EVALUATION OF MEMORY SYSTEM FOR INTEGRATED PROLOG PROCESSOR IPP

M. Morioka S. Yamaguchi T. Bandoh

Hitachi Research Laboratory, Hitachi, Ltd.

4026 Kuji, Hitachi, Japan

ABSTRACT

This paper discusses an optimal memory system to realize a high performance integrated Prolog processor, the IPP. First, the memory access characteristics of Prolog are analyzed by a simulator, which simulates the execution of a Prolog program at a micro instruction level. The main findings from this analysis are that: the write access ratio of Prolog is larger than that of procedural languages; and performance improvement requires the memory system to process concentrated, large write accesses effectively.

Then the Prolog acceleration strategies for conventional cache memories are discussed. Comparison is made of cache memories (store-swap, store-through) and a stack buffer, regarding not only performance but also reliability, complexity and effects on procedural languages. The advanced store-through cache with a multi-stage write buffer and an interleaved main memory are seen to have the same performance level as the store-swap cache. When considering data reliability, the advanced store-through cache is judged more suitable for the IPP than the store-swap cache. In a comparison between stack buffer and advanced store-through cache, the stack buffer is found to achieve higher peak performance, but this is affected by the program features. On the other hand, the advanced store-through cache constantly gets high performance for Prolog and procedural languages. As a result, it is concluded that the advanced store-through cache is best suited to the IPP.

1. INTRODUCTION

AI systems using Prolog and Lisp are expected to support human intelligent activities. Since Prolog is a declarative language, it is easier to program and maintain than procedural languages like C and Fortran. But to develop actual AI systems (e.g. expert systems using Prolog), it is sometimes necessary to link Prolog programs with procedural languages. Thus, in those systems, high speed execution of both Prolog and procedural languages is required. In order to meet these requirements, we developed an integrated Prolog processor (IPP) based on Warren's instruction set 1), which integrates Prolog acceleration hardware into a general purpose minicomputer 2) 3) 4).

The most characteristic mechanisms in executing Prolog programs are unification and backtracking. Especially for the latter, it is necessary to save and restore processor environments into/from a stack frequently to allow retrying of alternative clauses. Thus memory access behavior of Prolog programs differs from that of procedural languages. Therefore, for efficient execution of Prolog the memory system should guarantee large memory throughput. To do this, existing dedicated Prolog machines have adopted a hardware stack 5) 6) 7). But such an approach is often heavily affected by the locality of memory accesses. And also a dedicated Prolog hardware stack is not effective for speeding up execution of procedural languages.

The contribution of this paper is to clarify the most effective structure of the memory system to achieve high performance for both Prolog and procedural languages. This is done through the investigation of Prolog acceleration strategies for a conventional cache memory and the comparison between the hardware stack and the advanced cache memory, regarding not only performance but also reliability, complexity and effects on procedural languages.

In performance comparison of memory system organizations, the hit ratio and memory traffic of the cache memory and various hardware stack architectures suited to Prolog have been evaluated in 8) 9). However the average memory access time was not included, although it is as important as the hit ratio of the local memory. Therefore, in this paper, we compare the performance of each memory system in the average memory access time. The factor which determines the average access time consists of the hit ratio of the local memory and the recovery time when a miss hit occurs. As for the hardware stack, the overhead of a copy-back operation, which is necessary when overflow of the hardware stack occurs, must be taken into account. In particular, for the store-through cache with a write buffer and an interleaved main memory, the average memory access time is seriously affected by overflow of the write buffer and write processing ability of the interleaved main memory.

Section 2 describes the IPP design approach and topics related to the memory system. Then Section 3 describes a quantitative analysis of the memory access behavior of Prolog. Considering these results, Section

4 compares the cache memory and the hardware stack architecture.

2. PROLOG ACCELERATION STRATEGIES IN THE IPP

2.1 Design Targets

When developing actual expert systems, it is sometimes necessary to link Prolog with procedural languages. In these systems, numerical calculations and the tasks which have a fixed algorithm are programmed effectively by procedural languages. On the other hand, for diagnosis or inference according to the obtained results, Prolog is more suitable. Therefore, both Prolog and procedural languages should be run fast with less linkage overhead.

Dedicated Prolog machines, which have tagged architecture, cannot satisfy the requirement, because in general, they have poor ability in processing procedural languages (numerical calculations). While it seems more attractive to implement Prolog processing functions on a general purpose computer, the performance is reduced remarkably for the following two reasons.

(1) Tag manipulation

In Prolog, a tag is indispensable to recognize data types, such as variables, constants, and structures. In the case of Prolog implementation on a general purpose computer, the tag is often packed into a 32-bit datum, because of compatibility with standard I/O devices. This causes frequent tag manipulations such as separating the tag from the datum, evaluating the tag and packing it in the datum. According to our analysis, tag manipulations need 5-18 microprogram steps and as a whole they occupy about half the execution time of the Append program 3).

(2) Memory throughput

Memory accesses occurring in Prolog are mainly for stack manipulation, due to its stack oriented computations. In particular, saving and restoring of environments for backtracking occur frequently, because of the non-deterministic characteristics of Prolog, and that causes a heavy load on the memory system. For example, memory throughput of over 200 MB/s is required to get 1MLIPS 8).

Therefore, we designed the IPP with a focus on acceleration of tag processing and improvement of memory throughput. In the following sections, we discuss the memory system strategies of the IPP.

