IEICE TRANS. INF. & SYST., VOL.E95–D, NO.12 DECEMBER 2012

PAPER Special Section on Parallel and Distributed Computing and Networking

Using Cacheline Reuse Characteristics for Prefetcher Throttling

Hidetsugu IRIE[†a)], **Takefumi MIYOSHI**[†], *Members*, **Goki HONJO**[††], **Kei HIRAKI**[††], *Nonmembers*, *and* **Tsutomu YOSHINAGA**[†], *Member*

SUMMARY One of the significant issues of processor architecture is to overcome memory latency. Prefetching can greatly improve cache performance, but it has the drawback of cache pollution, unless its aggressiveness is properly set. Several techniques that have been proposed for prefetcher throttling use accuracy as a metric, but their robustness were not sufficient because of the variations in programs' working set sizes and cache capacities. In this study, we revisit prefetcher throttling from the viewpoint of data lifetime. Exploiting the characteristics of cache line reuse, we propose Cache-Convection-Control-based Prefetch Optimization Plus (CCCPO+), which enhances the feedback algorithm of our previous CCCPO. Evaluation results showed that this novel approach achieved a 30% improvement over no prefetching in the geometric mean of the SPEC CPU 2006 benchmark suite with 256 KB LLC, 1.8% over the latest prefetcher throttling, and 0.5% over our previous CCCPO. Moreover, it showed superior stability compared to related works, while lowering the hardware cost.

key words: *microarchitecture, cache, prefetch*

1. Introduction

One of the most important issues related to microarchitecture is the design of a memory hierarchy to conceal memory latency and to make the functional units busy [1]. Currently, microprocessors have a large last-level cache (LLC), the size of which is generally several megabytes. Off-chip memory bandwidth has been relatively increased because of the introduction of on-chip memory controllers. Moreover, several 3D manufacturing techniques are expected to significantly extend memory bandwidth in the near future. The current high performance execution of multicore processors is supported by such powerful features of cache, large capacity and bandwidth.

However, LLC is generally shared by multiple cores, and therefore, the available size for each process will vary at execution time. Besides, several studies have reported that the access behavior for LLC has little locality because locality is absorbed in the higher level cache [2], [3]. This means that a large capacity may not always contribute to the cache hit rate. Currently, microprocessors still suffer from huge off-chip memory latency. For each LLC miss, sequential performance will be severely degraded. Consequently, hardware prefetching [4], [5] is a natural approach to deal with such LLC trends, and have been widely adopted by commercially-produced processors [6]. However, the introduction of prefetching also introduces several complicated side effects.

For today's large LLC, aggressive prefetching is often effective [7], [8] because there are plenty of cache lines. Moreover, because the latency is so long, prefetching must start faster, even in the exchange of prefetch accuracy. To make the prefetcher aggressive, prediction becomes deep (to prefetch n times ahead of address) and wide (to prefetch multiple candidates at once or prefetch by region). The simpler prediction algorithms are also effective for aggressiveness by reducing the learning time and sending prefetch requests earlier. It often happens that the simple prediction algorithms show greater performance rather than the complicated Markov-based prefetch algorithms [9].

However, the proper aggressiveness of a prefetcher depends on the application behavior and the cache capacity. Aggressive prefetching is powerful, but it generates a large amount of prefetch access in exchange of accuracy. It may cause cache pollution, where useful cache lines are swept away by large amount of useless prefetched lines. Useless prefetches also abuse memory bandwidth. Thus, recent prefetch techniques generally involve mechanisms that dynamically optimize the amount of prefetching. However, in the existing adaptive prefetching, the accuracy of address prediction is used as a metric because the current data in the cache should be valuable [10]–[12]. This is not always true in LLC because aggressive depths and widths of the effective prefetching are gained in exchange of accuracy. A new approach for evaluating the properness of prefetching is required.

This study proposes a novel prefetcher throttling technique called Cache-Convection-Control-based Prefetch-Optimization Plus (CCCPO+), which exploits cache line reuse. Unlike existing techniques, it does not trace the prefetch usefulness, so the hardware is simple. Figure 1 illustrates our goal. The solid line shows the typical capacity vs. performance behavior of caches. Discussions regarding this behavior is important because the capacity of LLC will vary dynamically by multicore execution and power saving. The cache performance for all possible capacities must be examined. In the solid line, performance first increases with capacity, and it then saturates at a given performance, which depends on the program characteristics. Multiple plateaus will appear, and the line has a step-like shape in some cases

Manuscript received January 5, 2012.
Manuscript revised May 20, 2012.

[†]The authors are with The University of Electro-Communications, Chofu-shi, 182–8585 Japan.
[††]The authors are with The University of Tokyo, Tokyo, 113–8656 Japan.
a) E-mail: irie@is.uec.ac.jp

DOI: 10.1587/transinf.E95.D.2928

because of program's control or data structures. The behavior will shift to the thin line when the prefetcher is introduced. Prefetching can reduce compulsory misses, but it also requires extra capacities. Thus, the line is stretched to the upper-right. Our goal is to stretch this line to the upper-left, thereby realizing higher performance and saturation at lower capacity.

This study is organized as follows. Section 2 discusses the control techniques for prefetcher aggressiveness, while Sect. 3 provides observations about data convection in a cache, and shows that this behavior can guide the optimal control of prefetcher aggressiveness. Section 4 describes our throttling mechanism. Section 5 presents the evaluation environment, while Sect. 6 shows the results. Section 7 concludes this study.

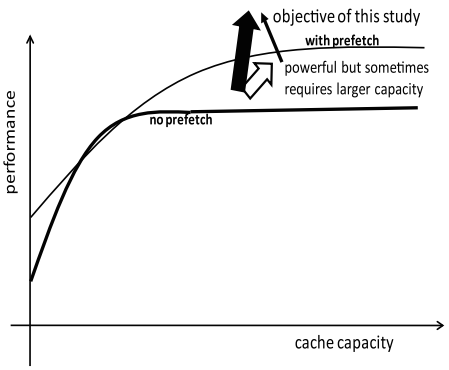

Fig. 1 Enhancement of prefetcher efficiency.

2. Prefetcher Throttling

2.1 The Effect of Prefetcher Throttling

Figure 2 shows an example of the effect of prefetcher aggressiveness. The relative IPC performance of various prefetcher settings is shown for each SPEC CPU 2006 benchmark program. Each program has six bars that indicate the performance of lower aggressiveness to higher aggressiveness from left to right, respectively. In this graph, the cache capacity is 256 KB, which is likely to be a portion of the multicore LLC for a process. Processor and prefetcher parameters are the same as in subsequent evaluations.

