JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 22, 215-227 (2006)

Short Paper

Advanced High-Level Cache Management by Processor Access Information

JONG WOOK KWAK, CHEOL HONG KIM, SUNGHOON SHIM AND CHU SHIK JHON
Department of Electrical Engineering and Computer Science
Seoul National University
Seoul, 151-742 Korea
E-mail: leoniss@panda.snu.ac.kr

In this paper, we propose an advanced high-level cache management policy based on the processor access information, named as L1VPAI (L1 plus Victim cache with Processor Access Information). The L1VPAI is a cache replacement policy that uses the frequency of the specific cache line. In this policy, conflicted lines in the L1 cache are placed selectively in the victim cache or the level 2 (L2) cache based on previous memory access patterns. In this manner, the L1VPAI policy can make the frequently used address of cache locations reside longer in the high-level caches. We simulate our policy with RSIM, the event-driven simulator, and analyze the simulation results. The simulation results show that the average execution time of the L1VPAI outperforms the simple victim cache (L1V) by up to 6.44%. Moreover, performance gain is expected to increase, in the case of multiprocessor systems.

Keywords: hierarchical memory system, L1 cache, victim cache, cache replacement policy, processor access information

1. INTRODUCTION

In recent years, the speed gap between microprocessors and dynamic memories has steadily increased. The key reason for this is that improvements in processor performance have outpaced that of the main memory performance. Further, processor technologies like pipelining and superscalar reduce the CPI significantly, but the reduction of average memory access time is meager. Therefore, there is a processor memory performance gap that computer architects must try to close [1].

To reduce the speed gap between processor cycle time and memory access delay, the hierarchical memory system was proposed. An L1 cache in its hierarchy is very effective in reducing average memory access time. Due to temporal or spatial locality, the small and fast L1 cache is able to satisfy most memory requests issued by the processor, and there is no need to wait for slow a main memory response in most cases. However, a problem still remains. If the L1 cache does not provide enough hit rate, frequent cache

Received February 19, 2004; revised June 11 & November 18, 2004; accepted December 22, 2004.
Communicated by Chu-Sing Yang.

line replacements will be generated, and this makes the memory's response time much slower. To support an enough hit rate without impairing access time, victim cache, another high-level cache, was proposed by Jouppi [2]. The victim cache contains the blocks that are discarded from the L1 cache, to give a second chance. On a processor's reference miss, the victim cache is usually checked in parallel with L1 cache to see if it has the desired data before going down to the next lower level cache. In this way, the victim cache reduces the conflict misses without impairing average memory access time.

Although the victim cache is effective when frequent conflict misses occur, the victim cache does not consider whether the conflicted data will be re-accessed again or not in the near future. Generally, there always exist frequently accessed parts in application programs, especially in the case of recursive or loop structure. However, this is not true in some application programs depending on the implement of their data structure. Johnson, *et al.* showed the main loop body of the *026.compress* program from the SPEC benchmark suite, and demonstrated the re-accessibility [3]. As shown in their research, the original victim cache policy does not provide adequate hit rate in this case. To overcome this problem, we propose an advanced high-level cache management policy, named as L1VPAI, based on the Processor Access Information. The L1VPAI is a cache replacement policy that uses the frequency of the processor's memory accesses and makes the higher frequency address of cache locations reside longer in the high-level caches than the briefer ones. This policy divides the data into frequently accessed parts and non-frequently accessed parts, and conflicted blocks in the L1 cache are placed selectively in the victim cache or the L2 cache by the previous access information of the processor.

2. BACKGROUND AND RELATED WORK

2.1 Hierarchical Memory System

A memory structure for computer systems is organized hierarchically with L1, L2, and even L3 caches in its hierarchy. As shown in Fig. 1, memory access time can be reduced by adding more cache levels between the processor and the main memory. In a cache design paradigm, the L1 cache should be small enough to match the clock cycle time of a fast processor, while L2 cache should be large enough to capture many accesses that would go down to main memory. Thus, L1 cache is usually implemented as low associative or direct mapped method for fast hit time, whereas L2 cache is implemented as high associative for low miss rate. Although cache that is implemented by low associative or direct mapped method is advantageous because of its fast hit time, it suffers from high miss rate. Hence, the victim cache is used as well to reduce the miss rate of L1 cache. Fig. 2 shows a hierarchical memory system with a victim cache.