2.2 The Memory System for Integrated Architecture

In Warren's architecture, three stacks are defined in the memory areas, that is, the local stack, heap stack and trail stack. The local stack contains a frame needed for backtracking called CHOICE-POINT, and a frame for goal calling denoted as ENVIRONMENT. The heap stack contains structures, which are created while executing Prolog programs. The trail stack contains a history of bound variables which must be unbound when backtracking occurs. The local stack and the heap stack accesses are comparatively high among these three stacks. Especially for a non-deterministic program, saving and restoring of CHOICE-POINT (whose size is about 11 words 8)) to/from the local stack impose a heavy load on the memory system. Accordingly it is necessary for performance improvement to support backtracking efficiently.

To realize this, dedicated Prolog machines provide hardware stack architecture such as a specialized stack buffer, or a large size register file. Though these hardware stacks are certainly effective when the access converges to the top of the stacks, they show poor performance for scattered accesses, such as accesses to plural stacks or to the deep stack area, which are characteristics to Prolog. A conventional cache memory is more flexible than a hardware stack, but much time is needed to access it due to the tag comparison. However if it is possible to achieve high performance in saving and restoring CHOICE-POINT, it seems attractive for the IPP to adopt the conventional cache memory, because it can get high performance for both Prolog and procedural languages. Thus the two major points for designing the IPP memory system are:

(1) To clarify the memory system strategies leading to a good processing ability for saving and restoring CHOICE-POINT within the conventional cache memory.

(2) To clarify the most suitable memory system organization for the IPP.

The former is done through a detailed analysis of the Prolog memory access behavior, and the latter is done through a comparative study of various memory systems, such as a cache memory and a hardware stack.

3. PROLOG MEMORY ACCESS CHARACTERISTICS

3.1 The Local and Heap Stacks

This section discusses the memory access characteristics for the local stack and the heap stack, which have higher access frequencies than the trail stack.

(1) Local Stack

CHOICE-POINT and ENVIRONMENT are stacked on the local stack. CHOICE-POINT is created at non-determinate goal calling, and stack control registers and argument registers are saved in it. Almost all accesses are limited to the top portion of the local stack for saving, restoring, and modification of CHOICE-POINT.

ENVIRONMENT is allocated on the local stack, if a selected clause has more than one goal. Access to ENVIRONMENT occurs when head parameters are unified with permanent variables on argument registers, and permanent variables in a goal are loaded for goal calling. Thus, access is not limited to the top portion of the stack, but the deep portion of the stack can be accessed as well. Fig.1 shows an example of local stack access. This result was obtained by executing the Qsort program on the simulator described later. In this case, Prolog creates CHOICE-POINT for backtracking, and if unification fails, a backtrack operation occurs and CHOICE-POINT is restored into the registers. According to this access behavior, the local stack accesses occur at shorter intervals especially for a non-deterministic Prolog program, and the access addresses have a continuous feature.

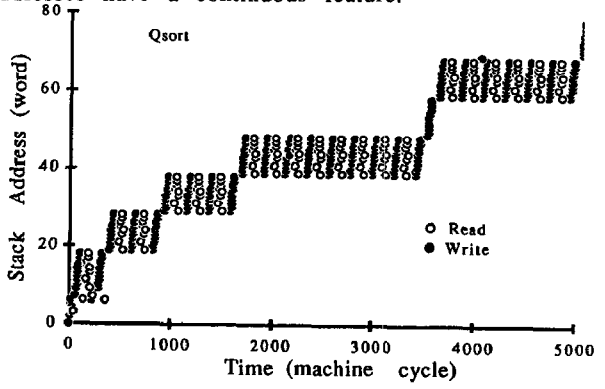

Fig. 1 Example of Local Stack Access

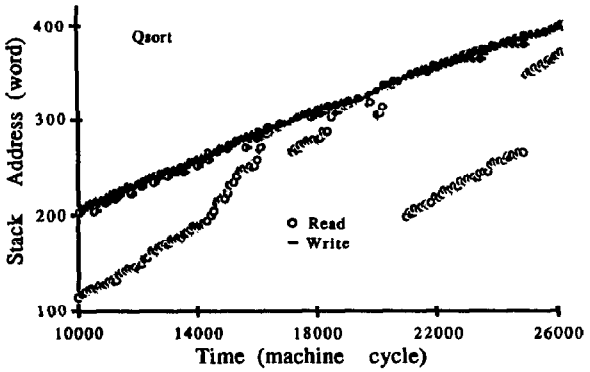

Fig. 2 Example of Heap Stack Access

(2) Heap Stack

The heap stack accesses are divided into two cases. One is stacking structures produced in program execution on the top of the heap stack. The other case is reference and binding to produced structures, which are usually in a deep portion of the heap stack. An example of heap stack accesses for the Qsort program is shown in Fig.2. From this figure, it is clear that almost all write accesses converge to the top of the heap stack, on the other hand read accesses are scattered throughout the stack. These features decrease the locality of the heap stack accesses.

3.2 Simulator Analysis

In order to clarify memory access characteristics, we developed a simulator. which simulates execution of a Prolog program at a micro instruction level. The simulator provides an execution sequence of Prolog instructions and a trace of the executed micro steps, including initiation for read and write requests and their addresses. The simulator is written in the C language, and executes object codes which are generated by our optimizing Prolog compiler 2). The summary of the benchmark programs used in the analysis are shown in Table 1. Details of the first six programs for the Prolog benchmarks are given in 12). Expert is a part of an expert system program written in Prolog. The four programs for C and Fortran are well known benchmarks for procedural languages.

