Access Map Pattern Matching Prefetch: Optimization Friendly Method

Yasuo Ishii†, Mary Inaba‡, and Kei Hiraki‡

†NEC Corporation
‡The University of Tokyo

Abstract

Recently, techniques for the optimization of microarchitecture and compilers have progressed significantly. However, sometimes these optimizations cause failure in prefetch detection. In general, to generate prefetch requests, prefetch algorithms use (a) data addresses, (b) memory access orderings, and (c) instruction addresses. However, (b) and (c) are often scrambled by optimizations.

In this paper, we propose a new optimization-friendly memory-side prefetch algorithm: Access Map Pattern Matching prefetch. In this algorithm, coarse-grained ordering information – recent-zone-access frequency (denoted by (d)) – is used instead of (b) and (c). The AMPM prefetcher holds the memory access pattern of recent memory access requests and generates prefetch requests by pattern matching.

We evaluate the AMPM prefetcher in a DPC framework. The simulation results show that our prefetcher improves performance by 53%.

1. Introduction

Optimization techniques such as out-of-order execution, speculative execution, and relaxed consistency of microarchitecture as well as those such as loop unrolling of compilers have improved system performance. However, sometimes, these optimizations cause failure in prefetch detection.

In general, conventional prefetchers [3] use (a) data addresses, (b) memory access orderings, and (c) instruction addresses to generate prefetch requests. However, memory access orderings are often scrambled by out-of-order execution with relaxed memory consistency, and memory access instructions such as load or store are duplicated by loop unrolling. Such scrambles degrade performance of conventional prefetch methods such as [1, 2]. These conventional prefetchers cannot determine the correct address correlation when the memory access ordering is scrambled or memory access instructions are duplicated since they use exact match to the previous memory access sequence for finding an address correlation.

In this paper, we propose a new optimization-friendly prefetch method: Access Map Pattern Matching (AMPM) prefetch. This prefetch method is tolerant to optimizations since it uses only memory access footprint which implies the memory location is recently accessed or not, instead of the fine-grained memory access sequence information. Further, since we do not use resources for (b) and (c), we can allocate more resources for storing (a) data addresses. Data addresses can be stored in the bitmap format instead of the conventional linked-list format. This enables more accurate prefetch detection and parallel prefetch detection.

The AMPM prefetch method involves the following steps: (1) detecting hot zones, (2) storing the 2-bit states for all cache lines in the memory access pattern map of these hot zones, (3) listing prefetch candidates by pattern matching of the memory access pattern map, and (4) selecting prefetch requests from among these candidates and issuing the requests to the main memory. This pattern matching does not suffer because of optimizations since it uses neither memory access ordering nor instruction addresses.

In section2, we present a design overview of the AMPM prefetcher, and we describe the data structure and the algorithm used in the AMPM prefetcher. In section 3, the hardware implementation for the AMPM prefetcher and the complexity of the AMPM prefetcher are discussed. In section 4, the other optimizations that are used for the DPC competition are presented. In section 5, the detailed budget count and evaluation results are presented. Finally, in section 6, we conclude the paper.

2. Design of the AMPM Prefetcher

2.1. Overview of the AMPM prefetcher

The AMPM prefetcher divides main memory space into fixed sized areas. Each area is called zone and it is treated a unit. First, the prefetcher detects “hot zones” on the basis of the recent-zone-access frequency, whose basic concept is equal to the *concentrated zone (Czone)* [6]. The AMPM prefetcher stores the 2-bit access states of all cache lines of hot zones in the memory access pattern maps. These access states in the map are stored without any access order. The number of hot zones and memory access pattern maps is fixed, and they are stored

