Streaming Prefetch *

Olivier Temam

PRiSM, Versailles University, 45 Av. des Etats-Unis, 78000 Versailles, France

Abstract. In most commercial processors, data prefetching has been disregarded as a potentially effective solution to hide cache misses because it may degrade global cache performance. The two main limitations are wrong address predictions and prefetch overhead. In this paper, a hardware/software scheme to limit wrong predictions and a proper hardware support to prevent prefetch requests from disrupting normal cache operations is proposed.

1 Introduction

While instruction prefetching has already been successfully implemented in numerous commercial processors (Dec 21164, MIPS T5...), data prefetching is scarcely used for the moment. Still, several processors do include hardware support for data prefetching. The DEC Alpha has a one-line prefetch buffer to implement tagged data prefetching, but to our knowledge, this prefetching facility is disabled on current Alpha versions. The SuperSparc also has a similar prefetch buffer. The PowerPC has a *touch* instruction which induces a load to a null register, however the PowerPC compiler does not seem to exploit this instruction yet. Clearly, data prefetching has not yet emerged as a widespread commercial optimization.

The problem is to understand why the approaches to prefetching that have been proposed or implemented up to now are not satisfactory. There are basically two aspects to distinguish: *accuracy of prediction* and *hardware support for prefetching*.

While very cheap, systematic prefetching and tagged prefetching [9] have the obvious flaw of generating many useless prefetches which induce cache pollution and additional memory traffic. Stream buffers [7] are more efficient, but they lose efficiency when the number of simultaneous streams is higher than the number of stream buffers or when stream strides are large. Now, several schemes based on prediction tables (one table entry per load/store instruction) have been proposed where the stride of a load/store reference is automatically computed [6, 5]. Such schemes exhibit high prefetch efficiency (accuracy of prediction) but their hardware cost is very significant since the table must be about 256-entry large [5]. Software prefetching [1] exploits the subscript expression for address predictions by prefetching `A(J+d,I)` for reference `A(J,I)` in a `I,J` loop nest (where `d` is the prefetch distance). The prediction is based on the inner loop index. In addition to the significant compiler overhead of software prefetching,

* Due to severe paper length constraints, the length of this article nearly had to be halved. Thus many explanations are replaced by references to PRiSM technical report 95/029 which can be obtained at `http://www.prism.uvsq.fr` in the `technical reports` section.

hardware support is still necessary (prefetch on miss, prefetch buffers...) and the numerous additional instructions corresponding to prefetch requests and the associated address computations can have a non-negligible impact on instruction cache performance. Mowry [10] proposed a software prefetching algorithm based on a data locality algorithm to determine which array references are likely to miss and thus reduce the number of useless prefetch requests often associated with software prefetching. Though the algorithm proved to be efficient, the compiler support is quite heavy and the fact all locality optimizations interact together is beneficial but also implies compiler optimizations may be hard to maintain when the memory architecture evolves or is augmented with new features.

It can be noted that existing prefetching schemes have similarities: they exploit strided streams of references, but *they are all disrupted when the stream of references changes stride.* However, in numerical codes, many linear algebra operations are based on multi-dimensional objects. It is usually admitted that Fortran storage of such objects ensures they correspond to contiguous memory elements. But many linear algebra operations deal with sub-matrices or domain borders which correspond to non-contiguous sets of memory locations. Consequently, an array reference stream in a numerical loop nest is likely to exhibit periodic stride modifications. This aspect of numerical codes is usually ignored.

With respect to *address prediction accuracy*, the puzzling fact is that, upon entering a loop nest, the coefficients of an array subscript are usually known and generally won't vary during loop execution. Consequently, *the reference pattern of most load/store instructions in a loop nest can be anticipated* and need not be *predicted.* The purpose of this paper is to exploit this property and also to show that taking into account more complex streams can induce significant performance improvements, by nearly eliminating wrong predictions and predicting stride changes in linear references. A software-assisted prefetching scheme, called *streaming prefetch*, that deals with complex streams is proposed. The compiler support sums up to providing the symbolic expressions of loop index coefficients in the linearized expression of array subscripts. Additional hardware support is necessary but no costly prediction table is required.