Table 1 Summaries of Benchmark Programs

Language	Program	Source step		Dynamic	Data access
		Rule	Fact	instruction step	frequency
Prolog	Nreverse	2	3	3,657	3,229
	Conslist	1	1	3,250	4,783
	Solve	5	5	7,724	10,288
	Qsort	3	3	4,545	7,820
	Queen	10	2	21,483	20,834
	Shallow	14	1	48,010	99,019
	Expert	36	29	291,263	203,928
C	Dhrystone	345		241,086	173,044
	Prime	26		1,064,871	687,975
Fortran	Whetstone	168		690,232	685,641
	Linpack	562		227,979	179,781

(1) Access ratios for read and write

Table 2 shows the memory access ratios for data read, and data write obtained from simulated executions of Prolog, C and Fortran programs. Instruction fetch is not included, because the local memory for instruction fetch is commonly separated from that for data fetch in recent memory systems. The result shows that the ratios of memory write accesses of Prolog are larger than those of procedural programs, especially for Fortran. On average for the Prolog programs, the write access occupies about 50% of all data accesses.

Table 2 Comparison of Memory Access Ratios

Language	Program	Data read (%)	Data write(%)
Prolog	Nreverse	49.9	50.1
	Conslist	29.3	70.7
	Solve	57.5	42.5
	Qsort	41.2	58.8
	Queen	57.6	42.4
	Shallow	69.7	30.3
	Expert	55.6	44.4
	AVERAGE	51.5	48.5
C	Dhrystone	54.4	45.6
	Prime	55.9	44.0
Fortran	Whetstone	70.1	29.9
	Linpack	70.6	29.4
	AVERAGE	62.8	37.2

(2) Memory access ratios of each stack

Table 3 shows the memory access ratios of the three stacks and other areas, which contain accesses to clause indexing tables and a built-in command area. On average, about 80% of all data accesses are for the local and heap stack. In Nreverse, Conslist and Solve (which form Group 1), accesses to the heap stack occupy about 60% of all data accesses. But for Qsort, Queen, Shallow and Expert (which form Group 2), the local stack accesses occupy about 60% on average, because of frequent allocations of CHOICE-POINT and ENVIRONMENT.

(3) Locality of stack accesses

Table 4 shows the locality of stack accesses. Locality is defined as the hit ratio of the full-associative cache memory, which has a 1-word block size and uses the LRU replacement algorithm. Capacity of the cache memory (window size) is changed from 4 words to 256 words. For a large window size, local stack accesses show higher locality than those of the heap stack, because local stack accesses mainly converge to the top of the stack in contrast with the scattered behavior of heap stack accesses. Heap stack accesses have high locality for a small window size, because read-modify-write accesses occur frequently for the heap stack.

(4) Distribution of write access occurrences

The distribution of write access occurrences for the Prolog programs is shown in Table 5. The first 4 columns represent ratios of write accesses, which occur in the inter-arrival time of 4, 6, 8 and 16 machine cycles, respectively. The last column is the mean inter-arrival time of write accesses. About 55% of the write accesses occur under the inter-arrival time of 6 machine cycles, although the average is 17.6 machine cycles. These features of Prolog are caused by the CHOICE-POINT accesses.

Table 3 Memory Access Ratios of Each Stack

Program		Local (%)		Heap(%)		Trail(%)		Others(%)	
		read	write	read	write	read	write	read	write
Gr.1	Nreverse	4.8	5.0	43.2	45.0	-	0.1	1.9	-
	Conslist	6.7	7.1	21.7	62.7	-	0.9	0.9	-
	Solve	13.2	13.9	31.3	27.9	0.3	0.6	12.8	-
	AVERAGE	8.2	8.7	32.1	45.2	0.1	0.5	5.2	-
Gr.2	Qsort	23.1	36.7	14.8	17.3	1.6	4.7	1.8	-
	Queen	22.1	27.4	25.2	10.7	4.2	4.3	6.1	-
	Shallow	66.7	29.3	1.0	1.0	-	-	2.0	-
	Expert	28.0	30.5	1.0	1.0	-	-	26.6	12.9
	AVERAGE	35.0	31.0	10.4	7.5	1.5	2.3	9.1	3.2

Table 4 Locality of Stack Accesses

Stack	Program	Locality of Stack Access			
		W *<4word	W<16word	W<64word	W<256word
Local	Group1	13.0%	51.2%	61.4%	72.1%
	Group2	11.8%	71.6%	86.3%	95.0%
	AVERAGE	12.3%	62.8%	75.6%	85.2%
Heap	Group1	38.5%	39.4%	53.7%	61.6%
	Group2	36.4%	45.9%	67.7%	67.8%
	AVERAGE	37.3%	43.1%	61.7%	65.1%

* W : Window Size

Table 5 Distribution of Write Access Occurrences

Program	Distribution of Inter-arrival Time				mean (mc)
	T *<4 mc	T<6 mc*	T<8 mc	T<16 mc	
Nreverse	2.3%	61.8%	63.6%	65.6%	15.6
Conslist	6.3%	39.5%	67.9%	97.5%	8.7
Solve	10.0%	62.9%	63.4%	65.1%	17.6
Qsort	25.7%	68.0%	71.2%	82.8%	11.7
Queen	22.2%	55.4%	67.9%	74.4%	23.3
Shallow	36.7%	46.7%	50.0%	56.7%	26.5
Expert	24.4%	52.2%	56.7%	71.2%	19.8
AVERAGE	18.2%	55.2%	63.0%	73.3%	17.6

* T : Inter-Arrival Time of Write Access
* mc: machine cycle

In summary :

(1) In Prolog, the ratio of memory write accesses occupy 48.5% of all data accesses, and it is larger than the 37.2% value of the procedural languages.
(2) In the case of a non-deterministic Prolog program, write access occurrences have shorter inter-arrival time, and their addresses have a continuous feature.
(3) Local stack accesses have high locality because of the frequent saving and restoring of frames. On the other hand, heap stack accesses have less locality because of their scattered feature.
(4) There are two types of Prolog programs. One has dominant accesses to the heap stack, while the other to the local stack.

