Journal of Instruction-Level Parallelism 13 (2011) 1-24 Submitted 3/10; published 1/11

Access Map Pattern Matching for High Performance Data Cache Prefetch

Yasuo Ishii Y-ISHII@BC.JP.NEC.COM
NEC Corporation
1-10, Nisshin-cho
Fuchu-shi, Tokyo 183-8501, Japan

Mary Inaba MARY@IS.S.U-TOKYO.AC.JP
Kei Hiraki HIRAKI@IS.S.U-TOKYO.AC.JP
Graduate School of Information Science and Technology
The University of Tokyo
7-3-1, Hongo
Bunkyo-ku, Tokyo 113-8656, Japan

Abstract

Hardware data prefetching is widely adopted to hide long memory latency. A hardware data prefetcher predicts the memory address that will be accessed in the near future and fetches the data at the predicted address into the cache memory in advance. To detect memory access patterns such as a constant stride, most existing prefetchers use differences between addresses in a sequence of memory accesses. However, prefetching based on the differences often fail to detect memory access patterns when aggressive optimizations are applied. For example, out-of-order execution changes the memory access order. It causes inaccurate prediction because the sequence of memory addresses used to calculate the difference are changed by the optimization.

To overcome the problems of existing prefetchers, we propose Access Map Pattern Matching (AMPM). The AMPM prefetcher has two key components: a memory access map and hardware pattern matching logic. The memory access map is a bitmap-like data structure for holding past memory accesses. The AMPM prefetcher divides the memory address space into memory regions of a fixed size. The memory access map is mapped to the memory region. Each entry in the bitmap-like data structure is mapped to each cache line in the region. Once the bitmap is mapped to the memory region, the entry records whether the corresponding line has already been accessed or not. The AMPM prefetcher detects memory access patterns from the bitmap-like data structure that is mapped to the accessed region. The hardware pattern matching logic is used to detect stride access patterns in the memory access map. The result of pattern matching is affected by neither the memory access order nor the instruction addresses because the bitmap-like data structure holds neither the information that reveals the memory access order of past memory accesses nor the instruction addresses. Therefore, the AMPM prefetcher achieves high performance even when such aggressive optimizations are applied.

The AMPM prefetcher is evaluated by performing cycle-accurate simulations using the memory-intensive benchmarks in the SPEC CPU2006 and the NAS Parallel Benchmark. In an aggressively optimized environment, the AMPM prefetcher improves prefetch coverage, while the other state-of-the-art prefetcher degrades the prefetch coverage significantly. As a result, the AMPM prefetcher increases IPC by 32.4% compared to state-of-the-art prefetcher.

1. Introduction

With the progress of the semiconductor process technology, the processor clock cycle time has been significantly reduced. However, the decrease in off-chip memory access time has been much less than that in the processor clock cycle time because the improvement in off-chip memory technology has primarily resulted in a large memory capacity. As a result, the off-chip memory latency of modern processor chips has become more than hundred of processor clock cycles. The pipeline stall time due to one cache miss often becomes more than hundred of processor cycles. This prevents the effective use of processor cores and memory bandwidth.

Cache memories are used for reducing the average memory access time. For future memory access, cache memories store recently accessed data and their neighbors. However, cache memories are not effective when the data stored in the cache memory are not accessed repeatedly. To enhance the cache memory performance, hardware data prefetching mechanisms have been proposed [1, 2, 3, 4], and several mechanisms are already being used in commercial processors [5]. The hardware data prefetcher predicts the memory address that will be accessed in the near future and fetches the data at the predicted address into the cache memory in advance. When the prefetch address is predicted correctly and the prefetch request is issued sufficiently early to corresponding memory requests, hundreds of cycles of off-chip memory can be hidden and the pipeline stall time due to off-chip memory access is eliminated.

Many prefetching methods have been proposed in order to hide the off-chip memory access latency [1, 5]. Sequential prefetchers are known as the simplest prefetchers. They fetch the cache lines that immediately follow the cache line that was accessed last. More advanced prefetching methods involve the use of a prediction table. The table is indexed by the instruction addresses or the address of the previous memory access. The prefetcher detects particular memory access patterns from the memory access history recorded in the table. The stride prefetcher detects a constant stride [2]. The Markov prefetcher detects probabilistic address correlation [3]. Since these prefetchers support only simple memory access patterns, the performance improvement achieved by using these prefetchers is limited. Prefetchers in which a global history buffer (GHB) is used are known as state-of-the-art prefetchers [4, 6, 7]. A GHB holds all recent memory accesses in a FIFO buffer and uses a linked list to store recently accessed memory addresses, which are later used to detect memory access patterns. This prefetching technique is more effective than the above-mentioned simple table-based prefetching methods because the hardware resource is used to store recently missed addresses. However, this technique involves sequential accesses and modifications to the linked list in almost all memory accesses. This results in long latency in the prediction of addresses to be prefetched.

Unfortunately, existing prefetchers suffer from one or more of following drawbacks: (1) The performance improvement is low when the prefetchers detect only simple memory access patterns (e.g., sequential prefetchers). The simple prefetchers such as sequential prefetchers achieve insufficient prefetch coverage. This is due to the prefetcher cannot detect more complex memory access patterns. (2) The latency in address prediction is long when the prefetchers cannot issue prefetch requests until the corresponding prefetch states become steady (e.g., stride prefetcher using reference prediction table). The stride prefetcher [2] cannot issue prefetch requests before the corresponding prefetch state of the prediction table become the steady state. The long latency in address prediction suppresses the rate of prefetch requests from prediction. (3) Large amount of hardware resources is required to record past memory accesses (e.g., stride, Markov, and GHB prefetchers). For predicting future memory accesses, most prefetchers hold address offsets of past memory accesses. Such prefetchers require fixed amount of history space for holding the information on each past memory access. (4) The modification and alternation of the order in the memory address sequence due to optimization techniques such as out-of-order execution often prevent the prefetcher from detecting memory access pattern accurately (e.g., stride, Markov, and GHB prefetchers). Prefetchers that predict memory access patterns on the basis of differences between the addresses in a sequence of consecutive memory accesses suffer from this drawback because the address sequence, which is used to calculate the difference, is changed by the optimizations such as out-of-order execution.

In this paper, we propose a new data prefetching method: access map pattern matching (AMPM). The AMPM prefetcher has two key components, namely, a memory access map and hardware pattern matching logic. A memory access map is a bitmap-like data structure for holding past memory accesses. The AMPM prefetcher divides the memory address space into memory regions of a fixed size. The memory access map is mapped to the memory region. Each entry in the bitmap-like data structure is mapped to each cache line in the region. Once the bitmap is mapped to the memory region, each entry in the bitmap-like data structure records whether the corresponding line has already been accessed or not. The hardware pattern matching logic detects memory access patterns from the memory access map.

The AMPM prefetcher helps to overcome the drawbacks of existing prefetcher in the following manner: (1) The AMPM prefetcher achieves the high performance because it improves the prefetch coverage significantly. The AMPM prefetcher achieves high prefetch coverage because the pattern matching detects all possible strides at a time. From the detected stride patterns, the AMPM prefetcher predicts more future memory accesses than that of existing prefetchers do. (2) The latency in address prediction is low because the hardware pattern matching logic can detect all possible memory access patterns immediately. The memory access map can issue prefetch requests when it detects memory access patterns in the memory access map. (3) The hardware cost for the prefetching mechanism is reasonable. The memory access map uses a bitmap-like data structure that can record a large number of memory accesses within a limited hardware budget. The pattern matching logic is composed of basic arithmetic units such as adders and shifters. (4) The prefetcher is robust to modification and alternation of the order in the memory address sequence. The AMPM prefetcher detects prefetch candidates under an aggressively optimized environment because the memory access map is affected by neither the memory access order nor the instruction address.

This paper is organized as follows: In section 2, our motivation for undertaking this study is presented. In section 3, the design overview, data structure, and the algorithm of the AMPM prefetcher are described. In section 4, the hardware implementation and the complexity of the AMPM prefetcher are discussed. In section 5, the evaluation methodology and experimental results are presented. In section 6, related studies are presented, and finally, in section 7, the paper is concluded.

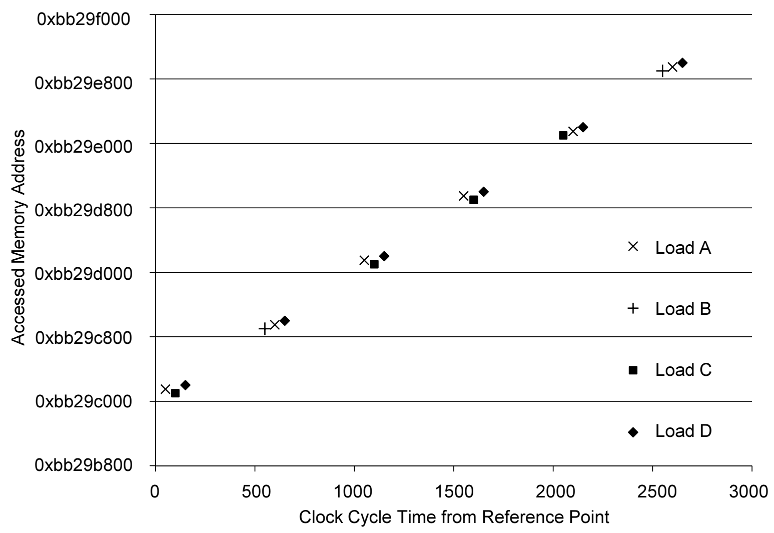

Figure 1: Memory Access Pattern of milc in SPEC CPU2006.

