The Power and Pitfalls of Data Prefetching

The terminology of data prefetching is introduced, which includes stride, repeat distance, stall, pending stall, prefetch degree, prefetch distance, and prefetch offset. The effectiveness of hardware data prefetching in reducing cache misses is shown by presenting a square matrix multiplication example. Thereafter the pitfalls of prefetching and possible solutions are discussed.

I. Introduction

The latency of memory accesses is an important issue in the design of computer systems because processors usually stall for the duration of memory requests. The speed disparity between processor and memory and the latency of the interconnection network contribute to this memory access latency.

The most fundamental approach to tackling this latency problem is to adopt a multi-level memory hierarchy, the goal of which is to bridge the speed gap between high speed processors and memories by introducing faster cache storage between processors and memories. The memory hierarchy works effectively because of the principle of locality [5]. Even though this approach works well, the processors must wait until the required data arrives if the data is not found in the nearest storage. Prefetching attempts to tackle this problem by bringing data close to the processor before it is actually needed.

Demand fetches are associated with processor idle time because they require data transfer between processor and memory while the processor stalls. On the other hand, prefetching which brings data near to the processor before it is actually used, can be done in parallel with computation, thus hiding the memory access time without stalling. Ideally, the memory access latency could be hidden completely if all data are brought into the cache before they are used. However, there are many pitfalls of prefetching which must be dealt with to achieve good performance.

Our goal in this paper is to investigate aspects of data prefetching by introducing the
terminology of data prefetching, and to show the effectiveness of prefetching in reducing cache misses. We also discuss the pitfalls of data prefetching and possible solutions.

The remainder of this paper is organized as follows. Section II introduces some terminology that is explained by presenting examples. Section III shows the effectiveness of prefetching by presenting an example. In Section IV, the pitfalls of prefetching are given. Finally, in Section V a summary is presented.

II. Terminology

In this section, we introduce the definitions of stride, repeat distance, prefetch degree, prefetch distance, prefetch offset, stall, and pending stall.

1. Stride and Repeat Distance

When we assume a load/store architecture processor where only load and store instructions access memory, there are four instruction classes: i.e. C for computation, L for load, S for store and J for jump. The code string \( (c_1, c_2, \cdots, c_s) \) represents the ordered instruction sequence of a compiled program. The subscripts indicate the order from the first instruction of the program. A \( c_i \) can be one of four instruction classes: i.e. \( c_i \in \{ C, L, S, J \} \). The code string is transformed into an execution string \( (e^1, e^2, \cdots, e^t) \) representing the ordered instruction sequence of program execution. The superscripts indicate the execution order from the start of execution of the given code string. An \( e^j \) can be written \( e^j_k \) because \( e \) should be one element of the given code string where \( i \in \{ 1, 2, \cdots, s \} \) and \( j \in \{ 1, 2, \cdots, t \} \): i.e. \( e^j_k \in \{ c_1, c_2, \cdots, c_s \} \).

Given a code string and its transformed execution string, assume that \( c_k \) is a load class instruction, say \( L_k \) and \( k \in \{ 1, 2, \cdots, s \} \), and it appears more than once in the execution string, say \( e_k^\alpha \), where \( 1 \leq r \leq n \) and \( \alpha \in \{ 1, 2, \cdots, t \} \). The \( e_k^\alpha \) should be \( L_k^\alpha \) and it loads its operand from memory address \( A_k^\alpha \).

