Prashant Singh Chouhan

Simulating and Evaluating Processor Cache Stream Buffers

Abstract:

With the increasing gap between processor speed and memory access time, memory speed has become a bottleneck in performance of any CPU. To ameliorate this problem, a small, fast cache memory is introduced between processor core and the main meory. But caches also have the problem of cache misses which degrades the overall performance of a system. To avoid the cache misses, prefetching techniques are used. One such popular technique was described by Joupi [1]. He proposed a mechanism know as stream buffer. Stream buffer is a very small memory used to keep the prefetched data and instruction instead of putting them in caches. They avoid cache pollution and give better performance then most of the other prefetching techniques.

Introduction:

CPU caches were introduced to improve the performance of a computer system by decreasing the access time of frequently accessed main memory values. It temporarily stores the copy of main memory data. It benefits from the inherent locality of general programs. Locality can be broken down into two types-- spatial and temporal. Spatial locality dictates that the probability of accessing the successive memory address is very high and temporal locality means that the memory address which was accessed just now will be re-accessed very soon. But since caches are small in size, the complete programs cannot fit in it leading to cache misses. A cache miss occurs when the target address requested by processor is not available in cache and will have to be fetched from the lower level memory. Since lower memories are not as fast as caches, it takes multiple cpu clock cycles to complete the fetch and during this time CPU will have to stall. When the requested address location is found in cache it is known as cache hit. Cache hits improve system performance.

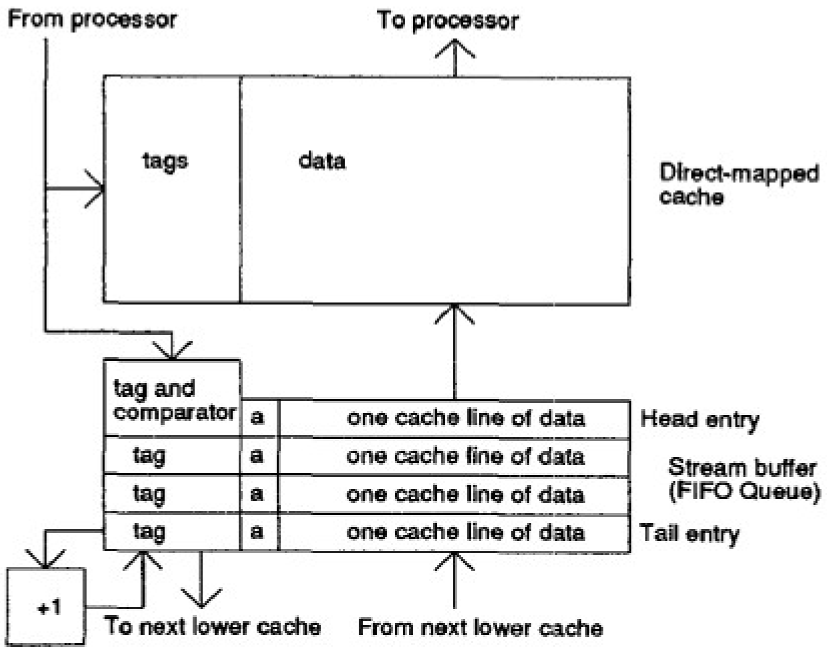
Every computer architect's goal is to lower cache misses and increase cache hits. Cache prefetching is one such technique used to increase cache hits by trying to fetch the memory addresses before they are demand fetched by the processor. Prefetching helps hide memory latency by increasing memory bandwidth or by using unused memory cycles for prefetching. Prefetching algorithms tries to mimic memory access pattern of program and possibly fetch the memory address required by the program in future.

Some of the most widely used prefetching techniques are always, miss and tagged prefetching. In always prefetch, block next to the currently accessed one is prefetched on every access. In miss prefetch, whenever a miss occurs, the missed block fetched and its next block is prefetched. In tagged prefetching, there is a tag bit associated with every cache block. Whenever the cache block is brought first time into the memory, whether by prefetching or demand fetching, the tag bit is set to one. And as soon as it is accessed, the tag bit is set to zero and the successive block is prefetched. The successive block is not prefetched if the tag is already zero. Mostly always prefetch gives fewer miss ratio but increases the memory bandwidth with lots of prefetches, the miss caching has lesser memory bandwidth requirements but doesn't gives a good miss ratio. Tagged prefetching gives fewer miss rate and also doesn't increase the memory bandwidth requirements.