Therefore in order to improve the performance of Prolog, it is necessary for the memory system to process large, concentrated write accesses efficiently, and to process multi-stack accesses which are scattered to the local stack and heap stack.

4. MEMORY SYSTEM EVALUATION

Due to multi-stack based computations, stack management is the main job in executing a Prolog program. Therefore, a technique for speeding up the stack accesses by using a high speed local memory has a good effect on performance improvement of Prolog. There are two ways to get speed up. One is a conventional cache memory, either a store-swap or store-through strategy, applied to general purpose computers, and the other is a hardware stack mainly applied to dedicated Prolog processors. In this Section, we make a comparative study of these local memory strategies based on the results obtained in Section 3.

4.1 Store-swap Cache vs. Store-through Cache

To compare a store-swap cache and a store-through cache, we considered two issues to be examined for the IPP, that is reliability and memory throughput for Prolog.

4.1.1 Reliability

Data reliability is very important for the high speed local memory, because the local memory is an extension of the main memory. In a store-swap cache, some valid data in it may not exist in the main memory, thus some way is needed to realize high reliability, such as an error correcting code. But this has a bad influence on the critical-path delay. On the other hand, copies of the valid data in the store-through cache always exist in the main memory, thus it has a higher reliability.

4.1.2 Memory Throughput for Prolog

It is said that a store-swap cache works better for Prolog than a store-through cache, because of the high ratio of write accesses. Thus it is implemented in commercial Prolog machines. But from analysis of the memory access characteristics of Prolog, we feel that a store-through cache can perform the memory write accesses of Prolog effectively, if a multi-stage write buffer and an interleaved main memory are provided. That is, the concentrated write accesses caused by saving CHOICE-POINT's are absorbed by the multi-stage write buffer. Furthermore the data stored in the multi-stage write buffer are effectively transferred to the interleaved main memory, because these data have contiguous write addresses. By using these techniques, an advanced store-through cache can provide high memory throughput for Prolog. But in this organization, if the modified data, which are not on the cache memory, are referenced immediately, the read access time becomes long, because the reference must wait until the modification to the main memory is completed. Table 6 shows the accumulated distribution of modified-data references, as an average of all benchmarks. For a 20-machine cycle interval time, which is almost equal to the main memory access time of the IPP, the accumulated distribution is under 11.4%. In addition to this, since most referenced data are on the cache memory, immediate reference to the modified data does not affect the performance of the store-through cache very much.

Table 6 Accumulated Distribution of Modified-Data References

Program	Distribution of Interval Time				
	T *<10mc*	T<15 mc	T<20 mc	T<25 mc	T<30 mc
Average of all	10.5 %	11.1 %	11.4 %	11.5 %	16.0 %

* T : Interval Time of Modified-Data References
*mc : machine cycle

4.1.3 Memory Throughput Evaluation with Simulator

A comparison of the memory throughput for Prolog is important for the choice of the memory system organization, thus we evaluated this with the simulator.

• Method

In order to evaluate the average memory access time for various memory organizations, we developed a memory system simulator. The memory system simulator is driven by memory access timing and memory addresses obtained by the micro level Prolog simulator, which was described earlier. In this memory simulator, the behavior of the write buffer and the interleaved main memory are simulated dynamically.

• Memory System Model

For a store-swap cache, two models are evaluated.

Model 1: if a block to be replaced has been modified when a cache miss occurs, the block is copied back before the required block is loaded in the cache memory.

Model 2: if a block to be replaced has been modified when a cache miss occurs, the block is pushed into a temporary buffer (copy-back buffer) before the required block is loaded in the cache memory. Then the contents of the copy-back buffer are copied back to the main memory in background.

The average access time T of the store-swap cache can be defined as follows:

$$T = (Nr * Tr + Nw * Tw) / (Nr + Nw)$$

where

$$Tr = hcr * Tca + (1 - hcr) * \{dr * (Ts+Tcb+Tbk) + (1 - dr) * (Ts+Tbk)\},$$

$$Tw = hcw * Tca + (1-hcw) * \{dw * (Ts+Tcb+Tbk) + (1-dw) * (Ts+Tbk)\},$$

Nr, Nw : access frequencies for read and write accesses, respectively,
hcr, hcw : cache hit ratios for read and write accesses, respectively,
dr, dw : the ratios that the replaced block is modified for read and write accesses, respectively,
Tca : cache access time,

Tbk : block transfer time from the main memory to the cache,

Tcb : block copy-back time from the cache to the main memory for Model 1, and from the cache to the copy-back buffer for Model 2, and

Ts : busy time of the main memory for Model 2, due to the copy-back operation from the copy-back buffer to the main memory.

The memory system simulator simulates the micro instruction steps taking into account all the above factors and relations that determine the average access time.

