Data Prefetching Championship

Lab3-Advanced Memory Systems

part A

Ramyad Hadidi

March 6, 2015

Abstract

This is the report for Lab3 assignment for Advanced Memory Systems (ECE-7103A) taught by Moinuddin Qureshi on Spring 2015. This assignment is about data prefetching policies. The framework is provided by second data prefetching championship (DPC2). The framework has some traces and basic policies. In this assignment we will measure the performance and sensitivity to prefetching degree of these policies with provided traces. Then, we will develop an online mechanism to change these policies during execution. Finally, we will design a new policy of our own.

1 Introduction

In this lab we experiment with different data prefetching policies. Simulation framework is provided by second data prefetching championship (DPC2), available on `http://comparch-conf.gatech.edu/dpc2/`. In this section we get familiar with policies, workloads, system configuration and how to execute simulation. In the following sections, first we measure the performance of these four prefetchers and compare them with no prefetching performance. Second, we will change the degree of provided prefetchers and measure the sensitivity of each workload to aggressiveness of the prefetcher.

1.1 Included Policies

Provided framework has these three included policies:

Next Line Prefetcher : A simple prefetcher which will prefetch next address of current accessed cache line.

Program Counter Based Stride Prefetcher : A stride prefetcher which will detect same strides coming from a single program counter and then prefetches additional lines.

Stream Prefetcher : This prefetcher monitors a window of addresses in a page. After confirming the direction of the accesses, it prefetches additional lines.

Access Map Pattern Matching (AMPM) Prefetcher : This prefetcher monitors a page for accesses. Then, it tries to match a pattern to the accesses. If the pattern matches, it do the prefetching. Patterns are offset prefetches with different offsets. However, this prefetcher is the lite version of the original AMPM prefetcher which has a larger monitoring address area.

1.2 Workloads

The framework has some short traces. For this assignment these traces will be used. They are 8 traces from SPEC CPU 2006 with almost 3 million instructions each. Here is the list of these workloads:

gcc	GemsFDTD
lbm	leslie3D
libquantum	mcf
milc	omnetpp

Table 1: Workloads

1.3 System Configurations

Framework models a simple out-of-order core with three level of caches. The memory model consists of a 3 level cache hierarchy, with an L1 data cache, an L2 data cache, and an L3 data cache. Instruction caching is not modeled. The L1 data cache is 16KB 8-way set-associative with LRU replacement. The L2 data cache is 128KB 8-way set-associative with LRU replacement. The L3 data cache is 16-way set-associative with LRU replacement. The size of L3 cache and bus bandwidth is a configuration parameter. The L2 data cache is inclusive of the L1, and the L3 data cache is inclusive of both the L1 and L2. Prefetcher works at cache line granularity and works in the physical address space only, and is restricted from crossing 4 KB physical page boundaries. The prefetchers can only see the stream of L1 cache misses / L2 cache reads (including those L2 reads caused by an L1 write miss). In other words, the prefetcher lives at the L2 level of cache. For more detailed description look at `http://comparch-conf.gatech.edu/dpc2/simulation_infrastructure.html`.

1.4 Framework

Prefetchers code are in `example_prefetchers` directory. The simulator code is provided as a library in `lib` directory. To compile an executable of the simulator below command is used:

```
gcc -Wall -o dpc2sim example_prefetchers/stream_prefetcher.c lib/dpc2sim.a
```

Simulator read traces from stdin. So to execute a gzip trace:

```
zcat trace.dpc.gz | ./dpc2sim
```

For easiness, I have written a python script, `run.py` to compile and execute different prefetches in `example_prefetchers` directory. Below is the help of this script:

```
usage: run.py [-h] [--dryRun] [-o OUTPUTDIR] [-e [EXE [EXE ...]]] [-s]
              [-d DEGREE]

run.py

optional arguments:
  -h, --help            show this help message and exit
  --dryRun              Print out the generated script(s) instead of launching
                        jobs
  -o OUTPUTDIR, --outputDir OUTPUTDIR
                        Root directory for result files
  -e [EXE [EXE ...]], --exe [EXE [EXE ...]]
                        Executables to run
  -s, --submit          Submit Jobs to qsub
  -d DEGREE, --degree DEGREE
                        Degree of Prefetcher, for name generation
```

This script save the output of the simulator to a directory. After that, finding the corresponding performance is easy with `grep` command.

