Streaming Prefetch *

O. Temam
Versailles University
45 Av. des Etats-Unis
78000 Versailles
France
Phone: 33-1-39-25-43-41
Fax: 33-1-39-25-40-57
Email: temam@prism.uvsq.fr

Abstract

In most commercial processors, data prefetching has been disregarded as a potentially effective solution to hide cache misses, multi-level caches being widely preferred. However, multi-level caches are mostly effective at removing capacity and conflict misses, while prefetching is particularly efficient for removing compulsory misses, especially in the regular accesses found in numerical codes. One of the main flaws of prefetching which stronly limits its popularity in current processors is that it can potentially degrade global cache performance. Wrong address predictions is the first cause of cache pollution as well as additional memory requests. All existing prefetch schemes are impaired by wrong predictions because they *speculate* on the next address to be referenced. In this paper, we show that all required informations to avoid the speculative aspect of prefetching can be easily obtained from the compiler, resulting in nearly no wrong predictions. Even when address prediction is flawless, prefetching can be hazardous to cache because cache checks (required before sending a prefetch request to limit memory traffic) and cache reloads of incoming prefetch requests can result in cache stalls and thus processor stalls, particularly in superscalar processors where the cache can be accessed every cycle. In this paper, we show that addressing these implementation issues can make a prefetching scheme nearly transparent to normal cache operations. We have combined software-assisted address prediction with dedicated hardware support and obtained a prefetching scheme called *streaming prefetch* where data can flow through the cache nearly without disruption.

Keywords: prefetching, memory architecture co-design, cache, software-assisted caches, data locality, numerical codes.

Workshop: W20, Instruction-Level Parallelism.

Part of this work was done while the author was at *IRISA, Rennes, France* and *University of Leiden, Leiden, The Netherlands.*

*This work was supported by the Esprit Agency DG XIII under Grant No. APPARC 6634 BRA III.

1 Introduction

Aside from reducing the number of cache misses through code and cache optimizations, there are two major approaches to reducing the average miss penalty: hiding memory latency with multi-level caches (also with non-blocking caches for small latencies) and hiding cache misses with prefetching.

For the moment, multi-level caches seem the most popular means for reducing the impact of high memory latencies. Indeed, multi-level caches are relatively straightforward to implement and they can potentially profit to any kind of code. On the other hand, they do not come at a cheap cost, as shown by the three-level hierarchy of the DEC 21164 (with a 96Ko on-chip secondary cache) or the 256Ko secondary cache of the Intel P6 (implemented on the same module). A more surprising aspect of this exclusive success is that multi-level caches are mostly efficient at hiding *capacity and conflict* misses, not *compulsory* misses. Actually, the larger lines used in outer caches does help reducing the average miss penalty of compulsory misses, but the necessity to keep conflicts low in these secondary caches, which are usually direct-mapped, limits their line size. Therefore, for codes with large working sets, i.e., numerous compulsory misses, and regular accesses such as numerical codes, prefetching appears as a good complement.

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 [12] have the obvious flaw of generating many useless prefetches which induce cache pollution and additional memory traffic. Stream buffers [9] 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 [7, 5, 8]. 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, 8]. 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, 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 [13] 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. However, 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. Consider for instance the following loop nest:

```
REAL A(N,N),X(N),Y(N)

DO I=1,P
  reg = Y(I)
  DO J=1,P
    reg = reg + A(J,I)*X(J)
  ENDDO
  Y(I) = reg
ENDDO
```

If $P = N$, the addresses referenced by `A(J,I)` are all contiguous. On the other hand, if $P < N$, the stride of the reference stream corresponding to `A(J,I)` changes at the end of every execution of loop `J`.

This paper is based on the observation that no present or proposed prefetching scheme can maintain a *streaming flow of prefetch requests* in numerical loop nests. There are two major reasons for this. First, the address prediction accuracy is limited to single-stride streams, and wrong predictions can degrade global cache performance (increased memory traffic, cache pollution). Second, the issue of implementing a streaming flow of requests through the cache without degrading cache performance has been barely addressed up to now.