We say that the instruction \( L_k \) has been revisited when \( n > 1 \) where \( n \) is the number of times an instruction is visited and is called the revisit number of the instruction \( L_k \) (or \( c_k \)). Two instructions \( e_k^\beta \) and \( e_k^\gamma \) are consecutive when \( \beta < \gamma \) and there is no \( e_k^\delta \) where \( \beta < \delta < \gamma \). When two of \( L_k^\alpha \) are consecutive loads, say \( L_k^\alpha \) and \( L_k^{\alpha+1} \), we define \( (A_k^{\alpha+1} - A_k^\alpha) \) as the stride. The stride represents the difference of address between two consecutive memory accesses by the same instruction. We define \( (\alpha_{q+1} - \alpha_q - 1) \) as repeat distance. The repeat distance represents how many instructions have been executed between consecutive visits to the same instruction. The smallest meaningful repeat distance is 1, in which case \( e^{\alpha_{q+1}} \) is \( e^{\alpha_q+2} \) \((=L^{\alpha_q+2})\) and \( e^{\alpha_q+1} \) is a conditional jump instruction. When repeat distance is 0, the only possible way is that the instruction is a jump instruction.

An access stream is a sequence of memory accesses generated by the same instruc-
tion. Thus, the load $L_k$ makes an access stream which consists of memory addresses $A_k^{\alpha_q}$ where $q$ sequences from 1 to $n$ and $\alpha_q \in \{1, 2, \ldots, t\}$. An access stream consists of one or more sub-streams. Consecutive sub-streams within a stream differ from each other in the magnitude of their stride. Each sub-stream consists of constant stride accesses, and therefore we use the terms sub-stream and constant stride access stream interchangeably. We say that the zero-stride sub-stream is a scalar access stream.

Let us consider some hypothetical code generated by a compiler, as shown in Table 1. Table 2 shows an execution string of Table 1. The LOAD instruction with address 804 has zero stride because the contents of r5 are not changed during k-th and (k+1)-th loops, see $e^{i+1}$ and $e^{i+7}$. This instruction makes a scalar access stream. The LOAD instruction with address 800 has stride 4 because the contents of r3 increase by 4, and its repeat distance is 5 because there are 5 instructions between $e^i$ and $e^{i+6}$. This instruction makes a constant stride access stream.

2. Prefetch Degree, Prefetch Distance, and Prefetch Offset

The prefetch distance is defined by the time period from the moment the prefetch request is issued to the moment the prefetched data is referred to by the processor. In this work, we use the number of processor cycles for time measurement.

<table>
<thead>
<tr>
<th>Code id.</th>
<th>Code addr.</th>
<th>Op-code operands</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{i-1}$</td>
<td>796</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>$c_i$</td>
<td>800</td>
<td>LOAD r3 r4</td>
<td>r4 $\leftarrow$ MEM(r3)</td>
</tr>
<tr>
<td>$c_{i+1}$</td>
<td>804</td>
<td>LOAD r5 r6</td>
<td>r6 $\leftarrow$ MEM(r5)</td>
</tr>
<tr>
<td>$c_{i+2}$</td>
<td>808</td>
<td>ADD r4 r6</td>
<td>r6 $\leftarrow$ r4 + r6</td>
</tr>
<tr>
<td>$c_{i+3}$</td>
<td>812</td>
<td>ADD 4 r3</td>
<td>Calc. new addr.</td>
</tr>
<tr>
<td>$c_{i+4}$</td>
<td>816</td>
<td>CMP r3 r2</td>
<td>Loop boundary</td>
</tr>
<tr>
<td>$c_{i+5}$</td>
<td>820</td>
<td>JNZ 800</td>
<td>Goto if r3&lt;r2</td>
</tr>
<tr>
<td>$c_{i+6}$</td>
<td>824</td>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code id.</th>
<th>Code addr.</th>
<th>Op-code operands</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^{i-1}$</td>
<td>820</td>
<td>LOAD r3 r4</td>
<td>k-th loop</td>
</tr>
<tr>
<td>$e^i$</td>
<td>800</td>
<td>LOAD r3 r4</td>
<td></td>
</tr>
<tr>
<td>$e^{i+1}$</td>
<td>804</td>
<td>LOAD r5 r6</td>
<td></td>
</tr>
<tr>
<td>$e^{i+2}$</td>
<td>808</td>
<td>ADD r4 r6</td>
<td></td>
</tr>
<tr>
<td>$e^{i+3}$</td>
<td>812</td>
<td>ADD 4 r3</td>
<td></td>
</tr>
<tr>
<td>$e^{i+4}$</td>
<td>816</td>
<td>CMP r3 r2</td>
<td></td>
</tr>
<tr>
<td>$e^{i+5}$</td>
<td>820</td>
<td>JNZ 800</td>
<td></td>
</tr>
<tr>
<td>$e^{i+6}$</td>
<td>800</td>
<td>LOAD r3 r4</td>
<td>(k+1)-th loop</td>
</tr>
<tr>
<td>$e^{i+7}$</td>
<td>804</td>
<td>LOAD r5 r6</td>
<td></td>
</tr>
<tr>
<td>$e^{i+8}$</td>
<td>808</td>
<td>ADD r4 r6</td>
<td></td>
</tr>
<tr>
<td>$e^{i+9}$</td>
<td>812</td>
<td>ADD 4 r3</td>
<td></td>
</tr>
<tr>
<td>$e^{i+10}$</td>
<td>816</td>
<td>CMP r3 r2</td>
<td></td>
</tr>
<tr>
<td>$e^{i+11}$</td>
<td>820</td>
<td>JNZ 800</td>
<td></td>
</tr>
<tr>
<td>$e^{i+12}$</td>
<td>800</td>
<td>LOAD r3 r4</td>
<td>(k+2)-th loop</td>
</tr>
</tbody>
</table>

