An Optimized AMPM-based Prefetcher Coupled with Configurable Cache Line Sizing

Qi Jia, Maulik Bakulbhai Padia, Kashyap Amboju and Huiyang Zhou

Department of Electrical and Computer Engineering

North Carolina State University

qjia2@ncsu.edu, mpadia@ncsu.edu, kamboju@ncsu.edu, hzhou@ncsu.edu

Abstract

Access Map Pattern Matching Prefetch (AMPM) is a state-of-art data prefetching technique. In this paper we present an optimized AMPM-based prefetcher coupled with configurable cache line/block sizing. Our optimizations include (a) prefetching from infrequently-accessed memory zones and (b) fixing inaccurate access states of the original AMPM prefetcher. Also we introduce a configurable cache line/block sizing scheme to exploit strong spatial locality if it is detected. We evaluate our prefetcher in the DPC2 framework. For a set of 27 SPEC CPU 2006 benchmarks, our experimental results show that our scheme achieves a 10.5% system-level performance improvement compared to it without prefetching.

1. Introduction

Memory prefetching is widely used in modern processors since it can fetch needed data in advance and hide the memory access latency effectively. Current prefetchers track data streams, which are identified based on program counters (PCs), spatial patterns or combinations of both, to predict and prefetch possibly-used data in the future.

Access Map Pattern Matching Prefetch (AMPM) proposed by Ishii [3] is a state-of-art prefetcher. Different from stride-based prefetchers and GHR-based [4] prefetchers, AMPM is not based on PCs. AMPM attaches concentration zones with cache-line bitmaps to identify the spatial pattern of each equal-sized zone. However, AMPM cannot make effective predictions when the zone just starts being accessed or when the zone is infrequently accessed, i.e., cold zones. Also, the cache-line bitmap cannot reflect accurate access states when parts of the blocks within the zone are evicted from the cache.

We propose two optimizations, (1) cold zone optimization and (2) clear zone optimization, to overcome the limitations of the AMPM prefetcher. Furthermore, we design an LLC prefetching algorithm based on configurable block sizing (CBS).

The cold zone optimization tracks the common offsets within zones and prefetches these cache lines when a new zone is encountered. The clear zone optimization tracks whether the bitmap information within zones is accurate. Once the bitmap of a zone is detected as inaccurate, the zone will be evicted from AMPM such that the new/updated spatial pattern of the zone can be reconstructed.

The CBS-based prefetcher samples the cache with different block sizes. At the end of each epoch, the block size which leads to highest performance, considering both hit rate and memory bandwidth, will be selected as the prefetching block size (PBS) for the next epoch.

In Section 2, we motivate our schemes. In Section 3 we present the design of our prefetcher and prefetching algorithm. In Section 4, the complexity and overhead are discussed. In Section 5 the performance results are presented. In Section 6, we discuss further optimizations and conclude the paper.

2. Motivation

2.1. Cold Zone Optimization

Figure 1 shows an example trace from the benchmark *milc*. The zone size is 4kB and we also show the zone offsets along the trace in Figure 1. From Figure 1, we can see that each zone is accessed relatively infrequently (around 8 times). As a result, AMPM cannot detect the spatial pattern before the zone is evicted from AMPM.

On the other hand, there are common offsets across different zones. From Figure 1, we can see that among the accesses to different zones, the offsets 0x880 and 0xa80 are common ones. If we could detect these common offsets, we

Access trace: 1bc9488d7c64880, 1bc9488d7c64a40, 1bc9488d7c648c0, … … , 1bc9488d7c64a80,
Offsets within a 4kB zone: *880, a40, 8c0, … a80*
Access trace: 1bc94d0de078080, 1bc9488d7c64900, 1bc94d0de078880, 1bc94d0de078bc0, 1bc94d0de078a80
Offsets within a 4kB zone: *080, 900, 880, bc0, a80*

Figure 1. L2 access trace and offset of benchmark *milc*.

can use them to prefetch data for those zones whose access patterns have not been trained, i.e., when zones are cold.

2.2. Clear Zone Optimization

In AMPM, the bitmaps of each zone tracks the access and prefetch status of each cache line in the zone. It is not updated when a cache blocks is evicted. Thus we may get wrong information from the bitmap and lose chances to prefetch. For example, a part of the access bitmap in a zone may appear as *1111* while the actual state is *1110* due to eviction, either replaced by demanded data or replaced due to the inclusion property. In this case, we lose the chance to prefetch the block represented by the fourth bit since the block is regarded as accessed in AMPM.

