ASTES

Advances in Science, Technology and Engineering Systems Journal
Vol. 7, No. 6, 126-138 (2022)
www.astesj.com
Special Issue on Innovation in Computing, Engineering Science & Technology

ASTES Journal
ISSN: 2415-6698

Extended Buffer-referred Prefetching to Leverage Prefetch Coverage

Jinhyun So[*], Mi Lu

Texas A & M University, Department of Electrical & Computer Engineering, College Station, TX 77843, US

A R T I C L E I N F O

Article history:
Received: 14 August, 2022
Accepted: 21 November, 2022
Online: 08 December, 2022

Keywords:
Prefetching
Prefetch Coverage
Cache

A B S T R A C T

This paper is an extension of the work originally presented in the 26th International Conference on Automation and Computing. This study regarding hardware prefetching aims at concealing cache misses, leading to maximizing the performance of modern processors. This paper leverages prefetch coverage improvement as a way to achieve the goal. Original work proposes two different storage buffers to enhance prefetch coverage; block offset buffer and block address buffer. The block offset buffer updates its contents with the offsets of a cache block accessed, while the block address buffer contains the address of a cache block prefetch-issued. The offset buffer is utilized to speculate a local optimum offset per page. The offset buffer is proposed to adopt multiple lengths of delta history in observing offset patterns from completely trained table. This paper advances to employ incompletely trained table as well, while in other prefetching methods including original work, only completely trained candidates are utilized. Furthermore, we construct the table on the fly. Rather than using only completely built tables, we offer utilizing and updating table concurrently. This paper also proposes a refined metric from existing prefetch accuracy metric, to measure net contribution of a prefetcher. Compared to the original work, we have 2.5% and 3.8% IPC speedup increment with single- and 4-core configuration, respectively, in SPEC CPU 2006. In SPEC CPU 2017, our work achieves 4.5% and 5.5% IPC speedup improvement with single- and 4-core configuration, respectively, over the original work. Our work outperforms the 2nd best prefetcher, PPF, by 2.9% and 2.7% IPC speedup with single- and 4-core configuration, respectively, in SPEC CPU 2006. In SPEC CPU 2017, our work surpasses both Berti by 1% and SPP by 2.1% IPC speedup with 4-core configuration in SPEC CPU 2017.

1 Introduction

IN [1], the author necessitates the implementation of memory hierarchy, which attempts to greatly shorten the average memory access time due to huge performance gap between the processor execution speed and memory latency. The execution speed of the processor has significantly increased while memory has pursued higher densities that causes increased memory latency. The enormous gap has been increasing due to the different objectives in developing the processor and memory. Hierarchy of cache memory has been introduced to reduce the performance gap by improving average memory access time, depending on two kinds of memory reference locality; temporal and spatial locality. However, the cache memory is still limited in reducing the gap with a trade-off between its capacity and the speed of cache hierarchy levels. That is, the cache levels closer to the cores are smaller size but have shorter latency. On the contrary, the cache levels farther apart from cores are of larger size but have longer latency. So, prefetching has been proposed as an effective technique that can bridge the performance gap by proactively fetching data ahead of processor's request into the cache closer to the cores. This paper proposes an effective technique in terms of prefetch coverage.

Prefetching is a mechanism that comprehends the memory access pattern of the program, and speculatively predicts and issues memory addresses ahead of the program's access to them. Hardware prefetching is to employ a standalone hardware that is dedicated to the prefetching. That is, hardware prefetcher predicts the future access to a memory data based on the observed access pattern of memory addresses in the past. Then, the prefetcher requests the data from low level of the memory hierarchy and stores it into a higher level cache close to the processor. Thus, by prefetching data, the prefetcher helps prevent cache misses due to the future access to the data, hiding long latency of low level cache access.

The objective of hardware prefetching is to proactively continue to fetch useful cache blocks from low level cache throughout run time based on memory access patterns. Usually, the algorithm

[*]Corresponding Author: Jinhyun So, & Email: jxs1525@tamu.edu

of a hardware prefetcher observes the past memory access patterns, predicts a future memory data access, and issues the address of the data. The patterns of a past memory access can be found based on the presumption of the existence of spatial and temporal locality of memory accesses.

In this paper, we propose an extended work of one of state-of-art prefetcher [2] with effective techniques maximizing prefetch coverage[1] at a moderate cost of prefetch accuracy. The contributions of this paper are categorized into the following three parts.

1.1 *Efficient use of hardware storage*

Following two buffers are employed for prefetching; one is block offset buffer and the other one is block address buffer to store the offset of a block and the address of prefetched block, respectively, in a FIFO manner. The offset buffer provides a reduction of hardware overhead by generating virtual tables[2] for prefetching instead of real table. The offsets of cache blocks are stored in the tables virtually generated whenever they are needed; storing the original offsets and generating the virtual tables can save hardware overhead by 45%[3] in terms of total hardware overhead.

1.2 *Maximizing prefetch coverage*

We propose following techniques for maximizing prefetch coverage from the two buffers as an offset prefetching: We use the history of a delta in multiple lengths to find diverse access patterns through access history. We mine access patterns from completely trained (virtual) table. Besides, they are also mined from incompletely trained (virtual) table.[4] Access patterns from the incompletely trained table are utilized until the table's training is completed. After the training is finished, access patterns from the completely trained table are used. Moreover, we also utilize an on-the-fly table that is on-going in building its entries. Taking advantage of the access patterns in the table under construction, prefetching can start as soon as possible even though there are few access patterns when new page is accessed for the first time. Moreover, to expand the opportunity in exploiting access patterns, we advance to make a use of entries of other table built for different pages. Especially, by referencing access patterns from the preceding pages, currently accessed page can have more diversified access patterns.

1.3 *Better accuracy metric*

We propose a modified metric in measuring prefetch accuracy by excluding the undetermined prefetches in existing metric. We name the metric as "accuracy ratio", which is the ratio of number of useful prefetches to the sum of number of useful prefetches and number of useless prefetches. The denominator of the existing metric is total number of prefetches which contains the undetermined prefetches in it. So, we construct denominator only with a determined portion, useful prefetches and useless prefetches, in order to well reflect prefetch accuracy.

2 Background

One of the early proposed hardware prefetching techniques is the simplest sequential prefetching, next line prefetching [3]. The next line prefetching is to prefetch a cache line that follows immediately the miss of the cache line. More advanced prefetching methods employ a prediction table by means of detecting memory access patterns. The table is used to record memory access history and identify its pattern. Also, a prefetching method is proposed for constant stride access pattern that refers to a sequence of memory accesses in which the distance of consecutive accesses is constant [4, 5]. The constant stride pattern also appears in pointer-based data structures [6].

In [7, 8], the author is proposed to find the most likely next address for currently accessed addresses by representing probabilistic correlation between accessed addresses with Markov model. This method needs not only a large storage to store the addresses into a table but it also requires high computational cost to calculate the correlation. Furthermore, Markov prefetching has stale data problem in the table.

Global history buffer (GHB) [9] as a FIFO manner is proposed to reduce a storage overhead as well as to solve a stale data problem. GHB holds recent miss addresses in the FIFO buffer and chains the same miss addresses in the buffer. Following the chain, deltas[5] are computed and adjacent deltas form a pair used as a prefetch key.[6]

Recent prefetching methods adopt a learning system since these mechanisms are possible to give a feedback of varying accuracy and coverage, depending on workloads [10]–[12]. In [12], the author dynamically adjusts prefetching aggressiveness in both positive and negative ways through a feedback system, to achieve better performance and bandwidth-efficiency.

Signature Path Prefetcher (SPP) [13] uses confidence value to adjust the length of a signature path to strike a balance between accuracy and coverage. SPP speculatively predicts memory access patterns, based on a 12-bit signature that represents a sequence of memory access. The 12-bit signature is calculated in the fashion of combining consecutive strides between adjacent accessed cache lines. The signature and a subsequent stride form a pair, used as a prefetching key and a prefetch prediction, respectively.

Perceptron-based Prefetch Filtering (PPF) [14] proposes an additional filter layer enhancing SPP as a way to increase prefetch coverage. PPF is a filter that uses a hased perceptron model to evaluate the usefulness of each prefetch generated by SPP, in order to reach better coverage. The perceptron model uses several features such as physical address, cache line, and page address, etc. to train PPF layer.

Recently, many offset prefetching descendants [11, 15, 16, 19]

[1]Prefetch coverage means the number of memory access patterns that are detected by prefetches. The coverage will be discussed in Sec. 2
[2]Virtual table refers an actually generated table for prefetching but it merely needs temporary hardware storage instead of permanent one
[3]Hardware overhead will be discussed in Sec. 4
[4]Incompletely trained table indicates a table of which the training period is in progress.
[5]Delta is a value of address difference between two adjacent addresses
[6]Each prefetch key has a corresponding prefetch prediction. Different prefetch keys may make different predictions.

have been proposed after Sandbox prefetching (SP)[20] was proposed. SP attempts to find a block offset that gives high performance. The offset is chosen from a set, named sandbox, of pre-selected offset candidates, based on calculated accuracy score of each offset candidate during run-time. Best-offset prefetching (BOP) [11] adds timeliness consideration to SP in order to pursue timely prefetching. BOP tests pre-selected offsets with arrived prefetch-requested block to figure out which offsets fit better in terms of latency of the arrival of prefetched block; BOP checks if the base address of previously prefetched block are equal to currently accessed block's address minus offset. If the equality is satisfied, the score for offset will increase. Otherwise, the score will decrease.