I have implemented stream buffer prefetching mechanism, introduced by Joupi [1], in this project. Stream Buffer is a small memory present between L1 cache and L2 cache used to keep the prefetched blocks. It consists of a series of entries, where each entry has three parts- a tag, an available bit, and a data line.

When a miss occurs, the stream buffer starts prefetching successive lines from the L2 cache. It prefetches the number of blocks equal to the number of entries in buffer. It sets the tag in the stream buffer and sets the available bit to false for the requested block. When the requested block arrives, it sets the available bit to true and the block can be accessed now. Since the prefetched blocks are placed in the buffer instead of cache, this technique gives better results then the above mentioned three prefetching techniques because it does not pollutes the cache caused by replacing useful cache blocks by the prefetched blocks.

Now whenever a request from processor arrives to the L1 cache, the requested block is also searched at the head of the buffer. If the block misses in the L1 cache and is found in the buffer then it can brought to the cache in just one cycle avoiding the miss. The stream buffers I have implemented are simple FIFO queues where head of the queue has a tag comparator used to compare the first entry in the stream buffer. Whenever a miss occurs and the missed block is present deeper in the queue, it will not be found by the comparator and hence the entire buffer will be flushed. When the prefetched block is moved from buffer to the cache, the entries are shifted up by one place and a new block is prefetched at the last vacant spot.



Single Stream Buffer

Implementation Details:

I have modified DineroIV [3] cache simulator to support Stream Buffers. I have modelled one additional prefetch strategy then given by Joupi [1]. My first technique is the basic strategy used by Joupi where we flush the buffer and start to prefetch on every miss and then prefetch one block on every successive hit in the buffer. I call this strategy as miss stream prefetching. The other strategy I have modelled is known as always stream prefetching which works similar to always prefetch introduced earlier. On every access to the memory, it will flush buffer if it was not a hit in the buffer and prefetches k successive blocks into the buffer. On a hit, it will shift up the buffer by one and will prefetch just one block.

When the stream buffer is empty, we will prefetch k blocks which is equal to the stream buffer's size. In my project I have used k=4. Also, I have implemented a single stream buffer which can handle single sequential pattern in the address stream. For accesses to the data cache which have non-unit strides, single stream buffer won't suffice because it will keep on flushing the stream buffers without producing any good results. For such streams, a Multi-way Stream Buffer is required.

I have used three levels of memory in my simulator-- L1, L2 and Main Memory. L1 cache is split between Instruction and Data while L2 is unified. L1 and L2 are both directly mapped.

The benchmarks I have used for my project are:

1. Dhrystone
2. Linpack
3. GCC

I have used Intel's pin tool [2] to produce data and instruction memory traces for all the three benchmarks and pipe them to dineroIV for simulation.

I recorded my results for three different cache block sizes-- 16, 32 and 64 Bytes, and three different L1 I&D cache sizes-- 4, 8, 16 KB.

I have plotted data to find out effect of different cache size and cache block size on the stream buffers. I have also plotted data for different stream buffer fetching techniques to find out the best one.

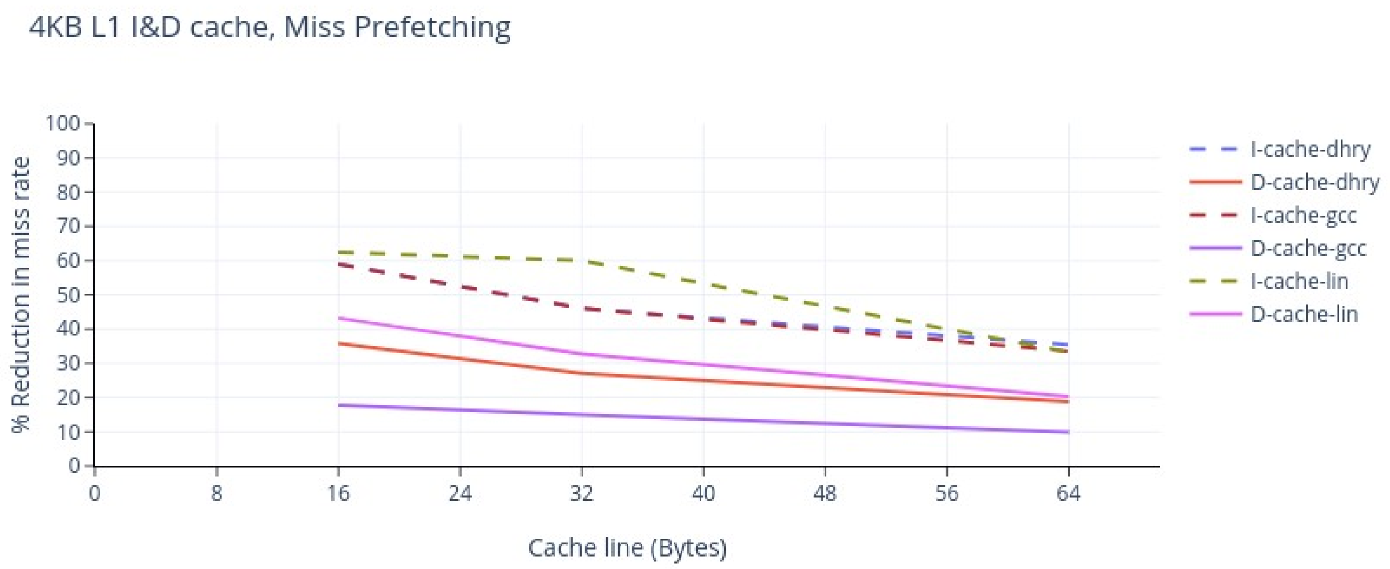
Cache prefetching policies can be compared using two parameters-

