An Adaptive Demotion Policy for High-Associativity Caches

Jubee Tada
Graduate School of Science and Engineering,
Yamagata University
4-3-16 Jonan, Yonezawa City
Yamagata, 992-8510, Japan
jubee@yz.yamagata-u.ac.jp

Masayuki Sato
Graduate School of Information Sciences,
Tohoku University
Aramaki Aza Aoba 6-3,
Aobaku, Sendai
Miyagi, 980-8578 Japan
masa@tohoku.ac.jp

Ryusuke Egawa
Cyberscience Center,
Tohoku University
Aramaki Aza Aoba 6-3,
Aobaku, Sendai
Miyagi, 980-8578 Japan
egawa@tohoku.ac.jp

ABSTRACT

Although the Least Recently Used (LRU) policy is known as a simple but high-performance cache replacement policy, high-associativity caches hardly adopt the LRU policy because of an increase in the hardware overheads. The Re-Reference Interval Prediction (RRIP) policy [3] is one of the high-performance policies that can suppress the hardware overheads. However, the RRIP policy cannot improve the performance when it is employed in higher-level caches, and in fact, the RRIP policy causes a significant performance degradation in the execution of several applications. This is because the RRIP policy controls the priority of the block without considering the priorities of all the blocks in the set at cache misses. In several applications, it causes the priorities of the existing blocks are minimized or unchanged.

To avoid this problem, this paper proposes a cache replacement policy named Adaptive Demotion Policy (ADP). This policy focuses on a subtraction value, which is subtracted from the priority value of each block at cache misses. According to the level of the cache hierarchy, ADP uses the half of average or the average of the priority values of all the blocks in the set as the subtraction value. This prevents that the priorities of the existing blocks are minimized or unchanged. Besides, ADP is suitable for various applications by the appropriate selection of its insertion, promotion and selection policies.

The evaluation results show that ADP can be implemented with fewer hardware overheads compared with the LRU policy. Moreover, the priority controller of ADP can operate faster than that of the LRU policy for high-associativity caches. The performance evaluation shows that ADP achieves the MPKI reductions at all the levels of the cache hierarchy and the IPC improvements, compared with the LRU and RRIP policies.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles

General Terms

Design

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2017) Bochum, DE, June 7-9, 2017.

Keywords

Cache memories, Cache replacement policy

1. INTRODUCTION

Recently, as the demand for the performance improvement of a memory system is increased, cache memories become more important. In set-associative caches, a cache replacement policy selects the victim for placing a new data from the lower-level memory hierarchy. As a cache replacement policy affects the performance of the cache, a sophisticated cache replacement policy is strongly required. One way to improve the performance of the set-associative cache is increasing its associativity.

Although the LRU policy is known as a simple but high-performance cache replacement policy, high-associativity caches hardly adopt the LRU policy due to the following problem. The LRU policy typically requires $\log_2 n$ state bits at all the cache blocks for the n-way set associative cache [1]. The growth of the associativity enlarges not only hardware resources for the LRU state bits but also the complexity of the control logic for updating the LRU state bits [1]. This will affect the cycle time of the processor, so it is hard to adopt the LRU policy in high-associativity caches. In addition, the LRU policy does not have a tolerance for scan access patterns because the LRU policy just considers the order of references [2].

The Re-Reference Interval Prediction (RRIP) policy [3] can suppress the hardware overheads with a high performance. Although the RRIP policy can achieve the performance improvement compared with the LRU policy at the last-level cache, the higher-level caches cannot benefit from the RRIP [3] because the RRIP policy is not suitable for several applications. As the RRIP policy does not consider the priorities of all the blocks in the set, the priorities of the existing blocks are minimized or unchanged at several applications. Therefore, an adaptive policy that controls the priorities of the blocks is needed to improve the performance.

This paper proposes a cache replacement policy called Adaptive Demotion Policy (ADP). The demotion policy decides the subtraction value, which is decreased from the priority values of the blocks at a cache miss. ADP generates the subtraction value based on the average of the priority values of all the blocks in the set. ADP can be implemented with small hardware overheads for high-associativity caches, and achieve the performance improvement compared with the LRU policy at all the levels of caches.