In [15], the author attempts better timely prefetching by calculating best offset page-by-page as compared to BOP's global best offset. The offset is calculated by recording access timing of each cache line in terms of the number of cycles. The measured cycles are referred to decide the proper latency of two arbitrary accessed cache lines in the same page. If there are cache lines accessed within the chosen latency, Berti uses burst mode for them.

Instruction Pointer Classifier based Prefetching (IPCP) [17] proposes two cache level, L1 and L2, prefetching at the same time with multiple instruction pointers to speculate different access patterns and cover a wide spectrum of access patterns. IPCP classifies instruction pointers into constant stride, complex stride, and stream. IPCP suggests different prefetching methods based on the type of instruction pointer.

In [18], the author introduces a reinforcement learning algorithm of prefetching to evaluate prefetch quality, pursuing system-aware prefetching. Observing the current memory bandwidth usage, Pythia intends to obtain highly accurate, timely prefetching by correlating program context information such as cache line address, program counter value, etc. to prefetch decision.

2.1 *Useful and Useless prefetch*

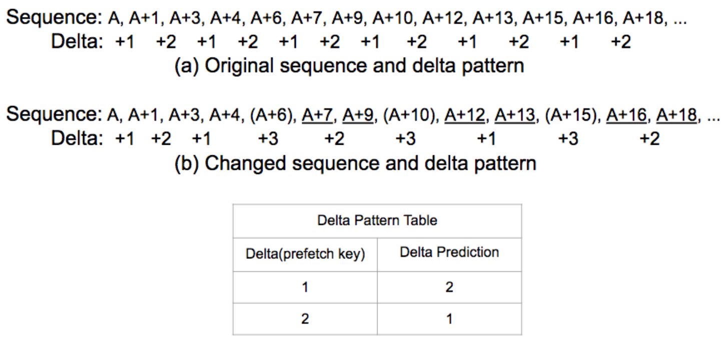
Usefulness of a prefetch is determined by whether a prefetched block is accessed by a program or not. In implementation, a prefetch bit of a block is employed to evaluate the usefulness of the prefetch. The prefetch bit is set when a prefetched block is inserted into cache memory. Then, it is unset when the block is accessed by the program. If the prefetch bit of a prefetched block is unset within the time that the block is evicted, corresponding prefetch will be regarded as a useful prefetch. In other words, the useful prefetch fetches a block that is accessed in the near future so the access occurs before the block is evicted. With the access, the useful prefetch results in eliminating a cache miss. On the contrary, if the prefetch bit remains set, corresponding prefetch will be considered as a useless prefetch. That is, useless prefetch fetches a block that is evicted with no access to it. So, the prefetch wastes cache space and memory bandwidth.

2.2 *Temporal and Spatial Locality*

Locality is that a program exhibits a tendency to reference the same data accessed recently or a data located closely to the recently accessed data. The principle of temporal locality is that recently accessed memory addresses by a program are likely to be accessed again in the near future.[21] The principle of spatial locality is that other nearby memory addresses have a likelihood of being referenced if a memory address is referenced. For example, sequential prefetching takes advantage of spatial locality; the simplest sequential prefetching scheme is to prefetch next cache block, one block lookahead of current access block[3].

2.3 *Physical page contiguity*

An address translation is an essential mechanism that maps a virtual address into a physical address to support virtual memory for modern processors. The translation is page-based operation so each virtual page corresponds to a physical page. The transfer between the two different address space can be a challenge to a hardware prefetcher since the translation can separate two contiguous virtual addresses into two distant addresses in physical address space. Hardware prefetcher has no knowledge of the translation since it is located at the side of a cache [4, 11, 22, 23]. So, prefetching should stop if prefetched data crosses over a page boundary of reference address.



Sequence: A, A+1, A+3, A+4, A+6, A+7, A+9, A+10, A+12, A+13, A+15, A+16, A+18, ...
Delta: +1 +2 +1 +2 +1 +2 +1 +2 +1 +2 +1 +2
(a) Original sequence and delta pattern

Sequence: A, A+1, A+3, A+4, (A+6), A+7, A+9, (A+10), A+12, A+13, (A+15), A+16, A+18, ...
Delta: +1 +2 +1 +3 +2 +3 +1 +3 +2
(b) Changed sequence and delta pattern

Delta Pattern Table	
Delta(prefetch key)	Delta Prediction
1	2
2	1

Figure 1: Effect of Prefetching on Miss Address Stream

2.4 *Miss addresses and Prefetch hit*

Miss addresses have been used for prefetching as a reference of memory access history. The miss addresses can be affected by prefetching; some of cache misses can be removed by useful prefetches. That is, correct prefetching based on miss addresses in the past changes the subsequent miss addresses, resulting in incorrect prefetching unless the prefetching reflects the change in the miss addresses.

Figure 1 is an example that shows the change in a miss address stream by prefetching. We ignore access latency for simplicity. We start with how to construct the Delta Pattern Table from the stream, and then explain about how a prediction from the table changes the miss address stream.

The original sequence shown in Figure 1a starts with a base address of A and shows a Delta pattern of 1 and 2, alternatively. The Delta Pattern Table shown in Figure 1c is constructed based on the Deltas observed in the sequence shown in Figure 1a. It is indexed by Delta (prefetch key) 1 and 2 accordingly. For each Delta, the Delta Prediction is entered. For example, from A to A+1, the table stores Delta 1. The next address is from A+1 to A+3 between which the Delta is 2, so the table stores in entry Delta Prediction 2. In the sequence, Delta 1 is followed by Delta 2. For the table lookup, the entry of Delta with value 1 will find in the corresponding entry, Delta Prediction, a value 2. For example, address A+6 can be predicted from address A+4 by the Delta Pattern Table since the Delta of 1 observed from A+3 to A+4 is matched with the Delta (prefetch key) in the first entry in the table. So, its corresponding Delta Prediction is 2 so the table predicts address A+6 by adding 2 to address A+4. Then, the address A+6 is changed to be a hit address (the parenthesis with A+6 shown in Figure 1b indicates a hit address); the A+6 is no longer used as a reference. Therefore, the Delta, +2 (circled in Figure 1a), is not observed in Figure 1b. Due to the missing Delta, next incoming address A+7 cannot be predicted by the table. Furthermore, next incoming address A+9 cannot be predicted since the observed delta between A+4 and A+7 is 3 that does not match with any Delta (prefetch key) in the table. Likewise, correct prediction of address A+10 disturbs next two incoming addresses, A+12 and A+13, to be predicted by the Delta Pattern Table. In Figure 1b, underlined addresses indicates such addresses.

Reconstruction of the pattern table could be proposed as a solution to above issue due to the changed miss address stream. However, the update of the table cannot be effective in reflecting the miss address patterns since the original sequence of the memory access does not hold the new delta value, 3, in this example sequence. Therefore, it is important to keep prefetch hit addresses to identify patterns from miss address stream. Also, it is necessary to record which address has been prefetched by a prefetcher.

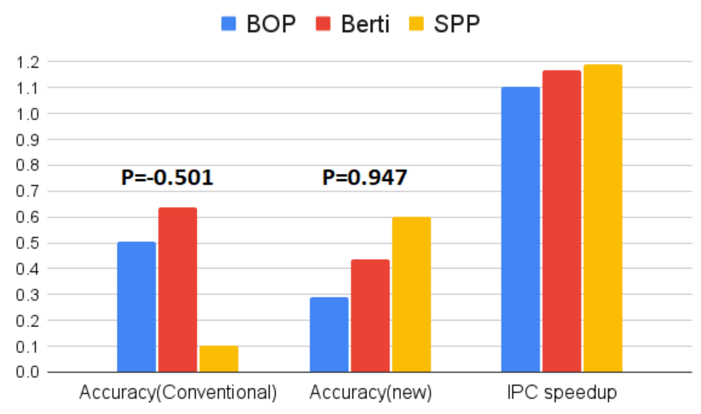


Figure 2: Pearson's Coefficient

2.5 *Evaluation Methodology*

There are two metrics to evaluate the performance of prefetching; prefetch accuracy and prefetch coverage. Prefetch accuracy measures a prefetcher in that how accurately the prefetcher predicts which memory addresses will be accessed in the future; on the other hand, prefetch coverage measures a prefetcher in that how diverse access patterns the prefetcher is capable of detecting against variations of the access patterns.

The accuracy and coverage has a relationship of inherent tradeoff between them. For example, next line prefetcher can make a prediction for simple access pattern having a stride of 1 with a high accuracy but has a limited scope in coverage, meaning that the prefetcher cannot generate diverse predictions other than the stride of 1. On the other hand, if a prefetcher wants to achieve a wide scope in coverage, it should sacrifice its accuracy as a cost. This is, for a wide scope in coverage, a prefetcher should consider a wide change in simple to complex access patterns. So, regarding the broad variation on all the patterns would be more likely to lead a prefetcher to an inaccurate prediction. The imprecise prediction results in useless cache blocks to be fetched so that cache space can be wasted as well as cache pollution and memory bandwidth consumption can increase. Consequently, the inaccuracy of a prefetcher degrades the system performance.