The prefetch degree is defined by the number of data units that have been initiated to be prefetched at the time of prefetching. The data unit is usually a single cache line. The prefetch offset is defined by the address difference between the access address that prompts the prefetching and the prefetching address.
Let us assume that a processor accesses a memory location that falls into a cache line \( A_0 \), at a certain time \( t_0 \). This reference initiates prefetching of a set of data, that contains cache lines \( A_1, A_2, \cdots, A_n \) at \( t_1 \), and then these cache lines appear in the cache at \( t_2 \). Thereafter one of the prefetched data set, say \( A_2 \), is referred to by the processor at \( t_3 \). Figure 1 depicts the above case, in which the \( x \)-axis represents time and the \( y \)-axis shows the address space. In this case, the prefetch distance is \( t_{pd} \) (\( = t_3 - t_1 \)), the prefetch degree is \( D_p \) (\( = n \)), and the prefetch offset is \( S_p \) (\( = A_1 - A_0 \)).

For example, in the one-block-lookahead (OBL) prefetching method [14], the prefetch offset is one data unit (i.e. one cache line) and the prefetch degree is one data unit, since methods only consider the next cache line for prefetching when a cache line is referenced. Stride prefetching methods [3, 6, 13] change the prefetch offset according to the calculation of stride.

Gornish [7] used the term \textit{prefetch degree} to mean the number of cache blocks ahead of the currently accessed block, equivalent to our prefetch offset. He also used the term \textit{prefetch distance}, as we do to represent the amount of time between the prefetch of data and its use. Dahlgren, Dubois, and Stenstrom [4] used the term \textit{degree of prefetching} as the number of consecutive cache blocks to be prefetched on a cache miss, and it is the same as our use.

3. Stall and Pending Stall

If the processor model uses a stall on miss model, the \textit{stall time} is defined as the time spent for the data to arrive at the processor due to cache misses. The \textit{pending} stall is defined the stall when the processor waits for data that has already been requested by previously issued prefetches but has not yet arrived. In Fig. 1, if \( t_2 > t_3 \) (or \( t_{ma} > t_{pd} \)), the processor accessing \( A_2 \) at time \( t_3 \), waits \( A_2 \) until arrives at time \( t_2 \).
III. The Power of Prefetching

This section shows the effectiveness of hardware data prefetching by presenting an analysis of matrix multiplication.

