Exploratory Study of Latest Prefetching Technologies

Aditya Ramkumar
Divyam Goel
Ritesh Singh
Shubhang Tripathi
Sushant Gour
Vinay Kumar Jain

31 October 2019

Abstract

The past decade has seen an exponential rise in computing speeds and abilities, which only seem to be limited by the laws of physics and the inherent memory transfer latencies due to the imperfections in our designs. Prefetching, i.e, exploiting the overlap of processor computations with data accesses, is one of several approaches to improve our tolerance towards these memory latencies. The process itself is both Hardware and Software directed with some of the most sophisticated systems combining both these techniques. In this Exploratory Research Project we will try to unmask the qualitative and quantitative impacts of existing prefetching methodologies with special focus on correlation based techniques. When complex data access patterns are considered, the software approach has compile-time information to perform sophisticated prefetching whereas the hardware scheme has the advantage of manipulating dynamic information.

1 Introduction

Prefetching occurs when a processor requests an instruction from main memory before it is actually needed. Once the instruction comes back from memory, it is placed in a cache. When an instruction is actually needed, the instruction can be accessed much more quickly from the cache than if it had to make a request from memory. Since programs are generally executed sequentially, performance is likely to be best when instructions are prefetched in program order. Alternatively, the prefetch may be part of a complex branch prediction algorithm, where the processor tries to anticipate the result of a calculation and fetch the right instructions in advance. In the case of dedicated hardware the prefetch can take advantage of the spatial coherence usually found in the texture mapping process. In this case, the prefetched data are not instructions, but texture elements that are candidates to be mapped on a polygon.

2 Motivation

The performance of modern computers is limited by the memory wall. They must make a trade-off between memory size, cost and access latency. Multi-stage caches introduced to mitigate this trade-off bring with them their own delays in the form of cache misses. Prefetching techniques aim to reduce these latencies by decreasing the number of off chip cache misses, anticipating and accommodating for irregular control flow and complex data access patterns and attempting to find connections and patterns in miss streams. This paper aims to study the most efficient ways to predict which lines to prefetch, the rate at which data is prefetched and the quantity of data prefetched that provide optimum performance on a given task.

3 Stream and Stride Prefetching

The Predictor Directed Stream Buffer uses a Markov predictor to generate the next addresses rather than employ a fixed stride with each buffer. Multiple iterations of this procedure follow, following which, the stream buffer is decoupled from the execution stream. In other words, it can independently prefetch down a predicted address stream.

The PSB architecture resides on chip and prefetches data from the L2 cache and main memory into the stream buffers. The general idea of a PSB is to use a predictor to generate an address stream for prefetching. The predictor takes as input the last address accessed, history information etc and generates a prediction for a

given stream buffer. This prediction is then stored back into the stream buffer, and the stream buffer is updated. The base of the recursion is a cache miss which causes a stream buffer allocation.

Each stream buffer holds (1) the PC of the load that caused the stream buffer to be allocated, (2) the last predicted address for the load, and (3) any additional prediction information needed to perform the next address prediction. The stream buffer is on-chip next to the address predictor, which in our case is a stride-filtered Markov predictor.

Allocation: A stream buffer is allocated, subject to allocation filters when a load executes and it misses both in the data cache and the stream buffer. If a load miss is assigned to a stream buffer the prediction information is copied to the it from the address predictor. This initialization stage is only done once per allocation.

Prediction: In each cycle, one stream buffer is chosen to make a prediction according to the priority heuristics. The information stored in the it is used to index into the address predictor, returning the next predicted address, and potentially updating the stream buffer's history information. Due to the fact that only one request (miss or prefetch) can be processed by the bus from the L1 to the L2 cache at a time, the predictor was not a bottleneck even with the one prediction per cycle limitation. The history of the last N addresses stored in the stream buffer is updated after a prediction, not the state in the address prediction table. Before inserting the prediction into the stream buffer, they are searched in parallel for the cache block of the predicted address. Once all entries have been predicted for a stream buffer, no further entries will be predicted until (1) an entry is cleared during a lookup (it is a hit), or (2) the stream buffer is reallocated.