We propose a refined metric for prefetch accuracy in order to make the accuracy metric more relevant to IPC speedup performance. Existing accuracy metric is the ratio of the number of useful prefetches to the total number of issued prefetches[12]. The numerator, the number of useful prefetches, in the metric is directly proportional to the IPC speedup performance since useful prefetches hides cache miss latency. However, the denominator, total number of issued prefetches, shows low correlation to IPC speedup performance. The total number of issued prefetches counts all the prefetches that are issued during only the "current" IPC speedup measure period. Actually, the IPC speedup measured in the current measure period can be affected by the prefetches issued in the "past"(before current) measure period. During the current measure period, the hit(or miss) of the prefetch issued in the past period increases(or decreases) IPC speedup. In other words, the total number of issued prefetches does not count the prefetches that were issued in the past period which, however, impacts the currently measured IPC speedup

Another reason of the low correlation between the total number of issued prefetches and IPC speedup is that the total number of issued prefetches contains the number of prefetches that turn out to be neither useful nor useless by the end of current IPC speedup measure period. They can be determined as either hit or miss "after" the current measure period. Thereby, in the current measure period, we do not have enough information about how those prefetches affects IPC speedup. However, in the conventional metric, such prefetches constitutes a portion of the total number of issued prefetches which is inversely proportional to the IPC speedup.

The low correlation between existing prefetch accuracy metric and IPC speedup is shown in Figure 2 with Pearson's correlation coefficient of -0.501. The value close to 0 means a low correlation while the value close to +1 or -1 indicates a linear correlation. One can see that in Figure 2 the prefetcher SPP has the lowest accuracy in the conventional metric but the IPC speedup of this prefetcher shows the highest IPC speedup, compared to other two prefetchers, BOP and Berti. We run the three prefetchers, BOP, Berti, and SPP, since they show almost the same value in the other metric, called prefetch coverage, in SPEC CPU 2006 benchmark to exclude the effect of prefetch coverage on the IPC speedup. We propose the modified metric, named prefetch accuracy ratio in this paper, as shown in Equation 1.

$$\text{Prefetch Accuracy Ratio} = \frac{\text{Number of Useful Prefetches}}{\text{Number of Useful Prefetches} + \text{Number of Useless Prefetches}} \quad (1)$$

Equation 1 evaluates the accuracy of a prefetcher based on the prefetches evaluated as useful and useless during "current" measure period only. The equation directly compares the useful prefetches and useless prefetches that are detected during the "current" measure time. Therefore, the metric, prefetch accuracy ratio, can show how much the useful prefetches contribute to the performance improvement over a performance degradation from useless prefetches. For example, a prefetch accuracy ratio of 1 means that a prefetcher gives only a positive effect to the system performance by generating only useful prefetches with no useless prefetches. As shown in Figure 2, Pearson's coefficient value of 0.947 shows a high correlation between the prefetch accuracy ratio and IPC speedup, so SPP represents the highest accuracy ratio, leading to the highest IPC speedup.

Another metric, prefetch coverage, is defined as the number of useful prefetches over the number of cache misses with no prefetching; prefetch coverage is proportional to the number of useful prefetches as shown in Equation 2:

$$\text{Prefetch coverage} = \frac{\text{Number of Useful Prefetches}}{\text{Total Number of Cache Misses}} \quad (2)$$

From Equation 1 and 2, we can conclude that the number of useful prefetch can be a key factor to achieve both high accuracy and wide coverage at the same time. The number of useful prefetches would not be proportional to the prefetch accuracy since increase in the number of useful prefetches involves increase in the number of useless prefetches in general. On the other hand, the number of useful prefetches is proportional to prefetch coverage since the change in the number of useful prefetches does not affect total number of cache misses.

Based on aforementioned metrics, we evaluate existing prefetchers and our proposed prefetcher in terms of IPC[7] speedup in section 3 and 4.

3 Design

Figure 3 illustrates the overall structure of our prefetcher, BRP. Both L2 cache miss and prefetch hit addresses trains our prefetcher. BRP sends a request to fill a prefetch into either L2 cache or last-level cache (LLC); LLC is requested to be filled by the prefetch when miss status holding register (MSHR)[8] has no opening because it is full of its items.

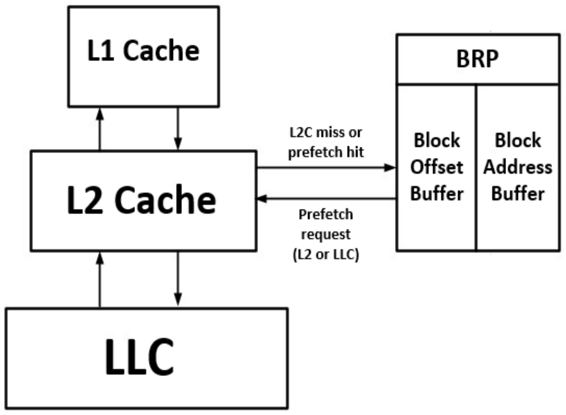


Figure 3: Overall BRP Structure[2]

The BRP module consists of two buffers as a storage; block offset buffer and block address buffer. The block offset buffer stores an offset of each block page-by-page. On the other hand, the block address buffer stores the address of blocks that are prefetched. The two buffers updates their data, according to FIFO manner; the oldest one is evicted first and the latest one is stored as a last item in the buffer, as avoiding stale data problem. In this section, we discuss how the two buffers are employed in reaching moderate accuracy ratio while maximizing prefetch coverage. Another feature of our BRP is that it does not request a prefetch in the case that it traverses a boundary of the page to which referred data belongs as discussed in Section 2.3.

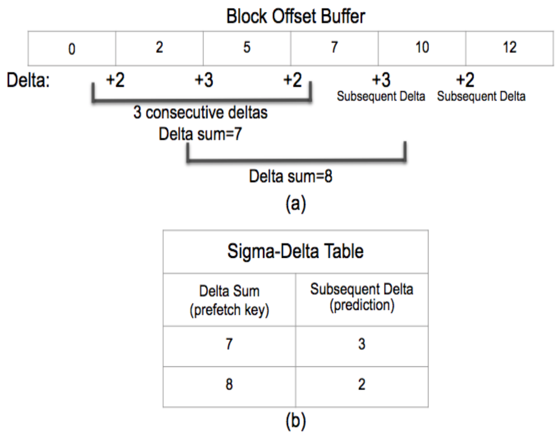


Figure 4: Example of Sigma-Delta Table generation (b) from block offset buffer (a) with delta history length of 3.

3.1 Offset Buffer

Offset buffer stores a block offset of pages that have been accessed recently. Based on the tag attached to the buffer, block offsets can be stored to a offset buffer with the same tag. That is, One of the offset buffers, of which the tag is the same as it of the page of the block accessed, collects the latest offset of the block. If a new page

[7]All IPCs, instruction per cycle, are computed as arithmetic average of the IPC across the benchmarks

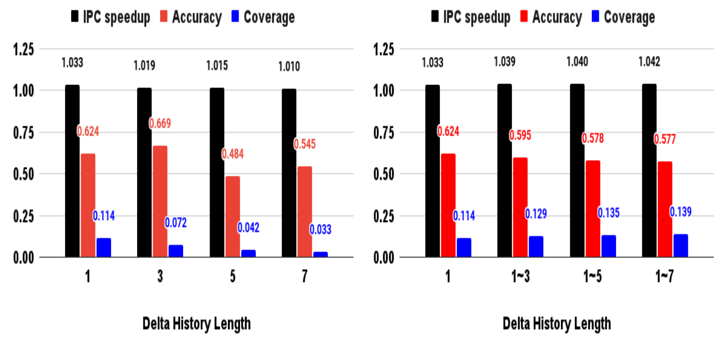
[8]MSHR holds pending load/store accesses that refer to the missing cache

is accessed for the first time, a new tag is generated based on a base address of the page, and a new offset buffer attached to the new tag is also built, loading the offset on it. The storage capacity of the buffer is set by users. If all the buffer is fully occupied by its offsets, the buffer evicts its oldest one and stores the latest offset as a last item in it.

The purpose of collecting block offsets is to observe the pattern of deltas among offsets; the delta is the arithmetic difference between the values of two block offsets that are successively accessed in the same page. [2] suggests a table, named Sigma-Delta Table, to identify the delta pattern from delta sequence as shown in Figure 4. Delta Sum, the sum of number of consecutive deltas, is used as an index; corresponding to it is the subsequent delta, the delta coming next.

Figure 4 represents an example of how a Sigma-Delta Table is generated from an offset buffer containing block offsets of 0, 2, 5, 7, 10 and 12. The example shows a case that three consecutive deltas are added up. The generated delta sequence is 2, 3, and 2 from the offset sequence of 0, 2, 5, and 7 as shown in Figure 4a, of which the sum is 7. The next Delta is 3 (to move from offset 7 to 10). So, the sum of 7 and the subsequent delta of 3 constructs a pair as an entry of the table shown in Figure 4b. In the table, in the case that Delta Sum 7 is given, from the same row and next column under Subsequent Delta, a value of 3 is read out.