In this structure, when the processor's reference miss occurs in the L1 cache, the newly fetched data from the L2 cache is allocated in the L1 cache and the conflicted line in L1 cache moves to the victim cache, not to the L2 cache. In this way, the victim cache has the lines that are discarded by the previous conflict misses in L1 cache. This is very effective because there are usually much re-referenced data in application programs, and the victim ache plays the role of giving a second chance to the previous conflicted data

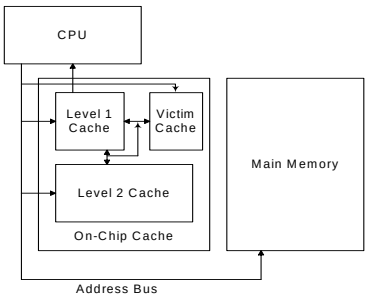

Fig. 1. Hierarchical memory system.

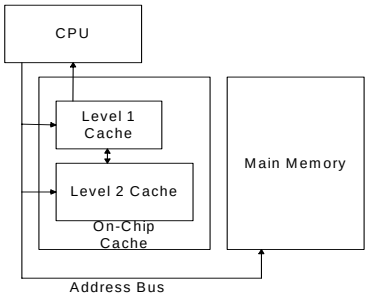

Fig. 2. Hierarchical memory system with victim cache.

by supplying the data directly without going down to the L2 cache. Norman P. Jouppi found that a victim cache of one to five entries is effective at reducing conflict misses, especially for small, direct-mapped caches. He also showed that, depending on the programs, a four-entry victim cache removes 20% to 95% of conflict misses in a 4KB direct-mapped cache [4].

2.2 High-Level Cache Management Policy

Several methods have been proposed to manage the high-level cache efficiently. First, the STS model [5] shows the STS cache design and formally defines the locality type. In addition, the MAT model [6] classifies memory space into frequently referenced areas and non-frequently referenced areas. On the other hand, Dual Cache [7] divides the cache into split temporal and spatial cache. Selective Cache [8] is able to provide the ability to disable a subset of ways in a set-associative cache during the period of cache activity mode. Also, many other evaluations have been performed on the victim cache as well, including Selective Victim Caching [9]. In Selective Victim Caching, the incoming blocks of the L1 cache are placed selectively in the main cache or small victim cache by a prediction based on past history. Interchanges of blocks between the main cache and victim cache are also performed selectively.

3. HIGH-LEVEL CACHE MANAGEMENT WITH PROCESSOR ACCESS INFORMATION

3.1 Cache Replacement Policy

In the hierarchical memory system which includes the victim cache, the high-level cache management is as follows. The memory system searches L1 cache lines to find a correct entry when a processor request is generated to the L1 cache. If the correct entry is found, the tag and valid field are checked. If the tag does not match, the processor's request moves down to the L2 cache. However, if the memory system includes a victim cache, the victim cache is searched in parallel with the L1 cache to see if it has the desired data or not. If the requested data is found in the victim cache, the victim cache responds to the processor's request and then the requested line in the victim cache is

re-allocated in L1 cache. On the other hand, if both L1 and victim cache do not have the requested data, the request moves down to the L2 cache, and the requested data in L2 cache (if exists) is loaded into the L1 cache. At this time, the conflicted line in the L1 cache is moved to the victim cache, and one of victim cache entries, if it is fully occupied, moves down to the L2 cache. Fig. 3 shows the operations between L1 and victim cache. The former shows the case of a reference miss in both L1 and victim cache, and the latter shows the case of a reference hit in victim cache.

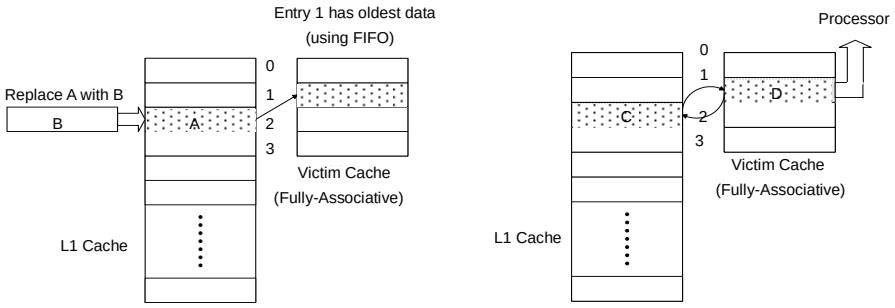

Fig. 3. Replacement policy between L1 cache and victim cache.