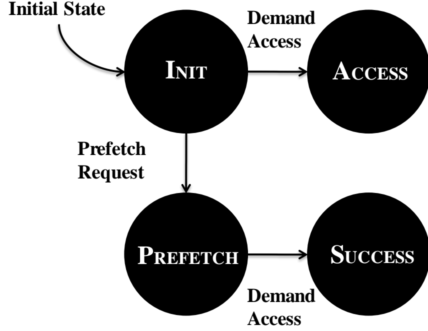

Figure 1 State Diagram for Memory Access Map

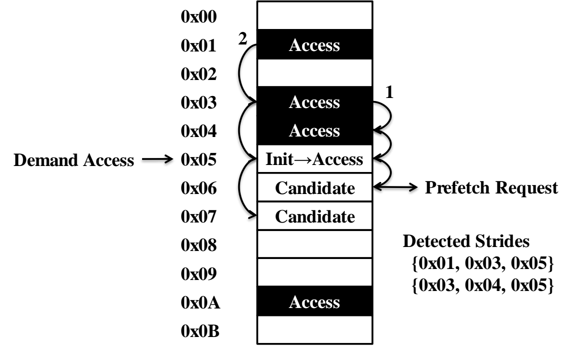

Figure 2 Access Map Pattern Matching Prefetch

in a memory access map table and replaced by the least recently used (LRU) policy. This LRU replacement for hot zones realizes coarse-grained ordering management based on the recent memory access frequency since the memory access pattern of hot zone is discarded when the zone is replaced. In this study, we adjust the total size of the memory access pattern maps so that it is almost equal to the L2 cache capacity.

When memory access requests arrive, the AMPM prefetcher tries to detect the stride address correlation by pattern matching using the memory access pattern map and determines the prefetch candidates. It also decides which prefetch requests are appropriate and issues these requests to the main memory. The AMPM prefetcher also decides the number of requests to issue on the basis of the profiled information.

The data structure in the memory access pattern map is described in subsection 2.2.; the algorithm for detecting the prefetch candidates and selecting the requests is described in subsection 2.3.; and adaptive prefetching is described in subsection 2.4.

2.2. Memory Access Map

The AMPM prefetcher uses memory access pattern map to generate prefetch requests. To collect memory access pattern map, the AMPM prefetcher employs a memory access map table which holds memory access maps of hot zones. The memory access map contains the information on the memory access pattern map of its corresponding zone. The memory access pattern map contains the information on previous accesses for all addresses in the corresponding zone as a bitmap data structure. The entry of memory access pattern map is cache line granularity. For example, when the cache line size is 64B, the memory access pattern map granularity becomes 64B. The status of each cache line in the zone is stored in a two-bit state machine. This two-bit state machine has four states (**Init**, **Prefetch**, **Access**, and **Success**). The diagram of the state machine is shown in Figure 1. The transitions between the states occur only when the prefetcher receives demand requests or the prefetch requests are issued to the main memory.

Since the transitions can only occur in one direction, the number of **Access** and **Success** states monotonically increases until the memory access maps are replaced. When almost all states in the zone become **Access** or **Success**, the prefetcher does not issue prefetch request. In this case, almost all the frequently accessed data is stored in the L2 cache memory since the total size of the memory access maps is almost equal to the L2 cache capacity. Since the hot zones are already fetched in the cache memory, there are no needs for additional prefetch requests.

2.3. Generating Prefetch Requests

The AMPM prefetcher generates the prefetch requests when the L2 cache memory receives a demand request. The prefetcher reads three consecutive memory access maps from the table and concatenates them. The prefetch generator selects the prefetch candidates by determining the address correlation on the basis of the pattern matching in the concatenated map. The basic concept of the pattern matching is based on stride detection. In pattern matching, a pattern matching detector checks whether the statuses of request addresses at –N and –2N are **Access** or **Success**. When a pattern is detected, the request address at +N becomes a prefetch candidate. This pattern matching detector generates many prefetch candidates within the memory access map in parallel. Finally, the prefetch generator selects the nearest candidates from the address of the demand request. The selected requests are issued to the main memory.

For example, when the addresses 0x01, 0x03, and 0x04 have already been accessed and a demand request for 0x05 reaches the L2 cache memory, the prefetch generator selects two candidates; (1) 0x07, whose address correlation is {0x01, 0x03, 0x05}, and (2) 0x06, whose address correlation is {0x03, 0x04, 0x05}. The candidate (2) 0x06 is selected first since 0x06 is nearer to 0x05 than 0x07 (Figure 2).

2.4. *Adaptive Prefetch Degree*

The prefetch generator optimizes *prefetch degree* which is the number of issuing prefetch request for achieving good performance. The generator controls the prefetch degree using (1) the frequency of the prefetch requests, (2) the frequency of conflict misses in the L2 cache memory, (3) the conflict misses in the memory access map, and (4) the ratio of prefetches success.