We assume, for this example, that the matrix is organized in row-wise array, and each matrix element is a word size. In the row-wise array, elements $A[i][j]$ and $A[i][j+1]$ occupy consecutive memory locations as do $A[i][N-1]$ and $A[i+1][0]$, when we assume the array $A[N][N]$ is $N \times N$ array. We also assume that the cache is infinite size, each cache line contains $l$ words. We only consider global load requests. We use an $N \times N$ square matrix where $N$ is divisible by $l$, so that the remainder of $N/l$ is zero: i.e. $\lceil N/l \rceil$ is equal to $N/l$ where $\lceil x \rceil$ means the least integer greater than or equal to $x$.

```c
for (a=0; a<N; a++) {
    for (b=0; b<N; b++) {
        tmp = 0;
        for (t=0; t<N; t++) {
            tmp += A[a][t]*B[t][b];
        }
        C[a][b] = tmp;
    }
}
```

Fig. 2. Sample code for matrix multiplication: $C_{N\times N} = A_{N\times N} \times B_{N\times N}$.

1. Matrix Multiplication Example

The code in Fig. 2 illustrates the multiplication of an $N \times N$ matrix $A$ by an $N \times N$ matrix $B$: i.e. $C_{N\times N} = A_{N\times N} \times B_{N\times N}$. Each data structure of $A$, $B$ and $C$ occupies a continuous address space and is aligned to the cache line size. Code line 5 makes one global load to each of arrays $A$ and $B$, and these loads make access streams. Figure 3 shows the accesses to arrays $A$ and $B$. The access sequence to array $A$ refers to consecutive array elements, thus the access stream of access to array $A$ has stride $l$. The access sequence to the array $B$ refers to array elements which are $N$ words apart, and the access stream of the access to array $B$ consists of sub-streams each having stride $N$.

![Fig. 3. Array access sequences for matrix multiplication example shown in Fig. 2.](image)

2. Cache without Prefetching

We calculate how many cache misses occur due to accesses to arrays $A$ and $B$, when there is no prefetching support for the cache.

For accessing array $A$, there is a cache miss for the first and every $l$ accesses thereafter, because each cache line will supply $l$ accesses. Therefore there are $\lceil (N\times N)/l \rceil$ cache
misses which will be $N^2/l$. For accessing array B, there are $N$ cache misses for the first and every $l$ column access thereafter. Therefore there are $N[N/l]$ misses, i.e. $N^2/l$. In summary, there are $2N^2/l$ cache misses in total.

3. Cache with Sequential Prefetching

In this subsection, we use the typical one-block-lookahead prefetching methods [14]: prefetch on-miss and tagged prefetch. In the prefetch-on-miss scheme, each miss initiates a prefetching request for the next cache line. In the tagged prefetching scheme, a demand for a cache line which is the first access since its prefetch or miss, initiates a prefetching request for the next cache line. Therefore, the prefetch offset is one cache line, and the prefetch degree is one cache line.

A. Using Prefetch-on-Miss

The prefetch-on-miss needs a cache miss to initiate prefetching. For accessing array A, there is a cache miss for the first and every $2l$ accesses thereafter. Therefore there are $[(N \times N)/(2 \times l)]$ cache misses, and it will be $N^2/(2l)$. (When we assume $N$ is divisible by $l$, i.e. $N=kl$ where $k$ is integer, and $l$ is power of 2, i.e. $l=2^i$ where $i$ is larger than 0, $[(N \times N)/(2 \times l)]$ is equal to $N^2/(2l)$.)

For accessing array B, there are $N$ cache misses for the first and every $2l$ column access thereafter. Therefore there are $N[N/(2 \times l)]$ cache misses, and it will be $N^2/(2l)$. (As matter of fact, the difference between $[N/(2 \times l)]$ and $N/(2l)$ is 0 or 1 regardless of the size of $N$. In here, we just assume $N$ is an even multiple of $l$ for simplicity.) In summary, there are $N^2/l$ cache misses in total.