To deal with the inaccurate status within a zone, we can clear the bitmaps of the zone to get a chance to reconstruct the spatial pattern.

2.3. CBS-Directed Prefetch

Applications with strong spatial locality prefer large cache line/block sizes. With larger block sizes, the cache hit rate will also increase. However, if the block size becomes too large, it would not only consume memory bandwidth and energy but also pollute the cache if the spatial locality is strong enough. In our scheme, we resort to samplers and design an algorithm to choose the best block size, which is then used to direct next/previous n-line prefetching.

3. Design Overview

3.1. Common Offset Table

For the cold zone optimization, we need to detect the common offsets across accesses to different zones. To do so, we use a common offset table to record these offsets within zones. Each entry is used for one distinct offset. There are 4 fields in each entry as shown in Figure 2. The 6-bit offset field records a zone offset at the cache line granularity. The 5-bit field "counter" indicates how possible this offset will be accessed in the future. The 1-bit "pref" field indicates whether a prefetching request has been issued using this offset. The LRU field is used for replacement from the common offset table. An offset is detected as a common one if its "counter" field is larger than a threshold. Then, when a new zone is encountered, prefetches requests are issued based on the common offsets detected from the table.

The prefetching degree based on common offsets, i.e., the number of common offsets, used for cold zones should be restricted by two aspects: (1) prefetch accuracy and (2) the overlap with original AMPM.

Prefetch accuracy is important to adjust the prefetch degree. The one-bit "prefetch" field is used to get the prefetch accuracy information. This bit is set when a prefetch request is issued with this offset. Then if a demand access results in a prefetch hit at this offset, the corresponding "counter" will be increased by 2 and the corresponding "pref" bit is reset. At the end of each epoch, all the entries in the table will be checked. The counter value will be right-shifted by 1 bit and the counter values of the entries whose prefetch bit is set will be further decreased by 2 since those offsets are not accessed after they are prefetched.

The prefetching address overlap with the original AMPM is also important. If the most of the prefetch detected by cold zone optimization can also be detected by the original AMPM, the prefetch degree used for cold zone optimization should be decreased. To detect the overlap, we check whether the prefetch hit comes from both AMPM and cold zone optimization.

Also, to prevent same prefetch requests from being issued multiple times we add a new state in AMPM called "cold zone access". A cache line will be set to this state if it is prefetched by cold zone optimization. The cache lines in this state will not be prefetched again.

Figure 2. Common Offset Table entry.

3.2. Detect Inaccurate BitMap States

To detect whether the current state of each zone is accurate or not, we construct a set of conflict counters. Every zone in AMPM is attached with a conflict counter to track how inaccurate the current information in the zone is. If one page is evicted from AMPM, its corresponding conflict counter will be reset. For an access, which misses in the cache but is marked as "access" or "prefetch" in AMPM, the counter associated with the zone will be increased. The larger the counter is, the more inaccurate the information in the zone is. In each epoch, the conflict counters will be checked and the corresponding page will be reset in AMPM (all bitmaps will be cleared) if the entry value is above a threshold that we set in advance.

Besides our approach, there are two alternative ways to fix inaccurate states in zone. The first one is to reset the corresponding bit in the zone once we detect inaccurate block status. However, this approach only reset the inaccurate information we have observed but have no capability to reset the inaccurate information we have not seen in advance. The second approach is to reset the corresponding bit on every block eviction. This approach can eliminate the inaccurate information once it is generated. But the disadvantage of this approach is that it cannot track the blocks which are evicted silently (e.g. the blocks eviction due to inclusion policy). Thus this approach cannot guarantee the information in the AMPM is accurate either. Considering the disadvantages of these two alternatives, we did not use them in our implementation.

3.3. Configurable Block Size Monitor

We use a block size monitor to select the best block size used for prefetching. As previous research [5] has shown, sampling provides an accurate way to get the complete cache information. The block sizing monitor is composed of a set of auxiliary tag directories (ATD) as shown in Figure 3.

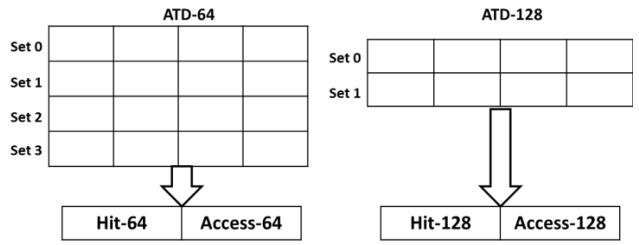

Figure 3. Organization of auxiliary tag directories (ATDs).

