A Study of prefetching methods (hardware and software) and their impact on performance

October 31, 2019

List of Group Members:
1. Aanand Vishnu - 18114001
2. Ankita Dhiman - 18116010
3. Dileep Kushwaha - 18116027
4. Ravindra - 18114064
5. Sanchit Gupta - 18116068
6. Sanjeev Kumar Nirmal - 18116069
7. Subhash Suryawanshi - 18117076

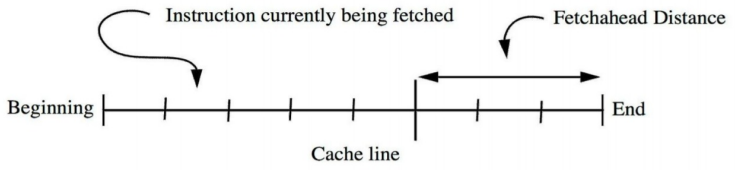
1 Abstract

The purpose of this paper is to study different methods that classifies various design concerns in developing a prefetching strategy. We discuss various prefetching strategies and issues that have to be considered in designing a prefetching strategy for multi-core processors. Data prefetching has been considered an effective way to mask data access latency caused by cache misses and to bridge the performance gap between processor and memory.
A good prediction method must not only be accurate, but prefetches must be initiated early enough to allow time for the instructions to return from main memory. Hybrid hardware/software controlled strategies are gaining popularity on processors with multi-thread support. To be effective, a prefetching strategy for multi-core processing environments has to be adaptive to choose among multiple methods to predict future data accesses. In this paper, hardware prefetching methods like Stream Buffer, Stride, NextLine (or Adjacent-Line) and Instruction Pointer as well as software based prefetching methods given by the complier and programmer are studied and analysed.

2 The Next Line Prefetching

The next line prefetching scheme also is known as one block look ahead. This scheme attempts to fetch the next sequential cache line before it is needed by

the processor.
It tries to prefetch sequential cache lines before they are needed by the CPU's fetch unit. The current cache line is the line that contains the instruction that is currently being fetched by the CPU. The next line is defined as the cache line which is located sequentially after the current line. If the next line is not found in the cache, then it will be prefetched when an instruction located some distance into the current line is accessed. This specified distance is called the fetch ahead distance and is measured from the end of the cache line.
These methods take advantage of the spatial locality in most programs, making use of the fact that instructions that are executed soon after each other are very often close to each other in memory.
The scheme requires little additional hardware since the next line address is easily computed.
Next-line prefetching is unlikely to reduce misses when execution proceeds down non-sequential execution paths caused by conditional branches, jumps, and subroutine calls. In these cases, the next line guess will be incorrect except in the case of short branches and the correct execution path will not be prefetched.
The performance of the scheme is dependent upon the choice of fetch ahead distance. If the fetch ahead distance is large, the prefetch is initiated early and the next line is likely to have been received from memory in time for the CPU to access it. However, increasing the fetch ahead distance increases the probability that a branch will be encountered in the current line after the prefetch begins, rendering the next-line prefetch ineffectual. This useless prefetch increases both memory traffic and cache pollution.
A simple extension to next line prefetching is called N-line prefetching. Here, the next n lines, rather than a single line, following the current line are brought into the cache. It has however been shown that the additional traffic and cache pollution means that this form of prefetching is infeasible for values of n >1.



3 Instruction Pointer

A pointer cache tracks pointer transitions to aid prefetching. A hardware prefetcher observes the memory accesses (sitting beside a particular cache level) and predicts the future accesses based on a specific patterns that are driven by signatures like instruction pointers (IPs), operating system (OS) page or a memory region.
Our goal is to propose an IP based prefetcher that can cover the majority of

access patterns. We find that each IP, can be classified into unique IP-classes, and the resulting classification could be used for better prefetching according to each class of IPs. We use multiple instruction pointer based prefetchers that suit different access patterns and overall cover a wide spectrum of access patterns. We classify IPs into three classes:

IP Constant Stride(CS): Consider the following access pattern (cache line aligned addresses) for an IP 0,2,4,6,8, with strides of 2,2,2,2. In this IP class, cache lines X, X+2, X+4 and so on are accessed by the same IP resulting in a constant stride pattern of 2. This is one of the more common patterns seen by IPs and can be prefetched using a simple IP-stride prefetcher. The prefetch coverage and prefetch accuracy for this pattern would be 100% if we classify this IP as IP for constant strides (IP-CS).