2 Policies Performance

Figure 2 shows the performance speedups for different policies. X axis is the traces and Y axis is speedup based on IPC. Positive values shows speed up and negative values shows slow downs. AMPM is the best policy across all other policies. The online policy is my own designed mechanism to detect best policy in execution.

3 Sensitivity of Workloads to Prefetcher Aggressiveness

In this section we evaluate the sensitivity of each workload to the aggressiveness of prefetchers. Aggressiveness is defined as prefetching degree in the code. It means that how many prefetches will be done for a demand access. In following subsections we investigate this for different policies.

3.1 Next Line

As figure 3.1 shows, increasing next line prefetcher degree causes cache pollution and also a significant negative speedup. This happens mostly for omnetpp, mcf

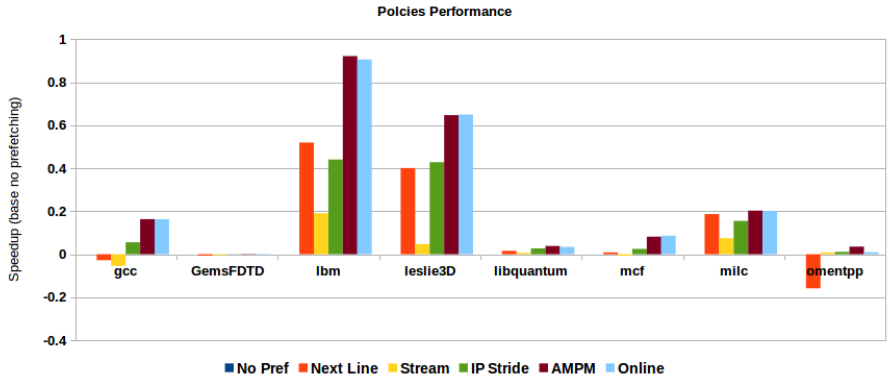

Figure 1: Policies Speedup in IPC - Normalized to No Prefetching

and gcc which as next subsection will show have not a very good detectable accesses. Therefore, when increasing the degree it will cause the useful cache lines to be evicted. lbm and libquantum are only workloads that get speedup with higher degree. As figure 3.3 shows, these two has a good speedup in stream prefetcher too. This suggest that these workload accesses have a locality which next line prefetcher exploits. However, we can not see the same result for leslie3D. This could be because of negative distance accesses in leslie3D. Therefore, leslie3D can get speedup in stream prefetcher and not next line prefetcher (which only prefetches next lines and not previous lines). GemsFDTD probably has a random access pattern, since non of the prefetchers have speedup with GemsFDTD (figure 2). For milc workload, it is possible have different patterns as figure 2 shows for prefetcher stream and ip-stride. However, increasing next-line prefetcher's degree caused more pollution than gains.

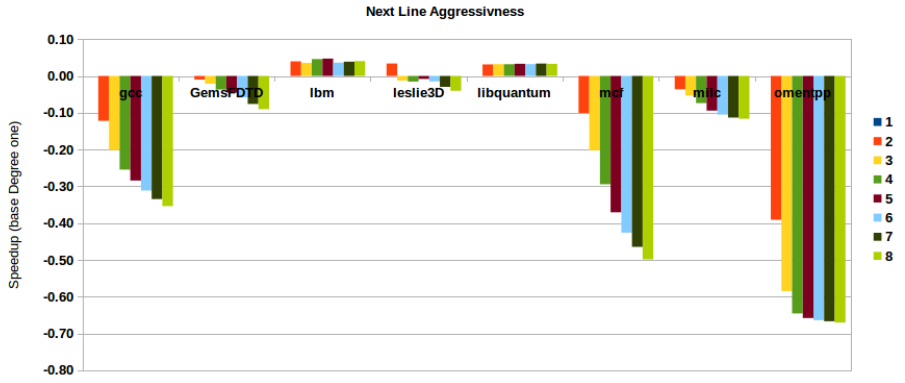

Figure 2: Next Line Policy Aggressiveness impact on Speedup - Normalized to degree one

3.2 IP Stride

The results is in figure 3.2. As we can see except milc workload, almost not other workloads get the speedup with higher degree for ip-stride prefetcher. Probably this is because almost all access are stride accesses in milc. This can be seen in figure 2 as the speed up of a more complex pattern detector like AMPM is the same as simple ones like ip-stride prefetcher. Degradation of speedup in other workloads it is probably because ip-stride prefetcher has no confirmation based prefetches like stream prefetcher (3.3). As for stream prefetcher could be seen, because this prefetcher need confirmation its degradation with degree is less than ip-stride.