As shown in Figure 3, ATD-64 is used for the 64B cache line size and ATD-128/ATD-256/ATD-512 maintains the shadow tags at the 128B/256B/512B cache line size granularity. With set sampling, the ATDs monitor the address stream to the sampled sets in the L2 cache. We ensure that all ATDs monitor the same address stream, which constrains the beginning index of the sampled sets to be a multiple of 8 in this case.

The cache line size configuration policy is as follows: all four hit counters and access counters are collected for the cache line sizes of 64B, 128B, 256B and 512B. Then to take both bandwidth and hit rate into consideration, we calculate a score as following:

$$Score = hit - A * (access - hit) * block\_size \quad (1)$$

We set the parameter A as 1/64 based on our experimental results.

Every time the locality monitor selects the cache line size with the highest score. We record the cache line size then shift all hit and miss counters to the right by 1 bit so that we can keep part of the history information but not rely too much on the history for prediction.

Prefetch requests issued by the CBS-directed prefetcher will be sent to LLC to save L2 MSHRs and avoid L2 cache pollution.

3.4. Two-Level Prefetching

In DPC2 framework the blocks can be prefetched into L2 or LLC. Thus, we add a new status into AMPM called "LLC_prefetch". If the block is prefetched into L2 then it will be placed into state: "prefetch". If it is prefetched into LLC then its state will be set as "LLC_prefetch". During AMPM prefetching, we will assign higher priority to the candidate which is never prefetched and lower priority to the candidate which is not prefetched in L2 but has been prefetched into LLC. The pseudo code of this scheme is shown in Figure 4.

3.5. Prefetch Filter

The original AMPM will issue prefetch requests no matter a demand access misses or hits in the cache. This will incur a large number of prefetch requests which will pollute the L2 cache. So, in our design we only prefetch on cache miss or prefetch hit. To achieve this we add one bit "prefetch" to each block in the cache to indicate if this block is brought into cache by demand or prefetch. The prefetch bit will be set when a block is inserted into cache by prefetching and reset when the block is accessed later.

```
//first AMPM prefetch loop
if(find_candidate) {
  if(candidate not prefetched) {
    if(enough mshr entries)
      prefetch candidate to L2
    else
      prefetch candidate to LLC
  } }
//second AMPM prefetch loop
if(prefetch block < prefetch degree) {
  if(find_candidate) {
    if(candidate prefetched into LLC) {
      if(enough mshr entries)
        prefetch candidate to L2
} }  }
```

Figure 4. Two-level prefetch pseudo code

4. Complexity and Overhead

In our prefetcher design, we include the following structures: a common offset table, a conflict table and block sizing monitors.

For the block sizing monitors we sample the cache sets with a ratio of 1/16. Therefore, with 64-bit addresses, for a 8-way set associative 256KB L2 cache with the 64B line size, the storage cost of an ATD-64 is: 1/16 * 256KB/64B * (64-6-9 bits) = 1.532KB; the cost of an ATD-128 is: 1/16 * 256KB/128B * (64-7-8 bits) = 0.766KB; the cost of an ATD-256 is: 1/16 * 256KB/256B * (64-8-7) = 0.383KB; and the cost of an ATD-512 is 1/16 * 256KB/512B * (64-9-6) =0.191KB. Therefore, the overall storage cost of the per-core ATDs is 2.872KB

The total storage requirement for our prefetcher is summarized in Table 1.

Components			Storage
Memory Access Map Table	Address Tag (64 b) LRU (6 b) Access Map (3*64 b)	64 entries	2.047KB
CBS monitor	ATD	4 ATD	2.872KB
Common Offset Table	Counter (6 b) LRU status (6 bits) Offset Map(64*6 bits +64*1bit)	8 entries	0.45KB
Conflict Table	Counter (6 bits)	64 entries	0.046KB
Prefetch Bit	Prefetch (1 bit)	4096 blks	0.5KB
Cold Zone MSHR	Tags (64 bits) LRU status (5 bits)	32 entries	0.27KB
Total			6.185KB

Table 1. Storage requirement for prefetcher.

