applied sciences

MDPI

Article

Adaptive Granularity Based Last-Level Cache Prefetching Method with eDRAM Prefetch Buffer for Graph Processing Applications

Sae-Gyeol Choi †, Jeong-Geun Kim † and Shin-Dug Kim *

Department of Computer Science, Yonsei University, Seoul 03722, Korea; saegyeol@yonsei.ac.kr (S.-G.C.); junggeun@yonsei.ac.kr (J.-G.K.)

* Correspondence: sdkim@yonsei.ac.kr

† These authors contributed equally to this work.

Abstract: The emergence of big data processing and machine learning has triggered the exponential growth of the working set sizes of applications. In addition, several modern applications are memory intensive with irregular memory access patterns. Therefore, we propose the concept of adaptive granularities to develop a prefetching methodology for analyzing memory access patterns based on a wider granularity concept that entails both cache lines and page granularity. The proposed prefetching module resides in the last-level cache (LLC) to handle the large working set of memory-intensive workloads. Additionally, to support memory access streams with variable intervals, we introduced an embedded-DRAM based LLC prefetch buffer that consists of three granularity-based prefetch engines and an access history table. By adaptively changing the granularity window for analyzing memory streams, the proposed model can swiftly and appropriately determine the stride of memory addresses to generate hidden delta chains from irregular memory access patterns. The proposed model achieves 18% and 15% improvements in terms of energy consumption and execution time compared to global history buffer and best offset prefetchers, respectively. In addition, our model reduced the total execution time and energy consumption by approximately 6% and 2.3%, compared to those of the Markov prefetcher and variable-length delta prefetcher.

Keywords: computer architecture; memory architecture; memory management; cache memory; prefetching; high-performance computing

Citation: Choi, S.-G.; Kim, J.-G.; Kim, S.-D. Adaptive Granularity Based Last-Level Cache Prefetching Method with eDRAM Prefetch Buffer for Graph Processing Applications. *Appl. Sci.* **2021**, *11*, 991. https://doi.org/10.3390/app11030991

Received: 30 October 2020
Accepted: 12 January 2021
Published: 22 January 2021

1. Introduction

The critical performance bottleneck caused by main memory latency has been a major limitation to modern computing architectures. Although we cannot exactly predict the memory access pattern, a prefetching method is key to reducing the average memory latency. By contrast, the generation of inaccurate prefetch candidates can cause cache pollution, which has a negative impact on the performance of the entire system. To accurately exploit memory access patterns, previous studies have suggested several methods for analyzing regular or irregular patterns using memory address history-based heuristic algorithms [1–12].

However, with the rapid growth of artificial intelligence and big data processing, it has become necessary for modern applications to process a large amount of data and possess low latency memory access. In addition, modern applications show more irregular memory access patterns than preceding ones. Therefore, we need to analyze memory access patterns in more detail to generate accurate prefetching candidates.

When we place a prefetcher at a high-level memory hierarchy such as an L1 or L2 cache, we can exploit more program contexts related to memory access patterns. Furthermore, we can analyze access patterns that are relatively similar to the overall behavior of the application. Therefore, most previous studies have suggested prefetching methods

targeting the L1 or L2 cache [1–5,11,12]. When prefetching is applied at the L1 cache, we analyze the overall access pattern of the application and apply memory access patterns that have not been filtered yet; such unfiltered patterns seem to be more regular than those that are passed through the cache layers.

However, the L1 cache has a smaller capacity than the other layers. Therefore, inaccurate prefetching can lead to cache pollution arising from the placement of useless data into the cache storage. Consequently, most studies are focus on prefetching data from the L2 cache, which has a larger capacity than the L1 cache. However, recent applications require a large cache/memory capacity, and the corresponding workloads are memory intensive. As the sizes of the working sets of the workloads increase, it becomes difficult to execute all memory accesses using only L1 and L2 caches. In other words, prefetching at the lower level (e.g., the last-level cache (LLC)) of the memory hierarchy is essential for processing large volumes of data.

However, in the case of a multi-core environment, system memory requests through the shared last-level cache show extremely irregular accessing patterns. In these environments, it is difficult to predict the memory access patterns accurately. These inaccurate predictions can lead to a decrease in the system performance by placing incorrect prefetching candidates in the target cache layer. To resolve these problems, in this paper, we propose a prefetching method that analyzes irregular memory access patterns by introducing adaptive prefetching granularities, i.e., line size, correlated line, and page granularities.

In most cases, prefetchers have different criteria for generating prefetch candidates. By employing more conservative criteria, we can expect a higher prefetching accuracy. However, sacrificing the coverage ratio to achieve increased accuracy can lead to a lower system performance. Hence, we need to find an optimal balance between accuracy and coverage ratio. To this end, we propose an analysis method that splits the memory access streams in detail using three different prefetching granularities. In addition, this method performs prefetching to various degrees with three modules. The first module analyzes memory access streams at each cache layer. In general, regular patterns that are difficult to detect through a common cache line granularity can be identified via memory access granularities of varied sizes.

The correlated line granularity consists of eight cache lines in our model. Hence, the occurrence of memory access streams with wide strides that exceed the size of the correlated granularity is an issue that should be addressed. To this end, our model analyzes such streams based on the page granularity using the second module. When three memory requests from the last-level cache occur for the same memory page, we conduct a pattern matching process for deltas among the delta histories stored in the history table. If the same delta pattern exists in the history table, our prefetching model generates prefetching candidates and sends prefetching requests to the LLC. These steps of triggering a particular prefetching operation make the first and second modules more conservative.

As a result, the two modules will be expected to yield a higher prefetch accuracy ratio but a lower coverage ratio. To cover a wider memory stream range, our model performs prefetch operations more aggressively using the third module. However, pursuing a high prefetching coverage ratio without generating the proper candidates negatively impacts the overall system performance. To maintain a high prefetching accuracy, our model changes its prefetching strategy more aggressively when memory accesses are requested across different correlated lines within the same page granularity. To conduct an aggressive prefetching, even if only one delta exists on any page, a prefetch operation is triggered by the analyzed correlation in the history table.

To evaluate our proposed model, we implement an in-house simulator that supports the complete cache hierarchy and main memory layers. The simulator uses the memory traces generated by Intel's Pin framework. We configure our prefetch buffer using embedded DRAM (eDRAM) [13], which is a promising technology that fills in the gap between on-core and off-core memory latency. The prefetch buffer is placed between the LLC and DRAM main memory. The prefetched data will be placed from the DRAM main memory

to the eDRAM-based prefetch buffer. A comparative analysis was conducted regarding the execution time and energy consumption with other state-of-the-art prefetchers, and other system configurations were evaluated in the same environment. As a result, our proposed model achieved an average improvement in the execution time of 15% compared to both the global history buffer (GHB) [1] and best-offset (BO) prefetchers [2,3] and 6% and 2% compared to the Markov prefetcher and variable-length delta prefetcher (VLDP), respectively. In addition, our proposed model reduced the energy consumption by approximately 18% compared to both the GHB and BO prefetchers, and by 6% and 2.3% compared to the Markov prefetcher and VLDP, respectively.

The rest of this paper is organized as follows. The background and related studies are described in Section 2. In Section 3, the details of our proposed model and its internal architecture and operation methodologies are elucidated. Furthermore, in Section 4, evaluation results and an analysis based on a simulation are described. Finally, Section 5 provides a summary and some concluding remarks regarding this paper.

2. Background and Related Work

2.1. Cache and Memory Prefetching

Hardware prefetching is a well-studied field, and various methodologies have been suggested by many computer architects. Among them, some basic ideas have had a particular influence on numerous prefetching algorithms, which still fit well to some workloads [14]. John et al. [4] proposed a stride-based prefetching method. Two types of approaches are used to seek stride patterns using instruction program counters and physical memory addresses. To find stride patterns from various memory access streams, this method records recent deltas or addresses [6] (a difference of an address value between two consecutive demand memory requests). If the same stride pattern consecutively occurs, the prefetching operation is triggered by adding the stride to the recent memory request address.

Joseph et al. [15] proposed a Markov predictor-based prefetching method. It analyzes correlations between physical memory addresses using an address history table. When a memory miss occurs, the address is recorded sequentially to observe the correlation. If the prefetcher finds the same memory access pattern in the history table, the next memory address in the list will be prefetched. This method is effective for covering temporal irregular patterns; however, this method has a disadvantage of requiring a large storage size for storing chains of memory requests. In addition, Srinath et al. [5] proposed a locality-based prefetcher. Most memory access patterns in applications do not show fixed strides. Some applications concurrently generate multiple memory streams of adjacent memory address requests, which look like multiple random memory access sequences. As analyzed by previous study, conducted by Shevgoor et al. [6], memory access sequences would sometimes appear random patterns, but in the view of different granularity and delta-based classification, in fact, those streams have multiple and separated regular patterns. To handle these kinds of irregular memory streams, the feedback directed prefetching (FDP) method [5] evaluates factors such as a prefetch accuracy, prefetch timeliness, and a degree of cache pollution caused by prefetching operation to control the aggressiveness or the conservativeness of the prefetching. This locality-based prefetcher shows effectiveness in a single-core environment. However, there are some disadvantages in the locality-based prefetcher, such as fixed stride detecting and a fixed feedback mechanism.

2.2. State-of-The-Art Prefetching Methodologies

Modern data intensive applications such as graph processing, scientific applications, and data mining programs are memory intensive and generate irregular memory streams. Therefore, a prefetching method should be more sophisticated than traditional fixed stride based prefetchers to find hidden patterns among various memory streams; this will allow it to predict memory access patterns much more accurately. Hence, some computer archi-

tects suggest several state-of-the-art prefetching methods to treat complex memory access patterns [1–3,6–13,16].

Nesbit et al. [1] suggest a structural methodology for prefetching. To provide more accurate prefetching, a prefetcher introduces a history table. Hence, the Global History Buffer (GHB) prefetcher suggests a management method for the history table. The GHB employs index tables and global history buffers to store and track recently requested physical addresses. Furthermore, the GHB structure can simply access the global history buffer by storing the memory access key from the index table. In addition, the GHB prefetcher manages the history table using the FIFO structure. Therefore, the prefetching operation can be activated based on the most recently accessed request.

Shevgoor et al. [6] proposed the variable length delta prefetcher (VLDP) that is based on the delta history table and the history table. The VLDP generates prefetching candidates by considering the correlation of deltas of recently access streams. To predict the next demand requests more accurately, the VLDP tries to match the same delta chain across pages and refresh delta chains information in the delta history buffer. In addition, a delta prediction table is introduced to find the correlation among deltas, and the offset prediction table controls the aggressiveness of the prefetching. To generate more accurate candidates, the VLDP uses multi-level tables to find the most fitted chain of deltas, whose lengths range from 1 to 3. When the most appropriate delta pattern is found throughout stored delta chains information, the VLDP activates a prefetching operation.