To hide the memory access latency effectively, the prefetcher has to fetch the target data to cache memory before the fetched data is accessed. It is decided by the memory latency and the access frequency of zone. The AMPM prefetcher employs the access frequency counter and decides its prefetch degree by the quotient of memory access latency and the memory access frequency. For example, when the memory access latency is 200 cycles and the demand memory access reach the L2 cache miss every 50 cycles, the AMPM prefetcher issues 4 prefetch requests to the main memory.

The prefetch requests often degrade the processor performance, since the prefetched data pollute the L2 caches. When above (2) or (3) happens, our prefetcher restricts the maximum number of the prefetch requests. (2) is detected when a memory access map entry of a cache missed address is **Access** or **Success** since such line is not only recently accessed but also evicted from the cache. Since the capacitance miss will replace the memory access map too, such line is caused by the conflict miss. (3) is detected from the replacement frequency of the memory access map.

When the number of occurrence of **Success** in the map is larger than that of **Prefetch**, the prefetcher uses the **Prefetch** states as **Access** or **Success**. In this case, the prefetch generator can detect more candidates.

3. Hardware Design & Complexity

An overview of the implementation of the AMPM prefetcher is shown in Figure 3. The memory access map table stores the memory access maps in a contents addressable memory (CAM). The prefetch generator is composed of two access map shifters, candidate detectors, priority encoders, and address offset adders.

The requests are generated in the following sequence of steps. First, the memory access map table is read by the address of the demand request. The shifters are used for memory access map alignment. The position of the demand request is aligned with the edge of the access map. Then, the detector determines candidate addresses by pattern matching. The priority encoders select prefetch requests to be issued. Finally, the addresses corresponding to the issued prefetch requests are calculated in the address offset adders and the requests are issued to the main memory.

3.1. *Memory Access Map Table*

A memory access map table is implemented as a multi ported CAM that holds approximately 64 maps. The complexity of the CAM is almost the same as that of the full-associative TLB. The TLB is feasible even though it is located in the fast clock domain (processor core domain). On the other hand, the memory access map table is located in a slower clock domain (e.g., the L2 cache clock domain). This implies that the memory access map table has enough feasibility since it has to operate in a slower clock domain. In order to increase the operation frequency, the memory access map table can be implemented as a multi banked set-associative structure, similar to the implementation of the multi banked cache memory.

3.2. *Prefetch Generator*

A prefetch generator is composed of memory access map shifters, candidate detectors, priority encoders, and address offset adders. Each component processes approximately 256 bits of data.

The memory access map shifters and the priority encoders are not so small, but it is feasible to implement them using existing hardware since the fast 128+ bit shifter and priority encoder have already been used in commercial processors [5]. The candidate detectors are implemented by using many simple combination circuits which checks -2N and –N to determine issuing an N_{th} address. Each component needs a few gates. The address adders are simple 32 – 64 bit adders.

3.3. *Pipelining for AMPM Prefetcher*

In order to use the AMPM prefetch technique at a higher operation frequency, the prefetch generator can be pipelined. Pipeline registers are inserted between the candidate detectors and the priority encoders. In the pipelined AMPM prefetcher, the priority encoders are used repeatedly until subsequent prefetch requests reach the pipeline registers. The generator makes only two requests in one cycle. When the next prefetch request reaches the pipeline stage, the previous prefetch candidates are discarded.

4. Other Optimizations

4.1. *Processor-Side L1 Prefetching*

Our prefetcher employs an adaptive stream prefetcher [4] as a processor-side prefetcher since an adaptive stream prefetcher has a good cost performance. Although this prefetch method was proposed as a memory-side prefetcher, we found that the method works well and improves performance for L1 prefetching.