In implementation, the number of consecutive deltas is set by users with regard to hardware overhead. In fact, the Sigma-Delta Table is built as a virtual table that does not require a permanent hardware overhead. That is, all the entries in the table is derived from offsets in the buffer only when they are referenced for prefetching. Hardware overhead of the table will be discussed in Sec. 4.



(a) Single Delta History Length (b) Multiple Delta History Length

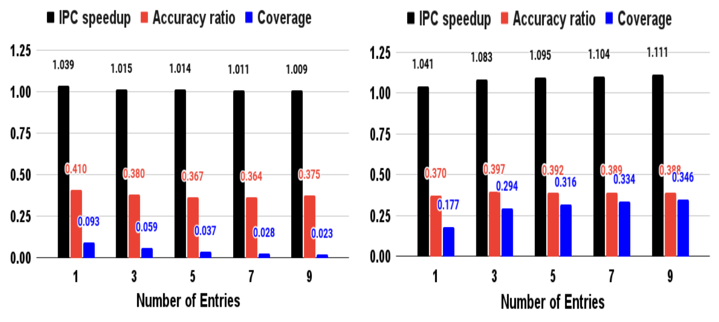
Figure 5: Comparison of delta history length (a) single (b) multiple

3.1.1 Multiple delta history length

As aforementioned, the Sigma-Delta Table is built by pairs of delta sum and subsequent delta which is calculated by deltas. The number of deltas used is equal to the delta history length. [2] suggests a method of using multiple delta history lengths rather than a single delta history length. So, the multiple Sigma-Delta Tables are built according to multiple delta history lengths. For example, if the maximum length of the delta history is set to be 4, four of the Sigma-Delta Tables are built; That is, each table belongs to different delta history length. Different number of consecutive deltas are used to construct the Sigma-Delta Table. All the generated sigma-delta pattern tables are subject to being used for prefetching.

The higher prefetch coverage is achieved by the proposed method as shown in Figure 5b, compared to the method of using single delta history length shown in Figure 5a. IPC speedup shown in Figure 5a gets decreased by the coverage reduction as a single delta history length increases. Compared to the decrease, in Figure 5b, IPC speedup is sustained as more multiple lengths of delta history gets involved. This is because the multiple delta history lengths cause an increase in prefetch coverage that cancels out the IPC speedup decrement caused by accuracy ratio reduction.

Briefly, multiple delta history lengths improves prefetch coverage by generating more useful prefetches, resulting in IPC speedup improvement. Several methods will be discussed to reach further increase in prefetch coverage in the section onward.



(a) Complete trained table only (b) (Complete+Incomplete) trained table

Figure 6: Completely and Incompletely Trained Table (a) complete only (b) complete+incomplete

3.1.2 Complete/Incomplete training

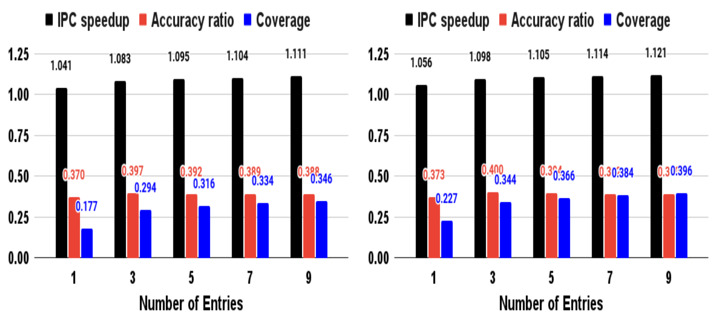
After a Sigma-Delta Table fills all of its entries, training session is started to evaluate the validity of the patterns that the table stores. Surely, we use a Sigma-Delta Table of which the training session is completely finished with its validity for prefetching. In addition, we propose to utilize the Sigma-Delta Table for prefetching while its training session is in progress in order to avoid no prefetch issued during the training session. Adopting the table with on-going training helps increase the scope of prefetch coverage. Figure 6b reflects the effect of the incompletely trained table additionally used, as compared to Figure 6a. Figure 6b exhibits a coverage increase by 2 to 15 times, compared to the coverage shown in Figure 6a, a result from using the completely trained table only. The increased coverage is due to the increase in the number of useful prefetches that a table under its training generates.

Furthermore, in Figure 6a, coverage decreases as the number of entries increases since the increase in the number of entries in the table requires longer training session. As a result, no prefetching duration increases, resulting in decrease in useful prefetches. However, Figure 6b shows the effectiveness of our proposed method especially when more entries are employed. This is because the increased duration of training session for the case of more entries makes incompletely trained tables having longer time to generate more useful prefetches.

According to our design intention, the IPC speedup increases by 1 to 11% due to the increased coverage; IPC speedup is 1.04 to 1.11 in Figure 6b versus 1.03 to 1.01 in Figure 6a. The proposed method is more effective in the case of greater number of entries in a Sigma-Delta Table. One interesting observation in Figure 6b is that the additional use of the table continuing its training provides slight increase in accuracy ratio, as compared to Figure 6a, except the case of single entry of Sigma-Delta Table; 3 to 7% increase across 3 to 9 entries and 10% decrease in single entry. Another interesting observation from Figure 6b is that the case of 3 entries exhibits better results in both accuracy ratio and coverage than the results of single entry.

Training mechanism adopted in this work is to check the validity of collected delta patterns stored in the Sigma-Delta Table. The training is to compare values of entries in the Sigma-Delta Table with subsequently accessed block offsets that update a block offset buffer; to check if the pair of delta sum and delta prediction in the entry is equal to a pair of them calculated from the updated offsets in the offset buffer. If the equality happens, the table will get a hit, so called table hit. Otherwise, the table will get a table miss.

In implementation, we employ a flag to denote whether each Sigma-Delta Table obtains a table hit or not. On the other hand, a table miss score is counted with 2-bit saturating counter. If a table reaches a maximum table miss score, 3 with the 2-bit counter, all the entries in the table will be removed and new entries will, then, be generated by more recent offsets stored in the offset buffer. After training is finished, the table attaining a table hit and having lowest table miss score is used as top priority for prefetching.



(a) Sigma-Delta Table with no on-the-fly (b) Sigma-Delta Table with on-the-fly

Figure 7: Sigma-Delta Table usage (a) no on-the-fly (b) on-the-fly

3.1.3 On-the-fly table

We propose to use the Sigma-Delta Table on the fly to avoid no prefetching issued until the table is full of its entries. That is, we employ the table even under construction to generate a candidate of a prefetch to reach further increase in coverage. The on-the-fly table is effective in increasing coverage since it can generate a prefetch while the table either accumulates its entries or replaces its existing entries with new entries due to reaching a maximum table miss score. Also, the table can generate prefetches sooner based on the few patterns that belong to the new page accessed for the first time. According to our design intention, the IPC speedup increases by 1.3% due to the increased coverage; IPC speedup is 1.055 to 1.126 in Figure 7b versus 1.04 to 1.11 in Figure 7a.

3.1.4 Referring to other pages

Multiple Sigma-Delta Tables are generated to figure out the offset patterns for individual page. In [2], the author proposes referring to those tables so other near pages can reference the offset patterns to issue their prefetches. In other words, each page refers to the offset pattern, the pairs of Delta Sum and a Delta, in the table that has been accessed little earlier. So, the patterns detected from both currently accessed page and earlier accessed pages are utilized for current prefetch issue. Especially, this referring method should be effective in generating diverse prefetches when the page does not have enough identified patterns due to first time access to the page, early stage of building the Sigma-Delta Table, etc.

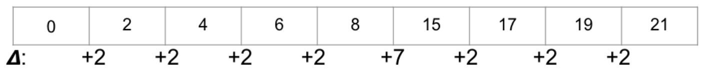


Figure 8: *libq*: Example of Spike Pattern

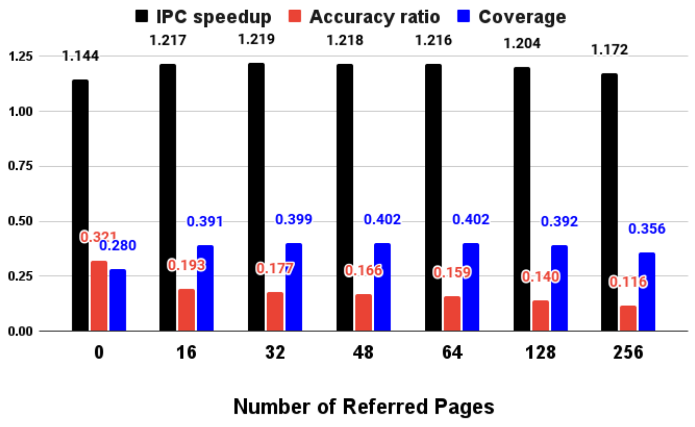


Figure 9: Number of Referred Pages Sensitivity

Furthermore, the referring mechanism gives a chance to mine performance improvement from irregular pattern. For example, spike pattern, the delta sequence with the single delta of +7, shown in Figure 8 from *leslie* benchmark is hard to be predicted due to its minority in patterns. Also, the delta spike can appear irregularly and its magnitude can be variable. Based on locality property among near pages, it is possible that preceding pages go through both the same timing and magnitude of spike pattern. So, we leverage access patterns of the preceding page in generating prefetches for subsequently accessed pages.