Generally, most prefetchers attempt to prefetch with a single approach, however, Kondguli et al. [7] suggested a novel method that is based on combining multiple methodologies. This prefetcher analyzes memory access patterns by three aspects: the first module targets stride streams, the second module traces pointer traversal, and the last module targets memory streams showing high spatial locality. The prefetching controller manages these three sub prefetchers, and prefetching is conducted by selecting the module that seems to be the most fitting strategy for current memory access behaviors.

And commercial off-the-shelves processor vendors apply hardware-based prefetcher for on-chip architecture, i.e., simple stride, next line, and adjacent lines prefetchers, as Youngjoo et al. surveyed [10]. However, many on-chip cache prefetcher research has tried to design lightweight and high coverage with high accuracy prefetchers as Bingo Spatial Data prefetcher and Domino Temporal Data prefetcher [11,12]. Bingo spatial data prefetcher finds the most appropriate memory patterns through analyzing both short-term and long-term events with program counter, memory offset and memory address histories. Bingo prefetcher also uses a history table to find matched access patterns with using two key pairs as signatures (e.g., PC+Offset and PC_Address) for two-step lookup mechanisms [11]. Domino prefetcher, also suggested by the same authors of Bingo spatial data prefetcher, concentrated on finding temporal data patterns through memory sequences. Domino prefetcher predicts next demand memory requests by one or two consecutive previous accesses with the enhanced index table (EIT), and it exploits the correlation between history of demand requests and the index (pointer) information to match history table entries, when the same address in the history table has been accessed [12]. With EIT, consisting of the history table (HT) and the index table (IT), Domino prefetcher can attempt that a certain prefetch candidate will be re-visited soon.

Also, there are several domain-specific prefetcher designs suggested by computer architects, Ainsworth et al. suggested graph prefetching method based on graph data structure's intrinsic characteristics [16]. This graph prefetcher targets for prefetching between L1 and L2 cache layers and requires internal filtering and timing analysis structures to refine memory addresses into address boundary information based on vertex and edge history.

However, to satisfy the generality of usage and to cover large working set size of modern data intensive and memory hungry applications, the prefetcher targeted to generate prefetch candidates for the main memory and feasible design complexity are required.

3. Methodology: Proposed Prefetcher

The proposed eDRAM-based prefetch buffer is placed between the LLC and the main memory system. We employ the adaptive prefetching granularity method that is introduced to predict irregular memory streams arising from modern data-intensive applications. To handle these kinds of memory streams, we extended the granularities for memory access analysis; a set of several cache lines, called the correlated line, which has a wider window of locality to find hidden memory access patterns. And our model attempts to find memory address deltas among irregular memory streams with various access granularities, i.e., cache lines (64 B), correlated lines (as suggested), and page granularities (4 KB). Furthermore, we employ a history table structure that manages irregular memory streams as abstract forms and a prefetch engine that achieves an optimal trade-off between the prefetch accuracy and prefetch coverage.

3.1. Overall Architecture

The overall architecture of our proposed model is depicted in Figure 1. The two major modules of our model are a history table and a controller. First, the history table contains the page table and the correlated line table that belongs to the page table. In addition, there is a picture table that updates contents if more than four memory accesses occur on the same page or the same correlated line. The prefetch engine can generate prefetch candidates based on past memory accessing history stored in this history table, and multiple granularities-based history storage enable that the prefetch engine considers various delta analysis windows to find frequently used delta patterns. Second, our proposed model provides the prefetch engine that operates three different types of prefetching by pattern analysis window's granularities, i.e., correlated-line size (a set of multiple cache-lines), page size (4 KB), and the history table wide. Third, the prefetch engine operation is triggered by pattern matching with the correlation information stored in the picture table when finding any appropriate delta patterns based on the history table's analysis phase.

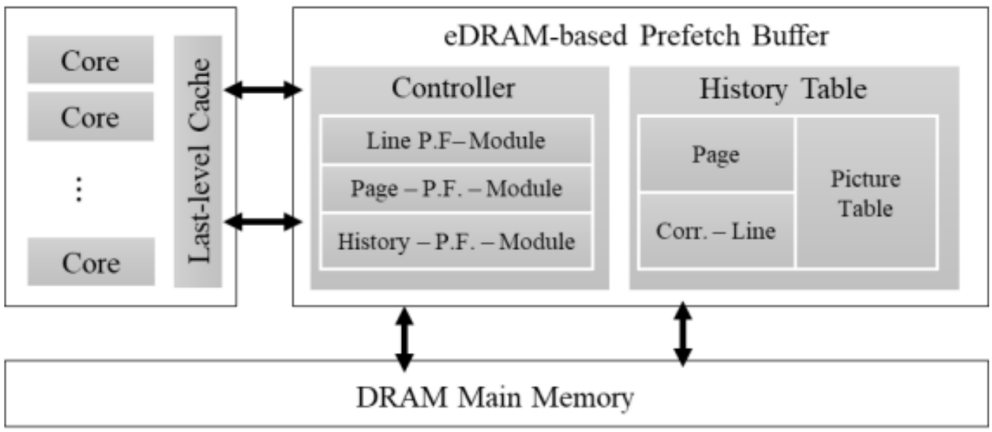


Figure 1. Overall Architecture.

3.2. History Table

The history table consists of two sub-tables, including a page table and a correlated line table that stores information regarding overall memory access patterns; further to this, it consists of the picture table that is updated only when 4 or 11 more accesses occur on the same page or on the same correlated line. The page table stores page history information about the previously accessed memory requests and has 64 entries of records. The correlated line table belongs to the page table, and up to 64 block indexes can be stored per page. The picture table has 64 entries, which is divided into two sets of 32 entries to store the page access pattern and correlated line access pattern, respectively. Thus, Figure 2 depicts an internal structure of the history table.

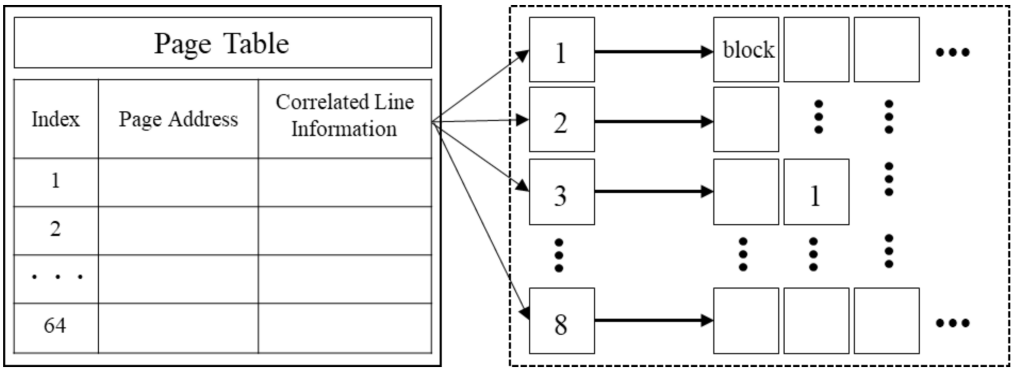


Figure 2. History table.

3.2.1. Page Table

Our proposed model suggests the prefetching method using the correlation of a delta given by the difference between consecutively accessed memory requests. However, it is difficult to observe correlations when deltas are stored without a certain condition. Thus, we stored the address by page, not an original physical address, which limits the delta range from −63 to +63. The page table has 64 entries. To analyze the overall memory access pattern, the table will be updated when a demand request is generated from the LLC. Each page entry stores information about Page_Adr and CorLine_Info. The former stores the accessed address shifted by 12 bits. Hence if both two entries' values are equivalent, we assume that accessed addresses are on the same page. The page table update depends on whether the page number of the current accessed address is already stored in it. If this page number is already stored, we update the CorLine_Info index stored by the block. By contrast, when the address does not exist in the page table, it will be stored sequentially if there is enough space. When there is no empty entry, a victim page is evicted under the Least recently used (LRU) replacement policy, and consequently, all CorLine_Info of the page is also evicted. The reason for adopting the LRU replacement policy is to match the patterns being accessed while storing information about frequently accessed pages for as long as needed.

3.2.2. Correlated Line (CorLine) Table

To analyze irregular memory access patterns in more detail, we defined the concept of a correlated line. In this paper, we assume that the page is 4 KB and the cache block size (cache line size) is 64 bytes. Therefore, there are 64 blocks on one page, and we group 8 blocks into 1 correlated line. That is, 8 correlated lines exist on 1 page, and 8 consecutive cache blocks exist on 1 correlated line. The concept of a correlated line makes analyzing the memory access patterns even more efficient. To efficiently manage the CorLine table, the accessed address is stored through following steps; when access to the same page occurs, the remainder operation is conducted with 64 bytes for the current accessed address, and this value becomes the block number. In addition, we apply a remainder operation with 8 blocks to obtain the correlated line number. The size of the correlated line table is assumed to be up to 4 KB. Because the correlated line table belongs to the page table, all information is removed when the page table is evicted. However, all accessed information is stored until a page eviction occurs because the correlated line table entails the criteria for prefetching. The correlated line table stores the correlated line number, accessed cache block number, index number, and accessed timestamp. In addition, when two or more cache line blocks are accessed on the same page, if the memory address distance between the two consecutively accessed blocks is within the range of -8 to +8. Hence, it is determined as the same correlated line. Information regarding access to the same correlated line is used as the criterion for deciding whether to perform prefetching in an aggressive or conservative manner.

3.2.3. Picture Table

The picture table is an essential part of our methodology along with the correlated line. We observed empirically that the delta pattern appears to be irregular when an access that changes the correlated line occurs. Therefore, to express this access pattern abstractly, we propose the picture table. Our model generates prefetching candidates with consideration of the delta correlation among previous memory requests, and the picture table stores history information based on cache-line size based (64 B) addresses. In this manner, prefetching could be activated more frequently than the case of a delta-based prefetcher module, because delta-based analysis requires more than three consecutive memory accesses in the same page address (page number = original memory address >> 12, when the page size is 4 KB [2^{12} = 4096 Bytes]). The picture table consists of 64 entries among which 32 entries are used as a table for analyzing page-granularity-based histories, and the other remaining 32 entries are for analyzing the correlated line-based accesses. The picture table does update its contents when at least four accesses occur on the same page or the same correlated line. When the update condition is satisfied, the history table refreshes contents in the picture table with the cache block address.