The geometric mean gradually increases with increasing aggressiveness. However, each benchmark program has its own optimum aggressiveness, which implies that each program phase has its own optimum prefetcher settings. The effect of the aggressiveness selection ranges from the lower limit −30% to the upper limit +250%, which is as significant as that of the address prediction algorithm.

2.2 Adaptive Prefetching Techniques

Related studies of adaptive prefetching are roughly divided into two approaches, namely, per access control and per period control. The first group determines whether proceeds the prefetching access or not for each predicted address. Zhuang et al. [10] proposed a technique that estimates the accuracy of each address prediction. The history regarding whether previous prefetched lines were actually accessed are stored in a table of 2-bit saturation counters. This ta-

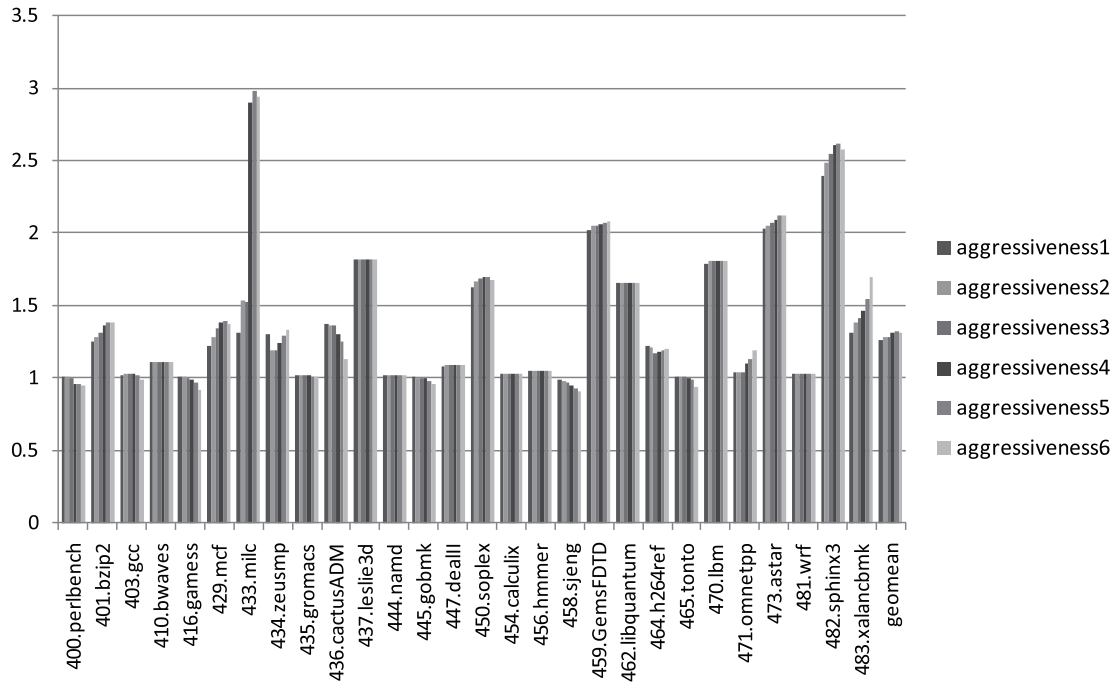

Fig. 2 The effect of prefetcher throttling.

ble is accessed for each predicted address to predict the usefulness of that prefetch using the load instruction's PC or predicted address as a key. The prefetch access is canceled when it is predicted as being useless. This approach requires additional tables to maintain the past prefetch histories. Liu et al. [13] exploited the dead block detection to avoid the negative-effect of prefetching. Prefetching is performed only when the new dead block is detected.

The second group observes prefetcher performance for a period whose length typically involves more than one thousand LLC misses. The estimated performance is used to adjust the prefetcher aggressiveness for the next period. The number of prefetches is controlled by periodically changing the parameters of the prefetcher, mainly depth, as opposed to cancelling a certain prefetch access. This technique is generally called "prefetcher throttling" and is suitable for controlling prefetchers that generate many prefetches at once. Hur et al. [11] proposed an adaptive stream prefetching that statistically estimates the most frequent stream length, and cancels prefetch accesses that exceed this length. This technique improves the prefetch efficiency by canceling the prefetch overruns that occur at the end of each stream. Srinath et al. [12] proposed Feedback Directed Prefetching, which estimates prefetcher accuracy, lateness, and pollution of every period, and compares them to the predetermined thresholds. The aggressiveness of the next period is updated according to the comparison. Using three metrics, it achieves detailed adaptive controlling that suppress the low accuracy prefetches, suppresses pollution (but without accurate prefetches), and accelerates accurate but late prefetches. However, the technique does not include a provision to promote low accuracy, but effective prefetches. Also, it requires a history table with thousands of bits (implemented in a bloom filter) for the detection of pollution.

In one of the latest related studies, Ebrahimi et al. [14] proposed Coordinated Prefetcher Throttling, which is similar to Feedback Directed Prefetching but uses prefetcher accuracy and coverage as metrics. One of the features of this technique is that it introduces the coverage as a metric. In LLC, a dependence on only the accuracy may reduce the chances of exploiting aggressive prefetches. The introduction of coverage encourages the exploitation of the latest aggressive prefetchers that gain coverage in exchange for accuracy. Ebrahimi et al. also proposed a prefetcher throttling technique for multicore processors with Feedback Directed Prefetching [15].

2.3 Limitation of Prefetch Usefulness Based Approach

Existing throttling techniques increase the aggressiveness of prefetchers when either the accuracy or coverage is greater than the fixed threshold, and decrease the aggressiveness when they are less than the threshold. Thus, the higher threshold value implies better braking, while a lower threshold value implies larger acceleration. However, it is hard to determine the threshold value that can properly control various applications, especially for the following cases: i) although the prefetch accuracy is high, useful lines are pushed out from the cache when the cache has no vacancies, ii) although the prefetch accuracy is low, aggressive prefetching improves performance when the cache has vacancies, iii) although both the accuracy and coverage are low, prefetching is still effective because of the poor locality of the application access pattern. These examples indicate that we must estimate the cache condition to properly throttle the LLC prefetching.

3. Cache Convection: Novel Metric for Prefetcher Throttling

3.1 Optimum Prefetcher Throttling

When a certain cache line is transferred to the cache earlier by prefetching, the swapped line is also evicted earlier from the cache. Moreover, if several lines are prefetched to the cache at the same time, several lines are evicted at the same time. The gain realized in prefetching is the difference between the values of the prefetched lines and the evicted lines. However, in general, previous techniques assume that the evicted lines have the same value, while they deliberately estimate the value of the prefetched line, for example, using accuracy statistics.