Obviously, there should be a cost of an initial miss in capturing the irregular pattern when the pattern shows up for the first time. With the sacrifice of the initial miss, the the pattern can be identified and referred by other near pages that may go through the same access pattern. This method is very effective to performance improvement for some benchmarks such as *calculix*, *lbm*, *leslie*, *libq*, *soplex*, etc. through simulation. In those benchmarks, some of the pages that are accessed sequentially tends to have the same offset patterns.

Compared to other prefetchers, BOP[11] is not successful in identifying the delta spike pattern since the prefetcher focuses only on globally optimal offset, using only delta value of +2 as a majority. GHB[9] could succeed in predicting the pattern as it captures two consecutive deltas as a pair, preserving the sequence of individual delta. By maintaining the delta sequence, GHB constructs pairs of two adjacent deltas such as (+2,+2), (+2,+7), and (+7,+2) that are used as a prefetch key. So, the pair, (+2,+2), is used to predict the spike delta, +7. However, in the case that the magnitude of next delta spike varies, GHB is hard to predict correct spike delta. Furthermore, GHB needs hardware storage in order to store the pairs of two deltas.

The performance improvement from referring to other pages is shown in Figure 9 with different number of referred pages. There is a 39% increase in prefetch coverage between no page referred and 16 pages referred. On the other hand, there is a 40% decrease in accuracy ratio between them. As a result, IPC speedup increases by 6%. On the other hand, coverage, accuracy ratio, and IPC speedup are almost the same among 16 to 128 referred pages. Also, referring to 256 pages degrades IPC speedup by 3.9% due to the decrease in prefetch coverage by 9.3%. Performance degradation occurs in the case that we refer many pages, which contains pages accessed in the distant past. With the observation, 32 pages are set to be referred.

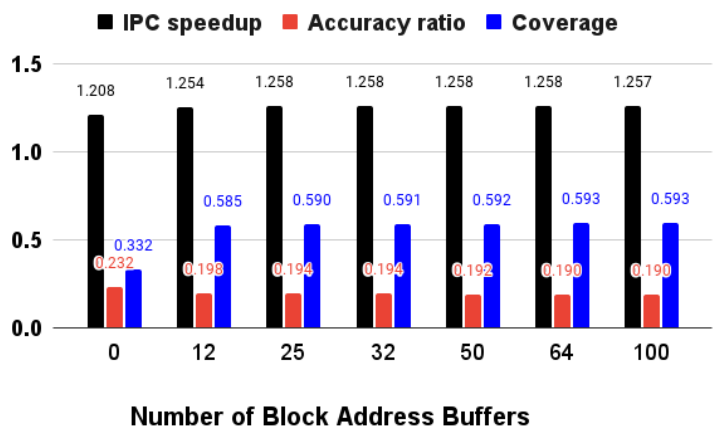


Figure 10: Block Address Buffer Sensitivity

3.2 *Block Address Buffer*

As discussed in Section 2.4, [2] proposes recording memory addresses that have been prefetch requested, including prefetch hit addresses. So, the access pattern can be preserved as the original pattern of the miss addresses. The paper offers a block address buffer to store the history of recent prefetch addresses. The record is used for multi-degree prefetching that indicates issuing multiple prefetchings per prefetch prediction. Additionally, the history helps filter out duplicate prefetches that are issued in the past.

As shown in Figure 10, there is an IPC speedup increase of 3.7% between no prefetch buffer and 12 prefetch buffers, because prefetch coverage increases by 76% while accuracy ratio decreases by 14%. On the other hand, compared to 12 buffers, 32 buffers shows 0.27% increase in IPC speedup. Also, there is no significant IPC speedup improvement with more than 32 buffers. The result represents that an optimal number of buffers is 32 since more than 32 buffers requires more hardware overhead with little return.

4 Performance Evaluation

We evaluate the performance of the prefetcher, BRP, with added techniques mentioned in the previous section. We compare it with other prefetchers' performance in terms of IPC speedup, accuracy ratio, and coverage, in this section.

4.1 *Simulation Methodology*

To evaluate BRP and other prefetchers, a subset of both the SPEC CPU 2006 benchmark[24] and 2017 benchmark[25] is used since the SPEC CPU 2006 benchmark is known as memory-intensive characteristics and SPEC CPU 2017 is developed recently. We use individual thread for a single core simulation and 4-thread mixes randomly selected from the benchmark for 4-core simulation. Simpoints [26], of which the interval is 10M instructions, are used to measure the IPC speedup. We simulate 200M instructions as a warmup and then measure the performance with the following 200M instructions. ChampSim simulator[27] is used for the evaluation. The framework of the Champsim simulator is the 3rd Data Prefetch Championship.

Table 1: Processor Configuration

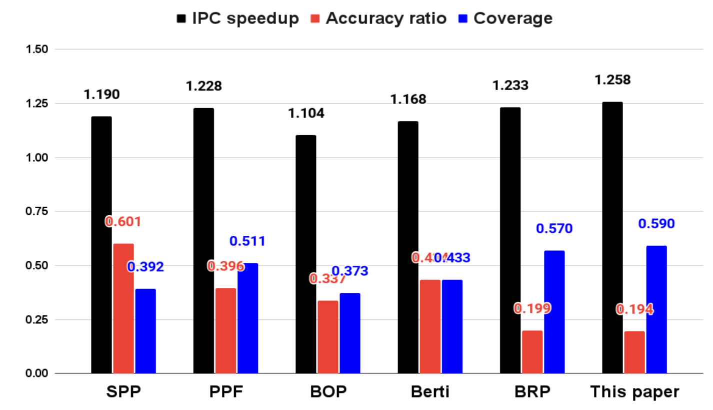
Core Parameters	1-4 Cores, OoO, 4GHz, 256 entry ROB, 4-wide
Branch Predictor	16K entry bimodal, 20 cycle misprediction penalty
Private L1 Dcache	32KB, 8-way, 8 MSHRs, LRU, 4 cycles
Private L2 Cache	256KB 8-way, 16 MSHRs, LRU 8 cycles, Non-inclusive
Shared L3 (LLC)	2MB/core, 16-way, LRU 12 cycles, Non-inclusive
Main Memory	4GB, 1-2 64 bits channels, 8 ranks/channel, 8 bank/rank, 1600MT/s

Table 1 summarizes the simulation configuration for the testing systems. The CPU is clocking at 4 GHz clock rates per core with an out-of-order scheduler. Each CPU core has its private cache; L1 and L2 cache. All the four cores shares a single L3 cache. The L1 cache is divided into instruction and data cache, of which size is 32 KB. L2 cache size is 256KB. The block size and page size of the cache is 64B and 4KB, respectively. In the single core simulation, single DRAM channel is adapted. In the 4-core simulation, two

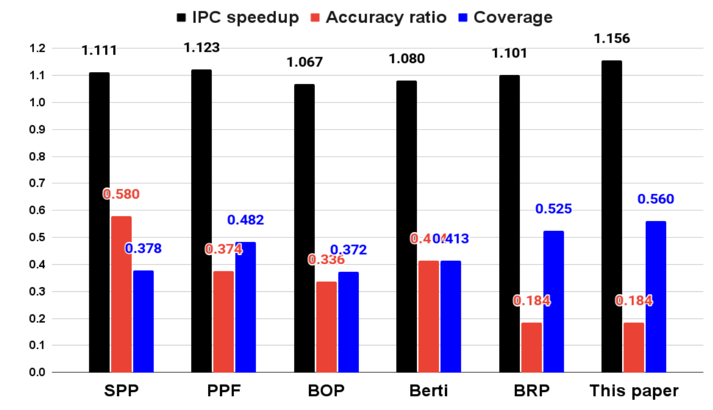
DRAM channels are adapted. Also, L1 and L2 cache contain 8 and 16 prefetch queues, respectively.

L1 and L2 cache have 8 and 16 miss MSHRs, respectively. The prefetch queue temporarily holds the block address which is requested for prefetching until it is issued. The MSHR and prefetch queue helps CPU continue to execute following instructions of a program since they store pending cache misses and prefetches. Only L2 cache access initiates the prefetcher and no prefetcher exists in other cache levels. All the prefetched data are stored into either L2 cache or L3 cache.

The performance of this work is compared with the origin work, BRP, as well as prefetchers such as SPP, PPF, BOP, and Berti, which are discussed in section 2. Those prefetchers are compared in terms of following three metrics; IPC speedup, prefetch accuracy ratio, and prefetch coverage. We use the original code of these prefetchers submitted to DPC-2 and DPC-3. We follows BRP's configuration which is set as follows: 1) Delta history length up to 3; length of 1, 2, and 3. 2) Maximum entries of 3 in sigma-delta table. 3) Block offset buffers for 512 pages. 4) Referring 32 pages. 5) 25 block address buffers.