Figure 3 shows the internal update flow of the picture table. First, we index the block number by conducting the remainder operations based on a value of 8. We define this value as the index number. The first index number is subtracted from each index number for a clear representation while maintaining the delta spacing. Throughout this step, all starting points in the picture table are initialized as zero. When subtracting each index number from the first index number, to fix the range of the index number from zero to 7, a value of 8 is added when a negative value occurs. By conducting the remainder operations and limiting the range of the index numbers, we analyze the correlation more comprehensively. To store the most frequently occurring pattern in the picture table for a prolonged time, we manage the picture table based on the number of times the same access occurs. In addition, the number of duplicates is applied by dividing the page-based table and the correlated line-based table. Through the picture table, we predict memory access patterns more accurately.

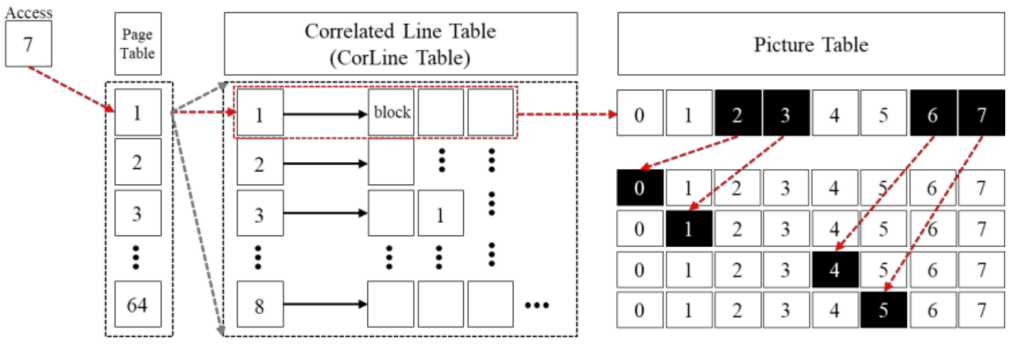


Figure 3. Update flow of picture table.

3.3. Prefetch Engine

The prefetching accuracy and coverage ratio generally have a trade-off relation. Prefetching based on a certain pattern within a small range can yield an extremely high accuracy, but it may be difficult to expect a positive impact on the overall performance owing to a low coverage concerning the overall access. Nevertheless, conducting a prefetching with high aggressiveness to achieve high prefetching coverage ratio may cause a negative impact to the overall performance, when prefetchers do generate prefetch candidates without considering accuracy ratio. Therefore, our proposed prefetch engine performs generating prefetch candidates via three different modules to obtain optimal accuracy and coverage, hence, when one prefetch module does not satisfy appropriate accuracy and performance enhancement, our prefetch engine will choose other prefetcher module to predict more accurate prefetch candidates. The prefetch engine is controlled by a prefetch con-

troller, and it consists of three different prefetch modules as previously described. The first module generates prefetch candidates by analyzing the correlated-line granularity-based access patterns, and the second module predicts future memory accessing by analyzing the page granularity-based access patterns. And the third module analyzes the correlation of the delta per page granularity across the overall history table. The first module performs well in terms of the prefetch accuracy. By contrast, the second module performs worse than the first module in terms of accuracy but performs better in terms of coverage. However, because the first and second modules conduct prefetching conservatively, to achieve a high coverage, the third module triggers prefetching aggressively even if the page has one delta. Prefetching is first applied based on the first module, and if the criteria of the first module are not satisfied, prefetching is applied based on the second module. If the prefetch criteria of both modules are not satisfied, prefetching is conducted through the third module.

3.3.1. Correlated Line-Based Prefetching Module

The first module conducts prefetching by analyzing the memory access patterns based on the correlated lines as shown in Figure 4. We trigger a prefetching when at least three accesses occur on the same correlated line. This is the same as conducting prefetching after considering two deltas when prefetching based on the delta is performed. Prefetching based on a larger number of previously accessed memory requests will yield valid results in terms of accuracy but unsatisfactory results in terms of coverage. By contrast, if prefetching is conducted based on fewer accesses, both the accuracy and coverage results are unsatisfactory. Therefore, we conduct prefetching through the first module when at least three accesses occur on the same correlated line. When the demand requests from the LLC do not exist in the prefetch buffer, the first module determines whether to conduct the prefetching. First, the number of accesses on the same correlated line of the currently accessed block is counted through the history table. When the number of times the correlated line has been accessed is less than three, we do not perform the prefetching. By contrast, when there are three or more consecutive accesses to the same correlated line, pattern matching with the picture table is conducted based on the three most recently accessed blocks. As in the updating method of the picture table, pattern matching is conducted by replacing three blocks with the appropriate index numbers. If the same pattern exists on the correlated line-based picture table, prefetching is conducted based on that pattern. When the same pattern does not exist in the correlated line-based picture table, the page-based picture table is checked. If the same pattern exists in the picture table more than once, prefetching is conducted using more duplicate numbers. When the same access pattern does not exist in both tables, we do not conduct a prefetching.

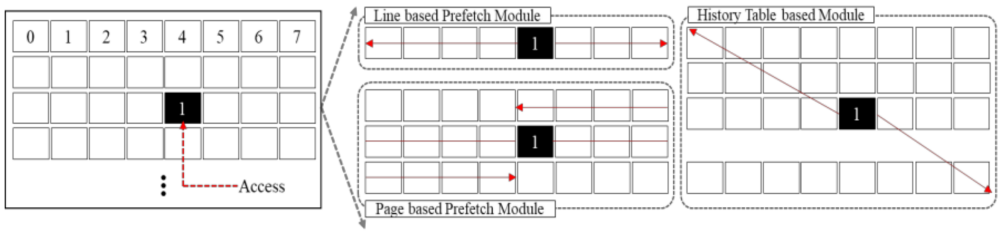


Figure 4. Selection criteria for the prefetching module.

3.3.2. Page-Based Prefetching Module

The first module, i.e., correlated line-based prefetching, shows an acceptable prefetch accuracy. However, depending on characteristics of the memory access pattern, a large number of accesses may not occur in the same correlated line belonging to the same page. These results show a high prefetching accuracy, but because a low coverage does not lead to positive results considering the overall performance, we conduct a more aggressive prefetching through a second page-based module. For a specific workload, the reason for the low coverage of the first module is that the range of the correlated line is too small.

Therefore, the second module does not divide the accessed addresses on the page into correlated lines but analyzes the patterns on the pages. A page-based access pattern is more irregular than a correlated line-based access pattern. However, because we update the picture table by dividing it into two, we can account for these irregular patterns. The second page-based module performs pattern matching with the page-based picture table when three or more consecutive accesses occur within the same page. When the same pattern is stored in the picture table, prefetching is applied by adding the delta stored in the table to the currently accessed address. For conducting pattern matching, our module extracts a page address from the demand request's physical address to find the matched entry of the page table, and thus, we can consider irregular deltas to be caused by the differences between correlated lines. This process makes it possible to analyze the correlation of the memory access pattern more comprehensively than when analyzing the correlation based on the delta only. Therefore, we can expect a higher coverage than that of delta-based prefetchers.

3.3.3. History Table-Based Prefetching Module

Although the first and second modules can cover some irregular patterns, the prefetching is conducted based on the picture table, and the coverage may be degraded. This is because we managed the history table considering the number of redundant memory accesses. In short, after many requests have occurred, although the access characteristics stored in the picture table appear to be irregular, they correspond to the most frequently employed access patterns for the current workload; to increase the coverage, we also need a third module that attempts prefetching based on a pure delta correlation, not based on a picture table.

However, it is difficult to decide the circumstances under which prefetching should be performed through a third module. Through numerous simulations, we decided that it is inappropriate to perform prefetching on a picture table at the moment when accesses corresponding to varied correlated lines occur. This is because when the two most recently accessed blocks are on different correlated lines, it is difficult to predict on which correlated line the next access will occur. In other words, when accesses repeatedly occur on the same correlated line, there is sufficient reason to assume that the next memory access will also occur on the same correlated line, and thus in this case, prefetching is performed through the first and second modules for accuracy. In addition, we determined that correlated line changes represent the starting point of an irregular access pattern, and thus, we conducted an aggressive prefetching in this case through the third module. The third module applies a prefetching in consideration of the correlation among recently accessed deltas in the overall history table to conduct a prefetching on patterns that are difficult to cover using the picture table. With the third module, we can expect a positive effect in terms of coverage, but a slightly negative impact on the accuracy.

3.4. Controller

Figure 5 describes the overall structure of the controller, and Figure 6 shows the internal operation flow of the prefetch controller. The controller manages all functions of the prefetch buffer. First, in the history table, an update is applied when memory access is requested from the LLC to the prefetch buffer. The controller checks whether the page number of the current demand request's address is stored in the page table and refresh the contents of the table consequently.

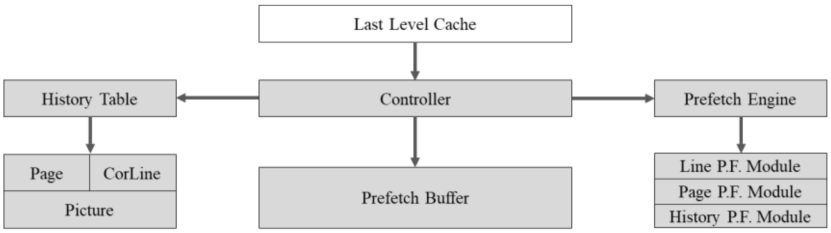


Figure 5. Controller.

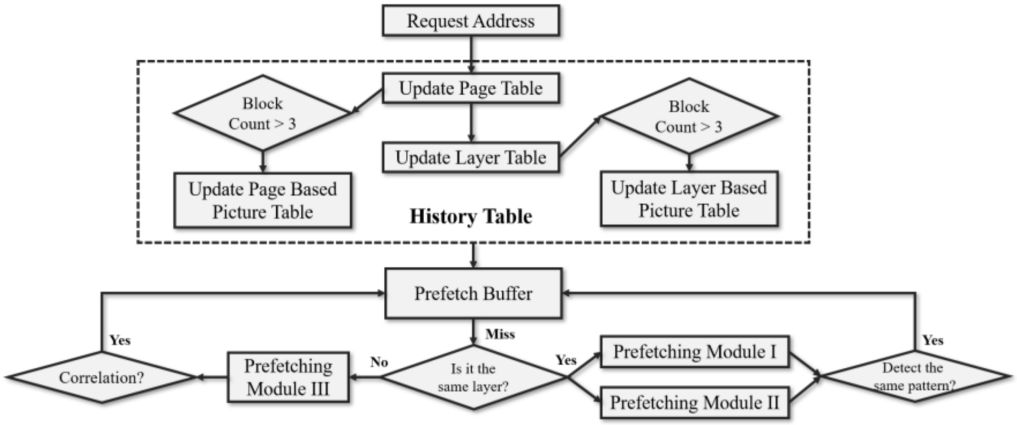


Figure 6. Operational flow of prefetch controller.