Instead, we focus on the cache contents. The basic idea behind our throttling is that useless prefetches are not actually useless. Conversely, they are necessary to encourage deep and wide prefetching, which is effective for poor locality access patterns. Our goal is to prefetch aggressively as far as no line is swept away before its last use. For example, if the locality of the program is low, aggressive prefetching is required even if the accuracy is low, because the cache may contain many dead blocks. However, in general, dead block prediction is costly. Srinath et al. tried to predict the usefulness of the evicted lines by storing the eviction history in a bloom filter [12]. However, the filter requires a table containing several thousand bits. Therefore, we propose another statistical technique for estimating the cache margin.

3.2 Line Reuse and Prefetcher Throttling

For example, let us assume a program that accesses the same cache line four times on an average. When the prefetcher aggressiveness is too low (Fig. 3 (a)), the first access may miss because the prefetch access began too late or was not performed. The subsequent three accesses are hit within the same lifetime. Then, the line will be evicted after the last use. Consider that the prefetcher aggressiveness is gradually increased. For certain degrees of aggressiveness, the prefetch for the first access meets the time (Fig. 3 (b)). Thus, four accesses are hits within the same lifetime. The reuse number saturates at this point because of the program characteristics. Here, the aggressiveness is the optimum. If the aggressiveness is further increased, pollution begins because a lot of data is loaded to the cache within the time in which

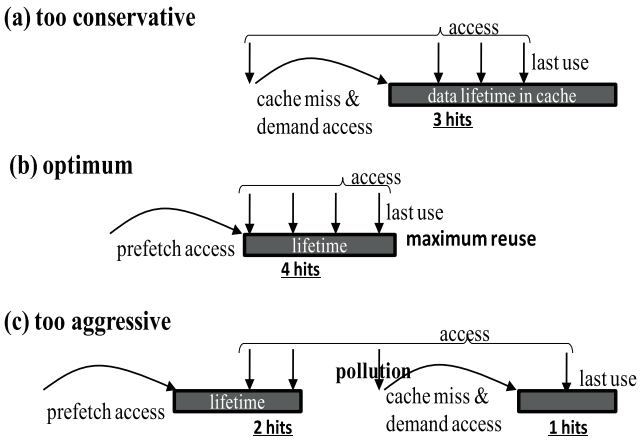

Fig. 3 Prefetcher aggressiveness and line reuse.

the line is evicted before the last use. This situation results in a significant degradation of reuse numbers for each cache line (Fig. 3 (b)). Consequently, we can maintain the prefetcher aggressiveness optimum to maximize the cache line reuse. The number of reuses per line is easier to estimate than the dead block prediction.

3.3 Cache Convection

We statistically estimate the reuse characteristics of each period. The basic idea is to divide the total cache hit count during a line's lifetime by the number of cache lines to be accessed. Using the following equation, this can be approximated by counting the events during a period which are triggered by a certain number of evictions of cache lines.

$$CC = \frac{hitcount}{(\frac{accessed\_evicted}{evicted}) \times N} \quad (1)$$

where *hitcount* is the number of cache hits, *accessed_evicted* is the number of evicted lines, which is accessed in the cache at least once, *evicted* is the number of evicted lines, and N is the number of cache entries. We define this value as "*Cache Convection*" or *CC* as the image of the cache line convection between the MRU side and the LRU side. The denominator of Eq. (1) approximates the number of cache lines that are to be accessed, and thus, *CC* indicates the hit count per demand line of that period. Note that the cache hits at a live block and not at all lines in the cache. Cache lines will be evicted without gaining access when at the demand misses of streaming access patterns, or at useless prefetches. A direct estimation of the reuse is possible by observing the access count of each evicted line. However, we instead use the proposed *CC* because the direct way requires a hit counter for each cache line, which uses a significant percentage of the entire cache capacity.

CC will show the best value when the proper aggressiveness is selected for the program phase. To confirm this, we examined the *CC* behavior of SPEC CPU 2006 benchmark suites. Figure 4 and Fig. 5 show typical examples of the relationship between *CC* and prefetcher aggressiveness.

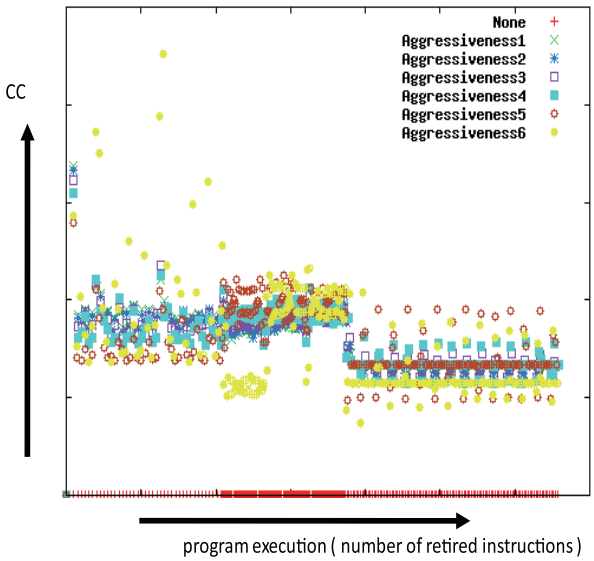

Fig. 4 CC and prefetcher aggressiveness (437.leslie3d).

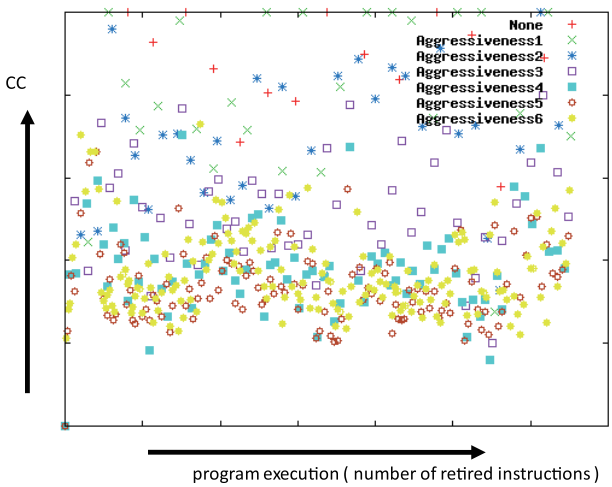

Fig. 5 CC and prefetcher aggressiveness (458.sjeng).