IP Complex Stride(CPLX): Consider this access pattern 0,3,6,10,13,16,20 with strides of 3,3,4,3,3,4. Here we see a complex pattern. A simple constant stride prefetcher would give 66% coverage. Also if the stride pattern is 1,2,1,2,1,2, a simple IP-stride prefetcher would lack the confidence to prefetch any stride since the two strides compete for the same entry in the stride table. Here, its coverage would be zero. These patterns are classified as complex strides. We create a signature of deltas seen by an IP and use it to index into a delta prediction table (DPT). Note that delta is nothing but a stride between two cache line aligned addresses.

IP Global Stream (GS): A global stream is a set of consecutive cache line aligned accesses observed independent of an IP, for example an access stream of 0,1,2,3,4,5,6,7,8,9. While this stream can be prefetched using using the CS or the CPLX class, looking at the global pattern we see that there is one IP that triggers a global stream and many IPs follow the global stream (makes the access stream independent of the IP). So, it makes more sense to prefetch the global stream as opposed to the IP-correlated stream since it preserves the global order of accesses and results in much better timeliness. So if an IP belongs to multiple classes, we prioritize the global stream class over others.

Our classifier, classifies instruction pointers at the L1 cache level and communicate the same to the L2 prefetcher. The prefetch requests issued into L2 and L1 are filled into last-level cache (LLC). We use the L1 prefetch requests to communicate the IP classification information to the L2 prefetcher. So, on a prefetch access to L2 (because of L1 prefetching), the L2 prefetcher issues prefetch requests according to the IP classification done at the L1. When a load misses in the L1 cache, but hits in the pointer cache, the first two cache blocks of the pointed to object are prefetched. In addition, the load's dependencies are broken by using the pointer cache hit as a value prediction.

Our prefetching framework provides 43.75% improvement for single-core and 22% for 25 selectively chosen multi-core mixes, respectively.

One of the advantage of using the IP based prefetchers is that it is less susceptible to IP reordering and random accesses across different memory regions, which may confuse a non IP prefetcher.

4 Stride Prefetching

The simplest way for prefetch hardware to identify a strided array referencing pattern also called stride prefetching is to explicitly declare when such a pattern occurs within the program and then pass this information on to the hardware. This is possible when programs are explicitly vectorized so that computation is described as a series of vector and matrix operations by the programmer, as is commonly done when programming machines which contain vector processors. Several techniques have been proposed which employ special logic to monitor the processor's address referencing pattern to detect constant stride array references originating from looping structures. This is accomplished by comparing successive addresses used by load or store instructions.

The scheme proposed by Chen and Baer is perhaps the most aggressive thus far. To illustrate its design, assume a memory instruction, mi, references addresses a1, a2 and a3 during three successive loop iterations. A prefetch for mi will be initiated if a1 - a2 = 0 where is now assumed to be the stride of a series of array accesses? The first prefetch address will then be A3 = a2 + where A3 is the predicted value of the observed address, a3. Prefetching continues in this way until the equality An = can no longer hold true.

The above approach requires the previous address used by a memory instruction to be stored along with the last detected stride if any.

Stride Prefetchers Chen and Baer investigate a mechanism for prefetching data references characterized by regular strides. Their scheme is based on a reference prediction table (RPT) and the look-ahead program counter (LPC). The RPT is a cache, tagged with the instruction address of load instructions. The entries in the RPT hold the previous address referenced by the corresponding load instruction, the offset of that address from the previous data address referenced by that instruction, and some flags. When a load instruction is executed that matches an entry in the RPT, the offset of the data address of that load from the previous data address stored in the RPT is calculated. When this matches the offset stored in the table, a prefetch is launched for the data address one offset ahead of the current data address. In [4], the reference address stream was used to index the reference prediction table, In practice, we found little performance difference between using the reference addresses or the miss address stream. Our later simulations of stride prefetchers use the miss address stream.

Advantages of this prefetching technique:-

To be effective, a prefetch mechanism must perform well for the most common types of memory referencing patterns. Scalar and unit-stride array references typically dominate in most applications and prefetching mechanisms should capture this type of access pattern. Sequential prefetching techniques concentrate exclusively on these access patterns. Although comparatively infrequent, large stride array referencing patterns can result in very poor cache utilization. RPT mechanisms sacrifice some scalar performance in order to cover stridden referencing patterns.