2. Background and Motivation

As described in the previous section, prefetchers often interfere with the optimization techniques such as loop unrolling and out-of-order execution. In this section, we discuss such interference in detail. One of our motivations for undertaking this work is the need to design a prefetcher that is compatible with other optimization techniques and whose performance is not degraded under an aggressively optimized environment.

2.1. Interference by Other Optimizations

To effectively detect the memory access pattern, existing prefetchers classify memory accesses according to the instruction that initiates the memory access. This is helpful for detecting the constant stride access pattern of a load instruction in a loop structure. However, when a compiler duplicates memory access instructions by static instruction scheduling such as loop unrolling, the locality of particular memory access instructions is reduced. This means that the optimization techniques such as loop unrolling interferes with the prefetcher that uses the addresses of memory access instructions that initiate a cache miss.

To detect memory access patterns, existing prefetchers use differences between addresses in a sequence of consecutive memory accesses. This strategy allows the prefetcher to effectively detect several memory access patterns such as constant strides and delta correlations. However, the prefetcher cannot detect memory access patterns accurately when the memory access order is changed from the program order. Out-of-order execution violates this restriction because the memory access order is changed by out-of-order execution.

Figure 1 shows an example of out-of-order memory access. The L2 cache miss addresses of milc in SPEC CPU2006 during 3000 processor cycles are plotted. Each symbol indicates the memory access instruction that initiates the corresponding cache miss. Four instructions and 18 cache misses are observed in Figure 1. Three cache misses that are observed in a short period form a group. Six groups can be found in the figure. The address correlation between two adjacent groups is a 2.0 KB stride access pattern and the interval time of two adjacent groups is about 500 cycles. In this memory access pattern, the memory access order in each group and the stride corresponding to each instruction vary. The properties are described in detail as follows. (1) Load C cannot be observed in the second and sixth groups because of the order of memory accesses between Load B and Load C. This prevents the prefetchers based on the reference prediction table [8] from detecting a correct stride. The prefetcher will detect not only 2.0 KB stride but also 4.0 KB stride around Load C because the Load C cannot be observed in the second and sixth groups. (2) The memory access order of the first group is medium (Load A), low (Load C), and high (Load D), while that of the fifth group is low (Load C), medium (Load A), and high (Load D). The modifications and alternations of the order in the memory address sequence prevent the prefetchers, which use address correlations of the previous memory accesses such as Markov prefetching [3], from detecting memory access patterns accurately.

2.2. Interference with Other Optimizations

Most existing prefetchers expect that actual memory accesses should not be affected by the prefetch requests. However, the actual memory accesses are often changed by the prefetch requests because some off-chip memory accesses that are initiated by cache misses are eliminated by the useful prefetches. In such cases, existing prefetchers try to reconstruct the history of past memory accesses that would have been observed if the prefetch had not been performed. To reconstruct the history of past memory accesses, many prefetchers add a bit (prefetch bit) to the cache line. A prefetch bit represents whether the corresponding line has already been fetched by the prefetch request. It is set when a prefetched line is inserted into the cache memory and unset when the corresponding line is accessed by the actual memory access. When the actual memory access hits a cache line whose prefetch bit is set, the memory access is handled as a cache miss in the prefetcher [4]. Although this approach helps to reconstruct the history of past memory accesses, it requires the prefetcher to be tightly coupled with the cache memory. This is because the prefetch bit is accessed frequently and the lookup should be completed within a short time. On the other hand, the prefetchers that use instruction addresses are required to be tightly coupled with the processor core because the instruction addresses are not typically available outside of the processor core [9]. When the prefetcher must be tightly coupled with both the cache memory and the processor core, the cache memory should be tightly coupled with the processor core. This leads to interference with the distributed cache architecture such as the NUCA.

Several adaptive prefetching techniques such as [10] involve the use the prefetch bit. Adaptive prefetching techniques evaluate the usefulness of the past prefetch requests to adjust the prefetch degree that determines the maximum number of prefetch requests at a time. When the usefulness is determined by the accuracy of the prefetch requests, the usefulness is evaluated on the basis of the prefetch bit. When the prefetch bit of the evicted cache line is set, the corresponding prefetch request is regarded as a useless prefetch because the prefetched cache line is not accessed before the corresponding memory access occurs. When the prefetch bit of the accessed cache line is set, the corresponding prefetch request is regarded as a useful prefetch because the prefetched cache line is accessed by actual memory access. The adaptive prefetching techniques collect the number of the useful prefetch requests and the useless prefetch requests to determine the usefulness of the past prefetch requests. This process interferes with the distributed cache architecture because the prefetch bit, which is the information on the cache memory, is used for adjusting the prefetch degree.

3. Access Map Pattern Matching

In this section, we propose a novel data prefetching method, namely, access map pattern matching (AMPM). As described in the previous section, the performance of the prefetch algorithm often degraded when either memory access orders or instruction addresses are used. To avoid such situations, we design the AMPM prefetcher such that it detects memory access patterns only on the basis of the memory locations that have been accessed in the past.

The AMPM prefetcher employs two key components: a memory access map and pattern matching logic. The memory access map is a bitmap-like data structure for holding memory locations that have been accessed. To predict future memory access accurately, the memory access map records only the information on the memory accesses which recently occurred because we assume that the information on the memory accesses in the distant past does not reflect the current memory access patterns. We call the period, in which the memory accesses reflect current conditions, as "recent past." Generally, the data corresponding to such recent past memory accesses have been stored on the cache memory because the corresponding data are recently accessed. The pattern matching logic is a combinational logic for detecting memory access patterns from the memory locations held in the memory access map. Since the AMPM prefetcher uses only the memory locations, it is not affected by the other optimizations such as out-of-order execution and loop unrolling.

Figure 2 shows an overview of the AMPM prefetcher. The memory address space is divided into memory regions of a fixed size; we call these regions "zones." The number of cache lines in the zone is equal to the number of the entries in the bitmap-like data structure. Recently accessed zones are called "hot zones," which are similar to the concentrated zone ($Czone$) [11]. The AMPM prefetcher uses the fixed number of hot zones. The total covered area by hot zones should not be larger than the capacity of the cache memory in order to prevent the memory access map from holding the information on the memory accesses that occur in the distant past[1].

A memory access map is mapped to each hot zone to holds memory locations that have been accessed in the recent past. Each entry in the bitmap-like data structure of the memory access map is mapped to each cache line in the corresponding hot zone. The entry records memory accesses to the corresponding cache line. The information on the history of recent past memory accesses corresponding to the entry is represented as a prefetch state. The memory access maps are stored in a table, which we call memory access map table. The memory access map is mapped to the zone when the memory accesses occur in zones to which no memory access maps are mapped. When all memory access maps are already mapped to zones, the memory access map in the table is replaced by the LRU policy. When a memory access map is evicted from the memory access map table, the information held in

1. In this work, the total covered area by the hot zones is 2.0 MB (8.0 KB zone $\times$ 256 memory access maps) while the capacity of L2 cache memory is 2.0 MB.

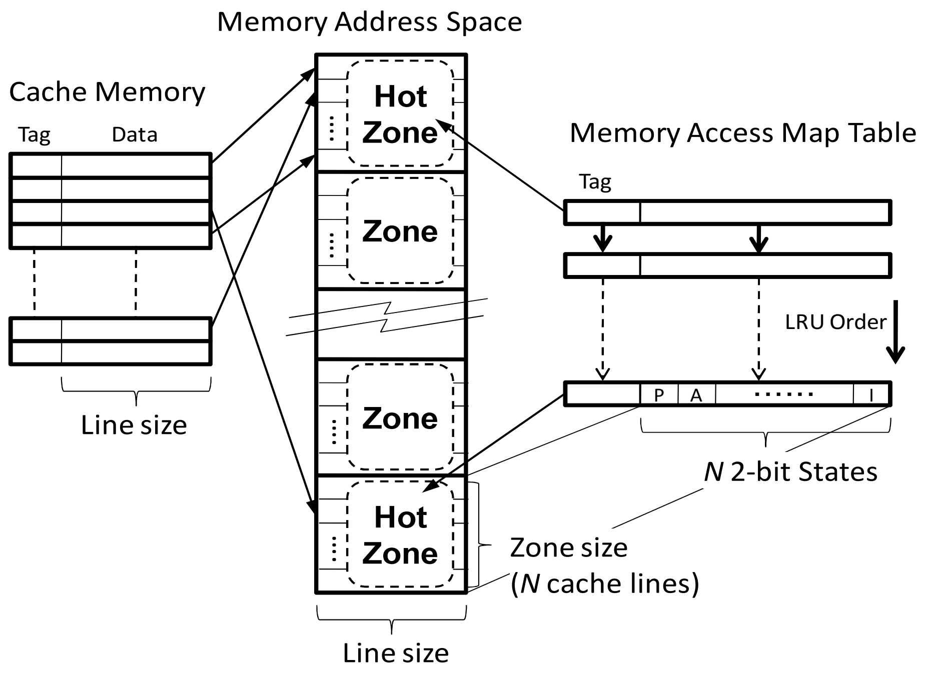

Figure 2: Overview of the AMPM Prefetch.

the memory access map is discarded. By discarding the old information on the evicted maps, the AMPM prefetcher avoids to use the information of memory accesses in the distant past for predicting future memory accesses. If the memory access map holds all past memory accesses, the AMPM prefetcher cannot detect memory access patterns effectively because the information of memory accesses in the distant past is involved to predict future memory accesses.