This paper implements the hardware that can dynamically change the insertion, promotion and selection policies. Then, this paper illustrates that ADP can achieve the performance improvement compared with the LRU and RRIP policies. In addition, this paper examines the effects of the insertion, promotion, and selection policies on the performance of ADP. By appropriate selection of these policies, ADP is suitable for various applications.

This paper is organized as follows. Section 2 outlines the basis of cache policies and reviews related work. Section 3 describes the behavior of ADP. Section 4 evaluates the performance of ADP. Section 5 concludes this paper.

2. RELATED WORK

The LRU policy selects the least recently used block as the victim at a cache miss. The LRU policy is well known as simple but a high-performance cache policy because the LRU policy can use the temporal locality.

However, high-associativity caches hardly adopt the LRU policy due to the following reasons. One reason is the complexity of the implementation of the control logic [1]. In order to implement n-way set associative caches, $\log_2 n$ bits are needed in every cache block for recording the order of references. Therefore, as the associativity increases, hardware resources required for these LRU state bits enlarge. In addition, the complexity of the control logic grows as the associativity increases [1]. Because it is required to update the LRU state bits every cache access, the updating cost will affect the cycle time of the processor. Therefore, it is hard to adopt the LRU policy in high-associativity caches, and the cache memory that has 4 or more associativity employs an approximate LRU policy like the Pseudo-LRU [4].

Another problem is a tolerance for scan access patterns. Because the LRU policy only considers the order of references, a new block remains in the cache for a long time, even if the block is never referred. Especially in scan access patterns, the cache will overflow by the blocks that are not re-referenced. Therefore, the LRU policy cannot be beneficial for scan access patterns.

The RRIP policy [3] is one of the high-performance policies that can suppress the hardware costs. The RRIP policy can achieve the performance improvement compared with the LRU policy at the last-level cache. However, the performance is the same as that of the LRU policy at the higher-level caches [3]. The RRIP policy prepares a priority value for each block, and the priority is promoted at a cache hit. At a cache miss, the block that has the minimum priority is searched from the lower way number in the set. If it exists, that block is evicted. Otherwise, the priorities of all the blocks in the set are decreased by one. The searching and decrement are repeated until a block with the minimum priority is found out. The above process is explained that the priorities of all the blocks are decreased by the smallest priority of all the blocks. When the hit rate of the cache is high, a cache miss minimizes the priorities of all the existing blocks in the set. This causes a harmful effect to the performance by evicting useful data. Besides, if there are blocks that have the minimum priority, the priorities of the blocks in the set are not decreased even though a cache miss occurs. This causes remaining unnecessary data in the cache and decreases the performance.

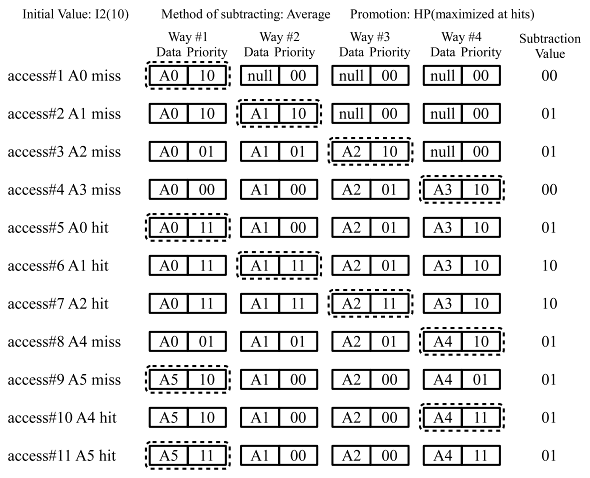
Initial Value: I2(10)
Method of subtracting: Average
Promotion: HP(maximized at hits)
Way #1 Data Priority
Way #2 Data Priority
Way #3 Data Priority
Way #4 Data Priority
Subtraction Value
access#1 A0 miss | A0 10 | null 00 | null 00 | null 00 | 00
access#2 A1 miss | A0 10 | A1 10 | null 00 | null 00 | 01
access#3 A2 miss | A0 01 | A1 01 | A2 10 | null 00 | 01
access#4 A3 miss | A0 00 | A1 00 | A2 01 | A3 10 | 00
access#5 A0 hit | A0 11 | A1 00 | A2 01 | A3 10 | 01
access#6 A1 hit | A0 11 | A1 11 | A2 01 | A3 10 | 10
access#7 A2 hit | A0 11 | A1 11 | A2 11 | A3 10 | 10
access#8 A4 miss | A0 01 | A1 01 | A2 01 | A4 10 | 01
access#9 A5 miss | A5 10 | A1 00 | A2 00 | A4 01 | 01
access#10 A4 hit | A5 10 | A1 00 | A2 00 | A4 11 | 01
access#11 A5 hit | A5 11 | A1 00 | A2 00 | A4 11 | 01

