CCCPO: Robust Prefetcher Optimization Technique Based on Cache Convection

Hidetsugu IRIE*, Takefumi MIYOSHI*, Goki HONJO†, Kei HIRAKI†, Tsutomu YOSHINAGA*
*The University of Electro-Communications, Japan
1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585, Japan
E-mail: irie@is.uec.ac.jp, miyoshi@is.uec.ac.jp, yosinaga@is.uec.ac.jp
†The University of Tokyo, Japan
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
E-mail: honjo@is.s.u-tokyo.ac.jp, hiraki@is.s.u-tokyo.ac.jp

***Abstract*—One of the significant issues of processor architecture is to overcome memory latency. Prefetching can greatly improve cache performance, however, it has the drawback of cache pollution unless its aggressiveness is properly set. Although several techniques for prefetcher throttling have been proposed which use accuracy as a metric, their robustness were not sufficient due to the variations between program working set sizes and cache capacities.**

In this paper, we revisit cache behavior with the viwepoint of data lifetime in a cache with prefetching. Based on this observation Cache-Convection-Control-based Prefetch Optimization (CCCPO) is proposed, which exploits the characteristics of cache line reuse and controls the prefetcher aggressiveness. Evaluation results showed that this novel approach achieved 4.6% improvement against the most recent prefetcher throttling algorithms in the geometric mean of SPEC CPU 2006 benchmark suite with 256KB LLC.[1]

I. Introduction

An important issue for micro-architecture is to conceal memory latency and to make the functional units busy [1]. Currently, off-chip memory latencies have been increased to several hundred cycles, which is hard to absorb by out-of-order execution that re-orders at most a few tens of instructions. Data parallel architectures [2] that exploit data- or thread-level parallelism, and overlap many memory accesses, are effective in concealing memory latency, their applications are limited since their effectiveness depends on a balance between computation time and data transfer time. On the other hand, prefetching, though it is speculative, has been proposed to hide memory latency and requires only simple address predictors.

Consequently, hardware prefetching [3], [4] has become an important solution for memory latency of general purpose processors. Recently, various prefetching techniques have been adopted by commercially-produced processors [5], and more advanced prefetching techniques have been proposed [6]–[9].

However, the proper amount of prefetching depends on the application and the cache capacity. Although aggressive prefetching may work effectively for certain types of applications, it can pollute the cache and degrade performance for other applications. It can also increase unnecessary memory transfers when the accuracy of address prediction is low. If prefetching is as aggressive as in the most recently proposed prefetching algorithms, the side effects can be serious. Thus, the most recent prefetching algorithms generally involve mechanisms that dynamically optimize the amount of prefetching. In most cases, the accuracy of address prediction is used as a metric for prefetch usefulness since the current data in the cache should be valuable [10]–[12]. In this case, the prefetched lines are traced to check if they are actually accessed later and, thus, estimate the prefetch accuracy. Prefetch access is suppressed when the accuracy falls below a predetermined threshold.

However, the latest study has shown that the newest prefetcher algorithm that spreads access to many possible address is effective for a large last level cache (LLC). Thus, it is not necessarily the case that prediction accuracy directly corresponds to performance [9]. On the other hand, even high accuracy prefetching can cause cache pollution when working set size is greater than the cache size. Deciding whether to throttle or to accelerate for optimal prefetching by accuracy statistics is difficult because of such variations.

This paper proposes a new prefetcher throttling technique called Cache-Convection-Control-based Prefetch-Optimization (CCCPO), which achieves stable prefetcher throttling for different programs. It focuses on the balance between the "convection" intracaches and the speed of data streaming, which have been ignored in previous work.

The paper is organized as follows. Section II discusses the control techniques for prefetcher aggressiveness, while Section III provides observations about data convection in a cache and shows that this behavior can guide the optimal control of prefetcher aggressiveness. Section IV describes the new technique. Section V presents the evaluation environment, while Section VI gives the results. Section VII introduces related work, and Section VIII concludes the paper.

II. Prefetcher Throttling

A. Controlling Prefetch Amount

Existing control techniques can roughly be divided into two approaches. The first group estimates the accuracy of address prediction at each prediction time [10]. A table entry that contains whether the past prefetched line was actually accessed is used to predict the usefulness of that prefetch, using the load instruction's PC or predicted address as a key. Prefetch access is canceled when it is predicted as useless. This approach has similar effects as confident counters for address prediction, but it allows the address predictor and accuracy predictor to be configured more flexibly. This approach requires additional tables to maintain the past prefetch histories.

The other group of algorithms calculates statistically whether prefetches are performing well at a given period, and uses this as feedback to adjust the aggressiveness of the prefetcher in the next period [11]–[13]. The aggressiveness is controlled not by cancelling a certain prefetch but by changing the parameters of prefetching, which determines how far ahead of the predicted address stream the prefetcher sends requests, such as prefetch distance, degree, or depth. This control is generally called "prefetcher throttling" and is suitable for controlling the aggressiveness of prefetchers that can generate many requests at once.