If the page number of the current accessed address is already stored in the page table, then the corresponding correlated line table is also updated as shown in Figure 6. When the number of blocks accessing the correlated line to which the currently accessed address belongs is four or more, the correlated line-based picture table will be updated. Otherwise, we check whether there has been a previously accessed history with four or more access times per page. If so, we update contents of the page-based picture table. In addition, the controller manages a prefetch buffer. In our proposed model, prefetching operations only occurs for DRAM. In other words, only prefetched data are stored in the prefetch buffer. Therefore, the controller that manages the buffer controls the prefetching. To conduct the prefetching, the controller determines whether the current access is on the same correlated line as the previous access. When the accessed address is on the same correlated line, the controller analyzes the access pattern through modules I and II. When it does not exist on the same correlated line, the controller conducts a prefetching aggressively through module III. If the prefetch engine executes the prefetching, the fetched data are stored in the prefetch buffer by the controller. The controller manages the prefetch buffer by applying the LRU replacement policy.

3.5. Prefetch Buffer

Our proposed model uses a separated eDRAM-based prefetch buffer to aggressively conduct prefetching without causing cache pollution and consuming high-expensive SRAM spaces. The eDRAM has the advantage of operating at low latency with high space per cost. In addition, it has a higher density than the traditional SRAM based on-chip caches. Therefore, we can access the buffer faster than the DRAM-based main memory and achieve a larger storage capacity than that of the LLC. The prefetch buffer is located between the LLC and the main memory. The buffer size is 8 MB, and it stores data in 64-byte block units. Furthermore, it is designed as 16-way set associative structure based on the LRU replacement policy, as shown in Figure 7. Thus, its internal organization is similar to that of LLC cache as Moin et al. suggested a DRAM-based buffer for PCM main memory system in previous study [17]. However, the only difference is a target functionality of the buffer. The prefetch buffer only stores data that are generated by the prefetch engine, not the

demand requests from LLC or main memory system. There are two types of eDRAM, i.e., 1T1C eDRAM and gain cell eDRAM. In this study, we assumed the use of gain cell eDRAM and do not consider the refresh operation issue of eDRAM [18].

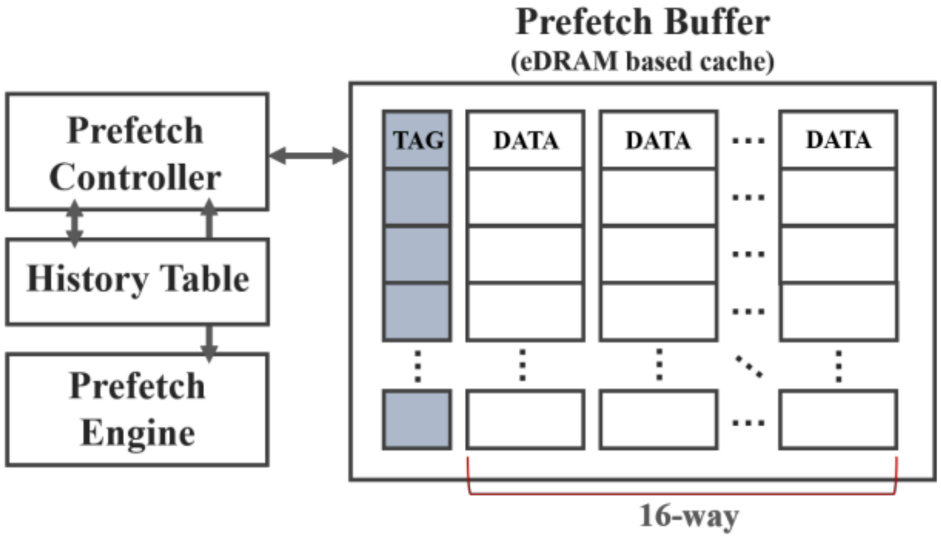


Figure 7. Internal structure of Prefetch Buffer.

4. Evaluation

In this section, we describe the analysis of the prefetching performance of our proposed model based on the adaptive granularity method with an eDRAM-based prefetch buffer. First, we analyze the optimal size of the eDRAM-based prefetch buffer and the trade-off between accuracy and coverage. Finally, we compare the performance of this model with that of other state-of-the-art prefetchers in terms of the execution time and energy consumption.

4.1. Experimental Environment

We evaluate our proposed model using an in-house trace-driven memory system simulator. The workloads used for the evaluation is described in Table 1, and the simulator's configuration is described in Table 2.

4.1.1. Memory Access Pattern of Graph Processing Workload

In this study, we select graph processing workloads to measure the impact of irregular memory access patterns on system performance. To evaluate the performance of the prefetching algorithm in various environments, nine workloads are used as follows, i.e., GraphColoring (gcolor), kCore (*kcore*), BFS (*bfs*), ConnectedComp (*ccomp*), BetweennessCentr (*bc*), ShortestPath (*sssp*), TriangleCount (*tc*), PageRank (*prank*), and degreeCentr (*dc*). All these workloads are from the GraphBig framework [19].

Table 1 shows the computation type of each graph kernel, a kind of memory access pattern, and real use case of each graph workload. As shown in Table 1, graph processing workloads are commonly memory-intensive and have irregular memory access patterns. When this characteristic permeates all levels of caches and memory hierarchy, it leads to a substantially more irregular and complex pattern. Figure 8 shows the distributions of delta values among memory accesses which occur on the same page by one million memory requests caused by LLC misses. In each graph, the x-axis is the number of memory requests from the LLC, and the y-axis is the delta value. In this study, the range of delta is limited from -63 to $+63$, because we analyze the delta based on page granularity. Throughout Figure 8, we can observe that kcore, ccomp, and prank generate demand requests based on various delta values. In addition, it is difficult to find access patterns and criteria for prefetching. By contrast, in the case of bc, repeated accesses that have the same delta

value are observed, which can be treated as a regular access pattern. Therefore, we can determine which prefetcher generates good candidates for irregular or regular demand memory request patterns through various graph processing workloads.

Table 1. Workloads of Graph Processing.

Workload	Computation Type, Feature, and Use Case
GraphColoring (gcolor)	Computation on graph structure Irregular access pattern, heavy read accesses Graph Labeling
kCore (kcore)	Computation on graph structure Irregular access pattern, heavy read accesses Social Media Monitoring
BFS (bfs)	Computation on graph structure Irregular access pattern, heavy read accesses Recommendation for Commerce
ConnectedComp (ccomp)	Computation on graph structure Irregular access pattern, heavy read accesses Social Media Monitoring
BetweennessCentr (bc)	Computation on graph structure Irregular access pattern, heavy read accesses Graph Labeling
ShortestPath (sssp)	Computation on graph structure Irregular access pattern, heavy read accesses Smart Navigation
TriangleCount (tc)	Computation on graph structure Irregular access pattern, heavy read accesses Data Curation for Enterprise
PageRank (prank)	Computation on graph structure Irregular access pattern, heavy read accesses Social Media Monitoring
degreeCentr (dc)	Computation on graphs with rich properties Heavy numeric operations on properties Social Media Monitoring

Table 2. System Configuration.

Proposed System Configuration	
Processor	Quad-cores, 4 GHz
Private L1 Instruction Cache (per core)	32 KB, 8-way set associativity, 64-byte block size, LRU replacement
Private L1 Data Cache (per core)	32 KB, 8-way set associativity, 64-byte block size, LRU replacement
Private L2 Unified Cache (per core)	256 KB, 4-way set associativity, 64-byte block size, LRU replacement
Shard L3 Unified Cache	8 MB, 16-way set associativity, 64 byte block size LRU replacement
Prefetching Buffer (eDRAM)	8 MB, 16-way set associativity, 64 byte block size LRU replacement
Main Memory (DRAM)	8 GB, fully associativity, 4 KB page size LRU replacement

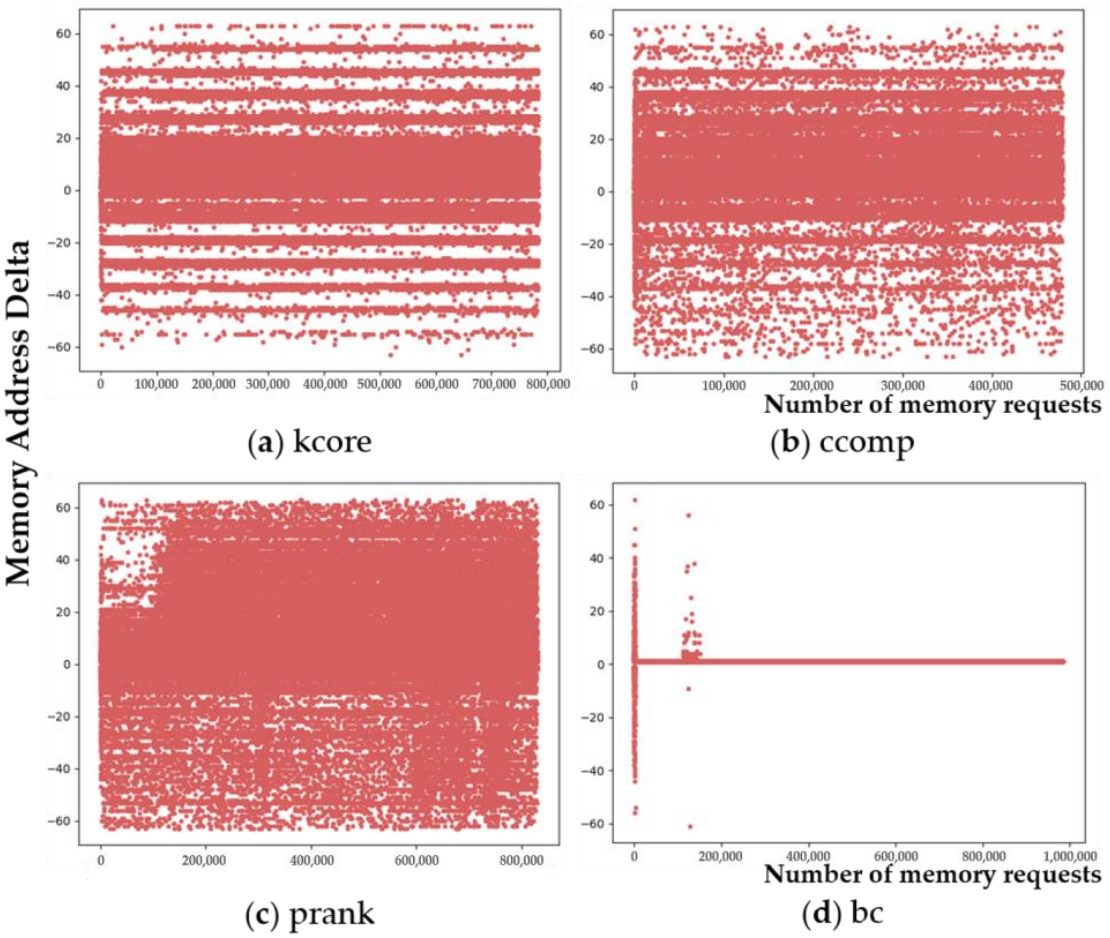


Figure 8. Memory access patterns of graph processing.