On a memory access, the corresponding memory access map is read from the table. The read map is sent to the pattern matching logic. The pattern matching logic generates prefetch requests and issues them to the memory subsystem.

3.1. Memory Access Map

The memory access map records memory accesses occur in each hot zone. Each entry in the bitmap-like data structure is mapped to each cache line in the hot zone. Each entry holds information on the history of recent past memory accesses to the corresponding cache line. The history of recent past memory access is recorded in a form of 2-bit state, which we call a prefetch state. The prefetch state of each cache line is one of the following three states: “Init,” “Prefetch,” and “Access.”

The state diagram of a memory access map is shown in Figure 3. When the memory access map is mapped to the hot zone, all prefetch states in the memory access map are initialized to the Init state. When the prefetch request is issued to a cache line in the Init state, the corresponding state changes to the Prefetch state. When the actual memory access is issued to the cache line, the corresponding state changes to the Access state. Collectively, the state transitions occur when (1) the actual memory access for the corresponding cache

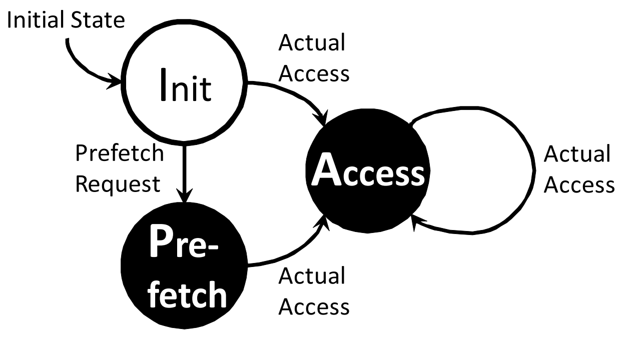

Figure 3: State Diagram of the Access Map.

line is observed or (2) the prefetch request for the corresponding cache line is issued. The AMPM prefetcher assumes that the prefetch state reflects a state of corresponding cache line because all recent past memory accesses occur in hot zones are recorded on the prefetch states. Unless the data of the memory accesses in the distant past have not been evicted from the cache, the assumption that the prefetch state reflects a state of corresponding cache line is correct.

The prefetch state indicates the state of the corresponding cache line. When the prefetch state is not the Init state, the data on the corresponding cache line have already been fetched because the prefetch request has been issued to the cache line or the actual memory access has occurred for the cache line in the recent past. Therefore, there is no need for a prefetch request to be issued to the corresponding cache line unless the fetched data are evicted from the cache memory[2]. This feature eliminates the need for cache probes to check whether the data of the prefetch target have already been fetched in the cache. In this work, the prefetch requests are issued only to the cache lines which are in the Init states in the AMPM prefetcher.

Since the state transitions are unidirectional, the number of the Access states monotonically increases until the memory access maps are replaced and the number of the Init states approaches zero. When the number of the Init states becomes zero, the AMPM prefetcher cannot issue additional prefetch requests because it can issue prefetch requests to the memory location whose prefetch state is the Init state. However, we assume that there is no need for additional prefetch requests in such situation since almost all data in the hot zones are already stored in the cache memory.

During a memory access, the lookups of the memory access map table is performed in parallel with the cache access. When the corresponding memory access map is not found in the table, the LRU memory access map is unmapped from the current mapped zone and maps the unmapped map to the new zone that corresponds to the current memory access. Then, all the prefetch states of the replaced memory access map are initialized to the Init state. Finally, the memory access map is updated to the MRU position of the LRU stack. When the corresponding memory access map is found in the table, the AMPM prefetcher reads the map from the table. The AMPM prefetcher sends the memory access

2. In our assumption, the fetched data which should be used in the future is rarely evicted in advance of the corresponding actual memory access because the long time is needed to evict the newly inserted cache line in the modern large cache memory.

map to the pattern matching logic for generating prefetch requests. The information on the prefetch requests that are generated by the pattern matching logic is sent as feedback to the corresponding memory access map. The feedback information is used to update the corresponding prefetch state. Finally, the memory access map is updated to correspond to the MRU position of the LRU stack.

3.2. Generating Prefetch Requests by Pattern Matching

The AMPM prefetcher employs a prefetch generator for generating prefetch requests. The prefetch generator employs hardware pattern matching logic for detecting memory access patterns from the memory access map. When the memory access map that is read from the memory access map table is issued to the prefetch generator, the prefetch generator tries to find the memory access pattern from the map. When the memory access patterns are found, the prefetch generator issues prefetch requests, which is generated based on the patterns, to the memory subsystem.

During an actual memory access, three consecutive memory access maps are read from the memory access map table in parallel. One is a memory access map of the accessed zone that corresponds to the actual memory access. The other two maps are maps of the adjacent zones that immediately precede and follow the accessed zone[3]. Three consecutive memory access maps are concatenated to make one large memory access map. From the concatenated map, the hardware pattern matching logic detects all possible stride access patterns.

Here, let t be the requested address and let N be the number of cache lines in one zone; the hardware pattern matching logic simultaneously checks the state of the requested addresses $t+k$, $t+2k$, and $t+2k+1$ for all possible $k=0,1,...,N/2-1$, and if the states of $t+k$ and $t+2k$ (or $t+2k+1$) are both the Access states, then $-k$ is considered to be a stride and the address $t-k$ becomes a prefetch candidate. In this manner, multiple prefetch candidates are generated in parallel. Then, the prefetch generator selects the d closest candidates to the address of the corresponding actual memory access where d is the prefetch degree. Finally, the selected prefetch requests are issued to the cache memory. The prefetch request that is nearest to the actual memory access is issued first, and the second nearest one is issued in next cycle.

Figure 4 shows an example of the prefetching scheme. When the addresses 0x01, 0x03, and 0x04 have already been accessed and an actual access for 0x05 reaches the cache memory, the prefetch generator detects the following two candidates: (1) 0x07, whose address correlation set is {0x01, 0x03, 0x05}, and (2) 0x06, whose address correlation set is {0x03, 0x04, 0x05}. In this case, the prefetch request for 0x06 is issued first since 0x06 is nearer to 0x05 than to 0x07.

3.3. Adaptive Prefetching

To control the prefetch degree, we can adopt an adaptive prefetch technique. An adaptive prefetching estimates the usefulness of the past prefetch requests and required memory bandwidth for throttling the prefetch degree. We propose an adaptive prefetching tech-

3. Let t be the address of the accessed zone and let z be the zone size; the address of one adjacent zone is represented as $t-z$ and the other zone is represented as $t+z$.

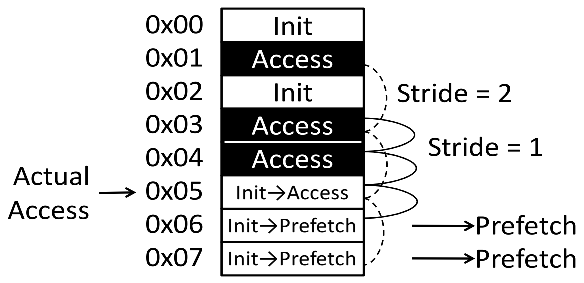

Figure 4: Pattern Matching on the Memory Access Map.