Both approaches generally focus on prefetch accuracy, which indicates how many prefetched lines were actually accessed later on. However, Ebrahimi et al. [13] showed

[1] ©2011 IEEE. Reprinted, with permission, from Hidetsugu IRIE, Takefumi MIYOSHI, Goki HONJO and Tsutomu YOSHINAGA: CCCPO: Robust Prefetcher Optimization Technique Based on Cache Convection, Int. Conf. on Networking and Computing, Dec. 2011.

that introducing a coverage metric that indicates how many cache misses were covered by the prefetcher to the feedback mechanism improves the performance of prefetchers.

B. Issues with Prefetcher Throttling

Previous throttling techniques increase the aggressiveness of prefetcher when the accuracy or coverage are greater than the fixed threshold and decrease when they are below the threshold. Thus, the higher threshold value implies better braking, while a lower threshold value implies larger acceleration. However, it is hard to control various applications using a fixed uniform threshold value, especially for the following cases: i) though the prefetch accuracy is high, useful lines are pushed out from the cache when the cache has no vacancies; ii) though the prefetch accuracy is low, aggressive prefetching improves performance when the cache has vacancies; iii) though both accuracy and coverage are low, prefetching is still effective due to the poor locality of the application access pattern.

III. Cache Convection

A. The Target of Prefetcher Throttling

Prefetcher throttling varies the timing and amount of data transfer to the cache. When a certain cache line is transferred to the cache earlier by prefetching, the swapped line is evicted earlier from the cache. Also, if several lines are transferred to the cache at once by prefetching, several lines are evicted at the same time. The gain in prefetching is the difference between the value of the prefetched and evicted lines. However, previous techniques generally assume that the evicted lines have a fixed value, while they deliberately estimate the value of the prefetched line, for example, using accuracy statistics. This naive assumption decreases the robustness of the prefetcher throttling for reasons i) to iii). For this problem, 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 costs several thousand bits of table, thus is not desirable.

Prefetching has a significant effect by accelerating the replacement of cache contents when executing an application where the working set of the application is replaced so rapidly that the cache algorithm cannot work well. To accelerate replacement, deep and net-spreading prefetching should be effective. On the other hand, too much acceleration makes the cache contents too advanced of the execution, thus causing cache pollution. This observation indicates that the optimal prefetcher throttling algorithm is to perform the most possible aggressive prefetching within which a given line is not evicted before it is used.

B. Cache Convection and Prefetcher Throttling

After transferred to the cache line, data are accessed at certain times that are inherent by the application program. Then, the data is moved to the LRU side of the set as new lines enter and finally be evicted from the cache. Although it is generally hard for each cache line to judge whether it will be used in the future, a characteristic value can be determined using the statistics of the access behavior. Hereafter we propose the estimation technique, and show that it can use as the metric for prefetcher throttling.

Consider a cache that has N cache lines. Focusing on a cache line l, which has not yet finished its last use, to remain in the cache; this line has to be accessed before at most N lines are pushed out. [2] Once the line is accessed, it returns to the MRU side. Here, if there are h cache hits while N lines are pushed out, the number of accesses to the focused line is statistically estimated as follows.

$$\frac{h}{N \times \mu} \qquad (1)$$

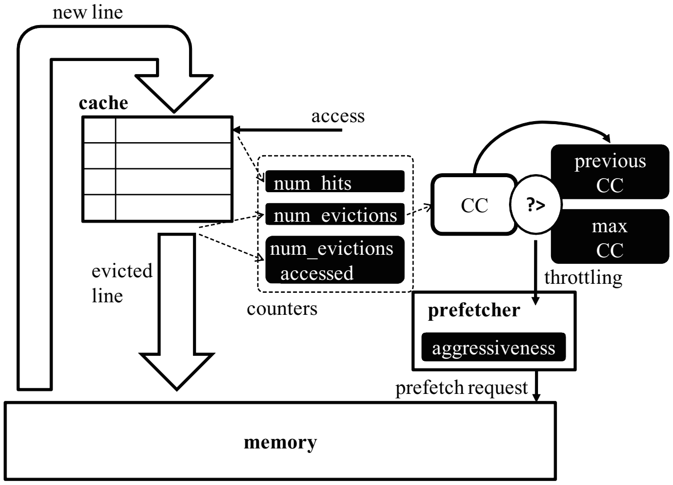

Fig. 1. Outline of the Proposed Technique

where μ is ratio of lines in the cache that have not finished their last use. Notice that the cache hits are at such lines, not all lines in the cache.

This value represents relatively how many times each cache line is used again. For example, the value becomes low when the access pattern shows little locality as streaming access patterns, and becomes high when access concentrates to small working sets. Also, for LLC, the value becomes low if higher level caches have enough size to store the temporal working sets. We define this value "*Cache Convection*" or *CC* as the image of the cache line convection between the MRU side and the LRU side.