The processor and prefetcher parameters are the same as in Fig. 2. *CC* is calculated at every 2,048 evictions, and is plotted on the graph. The Y axis represents *CC*, and the X axis indicates the program's execution point by the number of retired instructions. *CC* behaviors with different levels of aggressiveness are overlapped. In Fig. 4, for the behavior of 437.leslie3d, *CC* is obviously different for the cases with no prefetching when compared to the cases with a different aggressiveness. With no prefetching, *CC* shows a continuously low value while the others show higher values. This behavior indicates that any aggressiveness related to prefetching is effective, and corresponds to the IPC behavior of Fig. 2. Moreover, this graph shows that a phase change has a direct impact on the *CC* behavior.Conversely, in Fig. 5, for the behavior of 458.sjeng, the values are distributed over a relatively wide range, and the *CC* becomes low when the

aggressiveness is increased. This indicates that the lower aggressiveness achieves better performance, and corresponds to Fig. 2. We examined the *CC* behavior for all programs in the suite and confirmed that the prefetcher is optimized by choosing an aggressiveness that results in a higher CC. Thus, *CC* effectively determines the prefetcher aggressiveness.

4. Throttling Mechanism

4.1 Improvement over our previous CCCPO

Previously, we proposed CCCPO, which exploits CC to determine the prefetcher aggressiveness [16], [17]. In this technique, CC is calculated periodically, and the prefetcher aggressiveness is set to indicate higher CC. The feedback algorithm was simple; in that, it decreases the prefetcher aggressiveness if the CC is degraded form the last period, and vice versa. To mitigate the sampling noise, several hysteresis factors were introduced. CCCPO showed stable performance with simple hardware, but it tends to be too aggressive in several programs because of its simple feedback algorithm. For example, once the prefetcher becomes too aggressive, it is difficult for the above algorithm to decrease the aggressiveness. To achieve the best performance, we revise it to CCCPO+ with more intelligent control, which searches for the best aggressiveness while maintaining simple hardware and rapid response.

4.2 Hardware Outline

Figure 6 shows the outline of our CCCPO+. The cache behavior is counted by several counters, and *CC* is calculated at the end of every period from the counter values. The throttling algorithm determines the aggressiveness for the next period by using the *CC* values of recent periods, which are stored in the scoreboard. The prefetcher setting is updated at the end of every period. The proposed technique only requires several registers to be implemented, and it does not need any large tables for feedback.

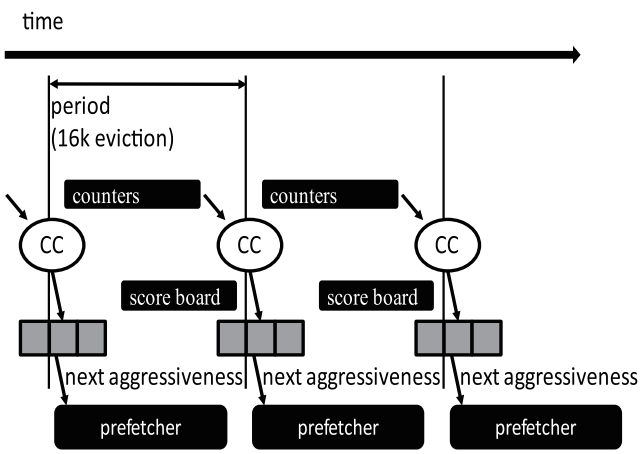

Fig. 6 Outline of the proposed technique.

To count the number of cache hits and other events, the following three counters are added:

- *num_hits*, which counts the number of cache hits. The counter value is used to calculate CC.
- *num_evictions_accessed*, which counts the number of lines that have been pushed out from the cache after being accessed at least once. The counter value is used to calculate CC.
- *num_evictions*, which counts the number of evicted lines. The throttling algorithm is triggered when this counter becomes a predetermined threshold, and the new period starts with a new prefetcher aggressiveness.

These counters are reset to zero at the start of every period, and are incremented at each cache access or line eviction. In addition, the following registers are added to store the control values:

- *reg_aggressiveness*, which indicates the current aggressiveness of the prefetcher.
- *scoreboard*, which stores the *CC* values of recent periods. As described later, at most three *CC* values are stored to scan for the best aggressiveness.

In addition to these unique counters and registers, the proposed technique requires 1-bit "access bit" on each line of the cache tag table. The "access bit" is initialized to zero when new data is transferred to the corresponding cache line, and is set to one when any accesses are performed on that line. Prefetching or cache replacement techniques often require such bits, and in those cases, a new budget for the "access bit" is not required. Moreover, a divider is used to calculate *CC*, but this division is not critical for latency, accuracy, or conflicts; so many approaches, such as exploiting the divider in the functional units or approximating with multiple saturation counters are available for reducing the divider's budget. In this study, an additional divider is used for the performance evaluation, but the difference in performance is negligible. CCCPO+ is able to independently introduce any cache hierarchy. The counters count only events of the corresponding cache.

4.3 Feedback Algorithm

The aggressiveness that makes *CC* highest for the program phase is unknown before the execution. The feedback algorithm searches for the best aggressiveness for each phase through the program execution. From Fig. 4 and 5, it can be seen that *CC* is somehow distributed, but makes clusters for each value of aggressiveness. It can also be seen that phases comprise many periods. Therefore, we can approach the best aggressiveness by comparing the result of short executions (periods). There are also enough periods to gradually update the aggressiveness to an optimum value.

The feedback algorithm has two modes, the scan mode and the hold mode. First, it starts with the scan mode, which consists of three periods. The aggressiveness is set to -1, ± 0, and $+1$ from the current aggressiveness, and each *CC*

value is stored to the scoreboard. At the end of the scan mode, the best of three is selected as the current aggressiveness, and the mode shifts to hold mode. An exception occurs if the value of aggressiveness +1 > previous aggressiveness > aggressiveness −1, in which case the scan mode is restarted with an increased current aggressiveness.

In hold mode, the current aggressiveness is held unless a phase change is detected. The algorithm detects phase changes when *CC* changes significantly. To distinguish the phase change from sampling noises, we introduce accumulation and hysteresis. First, the calculated *CC* is accumulated as follows:

$$accumulatedCC = \frac{previousCC}{2} + \frac{newCC}{2} \quad (2)$$

Then, *accumulatedCC* is compared to the value of the *CC* when this hold mode was started. If the changing ratio exceeds a certain threshold, the algorithm again shifts to scan mode. The hold mode also ends at certain consecutive periods to avoid local optimal.

