode shows a better performance than the lazy transfer mode, because the number of messages transferred is less than with the lazy transfer mode. Table 3 shows the number of messages transferred in copying 1000 integers of data, for remote access to the centralized pool, and for the distributed pool with no cache hits.

These measurements show that the distributed pool is effective for managing complex data when access locality is high. The measurements also indicate that the distributed pool is more effective for a system with slower communication, as in the distributed KLIC system.

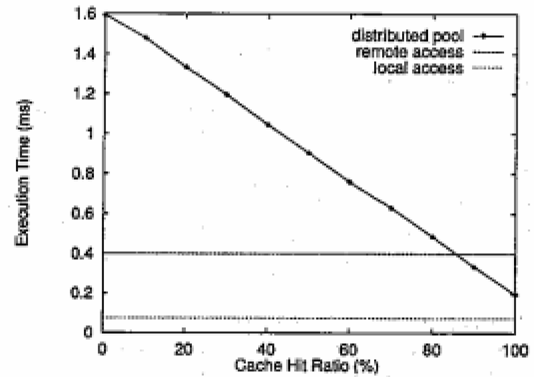

Figure 12: Execution Time for Copying Integer Data on PIM/m

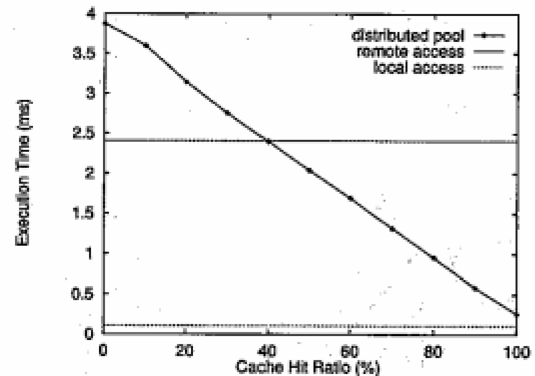

Figure 13: Execution Time for Copying List Data on PIM/m

4.2 Distribution of Computing Load

Figure 15 shows how computing load is distributed using the distributed pool. The x axis gives the number of nodes in which the user processes are located and the y axis gives the execution time for copying 1000 integers of data. The distributed pool is given a hit ratio of 85%. The programs was run on PIM/m. The execution time for the centralized pool is almost proportional to the number of the node, and with 15 nodes, it is about 10 times longer than with 1 node. The execution time for the distributed pool doesn't increase as rapidly as for the centralized pool, and the delay only doubles with 15 nodes. The reason for the increase is the increased load for supplying data to cache misses.

This measurement shows that the distributed pool can distribute the load for accesses to a single pool to many

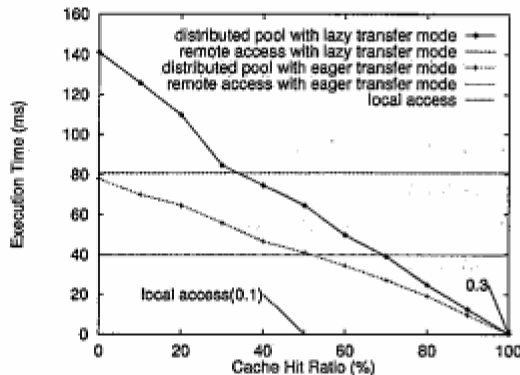

Figure 14: Execution Time for Copying Integer Data on the Distributed KLIC System

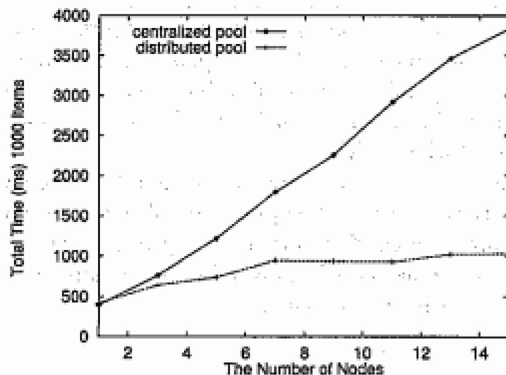

Figure 15: Execution Time for Concentrating Access on PIM/m

computing nodes.

5 Conclusions

We have introduced the distributed pool, which distributes data efficiently among many processing nodes using software caching. Using the distributed pool, application programmers can distribute the computing load flexibly without worrying about data consistency. It also uses memory more efficiently for applications processing a large amounts of data.

We intend to further evaluate the distributed pool as applied to practical applications processing large amounts of data.

Acknowledgments

We would like to thank Mr. Kazuaki Rokusawa at ICOT for his helpful comments. We would also like to thank Dr. Shunichi Uchida, Director of the ICOT Research Center, for his encouragement and support in this work.

References

[Archibald 1986] J. Archibald and J. L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, Vol. 4, No. 4 (1986), pp. 273-298.

[Chikayama 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp. 230-251.

[Chikayama 1994] T. Chikayama, T. Fujise and D. Sekita. A Portable and Efficient Implementation of KL1, In *Proc. 6th Int. Symp. PLILP'94*, Berlin, 1994, pp. 25-39.

[Goto 1989] A. Goto. Research and Development of the Parallel Inference Machine in FGCS Project. In M. Reeve and S. E. Zenith (Eds.), *Parallel Processing and Artificial Intelligence*, Wiley, Chichester, 1989, pp. 65-96.

[Hagersten 1991] E. Hagersten, A. Landin and S. Haridi. DDM - A Cache-only Memory Architecture. SICS Research Report R91:19, 1991.

[Hasegawa 1992] R. Hasegawa and M. Fujita. Parallel Theorem Provers and Their Applications. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1992, pp. 132-154.

[Lenoski 1990] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In sl Proc. 17th Annual Int. Symp. on Computer Architecture, IEEE, New York, 1990, pp. 49-58.

[Rokusawa 1994]
K. Rokusawa, A. Nakase, and T. Chikayama, Distributed Memory Implementation of KLIC. In sl Proc. Workshop on Parallel Logic Programming and its Programming Environments, Technical Report CIS-TR-94-04, University of Oregon, 1994, pp. 151-162.

[Ueda 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494-500.