Analysis of Data Prefetching Algorithms

Palak Agarwal
13453
Computer Science and Engineering
Indian Institute of Technology, Kanpur
Email: palakag@iitk.ac.in

Dheeraj Mekala
13405
Computer Science and Engineering
Indian Institute of Technology, Kanpur
Email: dheerajm@iitk.ac.in

Devashish Kumar Yadav
13240
Computer Science and Engineering
Indian Institute of Technology, Kanpur
Email: devyadav@iitk.ac.in

***Abstract*—As the number of prefetching algorithms that exist are very large and different from one another, we in this project compare some of them in the provided framework by DPC2. We analyze why some of these algorithms work in some cases and why these fail in others. More specifically, we compare the algorithms [4] [5] [7] that were submitted to the challenge with the more general prefetchers like OBL [6]. We also implement the modified version of AMP algorithm [1] and analyze it against these algorithms as well.**

I. Introduction

Modern microprocessors try to hide cache misses by prefetching data/instruction early. Prefetchers predict next several data/instruction load addresses and fetch them in the cache if they are not already present there. Prefetchers are useful but for optimal execution, they have to be careful of not polluting the cache and using bandwidth efficiently. The time at which a page is prefetched is also crucial. If a prefetch is issued too early, then it might get possibly evicted from cache before it is used and if a prefetch is issued too late, it might not hide memory latency properly.

OBL(Next-line prefetcher) [6] prefetches cache line of address $X + 1$ into L2 cache whenever cache line of address X is requested by the L1 cache. OBL gives significant performance benefits for applications with good spatial locality. Blind prefetchers like OBL(Next-line prefetcher) [6] suffer from two major problems:

1) Since next-line prefetcher prefetches next line every time, they overuse cache and bandwidth resources which might lead to reduction in benefit and more consumption of power
2) They may not be well-timed and hence may not provide expected performance

1) Best-offset prefetcher: Offset prefetcher [5] is a generalization of next-line prefetcher. Best-offset prefetcher prefetches cache line $X + O$ for every cache line X requested by the L1 cache which is a L2 miss or prefetched hit, where O is a non-zero offset value. The algorithm updates this offset by regular learning from the data.

2) Prefetching On-time: In SPAD [4], they design a helper mechanism that adaptively learns the degree of prefetch. At the end of each learning stage, the number of hits and prefetches are compared according to which the degree of prefetch is manipulated.

3) Slim AMPM: AMPM prefetching [3] concentrates hardware resources on collecting the access-footprint of frequently accessed area and prefetch requestss are generated from the pattern matching of the memory access map. Slim AMPM [7] is an extended version of AMPM [3] which optimizes bandwidth.

II. Comparing Prefetchers

For evaluation purposes, we have used the DPC2 simulator provided at the challenge website [2]. We have used 3 algorithms, which were the winner of DPC2-2015 [4], [5], [7] and the simple next-line prefetcher for comparing the scores. The analysis was done in 2 parts: Between applications and Between prefetchers. We ran these algorithms on traces of SPEC2006 benchmarks and the results are tabulated in Table I. For generating the trace, each of these benchmarks used 10,000,000 warmup instructions and then was simulated for the next 100,000,000 instructions. The scores reported here are the cumulative CPI when the size of L3 cache is 1 MB and memory frequency is 12.8 GB/second. Other details regarding the simulation environment can be found on the website.

A. Between applications

As can be inferred from the table, the performance improvement achieved by the applications can be different even if they are using the same prefetching algorithm. For instance, consider the benchmarks gcc and libquantum. On the same algorithm, let's say for next-line prefetcher, the IPC of the two benchmarks is very different. To analyze this further, we did a data footprint of the benchmark to find out what memory was being accessed and in what order. GCC's footprint shows that a lot of memory accesses are scrambled, i.e, though they are located in the same spatial area the access order is not sequential, so a next line prefetcher in this case might lose out on performance. Apart from this, there are multiple other memory areas from where gcc accesses data. This feature is very different from libquantum, another benchmark, whose

TABLE I. Benchmark Results (IPC)

Benchmarks	Best Offset	OBL (Next-line)	Prefetching On-Time	Slim AMPM
GCC	0.34389	0.284244	0.311999	0.3455086
leslie3d	1.631569	1.448414	1.708512	1.696512
milc	1.270079	1.212129	1.243693	1.31145
GemsFDTD	3.452212	3.425332	3.453519	3.450598
libquantum	3.254704	3.293444	3.290714	3.276598
omnetpp	2.155531	1.85322	2.127376	2.262831
lbm	1.998883	1.600095	2.017431	2.035816
mcf	0.371862	0.35061	0.37161	0.395652

TABLE II. BENCHMARK RESULTS ON AMP ALGORITHM

Benchmarks	IPC
GCC	0.28576
leslie3d	1.514827
milc	1.165884
GemsFDTD	3.453104
libquantum	3.175271
omnetpp	2.185091
lbm	2.064189
mcf	0.345872

memory accesses are highly ordered. The footprint showed that it accesses 3 memory areas in the following way:

$$A, B, X, A, B, X+4, A, B, X+8, \ldots \quad (1)$$

where A, B and X are addresses.