In the case of prefetch-on-miss, prefetching is suppressed when a large stride

is specified by the instruction. This avoids useless prefetches, which degraded the performance of the original policy. Although vector prefetching does issue prefetches for large stride referencing patterns, it is a more precise mechanism than other sequential schemes because it is able to take advantage of stride information provided by the program.

5 Stream Buffer

Stream buffer is one of the most common hardware based prefetching technique in use and was developed on the concept of "one block lookahead (OBL) scheme" .
The basic idea is that the cache miss address (k subsequent addresses) are fetched into a separate buffer of depth k. This buffer is called a stream buffer and is separate from cache.
Jouppi introduced Stream buffer as one of two significant methods to improved direct mapped cache performance. In contrast to stride prefetchers, stream buffers are designed to prefetch sequential streams of cache lines, independent of program context. The design presented by Jouppi is unable to detect streams containing non-unit strides. Palacharla and Kessler extended the stream buffer mechanism to also detect non-unit strides without having direct access to the program context. They also introduced a noise rejection scheme for improving the accuracy of stream buffers. Farkas further enhanced stream buffers by providing them with an associative lookup capability and a mechanism for detecting and eliminating the allocation of stream buffers to duplicate streams.

Stream buffers are used like a prefetch fill buffer for servicing L1 cache misses. Prefetches are placed in the stream buffer itself rather than a separate prefetch buffer. Stream buffers are allocated on L1 cache misses. If any stream buffer contains an entry that matches the current L1 miss reference address, it is taken from the stream buffer, the entries below the one removed are all shifted up to the head of the buffer, and prefetches are launched to sequentially consecutive cache line addresses to fill the vacancies that open up in the bottom part of the stream buffer. If there is no match in any stream buffer, a new stream buffer is allocated to the new stream.

In contrast to stride prefetchers, stream buffers are designed to prefetch sequential streams of cache lines, independent of program context. It can also detect non-unit strides without having direct access to the program context.

6 Software Prefecthing

Software-based prefetching is a technique for improving the performance of the cache memory subsystem to match modern high-performance processors. Software prefetching is done by either the compiler or programmer by adding lines to the code that prefetch data that is needed into the cache memory. Minimal hardware is required to take advantage of software prefetching. Experimental

results from studies show that software prefetching can significantly improve performance. In one study done by Mowry et al. (1992), the speedup in performance ranged from 5% to 100%, with 6 out of 13 benchmarks improving by over 45%.
Most microprocessors support 'fetch' instructions, which can be used to implement prefetching. Fetches are non-blocking memory operations. They are implemented in such a way as to not cause any exceptions, so as not to affect program correctness or increase overload. Prefetch scheduling is the task of choosing the placement of fetch relative to matching load or store instruction. They can be added either by the programmer or the compiler. Prefetching is mostly used within loops responsible for large array calculations. Loops and arrays are predictable and fetch instructions are placed inside loop bodies so that data for future loop iterations can be prefetched during the current operation. The loop-based fetch prefetching mechanism can be performed in three ways:-

1. Simple Prefetching
   Eg:-

```
1  for (int i=0;i<N;i++)
2 {
3 fetch( &a[i+1] );
4 s=s+a[i];
5 }
6
```

   Here, the data needed for the next iteration is prefetched into the cache. It improves the speed of array transversal.

2. Prefetching with loop unrolling
   Eg:

```
1  for (int i=0;i<N;i+=4)
2 {
3 fetch( &a[i+4] );
4 s=s+a[i];
5 s=s+a[i+1];
6 s=s+a[i+2];
7 s=s+a[i+3];
8 }
9
```

   This method improves on the simple prefetching method. Every prefetch bring one cache block (4 words) into the cache. So to improve performance and to remove unnecessary prefetches, prefetching is only done on every fourth iteration. Unrolling a loop involves replicating the loop body by the word size and then increasing loop increment alos by the same amount.