4.1.2. Simulator Configuration

Our proposed model is evaluated with an in-house memory system simulator which is trace-driven. The detailed configuration of the simulator is described in Table 2. We place an eDRAM-based prefetch buffer between the LLC and the main memory. As shown in Table 2, the processor consists of four cores, and L1 and L2 caches are privately allocated to each core. The L1 cache consists of an instruction cache and data cache, and each cache has 32 KB of space with an 8-way set associativity. The L2 cache has unified cache structure and is composed of 256 KB with a 4-way set associativity per core. The L3 cache (LLC) is shared by all cores and consists of 8 MB with a 16-way set associativity. All caches store data as cache block granularity (64-byte) and replace cache lines with the LRU replacement policy. The eDRAM-based prefetch buffer has similar configurations to the LLC. The prefetching operation is only carried out in the DRAM. Additionally, the DRAM capacity is 8 GB, and it stores data with 4 KB page granularity.

To generate prefetch candidates, we designed a history table to store information on the recently accessed memory requests. The history table consists of three main modules. First, the page table is configured with 64 entries to store information of the page. The history table uses the LRU replacement policy to retain information on the correlated line table belonging to the page table for as long as possible. The correlated line table belonging to the page table consists of 4 KB. Our prefetcher decides whether to apply a prefetching by tracking accesses corresponding to the same correlated line based on the information of correlated line table. Finally, the picture table is updated only when four or more accesses occur on the same correlated line or the same page granularity; it consists of 64 entries. The 64 entries are divided into one half (32 entries) that stores page-based access patterns

and another half that stores correlated line-based access patterns. To store the pattern related to the most redundant access for a long period of time, the picture table applies a replacement policy based on the number of times the duplicates are accessed. In addition, Table 3 shows the performance parameters of memories and the disk which are used for our in-house simulator.

Table 3. Simulation Parameters.

Parameter	eDRAM	DRAM	HDD
Write Latency	4.29 ns	20~50 ns	~5 ms
Read Latency	4.29 ns	20~50 ns	~5 ms
Write Energy	1.74 nJ/access	1.2 J/GB	65 J/GB
Read Energy	1.74 nJ/access	0.8 J/GB	65 J/GB
Idle Power	~49.01 mW/bank	~100 mW/GB	~10 W/TB

4.2. Optimal Size of eDRAM Based Prefetch Buffer

Considering the scalability of an eDRAM-based prefetch buffer, we analyze various experiments to find the sweet spot of configuring buffer size. The buffer size is set to 8, 16, and 32 MB, which are comparable to the sizes of LLCs in commercial on-the-shelf processors. In this experiment, we set various eDRAM buffer sizes, but the size of the history table is fixed. To determine the optimal buffer size for our system configuration, we analyze two key features, in terms of total execution times of workloads and energy consumption during running times.

Figure 9 shows the results of the energy consumption with different buffer size configurations. As in the case of the execution time measurement, the 16 MB buffer-based configuration and 32 MB buffer-based configuration possess larger storage spaces and provide better results than the 8 MB buffer in terms of dynamic energy consumption, except static energy consumption such as eDRAM's refresh energy. When all accessed data are stored in the buffer, the difference in performance may be larger owing to the difference in capacity. Hence, the total energy consumption is less sensitive to the size of the buffer.

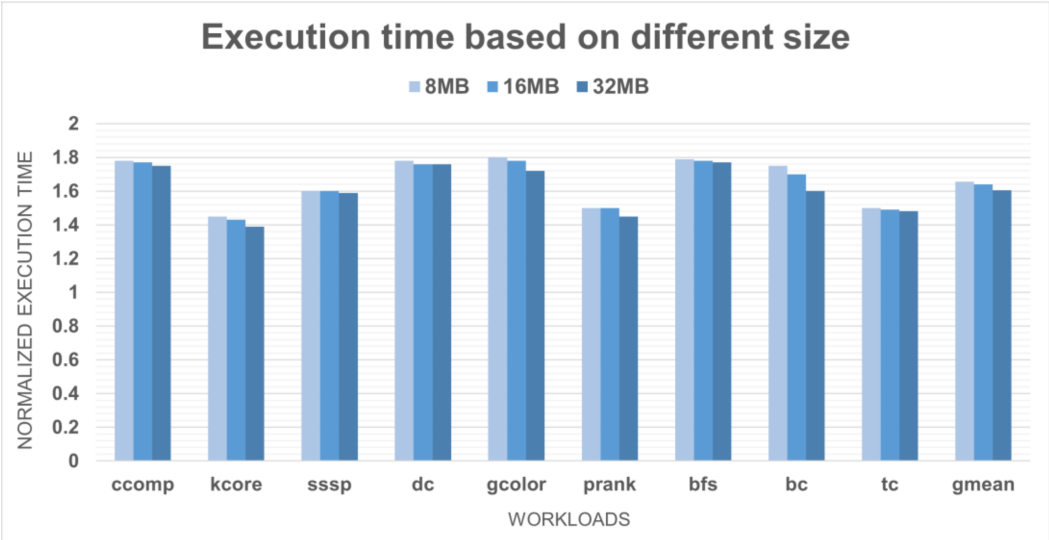


Figure 9. Execution time with respect to buffer size.

Figure 10 shows the results of the energy consumption with different buffer size configurations. As in the case of the execution time measurement, the 16 MB buffer-based configuration and 32 MB buffer-based configuration possess larger storage spaces and provide better results than the 8 MB buffer in terms of dynamic energy consumption, except static energy consumption such as eDRAM's refresh energy. When all accessed data are

stored in the buffer, the difference in performance may be larger owing to the difference in capacity. Hence, the total energy consumption is less sensitive to the size of the buffer.

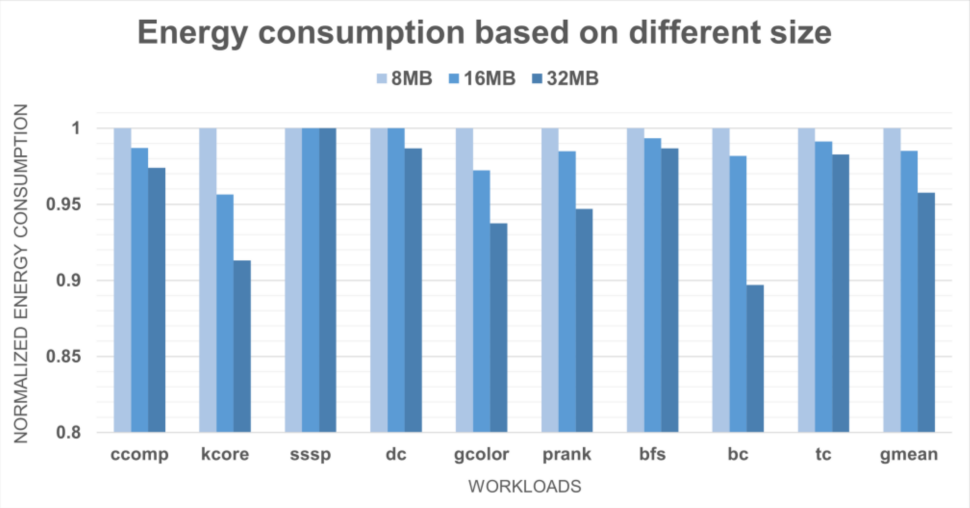


Figure 10. Energy consumption (dynamic energy consumption) with respect to buffer size.

Even if an inaccurate prefetching occurs and the required buffer size increases, it becomes less likely that useful data will be evicted in the cases of the 16 and 32 MB buffers. Therefore, they achieve a better performance than the 8 MB buffer for all workloads. However, the difference in performance was less than the difference in the actual capacity, and the actual performance is not sensitive to the size of the buffer. In addition, considering the cost aspect of the size of the buffer, we conducted the following evaluations based on an 8 MB buffer size only.

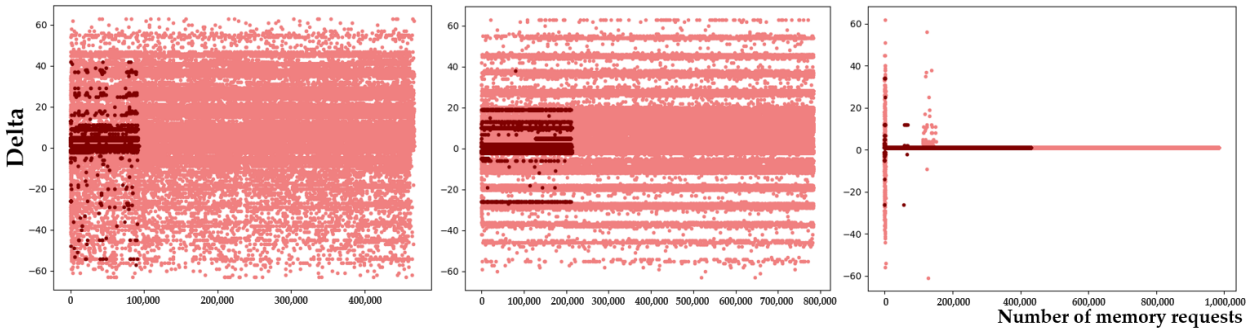
4.3. Evaluation of Coverage Exhibited by Prefetching Module

When we design a prefetcher, we should consider the trade-off between the prefetching accuracy and the prefetching coverage. If we analyze an increased number of deltas for prefetching on a regular pattern, we can expect a high prefetching accuracy. However, it is hard to find the exact regular patterns on certain workloads, and many misses will occur until the finding of exact patterns for generating exact prefetching candidates. By contrast if we design a prefetcher which sacrifices the accuracy to increase the coverage, the performance may not be improved, and the prefetching cost will be higher than before. Therefore, when computer architects construct prefetching strategy, they should consider the balance of coverage and accuracy in order to not degrade the entire system performance by triggering cache pollution.