The relationship between *Cache Convection* and the amount of prefetch can be determined as follows. As the amount of prefetch increases, initially, *CC* increases because the compulsory misses are reduced and h in Equation 1 is increased for the same line swapping period. Then *CC* saturates at a certain value that is inherent to the program code and begins to decrease significantly when cache pollution occurs due to too aggressive prefetching as the number of line swapping increases for the same program portion and h in Equation 1 decreases. Consequently, *CC* shows the highest value when the prefetcher aggressiveness is optimal for that program phase.

Focusing on this characteristic, a new prefetcher throttling technique, the Cache Convection Control-based Prefetch Optimization (CCCPO) is proposed. CCCPO calculates *CC* from the number of cache hits when a certain number of cache swaps has occurred. Then, CCCPO maintains the optimal prefetch amount by decreasing or increasing prefetcher aggressiveness for the next period according to whether the calculated *CC* was decreased or increased.

IV. Implementation of CCCPO

A. Hardware Outline

Figure 1 shows the outline of the CCCPO. Program execution is separated dynamically at the enough long period to calculate the statistics, for example, every 2 thousand cache swapping. *CC* is calculated for every period, and the prefetcher aggressiveness for the next period is determined based on whether *CC* was increased or decreased. The proposed technique only requires several registers to be implemented, and it does not need any history tables for feedback.

For counting the number of cache hits and other events, the following three counters are added:

- *num_evictions*, which counts the number of cache evictions and thus can trigger the process for starting a new period;

[2] From a statistical perspective, each set is assumed to be accessed equally.

- *num_hits*, which counts the number of hits in a given period; and
- *num_evictions_accessed*, which counts the number of lines that have been pushed out from the cache after having been accessed at least once and is used to estimate μ in Equation 1.

These counters are reset to zero after each period and incremented at each cache access or line swap. As well, the following three registers are added to store the control values:

- *reg_aggressiveness*, which indicates the current aggressiveness of the prefetcher. The prefetcher varies the number of prefetching requests or the prediction algorithms based on this value;
- *reg_previousCC*, which stores the last *CC* so that it can be compared when the next *CC* is calculated at the end of the current period; and
- *reg_maxCC*, which stores the maximum *CC* of the last several periods, so that phase transition can be detected.

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 correspond cache line and is set to one when any accesses are performed to 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. As well, 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 paper, an additional divider is used for the performance evaluation, but the performance difference is negligible.

B. Estimation of CC

μ in Equation 1, which represents the fraction of the cache line that has not been finished the last use can be roughly approximated with the fraction of evicted cache lines that has been accessed at least once. This approximation is not very accurate, but it works well because most cache lines in today's LLC are accessed less than twice due to the large size of higher level caches. Cache lines can be approximately split into two groups (those that are transferred to the cache and accessed and those that are transferred to the cache and not accessed) to estimate the ratio. The lines that are evicted while not having been accessed and, thus, are evicted with an access bit of '0', can be classified into the following groups: i) requested by a demand miss (cache miss), which is actually the last use so that the line is not used later; and ii) requested by prefetch, but the prediction was a miss.

Thus, *CC* can be estimated for this period using the values of the counters at the end of the period:

$$\mathrm{rawCC} = \frac{\mathrm{num\_hits}}{\mathrm{num\_evictions\_accessed}} \tag{2}$$

The value of Equation 2 could be affected by sampling noises because the period are separated regardless of the program code. To make the feedback robust, the *CC* are accumulated, and, then the current value is estimated using

$$\mathrm{CC} = \frac{\mathrm{rawCC} + \mathrm{reg\_previousCC}}{2} \tag{3}$$

C. Feedback Algorithm

As mentioned in Section III, the optimal control is achieved when *CC* shows the maximum value for that program phase. Whenever *num_evictions* reaches a certain value, the following feedback control is performed. First, the new *CC* that is calculated using Equation 3 is compared to the previous *CC*, which is stored in the register *previous_CC*. Prefetcher aggressiveness is decreased if the new *CC* is smaller than the previous value and *vice versa* for increasing the aggressiveness. *CC* gradually approaches the maximum value after several periods, and thus the prefetcher goes to optimal. In the case of decrease, hysteresis is applied to make the feedback robust to the sampling noise, as well as accumulation of *CC*. It requires greater difference than the threshold. After updating the aggressiveness, all counters are reset to zero, and a new period is started.