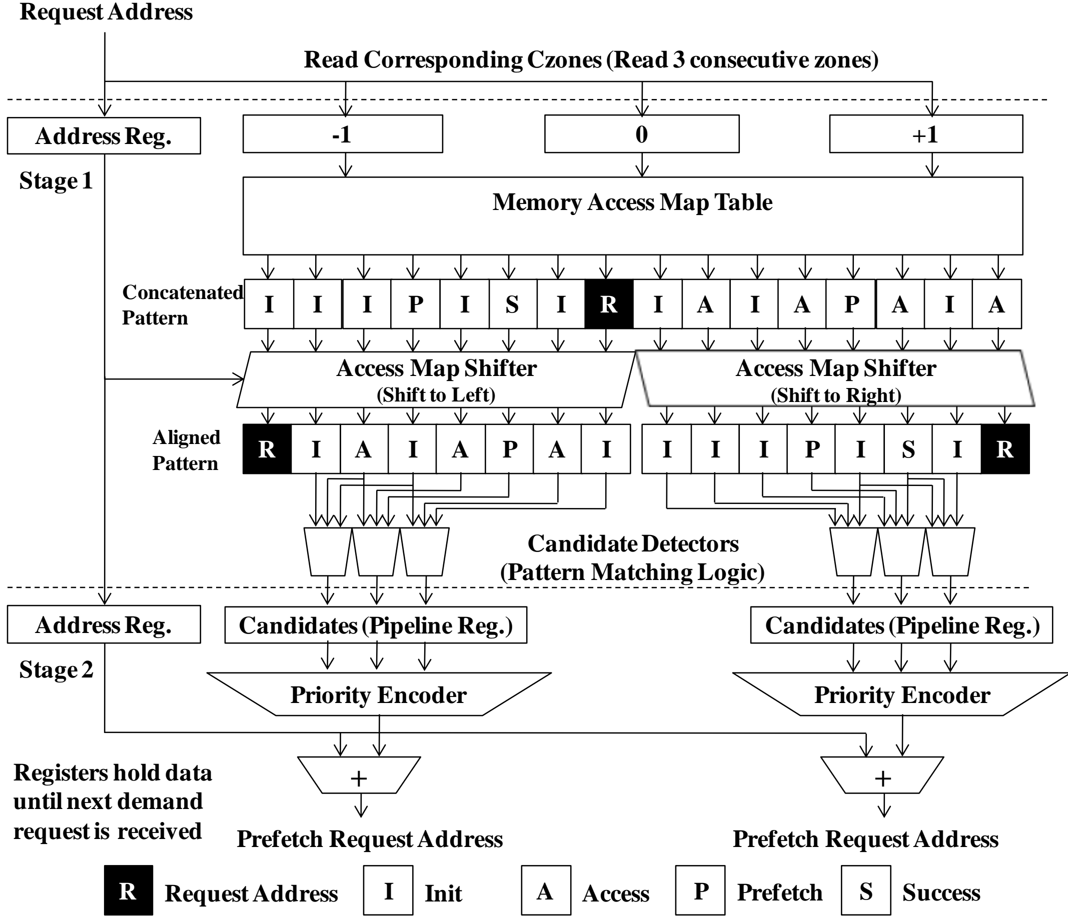

Figure 3 Implementation of AMPM Prefetcher

4.2. Miss Status Handling Register

The DPC framework provides a miss status handling register (MSHR) with 16 entries; however, this MSHR size is not sufficient for supporting the required number of in-flight prefetch requests. We employ another MSHR with 32 entries for handling the prefetch requests and use the default MSHR with 16 entries for handling the demand requests.

5. EVALUATION

5.1. Configuration

Table 1 shows the storage counts for our prefetcher. The prefetcher employs a pipelined AMPM prefetcher and an adaptive stream prefetcher. The AMPM prefetcher is composed of several pipeline registers and a full-associative memory access map with 52 entries. Each memory access map holds 256 addresses manages 16KB zone (256 x 64B). The adaptive stream prefetcher employs 16 stream filters for stream detection and 16 histograms for the prediction of the stream length. Since our prefetcher does not use line offset for the addresses, the addresses are counted as 26 bits (32 bits – 6 bits).

5.2. Results

We evaluated the proposed prefetcher in the DPC framework. We used the SPEC CPU2006 benchmark suite. The compile option was "-O3 -fomit-frame-pointer -funroll-all-loops." The simulator skips the first 4000M instructions and evaluates the next 100M instructions. We used ref inputs for the evaluation.

The evaluation results are shown in Figure 4. The AMPM prefetcher improved performance by 53% and reduced L2 cache miss counts by 74%.