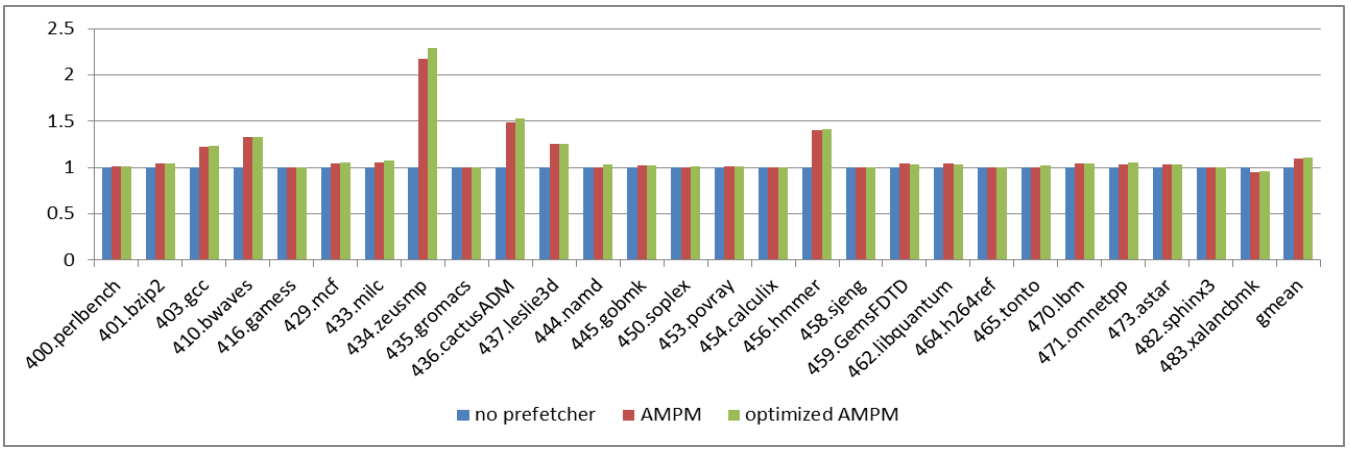

Figure 5. Speedups compared with no prefetcher and the original AMPM

5. Evaluation

We evaluate our prefetcher using the DPC2 framework using the provided configurations. We use the SPEC CPU 2006 benchmarks [2] in our experiments. For each workload, we skip the first 1 billion instructions, warm up the cache using the next 10 million instructions, and run for the next 100 million instructions to get the performance results.

Figure 5 shows the achieved speedups compared with the baseline without prefetching and the original AMPM prefetching scheme. From Figure 5 we could see our optimized prefetcher outperforms the baseline without prefetching by 10.8%. Compared with the original AMPM, our scheme achieves a speedup of 0.76% on average. Among all the benchmarks, our scheme can get a maximum speedup of 5.3% for the benchmark *zeusmp*.

The final performance is not improved too much compared with the original AMPM. The first reason is AMPM has done a good job within the zone and our optimization can work only for the benchmarks with some specific trace properties. The second reason is that since the current DPC2 framework will delay the L2 access until the block is returned if there is a hit in MSHR. It means that if we prefetch accurately but a little late, then the next chance to prefetch will be delayed. This scenario can be seen for benchmark *milc*, for which we reduce the L2 miss rate significantly but the performance improvement is not obvious.

6. Future Work and Conclusions

The other possible optimization which could be integrated with AMPM is the PC-directed prefetching within each zone. We can construct tables to track local trace delta within a zone and then detect any delta pattern using the similar way mentioned in previous research [1]. This optimization can direct AMPM to prefetch in time and more accurately since it can provide more PC-related information instead of spatial information only.

In this paper we optimize a state-of-art prefetcher, AMPM, and combine it with configurable block sizing. The resulting prefetcher has the capability to prefetch from cold zone and fix inaccurate states within AMPM table. Also the configurable block sizing scheme provides additional benefits when strong spatial locality is detected.

Reference

[1] Dimitrov, Martin, and Huiyang Zhou. "Combining local and global history for high performance data prefetching." *Journal of Instruction-Level Parallelism* 13 (2011): 1-14.

[2] Henning, John L. "SPEC CPU2006 benchmark descriptions." *ACM SIGARCH Computer Architecture News* 34.4 (2006): 1-17.

[3] Ishii, Yasuo, Mary Inaba, and Kei Hiraki. "Access map pattern matching for high performance data cache prefetch." *Journal of Instruction-Level Parallelism*13 (2011): 1-24.

[4] Nesbit, Kyle J., and James E. Smith. "Data cache prefetching using a global history buffer." *Micro, IEEE* 25.1 (2005): 90-97.

[5] Qureshi, Moinuddin K., Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. "A case for MLP-aware cache replacement." *ACM SIGARCH Computer Architecture News* 34, no. 2 (2006): 167-178