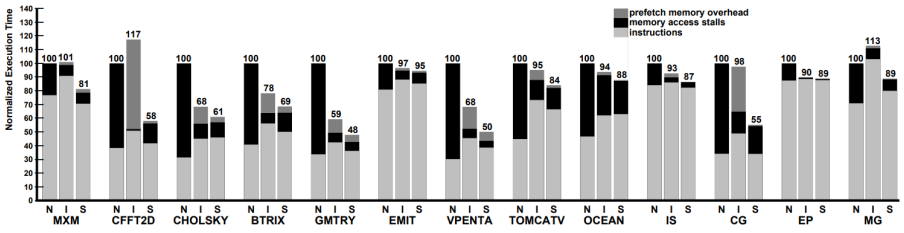
3. Software pipelining
   Unrolling loops improve performance, but cache misses will still occur

in the beginning, since prefetches are not issued initially. Unnecessary prefetches will take place in the end, where data beyond the loop boundary will be fetched. To rectify this, we can use software pipelining, where the loop is divided into prolog, main loop and epilog. In the prolog, initial elements of the loop are prefetched, in the main loop, both prefetching and computations take place, in the epilog only final computations are done. For maximum performance, following prefetches are needed:

$$\delta = \left\lceil \frac{l}{s} \right\rceil$$

where l is the average memory latency and s is the loop iteration time. The following is an example with l=100 and s=45 and word size 4

```
fetch( &s);
int i;
For (i=0;i<12;i+=4)                    //Prolog   - Initial Prefetching
{
fetch( &a[i]) ;
}
For (i=0;i<N-12;i+=4)                    //Main loop - Prefetching and Computation
{
fetch( &a[i+12]);
s=s+a[i];
s=s+a[i+1];
s=s+a[i+2];
s=s+a[i+3];
}
for(;i<N;i++)                  //Epilog   - Final Computation Only
{
s=s+a[i];
}
```



Overall performance comparison between the indiscriminate and selective prefetching algorithms (**N** = no prefetching, **I** = indiscriminate prefetching, and **S** = selective prefetching).

Limitations of this approach

1.Normally restricted to loops with array accesses
2.Hard for general applications with irregular access patterns
3.Fetch instructions increase process execution overhead
4.Code length is increased, which may degrade instruction cache performance.
5.It is done statically.

Non-binding prefetching - The above-discussed technique is an example of binding prefetching, in which the fetched data is bound or fixed to a named location such as a register.
In non-binding prefetching, data is brought to a cache close to the processor but is still available to the cache coherence protocol to keep it consistent. It has the advantage that there are no coherence issues since caches are coherent. A study on the Stanford Dash multiprocessor done by Mowry and Gupta (1991), showed that non-binding prefetching allows the fetching of data far in advance. In the three benchmark programs used in the study, the performance was enhanced by 86%, 83% and 14% in each of the programs.

References

[1]
Steven P. Vanderwiel and David J. Lilja. 2000. Data prefetch mechanisms. ACM Comput. Surv. 32, 2 (June 2000), 174–199. DOI:https://doi.org/10.1145/358923.358939
[2]
Todd Mowry, Anoop Gupta, Tolerating latency through software-controlled prefetching in shared-memory multiprocessors, Journal of Parallel and Distributed Computing, Volume 12, Issue 2, 1991, Pages 87-106, ISSN 0743-7315, https://doi.org/10.1016/0743-7315(91)90014-Z.
[3]
A Survey of prefetching techniques Nir Oren
July 18, 2000
[4]
Implementing and Testing Four Prefetching Cache Algorithms
Heba Alsultaan Mustafa Alkhafaji
heba.alsultaan@ucdenver.edu mustafa.alkhafaji@ucdenver.edu
Breawn Schoun Aljawharah Almuhana
breawn.schoun@ucdenver.edu aljawharah.almuhana@ucdenver.edu
[5]
Jim Pierce and Trevor Mudge. 1996. Wrong-path instruction prefetching. In Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture (MICRO 29). IEEE Computer Society, USA, 165–175.
[6]
Bouquet of Instruction Pointers: Instruction Pointer
Classifier based Hardware Prefetching
Samuel Pakalapati1 Intel Technology Pvt. Ltd. and BITS Pilani
Biswabandan Panda2 CSE Department, IIT Kanpur
[7]
J. Collins, S. Sair, B. Calder and D. M. Tullsen, "Pointer cache assisted prefetching," 35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings., Istanbul, Turkey, 2002, pp. 62-73. doi: 10.1109/MICRO.2002.1176239
[8]
Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov pre-

dictors. In Proceedings of the 24th annual international symposium on Computer architecture (ISCA '97). Association for Computing Machinery, New York, NY, USA, 252–263. DOI:https://doi.org/10.1145/264107.264207