Prefetching: Once an entry has a valid prediction associated with it, it is ready to be prefetched. When the L1-L2 bus is free, a stream buffer with valid entry is chosen using the priority scheduling algorithms. The prefetch is then sent to the lower levels of memory and the entry is marked as prefetched and waiting.

Lookup: It is assumed that the data cache lookup latency is 3 cycles whereas the stream buffer lookup latency is a single cycle. If there is a tag hit in the stream buffer, but the block is not ready yet, the tag is moved into a data cache MSHR, and the data cache handles the block when it comes back from memory. For a stream buffer hit, the corresponding stream buffer entry is freed for a new prediction and prefetch.

Stride-Filtered Markov Predictor: In order to reduce the size of the Markov predictor table we store into the table only the difference (at the cache block granularity) between consecutive cache miss addresses.

Stream Buffer Priority: The first heuristic is Round-Robin, giving each buffer an equal chance at performing a prediction or prefetch. A pointer is kept to the last stream buffer to perform a prediction and another pointer for the last entry to issue a prefetch. The second heuristic uses Priority Counters to guide which stream buffer gets to perform the next prediction or prefetch. Every time there is a lookup and the stream buffer gets a hit, the priority counter is incremented by a constant value (2 in our implementation).

Previous stream buffer architectures were limited to streaming only address patterns with a fixed stride, which limits their benefit for commercial pointer-based applications. Predictor-Directed Stream Buffers provided a 75% speedup on average over no prefetching, and 23% average speedup over the best performing prior stream buffer architecture as analysed in the paper published by Sair, Sherwood and Calder.

Even with the enhancements presented above, the stream stride prefetching techniques fail to overcome certain inherent performance bottlenecks like limited bandwidth overloading. Also, long stride accesses across more than one block, prefetching consecutive blocks are wasteful and pollute the cache. To overcome these bottlenecks, we continue our analysis on other prefetching techniques.

3.1 Wrong Path Prefetching

Wrong path prefetching was introduced by Pierce and Mudge . Next line, target line and hybrid schemes all try to prefetch instructions that are in the correct execution path of the program. However, wrong path prefetching prefetches on the simplest wrong path. It prefetches target lines that are not taken. This is done in hopes that mispredicted instructions brought into the buffer during speculative execution will turn out to be prefetched for later correctly predicted instructions.

Wrong path prefetching combines both next line and target prefetching. For the target line portion, no target line address information is saved and no attempt is made to prefetch only in the correct execution path. Instead the line containing the target of the conditional branch is prefetched. Thus, both paths of the conditional branch are always prefetched. At the first look, it would seems that hybrid and wrong path prefetching are similar as they both prefetch both paths of the conditional branch. However, hybrid prefetching prefetches instructions that will be executed almost immediately with the help of a prefetch history table. Wrong path prefetching on the other hand prefetches instructions that will not be executed immediately and no history information is needed. However, in a typical pipeline, because target of a branch is computed at such a late stage, prefetching the target when the branch is taken is unproductive. A cache miss and a prefetch request will be generated at the same

time. Thus, the target prefetch portion of wrong path prefetching can only perform potentially useful prefetch if the branch is not taken. It is hoped that if execution returns to the branch in the near future and the branch is taken, the target line will already be in the cache from the previous prefetch.