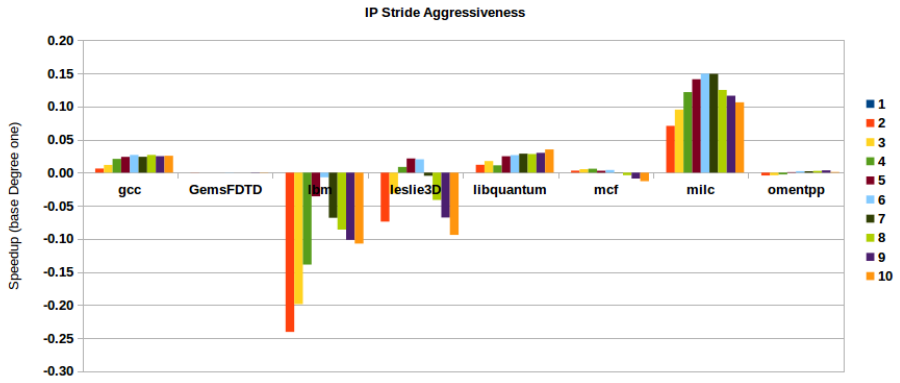

Figure 3: IP Stride Policy Aggressiveness impact on Speedup - Normalized to degree one

3.3 Stream

For stream prefetcher, workloads lbm, leslie3D and milc get the most benefit with prefetcher degree increments. milc, as we talked about before, is consist of strided accesses therefore it gets benefit with more degree. lbm is the only benchmark that gets more benefit for next-line prefetcher (figure 3.1), it suggest that it will also get benefits from stream prefetcher. Since, stream prefetcher is like a confirmation based next/previous line prefetcher. gcc, omnetpp, GemsFDTD and mcf are all like each other. Stream prefetcher cannot detect their misses. Also, gcc has a negative speedup in stream prefetcher. This is probably because gcc have a random locality, therefore cache pollution (which is stream is higher) has more impact.

3.4 AMPM

For AMPM larger degrees act as degree of two as seen in figure 3.4. This could be because of various reasons. First, it would be because there is not matching

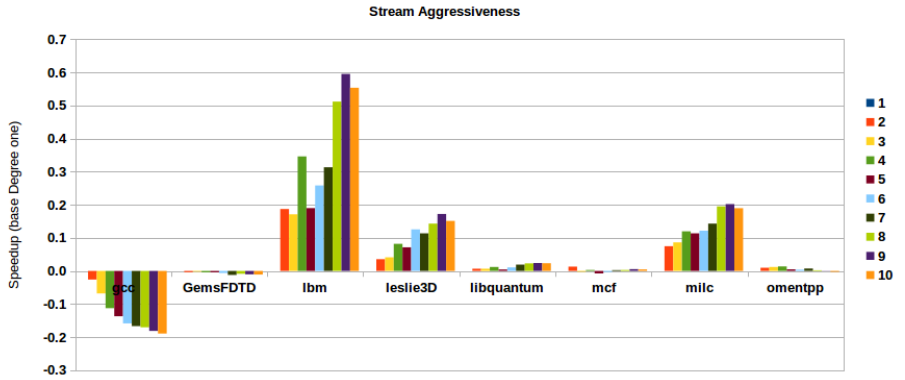

Figure 4: Stream Policy Aggressiveness impact on Speedup - Normalized to degree one

in the patters. Or, there as a match, since the code has a lot of breaks in it, it usually does not prefetch the data. This is because this version of AMPM monitors a smaller indexes and also it can only fetch in the current working memory page.