5. Evaluation Method

5.1 Baseline Processor

A performance evaluation is done using the cycle accurate simulator, which models out-of-order super scalar in detail, including the prefetcher and the proposed throttling techniques. The instruction set architecture and microprocessor parameters are shown in Table 1. In the evaluation, a single thread is executed in a single core. Prefetching is applied to LLC, that is, the L2 cache in this model, because the prefetcher's ability to hide the large memory access latency is focused on.

5.2 Prefetcher

The proposed throttling was applied to the sequential prefetcher (stream-based prefetcher) [5] for evaluation. The prefetch is performed on the sequential addresses of the missed address on a cache miss. The aggressiveness was set to seven levels (Table 2), which included the addition of two more aggressive levels "level5" and "level6" to the 5-level setting that was used by Ebrahimi et al. [14].

5.3 Throttling Parameters

In the evaluation, CCCPO+ was configured as follows. When we observe the throttling timeline in Sect. 6.1, feedback is performed for every 2048 cache line evictions, and for every 16 K evictions in a later performance evaluation. The hysteresis threshold of the hold mode is set to 20%.

5.4 Compared Models

The most recent related studies are implemented and evaluated for comparison purposes [12], [14], [17]. Feedback Directed Prefetching (FDP) and Coordinated Prefetcher Throttling (CPT) are both throttling techniques that are guided by prefetcher performance. Moreover, FDP has the feature that detects polluting prefetches, and CPT has the feature that can encourage aggressive prefetching. To verify the improvement in the feedback algorithm, we also performed a comparison with our previous CCCPO. From the preliminary evaluation, we set the threshold as Table 3.

Table 1 Parameters of baseline processor.

Instruction Set Architecture	Alpha AXP
Front-end	4 way, 7 cycle
Instruction Window	i64 entry, f32 entry
LSQ	32 entry
Functional Units	2 iALU, 1 iMUL/DIV, 2 LD/ST, 1 fpADD, 1 fpMUL/DIV/SQRT
L1 I-Cache	32 KB, LRU, 8 way, 64 B line, 1 cycle latency
L1 D-Cache	32 KB, LRU, 8 way, 64 B line, 1 cycle latency
L2 I/D-Cache	64 KB–2 MB, LRU, 16 way, 64 B line, 20 cycle latency
memory access	200 cycle

Table 2 Settings of prefetcher aggressiveness.

aggressiveness 0	no prefetching
aggressiveness 1	sequential depth 4
aggressiveness 2	sequential depth 8
aggressiveness 3	sequential depth 16
aggressiveness 4	sequential depth 32
aggressiveness 5	sequential depth 64
aggressiveness 6	sequential depth 128

Table 3 Threshold values for related works models.

Accuracy High (FDP)	0.75
Accuracy Low (FDP)	0.40
Lateness (FDP)	0.01
Pollution (FDP)	0.05
Accuracy (CPT)	0.60
Coverage (CPT)	0.20
Coverage (CPT)	0.20
Hysteresis (CCCPO)	0.25
Phase Detection (CCCPO)	0.05
Period (FDP, CPT, CCCPO)	2k eviction

5.5 Benchmarks for the Evaluation

We evaluated all of the benchmark programs of SPEC CPU 2006 (except for 453.povray because of simulator issues). Each program was compiled using gcc version 4. 2. 2 with the -O3 option. A cycle accurate execution of 100 million instructions after skipping 10 billion instructions from the program head was simulated.

6. Evaluation

6.1 Throttling Behavior

First, Fig. 7 and Fig. 8 show how our throttling works. A prefetcher aggressiveness of 437.leslie3d and 458.sjeng are

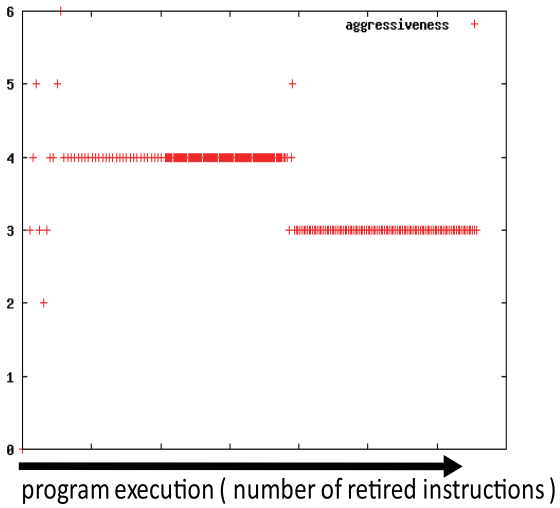

Fig. 7 Throttling timeline (437.leslie3d).

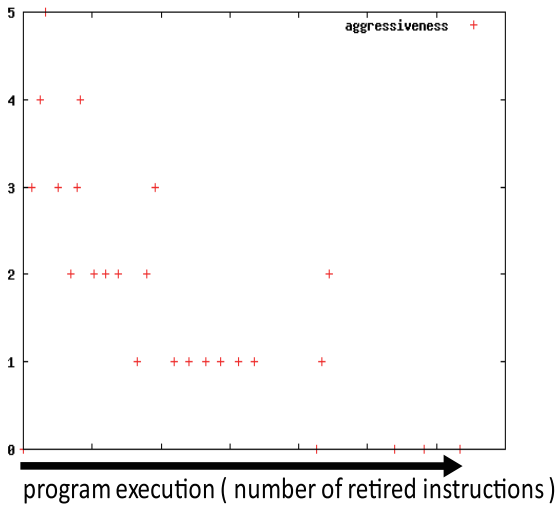

Fig. 8 Throttling timeline (458.sjeng).

shown in a manner similar to Fig. 4 and Fig. 5. In Fig. 7, the prefetcher aggressiveness is set to high and is held during each phase. In this example, the CCCPO+ promotes aggressive prefetching. In Fig. 8, CCCPO+ repeats the scan mode and gradually suppresses the aggressiveness. In this example, CCCPO+ avoids polluting prefetching. This shows that the CCCPO+ is available to correctly suppress the prefetcher even when the aggressiveness become too high, which is the case that could not be suppressed by CCCPO.

6.2 The Performance of CCCPO+

Figure 9 shows the execution performance of various degrees of prefetcher aggressiveness and CCCPO+ at a 256 KB LLC. Here, the bar that indicates the performance of CCCPO+ is added to the right-most side of each program bar. Aggressive prefetching is generally effective, and CCCPO+ was able to promote aggressive prefetching in many programs. Moreover, it properly suppressed prefetching for programs for which higher aggressiveness is not effective, such as 400.perlbench, 416.games, 436.cactusADM, and 458.sjeng. The performance is increased by 32.6% from the baseline in geometric mean, and also shows the best performance over fixed aggressiveness. Figure 10 shows the MPKI (miss per kilo instructions) for each level of aggressiveness. CCCPO+ almost achieves a reduction in MPKI to a minimum level, and the behavior of the graph is similar to that of the IPC graph.