Figures 11 and 12 depicts the visualized results of the prefetcher's coverage. The dark red area shows the demand memory requests that are covered by prefetcher, hence all those requests are hit in the buffer. And, the light red area shows uncovered demand memory requests, which cause buffer misses. To simplify the visualization, we set the y-axis as memory access delta value, not actual memory addresses, that varies -64 to $+64$.

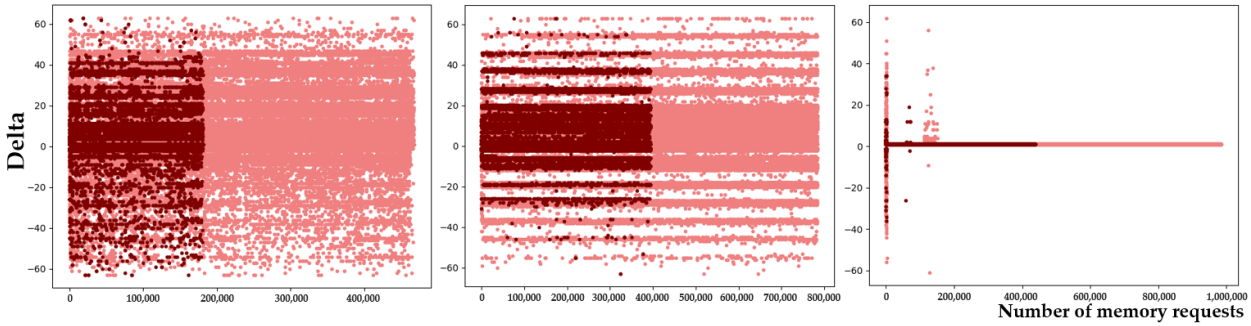
Figure 11 shows the results of prefetching generated by only two prefetching engine modules, module— module I and module II. We will refer to modules I and II as module S and to module III as module B. Module S only generates prefetching candidates when memory accesses occur on the same correlated line. In this case, pattern matching with the picture table is performed only if there are at least three accesses on the same current correlated line or page. Finally, prefetching will be triggered when the same pattern exists in the picture table throughout the pattern matching operation. These conditions trigger the module S to operate prefetching conservatively. As a result, when prefetching is performed only through the module S, it is difficult to expect a high coverage, as shown in Figure 11.

However, these conditions allow both modules to exhibit higher accuracy by analyzing memory access patterns in more detail. In addition, considering the location of the current prefetch buffer, which is between the LLC and the main memory, we can expect better coverage if the prefetcher is placed at the higher-level cache such as the L1 or L2 caches. In fact, module S shows higher coverage and accuracy for workloads that involve intensive accesses on a certain page or correlated line.



(**a**) gcolor (**b**) kcore (**c**) bc

Figure 11. Coverage of module-S.



(**a**) gcolor (**b**) kcore (**c**) bc

Figure 12. Coverage of propose model.

In addition, Figure 12 shows the coverage ratios of our proposed model that complement the coverage ratio of module S by adding module B. We observe that the coverage ratio has been significantly improved by introducing module B for some workloads like *gcolor* and *kcore*. However, the most important aspect is to enhance the coverage ratio while keeping the degree of prefetching accuracy. Through the concept of adaptive granularities for prefetching, we determine an access corresponding to a different correlated line that indicates an appearance of irregular memory streams. Therefore, we aggressively activate prefetching operations when the current demand request does not correspond to the same correlated line as in the previous case. However, as shown in Figure 12, we observed that even if the prefetcher aggressively generates prefetching candidates, there are still many uncovered cache misses. This is because aggressive prefetching is activated after analyzing at least two cache misses. In addition, for the case of *bc* as shown in Figure 12, in the case of regular streams, the coverage of module S is as high as our proposed model's one.

4.4. Accuracy Evaluation Using Prefetching Module

Analyzing the results from Figures 11 and 12, we confirmed that the model attaches module B to module S, which aggressively activates prefetching operations, hence the coverage ratio of our model can be enhanced. However, the aggressive prefetching mode inevitably leads to a degradation of prefetching accuracy, highly related to the entire system's performance.

Figure 13 depicts the comparison of prefetching accuracy on various workloads based on module S and our proposed model. In Figure 13, we observe that our model shows a 10% reduction in the prefetching accuracy on average, compared to module S. Additionally, the results of prefetching accuracy show our model achieved 17% reduction in *bfs* workload. Because our model manages all memory access information requested by the LLC with the history table, the components of the table are not changed owing to prefetching. Hence, every degradation tendency of the accuracy ratio is caused by module B. Module B activates the prefetching operation considering the correlation of the recently accessed requests which are stored in the history table. Thus, the degradation of prefetching accuracy indicates that analyzing the correlation with only data from the history table is not a simple solution. In addition, the accuracy ratio is degraded by an aggressive prefetching; nevertheless, Figure 14 shows that the buffer hit count increases in the case of aggressive prefetching. Increasing the buffer-hit count means a decrease in the miss ratio for the total memory access. This indicates an improvement in the coverage aspect and leads to an improved performance in terms of total execution time and energy consumption. When an appropriate accuracy is maintained, the coverage will have a more decisive effect on the overall performance. Therefore, our proposed model can achieve a better performance than module S, which performs prefetching conservatively by maintaining a valid accuracy.

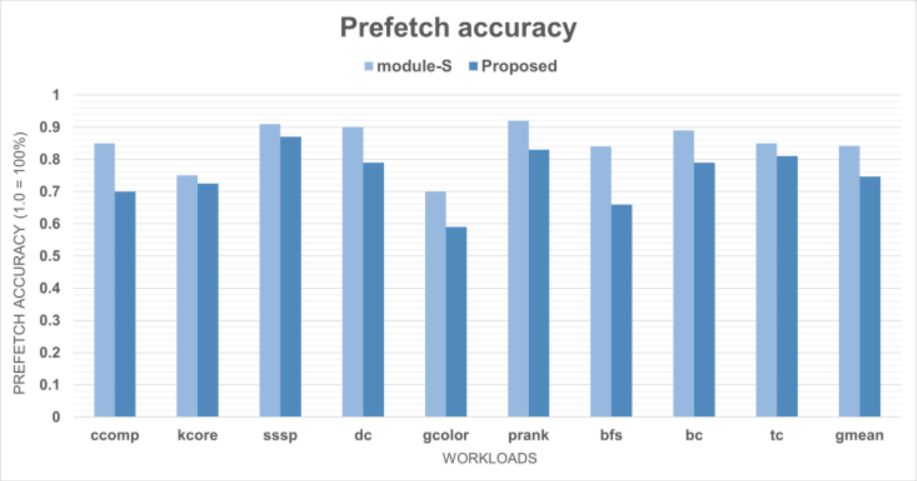


Figure 13. Prefetch accuracy for graph processing.

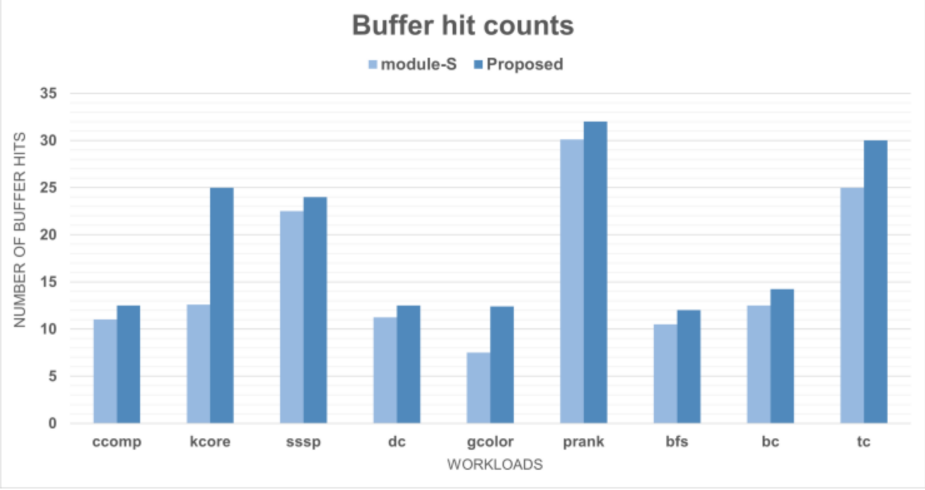


Figure 14. Buffer hit count of graph processing.

4.5. Analyzing Memory Access Pattern for Graph Processing

Figure 12 shows that there are large areas that we fail to cover, despite aggressively conducting prefetching by adding module 3. We believe that this situation may occur for two major reasons. First, when we prefetch the data, a buffer miss may occur owing to inaccurate prefetching. Second, prefetching may not be attempted because the prefetching criteria are not satisfied. Figure 13 shows that our proposed model has a high prefetch accuracy in most cases. Therefore, there are many situations in which prefetching operation is not triggered.

Figure 15 shows the number of blocks stored on the page when the currently accessed data are in a different correlated line from the previous access. In every workload except *kcore*, there are many cases in which only one block was stored. Prefetching cannot be activated by just accessing of one cache block because at least two accesses are required to analyze the correlation based on the delta. Therefore, even if we conduct a prefetching aggressively through module B, some areas cannot be covered.

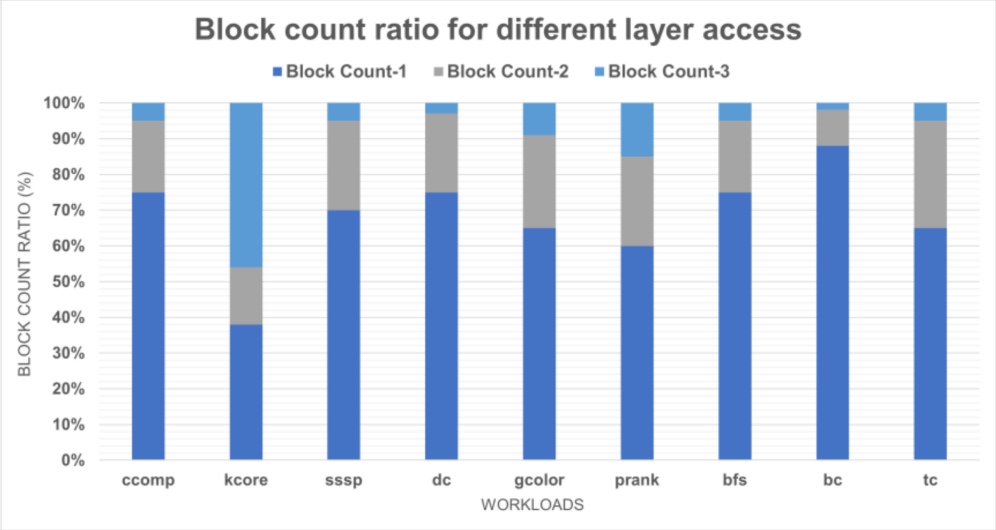


Figure 15. Block count ratio for different correlated line accesses.