3.2 Cache Replacement Policy with Processor Access Information

Although the L1 cache and victim cache together increase the hit rate of frequently re-referenced data which causes conflict misses, this structure allocates the conflicted data to the victim cache without any concern for re-use, and even allocates data which will not be re-referenced again. To avoid this unnecessary data allocation, we use the processor access information to predict whether it will be accessed again or not. In this policy, we divide cache management policy into 1) *replacement between L1 and victim cache*, and 2) *replacement between L1 and L2 cache*. That is, the conflicted lines in the L1 cache are replaced selectively to the victim cache or the L2 cache, based on the previous access pattern. In this section, we suggest how to obtain the access information from the processor's request, and propose a new high-level cache management policy, called L1VPAI (L1 cache plus Victim cache with Processor Access Information).

To predict whether the line will be re-accessed again, we should store the access patterns of the previous requests. For this purpose, we add "*Access Pattern Bit* (*AP bit*)" to each cache line. The AP bit is similar to a counter, which stores the number of accesses for each cache line. If a new cache line is allocated, the value of the AP bit in the requested line is set to 0, and the counter increments when processor access occurs. The AP bit is implemented as a saturated counter. If the counter reaches its maximum value, it does not increase its value and just maintains the maximum value. In this way, when the cache line is replaced, we check the counter value with a predefined threshold value to decide whether the line is re-accessed or not. If the counter value is greater than the threshold value, we predict that the replaced line will be accessed again in the future. Fig. 4 shows the normal cache line and the modified cache line which contains the AP bit.

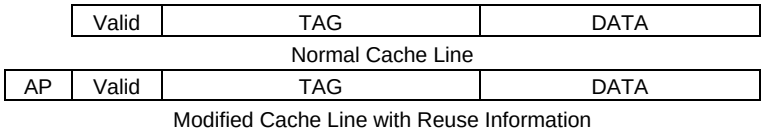

Fig. 4. Structure of L1 cache line.

An example of the operation of the L1VPAI policy is implemented as follows. A processor issues a data request to the L1 cache, and then the L1 cache and the victim cache are searched in parallel to check if they have the desired data. If the requested data is found in the L1 cache, the requested data is transferred to a processor, with incrementing the counter of the requested cache line. On the other hand, if the requested data is found in the victim cache, it is transferred to the processor, and the requested line in the victim cache is moved to the L1 cache line. At this point, the counter value of the conflicted line in the L1 cache (i.e., AP bit) is checked to see whether its value is larger than the threshold value. If the counter value is larger, the replacement occurs between the L1 and the victim cache. If the counter value is smaller, the replacement occurs between the L1 and L2 caches. Also, if the requested data is missed in both the L1 and victim caches, the requested data is loaded from the L2 cache and is placed in the L1 cache. Then the counter value of conflicted line in the L1 cache is checked. If the counter value is larger than the threshold value, the conflicted line is moved to the victim cache. If not, the conflicted line is moved to the L2 cache. Fig. 5 shows two scenarios which use the processor access information: read hit in victim cache, and read miss in all high-level caches.

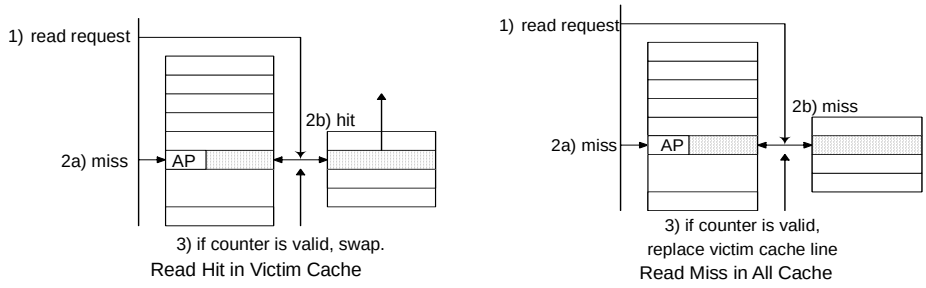

Fig. 5. Replacement policy with processor access information.