6.3 Comparing Throttling Techniques

Next, Fig. 11 shows the performance of CCCPO+ and various prefetcher throttling techniques. Basically, all of the techniques have a similar performance, which means that the aggressiveness is properly determined by each technique. However, in several programs, such as 416.gamess, 429.mcf, 436.cactusADM, 471.omnetpp, and 483.xalancbmk, a difference is seen in the algorithms. Even

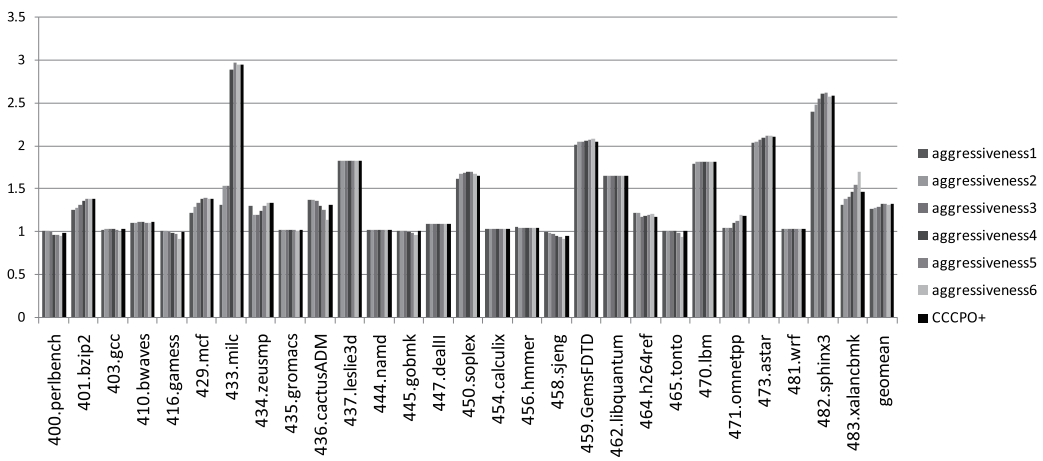

Fig. 9 Performance comparison between fixed aggressiveness and CCCPO+ (IPC).

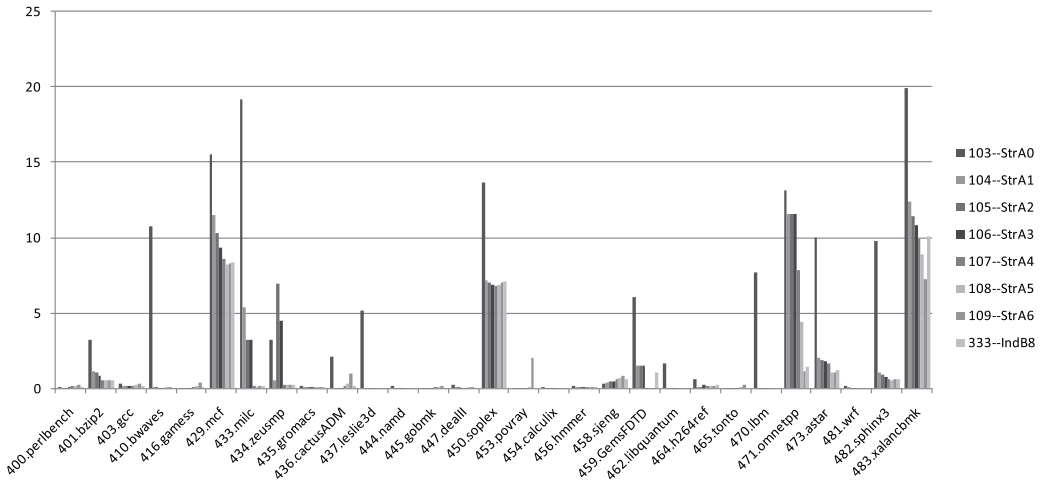

Fig. 10 Performance comparison between fixed aggressiveness and CCCPO+ (MPKI).

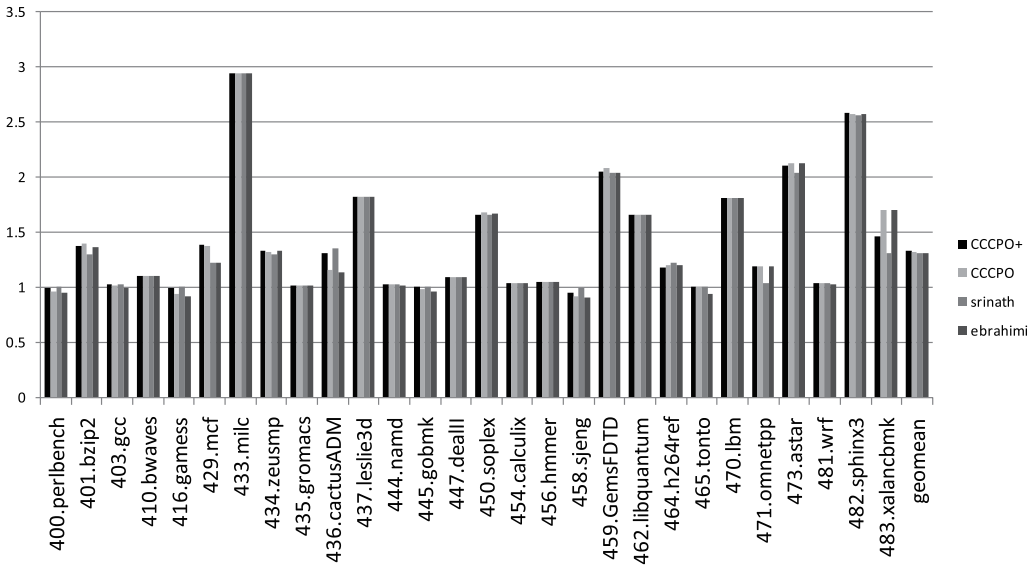

Fig. 11 Performance comparison of prefetcher throttling algorithms.

in these cases, CCCPO+ shows the best performance among the techniques (with the exception of 483.xalancbmk). For example, the performance is 17% better than that of FDP and CPT at 429.mcf. For the geometric mean, it has a performance that is 1.8% better than that of FDP and CPT, and is 0.5% better than that of CCCPO.