	Description	State Transition
N_{GP}	Number of Good Prefetches	From "Prefetch" to "Access"
N_{TP}	Total Number of Prefetches	From "Init" to "Prefetch"
N_{CM}	Number of Raw Cache Misses	From "Init" or "Prefetch" to "Access"
N_{CH}	Number of Raw Cache Hits	From "Access" to "Access"

Table 1: Collecting Statistics as Run-time information.

nique that is implemented on the AMPM prefetcher. The AMPM prefetcher collects four statistics to estimate the usefulness. These statistics are collected from the state transitions of the memory access map. The manner in which four state transitions can be collected is summarized in Table 1. Note that N_{CM} (raw cache miss) and N_{CH} (raw cache hit) denote the expected number of cache misses and cache hits if a prefetch is not performed. To estimate the usefulness of the past prefetch requests and the required memory bandwidth, the AMPM prefetcher estimates the prefetch accuracy, prefetch coverage, cache hit ratio, and the number of off-chip memory requests from the four state transitions that are collected. The prefetch accuracy and the prefetch coverage are defined as $P_{accuracy} = N_{GP}/N_{TP}$ and $P_{coverage} = N_{GP}/N_{CM}$, respectively. The cache miss ratio is defined as $P_{cachehit} = N_{CH}/(N_{CM} + N_{CH})$, and the number of off-chip memory requests is defined as $N_{requests} = N_{CM} - N_{GP} + N_{TP}$. The AMPM prefetcher determines these values in each fixed-length epoch (T_{epoch}). In this study, T_{epoch} is 256K processor clock cycles.

The AMPM prefetcher adjusts its prefetch degree on the basis of the estimated metrics. It provides the "required memory bandwidth" and "usefulness of the past prefetch requests" for adjusting the prefetch degree. The AMPM prefetcher limits the prefetch degree on the basis of the bandwidth-delay product, which is calculated from the required memory bandwidth, i.e., $N_{requests}/T_{epoch}$. The bandwidth-delay product is defined as $M_{bandwidth} = (N_{requests}/T_{epoch}) \times T_{latency}$, where $T_{latency}$ is the latency for the off-chip memory in processor clock cycle. The AMPM prefetcher also limits the prefetch degree on the basis of the usefulness of the past prefetch requests. The limit of the prefetch degree M_{useful} is adjusted on the basis of the statistics collected in the previous epoch. The estimated metrics of the prefetch accuracy, prefetch coverage, and cache hit ratio are compared with the corresponding threshold values. On the basis of the result of the comparison, the prefetcher throttles the prefetch degree M_{useful} using the policy shown in Table 2. The minimum value of the limits ($M_{bandwidth}$ and M_{useful}) is used as the actual maximum prefetch degree.

$P_{coverage}$	$P_{accuracy}$	$P_{cachehit}$	Action
High (> 25.0%)	*High* (> 50.0%)	*Don't care*	Increment Prefetch Degree
High (> 25.0%)	*Don't care*	*Low* (< 75.0%)	Increment Prefetch Degree
Don't care	*Low* (< 25.0%)	*High* (> 87.5%)	Decrement Prefetch Degree
Low (< 12.5%)	*Don't care*	*High* (> 87.5%)	Decrement Prefetch Degree
Low (< 12.5%)	*Low* (< 25.0%)	*Don't care*	Decrement Prefetch Degree
Other cases			No Action

Table 2: Prefetch Degree Control Performed by Collecting Run-time Information.

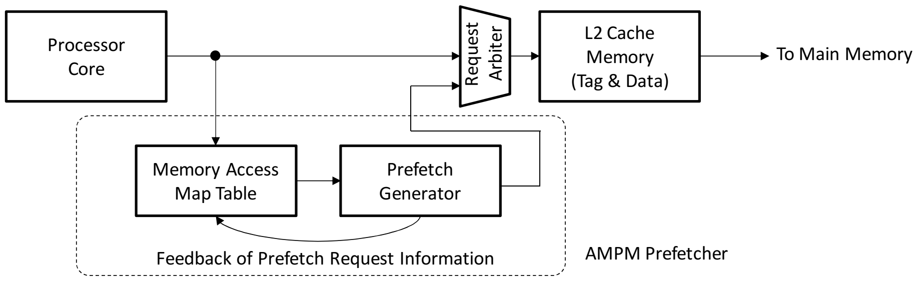

Figure 5: Block Diagram of AMPM Prefetcher.

The feature of the adaptive prefetching is that the AMPM prefetcher collects run-time information only from the memory access map, while the other adaptive prefetching methods like Feedback Directed Prefetching [10] involve the use of run-time information on other components such as the cache memory. Owing to this feature, the AMPM prefetcher is decoupled from other components such as the cache memory and the processor core.

4. Hardware Design & Complexity

4.1. Hardware Design

As shown in Figure 5, the AMPM prefetcher consists of memory access map table and a prefetch generator. The memory access map table holds memory access maps in a set-associative cache structure. The prefetch generator implements the pattern matching logic. The request path from the processor core to the L2 cache memory is also distributed to the memory access map table. When the actual memory request is issued to the L2 cache, the request is also issued to the memory access map table. Then, the corresponding memory access maps are read from the table and the read maps are sent to the prefetch generator. The prefetch generator tries to detect memory access patterns by the pattern matching. When the patterns are found, the prefetch generator makes prefetch requests on the basis of the detected memory access patterns. The generated requests are sent to the requests arbiter. The request arbiter selects the request sent to the L2 cache. In the request arbiter, the actual request has higher priority than the prefetch requests. Finally, the information on the issued prefetch request is feedbacked to the corresponding memory access map.

The advantage of the AMPM prefetcher is that the AMPM prefetcher obtains sufficient run-time information from the request path, while existing prefetchers have to collect more

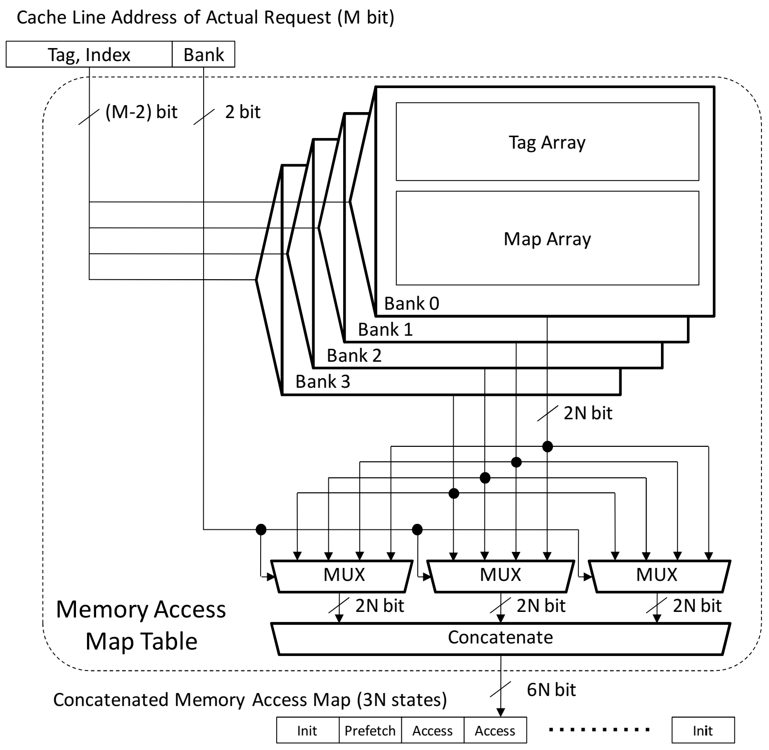

Figure 6: Four-bank Memory Access Map Table.

information from cache memories and processor cores. Thus, it allows the AMPM prefetcher can be decoupled from both the processor core and cache memories.

4.1.1. Memory Access Map Table

The four-bank configuration of the memory access map table is shown in Figure 6. The design of the memory access map table is similar to that of a multi-banked set-associative cache memory. The memory access map table is referred by an address tag, which is represented by the high-order bits of a memory address. Each memory access map holds N states, where N is the number of cache lines in one zone. The input of this component is the address of the actual memory access. The output is a concatenated map that is composed of three consecutive memory access maps. The concatenated map consists of $3N$ prefetch states.

The memory access map table has more than three banks for reading three consecutive memory access maps in parallel. Each bank consists of a tag array and a map array. The tag array holds the high-order bit of the corresponding zone address and LRU information. The map array holds 2-bit state maps of the memory access map. When a request reaches the memory access map table, the corresponding map and two adjacent maps are read from the table. The read maps are rotated by the lower bit of the address, and the rotated maps are concatenated. Finally, the concatenated map is sent to the prefetch generator.

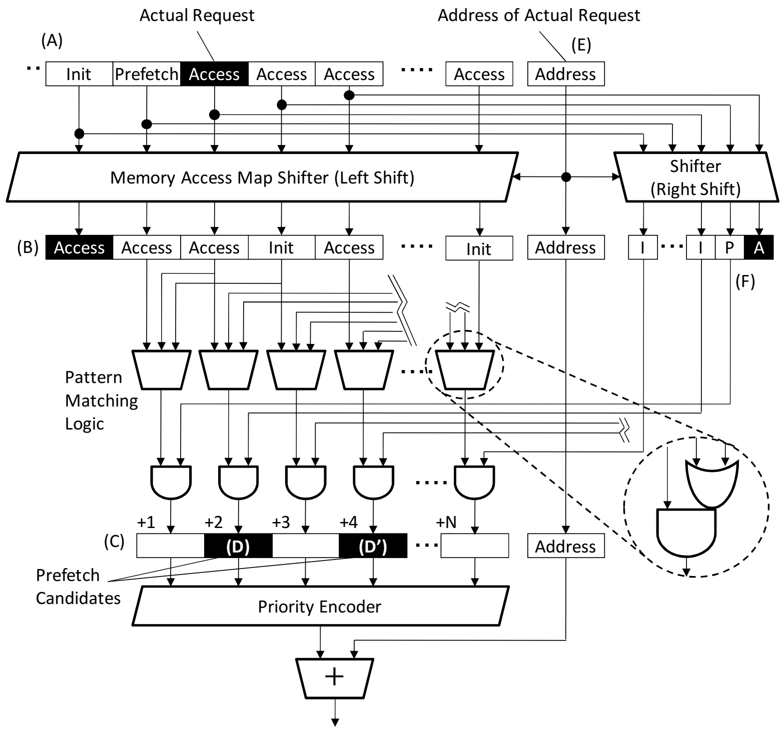

Figure 7: Pattern Matching Logic for Stride Detection.