However, victim cache usually plays a major role of resolving the so called Ping-Pong effect. In this case, reduction of frequent conflict misses in a short period of time is critical for overall system performance. Therefore, if we focus only on the re-access information without considering the ping-pong effect, the system performance will not improve as we expect. Thus, we adjust our policy as follows. In case of cache lines that cause the Ping-Pong effect, we just use the original victim cache policy. A policy adjustment is a case of "*miss in L1 cache and hit in victim cache*." In this case, we do not use the access information in order to resolve the Ping-Pong effect. Instead, we just load the requested entry of victim cache to L1 cache and the conflicted line in L1 cache moves to victim cache. In other words, the processor access information is applied only

when the reference miss occurs in both L1 and victim cache. Fig. 6 shows an example of the adjusted high-level cache replacement policy

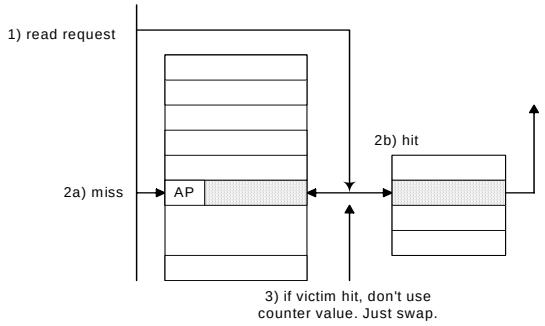

Fig. 6. Replacement policy adjustment in case of “Hit in Victim Cache”.

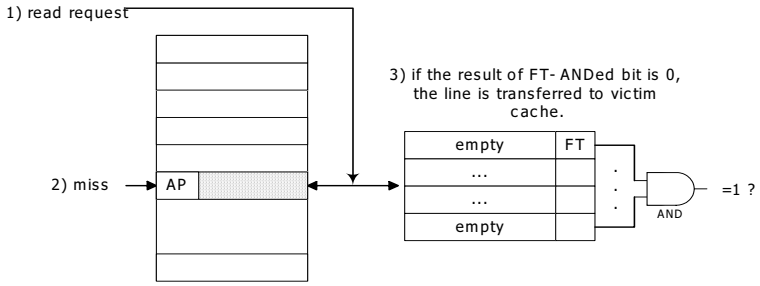

Fig. 7. Management of empty victim cache.

Fig. 7 shows another adjustment for the L1VPAI policy. The FT bit is newly added in the victim cache. The FT bit is 0 when the entry of a victim cache is available and is 1 when the entry of a victim cache is occupied. The output of the AND gates for each FT bit (FT-ANDed) indicates whether there exists available victim cache entries. If the output of the FT-ANDed is 0, the victim cache still has available empty entries and if the output of the FT-ANDed is 1, the victim cache is fully occupied. In this manner, we do not use the processor access information if the output of the FT-ANDed is 0. Instead, the conflicted line in the L1 cache is just moved to the victim cache. This is for the resolution of the empty victim cache problem. That is, some parts of the victim cache may be empty at an early execution time. Cache flushing due to the cache coherence protocol is another factor that causes an empty victim cache line. In addition, some entries of the victim cache may be invalidated by other processors’ requests or inclusion property. In theses cases, there may be no candidate lines in the victim cache to replace the line with L1 cache, so we just use the original victim cache policy when FT-ANDed is 0. In other point of view, if there are available entries in the victim cache, we just use them because the victim cache entries are currently available. It helps utilize the victim cache more efficiently. Fig. 7 shows the scenario of an empty line management in the victim cache.

Finally, Fig. 8 shows the overall algorithm of the L1VPAI policy. It utilizes the processor access information in addition to the considering the ping-pong effect and the empty victim cache management.

```
// Memory Initialization, AP bit and FT bit reset
Memory_Init();

while(Processor_Request()) {

Case 1: Access to block b, hit in main cache;

        // Increment the Counter Value
        if (AP_count[b] < Access_Threshold) then
        AP_count[b]++;

        Fetch_to_Processor(b);

Case 2: Access to block b, hit in victim cache;
Let a be the conflicting block in main cache;

        // cache replacement without access information
        interchange a and b;

        reset(AP_count[b]);
        Fetch_to_Processor(b);

Case 3: Access to block b, miss in both main and victim;
Let a be the conflicting block in main cache;

        Fetch_from_L2Cache(b);

        // check the result of FT-ANDed
        if (! FT-ANDed) then

                move a to victim cache;
                move b to L1 cache;

        // compare the counter value with access threshold
        else if (AP_count[a] >= Access_Threshold) then

                move a to victim cache;
                move b to L1 cache;

        else
                move a to L2 cache;
                move b to L1 cache;

        end if

        reset(AP_count[b]);
        Fetch_to_Processor(b);
}
```