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. It can be noted that software prefetching is also based on subscript expressions, but predicting stride changes with software prefetching would require to add a guard around prefetch instructions to detect loop end and prefetch at `A(d-1,I+1)` instead of `A(J+d,I)` for other iterations (see example mentioned above). A technique implemented in the HP-7200 [11], which is also intermediate between prediction tables and software prefetching, consists in storing the reference stride in one of the registers used by the load/store instruction and the current address in another register. After computing the current address, the stride is added again providing the next address which is then prefetched. Though more efficient than software prefetching, this scheme is again limited to single-stride streams and wrong predictions occur every time the stride changes. Besides, the limited number of registers per instruction makes it impossible to have this scheme accomodate multiple strides.

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 [10]) 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 presented. Finally, in section 4, the performance of the scheme is evaluated.

2 Streaming Prefetch

The basic principle of *streaming prefetch* is to use array reference subscripts to drive a prefetch mechanism. Optimally, the remaining cache misses in a loop nest with linear array references are only conflict misses. To achieve this goal, it is necessary: (1) to get and store the subscript of all array references in a loop nest, (2) to be able to predict a few iterations in advance (so that prefetch requests can be sent early enough) the end of a loop, (3) to implement prefetching so that normal cache operations are not disrupted and memory traffic is not significantly increased. These issues are addressed in the next sections.

2.1 Using Subscripts to Drive Prefetching

In the table prediction schemes proposed in [7, 5, 8], 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 [7, 5, 8]. 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.

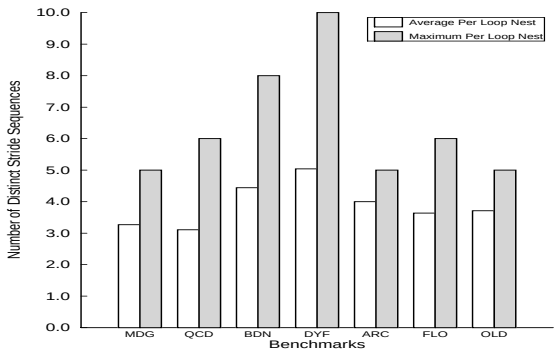

Figure 1: *Average and Maximum Number of Stride Sequences.*

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. Figure 1 shows the average number of distinct stride sequences per loop nest for all benchmarks used and the maximum number of stride sequences observed. It can be seen that there are usually few distinct stride sequences in a given loop nest. 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>`.