4.1.2. Prefetch Generator

The prefetch generator is responsible for pattern matching in the AMPM prefetching. It generates prefetch requests when it receives the concatenated memory access map from the memory access map table. The prefetch generator produces one prefetch request in each cycle. Prefetch requests to the forward and the backward of the address of the actual memory access are generated separately, and thus, the prefetch generator employs a copy of pattern matching logic[4]. Figure 7 shows the block diagram of the forward prefetch generator whose zone size is N.

The prefetch requests for each direction are produced by the following steps. First, the memory access map (A) from the memory access map table is shifted for aligning the map. The state corresponding to an actual accessed address is aligned at the edge of the memory access map (B) for forward prefetching. The memory access map (F) is a shifted map for backward prefetching. Second, the pattern matching logic produces multiple prefetch candidates in parallel. These candidates are generated as a bitmap. The bitmap is stored in the pipeline register (C). The k_{th} request is generated by pattern matching of the entry at k, $2k$, and $2k+1$ in (B). N OR-AND logics of the pattern matching logic check whether the corresponding states are the Access states. When k and $2k$ (or $2k+1$) in the (B) are the Access states, the k_{th} position is identified as a prefetch candidate. The output of the pattern matching logic is filtered by the k_{th} entry in (F). This filter checks whether a state

4. The forward prefetch means the prefetch request to $t+k$ when the t is the address of actual memory request and k is positive integer. The backward prefetch means the prefetch request to $t-k$.

of prefetch target is the Init state or not. The prefetch candidate is rejected when the corresponding state in (F) is not the Init state. In Figure 7, (D) and (D') are the prefetch candidates. The nearest candidate bit (D) is selected by the priority encoder. The selected candidate bit is cleared to issue other requests in the next cycle. Then, the encoded address offset is added to the actual accessed address. Finally, the generated address is issued as a prefetch request.

4.2. Hardware Cost and Complexity

The hardware resource necessary for the AMPM prefetcher is storage for the memory access map table and a combinational circuit for the pattern matching logic. The size of the map array for the memory access map table is $2N$ bits when the map holds N states. The tag array holds the higher-order bit of memory address and LRU information. When the AMPM prefetcher uses 48-bit address, 64 states, 256 maps, an 8-way set-associative, and a 128B cache line size, the total budget size becomes 256 maps $\times$ ((2 bits $\times$ 64 states) + 35 bits (tag) + 3 bits (LRU)) = 42496 bits (approximately 5.2 KB). The complexity of the memory access map is comparable to that of the set-associative cache memory, as shown in this section. The pattern matching logic requires N-bit integer shifters for the access map shifter, N OR-AND logics for pattern matching, N-bit priority encoders, and offset adders that are composed of small adder and increment logic. All of them are primitive functions in the arithmetic logic unit (ALU). This indicates that the complexity and the hardware cost of the pattern matching logic are comparable to that of the ALU. Collectively, the hardware cost and complexity of the AMPM prefetcher is reasonable for a modern processor.

5. Evaluation

In this section, we evaluate the AMPM prefetcher with the following objectives: (1) to show that the AMPM prefetcher is effective and is particularly efficient compared to other prefetchers when other optimizations are applied, and (2) to present a detailed analysis of the AMPM prefetcher, e.g., in terms of a comparison of adaptive and fixed degree prefetch, the accuracy of metrics that are used for adaptive prefetching, and comparison of zone sizes.

5.1. Evaluation Methodology

We evaluate the AMPM prefetcher with the SESC simulator [12]. The system is a four-way superscalar processor with a large L2 cache. The main parameters are listed in Table 3. We evaluate both the in-order core and out-of-order core for evaluating the effect of an out-of-order execution. This model does not employ any prefetch buffers, and thus, the prefetch data are directly stored in a cache memory.

In the evaluated configuration, the AMPM prefetcher is constructed with an 8-way 256-entry memory access map table. The map is replaced by the LRU policy. We use 8 KB as the baseline zone size of memory access map. This implies that each zone includes 64 cache lines. In this configuration, the total budget size is 5.2 KB (see section 4.2). In the baseline prefetcher, the prefetcher attempts to issue four prefetches in one actual memory access (prefetch degree = 4).

Processor	4 Decode / 4 Issues / 4 Commits
Branch Prediction	Combined 8KB Gshare, 8KB BIM, 8KB Meta
Reorder Buffer	128 entries
Load Queue	32 entries
Store Queue	32 entries
Execution Unit	2 ALU (latency 1 cycle), 2 FPU (latency 4 cycle) 2 MEM (1 cycle for address generation)
L1 Instruction Cache Memory	32KB, 4-way 128B line, latency 1 cycle, 1 access / cycle
L1 Data Cache Memory	32KB, 4-way 128B line, latency 1 cycle, 2 access / cycle
L2 Unified Cache Memory	2MB, 8-way 128B line, latency 16 cycles, 1 access / 4 cycles
Memory System	64 MSHR, latency 400 cycles, Clock ratio 1:4, 16B memory bus

Table 3: Configuration of Processor Model.

For comparison, we evaluate the PC/DC/MG prefetcher [7]. We select PC/DC/MG as a representative of a modern high-performance prefetcher. For the prediction, PC/DC/MG uses the memory address sequences categorized by instruction addresses and a *miss graph*, which represents correlations between instructions that cause cache misses. PC/DC/MG uses (1) temporal correlations of memory access instructions, (2) memory access orders, and (3) a prefetch bit stored in the cache tag. As shown in [7], it probes the cache memory before issuing a prefetch request, and a four-cycle penalty is added to each cache probe. For comparison, we configure PC/DC/MG so that its size is equivalent to that of our AMPM configuration: 256 entries for a global history buffer (GHB) and index table (IT), and a 2.0 KB budget for prefetch bit for collecting run-time information. Consequently, the total budget size of PC/DC/MG becomes 47872 bits (approximately 5.8 KB) with a 48-bit address; the IT requires 256 entries $\times$ (48 bits (tag) + 8 bits (head pointer of GHB) + 8 bits (next stream) + 3 bits (confidence counter)), GHB requires 256 entries $\times$ (48 bits (address) + 8 bits (pointer to next entry)), and prefetch bit requires 16384 bits. The prefetcher attempts to issue 16 prefetches at once (prefetch degree = 16).

We use 15 benchmark programs selected from the SPEC CPU2006 and the NAS Parallel Benchmark for our evaluation. We use a reference input for the SPEC CPU2006 and class A for the NAS Parallel Benchmark. The criterion in benchmark selection is the number of off-chip memory requests. The omitted benchmark program has the small number of off-chip memory accesses. Each selected benchmark generates at least 1.0 M off-chip memory requests in the 1.0 G instruction program slice, which is analyzed by SimPoint [13].

We compile selected benchmarks using GCC 4.4 with two different compile options: "-O3 -fomit-frame-pointer -funroll-all-loops," which we call "*aggressive*," and "-O2," which we call "*conservative*." The most important difference between these two options is loop unrolling of the aggressive configuration. All the benchmarks are fast forwarded to evaluate the program slices that are analyzed by SimPoint [13].

5.2. Comparison with Competitive Prefetcher

We compare the AMPM prefetcher with a competitive prefetcher (PC/DC/MG). To evaluate the compatibility with other optimizations, we examine four types of configurations with different optimizations listed in Table 4.

	(a) **OO-O3**	(b) **IO-O3**	(c) **OO-O2**	(d) **IO-O2**
Core Configuration	Out-of-order core	In-order core	Out-of-order core	In-order core
Compile Option	Aggressive (-O3)	Aggressive (-O3)	Conservative (-O2)	Conservative (-O2)

Table 4: Benchmark Configurations.