Fig. 8. High-level cache management policy (L1VPAI).

4. PERFORMANCE EVALUATION

In this section, we simulate and analyze the performance of the L1VPAI policy. We use the execution driven simulator, RSIM [10], to evaluate the performance of our policy. The RSIM, which stands for the Rice Simulator for ILP Multiprocessors, is a software package designed to evaluate shared memory multiprocessor architectures built from state-of-the-art processors. In the multiprocessor system features, the RSIM supports the sequential consistency memory model, wormhole-routed mesh network, and MESI coherence protocol. The RSIM directory protocol and a cache controller support cache-to-cache transfers as well. For system parameters, we assume that processor frequency is 1GHz, and system bus frequency is 100Mhz. L1 cache size is changed from 8KB to 32KB, and cache line size is 32Bytes. Victim cache is fully associative and has four entries. We randomly select FFT, MP3D, RADIX, LU, and WATER from SPLASH/ SPLASH-2 [11] as workloads.

The value which gives a clear distinction when changing the cache replacement policy is the miss rate of the cache memory system. First we determine an adequate counter size as the AP bit. This is the factor in deciding the re-access information for each cache line. Furthermore, after selecting the appropriate counter size, we can deduce the additional hardware costs in the cache module. Fig. 9 depicts the average miss rate for each cache size for five benchmark programs when we change a threshold value of the counter. The candidate counter value is changed from 2 to 5, and the numbers of processors are 1, 4, and 16. CNT in this result means the threshold value. As shown in Fig. 9, the best scenario is the case of "CNT = 2", and thus we use the "CNT = 2" as the threshold value to predict the re-access of the cache line. These results additionally indicate that each application program may in general have two parts. The two parts are a sequential part that is executed once and an iterative part that is executed repeatedly. The sequential part is executed "once" and the counter value will not be 2, and the iterative parts like a loop-structure will be executed "more than once" and the counter value will be at least "more than 2". Therefore, "CNT = 2" is enough to predict the processor's next access.

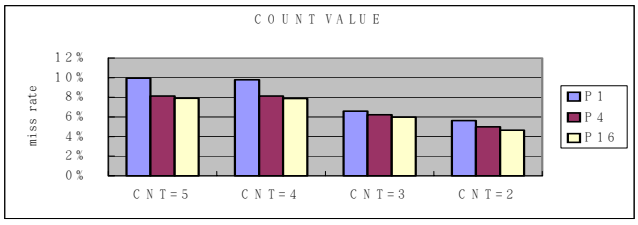

Fig. 9. Comparison of counter size.

The miss rates for each benchmark program are shown in Figs. 10 to 12. For each application, the *x-axis* is the number of processors, and the *y-axis* is the miss rate of the high level cache. The comparison targets are L1V, L1VPAI, and L1VSVC. The L1V means L1 plus victim cache in the hierarchical memory system, using the original victim cache policy. The L1VPAI is our policy which uses the processor access information.