With respect to the appropriate *hardware support* for implementing *transparent* prefetching, i.e., without disrupting cache, several issues must be addressed. First, the necessity to check the cache prior to prefetch (*prefetch on miss*, see [8]) so as to maintain coherence and to reduce the additional memory traffic. These cache accesses are likely to induce processor stalls, especially with superscalar processors where the cache can be accessed every cycle. Second, reloading prefetch requests in cache also induce processor stalls. If a prefetch buffer is used, each buffer access may result in an additional cycle if simultaneous cache and buffer checks are not possible because of cache access time constraints. Even with a buffer, copying a cache line from the buffer to the cache would usually result in cache stalls. All these issues can strongly disrupt the efficiency of a prefetching scheme. In this paper, a hardware support for implementing prefetching with a very low impact on cache performance is proposed, thus eliminating the main hazard of prefetching: possible performance degradations. It is shown that when prefetching exhibits a high issue rate like *streaming prefetch*, implementation issues are critical.

In section 2 the prefetch scheme proposed and its implementation are explained. In section 3, the experimental framework is briefly presented. Finally, in section 4, the performance of the scheme is evaluated.

2 Streaming Prefetch

The different components of *Streaming Prefetch* are described in the next sections.

2.1 Using Subscripts to Drive Prefetching

In the table prediction schemes proposed in [6, 5], one table entry is created for each load/store instruction found. Because some numerical loop nests are complex and because loop unrolling is a now a common optimization, the number of load/store instructions in a loop body can be very large. As a consequence, the optimal size of such tables is about 256 entries [6, 5]. In each entry, two data must be stored: the last data address referenced and the last stride (difference between the last address and the address before), plus a number of flag bits. Consequently, the table size makes it a costly piece of hardware.

Now, a simple analysis of the strides found in numerical loop nests shows that most of these strides are identical. We have counted the number of distinct stride sequences per loop nest for the loop nests of the 7 benchmarks considered. A stride sequence here corresponds to the set of coefficients in the linearized form of an array reference subscript. For instance, consider array reference `A(i,j)` where the array declaration is `A(N,N)`, the linearized expression of this reference is `A(i+N*(j-1))`, so that the stride sequence is `1,N`, 1 for the inner loop, `N` for the outer loop. We have found that there are usually few distinct stride sequences in a given loop nest (see [12]). Therefore, in a prediction table, the stride in many entries is likely to be the same, uselessly duplicating informations.

So, a first improvement is to have a much smaller table, called a *stride table*, to store only the distinct stride sequences found in a loop nest. If the table size is S, each load/store is then fitted with $log_2(S)$ bits to indicate the table entry corresponding to its stride sequence. This is where the compiler is used to assist the prefetch mechanism. The compiler computes the linearized subscript expression of all array references, and determines the number of distinct stride sequences. Stride sequences are then numbered and one such number is given to each array reference which corresponds to the tag bits mentioned above. The last role of the compiler is to insert special instructions right before the beginning of the loop nest to reload the stride table with the appropriate stride values. Since stride reload is done at run-time, prior to loop nest execution, the stride value is usually known at that time. For reloading the *stride table*, a specific instruction is necessary: `UpdateStrideTable <TableEntryNumber> <CoefficientDepth> <StrideValue>`. Statistics show a stride table size of 8 is sufficient [12].

Strides are not equal to coefficients

The issue now is to determine which value to use as strides. Indeed, the loop index coefficients of a linearized subscript usually do not correspond exactly to the strides. Consider the example loop below.

The inner stride of references `B(I3,I2,I1)`, `D(I1,I2,I3)` are respectively 1 and N*N words. The linearized subscripts corresponding to reference `B(I3,I2,I1)` is `B(I3+N*(I2-1)+N*N*(I1-1))`. Since the loop bounds do not necessarily correspond to 1 and N, the stride of the reference can change at the end of each execution of loop `I3` (this stride can be called the `I3` stride). For given values of `I1,I2`, loop `I3` terminates at element `B(U3+N*(I2-1)+N*N*(I1-1))`, and the

```
C— Array Declarations
REAL A(N),B(N,N),D(N,N),E(N),F(N)

C— Loop Nest
DO I1=L1,U1,S1
  C = A(I1)
  DO I2=L2,U2,S2
    F(I2) = F(I2) + 1
    DO I3=L3,U3,S3
     C = C + B(I3,I2,I1) + D(I1,I2,I3)
    ENDDO
  ENDDO
  E(I1) = C
ENDDO
```

Table Entry No.	Depth=0	Depth=1	Depth=2
1 (references A(I1), E(I1))	0	0	1
2 (reference F(I2))	0	1	0
3 (reference B(I3,I2,I1))	1	N	N*N
4 (reference D(I1,I2,I3))	N*N	N	1

Fig. 1. *Example Loop and Corresponding Stride Table.*

next loop I3 begins at B(L3+N*I2+N*N*(I1-1)) (assuming loop I2 did not terminate also). So the I2 stride is equal to (L3-U3)+N, where N is the coefficient of loop index I2 in the linearized expression of the subscript. If loop I2 terminates when loop I3 terminates, the last reference is B(U3+N*(U2-1)+N*N*(I1-1)) and the next reference is B(L3+N*(L2-1)+N*N*I1), so the I1 stride is equal to (L3-U3)+N*(L2-U2)+N*N.