(a) SPEC 2006: The impact of prefetch accuracy ratio and coverage on IPC speedup

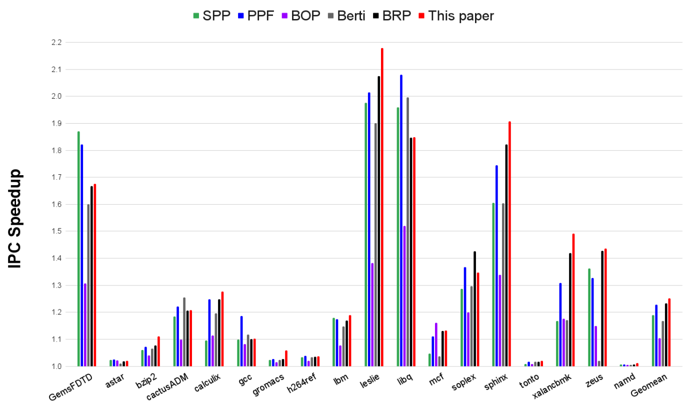


(b) SPEC 2017: The impact of prefetch accuracy ratio and coverage on IPC speedup

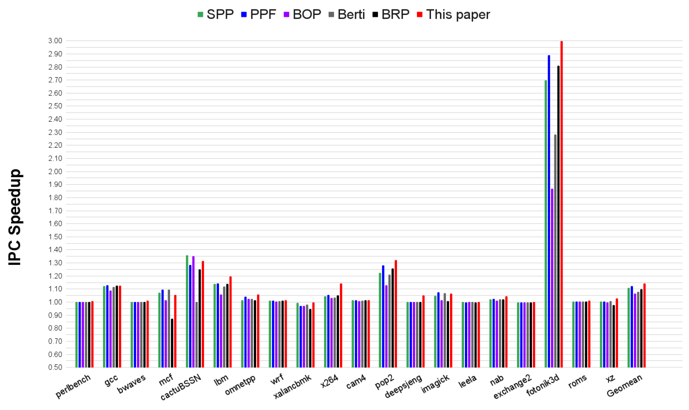
Figure 11: Single-core IPC speedup

4.2 Single Core Performance

Figure 11 shows the average IPC speedup of all the prefetchers with both SPEC 2006 and 2017 benchmark. The IPC speedup is a normalized IPC value to the IPC value of no prefetching baseline. Our work outperforms all the other prefetchers with the best average IPC speedup of 1.257 and 1.143 on SPEC CPU 2006 and 2017 benchmark, respectively, due to the widest prefetch coverage. As shown in Figure 11a, in SPEC CPU 2006 benchmark, our work exhibits 58.2%, 50.3%, 36.3%, 15.4%, and 3.5% improvement in coverage over BOP, SPP, Berti, PPF, and BRP, respectively, as our work shows the lowest accuracy ratio according to the design intention. So, our work achieves 25.7% IPC speedup improvement, which is 15.3%, 8.9%, 6.7%, 2.9%, and 2.5% more than the BOP, Berti, SPP, PPF, and BPR, respectively. Also, in SPEC CPU 2017 benchmark, as shown in Figure 11b, our work exhibits 50.5%, 47.9%, 35.6%, 16%, and 6.5% improvement in coverage over BOP, SPP, Berti, PPF, and BRP, respectively. So, our work achieves 14.3% IPC speedup improvement, which is 7.7%, 6.4%, 3.2%, 2.7%, and 4.5% more than the BOP, Berti, SPP, PPF, and BPR, respectively.



(a) SPEC 2006: IPC speedup by individual benchmark



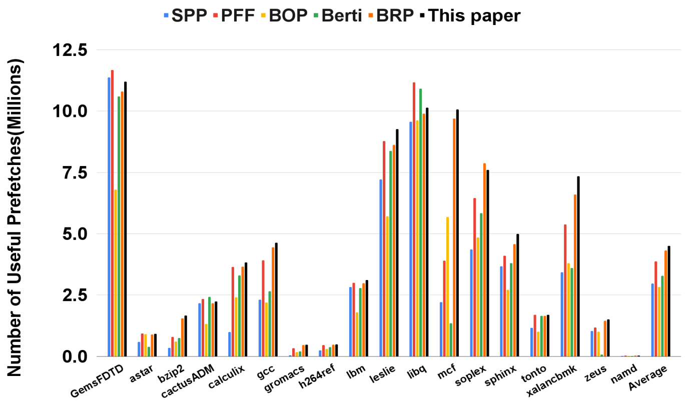
(b) SPEC 2017: IPC speedup by individual benchmark

Figure 12: IPC speedup by Benchmark

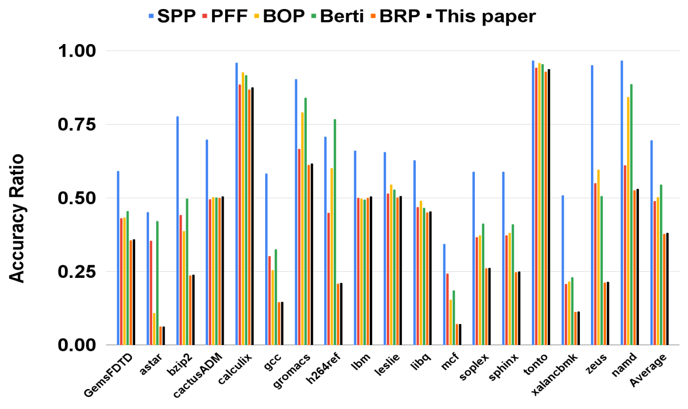
Figure 12 represents IPC speedup by individual benchmark. In SPEC CPU 2006(Fig. 12a), our work shows significant IPC improvement from benchmarks such as *GemsFDTD*, *leslie*, *libq*, *so-*

plex, *sphinx*, *xalancbmk*, and *zeus*, as compared to no prefetching baseline. This is because our work achieves high coverage from them by generating large number of useful prefetches as shown in Figure 13a. Especially, our work achieves the best IPC speedup in 10 applications such as *calculix*, *gromacs*, *leslie*, *soplex*, etc., compared to other prefetchers. As compared to the original work, BRP, our work shows better IPC speedup across nearly all the benchmarks. For example, our work exhibits 5%, 8%, and 7% IPC speedup improvement in *leslie*, *sphinx*, and *xalancbmk*, respectively.

In SPEC CPU 2017(Fig. 12b), our work shows outstanding IPC improvement from applications such as *cactuBSSN*, *pop2*, and *fotonik3d*, as compared to no prefetching baseline. This is because our work achieves high coverage from them by generating large number of useful prefetches as shown in Figure 14a. Especially, our work achieves the highest IPC speedup in 16 out of 20 benchmarks. As compared to BRP, our work also shows better IPC speedup across nearly all the benchmarks; especially, our work exhibits 20%, 8%, and 6% IPC speedup increase in *mcf*, *x264*, and *fotonik3d*, respectively.



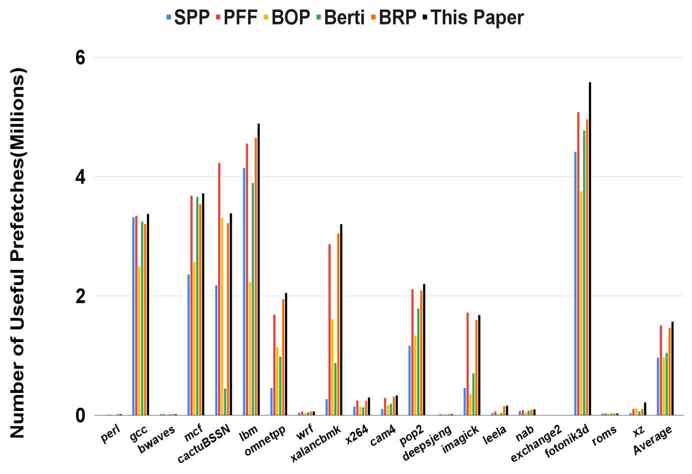
(a) SPEC CPU 2006: The number of useful prefetches by Individual Benchmark



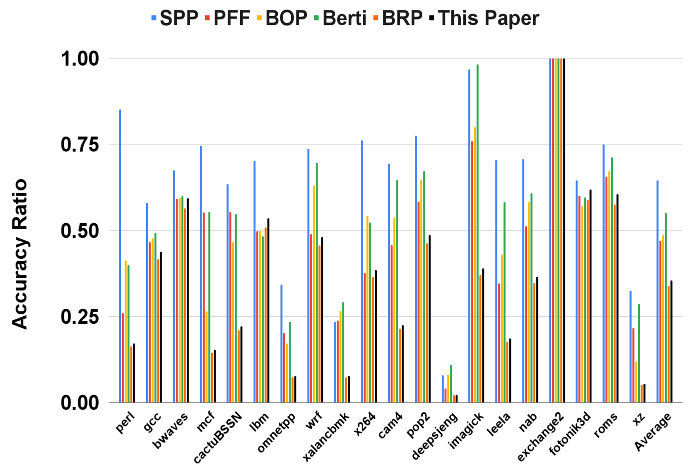
(b) SPEC CPU 2006: Accuracy Ratio by Individual Benchmark

Figure 13: Coverage and Accuracy Ratio with SPEC CPU 2006

As shown in SPEC CPU 2006(Fig. 13a), our work generates the largest amount of useful prefetches in the applications such as *bzip2*, *calculix*, *gromacs*, *lbm*, *leslie*, *sphinx*, *xalancbmk*, and *zeus* so that we have the highest IPC speedup. For those applications, high coverage is a key to predict their patterns, which would be diverse and irregular. On the other hand, in those applications such as *gcc* and *mcf*, our work generates the largest number of useful prefetches, which brings the highest coverage on them, but our work does not reach the highest IPC speedup. This is because that the comparable accuracy ratio would be also required to achieve better IPC speedup.