Figure 16 shows the access ratio between the number of times that the same correlated line was accessed and not accessed for the entire memory access. As a result, we can observe that the greater the number of accesses that occur on the same correlated line, the higher the accuracy of our proposed model. This is because, as the number of accesses to the same correlated line increases, there are more opportunities for prefetching through module S, which can cover a large memory range with high prefetch accuracy. Therefore, our proposed model shows the best performance in the case of a *prank*.

4.6. Performance Evaluation of Basic Ideas

In this experiment, we evaluated the prefetch accuracy using a Markov prefetcher, which is one of the correlation prefetchers, and a stride prefetcher, which shows positive results with the most basic ideas. This is because module B performs correlation-based prefetching and many stride patterns occur during a correlated line level access. In this experiment, we showed that even if a similar prefetching method is used, a difference in performance may occur depending on how the prefetching is applied.

Figure 17 shows the prefetching accuracy for each model. In workloads *bc* and *tc*, we can observe that the stride prefetcher shows an extremely high accuracy. Figure 12 shows that *bc* has a very frequent accessing characteristic for delta 1. In addition, through Figure 17, we can predict *tc* to have many stride patterns. This result shows that the stride prefetcher performs very well under regular access patterns.

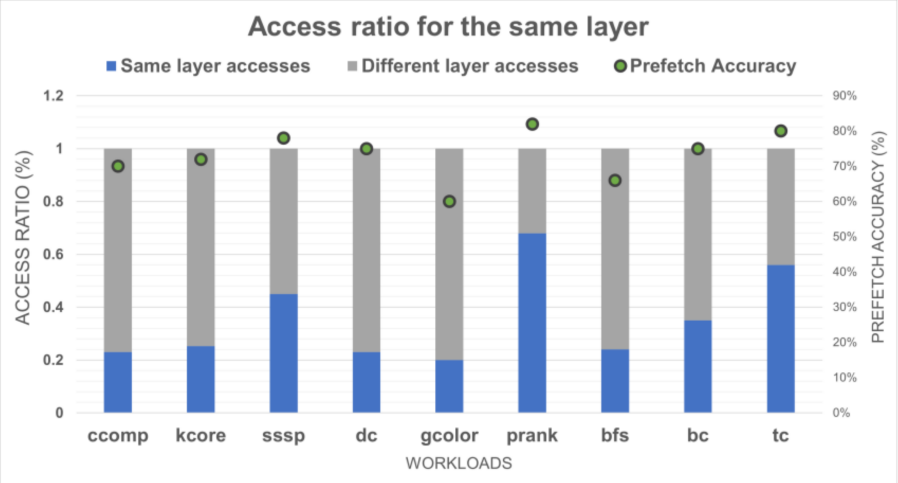


Figure 16. Access ratio of the same correlated line.

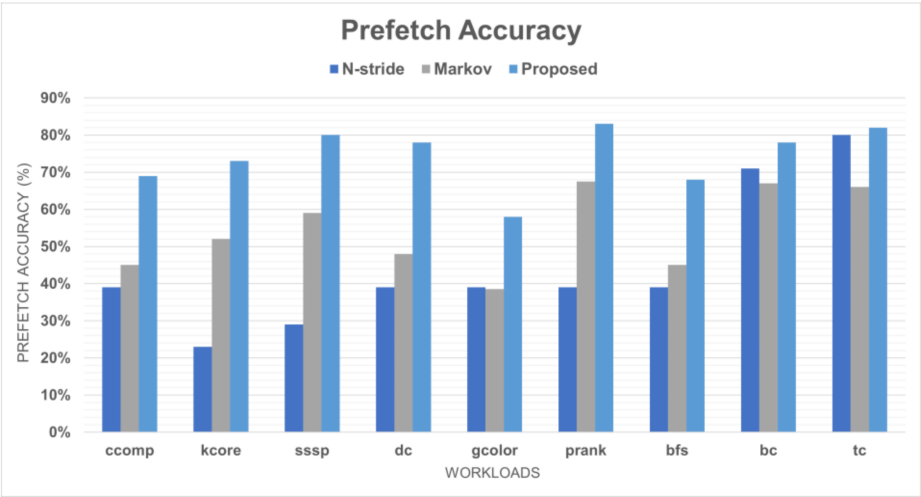


Figure 17. Prefetch accuracy of basic prefetcher.

A Markov prefetcher shows a valid accuracy in most cases. The Markov prefetcher analyzes the correlation through the history table and can cope with both irregular and regular patterns. In a correlated line level access, many memory requests show simple stride patterns. However, it is difficult to find such a stride when analyzing a pattern in the overall access order without considering the correlated line level. In other words, our proposed model can detect the memory access pattern by analyzing more details in the correlated line level that other prefetchers fail to consider. As a result, our proposed model shows the highest prefetch accuracy in every workload. Figures 17 and 18 demonstrate that our proposed model outperforms the two basic models by analyzing memory access patterns in more detail.

Figure 18 shows the total number of buffer hit count by three different prefetchers. In terms of accuracy, the stride prefetcher shows a good performance compared with the rest of the prefetchers.

However, as shown in Figure 18, we can observe that the actual buffer hit count of the stride prefetcher is extremely low. This means that prefetching is applied conservatively. The corresponding reasons can be divided into two cases. First, the algorithm of the prefetcher is conservative. This means that prefetching is conducted only when there is a clear pattern or significant information. Second, the basis for prefetching may not be found. The stride prefetcher generates prefetch candidates when the same pattern is repeated.

Therefore, through Figure 18, we can assume that the graph processing workload does not infrequently access the same pattern.

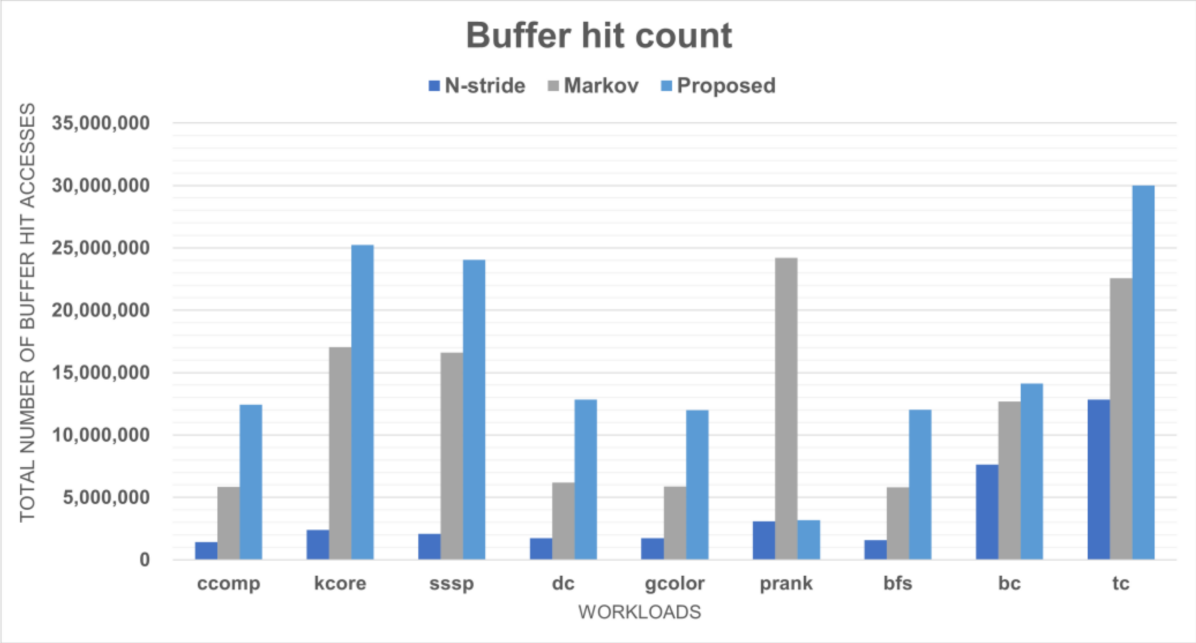


Figure 18. Buffer-hit count of basic prefetcher.

By contrast, the Markov prefetcher shows a good performance in terms of the buffer hit count. This means that prefetching was conducted well when considering the correlation between deltas. Nevertheless, our proposed model outperforms all other models in all cases except for *bc*. This is because the proposed model performs aggressive prefetching while maintaining the maximum accuracy through the three modules.

4.7. Overall Performance Analysis

We conducted performance evaluations using state-of-the-art prefetchers for a more accurate performance evaluation. The prefetchers used for comparison are GHB, BEST-OFFSET (BO [2], derived from Sandbox prefetcher [3]), Markov, and VLDP. For an accurate evaluation, all system configurations were kept identical, and the performance was evaluated in terms of the execution time and energy consumption.

Figure 19 shows the results of the execution time of the proposed model and the other prefetchers. In all cases except for workload bc, our proposed model shows the best performance. The GHB and BO prefetchers show fewer prefetching attempts compared to the other three prefetcher models. Because this prefetching is not conducted at the upper cache level, which is targeted by the two aforenoted prefetchers, there may not have been many conditions suitable to trigger the prefetching. This means that the information requested by the LLC has an irregular pattern. Our proposed model shows an average performance improvement of approximately 15% compared to the GHB and BO prefetchers. Markov, VLDP, and our proposed model are memory address correlation-based prefetching methodologies. All three models show an improved execution time compared to the GHB and BO prefetchers for all workloads. This shows that correlation-based prefetchers are more efficient in dealing with LLC cache misses with much more irregular patterns than those of higher-level cache misses. However, correlation prefetching based on the history table consequently performs prefetching on regular or frequently repeated patterns over time. To overcome these weaknesses, our model adds module B, which can perform prefetching more aggressively. As a result, our proposed model shows the best performance in all cases except for bc, which has a pattern of intensive access to a specific delta.

On average, the performance of our model was greater by 6% and 2%, respectively, than those of the Markov and VLDP prefetchers. Figure 20 shows the results in terms of the energy consumption. In terms of energy consumption, our proposed model shows a significant improvement over the GHB and BO prefetchers. These two models and our proposed model show a good performance regarding the prefetch accuracy. However, the performance gap corresponds to a difference in coverage ratio according to the number of prefetch candidates. In other words, the number of hits in the buffer differs owing to the coverage, which causes a significant difference in terms of energy consumption. As a result, our proposed model showed an 18% performance improvement over the two aforementioned models. In addition, when compared to the Markov and VLDP prefetchers, our proposed model shows an extremely improved performance in the cases of *gcolor* and *kcore*. These two workloads have the most irregular access pattern among those evaluated. This result shows that proposed model reacts better to irregular patterns than the other models. In addition, these two models have many opportunities to perform prefetching when accesses to other correlated lines occur. Therefore, this means that the prefetch coverage has been increased significantly owing to module B. As a result, our proposed model achieved an average improvement of 6% and 2.3%, respectively, in terms of energy consumption compared to the Markov and VLDP prefetchers.