TABLE I
MICRO-ARCHITECTURAL PARAMETERS FOR 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	256 KB, LRU, 16 way, 64 B line, 20 cycle latency
memory access	200 cycle + arbitration latency

The maximum value of *CC* will change as the program phase changes. For example, consider the case when a phase which accesses the same cache lines repeatedly is shifted to different phase which scans a large address space and does not reuse the lines. In this case, *CC* rapidly decreases from a high value to a low value at the point of phase change. Though throttling should accelerate aggressiveness for the stream access after the phase change, throttling may decrease the aggressiveness because *CC* decreases gradually from the high value of the previous phase to the low value of the current phase due to accumulation in equation 3. Generally, programs often contain many such phase changes.

Thus, the proposed algorithm detects phase changes using *reg_maxCC*. *Reg_maxCC* stores the maximum value of *CC* for recent past values so that it maintains the maximum value of that phase, and this value can be considered as unique to the phase. Phase change is detected when the new *CC* is significantly below *reg_maxCC*. In this case, the accumulation of *CC* and *reg_maxCC* is reset, and prefetcher aggressiveness can be incremented rapidly to the optimal level.

V. EVALUATION METHOD

A. Baseline Processor

Performance evaluation is done thorough 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 I. In the evaluation, a single thread is executed in a single core. Prefetch is applied to LLC, that is L2 cache in this model, because the prefetcher abilities for hiding the large memory access latency are focused on. Memory latency is modeled as the sum of 200 CPU cycles of access latency and queuing latency for the memory bus confliction. Memory bandwidth is set to at most one line transfer per ten CPU cycles, which is similar to the regulation of DPC-1[3].

B. Prefetcher

The proposed throttling was applied to the sequential prefetcher (stream-based prefetcher) [4] for evaluation. Prefetch is performed to the sequential addresses of the missed address on a cache miss. Aggressiveness was set to 7 levels (Table II), which included adding two more aggressive levels "level5" and "level6" to the 5-level setting that Ebrahimi et al. used [13].

[3] http://www.jilp.org/dpc/

TABLE II
SETTINGS OF PREFETCHER AGGRESSIVENESS

level 0	none
level 1	sequential depth 4
level 2	sequential depth 8
level 3	sequential depth 16
level 4	sequential depth 32
level 5	sequential depth 64
level 6	sequential depth 128

C. Throttling Parameters

In the evaluation, CCCPO was configured as follows. Feedback is performed for every 2-K line evictions, aggressiveness is decremented when *CC* is reduced by 25% from the previous *CC*, and a phase change is detected when *CC* falls below 5% of *reg_maxCC*. These values were determined based on a preliminary evaluation.

D. Benchmarks for the Evaluation

For the evaluation of section VI-A, eighteen programs from the SPEC CPU 2006 benchmark suite were selected where LLC misses and prefetcher aggressiveness affect performance significantly, more than 20%, around the LLC capacity of 256KB. For the evaluation of section VI-B, we evaluated all the benchmark programs of SPEC CPU 2006. Each program was compiled by gcc version 4. 2. 2 with the -O3 option. Cycle accurate execution of 100 million instructions after skipping 10 billion instructions from the program head was considered.

VI. EVALUATION

A. The Performance of CCCPO

Figure 2 shows the execution performance for the 18 benchmark programs. The *y*-axis shows the relative IPC that is normalized by the performance with no prefetching. The performance with fixed prefetcher aggressiveness (no feedback) is shown in the bar “fix 1” to “fix 6”, from the leftmost to the right, each of which represents the fixed aggressiveness of “level 1” to “level 6” in the tableII.

Focusing on the relationship between aggressiveness and IPC, it is shown that each program has proper aggressiveness for which the performance impact is significant. For example, 433.milc is significantly accelerated with aggressive prefetching; on the other hand, 458.sjeng decreases its performance with aggressive prefetching. The performance change ranges from a low of -30% to a high of +250%. The proposed technique shows optimal performances for all the benchmarks except 483.xalancbmk. This suggests that proper dynamic throttling was achieved. Using the geometric mean of the 18 programs, the performance is increased by 53% from the baseline and 3.7% from “fix 5”, which showed the highest performance in fixed aggressiveness in this processor model.

A detailed figure of throttling is shown in Figure 3. The *y*-axis shows the miss ratio (notice that it is the ratio of L2 misses to L1 misses) and *CC* in the bottom graph, and the aggressiveness in the top graph. The graph plots these values of each throttling period in the execution of 458.sjeng. The *x*-axis represents the retired instructions (execution point), and thus the density of the graph represents the frequency of cache line swapping because throttling is performed for each 2 thousand line swapping. This graph is expanded for certain execution parts so that the variation in each period can be seen clearly. It can be seen that *CC* decreases when the aggressiveness increased from 0 to 1, and then the aggressiveness of the next period is decreased to 0 based on *CC*, so that the aggressiveness remains around the proper level. It can be seen that this throttling actually prevented cache pollution, which is seen in Figure 2. For the other example, Figure 4 shows a similar plot but for execution of 450.soplex. This graph plots all periods of thorough 100 million instruction execution. The graph is so dense that the relationship between each period is hard to see. However, we can see that the characteristic phases are repeated during execution. Prefetcher aggressiveness shows different values depending on the phase of the program.