B. Between prefetchers for the same application

Next, we analyze the difference in performance when we use different prefetching algorithms for the same benchmark. We observe that in the two benchmarks, omnetpp and lbm, the difference in the performance when next-line prefetcher is used and the other 3 prefetchers are used is quite stark. In this section, we try to give a reasoning as to why this might be the case.

1) Omnetpp: In the case of omnetpp, the memory accesses are quite spread out and so using a next-line prefetcher will not be useful, however a slightly larger prefetch offset might be helpful which is why the Best Offset prefetcher gives quite good results in this case. The offset learned by the Best offset algorithm will be quite high (we found it to be as big as 13 for some cycles) which makes sure that the accessed data is actually prefetched in the cache. The prefetching on-time algorithm also works well in this case because it takes care of the strides and also learns offset with time. The AMPM algorithm uses the concept of zones which becomes useful in this case as most of the accesses take place in a limited number of memory areas only.

2) lbm: The data access in lbm is highly spatial as can be seen by the data footprint. Therefore, it is expected that the algorithms which have a higher prefetch offset would perform better than those with a lower offset because the time taken to prefetch sequential memory takes less time than accessing the same memory in parts.

3) mcf: The data access in mcf are not spatial because of which a learning algorithm would not end up using a large offset for prefetching. Due to this, the algorithms mostly use the prefetch value of 1 and hence, have almost the same IPC.

III. AMP ALGORITHM

The AMP algorithm [1] is based on the idea of adaptive asynchronous prefetching. Apart from the degree of prefetch, it uses one other variable called the trigger distance which is used to specify a distance from the last prefetched page. A trigger distance of 0 means new pages will be prefetched only when the last page of the prefetched set is hit, on the other hand, a distance of 1 implies new pages can be prefetched as soon as the first page in the prefetched set is hit. This algorithm learns both the degree of prefetch and the trigger distance.

A. Implementation

Since the dpc2 framework allows access to only L2 read accesses and L2 misses, a few changes were made to the algorithm so that it could be used in this framework. Following are the highlights of our implementation :

- We use an array structure called "cache" which stores what all prefetches has been issued by the L2 prefetcher.
- We use a global prefetch degree variable called "$curr\_offset$" which tells you what is the current prefetch being used by the algorithm. When the last page in the prefetched set is hit, we double our degree of prefetch.
- A page structure is defined similar to how it was defined in the paper and is used to store information about a prefetched page.
- Instead of changing the trigger distance directly, we change a float variable ("$trigger$") which is then used to calculate the trigger distance
- On a cache hit, we inspect if the page that is hit is within the trigger distance of the prefetched set or not. If it is, we prefetch the next set of pages.
- On a cache miss, we prefetch pages equal to "$curr\_offset$"
- In the case of cache fill, if the evicted address was one of the prefetched addresses, then "$curr\_offset$" is reduced by 1 and the trigger variable is reduced by 0.1.
- The degree of prefetch has been limited to 120 as a value higher than this will not be useful.

B. Evaluation

The algorithm was able to adopt a good degree of prefetch during the course of the execution as can be seen by comparing the results between the 3 winners and our implementation. Our algorithm even performs better than all 3 in the omnetpp benchmark.

IV. CONCLUSION

Prefetching is used to boost performances by fetching data or instruction to a faster local memory (cache) ahead of time. Prefetching exploits the observed fact that data requested from main memory have a spatial locality. The performance boost attainable from prefetching of data or instruction depends on the prefetching algorithm in use. Common prefetching algorithm like the next-line prefetcher, fetch the immediate next line from the main memory. This is based on the assumption that data adjacent to the current data might be requested later in the program. It is fair to assume this in case of fetching instructions and data structures like arrays.

We analysed the performance of a few algorithms from DPC2 and noted how they differed from each other. For a

greater insight we implemented the AMP prefetching algorithm and compared the benchmark scores against the other algorithms. It could be concluded from our observations that performance of an individual algorithm depends highly on intelligently guessing the next line to be prefetched, when to prefetch, and at the same time keeping the overhead of prefetching on the memory bandwidth to a minimum.

REFERENCES

[1] Binny S. Gill and Luis Angel D. Bathen. Amp: Adaptive multi-stream prefetching in a shared cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 26–26, Berkeley, CA, USA, 2007. USENIX Association.

[2] Hparch. The 2nd data prefetching championship (dpc2).

[3] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd international conference on Supercomputing*, pages 499–500. ACM, 2009.

[4] Ibrahim Burak Karsli, Mustafa Cavus, and Resit Sendag. Pefetching on-time and when it works.

[5] Pierre Michaud. A best-offset prefetcher. In *2nd Data Prefetching Championship*, 2015.

[6] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982.

[7] Vinson Young and Ajit Krisshna. Towards bandwidth-efficient prefetching with slim ampm.