As for store-through cache, it is combined with the multi-stage write buffer and an interleaved main memory. If a read miss occurs in the store-through cache, the required block is fetched after the contents of the write buffer are completely copied back to the main memory. The factors and their relationship, incorporated into the simulator, for the average access time Ta of the advanced store-through cache are represented as follows:

$$Ta = (Nr * Tr + Nw * Tw) / (Nr + Nw)$$

where

$$Tr = hcr * Tca + (1 - hcr) * (Twe + Tbk),$$

$$Tw = k * Twb + (1 - k) * (Tws + Twb),$$

k: acceptance ratio of the write buffer,

Twb: write buffer access time,

Twe: wait time until the write buffer becomes empty, and

Tws: wait time until the write buffer has room for write access.

Twe and Tws is affected by the write buffer size, the number of interleaves and the store access time of the main memory.

• Measurement Condition

(1) The values of hcr, hcw, dr, dw, Ts, k, Twe and Tws are obtained by simulation of each memory system. The hit ratio of the cache memory is a function of the cache capacity, block size, the number of associated sets, and the replacement algorithm. In these models, we evaluated the effects of the hit ratio mainly as a function of the cache capacity. We assumed a 16-byte block size, 2-set size for the set-associative cache, and the FIFO replacement algorithm in all the evaluations.

(2) The access time of the cache memory, the store access time of the main memory and the block transfer time from the main memory to the cache memory are in the ratio of 2 : 15 : 20, which is based on the IPP ratio. The access time of the write buffer is equal to that of the cache memory.

(3) The block transfer time Tbk of each cache is equal to 10 times the cache memory access time, irrespective of the main memory interleaves.

(4) The block copy-back time Tcb is equal to 10 times the cache memory access time for Model 1 of the store-swap cache, and is equal to 4 times for Model 2, which is based on the IPP ratio.

(5) In the initial state of the store-swap cache, the ratio of the modified block is 50% 11) in order to take the start up effect into account.

(6) Instruction fetch is not evaluated, because the local memory for instruction fetch is commonly separated from that for data fetch in recent memory systems.

• Result

Fig.3 shows the average access time obtained from the simulation of all benchmarks for the store-swap cache and the store-through cache. In the store-through cache, we evaluated several organizations, which have different write buffer sizes and different

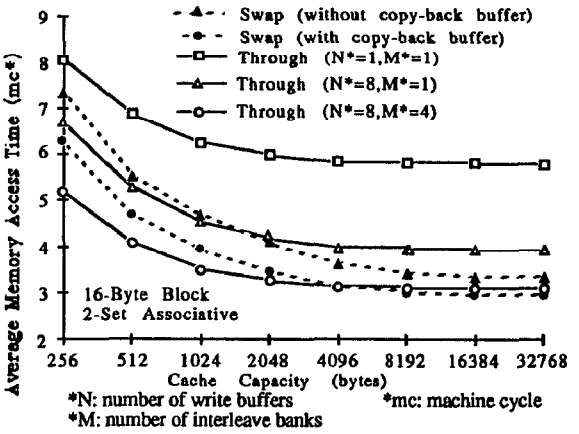

Fig. 3 Comparison of Average Access Time

numbers of interleaves of the main memory. Fig.3 indicates that the advanced store-through cache with the 8-stage write buffer and the 4-way interleaved main memory always has better performance than the store-swap cache without copy-back buffer, and the same performance as the store-swap cache with copy-back buffer. However in a small cache capacity, the advanced store-through cache has better performance than each model of the store-swap cache. This result is mainly due to the hit ratio degradation of the store-swap cache. In executing Prolog programs, the local and the heap stack are accessed simultaneously, and the accesses for each stack are scattered through the top and deep regions of the stack. In the store-swap cache, access conflicts occur more frequently than in the store-through cache, because the block is loaded when both read and write miss hits occur. In addition to this, the overhead to copy back the modified block to the main memory makes the average access time of the store-swap cache worse. On the other hand, in the store-through cache, the write access throughput is improved greatly by the multi-stage write buffer and the interleaved main memory, so concentrated write accesses caused by saving CHOICE-POINT's are processed effectively.

Fig.4 shows the read and write average access time of several organizations of each cache with the 16-Kbyte cache capacity. In the store-through cache with one-stage write buffer and no interleaved main memory, the processing ability of the write access drops remarkably due to overflow of the write buffer. If an 8-stage write buffer is provided to the store-

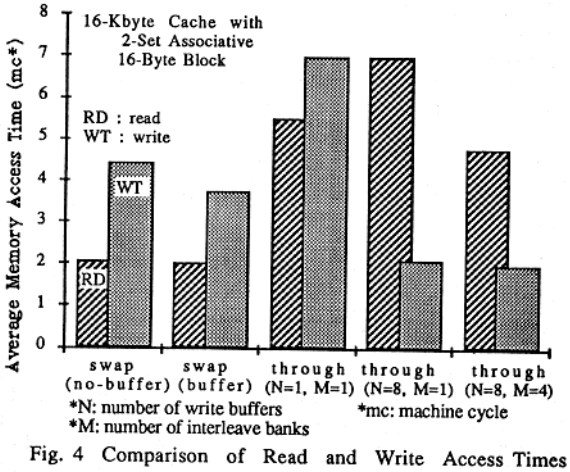

Fig. 4 Comparison of Read and Write Access Times

through cache, the processing ability for the write access is improved, but the average read access time becomes large. This is because the read must wait until the write buffer becomes empty on occurrence of a read miss. When the store-through cache is provided with the 8-stage write buffer and the 4-way interleaved main memory, it has good performance for read and write accesses, because the interleaved main memory provides a high service rate for the write buffer.

By simulator evaluation, it is clear that the advanced store-through cache shows the same performance as the store-swap cache with copy-back buffer. As a result, in consideration with reliability, we concluded the advanced store-through cache is more suited to the IPP than the store-swap cache.