More generally, for an array subscript A(cn*In+...+c2*I2+c1*I1), the stride used when all loops $n \ldots p$ are about to complete is equal to cn*(Ln-Un)+...+cp+1*(Lp+1-Up+1)+cp. Therefore, in order to derive the stride values, it is necessary to get the value of the Lk-Lk expressions. This issue is addressed in section 2.2

In addition to stride redundancy, a second flaw of prediction tables is their sensitivity to stride disruption. Whenether the loop nest depth is higher than 1, stride disruption occurs every time the inner loop has completed if the inner loop bound does not correspond to the first array dimension. Each time a stride disruption occurs, not only the next address is wrongly predicted (possibly polluting the cache), but stride stability must be reached before prefetching again (requiring 2 [5] or 3 [6] references depending on the scheme). While the impact of this flaw is negligible if the number of iterations in the inner loop is high, it can become significant otherwise.

Because in *streaming prefetch* the stride is provided by the compiler, not only there is no significant *cold-start* effect due to stride stability, but the target address of the stride disruption can be computed and prefetched. Consequently, except for conflict misses, theoretically only *one miss* should occur (for the first reference) when this scheme is applied to a linear subscript. Actually, the necessity to prefetch at a distance higher than one iteration (see section 2.3) induces a few more misses (the number of misses is directly related to the prefetch distance).

2.2 Stream Length

Most prefetching schemes are restricted to inner loops because they can only exploit single-stride streams. As soon as streams change strides, which can happen every time the inner loop ends, prefetching will either stop or breed useless memory requets. We have measured inner stream lengths for all benchmarks and found a majority are fairly small, i.e., about 20 iterations, even though high peaks can be observed [12]. Thus, current prefetching schemes performance, though already high, is bounded by multi-stride linear reference patterns. In the previous section, it has been shown that strides are completely determined by the subscripts index coefficients and the stream lengths Lk-Uk.

The expression Lk-Uk, mentioned in the previous section, does not correspond to the difference between the current lower and upper bounds, but to the

difference between the next lower bound and the current upper bound. Because it may not be trivial to obtain the next lower bound, we based the mechanism on the assumption that the value `L-u` seldom changes, or that its stride `dLu` is constant. This means we assumed a stream length is either constant or varies in a strided manner. This hypothesis proved to be true in a large majority of cases. In the case where the loop bounds are multilinear (i.e., they depend on several outer loop indices) or even more complex expressions, the scheme proposed below will not behave properly. However, multilinear loop bounds are much less frequent than multilinear subscripts.

The issue now is to provide the necessary hardware/software support to compute the stream length on-the-fly so as to get the `Lk-Uk` value. Actually, the stream length can be obtained by extending a piece of hardware now found in many processors: the branch prediction table (Dec 21164, PowerPC620...).

Indeed, each time a loop iteration completes, a branch is taken. The number of times the branch is taken corresponds to the number of iterations of the loop associated with this branch.[2] Therefore, the principle is to add length counters to each entry of the prediction table. To implement length prediction the following data must be stored: the current stream length, and the last stream length which corresponds to the current stream length until the branch was not taken. Also the last stream length information must be duplicated in the *stride table* for each depth (since it will then be used to compute the strides).

Adding several new data per branch table entry is a costly solution. It can be noted that only the branches in the loops active at a given time need have these counters. Therefore, instead of fitting each branch prediction table entry with two length data, a small additional table can be used. Each entry in this table contains the two length data, and the table is indexed by the branch PC, just like the branch prediction table. The cost of this solution can be further reduced by having the compiler indicate which branch corresponds to a loop branch. Considering the benefit of multi-stride prefetching beyond the 4th loop level is often negligible, a 4-entry table can then be used.

For triangular loops where bounds are non-constant but vary with a constant stride, the scheme has been augmented with a *length stride field* in addition to the *current length* and *last length fields.*

An overview of *Streaming Prefetch* with the example of figure 1 can be found in [12].