B. Using Tagged Prefetching

When accessing array A, there is only one cache miss which occurs when accessing the first element of the array A. There are $N$ cache misses for accessing array B occurring when the first column is accessed. In summary, there are $1+N$ cache misses in total.

4. Cache with Stride Prefetching

We use a typical stride prefetching method which issues a prefetching when two consecutive accesses, generated by the same instruction, have the same stride. This means that at least three accesses are needed to generate a prefetching.

For accessing array A, there is only one cache miss if the size of cache line $l$ is larger than or equal to 3 words, since the access stream sustains its stride from the beginning to the end. For accessing array B, there are three cache misses for the first and every $l$ column accesses thereafter, since each column access results in a stride $N$ sub-stream. There are $N$ sub-streams, and, however, each $l$ consecutive sub-stream will hit on the cache. Therefore, there are $3[N/l]$ cache misses, and it will be $3N/l$. In summary, there are $1+3N/l$ cache misses in total.
5. Comparison

Table 3 summarizes the cache misses of the three hardware data prefetching schemes discussed above. It is quite clear that the prefetch-on-miss method cannot eliminate more than half of the cache misses incurred by no-prefetching. The tagged prefetching method works extremely well and the stride prefetching method even better if the cache line size is larger than or equal to three words.

(Table 3) Comparison of cache misses for $N \times N$ matrix multiplication ($l$ is cache line size in words).

<table>
<thead>
<tr>
<th></th>
<th>No-prefetching</th>
<th>Prefetch-on-miss</th>
<th>Tagged prefetching</th>
<th>Stride prefetching</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2N^2/l$</td>
<td>$N^2/l$</td>
<td>$1+N$</td>
<td>$1+3N/l$</td>
<td></td>
</tr>
</tbody>
</table>

In this section, we did not consider software directed data prefetching in which the compiler or programmer directs prefetching by inserting prefetching instructions into the code [2, 8, 11]. Software prefetching needs the non-blocking load or prefetching load instruction, and special hardware to treat this instruction. Software prefetching increases compilation time, run-time overhead and code size because additional instructions are needed to calculate the prefetching address and issue the prefetching loads.

Another solution which combines software and hardware prefetching methods, is integrated or hybrid data prefetching. For example, if software causes the prefetching of the first cache line that contains A[0][0] and cache lines that contains the first column data B[][0], then tagged prefetching will not experience any cache misses.

IV. Pitfalls of Prefetching

We have confirmed that prefetching can reduce cache misses. However, the decrease of cache misses does not always result in reduction of memory access stall times. This section explains the pitfalls of prefetching.

1. Small Repeat Distance

We have seen that the number of misses is reduced by using prefetching, see Table 3. Unfortunately we cannot say that the reduction of the number of cache misses always results in hiding memory access latency. When a request from the processor hits on the cache block that has already been requested by a previously issued prefetching request which has not yet finished, a cache hit occurs but the processor has to wait for the data. In this case, the prefetch distance is not enough to hide the memory access latency completely.

If we look at an example, this will become clear. Let us assume the matrix multiplication example(Fig. 2) with the tagged prefetching scheme. At the first access of A[0][0] (see the line numbered 5 of Fig. 2), it should miss, and as a result the cache line that contains A[0][0] is brought into the cache. The request for A[0][1] by the next for-loop with t variable (the lines numbered from 4 to 6) will hit if
the cache line size is larger than 1. As a result of the second access on the same cache line, the prefetch for the next cache line is initiated. While the prefetching request is progressing, a request for A[0][t] by the iteration of the for-loop with t variable eventually reads the next cache line, if the repeat distance of the for-loop is small.

Fortunately, the case described above could be a rare one, because a cache line can satisfy multiple loop iterations, and during an iteration of loop, another load or store requests might be issued and these requests could take a long time. However, this short repeat distance is still a potential problem. When the repeat distance is small, the prefetch distance should be enlarged by issuing prefetching requests more than one loop iteration ahead.