Figure 1: Behavior of ADP

3. THE ADAPTIVE DEMOTION POLICY

3.1 Behavior of the Adaptive Demotion Policy

The RRIP policy subtracts the smallest priority of all the blocks from the priorities of existing blocks when cache misses occur. This causes the priorities of the existing blocks are minimized or unchanged at several applications, and degrades the performance. To avoid this problem, ADP uses an adaptive demotion policy, which is based on the average of the priority values of all the blocks in the accessed set.

ADP prepares priority values for each cache block. Regardless of the associativity, the 2-bit value is enough for storing the priority [3]. Figure 1 shows the behavior of ADP. Since caches should quickly send the data to computational resources when cache hits occur, the control logic at cache hits for updating priority value should be simple. The same as the RRIP policy, the priority value of a block is promoted by the promotion policy when cache hits occur in that block.

The behavior at a cache miss is as follows. First, the block that has the smallest priority in the accessed set is selected as the victim. If there are two or more blocks which have the smallest value in the accessed set, one block is selected by the selection policy. Next, the priority values of all the blocks in the accessed set are decreased by the subtract value, which is generated by the demotion policy. Finally, a new block coming from the lower-level memory hierarchy is stored, and the priority value of the block is set to the initial priority value.

The demotion policy affects the performance of ADP. The large subtraction value decreases the priority of the existing blocks in the cache, and increases the priority of the incoming block. The small subtraction value causes a reverse trend. Therefore, an adaptive demotion policy with considering the priority values of all the blocks in the set is needed to improve the cache performance.

One idea to adaptively generate the subtraction value is the averaging of the priority values of all the blocks in the set. This paper examines two demotion policies; the HOA policy and the AVE policy. The HOA policy takes the half of the average of the priority values in the accessed set, and sets it as the subtraction value. The HOA policy prevents the rapidly decreasing of the priority values. Besides, the

HOA policy decreases the priority values even if there is the block that has the minimum value, and does not decrease the priority value when the total priority value is low. The AVE policy uses the average of the priority values in the accessed set. Compared with the HOA policy, the AVE policy tends to rapidly decrease the priority values at a cache miss. Although the AVE policy causes the problem that is also caused by the RRIP policy when the existing blocks have the maximum priority values, other features are the same as the HOA policy. In Figure 1, the AVE policy is adopted. For example, after the access 2, the average of the priority values is 01, so the priority values are subtracted by 01 at the access 3. In the other case, after the access 7, the average of the priority values is 10, so the priority values are subtracted by 10 at the access 8.

3.2 The Other Aspects of ADP

The performance of ADP will be affected by its insertion policy, promotion policy, and selection policy, so the tradeoff among these policies should be examined.