Strides are not equal to coefficients

The issue now is to determine which value to use as strides. Indeed, the loop index cofficients 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 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

```
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 J3=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

Figure 2: *Example Loop and Corresponding Stride Table.*

`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

Limitations of using subscripts to derive strides

There are two main limitations to this scheme. First, subscripts which coefficients are functions of induction variables cannot be instrumented, since the coefficients cannot be determined at loop nest entry. Second, determining the number of distinct stride sequences at compile-time is not optimal since some coefficients can have different symbolic expressions and identical values. Consequently, the number of distinct stride sequences can be found more important than it actually is. Still, most stride redundancies usually correspond to object-code optimizations (load/store duplications) so that compile-time analysis of stride sequences is sufficient. The statistics provided in figure 1 suggest a stride table size of approximately 8 entries.

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 [7, 8] 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 2.4) 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. As mentioned above, this flaw can be significant in numerical codes which deal with multi-dimensional objects. More precisely, the frequent use of square blocks in matrix operations or other mathematical objects induces small inner loops, therefore restricting the efficiency of single-stride prefetching. Besides, a significant share of inner loop spatial locality is already exploited by the cache line as far as the inner stride is small, so that the usefulness of prefetching for inner loops is further reduced. These assertions were validated with statistical studies of loops in the benchmarks considered. Figure 4 shows the distribution of inner lengths, while figure 3 shows the average and maximum inner loop lengths found. It can be seen that a majority of inner stream lengths are fairly small (about 20 iterations), even though high peaks can be observed. Note

that these statistics do not take into account the 0-length streams of references to constants. Figure 5 shows the distribution of inner stride values and confirms that a majority of inner strides are small (4 or 8 bytes).

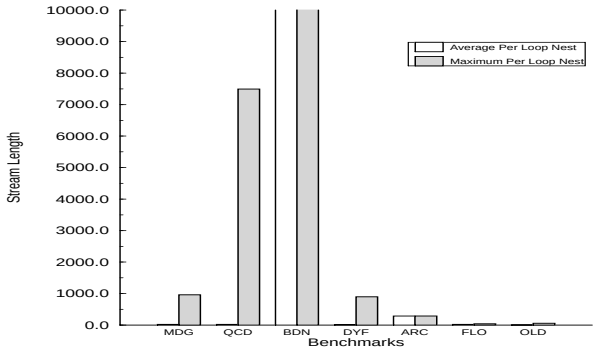

Figure 3: *Average and Maximum Stream Lengths.*

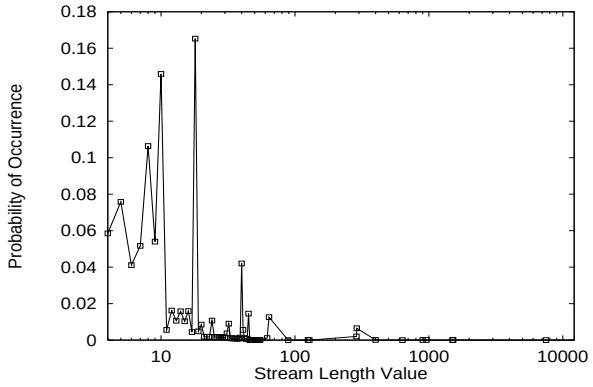

Figure 4: *Distribution of Inner Stream Lengths.*

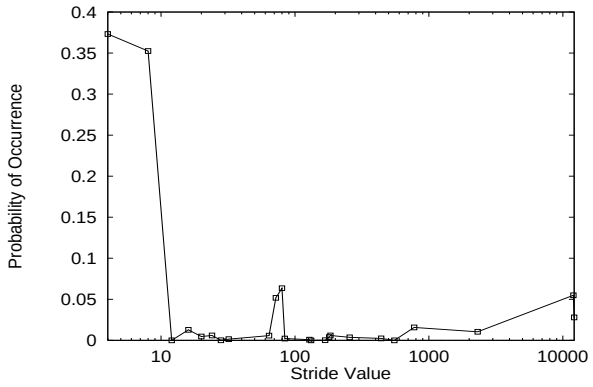

Figure 5: *Distribution of Inner Stride Value.*

Computing the stream length

These observations show that 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.[1] 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.

Note that the length informations can be used to assist branch prediction, but the associated benefit has not been taken into account in the experiments of section 4.

Non-Constant Stream Lengths

The scheme presented above can predict loop ends when the associated stream length is constant. However, many numerical methods are based on triangular matrices and therefore include loops with non-constant boundaries, i.e., non-constant stream lengths. However, in these algorithms the stream length usually increases or decreases regularly, i.e., with a constant stride (when scanning the rows of a triangular matrix...). Therefore, the previous scheme can be augmented with a *length stride field* in addition to the *current length* and *last length fields*. In opposition to loops with constant number of iterations, multi-stride is then effective only after the length stride has been determined, i.e., after 2 executions of the loop nest.

2.3 Overview of Streaming Prefetch

Obtaining the appropriate stride

Let us now consider how the *stride table* can be used to select the appropriate prefetch stride. The last modification to the *stride table* is a flag bit associated to each depth column except for the first one (inner loop/depth). These flag bits indicate which coefficients should be used for computing the stride. Initially (when entering a loop nest), all flag bits are set to 0. Whenever one of the length tests described above is true, the flag bit of the next column (corresponding to the next outer depth) is set to 1. If the last column flag is set, no modification occurs. When the inner loop branch is not taken, all flags are reset to 0. This reset condition is a little difficult to understand. Basically, the flags of the outer loops are set when passing through the corresponding branches. When the inner loop has been executed again, the outer strides may have changed and therefore should be reset until further notice.

Let us consider again the example of figure 2. Initially, all flag bits are set to 0. For reference `B(I3,I2,I1)`, the prefetch stride is equal to 1 since no flag is set. After the first time the branch of loop `I3` (called branch `I3`)

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