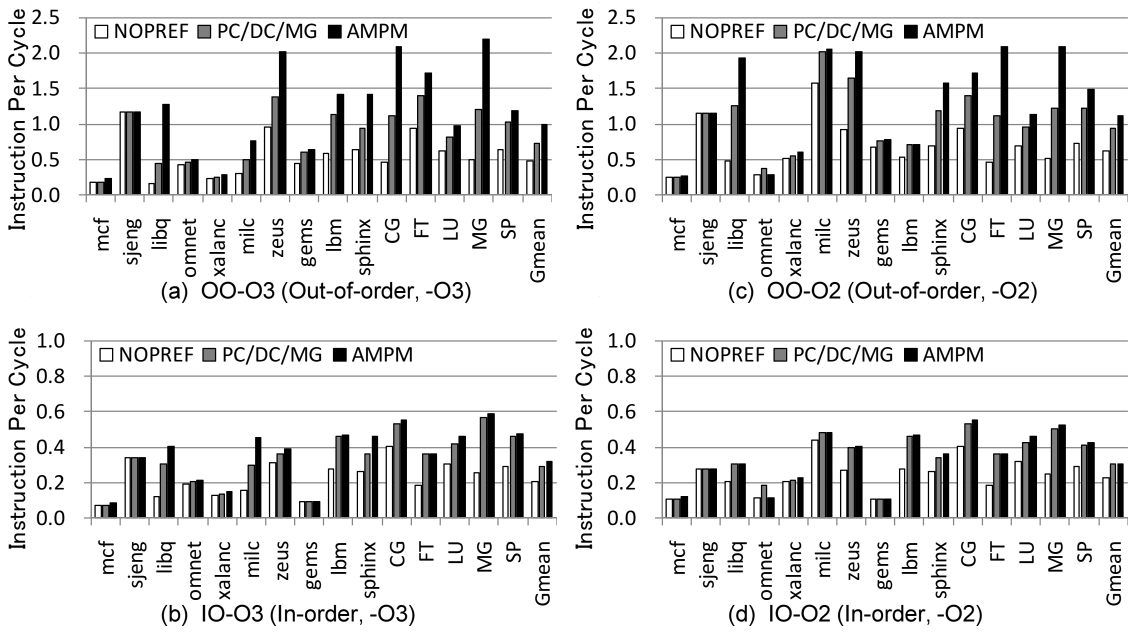

Figure 8: IPC performance with Different Configurations.

5.2.1. Overall Performance

Figure 8 shows the overall performance of AMPM, PC/DC/MG, and the performance in the case without any prefetch, which we denote NOPREF, for four configurations listed in Table 4.

In all cases, both AMPM and PC/DC/MG improve their performance compared to NOPREF. They show almost the same performance for (d) IO-O2 (in-order, -O2): AMPM degrades IPC by 0.5% from PC/DC/MG in Gmean, but AMPM attains an IPC that is better than that of PC/DC/MG by 32.4% for (a) OO-O3 (out-of-order execution, -O3). The result indicates that the AMPM prefetch is robust to other optimizations. In omnetpp, PC/DC/MG outperforms AMPM for (c) and (d) xx-O2, while AMPM outperforms PC/DC/MG in (a) and (b) xx-O3. On the other hand, in FT, PC/DC/MG outperforms AMPM in (b) and (d) IO-xx, while AMPM outperforms PC/DC/MG in (a) and (c) OO-xx. These results show that both an out-of-order execution and loop unrolling often degrade the performance in PC/DC/MG.

Note that (1) the reason why the IPC of the in-order core is lower than that of the out-of-order core is that the in-order core suffers from other bottlenecks such as instruction scheduling and (2) the reason why IPC of -O2 configuration is higher than that of -O3 configuration is that -O2 configuration includes some non-critical (or not essential) instructions such as frame-pointer handling operations. Generally, non-critical critical instructions have

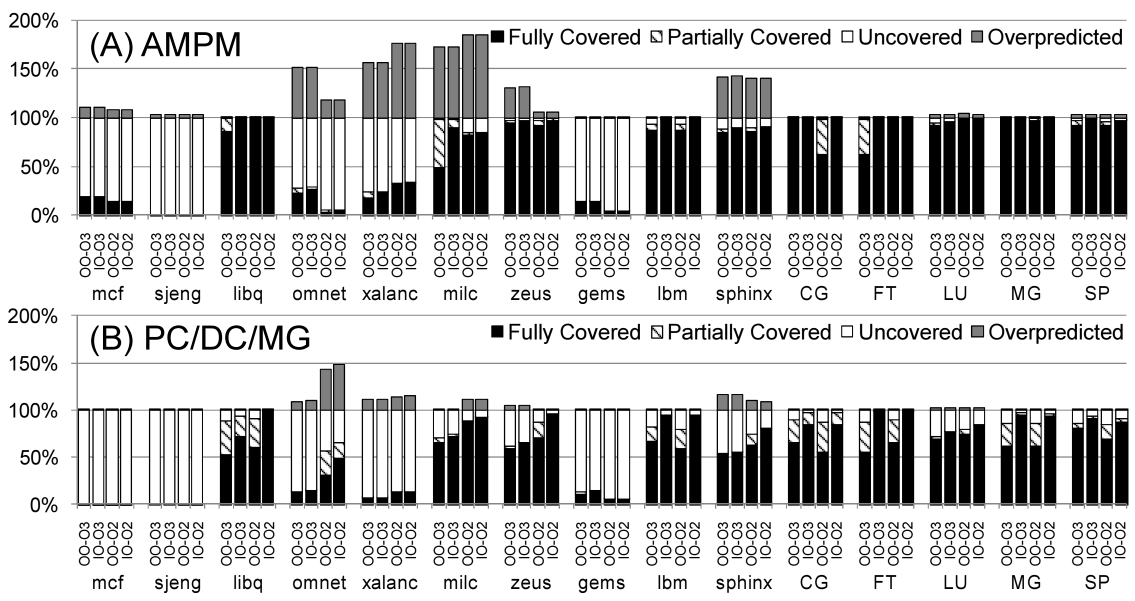

Figure 9: Analysis of Off-chip Memory Requests: AMPM (upper), PC/DC/MG (lower).

high instruction level parallelism, thus the -O2 configuration increases IPC but it does not reduce overall execution time.

5.2.2. Analysis of Off-chip Memory Requests

For further analysis of the difference shown above, we categorize all off-chip memory accesses into four groups on the basis of their prefetch coverage; “fully covered” indicates a useful prefetch which hides latency completely, “partially covered” indicates a useful prefetch which hides part of latency, “Uncovered” indicates actual memory access, and “Overpredicted” indicates useless prefetch.

Figure 9 shows the detailed analysis of off-chip memory requests. The prefetch coverage, which is the sum of fully covered and partially covered, of the AMPM exceeds 90.0% in 9 out of the 15 benchmarks for OO-O3; on the other hand, that of PC/DC/MG can not exceed than 90.0% for OO-O3.

Another feature obtained in Figure 9 is the difference in the prefetch performance among different configurations. The coverage achieved by AMPM is 5.0% greater than that achieved by PC/DC/MG in 13 out of the 15 benchmarks in OO-O3. On the other hand, the prefetch coverage achieved by AMPM is 5.0% higher than that achieved by PC/DC/MG in 5 out of the 15 benchmarks in IO-O2 because PC/DC/MG reduces the prefetch coverage in a more aggressive configuration. In 9 out of the 15 benchmarks, PC/DC/MG reduces the prefetch coverage in OO-O3 by more than 10.0% compared to that in IO-O2. In omnetpp, milc, zeusmp, and sphinx3, the compiler optimizations degrade the performance by more than 10.0% in PC/DC/MG. In lbm, FT, and MG, the out-of-order execution degrades the performance by more than 10.0% in PC/DC/MG. AMPM reduces the prefetch coverage in OO-O3 by a maximum of 5.0% from compared to that in IO-O2. Thus, it is confirmed that the AMPM prefetch helps to realize a highly robust prefetching algorithm.

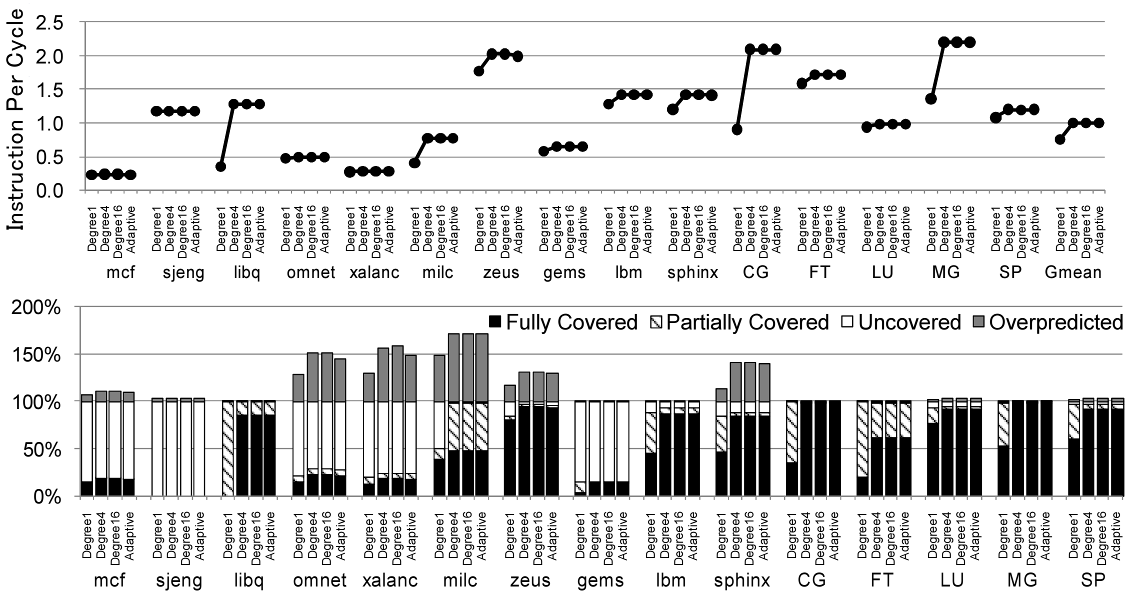

Figure 10: Performance Impact of Prefetch Degree (Prefetch Degree=1,4,16,and Adaptive).