3.2.1 Insertion policy

The insertion policy decides the priority of the incoming block. As ADP uses a 2-bit value for each block, the candidates of the initial priority value are of 00, 01, 10, and 11. In Figure 1, the initial priority value is of 10, and the priority value of the incoming block is set to that value. (E.g., at the access 1.) The small initial priority provides scan-resistant. However, if the initial priority is smaller than the smallest priority in the set, and the new block is not re-referenced before a cache miss occurs, the block will disappear. On the other hand, if the initial is too high, the new block remains in the cache for a while even though the block is never re-referenced. Based on the above, 01 and 10 are selected as the initial priority value in the experiments, and these values are illustrated as I1 and I2, respectively.

3.2.2 Promotion policy

At a cache hit, the priority of the hit block is promoted. There are two promotion policies; Hit Promotion (HP) and Frequency Promotion (FP). The HP policy gives the highest priority to the hit block, and the FP policy increases the priority by one. Figure 1 shows the behavior when the HP policy is adopted, and the priority of the hit block is maximized. (E.g., at the access 5.) Because the HP policy gives the highest priority to the hit block, the HP policy enlarges the importance of the hit compared with the FP policy. As the FP policy increases the priority by one, that provides scan-resistant when the initial priority is small. Although the HP policy achieves higher performance than the FP policy in the RRIP policy [3], the FP policy is suitable for several benchmarks. In the experiments, the effects of both promotion policies on ADP are examined.

3.2.3 Selection policy

If there are two or more blocks which have the smallest value in the accessed set, one block is selected by the following selection policies. One policy is the Left side Selection (LS) policy, and another is the Random Selection (RS) policy. The LS policy selects the block that has the smallest way number. The RS policy selects one block at random. Figure 1 shows the behavior when the LS policy is adopted. For example, after the access 8, three blocks have the same

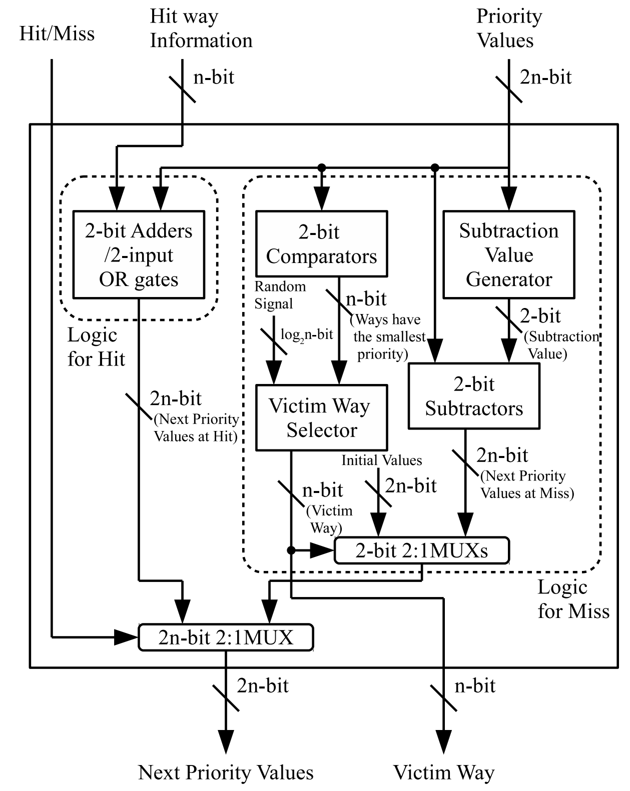

Figure 2: Structure of the priority controller.

number that is smallest in the set, so the way 1 is selected as the victim at the access 9. The LS policy provides the tolerance for scan access patterns and thrashing access patterns, but the small initial priority causes the initial priority problem discussed above. The RS policy will be able to achieve the stable performance with any initial priority, but the selection logic is complex compared with that of the LS policy.

3.3 Design of the Priority Controller of ADP

To implement ADP, the priority controller that can dynamically change the configuration of policies is needed. Figure 2 shows the structure of the priority controller that decides the victim and updates the priority values. When the associativity of the cache is n, the input signals of the priority controller are the signal for judging hit or miss (1-bit), the signal for identifying the way in which a cache hit occurs (n-bit), and the priority values from all the blocks in the accessed set ($2n$-bit). The output signals are the signal that shows the way where the block is victimized (n-bit), and the updated priority values ($2n$-bit).