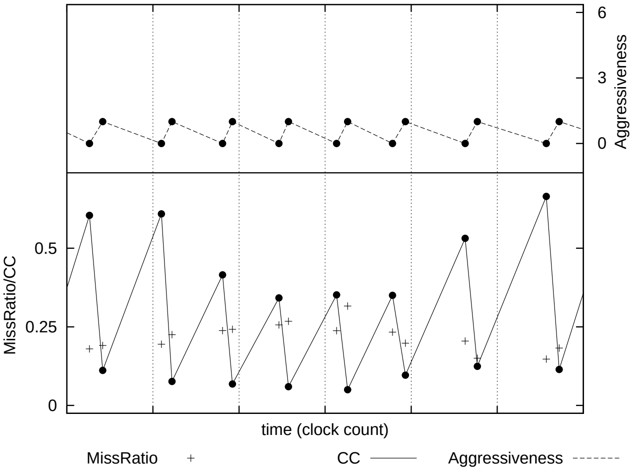

Fig. 3. Throttling Timeline (458.sjeng)

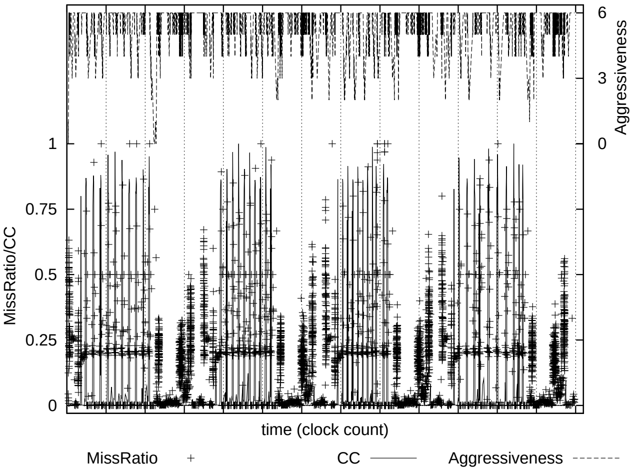

Fig. 4. Throttling Timeline (450.soplex)

B. Comparing Throttling Techniques

For further evaluation, here we examine the robustness of CCCPO. Also, the comparison to other prefetcher throttling techniques is performed. For the caches on nowadays processors that is shared by multiple cores or has the function of power-gating, prefetcher is desirable to be robust for the dynamic change of cache capacity. Our CCCPO is expected to achieve such robustness since it estimates the line-reuse frequency dynamically and thus can adapt to various programs and phases. In contrast, previous techniques which is guided by prefetch accuracy or coverage may not always work well for various programs or cache capacities.

To evaluate this robustness, we measured the IPC of each throttling techniques for the execution of all of the each benchmarks in SPEC CPU 2006 by varying the cache capacity from 64KB to 8MB. The parameters except LLC capacity are set to the same value of the previous evaluation (Table I). Notice that the parameters for throttling mechanism is constant while the cache capacity varies. Compared techniques are as follows;

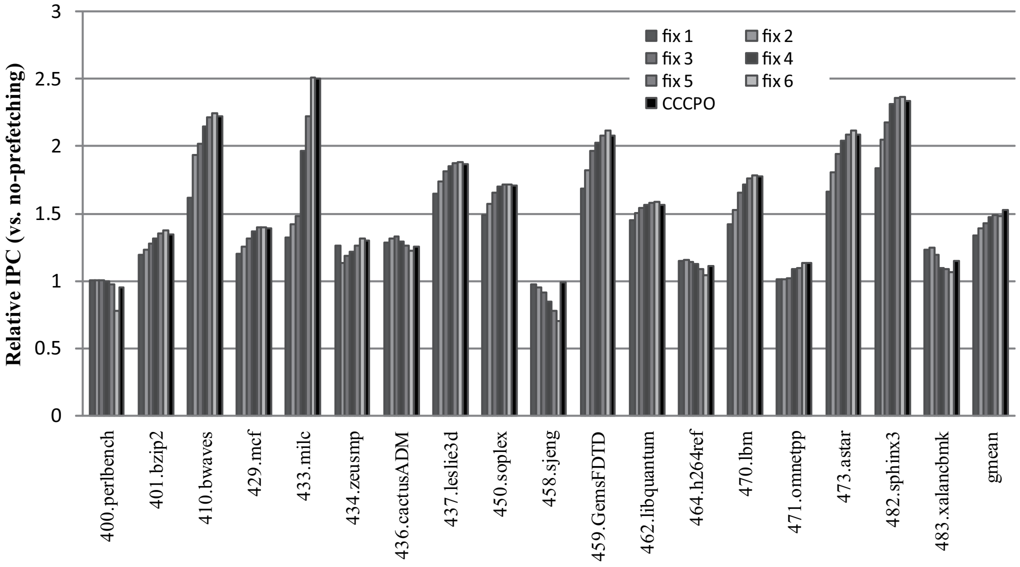

Fig. 2. Performance Comparison between Fixed Aggressiveness and CCCPO (IPC)