2. Finite Cache Size

The figures in Table 3 assume an infinite size cache or a cache sufficiently large to contain the whole data. In this subsection we consider the effects of data prefetching with a finite size cache.

There is a data set, $D_T$, that occupies a continuous address space and is aligned to the cache line size. When a processor with an infinite size cache accesses the data set $D_T$, this processor experiences at most $\lceil \text{size}(D_T)/l \rceil$ cache misses, because the $D_T$ occupies $\lceil \text{size}(D_T)/l \rceil$ cache lines, where $l$ is cache line size. However, if the cache is not sufficiently large enough to hold the data set $D_T$, extra cache misses can occur because some cache lines that are brought in previously, can be replaced by newly arriving cache lines.

Prefetching can cause more replacement by bringing data prior to actual use. To avoid this cache pollution problem due to size limit, a prefetching buffer can be used. The prefetching buffer keeps the prefetched data instead of placing it into the cache. However, the prefetching buffer is another memory hierarchy and it makes data coherency protocol difficult.

Prefetching aims to initiate a request prior to the actual usage, and, therefore, it makes worse the replacement problem. Under a certain circumstances, prefetching with a finite cache can cause more cache misses. However, apart from the sharing effect described in the next subsection, we certainly can say that prefetching with an infinite size cache always reduces cache misses, and more prefetching will reduce cache misses even more.

3. Data Sharing

Shared-memory multiprocessor systems, where each processor has a private cache hierarchy, have a data coherence problem due to the existence of several copies of shared data [1]. This data sharing causes coherence misses that occur as a result of invalidations made to preserve cache coherency. Data sharing allows cache misses to be divided further
into true sharing and false sharing. False sharing is the sharing of a coherence block without actual sharing of data item whereas true sharing is due to genuine sharing of the data item [9].

Prefetching can cause additional true and false sharing by bringing shared data into cache. When a processor prefetches a cache line, this cache line is going to be in a shared state if it is also in another processor’s cache. Thereafter, before the prefetched data is used, the other processor might want to use the data exclusively, and the prefetching turns out useless and, even worse, causes extra invalidation traffic.

To overcome the sharing effect of prefetching, we can use a binding prefetching method instead of a non-binding one. With a binding prefetch, the value obtained by a later reference is that obtained at the time of the prefetch. On the other hand, a non-binding prefetch [12] brings the data close to the processor, but the data remains visible to the memory coherency activity and it is kept consistent until the processor actually uses the value. The former approach has severe limitation of how far prefetches can be made before their actual use because it must make it sure that there is no update during the interval between prefetching and reference.

4. Network Traffic

Prefetching tends to increase the number of memory requests, and prefetching of shared data can cause additional coherence traffic. The additional memory requests come from the prefetching of unnecessary data and from early replacement. The additional coherence traffic results from the effect of true or false sharing. These additional activities increase the load on both the network and the memory and can increase memory access latency. Therefore prefetching without sufficient network bandwidth could degrade performance [15].

5. Cache Busy

When prefetching places data into the cache, the prefetching unit needs to access the cache tag memory to ensure that it does not issue a prefetch for data that already exists in the cache. If the cache cannot handle multiple requests, the cache could be out of service for normal cache references from the processor while prefetching is progressing.

To solve the cache tag busy problem, we can use a prefetch buffer. When the prefetch unit initiates a prefetching request, it does not check the existence of the data, and places the prefetched data into the prefetch buffer. The processor looks at the prefetch buffer before it initiates a memory request when a cache miss occurs. This approach could possibly increase network traffic and memory requests because some prefetching requests fetch data already residing in the cache, and it makes data coherence protocol difficult.

Lockup-free cache [10] organization is a widely accepted technique. The lockup-free
cache can handle multiple cache requests so that the prefetching unit can access the cache without interfering with the normal cache accesses, and it can serve cache requests while there are outstanding requests to the memory.

In spite of these limitations described above, data prefetching is still an attractive approach to tackle the memory latency problem.

References