For further evaluation, we examined the performance of CCCPO+ and other existing throttling algorithms in various cache capacities. We focused on the characteristics of performance vs. size, as previously mentioned. Our CCCPO+ is expected to achieve robustness because it dynamically estimates the line-reuse frequency, and therefore, can adapt to various capacities and phases. In contrast, previous techniques, which are guided by the prefetcher accuracy or coverage, may not always work well because the threshold value will not always be valid for various capacities.

Figure 12 shows the results for 436.cactusADM for the example of polluting programs. The x-axis indicates

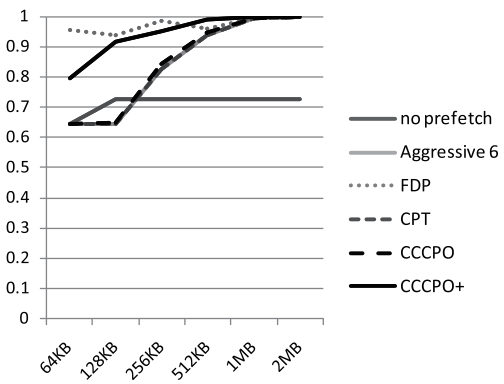

Fig. 12 Performance comparison of prefetcher throttling algorithms (436.cactusADM).

the LLC capacities and the y-axis indicates the relative IPC when compared to the IPC with perfect LLC (always hits). The effect of cache pollution is seen at smaller LLC regions

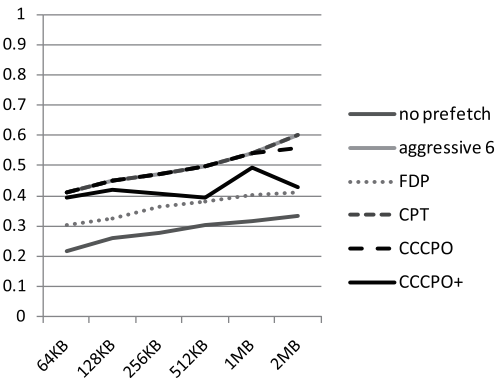

Fig. 13 Performance comparison of prefetch throttling algorithms (483.xalancbmk).

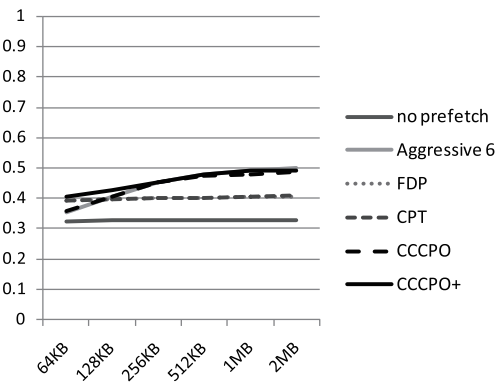

Fig. 14 Performance comparison of prefetch throttling algorithms (429.mcf).

in this graph. The performance of aggressive prefetching bellows even the performance of no prefetching at 128 KB. CPT and CCCPO show behavior that is similar to Aggressive 6, which is affected by cache pollution. We can see that FDP mitigates the pollution and performs proper level prefetching. CCCPO+ shows an average performance between FDP and others at 64 KB, but exhibits similar performance to FDP from the region of 128 KB. It shows that CCCPO+ and FDP properly accelerate the prefetcher better than both the no-prefetch and most-aggressive cases. In the larger LLC area, the effect of pollution decreases, and all of the techniques have a similar performance. We also observe poor locality because the performance of the no prefetch case exhibits no relation to the capacity. Proper prefetching is quite effective in such programs.

In Fig. 13, we see an example, in which even inaccurate prefetches are effective. Aggressive prefetching achieves a performance that is almost twice than that of no prefetching. Here, we see that CPT and CCCPO promoted the aggressiveness. Conversely, FDP fails to exploit prefetching because the prefetching accuracy is low in this program. CCCPO+ exhibits an average performance. It mostly accelerates the prefetcher, but the performance at 512 KB and 2 MB are somewhat limited.

Another example is shown in Fig. 14, which is the result of 429.mcf. This program also shows poor locality because the behaviors of the no prefetch and aggressive prefetching cases are relatively effective. However, only CCCPO+ and CCCPO were able to exploit the aggressive prefetching. In this program, both accuracy and coverage are below the threshold. This shows that there are cases when inaccurate and low coverage prefetchers still lead to improved performance. Our technique can adequately promote such cases.

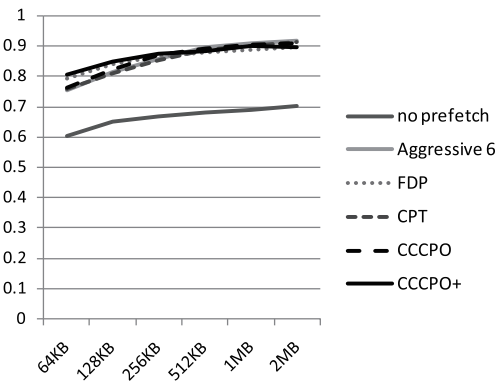

Fig. 15 Performance comparison of prefetch throttling algorithms (geomean).

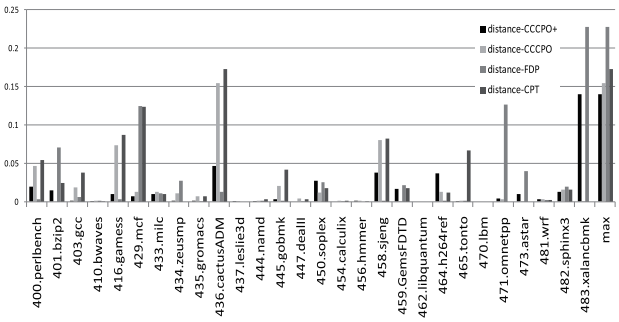

Fig. 16 Distance from the best throttling.

Figure 15 shows the geometric means of all the benchmarks. The result showed that FDP is superior at braking and CPT is superior at accelerating. This is considered to be the expected result considering their characteristics. Thus, FDP shows better performance at lower capacity, and CPT shows better performance at larger capacity. However, it depends on the programs to determine the technique that is most suited for a given capacity. CCCPO+ shows the best performance, especially at the lower region, which is sensitive to pollution and aggressive prefetching. CCCPO exhibits an intermediate position between FDP and CPT, and shows that the improved control algorithm of CCCPO+ was able to enhance the CC-based throttling.

Moreover, the distinguishing feature of CCCPO+ is its stability. Figure 16 shows the performance distance for the best aggressiveness at 256 KB LLC. The existing threshold-based approach sometimes shows a large distance from the correct aggressiveness. In the worst case, both FDP and CPT show more than 17% performance degradation, while

Table 4 Hardware budget.