4.2 Stack buffer vs. Store-through Cache

In this comparison, we examined four issues to judge the suited memory system for the IPP, that is effects on procedural languages, reliability, control complexity and memory throughput for Prolog.

4.2.1 Effects on Procedural Languages

To achieve high memory throughput is also necessary to speed up procedural languages like C, which is a stack-oriented language 10). However, if the hardware stack is employed, it needs to be managed explicitly by instructions. Thus, extension of the general purpose instructions and development of a sophisticated compiler are necessary. On the other hand, a cache memory, which is already applied to general purpose computers, can give good performance for both Prolog and procedural languages.

4.2.2 Reliability

As mentioned in the previous Section, a store-through cache has high reliability. On the other hand, in a hardware stack, some valid data in them may not exist in the main memory, thus some way is needed to realize high reliability.

4.2.3 Control Complexity

In general, the hardware stack is accessed via a virtual address, and some valid data in it may not exist in the main memory. Therefore, whenever task switching occurs in the multi-virtual memory, all the data in the hardware stack must be saved in the main memory. A cache memory reduces the task switching overhead, because it can be accessed by a physical address. The task switching overhead is a serious problem for the IPP, because there are many transactions to be processed in the real expert system.

4.2.4 Memory Throughput for Prolog

The most significant advantage of a hardware stack is a high access speed, since there is no need for accessing and comparing an address tag which are indispensable for a cache memory. But performance of the hardware stack depends heavily on the access locality. Namely stack accesses must be concentrated on the top of the stack. The analysis shown in Section 3 shows that the hardware stack is effective for the local stack accesses but not for the heap stack accesses because of the low locality. Therefore, the performance improvement is not expected for the program in which the heap stack accesses are dominant. As for a cache memory, its access time is worse than that of hardware stack. But it can hold the plural stack data or the top and deep portion of the stack data simultaneously. Thus it has high flexibility for scattered accesses.

4.2.5 Memory Throughput Evaluation with Simulator

The method of evaluation, the memory system model of a store-through cache and measurement condition were already mentioned in Section 4.1. Here we describe the hardware stack model and an additional measurement condition.

• Hardware Stack Model

There are several strategies for the Prolog-oriented hardware stack, such as a stack buffer, which holds CHOICE-POINT and ENVIRONMENT, and a choice-point-buffer, which holds only the current CHOICE-POINT 5) 8) 9). We took the more effective stack buffer strategy for evaluation. The stack buffer is combined with a store-through cache memory in the hierarchy as in a commercialized Prolog processor 6). The stack buffer is organized on the wrap around buffer shown in Fig.5, and it holds the top portion of the local stack. The valid data area on the stack buffer is shown with two pointers, the top pointer and the base pointer. The valid flag represents the validity of the whole buffer.

The control algorithm of the stack buffer is based on the algorithm shown in 8). The instructions, which create CHOICE-POINT or ENVIRONMENT, are used to make room in the stack buffer for new frames. If there is not enough space for these new frames, the data pointed by the base pointer are copied back to the combined cache memory until sufficient space is obtained. The instructions discarding frames do nothing but modify the top pointer of the stack buffer. If all the frames on the stack buffer become unnecessary, the valid flag is invalidated. In this case, the valid frames in the stack are not loaded into the stack buffer. The accesses of the local stack which access out of the stack buffer are serviced by the cache memory. Memory accesses to the heap stack, the trail stack, and the other areas are serviced by the cache memory. The factors and their relationship, incorporated into the simulator, for the average access time Ts of the stack buffer are represented as follows:

$$Ts = (Nr * Tr + Nw * Tw + Ncb * Tcb) / (Nr + Nw)$$

where

$$Tr = hbr * Tsb + (1 - hbr) * \{ hcr * Tca + (1 - hcr) * (Twe + Tbk) \},$$
$$Tw = hbw * Tsb + (1 - hbw) * \{ k * Twb + (1 - k) * (Tws + Twb) \},$$
$$Tcb = k * Twb + (1 - k) * (Tws + Twb),$$

Ncb: number of copy-backed words,
hbr, hbw: stack buffer hit ratio for read and write accesses, respectively, and
Tsb: stack buffer access time.

• Additional Measurement Condition

(1) The values of Ncb, hbr, hbw, hcr, k, Twe and Tws are obtained by the memory system simulation.
(2) The access time of the stack buffer, and that of the cache memory are in the ratio of 1 : 2.

• Result

The average memory access time for the stack buffer and the cache memory are plotted in Figs.6 and 7 for the various cache capacities, measured

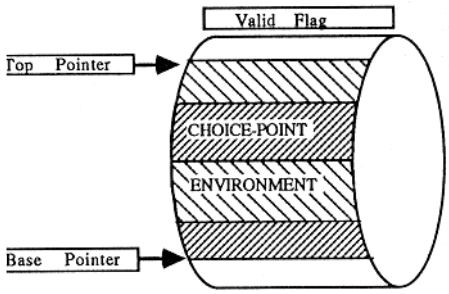

Fig. 5 Structure of Stack Buffer

respectively for Group 2 benchmark, and Group 1 benchmark programs. In this evaluation, the store-through cache was assumed to be combined with the 4-way interleaved main memory for all organizations. The evaluated organizations are as follows:

1. Store-through cache with 1-stage write buffer
2. Advanced store-through cache with 8-stage write buffer
3. Stack buffer combined with store-through cache
4. Advanced stack buffer combined with advanced store-through cache