2.3 Hardware Implementation Issues

As mentioned in section 1, a prefetching scheme can behave poorly or even degrade global processor performance if several implementation issues are not addressed. First, to limit the number of useless prefetch requests and to maintain cache coherence, it is necessary to check whether an address is in cache before issuing a prefetch. Such tests can result in numerous cache stalls and consequently processor stalls. Second reloading incoming prefetch requests is another major source of cache stalls. In most commercial implementations of data prefetching (Sun Supersparc, Dec 21164), cache stalls due to prefetch reloads have been avoided by using a prefetch buffer. Incoming prefetch requests are stored in the buffer and are then transferred to cache when the processor hits in the buffer.

[2] Actually, to the number of iterations minus 1 since, on the last iteration, the branch is not taken.

The main flaw of this solution is that a hit in the buffer can cost one more cycle than a cache hit, especially if cache access time is critical and the buffer cannot be tested simultaneously. These issues have been barely addressed up to now, except for Drach [3] where a modification of the instruction pipeline is proposed to partially hide these additional cache accesses.

In this implementation, we have addressed both issues simultaneously. The basic idea is to separate the cache into several independent banks which will be called *subbanks*. A similar technique was proposed on the Dec 21164 to allow multiple cache accesses per cycle, and we also introduced it for a different purpose in [11]. These subbanks should not be confused with the banks used for set-associativity. They actually correspond to a division of such banks. Consider a $w - way$ associative cache where each bank is partitioned in b subbanks. If the lines of this cache are numbered from 0 to $N-1$, two lines, which indices are L_1 and L_2, belong to the same bank if $L_1 \equiv L_2$ *modulo* N/w. Now, L_1 and L_2 belong to the same subbank if they belong to the same bank and if $L_1 \equiv L_2$ *modulo* b.

Subbank partition allows multiple accesses per cycle, so that cache checks can be done in parallel with normal cache requests without ever stalling the cache. Still, since a processor request has always priority over a prefetch request, a small 2-entry buffer must be added to each subbank to buffer pending cache check prefetch requests. Furthermore, we use this subbank technique to reload incoming prefetch requests without stalling the cache. A prefetch buffer is still necessary but its size (i.e., its cost) needs not be large (2 lines) since prefetch requests are systematically reloaded in cache. In addition to prefetch buffer size reduction, the main asset of this technique is that *hits on prefetched lines do not cost additional cycles.*

To compensate for memory latency (20 cycles in our case), a prefetch distance of $\delta = 8$ iterations was used for inner loop and 1 iteration for outer loops. The cache check step can become a bottleneck if prefetch requests are numerous. Actually if prefetch requests are filtered according to the following criteria most useless requests can be removed: (1) If the stride is larger than the line size, the prefetch request is issued whatsoever. (2) If the stride is smaller than the line size, the request is issued only if the target address modulo the stride is smaller than the stride. (3) If the stride is null no prefetch request issued. Further explanations on these criteria can be found in [12].

3 Experimental Framework

Seven benchmarks from the Perfect Club Suite [2] were used. For each benchmark, a 50 million-entry trace was extracted. Additional code instrumentation like instructions to table reload instructions took the form of calls to specific routines that are detected at tracing time. All source-code instrumentation was inserted with the Sage++ [4] compiler. More details on the methodology can be found in [12].

4 Performance Evaluation

For all experiments we have used a 32-Kbyte cache, 4-way associative with a 64-byte line; the cache is pipelined with a 2-cycle access time. Such cache parameters are fairly standard for current or soon to appear processors. The miss latency is equal to 20 cycles. In Figure 2a, the miss ratio of all codes with a standard cache and a cache plus *streaming prefetch* is shown. Though original miss ratios are low, *streaming prefetch* removes misses in most codes and brings

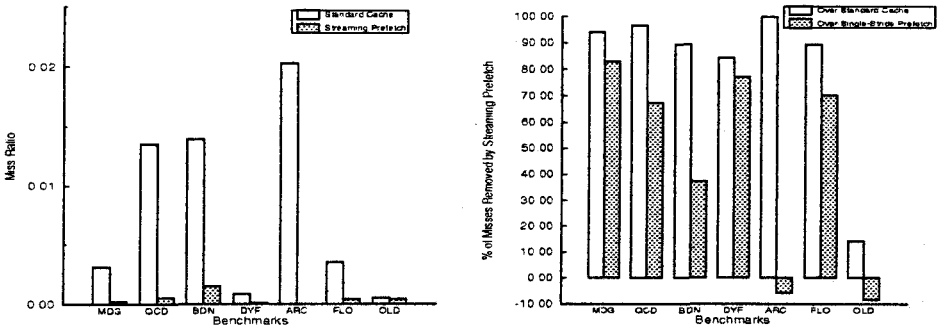

Fig. 2. *(a) Miss Ratio. (b) Fraction of Misses Removed.*

all miss ratios down to 0.15% or less. Figure 2b shows the fraction of misses removed over a standard cache and over single-stride prefetching. Single-stride prefetching is similar to prediction tables schemes: only one stride is considered and stride modifications induce wrong predictions. The high percentage of misses removed over this latter scheme is artificially increased by the fact the benchmarks already exhibit very few misses. Still, the improvement appears to be significant, meaning that predicting stride changes is worth the additional hardware support. The improvement is due both to the better predictions and to the avoided cache pollution. For two codes, *streaming prefetch* performs worse than single-stride prefetch. By analyzing which loops breed this difference, we found out that sometimes wrong predictions induce prefetching of data used in next loops, accidentally avoiding misses.