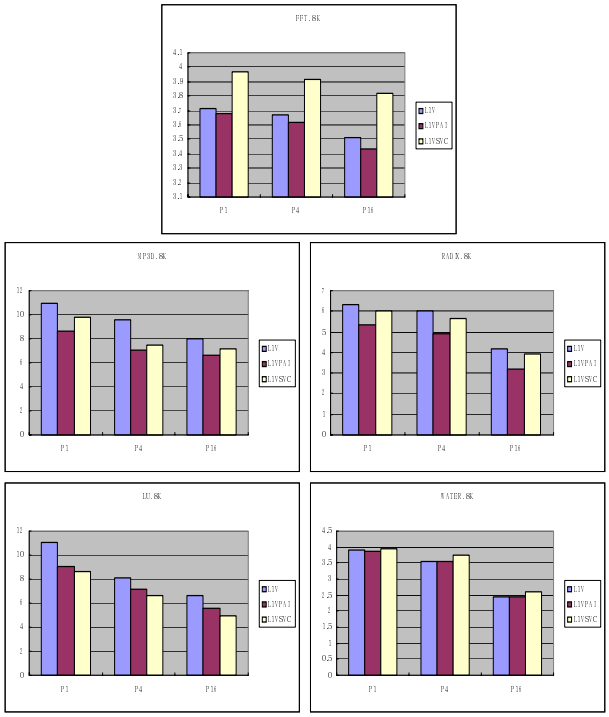

Fig. 10. Miss rate with 8KB cache.

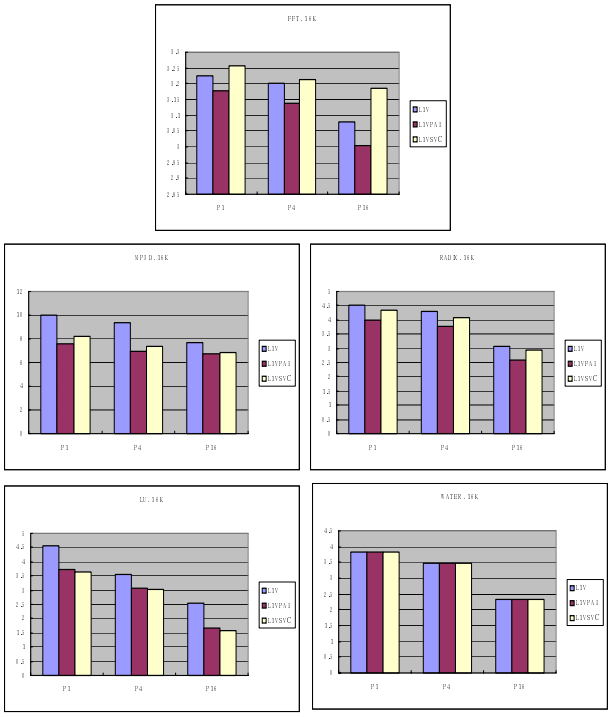

Fig. 11. Miss rate with 16 KB cache.

Finally, the L1VSVC is the Selective Victim Caching policy that was proposed by D. Stiliadis and A. Varma [9].

Fig. 10 shows the percentage of miss rate when the L1 cache size is 8KB. The miss rates are generally reduced as the number of processors increases for all applications. As shown in Fig. 10, the L1VPAI policy outperforms the L1V policy in all applications, and it outperforms the L1VSVC policy as well, except LU. Compared to the L1V policy, the miss rate enhancement is up to 2.43%. The miss rate improvement of three applications (MP3D, RADIX and LU) is 1.72% on average. In the cases of FFT and WATER, their miss rates are originally low, so the miss rate enhancements are relatively small compared to other benchmark programs. From these results, we can infer that the miss rate enhancement of the L1VPAI policy is especially large in cases of high miss rate applications. Compared to other applications, WATER has similar results regardless of the policies. We think that this property represents application dependent characteristics, and it suffers from severe interference of shared data among the processors. The L1VSVC policy, on the other hand, generally outperforms the L1V policy as well. Though the result of the L1VSVC policy has the lowest miss rates in LU, the miss rate of the L1VSVC policy is higher than the L1VPAI in other applications. Further, the miss rates of L1VSVC are even higher than the L1V policy in the cases of FFT and WATER.

Fig. 11 shows the 16KB case cache. Compared to Fig. 10, the miss rate in all benchmark programs is reduced due to the larger cache size. The miss rate reduction of P16 in all applications is especially large. This is due to the enough amounts of cache and their sizes for each 16-processor. Except for the facts mentioned above, the result of all applications are similar to the 8KB cache. The L1VPAI policy outperforms the L1V policy in all applications, and it outperforms the L1VSVC policy as well, except the LU. However, the miss rate difference in LU is reduced between the L1VPAI policy and the L1VSVC policy. In case of FFT, the L1VSVC policy still has the worst result.