(a) SPEC CPU 2017: The number of useful prefetches by Individual Benchmark



(b) SPEC CPU 2017: Accuracy Ratio by Individual Benchmark

Figure 14: Coverage and Accuracy Ratio with SPEC CPU 2017

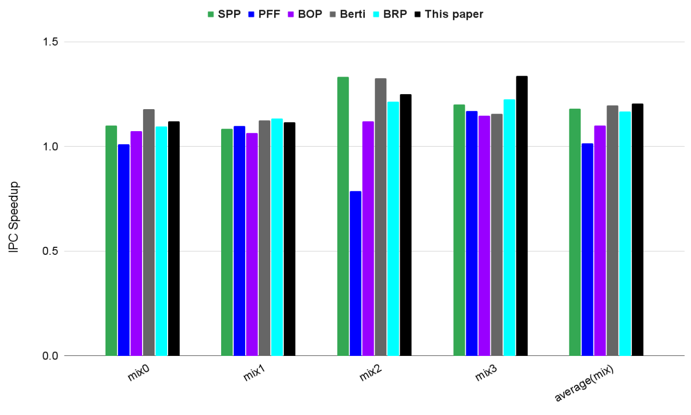
As shown in SPEC CPU 2017(Fig. 14a), our work generates the largest amount of useful prefetches in the applications such as *lbm*, *omnetpp*, *xalancbmk*, *x264*, *pop2*, *deepsjeng*, *nab*, *fotonik3d*, and *xz* so that we have the highest IPC speedup on them, accordingly. For those applications, high coverage is a key factor to achieve the high IPC speedup because they would have diverse and irregular patterns. On the contrary, in those benchmarks such as *mcf* and *camp4*, our work does not achieve the highest IPC speedup even though the highest coverage is reached with the largest number of useful prefetches. This is because the enough accuracy ratio would be also involved.

Table 2: SPEC CPU 2006: Multi Programmed Workloads

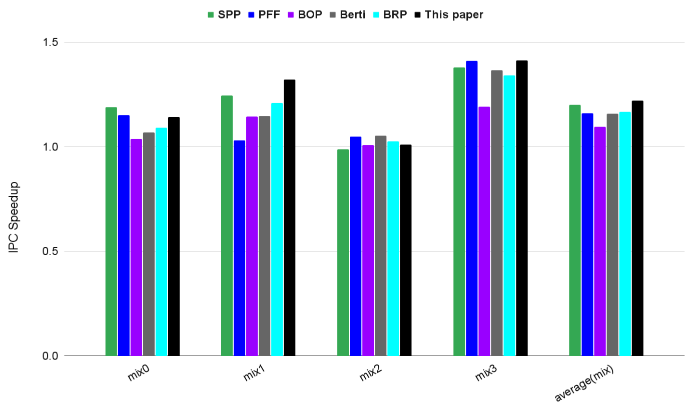
mix0	GemsFDTD, astar, bzip2, cactusADM
mix1	calculix, gcc, gromacs, h264ref
mix2	lbm, leslie3d, mcf, soplex
mix3	sphinx, tonto, xalancbmk, zeus

Table 3: SPEC CPU 2017: Multi Programmed Workloads

mix0	perlbench, gcc, bwaves, mcf
mix1	cactuBSSN, lbm, omnetpp, wrf
mix2	xalancbmk, x264, cam4, pop2
mix3	exchange2, fontonik3d, roms, xz



(a) SPEC CPU 2006



(b) SPEC CPU 2017

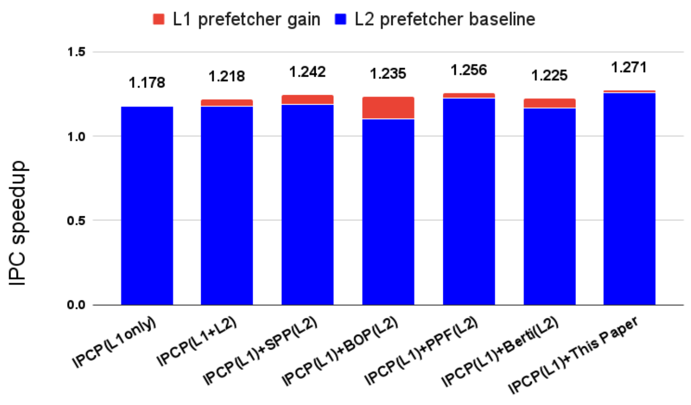
Figure 15: 4-core IPC speedup

4.3 Multi-Core Performance

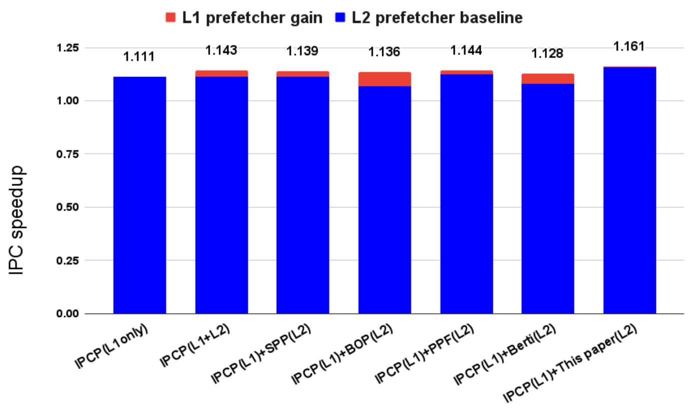
We generate 4 multi-programmed mixes, each consists of 4 traces as shown in the two tables; table 2 and table 3. Each trace in the mix is assigned to a different core for the multi-core simulation.

In SPEC CPU 2006(Fig. 15a), our work accomplishes 20.6% geometric mean IPC speedup increment across 4 mix workloads, compared to no prefetching baseline. Especially, our work exhibits 33.8% IPC speedup improvement in mix3 over the baseline, since our work is effective to appllications such as *sphinx*, *xalancbmk* and *zeus* in 4-core configuration as well. The performance improvement of our work is the best among all the prefetchers. our work surpasses the second highest one, Berti, by a 1.0% IPC speedup. Compared to single core test, in multi-core test, the IPC speedup improvement of our work over Berti reduces by 7.9% since the our work issues more prefetches than berti, throttling LLC and DRAM bandwidth.

In SPEC CPU 2017(Fig. 15b), our work accomplishes 22% geometric mean IPC speedup increase over no prefetching baseline across 4 mix workloads, which is the best performance improvement among all the prefetchers. Especially, our work shows 41.2% IPC speedup improvement in mix3 over the baseline, since our work is effective to appllications such as *fontonik3d* and *xxz* in 4-core configuration as well. Our work outperforms the second highest one, SPP, by a 2.1% IPC speedup. Compared to single core test, in multi-core test, the IPC speedup improvement of our work over SPP reduces by 1.2% since the aggressive prefetching of our work throttles LLC and DRAM bandwidth more in the multi-core configuration by generating both high number of useful and useless prefetches.



(a) SPEC CPU 2006



(b) SPEC CPU 2017

Figure 16: Single-Core: L1$ and L2$ prefetching IPC speedup

4.4 *Multi-level prefetching performance*

We added an L1 cache prefetcher to the test configuration to compare the performance of the prefetchers with IPCP which proposes multi-level prefetching, by an L1 prefetcher and an L2 prefetcher at the same time. We combine the IPCP L1 prefetcher with other proposed prefetchers acting as L2 prefetchers to conduct the test.

The L1 prefetcher of IPCP provides 17.8% and 11.1% IPC speedup in SPEC CPU 2006 and 2017, respectively, as compared to no prefetching. As shown in Figure 16a, our work shows the highest IPC speedup with an additional of 0.093% IPC speedup made by the L1 prefetcher in SPEC CPU 2006 benchmark; Our work outperforms the second highest, PPF, by 1.2% IPC speedup. In this test, BOP receives the maximum benefit of 0.131% IPC speedup, an increase from 1.104% to 1.235%, due to the IPCP L1 prefetcher. As shown in Figure 16b, in conjunction with the IPCP L1 prefetcher, our work also achieves the highest IPC speedup, surpassing the second highest, IPCP(L1+L2), by 0.018% in the IPC speedup. Please note, the L1 prefetcher gain here is only 0.004%, of the lowest amount.