For the average access time of all accesses for Group 2 shown in Fig.6 (b), the 256-stage stack buffer can give an 80% performance improvement to the store-through cache with 1-stage write buffer. This performance improvement is larger than that of the advanced store-through cache of 40%. This is because the 256-stage stack buffer works well for Group 2 benchmarks due to the high locality of the local stack accesses, as shown in Fig.6 (a). But if the stack buffer size is not sufficient (for example 32-stage in Fig.6 (b)), the effect of the stack buffer is decreased, mainly due to the copy-back operation to the main memory needed to make room for new frames. Then the 32-stage stack buffer with no-multi-stage write buffer only has a performance equal to that of the advanced store-through cache. If the stack buffer combined with the advanced store-through cache, it can double the performance of the store-through cache with 1-stage write buffer, even if it does not have too large a stack buffer size. The reason for this is that the copy-back operation to the main memory is processed effectively by the multi-stage write buffer.

From Fig.7 (a), the average local stack access time is seen to be improved greatly for the Group 1 benchmarks, if a 256-stage stack buffer is provided. But in the average access time for all accesses shown in Fig.7 (b), the 256-stage stack buffer cannot give much performance improvement to the store-through cache with 1-stage write buffer. This is because heap accesses occupy the main parts of all accesses and the stack buffer cannot give any improvement to heap stack accesses. On the other hand, the advanced store-through cache provides a 35% performance improvement to the store-through cache with 1-stage write buffer, because the multi-stage write buffer can process the heap stack access effectively. The stack buffer can achieve the same performance as that of the advanced store-through cache, when it is combined with the multi-stage write buffer.

Another observation is obtained from the results for Group 1 benchmarks. Performance becomes worse in local stack accesses, if the small size stack buffer is

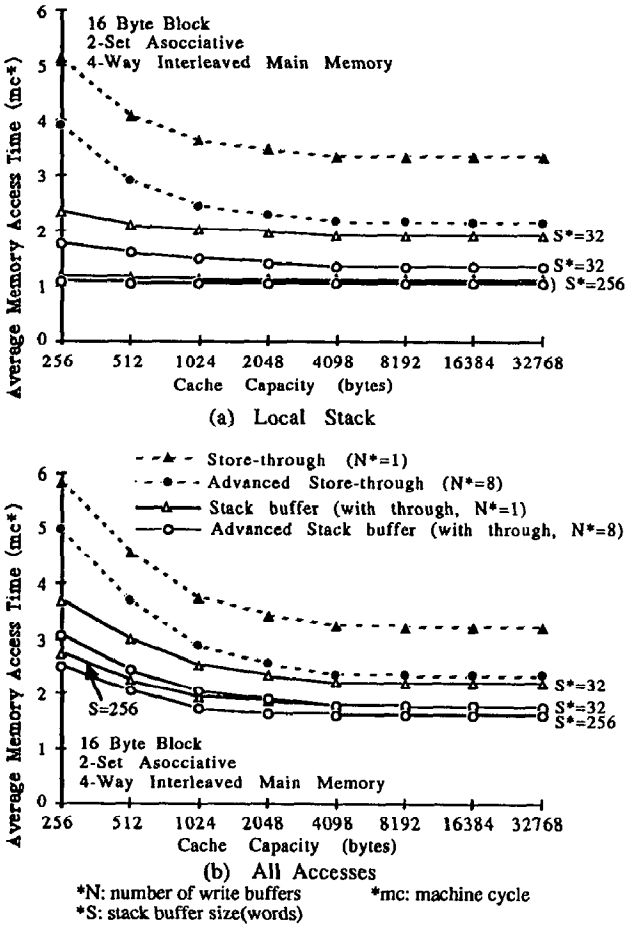

Fig. 6 Average Access Time of Group 2 Benchmarks

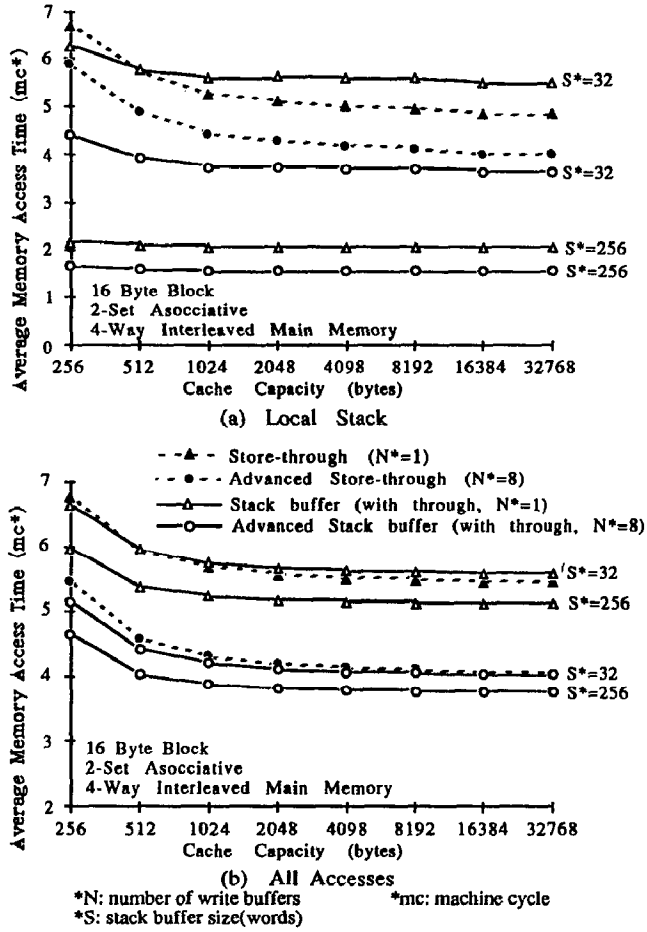

Fig. 7 Average Access Time of Group 1 Benchmarks

combined with a store-through cache with 1-stage write buffer. This is because ENVIRONMENT's are successively stacked on the local stack, and frequent copy-back operations are needed to make room for new ENVIRONMENT's.