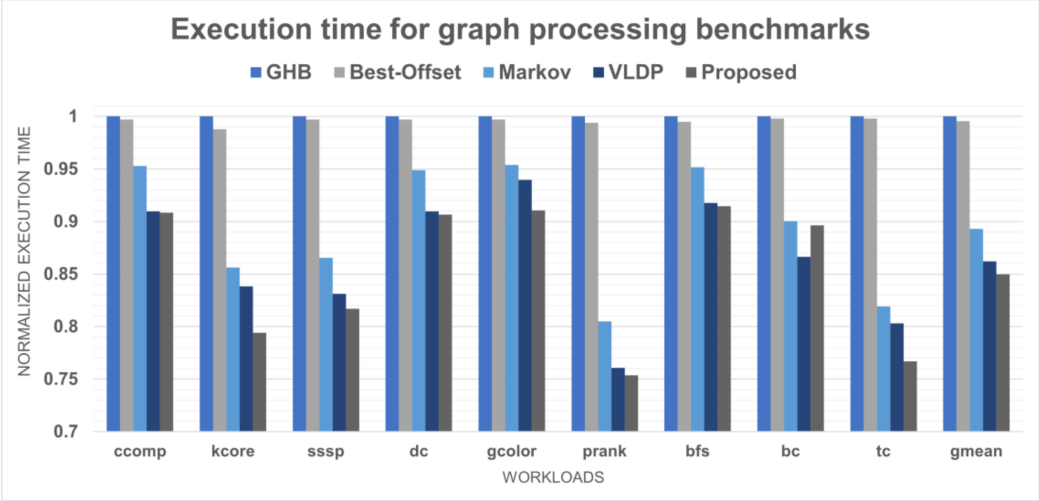


Figure 19. Execution time for graph processing workloads.

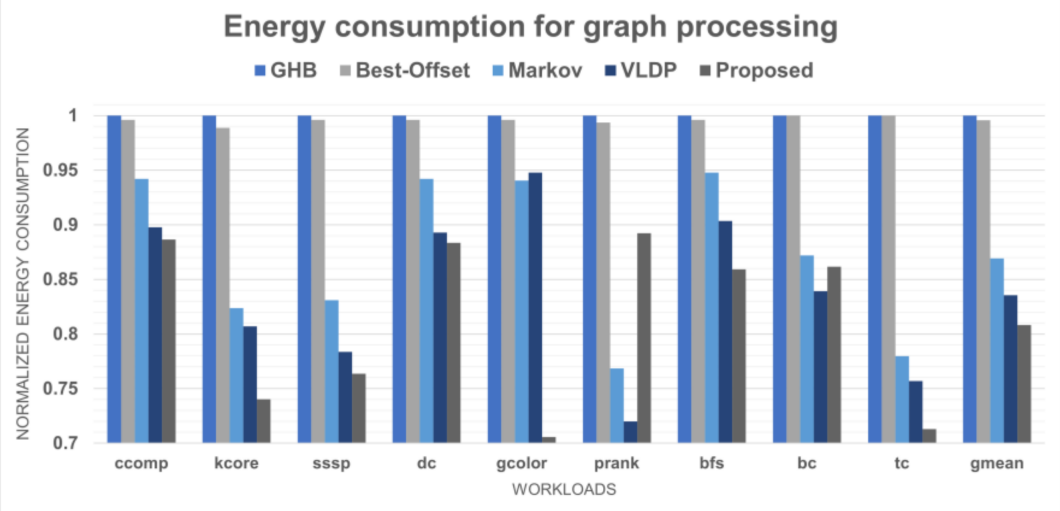


Figure 20. Energy consumption for graph processing workloads.

5. Conclusions

Modern computing architectures still suffer from high main memory latency. Hence, we propose a novel prefetching methodology that analyzes the memory access patterns by introducing the concept of adaptive granularities in memory access streams. By conducting prefetch operations when more program contexts are gathered and sufficiently long patterns are requested, we can expect a higher prefetching accuracy, although the coverage decreases. By contrast, performing prefetching without considering the accuracy will cause an increase in the memory bandwidth and incur larger prefetching costs. Hence, the most appropriate criterion regarding whether to perform a prefetching needs to be determined.

To resolve this problem, we adjusted the intensity of the prefetching based on the case in which the current and previous accesses corresponded to different correlated lines. When this occurs on the same correlated line, we classify it as a consecutive regular access, and prefetching will be performed to increase the accuracy. However, if memory accesses to other correlated lines are requested, we assume that there are insufficient grounds to apply an accurate prefetching, and we therefore perform prefetching aggressively even if the accuracy decreases.

To this end, we proposed a novel prefetching method that consists of three submodules to achieve a high accuracy and high coverage. The first prefetcher applies prefetching by analyzing the access pattern at the correlated line, which is the smallest unit. The second module analyzes memory access patterns on a page granularity that is larger than the correlated line range. The third module performs prefetching by considering the correlation between consecutive deltas in the history table without setting any limits.

By exploiting these strategies for prefetching via adopting the three proposed submodules of the prefetching units with embedded DRAM-based buffers, our proposed model outperforms previous prefetcher models. Compared to the GHB and BO prefetchers, our model shows 18% and 15% performance improvements in terms of energy consumption and execution time, respectively. Furthermore, our model reduces the total execution time and energy consumption by approximately 6% and 2.3% compared to the Markov and VLDP prefetchers. In addition, because we employed the embedded DRAM technology for the prefetching buffer to reduce costs and the area of the history table, the proposed prefetcher method and architecture can be easily introduced into common computing systems without expensive peripheral logic and storage spaces.

Author Contributions: Conceptualization, S.-G.C. and J.-G.K.; methodology, S.-G.C.; validation, J.-G.K. and S.-D.K.; investigation, J.-G.K.; resources, J.-G.K. and S.-D.K.; writing—original draft preparation, S.-G.C.; writing—review and editing, J.-G.K. and S.-D.K.; supervision, S.-D.K.; funding acquisition, S.-D.K. All authors have read and agreed to the published version of the manuscript. And this article is based on the revision and extended version of co-first author (S.-G.C.)'s master's thesis (unpublished work) from Yonsei University [20].

Funding: This work was supported by the National Research Foundation of Korea funded by the Korean government (MSIT) (NRF-2019R1A2C1008716).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The additional experiment data are not publicly available due to all the experiment data are already declared throughout this article's figures and tables.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Nesbit, K.J.; Smith, J.E. Data cache prefetching using a global history buffer. In Proceedings of the IEEE 10th International Symposium on High Performance Computer Architecture (HPCA'04), Madrid, Spain, 14–18 February 2004.
2. Michaud, P. Best-offset hardware prefetching. In Proceedings of the 2016 IEEE International Symposium on High Perfor-mance Computer Architecture (HPCA), Barcelona, Spain, 12–16 March 2016.

3. Pugsley, S.H.; Chishti, Z.; Wilkerson, C.; Chuang, P. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014.
4. Fu, J.W.C.; Patel, J.H.; Janssens, B.L. Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsl.* **1992**, *23*, 102–110. [CrossRef]
5. Srinath, S.; Mutlu, O.; Kim, H.; Patt, Y.N. Feedback directed prefetching: Improving the performance and band-width-efficiency of hardware prefetchers. In Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, Scottsdale, AZ, USA, 10–14 February 2007.
6. Shevgoor, M.; Koladiya, S.; Balasubramonian, R.; Wilkerson, C.; Pugsley, S.H.; Chishti, Z. Efficiently prefetching complex address patterns. In Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Waikiki, HI, USA, 5–9 December 2015.
7. Kondguli, S.; Huang, M. Division of labor: A more effective approach to prefetching. In Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 1–6 June 2018.
8. Ishii, Y.; Inaba, M.; Hiraki, K. Access map pattern matching for high performance data cache prefetch. *J. Instr.-Level Parallelism* **2011**, *13*, 1–24.
9. Young, V.; Krishna, A. Towards Bandwidth-Efficient Prefetching with Slim AMPM. In Proceedings of the 2nd Data Prefetching Championship, Portland, OR, USA, 13 June 2015.
10. Youngjoo, S.; Hyungchan, K.; Dokeun, K.; Jihoon, J.; Junbeom, H. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada, 15–19 October 2018.
11. Bakhshalipour, M.; Shakerinava, M.; Lotfi-Kamran, P.; Sarbazi-Azad, H. Bingo Spatial Data Prefetcher. In Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 16–20 February 2019.
12. Bakhshalipour, M.; Lotfi-Kamran, P.; Sarbazi-Azad, H. Domino Temporal Data Prefetcher. In Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Vienna, Austria, 24–28 February 2018.
13. Khoshavi, N.; Demara, R.F. Read-Tuned STT-RAM and eDRAM Cache Hierarchies for Throughput and Energy Optimization. *IEEE Access* **2018**, *6*, 14576–14590. Available online: https://ieeexplore.ieee.org/document/8308725 (accessed on 30 October 2020). [CrossRef]
14. Jouppi, N.P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Comp. Arch. News* **1990**, *18*, 364–373. [CrossRef]
15. Joseph, D.; Grunwald, D. Prefetching using Markov predictors. *IEEE Trans. Comput.* **1999**, *48*, 121–133. [CrossRef]
16. Ainsworth, S.; Jones, M.J. Graph prefetching using data structure knowledge. In Proceedings of the 2016 International Conference on Supercomputing, Istanbul, Turkey, 1–3 June 2016.
17. Qureshi, K.M.; Srinivasan, V.; Rivers, J.A. Scalable high performance main memory system using phase-change memory technology. In Proceedings of the 36th annual international symposium on Computer architecture (ISCA), Austin, TX, USA, 20–24 June 2009.
18. Agrawal, A.; Ansari, A.; Torrellas, J. Mosaic: Exploiting the spatial locality of process variation to reduce refresh energy in on-chip eDRAM modules. In Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014.
19. Nai, L. GraphBIG: Understanding graph computing in the context of industrial solutions. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, TX, USA, 15–20 November 2015.
20. Choi, S.-G. Prefetching Method Analyzing Memory Access Pattern by Adding the Concept of Layer. Master's Thesis, Yonsei University, Seoul, Korea, August 2020.