When cache hits occur, the promotion policy promotes the priority value of the block. When the FP policy is adopted, 2-bit saturating adders are used. If the HP policy is adopted, 2-input OR gates are used for maximizing the priority value.

The behavior of the priority controller when cache misses occur is as follows. First, the victim selection signal is generated by the 2-bit comparators and the victim way selector. The 2-bit comparators generate the n-bit signal that indicates the way in which the block has the smallest priority value. The victim way selector generates the n-bit signal, which indicates the way with the victimized block. In the case of the LS policy, the lowest asserted bit is selected from the output of the comparators. In the case of the RS policy, the $\log_2 n$-bit random signal is used to select one asserted bit from the output of the comparators.

In parallel to the above process, the subtraction value is generated. The subtraction value generator adds the priority of the blocks and, from the result, acquires the subtraction value. It is simple to obtain the subtraction values from the result. The average value is obtained by taking the most significant two bits of the result, and the half of the averaged value is taken from the most significant one bit of the result.

Next, the priority values of all the blocks are subtracted by the generated subtraction value. In this change in the priority value, 2-bit saturating subtractors are used. Finally, n 2-bit 2:1 multiplexers are used to set the initial priority value of the new block, and the $2n$-bit updated priority values are generated.

The hardware resources for implementing the priority controller of ADP is lower than that of the LRU policy because the LRU policy should keep the order of references in the set. To update the order at every cache access, the control logic for a cache hit cannot be designed faster by pipelining. Therefore, it is difficult to employ the LRU policy in the high-associativity caches. On the contrary, the control logic of ADP has a lower complexity because, when a cache hit occurs, the logic needs a cache hit signal and the priority value of the hit block. Besides, the hardware resources for storing the priority values are the same as that of the RRIP policy, and smaller than that of the LRU policy.

4. EVALUATION OF ADP

To evaluate the critical path delay and the required hardware resources, ADP is implemented on the hardware. In addition, the performance of ADP at all levels of the memory hierarchy is evaluated.

4.1 Implementation of ADP

The $65nm$ CMOS standard cell library is used for our implementation. Synopsys Design Compiler is used for logic synthesis, and Synopsys NanoTime simulator is used to evaluate the critical path delay under a 0.55V supply voltage. To measure the critical path delay, the control logic for a hit and that for a miss are designed individually.

Table 1 shows the critical path delay and the number of transistors for each policy. The critical path delay of the LRU policy tends to increase as the associativity increases. Because the control logic for a cache hit cannot be pipelined, it is difficult to implement the LRU policy for high-associativity caches. On the other hand, the critical path delay of the control logic for a hit of ADP is not affected by the associativity. Although the critical path delay of the control logic for a miss is longer than that of the control logic for a hit, the cycle time of the processor is not affected because the control logic for a miss can be pipelined. As compared with the LS policy, the RS policy requires more transistors, and the critical path delay is longer. That is because the RS policy should select one block from the blocks that have the smallest priority value by the log_2n-bit random signal.

4.2 Results and Discussion

Table 1: Critical path delay and the number of transistors.

policy	assoc.	delay(ns)	transistors
LRU	8	3.08	2772
	16	4.96	11820
ADP(Hit_FP)	8	0.275	304
	16	0.275	608
ADP(Hit_HP)	8	0.094	96
	16	0.094	192
ADP(Miss_LS)	8	2.41	1831
	16	3.49	5310
ADP(Miss_RS)	8	2.44	2540
	16	3.82	6799

The simulation experiments are performed to show the performance of ADP and the effects of the insertion, promotion, selection policies on the performance. For these experiments, the simulator including ADP is developed based on the gem5 simulator system with Alpha 21264 instruction set [5]. The benchmarks included in the SPEC CPU2006 [6] are examined as workloads for the experiments. For each benchmark, a representative phase of one billion instructions is extracted from the full execution by SimPoints [7]. The extracted phase is used for the evaluations. In each simulation, the first 50M instructions are used for warming up, and the following 950M instructions are for obtaining the simulation results. Table 2 shows cache parameters used in the simulation. In the evaluations, the performances of the LRU policy and the Static-RRIP (SRRIP) policy are also evaluated. The SRRIP policy adopts the HP policy as the promotion policy, and the initial priority is set to I1 and I2. In the experiments, the configurations of ADP are illustrated like as HOA_I1_HP_LS that means the HOA policy with the initial priority value of I1, and the HP and LS policies are adopted.