FDP	11 counters (16 bit), 1 pollution filter (4,096 entries) 1 bit prefetched bit each tag and MSHR entry
total cost	20,784 bit/1 MB LLC
CPT (with single prefetcher)	11 counters (16 bit) 1 bit prefetched bit each tag and MSHR entry
total cost	16,688 bit/1 MB LLC
CCCPO	5 counters (16 bit) 1 bit accessed bit each tag and MSHR entry
total cost	16,592 bit/1 MB LLC
CCCPO+	6 counters (16 bit) 1 bit accessed bit each tag and MSHR entry
total cost	16,608 bit/1 MB LLC

CCCPO+ shows at most 14% degradation. By focusing on the number of programs that show a degradation that is larger than 5%, CCCPO+ is found to be 1, while FDP is 4, and CPT is 5. Because it is important for the worst-case performance to be introduced into the processor design, CCCPO+ is effective with respect to cost, performance, and stability, and is suited to current incoming LLC prefetchers.

6.4 Hardware Budgets

Note that CCCPO+ does not require large tables or additional tag array bits to trace the result of prefetching, as are required with existing throttling approaches. As shown in Table 4, CCCPO+ and CCCPO require fewer registers because they do not trace the line to determine whether it is prefetched. Of the related studies, FDP requires several thousand bit tables for the pollution filter [12]. CCCPO+ achieves superior throttling with a smaller hardware budget.

7. Conclusion

Hardware prefetching can efficiently hide the long memory latency, but it requires proper aggressiveness. This study proposed CCCPO+ as a novel approach to control prefetcher aggressiveness. Based on our previous CCCPO, the feedback algorithm is enhanced to dynamically scan the CC for neighbor aggressiveness, and to determine the best yet simple hardware. Evaluation results showed that the performance had improved by 30% when compared to no prefetching at 256 KB LLC. It also showed its superior stability when compared to other existing throttling approaches. The main reason is that we focused on cache line reuse, as opposed to the conventional way of achieving prefetcher accuracy or coverage. We also used an enhanced scan algorithm for the feedback. This approach effectively enhances current shared LLCs by improving their prefetcher efficiencies. Attempts are now being made to introduce this *CC* approach to other cache enhancement techniques, such as replacement and partitioning.

References

[1] D. Patterson, "Latency lags bandwidth," Commun. ACM, vol.47, no.10, pp.71–75, 2004.
[2] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," Int. Symp. on Computer Architecture, pp.240–251, 2001.
[3] A.C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead block correlating prefetchers," Int. Symp. on Computer Architecture, pp.144–154, 2001.
[4] A. Smith, "Sequential program prefetching in memory hierarchies," Computer, vol.11, no.12, pp.7–21, 1978.
[5] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," Int. Symp. on Computer Architecture, pp.364–373, 1990.
[6] J. Tendler, J. Dodson, J.J.S. Fields, H. Lee, and B. Sinharoy, "Power4 system microarhitecture," IBM J. Research and Development, vol.46, no.1, pp.5–26, 2002.
[7] W. Lin, S. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," Int. Symp. on High Performanc Computer Architecture, pp.301–312, 2001.
[8] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," Int. Conf. on Supercomputing, pp.499–500, 2009.
[9] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," Int. symp. on Computer Architecture, pp.252–263, 1997.
[10] X. Zhuang and H. Lee, "A hardware-based cache pollustion filtering mechanism for aggressive prefetches," Int. Conf. on Parallel Processing, pp.286–293, 2003.
[11] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," Int. Symp. on Microarchitecture, pp.397–408, 2006.
[12] S. Srinath, O. Mutlu, H. Kim, and Y. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," Int. Symp. on Highi-Performance Computer Architecture, pp.63–74, 2007.
[13] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," Int. Symp. on Microarchitecture, pp.222–233, 2008.
[14] E. Ebrahimi, O. Mutlu, and Y. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," Int. Symp. on Highi-Performance Computer Architecture, pp.7–17, 2009.
[15] E. Ebrahimi, O. Mutlu, C. Lee, and Y. Patt, "Coordinated control of multiple prefetchers in multi-core systems," Int. Symp. on Microarchitecture, pp.316–326, 2009.
[16] H. Irie, G. Honjo, and K. Hiraki, "Prefetch throttling technique based on dyanamic assumption," IPSJ Trans. Advanced Computing Systems, pp.56–66, 2010 (in Japanese).
[17] H. Irie, T. Miyoshi, G. Honjo, K. Hiraki, and T. Yoshinaga, "Cccpo: Robust prefetcher optimization technique based on cache convection," Int. Conf. on Networking and Computing, pp.127–133, 2011.

Hidetsugu Irie is an associate professor of Network Computing at the Graduate School of Information Systems at the University of Electro-Communications, Japan. His research interests include microarchitectures and network systems. He has a PhD in Information Science and Technology from the University of Tokyo. He was a researcher at the Japan Science and Technology Agency from 2004 to 2008, and was an assistant professor at the University of Tokyo from 2008 to 2010. He is also a member of the IPSJ, ACM, and IEEE.

Takefumi Miyoshi received his B.E., M.E., and D.E. from Tokyo Institute of Technology in 2003, 2005, and 2007, respectively. Since Apr. 2010, he has been with the Graduate School of Information Systems, UEC, where he is an assistant professor. His research interests are compiler techniques, many-core processor architecture, and co-design of hardware and software. He is also a member of the ACM, IEEE, and IPSJ.

Goki Honjo received his Bachelors and Masters degree from the University of Tokyo in 2009 and 2011, respectively. Since December 2011, he has been with the Graduate School of Science at the University of Tokyo, where he is an Assistant Professor. His research interest is computer architecture. He is also a member of the ACM and IEEE.

Kei Hiraki is a professor of computer science at the University of Tokyo. His research interests are parallel and distributed systems, high-performance computing, and networking. Hiraki received a PhD in physics from the University of Tokyo.

Tsutomu Yoshinaga received his B.E., M.E., and D.E. degrees from Utsunomiya University in 1986, 1988, and 1997, respectively. From 1988 to July 2000, he was a research associate of Faculty of Engineering, Utsunomiya University. He was also a visiting researcher at Electro-Technical Laboratory from 1997 to 1998. Since August 2000, he has been with the Graduate School of Information Systems, UEC, where he is now a professor. His research interests include computer architecture, interconnection networks, and network computing. He is a member of the IEEE and IPSJ.