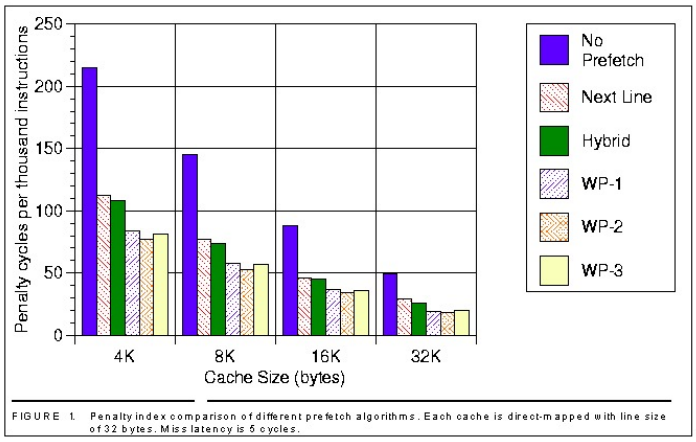

Figure 1: Penalty Index Comparison

The advantages are-

- No or very little extra hardware is required. The next line portion requires minimal additional hardware, and the address from the target portion is calculated by the existing instruction decoder.
- Performs better if there is a large difference between CPU and memory speed.This is because in wrong path prefetching, prefetches are down a path which is not immediately taken, thus there is more time to prefetch the line from slow memory.

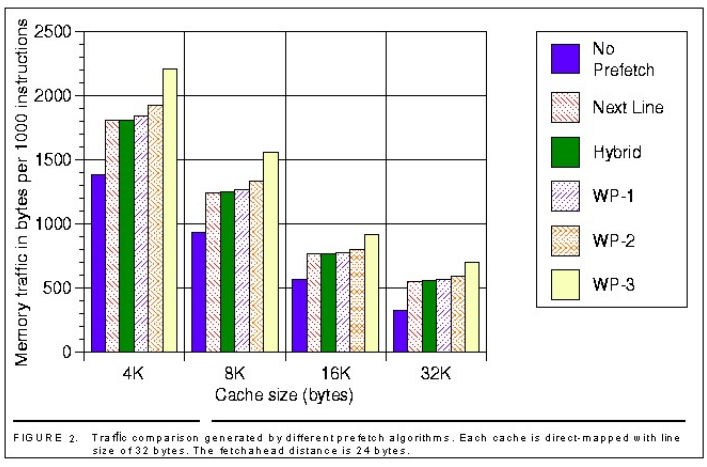

Figure 2: Traffic Comparisons

The disadvantages are- Increase in memory traffic and cache pollution as lines may be prefetched down the wrong path that will never be executed.

4 Correlation Prefetching

Today's systems primarily employ stride-based prefetchers, which require only simple hardware additions and minimal on-chip area. However, these prefetchers are only partially effective in commercial server workloads, such as on-line transaction processing, which are dominated by pointer-chasing access patterns. In contrast, address-correlating prefetchers are effective for repetitive, yet arbitrarily irregular, access patterns, particularly pointer chasing. Address-correlating pre-fetchers associate a miss address with a sequence of possible successor misses.An address-correlating prefetcher learns temporal relationships between accesses to specific addresses. For instance, if address B tends to be accessed shortly after address A, an address-correlating prefetcher can learn this relationship and use the occurrence of A to trigger a prefetch of B.

More than a decade of research has repeatedly shown that address-correlating prefetchers can eliminate half (or more) of all off-chip misses in pointer-intensive commercial server workloads. Address-correlating prefetchers have never been deployed in production designs because they require megabytes of address correlation metadata—far too large to store practically on chip.

Some current techniques include-

- Markov prefetcher: The Markov prefetcher hardware comprises a set-associative table that maps an address to several recently observed possibilities for the succeeding miss. On each miss, the prefetcher searches the table for the miss address, and if it finds an entry, it prefetches the likely successors.
- Epoch Based: Program has periods of memory accesses and computations called an epochs. It uses the first miss in each epoch to prefetch all in successive epochs. Can store correlation table in main memory (hide table access latency under previous epoch). The miss address of epoch i is used to prefetch for the misses in epoch i+2 and epoch i+3.