Fig. 12 shows the case of 32KB cache. Compared to Figs. 10 and 11, the miss rate in 32KB cache is reduced more, and the miss rate reduction of P16 is greater than the 16KB case as well. These results are due mainly to the large cache size. For the 32KB cache, the L1VPAI policy outperforms the L1V policy and it outperforms the L1VSVC policy as well. The performance improvement is 1.87% on average. Compared to Figs. 10 and 11, the miss rate of LU in the L1VPAI policy is now almost the same as that of the L1VSVC policy. From the above, the results of the L1VPAI policy are mostly better than other cases, and the performance gain is expected to increase as the number of processors increases in the MP3D, RADIX, and LU cases.

The average execution times of five application programs for each policy are shown in Fig. 13. In these graphs, the execution times are normalized to those of the L1V policy, and each execution time is an average value for each cache size. As shown in Fig. 13, the results are similar to those of the miss rate. Generally, the L1VPAI policy requires less time than the L1V policy in all cases, and its enhancement is 6.44% at the most and 2.81% on average. The performance improvements of average execution times are more than those of the miss rate, especially in P4 and P16. This is due to the miss penalty. Within our survey and knowledge, our research is the first to evaluate the performance improvement of the average execution time with the victim cache policy in parallel systems. In parallel systems, the miss penalty requires local bus transactions and, sometimes, global network transactions. In the latter case, the miss penalty is larger than the case of

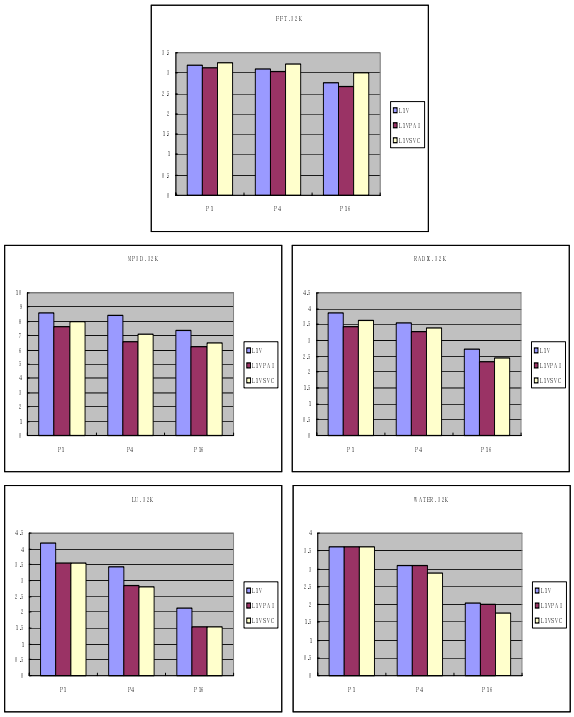

Fig. 12. Miss rate with 32KB cache.

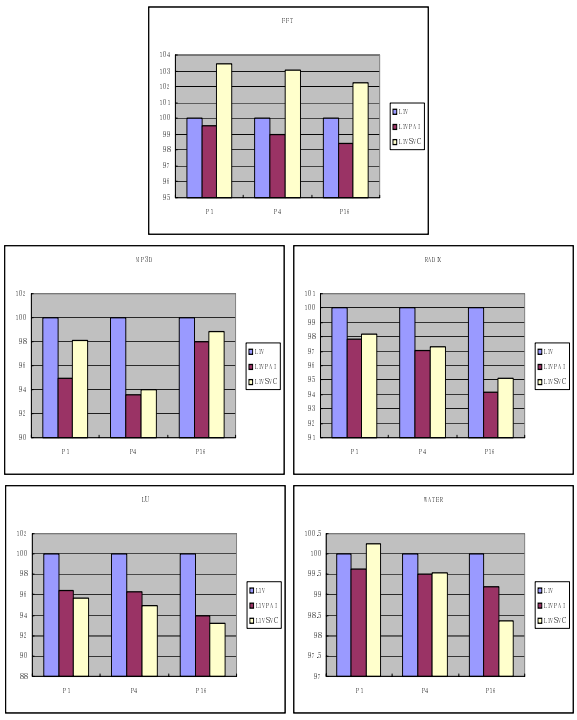

Fig. 13. Average execution time.

single processor systems and it becomes much larger if it requires long-distance global network transactions. The results of our L1VPAI policy are especially good in cases of MP3D, RADIX, and LU. For these three applications, the performance improvement is 4.18% on average.