5.3. Characteristic Evaluation of Access Map Pattern Matching

In this subsection, we evaluate (1) the impact of the adaptive prefetch on performance and (2) the impact of the zone size of the memory access maps on performance.

5.3.1. Adaptive Prefetch

As shown in the previous section, AMPM can collect run-time information to adjust the prefetch degree. The run-time information is collected only from the state transitions within the memory access map. AMPM does not probe the cache memory, unlike existing adaptive prefetching algorithms. To show this feature, we compare the adaptive prefetching and constant prefetch degrees that vary among Degree 1, Degree 4, and Degree 16.

The performance impact of adaptive prefetching is shown in Figure 10. From this figure, it can be seen that the performance is not improved among Degree 4, Degree 16, and Adaptive. With Degree 16, the highest performance is achieved, but it outperforms the other prefetch degrades by only 0.2% in geometric mean. Moreover, Degree 16 increases the number of over-predicted prefetches by 3.2% compared to Degree 4 in xalancbmk. On the other hand, the adaptive prefetching reduces the number of over-predicted prefetches by 14.5% compared to Degree 4 and Degree 16 in omnetpp and xalancbmk. Thus, it is shown that adaptive prefetching reduces the number of useless prefetches without degrading the performance.

The other feature of the adaptive prefetching of AMPM is that the prefetcher collects run-time information only from its own state transitions. We collect estimated metrics ($P_{coverage}$, $P_{accuracy}$, and $P_{cachehit}$) and actual metrics (prefetch coverage, prefetch accuracy, and cache hit ratio) and plot the relationship in Figure 11. We evaluate it with the adaptive prefetching configuration. Each point represents the relationship of each bench-

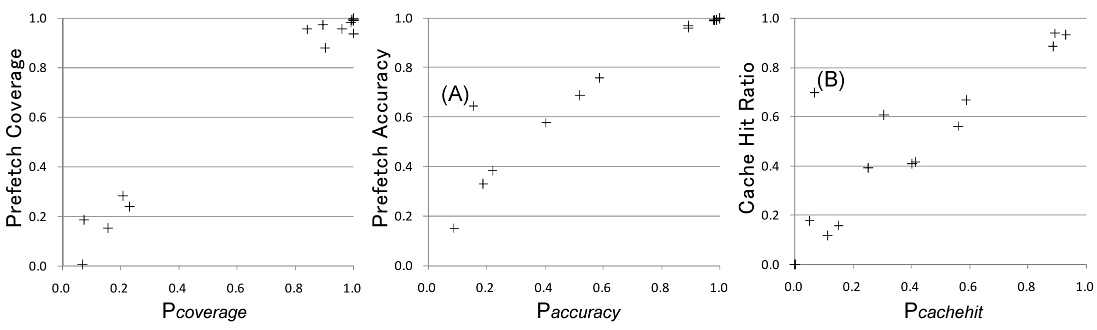

Figure 11: Correlation between estimated metrics and actual metrics.

mark. The results reveal correlations between the estimated metrics and the actual metrics. We evaluate the relationships by the regression analysis method. The analysis results show that the coefficient of correlation is 0.98 for prefetch coverage, 0.94 for prefetch accuracy, and 0.87 for the cache hit ratio. Qualitatively, metrics generated by AMPM are more negative estimates than the actual metrics. This is because of the undesirable replacement of the memory access map. (A) and (B) in Figure 11 show that the estimated metrics are much more negative than the actual metrics. These points are observed in mcf. In mcf, numerous off-chip memory requests are issued to a large area, and therefore, the access maps are replaced within a very short interval. When the memory access map is replaced too frequently, the prefetcher cannot count N_{GP}. It is because the memory access maps are evicted before the state of the corresponding cache line change from the Prefetch state to the Access state.

5.3.2. Zone Size

Figure 12 shows the impact of the zone size on performance. In this figure, we evaluate zones of size 2 KB, 8 KB (baseline), 32 KB, and 128 KB. The total number of employed memory access maps is adjusted to fit the total budget size (approximately 5.5 KB). The figure indicates that the total number of memory access maps is inversely proportional to the zone size. In this evaluation, the prefetch degree is adjusted by adaptive prefetching.

The results show that the optimal zone size differs for different benchmarks. For a size of 2 KB, libquantum, FT, and SP achieve the highest performance. In such a case, a large area is accessed in a short time. This leads to undesired replacement of memory access maps when the zone size is large because the total map count for the 2 KB zone is 64 times larger than that for the 128 KB zone. On the other hand, CG achieves the highest performance for the 128 KB zone. This is because of the large stride that cannot be detected when the zone size is small.

The other interesting case is that of GemsFDTD. The achieved prefetch coverage is 90.5% in the 2 KB zone and 69.1% in the 128 KB zone, as compared to 14.4% in the 8 KB zone and 14.3% in the 32 KB zone. This is due to the large stride access pattern in multiple loops. This large stride can be detected only when the zone size is 128 KB. Therefore, higher coverage is achieved in the 128 KB zone than in the 8 KB zone and the 32 KB zone. However, this higher coverage cannot help to improve the performance because the detection by the large stride is not early enough to completely hide the latency. On

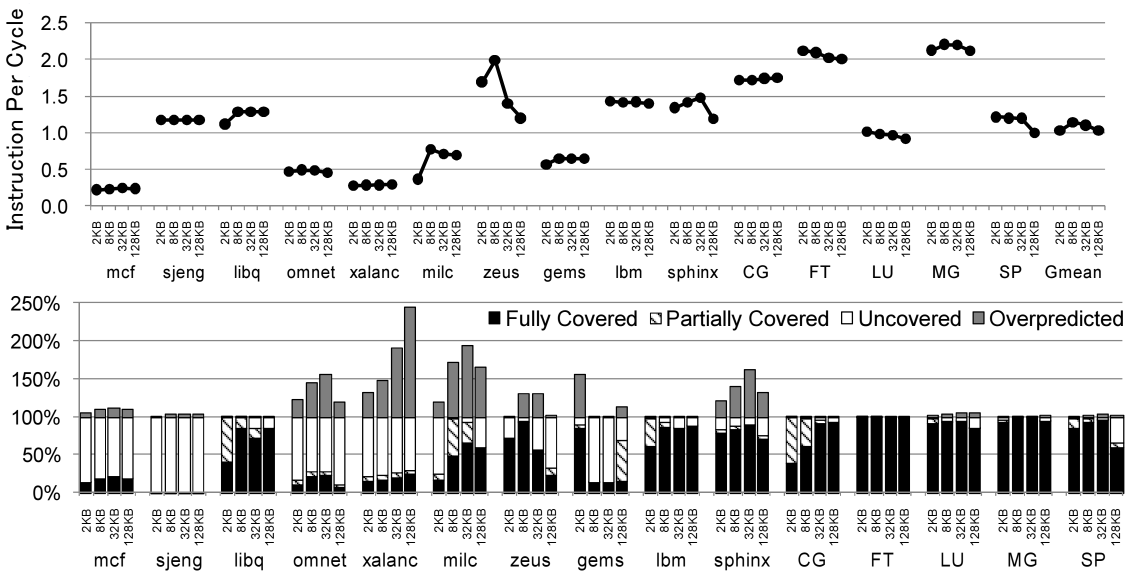

Figure 12: Performance Impact of Zone Size.

the other hand, a greater number of the prefetch requests are generated in the 2 KB zone than in the 128 KB zone. This is because the outer loop is not large in this program and AMPM holds sufficient entries to cover the entire inner stride in 2 KB zone. Therefore, the memory access map is retained throughout the iteration of the innermost loop, while the maps for larger zones are replaced. Then, memory accesses of the next iteration of the multiple loops reach neighbors of the previous iteration. In other words, the access patterns in the outer loop become multiple simple stream accesses to the consecutive area. As a result, the prefetch coverage is well improved. Unfortunately, in this case, a large number of useless prefetches are generated, and the memory bus is saturated with useless traffic. This prevents performance improvement in the case of GemsFDTD when the zone size is 2 KB. This evaluation shows that there are advantages and disadvantages for each zone size, but the 8 KB zone is adequate to satisfy many benchmarks.