4.1 Helper Thread Based Prefetching

In processors with multithreading, while the main thread executes the program, another thread can redundantly execute full or reduced version of the program to speculatively generate data addresses for performing prefetching. Such a thread is known as speculative slice, pre-execution (or pre-computation) thread, helper thread or future thread. Executing a slice of a program in a software or hardware thread ahead of the normal execution resolves memory addresses and prefetches data into the caches. By doing so the latency of memory reads is reduced in the main thread.

This helper thread could be for example a micro thread automatically started by hardware; could be a software thread extract of the original program started automatically or manually, or speculative extract (slice) of the original algorithm. The goal of pre-execution is to handle arbitrary memory access patterns, such as finding items in a hash table-based dictionary.
Some current techniques include-

- Precomputation based approach to predict prefetch addresses.
- A software-controlled pre-execution scheme to accelerate programs with irregular access patterns.
- Idle HW contexts are used to spawn speculative threads which aim to hide miss latency by triggering upcoming cache miss events well in advance of access by main thread.
- Using a pointer cache (PtC), a dedicated cache that stores pointer transitions in the application, the pointer cache provides, for a given pointer's effective address, the base address of the object pointed to by the pointer.
- An event-driven helper threading approach for software emulation of simple or complex HW prefetchers.

If the main thread is far behind the data the pre-execution thread loads into the cache could get overwritten by other data before it is referenced by the main thread; on the other hand if the thread are to close, the pre-execution thread has little chance of making the prefetch early enough and becomes a waste of system resources making no contribution to the execution time.(Temporal distance should be kept optimal).

5 Memory side prefetching

In memory side prefetching, the prefetch engine resides in main memory and issues commands from there. This approach offers a number of advantages-

- Eliminates the overheads introduced by prefetch requests in the path between processor and caches.
- Does not require modifications to processor core
- Frees up space on chip
- Can exploit proximity to main memory

The memory processor observes the requests from the processor core that reach the main memory and based on this information it prefetches other lines that it expects will be requested. Since it effectively only sees the L2 cache miss stream, it attempts to eliminate L2 cache misses by pushing prefetched data into the L2 cache. L2 cache miss penalty is the largest component of memory access latency.
The response time is the time beginning when the prefetching thread obtains a miss address to when it produces the prefetch addresses. The occupancy time is the time the thread is busy and cannot process another miss.

5.1 Pair Wise Temporal Correlation Methods

Pair based algorithms try to identify a correlation between pairs of misses – say a miss and its immediate successor. They record a sequence of miss addresses in a table. When a miss occurs, they look up the table, identify the sequence and prefetch the rest of the sequence. For a sequential prefetcher, the upcoming address matches exactly matches the one predicted while for pair based schemes it must match one of the predicted successors. The following techniques are implemented via a user level helper thread.

5.1.1 Basic Table Organization

Each row stores the tag of the miss address and the addresses of immediate successor misses in MRU order. There are two algorithms possible in this frame work : *Base* and *Chain*.
The Base algorithm only tries to fetch immediate successors. Its parameters are number of immediate successors predicted (*NumSucc*), number of misses the correlation table can store predictions for (*NumRows*) and associativity of the table (*Assoc*).
The Chain algorithm prefetches multiple levels of successors. It takes an additional parameter *NumLevels*. If a miss occurs, it prefetches all NumSucc successors in the corresponding row, then does the same for the most-recently-used successor and repeats this process *NumLevels* – 1 times.
While the coverage and latency hiding abilities of the Chain algorithm are superior to the Base algorithm, the response time is high. It does not prefetch the correct MRU successor. It only prefetches successors along the MRU path.

5.1.2 Replicated Table Organization

Each row of the table stores the tag of the miss address as well as *NumLevels* levels of successors. Each level has *NumSucc* successors that are MRU ordered.
The *Replicated* algorithm takes the same parameters as the Chain algorithm. In the learning phase, *NumLevels* pointers pointing to the rows for the last miss, second last miss and so on are maintained for efficient access. When a miss occurs, it's address is stored in the correct position of MRU successors of the last few misses using these pointers.
The response time is much shorter as several levels of successors can be fetched with one row access. The cor-

relation table requires more storage space but as it is allocated in memory, this is a minor issue.