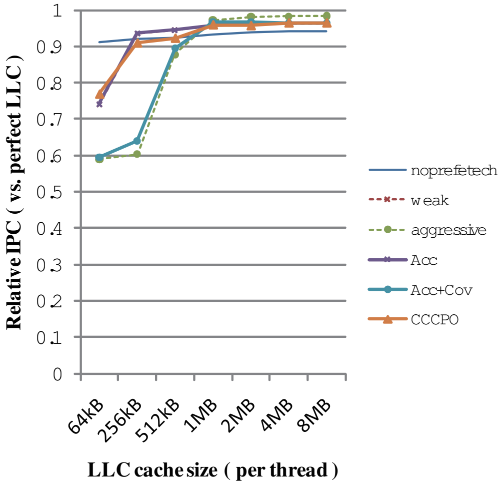

Fig. 5. Performance Comparison of Prefetch Throttling Algorithms (403.gcc)

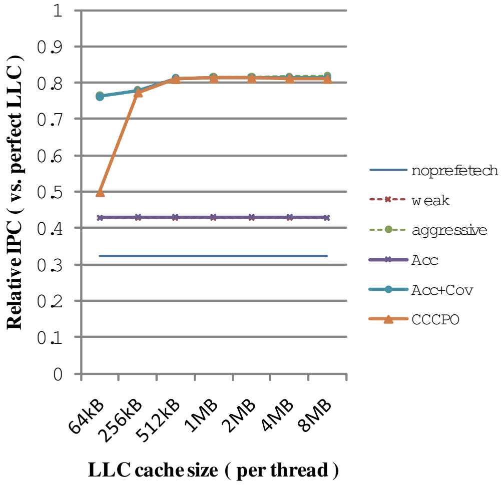

Fig. 6. Performance Comparison of Prefetch Throttling Algorithms (433.milc)

- Acc : Guided by prefetcher accuracy.
- Acc+Cov : Guided by both accuracy and coverage, as the technique of Ebrahimi [13].
- CCCPO : Guided by *CC*.

Acc increments the aggressiveness for the next period when the prefetch accuracy of a period is greater than the threshold, decrements otherwise. Acc+Cov increments the aggressiveness when either accuracy or coverage is greater than its threshold, decrements otherwise. We set the threshold to 60% for accuracy, 20% for coverage, which is the best configuration according to preliminary evaluation when LLC is 256KB.

Here we show the result for execution of 403.gcc in figure5. The x-axis indicates the LLC capacity, y-axis indicates the relative IPC compared to the IPC with perfect LLC (always hits). Besides the three lines which show each throttling, no-prefetch, fixed to weak aggressiveness and fixed to strong aggressiveness are shown for reference. This graph shows aggressive prefetching causes cache pollution when LLC capacity is less than 512KB. Acc succeeds to brake the prefetcher while Acc+Cov cause pollution by accelerating the prefetcher. CCCPO avoids the pollution as well as Acc. In the large LLC area, the effect of pollution decreases and all techniques show similar performance.

On the other hand figure6 shows the result of 433.milc. 433.milc is known as a program which the aggressive prefetching achieves significant performance improvement. However the graph shows the Acc fails to promote the prefetching. As in this case, Acc is not effective for programs which inaccurate but high coverage prefetcher has significant effects. Acc+Cov and CCCPO (except the case for 64KB) succeed to acceleration. This graph also shows that the performance of no-prefetch hardly changes with cache capacity. For the access pattern of this program, prefetching is much effective than the larger cache capacity.

Another example is shown in figure7, the result of 482.sphinx. For this program CCCPO shows the best acceleration at 256KB LLC. This shows that there are cases

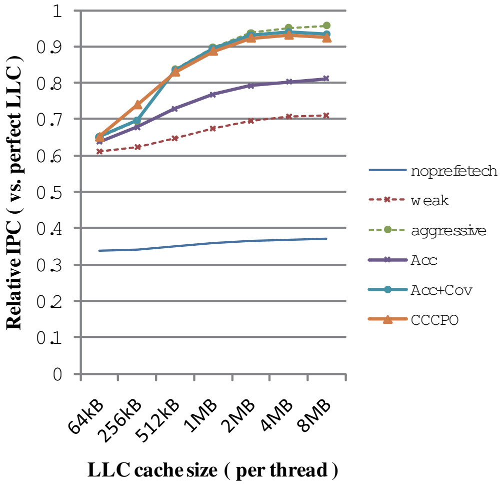

Fig. 7. Performance Comparison of Prefetch Throttling Algorithms (482.sphinx)