Figures 3 and 4 show the geometric mean of MPKI normalized by that of LRU policy in all the benchmarks when the HOA and AVE policies are adopted, respectively. For the L1 instruction cache, AVE_I2_HP_LS achieves the highest MPKI reduction rate, and the reduction rate is 5.0%. HOA_I2_HP_LS achieves the highest MPKI reduction rate for the L1 data and L2 caches, and the reduction rates are 0.69% and 2.1%, respectively. For the L3 cache, HOA_I2_HP _RS achieves the highest MPKI reduction rate, but the performance difference of between that and HOA_I2_HP_LS is small. The MPKI reduction rates of ADP with these configurations are 1.6% and 1.5%, respectively. From these results, it is observed that the AVE policy is suitable for the L1 instruction cache, and the HOA policy is suitable for the other caches.

Except for the L1 instruction cache, ADP with the HOA policy achieves higher MPKI reduction rate compared with the SRRIP policy. Although ADP with the AVE policy reduces the MPKI at the L1 instruction cache compared with the LRU and SRRIP policies, ADP with the AVE policy increases MPKI at the other caches. Because the temporal locality is too high at the L1 instruction cache, the HOA policy cannot reduce enough the priority values of the existing blocks to prevent the new block from being evicted. The AVE policy can reduce enough the priority values of the existing blocks, and help evicting unnecessary data. At the

Table 2: Cache parameters.

cache	size	assoc.	load-to-use-latency
L1I	16KB	16	1
L1D	16KB	16	1
L2	256KB	8	10
L3	2MB	16	25
Main Memory	-	-	175

other-level caches, the AVE policy reduces too much priority values, so it increases the probability of evicting useful data.

Figure 5 shows the MPKI rates of ADP with various configuration compared with the LRU policy at the L3 cache. As shown in Figure 5, several benchmarks require the small initial priority to improve the performance of ADP. It is considered that these benchmarks include scan accesses, and the default configuration, especially the initial priority value I2 is not suitable for these benchmarks. To improve the performance of ADP, the configuration should be changed dynamically. The LS policy with I1 causes a significant performance degradation in several benchmarks. Therefore, the RS policy is suitable for I1. Among the configurations with I1 and the LS policy, the FP policy tends to achieve the higher performance compared with the HP policy. Therefore, HOA_I1_FP_RS is suitable for the applications that HOA_I2_HP_LS is not suitable.

Figures 6 and 7 show the MPKI rates of the SRRIP policy and ADP compared with the LRU policy at L2 cache and L3 cache, respectively. The configuration of ADP is HOA_I2_HP_LS. Figures 6 and 7 also show the best performance of HOA_I2_HP_LS and another configuration, HOA_I1_FP_RS. This means the ideal performance of switching the two ADP configurations by the set-dueling [2]. Here, HOA_I1_FP_RS is selected because the configuration is suitable for providing scan-resistant and can achieve the stable performance among the scan-resistant configurations as shown in Figure 5. As shown in Figures 6 and 7, ADP achieves stable MPKI reduction compared with the SRRIP policy. The SRRIP policy causes the significant increase of the MPKI at several benchmarks because the subtraction value is not adaptively controlled as with ADP. Although ADP achieves higher MPKI reduction rate compared with the LRU and SRRIP policies, the MPKI reduction rate obtained by ADP is affected by the behavior of the benchmarks. By selecting an appropriate configuration, ADP achieves much higher performance.