6. Related Works

6.1. Hardware-Based Prefetching

Many prefetching methods have been proposed to hide the long off-chip memory access latency [14]. In some of these methods, software support is used to issue prefetches [15, 16], while the others are strictly hardware-based methods.

A sequential prefetcher [1] is the simplest hardware-based prefetcher. When a cache miss is detected, the sequential prefetcher issues requests that obtain the following cache lines of the accessed cache line. This prefetcher requires low-cost hardware and is already used in commercial systems [5], but it improves the performance when application programs access consecutive areas. More advanced prefetchers employ prefetching tables [2] to hold memory access histories. The access key to these tables is an instruction address or data address that causes a cache miss. In stride prefetching, a combination of the last memory address that causes a cache miss and an address stride is stored in a prefetch table entry. When the same stride pattern is repeatedly detected in the entry, the prefetcher issues prefetch requests to access the memory address with the same stride. In Markov prefetching [3], the address correlation in a state transition is used for issuing prefetch requests.

In order to improve the efficiency of prefetching tables, global history buffers have been proposed [4]. GHBs hold all cache miss requests in a circular buffer. The oldest entry is overwritten when a new request is inserted. Each entry is classfied by the instruction address and the memory region. Each entry in a GHB has a pointer to the next entry to keep track of the history. The memory accesses are managed in linked lists. In this data structure, the hardware resource focuses on recently missed addresses. This technique is more effective than table-based prefetching and has many extensions. PC/DC/MG [7] is one of the representative extensions of GHB. It exploits the correlation between multiple streams for improving prefetch performance. The C/DC prefetcher [6] is also one of the extensions of the GHB. The C/DC uses memory region ($Czone$) for classifying the memory accesses. However, the C/DC cannot use zone concatenation that the AMPM prefetcher uses. The C/DC fails to detect the prefetch opportunity on a border between the zones.

Spatial memory streaming (SMS) has been proposed in [17], and it is extended to combining temporal patterns [18]. It holds spatial memory access patterns to predict the future memory access patterns. In the case, high prefetch coverage can be achieved in commercial jobs such as database benchmarks. This approach appears to be similar to that adopted in the AMPM prefetcher. However, there are two differences between AMPM and SMS. First, the address of memory access instructions are used as the triggers of the prefetch in SMS. As described in section 2, the prefetcher using instruction address degrades its performance with existing optimizations such as loop unrolling. Second, the pattern history table for SMS requires a large budget for improving performance, and it must be stored in off-chip memory. It increases the complexity of memory subsystem. On the other hand, the AMPM prefetcher uses reasonable cost (approximately 5.2KB) for its prediction, and therefore, the all budget for the AMPM prefetcher can be implemented with the on-chip memory. The spatial memory streaming with rotated patterns [19] tries to reduce the pattern history table size, but it could not achieve higher performance in the first data prefetching championship [20].

6.2. Control of Prefetching Degree

Enhancements of the performance of existing prefetchers have been proposed in several studies. The AC/DC prefetcher [6] is an extensions of C/DC. The AD/DC prefetcher controls not only the prefetch degree but also the zone size for improving performance. Adaptive zone size will be also useful for the AMPM prefetcher, and it will be investigated in one of our future works. The Feedback Directed Prefetching (FDP) [10] adjusts the prefetch degree and prefetch distance on the basis of the prefetch accuracy, timeliness, and cache pollution caused by the prefetcher. To collect run-time information, the FDP adds several flags to the cache memory. It increases the complexity of the cache memory and requires the prefetcher to be tightly coupled with the cache memory. The Adaptive Stream Prefetcher [21] adjusts the prefetch degree on the basis of the probability technique that estimates stream length. It is extended in recent study [22]. The extended technique adjusts the prefetch degree without the status of the cache memory, and thus, it involves less restrictions on other components. Focused prefetching [23] detects the load instructions that cause the ROB commit stall and focuses hardware resources to avoid the commit stall. It uses the memory access instructions to detect the LIMCOS (Load Incurring Majority of COmmit Stall). As shown in this paper, localities of the LIMCOS are reduced by loop unrolling. The performance improvement are also reduced in such a case.

These techniques are similar to our adaptive prefetching technique. However, these prefetching techniques, except those in the adaptive stream prefetcher, use either the localities of instruction addresses or prefetch bits. To avoid the interference with the existing optimizations, these adaptive prefetching techniques also should be robust to existing optimizations.

7. Conclusion

In this paper, we propose a new prefetching method: access map pattern matching (AMPM). The AMPM prefetcher has two key components: a memory access map and hardware pattern matching logic. It detects all possible stride patterns in parallel. The pattern matching is robust to optimization techniques such as out-of-order execution because the pattern matching uses neither the memory access order nor the instruction address. Therefore, the AMPM prefetcher achieves high performance in an aggressively optimized environment.

The AMPM prefetcher has the following advantages over existing prefetchers: (1) It achieves better coverage than existing prefetching mechanisms. It hides off-chip memory access latency and improves performance. The experimental results show that the performance of the AMPM prefetcher with a 5.2 KB budget is better than that of the PC/DC/MG with a 5.8 KB budget by 32.4% in an aggressively optimized environment; the AMPM prefetcher achieves better prefetch coverage than the PC/DC/MG. The prefetch coverage of the AMPM prefetcher exceeds 90% in 9 out of the 15 benchmarks. The adaptive prefetching mechanism reduces the number of over-predicted prefetch requests. (2) As shown by the experimental results, the AMPM prefetcher does not degrade the prediction performance in an aggressively optimized environment where the memory access order changes, while the existing prefetchers degrade the prefetch performance under the same conditions. This is because the AMPM prefetcher eliminates restrictions via the instruction address and memory access orders. As a result, the AMPM prefetcher achieves higher prefetch coverage and better performance than do existing prefetchers under an aggressively optimized environment. (3) The AMPM prefetcher can generate prefetch requests without probing the cache memory for obtaining run-time information such as the prefetch bit. The AMPM prefetcher estimates the cache status from the statistics of the state transition of the memory access maps because the memory access map holds sufficient information for estimating the cache status. It allows the prefetcher to be decoupled from the cache tags because the prefetcher need not communicate with the cache memory. The prefetcher need not obtain instruction addresses from the processor core. This is advantageous for the design of distributed architectures such as the non-uniform cache architecture (NUCA).

As described in this paper, the AMPM prefetch hides the memory access latency for many benchmarks. The AMPM prefetcher enables us to eliminate the performance bottlenecks due to the memory latency problem. The basic idea of the AMPM prefetcher has recently been proposed [24] in the first data prefetching championship competition (DPC-1) [20]. In this competition, the prefetcher based on the concept of the AMPM prefetcher exhibited the highest performance among the finalists. The comparative study reported in this paper and the result of the DPC-1 shows that the AMPM prefetcher described in this paper is considered to be the most effective prefetcher for a general workload such as the SPEC CPU2006 and the NAS Parallel Benchmark.

References

[1] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[2] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *SIGMICRO Newsl.*, vol. 23, no. 1-2, pp. 102–110, 1992.

[3] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pp. 252–263, 1997.

[4] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, p. 96, 2004.

[5] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "Ibm power6 microarchitecture," *IBM J. Res. Dev.*, vol. 51, no. 6, pp. 639–662, 2007.

[6] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 135–145, 2004.

[7] P. Diaz and M. Cintra, "Stream chaining: exploiting multiple levels of correlation in data prefetching," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pp. 81–92, 2009.

[8] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 609–623, 1995.

[9] S. Kim and A. V. Veidenbaum, "Stride-directed prefetching for secondary caches," in *ICPP '97: Proceedings of the international Conference on Parallel Processing*, p. 314, 1997.

[10] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 63–74, 2007.

[11] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," *SIGARCH Comput. Archit. News*, vol. 22, no. 2, pp. 24–33, 1994.

[12] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005. http://sesc.sourceforge.net.

[13] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pp. 45–57, 2002.

[14] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, 2000.

[15] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 222–233, 1996.

[16] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 12, no. 2, pp. 87–106, 1991.

[17] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 252–263, 2006.

[18] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pp. 69–80, 2009.

[19] M. Ferdman, S. Somogyi, and B. Falsafi, "Spatial memory streaming with rotated patterns," *The first JILP Data Prefetching Championship (DPC-1)*, 2009.

[20] "Data prefetching championship." http://www.jilp.org/dpc/.

[21] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 397–408, 2006.

[22] I. Hur and C. Lin, "Feedback mechanisms for improving probabilistic memory prefetching," in *HPCA*, pp. 443–454, 2009.

[23] R. Manikantan and R. Govindarajan, "Focused prefetching: performance oriented prefetching based on commit stalls," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pp. 339–348, 2008.

[24] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching prefetch: Optimization friendly method," *The first JILP Data Prefetching Championship (DPC-1)*, 2009.