5. CONCLUSIONS

The hierarchical memory system is a general solution to reduce the time gap between processors and main memories. In this paper, we proposed a new high-level cache management policy with the processor access information (L1VPAI). In the L1VPAI, we used the access pattern bit (AP bit) to trace memory access patterns. As a result, we showed the possibility of changing the high-level cache replacement policy with this information. In addition, we can resolve the empty victim cache problem and the Ping-Pong effect by suggesting use of the FT bit and some adjustments. Our advanced high-level cache management policy (L1VPAI) showed better average execution time up to 6.44% compared to the previous simple high-level cache management policy (L1V). It also outperformed the Selective Victim Caching policy (SVC). Moreover, the performance gain is expected to increase in case of multiprocessor systems.

REFERENCES

1. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann Publishers, Inc., 2001.
2. N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of 17th International Symposium on Computer Architecture*, 1990, pp. 364-373.
3. T. Jonson and W. W. Hwu, "Run-time adaptive cache hierarchy management vi reference analysis," in *Proceedings of 24th Annual International Symposium on Computer Architecture* (*ISCA-24*), 1997, pp. 315-326.
4. N. P. Jouppi, "Improving direct − mapped cache performance by the addition of a small fully-associative cache and prefech buffer," WRL Technical Note TN-14, 1991.
5. M. Prvulovic, *et al.*, "Split temporal/spatial cache: A survey and reevaluation of performance," *IEEE TCCA Newsletters*, 1999, pp. 1-10.
6. J. Yang and R. Gupt, "Frequent value locality and its application," *ACM Transactions on Embedded Computing Systems*, Vol. 1, 2002, pp. 79-105.
7. B. Juurlink, "Unified dual data caches," in *Proceedings of Euromicro Symposium on Digital Systems Design* (*DSD '03*), 2003, pp. 33-40.
8. D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *Proceedings of International Symposium on Microarchitecture*, 1999, pp. 248-259.
9. D. Stiliadis and A. Varma, "Selective victim caching: A method to improve the performance of direct-mapped caches," *IEEE Transactions on Computers*, Vol. 46, 1997, pp. 603-610.
10. C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve, "RSIM: simulating shared-memory multiprocessors with ILP processors," *IEEE Computer*, *Special Issue on*

High Performance Simulators, Vol. 35, 2002, pp. 40-49.
11. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "Methodological considerations and characterization of the splash-2 parallel application suite," in *Proceedings of 22nd International Symposium on Computer Architecture*, 1995, pp. 24-36.

Jong Wook Kwak (郭鍾旭) received the B.S. degree in Computer Engineering from Kyungpook National University, Taegu, Korea in 1998 and the M.S. degree in Computer Engineering from Seoul National University, Seoul, Korea in 2001. He is currently a Ph. D. candidate in the Department of Electrical Engineering and Computer Science at Seoul National University. His research interests are computer architecture, high-performance parallel computing, and low-power embedded systems.

Cheol Hong Kim (金哲弘) received the B.S. degree in Computer Engineering from Seoul National University, Seoul, Korea in 1998 and M.S. degree in 2000. He is currently a Ph.D. candidate in the Department of Electrical Engineering and Computer Science at Seoul National University. His research interests are computer architecture, embedded processor, low power system and multiprocessors.

Sunghoon Shim (沈聖勳) received the B.S. degree in Computer Engineering from Dongguk University, Seoul, Korea in 1997 and M.S. degree in Computer Engineering from Seoul National University, Seoul, Korea in 1999. He is currently a Ph.D. candidate in the Department of Electrical Engineering and Computer Science at Seoul National University. His research interests are computer architecture, multiprocessors system, embedded processor and low power system.

Chu Shik Jhon (全洲植) received the B.S. degree in Applied Mathematics from Seoul National University, Seoul, Korea in 1975, the M.S. degree in Computer Engineering from Korea Advanced Institute of Science and Technology, Taegeon, Korea in 1977, and the Ph.D. degree in Computer Engineering from the University of Utah in 1982. He was an Associated Professor in Computer Engineering Department at Iowa State University from 1983 to 1985. He is currently a professor in the Department of Electrical Engineering and Computer Science at Seoul National University. His research interests include parallel computing, and VLSI design automation.