To evaluate the IPC improvement compared with the other policies, ADP is adopted in all the caches. In this experiment, Static-ADP (S-ADP) and Dynamic-ADP (D-ADP) are evaluated. D-ADP dynamically selects the configuration by the set-dueling [2]. One configuration is the same as that of S-ADP, and another configuration is HOA_I1_FP_RS. In the experiments, 32-entry SDMs and 11-bit PSEL counter are used [2]. S-ADP policy uses the following static configuration. At the L1 instruction cache, the configuration is AVE_I2_HP_LS. At the other caches, HOA_I2_HP_LS is adopted. The SRRIP policy is adopted in all the caches with I1_HP.

Figure 8 shows the IPC improvement compared with the LRU policy. The SRRIP, S-ADP, and D-ADP policies achieve a 0.55%, a 0.73%, and a 1.0% IPC improvement by the ge-

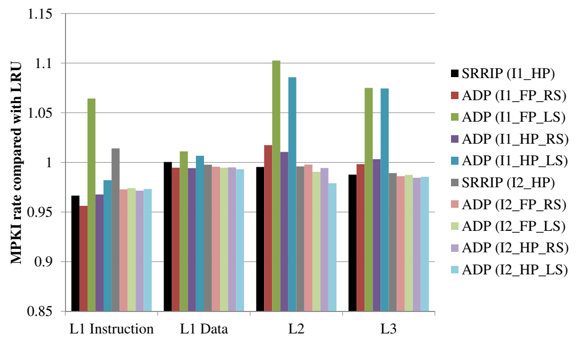

Figure 3: MPKI rate compared with the LRU policy (HOA).

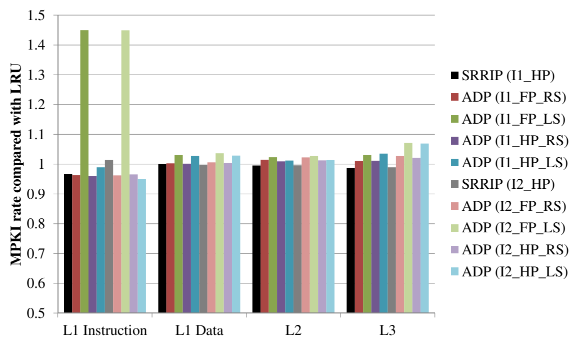

Figure 4: MPKI rate compared with the LRU policy (AVE).

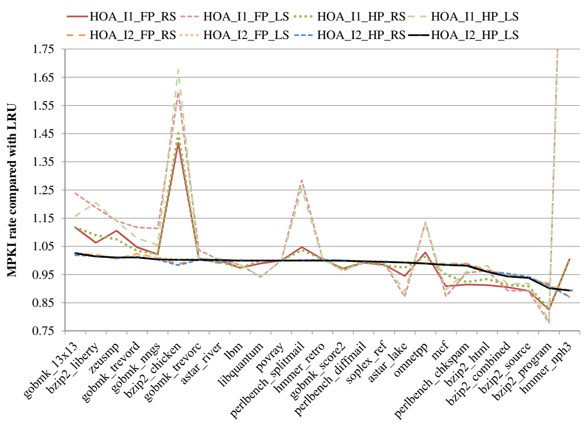

Figure 5: the MPKI rate of ADP compared with the LRU policy (L3 cache, with various configurations).

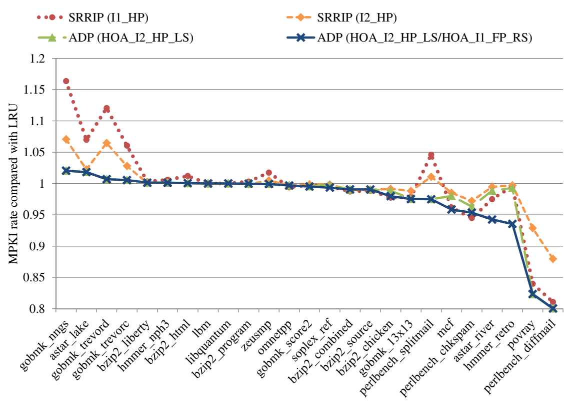

Figure 6: MPKI rate compared with the LRU policy (L2 cache).