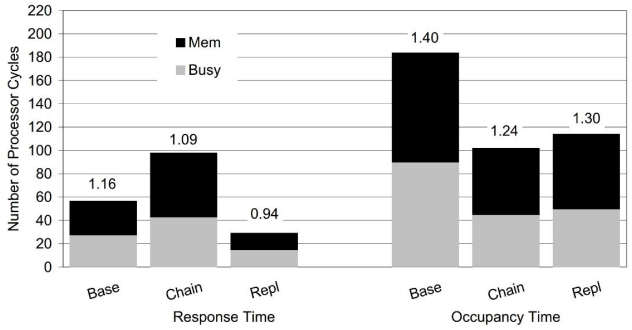

Figure 3: Response and Occupancy Time

6 Region Correlated Temporal Prefetcher

Temporal prefetchers go a step beyond regular preftching techniques by way of predicting future memory accesses based on temporal patterns of existing memory regions and hence try and improve the hit ac curacies of the system. State of the art technologies include ISB and VLDP, both of which have been proven proficient in handling irregular streaming applications. One of the major bottlenecks of the existing technologies has been their inability to predict cache line addresses before their first accesses also these traditional temporal prefetchers require an off-chip storage. One of the latest technologies we explored was RCTP or Region Correlated Temporal Prefetching.

The architecture allows for numerous advantages:

- RCTP has been modelled in a way that allows it to overcome these bottlenecks while providing performance enhancements across various standard benchmarks.
- It involves predicting memory patterns for sparsely populated regions by correlating temporal patterns of existing memory regions.
- On an average, RCTP outperforms both ISB by 26% and VLDP by 6%.

6.1 Implementation

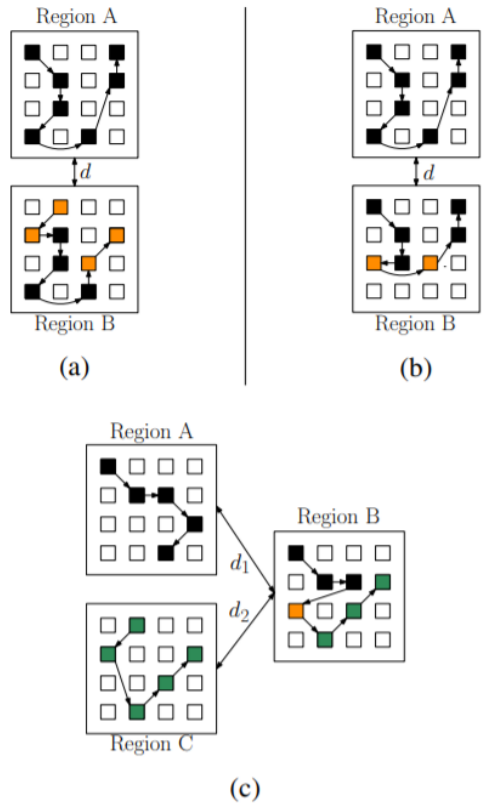

Figure 4: Scenarios of Region Correlation

RCTP introduces “region correlation” and then proceeds to incorporate the classic offset-based correlation by building the region correlation on top of it. This helps tackle the issue of failing prefetch accuracy and consumption of DRAM bandwidth which have been identified as the major performance bottlenecks of offset-based correlation.

As is the case with ISB, RCTP too uses the 8-way caches PS-AMC and SP-AMC to store the meta data of memory accesses. After mapping the offsets from the structural address mappings to the current access, this address is used to index into the SP-AMC to obtain the temporal stream(Ai, Ai+1,. . . .).