A comparison of memory throughput for Prolog between the stack buffer and the store-through cache can be summarized as follows:

(1) For the Group 2 benchmarks, which access mainly the local stack, a stack buffer can give a 50%-80% performance improvement to the store-through cache with 1-stage write buffer. But for the Group 1 benchmarks, which access mainly the heap stack, the stack buffer cannot give as much performance improvement.

(2) An advanced store-through cache with 8-stage write buffer can achieve the 35%-40% performance improvement over the store-through cache with 1-stage write buffer irrespective of the benchmark characteristics.

(3) A stack buffer combined with advanced store-through cache gets the highest performance and flexibility of all organizations. It can give a 50%-100% performance improvement to the store-through cache, independent of the benchmark program features.

Based only on performance, the stack buffer with a store-through cache and a multi-stage write buffer is the best choice. But in consideration of the amount of hardware, reliability and complexity, it takes a great effort to implement this. On the other hand, the advanced store-through cache with the multi-stage write buffer and the interleaved main memory can realize performance improvement both in Prolog and procedural languages with comparatively little hardware, and less effort to implement. Therefore, we concluded that the advanced store-through cache is the most suitable organization for the integrated Prolog processor.

The IPP is composed of 144 ECL LSI's, each of which is a 2000- or 5000- gate gate-array. It achieves a machine cycle time of 23 ns. The store-through cache with the 8-stage write buffer and the 4-way interleaved main memory is adopted to the IPP. The integrated hardware occupies only 2 or 3% of the entire processor hardware. The performance of the IPP measured for Append program is 1.36 MLIPS 4).

5. CONCLUSIONS

To realize a high performance integrated Prolog processor (IPP), we investigated the most suitable memory system strategy for it in consideration of the performance for Prolog, effects on procedural languages, reliability, and control complexity.

The memory access characteristics of Prolog were analyzed with a micro level simulator. We found that: the write access ratio of Prolog was larger than that of procedural languages; in order to improve performance, it was necessary to process large, concentrated write accesses effectively; and the memory system must have flexibility for scattering accesses, such as accesses to plural stacks or to the deep stack area.

We compared stack buffer strategies and the conventional cache memory (store-through and store-swap cache). In comparison between store-swap cache and store-through cache, it was clarified that the advanced store-through cache, which was combined with a multi-stage write buffer and an interleaved main memory, showed the same performance as the store-swap cache with copy-back buffer. From consideration of data reliability, we concluded that the advanced store-through cache was more suitable for the IPP than store-swap cache.

As for a comparison between the stack buffer and advanced store-through cache, it was clarified that the stack buffer could achieve higher peak performance, but it was affected by the features of the benchmark programs. On the other hand, the advanced store-through cache had a high flexibility for scattering access and consistently gave a 35%-40% performance improvement to store-through cache without multi-stage write buffer. It was clarified that a combination of these strategies gave the highest performance.

As a result, in consideration of reliability, effects on procedural languages, control complexity, and the ease of implementation, we concluded that the advanced store-through cache with a multi-stage write buffer and an interleaved main memory is the most suitable organization for the IPP.

REFERENCES

1) D. H. Warren, "An Abstract Prolog Instruction Set," Technical Note 309, Artificial Intelligence Center, SRI International, October 1983.

2) S. Abe et al., "High Performance Integrated Prolog Processor IPP," Proceedings of the 14th International Symposium on Computer Architecture, June 1987, pp 100-107.

3) S. Yamaguchi et al., "Architecture of High Performance Integrated Prolog Processor IPP," Proceedings of 1987 Fall Joint Computer Conference, October 1987.

4) K. Kurosawa et al., "Instruction Architecture for a High Performance Integrated Prolog Processor IPP," Fifth International Conference and Symposium on Logic Programming, August 1988, pp 1506-1530.

5) A. M. Despain and Y. N. Patt, "Aquarius -- A High Performance Computing System for Symbolic/Numeric Applications," Proceedings of Compcon 85 Spring, February 1985, pp 376-382.

6) K. Taki et al., "Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI)," International Conference on Fifth Generation Computer System, November 1984, pp 398-409.

7) E. Tick and D. H. Warren, "Towards a Pipelined Prolog Processor," 1984 International Symposium on Logic Programming, February 1984, pp 29-40.

8) E. Tick, " Prolog Memory Referencing Behavior," Technical Report No.85-281, Stanford University, September 1985.

9) E. Tick, "Data Buffer Performance for Sequential Prolog Architectures," Proceedings of the 15th International Symposium on Computer Architecture, June 1988, pp 434-442.

10) D. R. Ditzel and H. R. McLellan, "Register Allocation for Free : The C Machine Stack Cache," Symposium on Architecture Support for Programming Languages and Operating Systems, 1982, pp 48-56.

11) A. J. Smith, "Cache Evaluation and the Impact of Workload Choice," Proceedings of the 12th International Symposium on Computer Architecture, June 1985, pp 64-73.

12) H. G. Okuno, "The Report of the Third Lisp Contest and the First Prolog Contest," WG SYM 33-4 IJP Japan, Sept. 1985, pp 1-24.