1. Prefetch coverage = Cache misses eliminated by prefetching / Total cache misses
2. Prefetch accuracy = Cache misses eliminated by prefetching / Total prefetches

I will use these policies to evaluate prefetching policies implemented in this project.

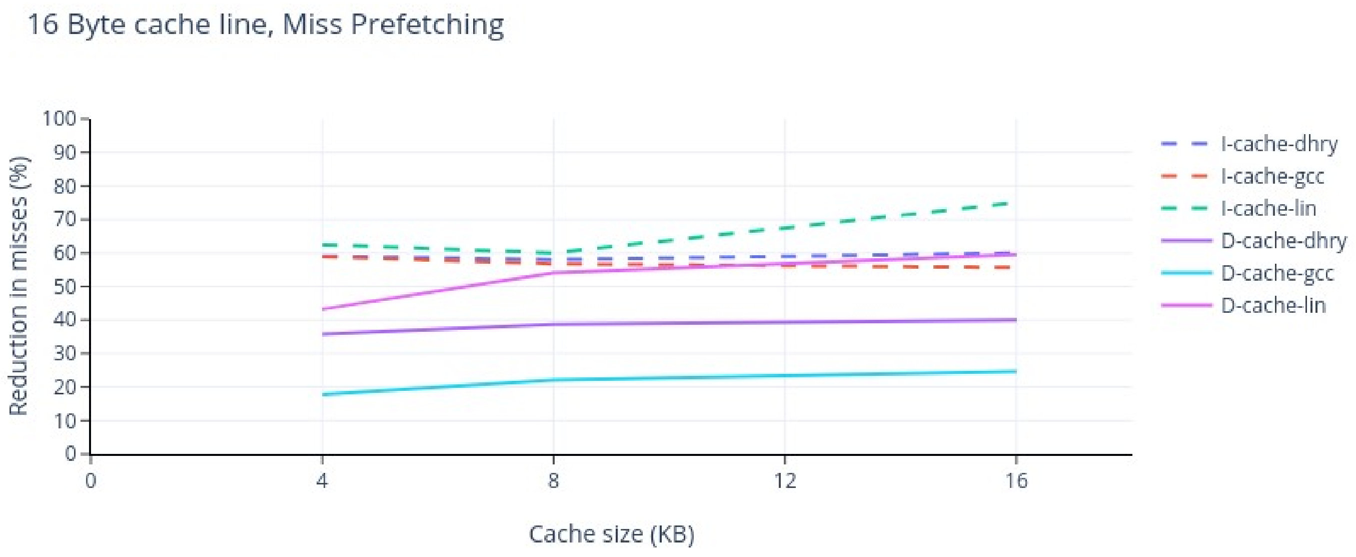
Results:

Cache line vs Miss reductions



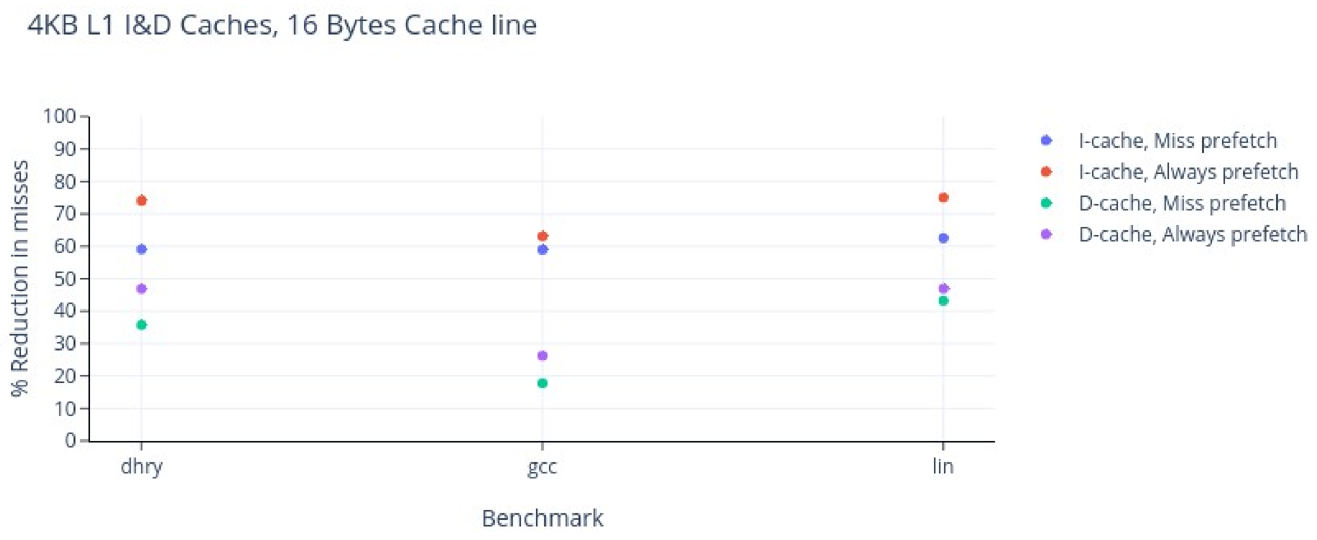
This plot shows effect of miss prefetching on different benchmarks with different cache line sizes. Here it can be observed that stream buffers most effectively remove instruction cache misses because instruction stream is more sequential in general than data stream. Also, it shows the effect of line size on the miss reduction. As we increase the cache line size, the amount of reduction reduces. This can be explained by the fact that as we increase the cache line size, we are already bringing in more amout of instruction and data in a single fetch which is similar to prefetching in sequential order. So it already reduces the sequential misses leaving less amount to be removed by prefetching.

Cache size vs Miss reductions



This plot shows the effect of cache size on the reduction in misses. As can be seen in the plot that as we increase the cache size, the number of misses also reduces. As we increase the size of the cache, the number of capacity misses also reduces which leads to this observed pattern. We can also see that the miss reduction for linpack suddenly bumps up from 8KB to 16KB cache size, for both the instruction and data cache. This can be explained by the fact that linpack had the largest number of instruction and data references which were leading to a lot of cache misses, but as soon as we increase the cache size from 8KB to 16KB, most of all the references fits in the cache leading to the reduction.

Prefetch policy vs Miss reduction



As can be seen from the graph that always stream prefetch almost always performs better than miss stream prefetch for all the benchmarks. This can be explained by the fact that in miss prefetching, a prefetch always starts after a miss has occured but in always prefetch, it need not wait for the miss to occur.

Conclusion:

Although always prefetch performed better than miss prefetch, the amount of prefetches generated by always prefetch were overwhelming.

Average prefetch accuracy of always prefetch is 0.24 and for miss prefetch is 18.3

Average prefetch coverage of always prefetch is 41.3% and for miss prefetch its 38.8%.

By looking at the above data we can easily to conclude that miss prefetch outperforms always prefetch.

For detailed data analysis, refer to [5]

Future work:

I plan to extend this work and implement multi-way stream buffers, which can handle non-unit strides and multiple sequential streams in the data. I also plan to evaluate the effect of changing the length of stream buffers on different benchmarks. Furthermore, Jouppi suggested to access the stream buffers in the FIFO fashion, but I plan to also evaluate the performance of using a fully-associative cache. It can further lower the amount of misses as the blocks hidden deep down in the buffer can be found in a fully-associative cache.

References:

[1] Jouppi, Norman P. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers." *ACM SIGARCH Computer Architecture News*. Vol. 18. No. 2SI. ACM, 1990.

[2] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[3] http://pages.cs.wisc.edu/~markhill/DineroIV/

[4] https://en.wikipedia.org/wiki/CPU_cache

[5]https://docs.google.com/spreadsheets/d/
1rVVVKHMMFtEjnqc37KIXsMy2XOZvZux7AJH_LfsRuas/\edit?usp=sharing

[6] Code for the project: https://github.com/ItsPrashant/Stream-buffers