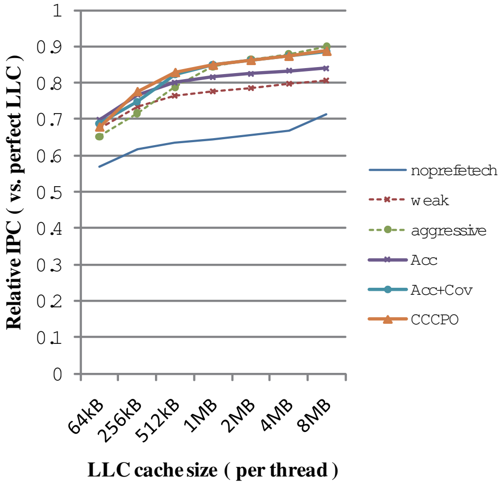

Fig. 8. Performance Comparison of Prefetch Throttling Algorithms (geomean)

when inaccurate and low coverage prefetcher still improves the performance. Our technique can promote such cases properly.

As seen above, CCCPO achieved robust throttling for various programs and cache capacities. CCCPO properly accelerate even inaccurate prefetcher while properly avoids cache pollution. On the other hand, existing throttling sometimes fails to control. The geometric means of all benchmarks are shown in figure8. Acc works effectively at low capacities, but remains low improvement at large capacities. Acc+Cov works effectively at large capacities. CCCPO shows stable performance for all capacities. Especially, it shows the best performance at 256KB to 512KB, which is hard to determine whether increments or decrements the aggressiveness. Figure 9 shows the worst distance from the optimal throttling (the best of three) through all the programs. It shows CCCPO hardly miss-throttles the prefetcher. These results show that CCCPO is effective, though it is simple, for the caches of nowadays multicore processors.

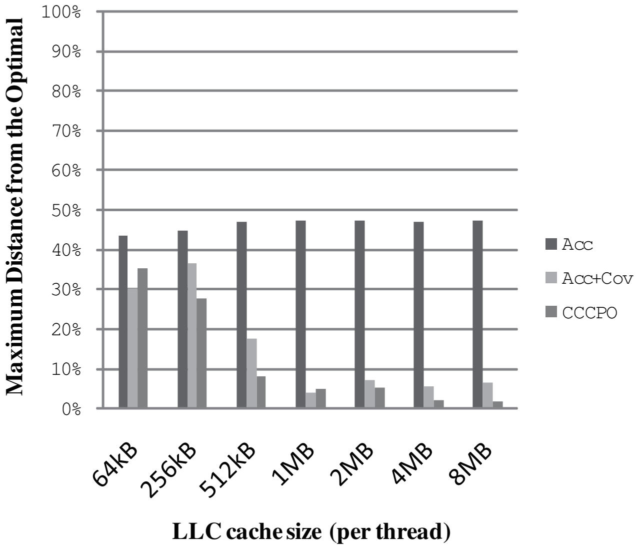

Fig. 9. Worst Distance from the Optimal Throttling

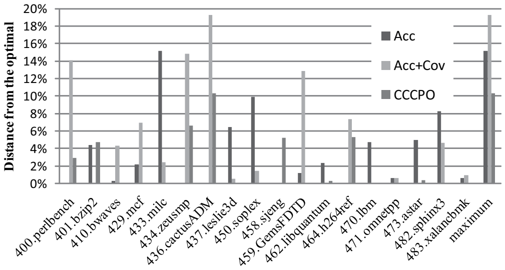

Fig. 11. Distance from the Optimal Throttling

C. Applying to other Prefetcher Algorithms

Application of CCCPO is flexible and is not limited to sequential prefetchers. For example, figure10 shows the result , in the similar manner to figure2, when throttling are applied to the stride prefetcher. The parameters of processor and prefetcher throttling are same as the parameters shown in Table 1 and 2, respectively. The introduced stride predictor is C-Zone based [14], implemented in GHB [7], and its depth is varied according to the aggressiveness as in the tableII.

The results of the geometric mean show that the settings of “fix3” or “fix4” were the best for this environment, but generally it is not known in advance, also, it varies from environment to environment. Prefetcher throttling techniques achieve the similar performance to the second best settings as “fix2” or fix“5” by feedback controlling. Among the feedback techniques CCCPO achieves higher performance than Acc and Acc+Cov for most of the benchmarks. While geomean of Acc and Acc+Cov respectively are 1.23 and 1.21, geomean of CCCPO is 1.25. Thus CCCPO achieves 4 % better performance than the latest technique and even 2% better than Acc that is suitable for this capacity. Figure 11 shows the distance from the optimal (the best performance of three). CCCPO shows the robust performance for various programs, capacities and predictors than other techniques.