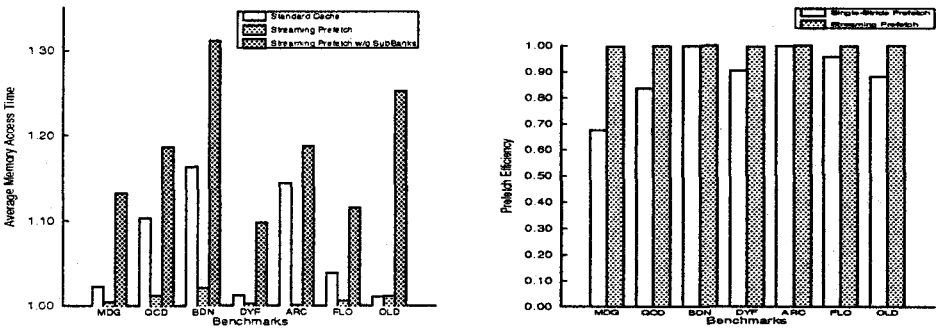

Fig. 3. *(a) Average Memory Access Time. (b) Prefetch Efficiency.*

The second aspect of *streaming prefetch*, the efficiency of the dedicated hardware support used to minimize cache perturbations, is shown in Figure 3(a). First, it can be seen that, as for the miss ratio, *streaming prefetch* brings the average memory access time close to the 1-cycle optimum. Also, to test the efficiency of the additional hardware support, we implemented *streaming prefetch* without any of the subbank support. In all cases, *streaming prefetch* performs worse than the standard cache.

The last important aspect of a prefetching scheme is its impact on memory traffic. We measured the number of excessive memory requests through the prefetch efficiency, i.e., the ratio of the number of hits on a prefetched line over the number of lines prefetched. This ratio is shown in Figure 3(b). Since the

efficiency of *stream prefetching* is close to 1, nearly no additional memory traffic is induced. Also, as expected, single-stride prefetch is less efficient because of the wrongly predicted stride changes mentioned throughout this paper.

5 Conclusions

In this paper, we wanted to show that it is possible to implement prefetching in such a way that the reference streams usually found in numerical loop nests can be accessed without disruption. This required not only to improve address prediction over classic single-strided schemes, but also to implement prefetching so that normal cache operations are not affected and memory traffic is not increased. *Streaming prefetch* proved to be efficient both at hiding misses and reducing the average memory access time accordingly (meaning it has no negative impact on cache). Finally, this scheme showed that significant performance improvements can be obtained if the hardware is tailored to exploit compiler analysis.

References

1. David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.
2. George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputing Performance Evaluation and the Perfect Benchmarks. In *Supercomputing '90*, pages 254–266, 1990.
3. N. Drach. Hardware Implementation Issues of Data Prefetching. In *ACM International Conference on Supercomputing'95, Barcelona, Spain*, July 1995.
4. D. Gannon et al. SIGMA II: A Tool Kit for Building Parallelizing Compiler and Performance Analysis Systems. Technical report, University of Indiana, 1992.
5. John W. C. Fu, Janak H. Patel, and B. L. Janssens. Stride Directed Prefetching in Scalar Processors. In *MICRO-26*, 1992.
6. T-F. Chen J-L. Baer. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proceedings of IEEE Supercomputing*, 1991.
7. Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small, Fully-Associative Cache and Prefetch Buffers. In *International Symposium on Computer Architecture*, pages 364–373, May 1990.
8. Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. In *International Symposium on Computer Architecture*, pages 43–53, May 1991.
9. Alan Smith. Cache Memories. *Computing Surveys*, 14(3), September 1982.
10. M. Lam T. Mowry and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, September 1992.
11. O. Temam and N. Drach. Software Assistance for Data Caches. In *1st Symposium on High Performance Computer Architectures*, January 1995.
12. Olivier Temam. Streaming Prefetch. Technical Report 95/029, PRiSM, Versailles University, 1995.