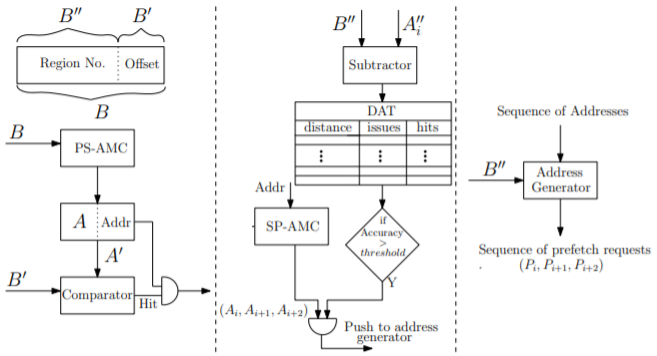

Figure 5: Steps of RCTP

Next, RCTP prunes the addresses from the temporal stream using a new heuristic called the region distance, defined as the value obtained by dividing the difference in the addresses of two regions by the region size. Correlations involving regional distance are found to improve prefetch accuracy. Prefetching is stopped when the accuracy falls in order to prune unnecessary accesses.
The implementation also introduces a new data structure called the Distance accuracy Table(DAT), the components of which are used to calculate the region accuracy for RCTP. The accuracy is thus obtained as prefetch hits /prefetch issues. If the accuracy of the region-distance is above a threshold then the address of the temporal stream is passed to the address generator. The pruned temporal stream is mapped to the current access region in the process for the purpose of address generation. Overall, RCTP has a hardware overhead of less than 33KB per core. Compared to ISB, RCTP has an additional on-chip hardware overhead of 1KB; there is no off-chip hardware overhead in RCTP, unlike as required by ISB (8MB).

7 Conclusion

The bottlenecks presented by stream and stride prefetchers are efficiently mitigated by correlation prefetchers. They use a variety of prediction mechanisms to better utilize the limited storage space available by only prefetching lines with a higher probability of being accessed in the short run. They introduced parallelism through the use of helper threads, thereby reducing the latency of memory reads in the main thread.
The effects of the location of the prefetcher (on chip or in memory) and the mechanism of prefetching (push or pull) were also observed. On chip prefetchers must be more accurate and most operate at a higher speed than in memory. However they take up precious chip space that could have been utilized in other ways (Eg. adding more processor cores).
The superiority or inferiority of algorithms is relative to the computer systems they are used on and must therefore be chosen appropriately. Premature optimization is the root of all evil. Modern CPU's utilize a combination of prefetching schemes at different levels of the memory hierarchy.
By analysing different techniques relevant to specific applications, this paper provides insight in the various directions that one can choose to conform to in the area of prefetching research. The techniques discussed, especially those pertaining to temporal prefetching are yet far from perfect and offer great opportunities for further research work.

8 References

1. Sair, Suleyman, Timothy Sherwood, and Brad Calder. "A decoupled predictor-directed stream prefetching architecture." IEEE Transactions on Computers 52.3 (2003): 260-276.
2. Pierce, Jim, and Trevor Mudge. "Wrong-path instruction prefetching." Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29. IEEE, 1996.
3. Wenisch, Thomas F., et al. "Making address-correlated prefetching practical." IEEE micro 30.1 (2010): 50-59.
4. Dudás, Akos, and Sándor Juhász. "Using preexecution and helper threads for speeding up data intensive applications." Proceedings of the world congress on engineering. Vol. 2. 2011.
5. Mittal, Sparsh. "A survey of recent prefetching techniques for processor caches." ACM Computing Surveys (CSUR) 49.2 (2016): 35.
6. Solihin, Yan, Jaejin Lee, and Josep Torrellas. "Using a user-level memory thread for correlation prefetching." Proceedings 29th Annual International Symposium on Computer Architecture. IEEE, 2002.
7. Varkey, Dennis Antony, Biswabandan Panda, and Madhu Mutyam. "RCTP: Region Correlated Temporal Prefetcher." 2017 IEEE International Conference on Computer Design (ICCD). IEEE, 2017.
8. Chou, Yuan. "Low-cost epoch-based correlation prefetching for commercial applications." Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2007.