VII. Related Work

Hur et al. [11] proposed an adaptive stream prefetching that statistically estimates the most frequent stream length, and

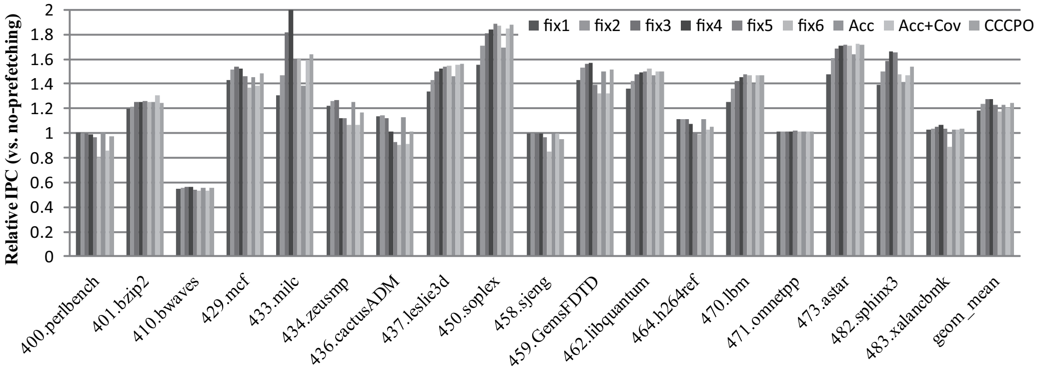

Fig. 10. Performance of Stride Prefetcher

cancels prefetch access that exceeds this length. This technique improves the prefetch efficiency, especially in programs that have many short streams by canceling the prefetch overruns that occur at the end of each stream. Srinath et al. [12] proposed feedback directed prefetching that uses prefetch accuracy, lateness, and pollution during a given period to determine the prefetcher aggressiveness of the next period. Besides accuracy, this feedback approach introduces lateness, which represents the hit in the access pending list, and pollution, which detects access to evicted lines. However, it requires thousands bits history table (implemented in a bloom filter) for detection of pollution. Ebrahimi et al. [13] proposed the technique for throttling multiple prefetchers by improving the feedback-directed prefetching, by using the accuracy and coverage for feedback. Furthermore, Ebrahimi et al. proposed a technique for using accuracy-based throttling on multicore processors [15]. Ramos et al. proposed a throttling technique that uses cache hit counts [16]. Comparing the total cache hits of a period to that of the previous period, aggressiveness is increased or decreased. This technique does not require tracing the prefetch result, like the proposed CCCPO, and can be implemented with simple hardware. However, unlike CCCPO, the technique can be affected by phase changes, because cache hit counts may change drastically between the phases.

VIII. Conclusion

Hardware prefetching can efficiently hide the long memory latency, but it requires proper aggressiveness tuning. This paper proposed new prefetcher throttling technique that focuses on the cache line re-use, instead of using prefetch accuracy or coverage-based approaches. Based on the observation that the number of cache line re-uses is statistically a unique value for each program phase, the metric "cache convection" is introduced. "*CC*" indicates the maximum value when the speed of the program advance and cache data swapping are balanced, which implies that proper prefetcher aggressiveness is achieved.

The proposed prefetcher throttling technique CCCPO was evaluated. The evaluation results showed that CCCPO is able to throttle the prefetcher properly for various programs and cache capacities. It showed better performance than other throttling approaches, by 4.6% on the geometric means of SPEC CPU 2006 with 256KB LLC. Besides, it showed the best performance for the most cases in spite of that existing techniques have each sweet spots of cache capacity. As well, compared to existing techniques, it does not require sensitive threshold values.

References

[1] D. Patterson, "Latency Lags Bandwidth," *Commun. ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[2] J. Gebis and D. Patterson, "Embracing and Extending 20th-Century Instruction Set Architecture," *IEEE Computer*, vol. 40, no. 4, pp. 68–75, 2007.

[3] A. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[4] 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.

[5] J. Tendler, J. Dodson, J. J.S. Fields, H.Lee, and B. Sinharoy, "Power4 system microarhitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–26, 2002.

[6] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 279–290, 2002.

[7] K. Nesbit and J. Smith, "Data cache prefetching using a global history buffer," *Int. Symp. on High Performance Computer Architecture*, pp. 96–105, 2004.

[8] S. Somogyi, T. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-Temporal Memory Streaming," *Int. Symp. on Computer Architecture*, pp. 69–80, 2009.

[9] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," *Int. Conf. on Supercomputing* , pp. 499–500, 2009.

[10] X. Zhuang and H. Lee, "A Hardware-based Cache Pollution 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] 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.

[14] S. Palacharla and R. Kessler, "Evaluating stream buffers as a secondary cache replacement," *Int. Symp. on Computer Architecture*, pp. 24–33, 1994.

[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] L. Ramos, J. Briz, P. Ibanez, and V. Vinals, "Multi-level Adaptive Prefetching based on Performance Gradient Tracking," *Workshop on JILP Data Prefetching Championship*, 2009.