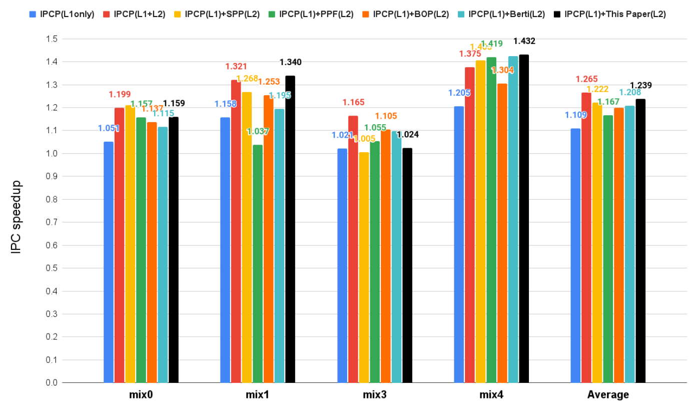


Figure 17: Multi-Core: L1$ and L2$ prefetching IPC speedup

In 4-core simulation, the two level IPCP prefetcher reached the hightest IPC speedup in SPEC CPU 2017, with 1.265 in average. As indicated in Figure 17, our work achieved the second highest performance in average, with an IPC speedup of 1.239, surpassed by the IPCP(L1+L2) only by 0.026%.

Table 4: Prefetcher Storage Overheads

Buffer-referred Prefetching(BRP)	5.512KB
Signature Path Prefetching(SPP)	5.507KB
Perceptron-Based Prefetch Filtering(PPF)	39.34KB
Best-Offset Prefetching(BOP)	1.85KB
Best-Request-Time Prefetcher(Berti)	22.1KB

4.5 *Storage Overhead And Performance Contribution*

The total hardware storage of our work is equal to the hardware overhead of the original work, BRP, of which the overhead is 5.512KB, with each individual component shown in Table 5. Our work does not increase the hardware overhead from the original work. There is no extra hardware overhead required with the introduced mechanism of employing incompletely trained Sigma-Delta Tables for generating prefetch candidates. Furthermore, we utilize the original Sigma-Delta Table under construction so no additional hardware is needed. The component with the largest overhead is block offset buffer (3.32KB). The buffer stores multiple 6-bit offsets per page. Berti holds heavy hardware overhead of 22.1KB, because it pursues an increase in both prefetch coverage and accuracy at the same time. Especially, the heavy overhead of Berti is caused by adapting diverse features such as instruction pointer, recorded page table, etc. in order to ensure prefetch accuracy. As shown in Table 5, BOP has the lowest overhead since its storage contains a few cache lines that has been requested recently. However, BOP derives low performance since the storage is employed to find out global best offset value as a simple manner.

Table 5: BRP Storage Overhead

Structure	Quantity	Component	Storage
Block Offset buffer	512	Offsets(6bit), Tag(16bit), lru(9bit), burst mode(2bit), aggressiveness(2bit),	33280 bits
Sigma-delta table accessories	1536	history length(2bit), valid(1bit), miss count(2bit), hit flag(1bit)	9216 bits
Block Address buffer	25	address(64bit)	1600 bits
Total = 33280 + 9216 + 1600 = 44096 bits = 5.512KB			

With the virtual Sigma-Delta Tables, nearly 45% of hardware storage is saved against using real Sigma-Delta Tables in total storage overhead. Hardware storage is allocated for storing block offsets in the buffers but then the table is virtually generated only when they are used, instead of giving permanent hardware storage for the real tables. The block offset buffers only except its accessories occupy 2.304KB; 512 buffers exist(each for a single page), each buffer has 6 block offsets, and each offset is 6-bit long. On the other hand, when the real Sigma-Delta Table is adapted with assigned constant storage, the total hardware overhead occupies 6.912KB; 3 tables exists per page (total 512 pages), there are 3 entries per table and each entry is of 12 bits (6 bits for delta sum and the other 6 bits for delta prediction). About 67% storage is reduced in terms of constructing pattern tables for prefetching, and about 45% hardware overhead decreases in terms of total hardware overhead.

5 Conclusion

In this paper, we have shown an extended work of Buffer-referred Prefetching, achieving high performance gain with low hardware overhead. First of all, it provides techniques that increase the prefetch coverage with no additional hardware overhead, compared to the original work. With such techniques, our work helps bring out lots of useful prefetches from Sigma-Delta Tables under construction even though the table is not complete in its training. The technique is effective to the cases that few access patterns generated due to new page access, pattern transfer within the same page, and irregular patterns. This paper also offers a refined metric, called accuracy ratio, for measuring prefetch accuracy in order to directly take into account both performance improvement and degradation from a prefetcher. Future work would be to add a perceptive filter as a replacement of the block address buffer to achieve better prefetch filtering or well-tuned aggressive prefetching. Our work accomplishes 25.7% and 20.6% IPC speedup improvement over no prefetching baseline with single- and 4-core configuration, respectively, in SPEC CPU 2006 benchmark. In SPEC CPU 2017 benchmark, our work reaches 14.3% and 22.1% IPC speedup increase over no prefetching baseline with single- and 4-core configuration, respectively. Compared to the original work, we have 2.5% and 3.8% IPC speedup increment with single- and 4-core configuration, respectively, in SPEC CPU 2006. In SPEC CPU 2017, our work achieves 4.5% and 5.5% IPC speedup improvement with single- and 4-core configuration, respectively, over the original work. Our work outperforms the 2nd best prefetcher, PPF, by 2.9% and 2.7% IPC speedup with single- and 4-core configuration, respectively, in SPEC CPU 2006. In SPEC CPU 2017, our work surpasses both Berti by 1% and SPP by 2.1% IPC speedup with 4-core configuration in SPEC CPU 2017.

References

[1] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News, **23**(1),20–24, 1995.

[2] J. So and M. Lu, Buffer-referred Data Prefetching: An Effective Approach to Coverage-Driven Prefetching, 2021 26th International Conference on Automation and Computing (ICAC), 2021, 1-6, doi: 10.23919/ICAC50006.2021.9594139.

[3] A. J. Smith. Sequential program prefetching in memory hierarchies. In IEEE Transactions on Computers, **11**, 7–21, 1978.

[4] J. W. C. Fu, J. H. Patel,and B. L. Janssens. Stride directed prefetching in scalar processors. In Wen-mei W. Hwu, editor, Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, USA, November 1992, pages 102–110. ACM / IEEE Computer Society, 1992.

[5] J.Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In Joanne L. Martin, editor, Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991, pages 176–186. ACM, 1991.

[6] F. Dahlgren and Per Stenstroom. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture (HPCA 1995), Raleigh, North Carolina, USA, January 22-25, 1995, 68–77. IEEE Computer Society, 1995.

[7] D. Joseph and D. Grunwald. Prefetching using markov predictors. In Proceedings of the 24th Annual International Symposium on Computer Architecture, ISCA '97, pages 252–263, New York, NY, USA, 1997. ACM.

[8] P. Pathak, M. Sarwar, and S. Sohoni. Memory prefetching using adaptive stream detection. In Proceeding of 2nd Annual Conference on Theoretical and Applied Computer Science 5 November 2010.

[9] K. J. Nesbit and J. E. Smith.Data cache prefetching using a global history buffer. In In Proceedings of the International Symposium on High-Performance Computer Architecture, 2004.

[10] I. Hur and C.Lin. Memory prefetching using adaptive stream detection. In 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA, 397–408. IEEE Computer Society, 2006.

[11] P. Michaud. Best-offset hardware prefetching. In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 469–480, 2016.

[12] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In 13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA, pages 63–74. IEEE Computer Society, 2007.

[13] J. Kim, S. H. Pugsley, P. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In The 49th Annual IEEE/ACM International Symposium on Microarchitecture, 1-6, 2016.

[14] E. Bhatia, G. Chacon, S. H. Pugsley, E. Teran, P. Gratz, and D. A. Jimenez. Perceptron-based prefetch filtering. In in Proceedings of the 46th International Symposium on Computer Architecture, 2019.

[15] A. Ros. Berti: A per-page best-request-time delta prefetcher. In 3rd Data Prefetching Championship, 2019.

[16] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran,and H. Sarbazi-Azad. Multi-lookahead offset prefetching. The Third Data Prefetching Championship, 2019.

[17] S. Pakalapati and B. Panda, Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching, 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 118–131, 2020

[18] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, Pythia: A customizable hardware prefetching framework using online reinforcement learning, 54th Annual IEEE/ACM International Symposium on Microarchitecture, 1121–1137, 2021

[19] D.Yadav and C. Paikara. Arsenal of hardware prefetchers. Computing Research Repository, abs/1911.10349, 2019.

[20] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. F. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian. Sandbox prefetching: safe runtime evaluation of aggressive prefetchers. In 2014 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2014.

[21] S. Mittal. A survey of recent prefetching techniques for processor caches. ACM Comput. Surv., 49(2):35:1–35:35, 2016.

[22] S. Somogyi, T. F. Wenisch, M. Ferdman, and B. Falsafi. Spatial memory streaming. J. Instr. Level Parallelism, 13, 2011.

[23] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In Milos Prvulovic, editor, Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015, pages 141–152. ACM, 2015.

[24] SPEC CPU 2006.https://www.spec.org/cpu2006.

[25] SPEC CPU 2017.https://www.spec.org/cpu2017.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In Kourosh Gharachorloo and David A. Wood, editors, Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, USA, October 5-9, 2002, pages 45–57, 2002.

[27] DPC-3.The champsim simulator. https://github.com/ChampSim/ChampSim.