6. CONCLUSIONS

In this paper, we proposed an AMPM prefetcher that is suitable for modern optimized processors. The prefetcher uses memory access maps to profile previous memory

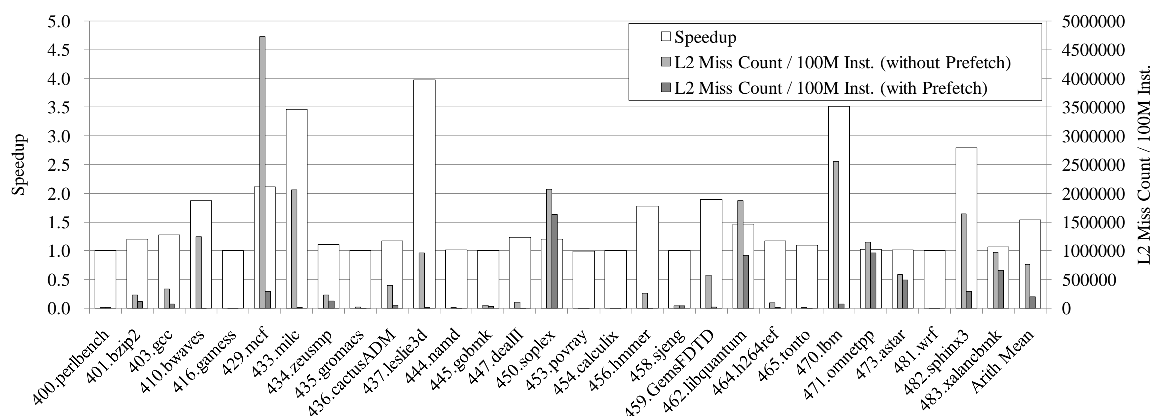

Figure 4 Evaluation Results for SPEC CPU2006

Components			Budget
MSHR	Valid bit (1bit) Address bit (26 bit)	16 entries	0bit (Default)
Prefetch MSHR	Valid bit (1bit) Address bit (26 bit) Issue bit (1 bit)	32 entries 5bit pointer	901 bit
Memory Access Map Table	Address Tag (18 bit) LRU status(6 bit) Access Counter (4 bit) Interval Timer (18 bit) Access Map (256 x 2 bit)	52 entries + mode register (3 bit) + performance counter (32 bit x 4)	29147 bit
Adaptive Stream Filter	Valid bit (1bit) Address bit (26 bit) Lifetime (10 bit) Stream Length (4 bit) Direction (1 bit)	16 entries	672 bit
Stream Length Histogram	Counter (16 bit)	16 entries 2 series 2 direction	1024 bit
Pipeline Registers			292 bit
Total			32036 bit

Table 1 Budget Counts for AMPM prefetcher

accesses. Memory access maps are stored in a memory access map table and replaced by LRU policy. Each memory access map holds only memory access pattern map where the previous memory requests are accessed. The memory access map stores many previous requests since it does not need to store the ordering information or instruction locations. Since the memory access patterns are not affected by the memory access ordering, the AMPM prefetcher is tolerant to the out-of-order memory accesses.

We evaluated an optimized AMPM prefetcher with a 32K-bit budget in the DPC framework. The evaluation results show a 53% improvement in the performance of the SPEC CPU2006.

REFERENCES

[1] K. J. Nesbit, A. S. Dhodapkar and J. E. Smith, AC/DC: An adaptive data cache prefetcher. In *Proc. of Int. Conf. On Parallel Architecture and Compilation Techniques,* 2004.

[2] K. J. Nesbit and J. E. Smith, Data Cache Prefetching Using a Global History Buffer. In *Proc. of Int. Symp. On High Performance Computer Architecture,* 2004.

[3] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys, 32(2):174-199,* June 2000.

[4] I. Hur and C. Lin. Memory Prefetching Using Adaptive Stream Detection. In *Proc. of Int. Symp. On Microarchitecture,* 2006

[5] S. D. Trong, M. Schmookeler, E. M. Schwarz, and M. Kroener. POWER6 Binary Floating-Point Unit, *Proc. of Int. Symp. On Computer Arithmetic*, 2007,

[6] S. Palacharla and R. Kessler, Evaluating stream buffers as a secondary cache replacement, In *Proc. of. Int. Symp. On Computer Architecture, 2004.*