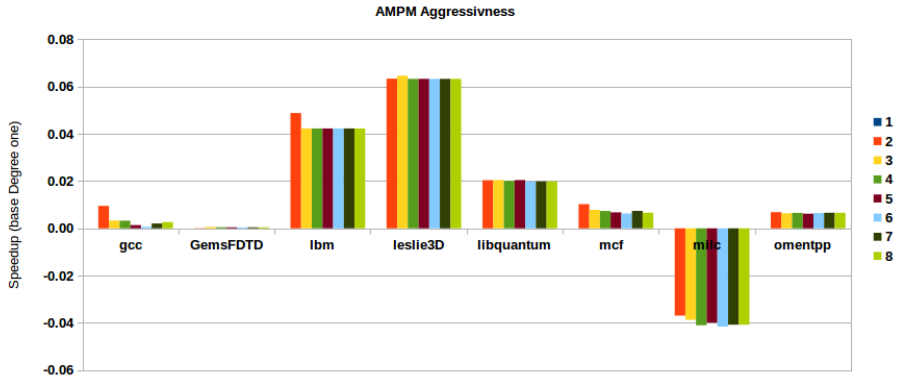

Figure 5: AMPM Policy Aggressiveness impact on Speedup - Normalized to degree one

4 Online Prefetching Mechanism

I tired different kinds of ideas to change prefetcher during execution. I included two of them here: Parallel sandboxes and page dueling. I used parallel sandboxes for my online policy. However, to be able to get the performance near AMPM I had to add many others features. This problem of lower performance in online

policy raise mainly because:

- AMPM is a mix of Stream and Next-Line prefetcher, therefore it is not a good idea to mix these three together.
- I understand that Next-Line prefetches usually will be hit in the cache. This is because high locality of access.
- Also, amount of traffic generated by IP stride and Next-Line prefetcher is usually high, therefore their relative number of hits are higher

In following subsections I have described my online policies and also how I overcame these problems in my implementations. I have used parallel sandbox method for my implementation. I did not implement the other two methods, but I have explained their strengths and weaknesses.

4.1 Parallel Sandboxes

I get the idea for this mechanism from the last paper about prefetchers in HPCA 2014, "Sandbox Prefetching: Safe Run-Time Evaluation of Aggressive Prefetchers". However, since there is no budget limitations, I implemented the parallel version of this paper to evaluate four different prefetchers. General mechanism is done on every fixed number of L2 accesses. In every period, each prefetcher imports its prefetches lines in their own assigned sandbox. In subsequent accesses, we also check these sandboxes to see if a prefetcher predict a line correctly or not. If yes, we will increase the score of that prefetcher. In the end of that period, we make a decision about the next period's prefetcher based on these scores. To overcame the problems I mentioned above I change the sandboxes like this:

- I always carry a fraction of score from previous period to the current period. Current coefficient is 3, which means 1/3 of the last period's score will make the new score. This will help to detect the best prefetcher in a longer monitoring period ans omit large spikes.
- I edited next-line prefetcher to only operate on cache-misses since this prefetcher's performance is not very good in compare to AMPM. This is because of high locality, if we hit the current line, it is possible to hit the next-line to. With this change I reduced the number of prefetch address generated by this prefetches, therefore this prefetcher will not get a higher score because of higher locality.
- To omit high number of locality hits, I have delayed score counting a period later. Therefore, I will actually count the scores for prefetches that are really separated in time. Since the closer the access are, it is more probable that we have the cache line already in our cache.

4.2 Page Dueling

In this model we will monitor different pages which will use a dedicated prefetcher, and based on the performance they get for that page we will extend the decision to the other pages. However, there are many complications. First, programs usually use some fixed pages and accesses are not spread out between all pages. Second, it is highly probable that a program only operates in a single page for a short period, therefore it is really hard to make a decision. To solve this problem, we could divide each page between our prefetchers. However, since our prefetchers usually work in a page limit, this separation will have a negative impact on their performance.

4.3 Feedback Directed Prefetching

Another promising method to measure the performance of prefetchers is presented in this paper: "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers". The method proposed in this paper decides between stream prefetchers with different distance and degree based on three different metrics. These metrics are prefetcher accuracy, timeliness, and an estimation of cache pollution. Categorizing of how change the aggressiveness of prefetchers based on these metrics are done dynamically but based on static thresholds. Although we could measure the performance of each prefetcher with their accuracy and pollution they cause, for timeliness we do not have access to MSHR. For implementing these we need to duplicate the cache contents and add a bit for prefetching to them because we do not have access to cache lines directly. Also, another problem is how the accuracy of a prefetcher implies better performance. In some cases there is a direct relation, on some cases not. For instance, a prefetcher can have high accuracy compared to other one, but the performance of the other one is better since it prefetches more lines. Therefore, we need a notion of coverage. The problem with this method is how we are going to decide the thresholds of these metrics. It is a time consuming procedure to find an optimal values for these thresholds, therefore I have chosen the parallel sandbox method.