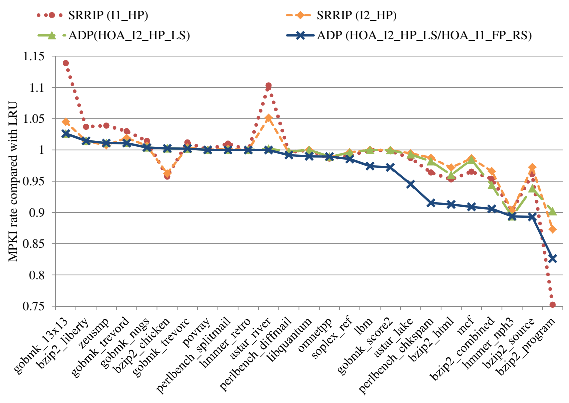

Figure 7: MPKI rate compared with the LRU policy (L3 cache).

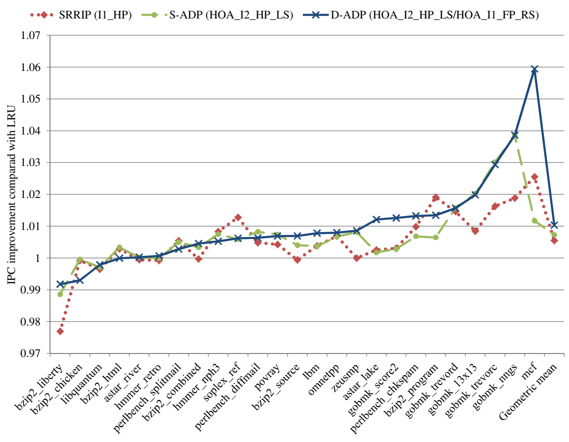

Figure 8: IPC improvement on all the benchmarks.

ometric mean of all the benchmarks, respectively.

As compared with S-ADP, D-ADP achieves higher IPC improvement. However, the performance of D-ADP is lower than that of S-ADP at several benchmarks even though D-ADP selects the suitable configuration from the two configurations by the set-dueling. This is because one configuration is not suitable for the benchmark, and it increases the overheads of the set-dueling. To improve the performance of D-ADP, the configuration should be changed by the behavior of the application. ADP can dynamically change the demotion, insertion, promotion and selection policies. In future work, a dynamic method for optimization of these policies will be proposed and evaluated.

5. CONCLUSIONS

This paper proposes a high-performance cache replacement policy called ADP, which is suitable for implementing high-associativity caches. The proposed policy can suppress the hardware overheads, and does not affect the processor cycle time even though the associativity is high. Evaluation results show ADP achieves MPKI reduction in all levels of the memory hierarchy compared with the LRU and RRIP policies.

6. ACKNOWLEDGMENTS

This work was supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc.

7. REFERENCES

[1] T. S. B. Sudarshan et al., "Highly efficient LRU implementations for high associativity cache memory," in Proceedings of 12th IEEE International Conference on Advanced Computing and Communications, 2004.

[2] M. K. Qureshi et al., "Adaptive Insertion Policies for High Performance Caching," in Proceedings of International Symposium on Computer Architecture (ISCA 2007), 2007

[3] A. Jaleel et al., "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in Proceedings of International Symposium on Computer Architecture (ISCA 2010), pp. 60–71, 2010

[4] H. Ghasemzadeh, et al., "Modified pseudo LRU replacement algorithm," in Proceedings of 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, 2006.

[5] N. Binkert et al., "The gem5 Simulator," ACM SIGARCH Computer Architecture News, vol. 39, issue 2, pp. 1–7, 2011

[6] J. L. Henning, "SPEC CPU2006 benchmark descriptions," ACM SIGARCH Computer Architecture News, vol. 34, issue 4, pp. 1–17, 2006

[7] G. Harmerly et al., "Simpoint 3.0: Faster and more flexible program phase analysis," Journal of Instruction Level Parallelism, vol.7, no. 4, pp.1–28, 2005