is not taken, the *last length* entry of the *stride table* is set. For reference `F(I2)` the stride used is equal to 1. The first time each loop is executed, no multistride prefetching is used. On the second iteration of loop `I3`, a signal is sent to the stride table when the current length associated with branch `I3` is equal to the last length minus δ (the prefetch distance), a signal is sent to the *stride table* and the coefficients of the next loop are activated (coefficients of loop `I2`). From then on till then end of loop `I3`, `B(I3,I2,I1)` and `D(I1,I2,I3)` respectively prefetch with a stride equal to `1*(L3-U3)+N`, `N*N*(L3-U3)+N`. When branch `I3` is not taken, all flags are reset. Assume now that branch `I2` sends a signal that loop `I2` is about to terminate (this can take place only after loop `I2` has been executed at least once). Then, loop `I3` starts again, when the prefetch distance test is true, a signal is sent to the *stride table*. Because one flag is already set, the flags corresponding to both depth=1 and depth=2 are set. Consequently, from then on till the end of the loop, the stride for references `B(I3,I2,I1)` and `D(I1,I2,I3)` is equal to `1*(L3-U3)+N*((L2-U2)+N*N`, `N*N*(L3-U3)+N*(U2-L2)+1`. Also, the stride for reference `F(I2)` is still equal to 1 because the coefficient corresponding to loop index `I1` in its subscript is equal to 0.

Let us now briefly summarize the events occurring upon executing a branch or load/store instruction.

Branch Instruction:

Branch taken: The current stream length is incremented. Then, the following test is done:

$$current\_length == last\_length + length\_stride - \delta?$$

In the current implementation, δ is hardware coded, but other solutions are discussed in section 4.

Branch not taken: The last length is set to the current length and the current length is reset. The *stride table* is unaffected, except if this branch corresponds to the innermost loop. In that case, all flags are reset to 0.

Load/Store Instruction:

Using the *stride table* bits of the load/store instruction, the corresponding *stride table* entry is selected. References which were not considered multi-linear at compile-time have a null index which misses in the table. For references which hit, the prefetch target address is computed according to the set *stride table* flags.

Also, note that once a flag is activated, it is necessary to prefetch with an outer stride during the δ last iterations of the stream, note only *at* the iteration where the prefetch distance test was true.

2.4 Hardware Implementation Issues

As mentioned in section 1, a prefetching scheme can behave poorly or even degrade global processor performance if several impementation 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. 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 independant 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 [14]. 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 ovrer 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.*

With this subbank partitioning technique, it is possible to achieve a real *streaming effect*: requests are prefetched without stalling the cache, i.e., prefetch is transparent, and they are reloaded the same way. The result is an uninterrupted flow of data through the cache.

To evaluate the importance of careful prefetching implementation, we have simulated *streaming prefetch* with and without the subbank technique and deduced the benefit due to this implementation. Without subbanks, *streaming prefetch* results in performance degradations (see section 4 for more details).

Because issuing a prefetch request with the *streaming prefetch* scheme requires passing through several steps (hit in the *stride table*, computation of the prefetch address, cache check...), sending the prefetch request to memory is often delayed by several cycles. Therefore, it is necessary to prefetch with a distance higher than 1 iteration. We found the optimal prefetch distance to be $\delta = 8$ iterations for the inner loop and 1 iteration for any outer loop (for outer loops, it is not necessary to prefetch in advance since one whole execution of the inner loop elapses before the prefetch request can be exploited). Even with a large prefetch distance, the cache check step can become the performance bottleneck in case numerous requests (processor and prefetch) correspond to the same subbank (this can happen with specific strides). In such cases, the requests can be strongly delayed. Buffering them is no solution since a too much delayed prefetch request can arrive too late. The most practical solution is to discard them, but the prefetch scheme then looses its efficiency. Actually, a simple observation can be used to strongly reduce the number of cache checks and restore the scheme fluidity. Because the accesses are strided and because the strides are usually small (especially the inner strides; see Figure 5), many prefetch requests fall into the same cache line. Consequently, the following filtering technique was used:

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.[2]
3. If the stride is null no prefetch request issued.

Criterions 1 and 3 are obvious. If criterion 2 is not true, it means that the target address minus the stride is an address of the same line. Most probably, it means the previous prefetch address was already in this line. When the stride is equal to one or two words (the most frequent case) only $\frac{1}{L}$ and $\frac{2}{L}$ of the prefetch requests are effectively issued (where L is the line size in words). Because the logic for computing modulos which are not powers of two is complex, the criterion has been implemented by using the next power of two greater than the stride. This filtering technique proved to be economical and highly effective. The most frequently used criterions are 2 and 3.

3 Experimental Framework

Subscripts coefficients

To get the coefficients expression, the Sage++ [4] compiler was used to obtain a linearized expression of array subscripts. For the loop nest of section 1, the following instrumentation calls are inserted in the source code:

[2] If the stride is negative, the criterion is that the target address modulo the stride is larger than the address minus the stride.

```
REAL A(N,N),X(N),Y(N)

C--- call trace(coefficient expression, loop index depth, reference number)
C--- Here reference 1 = A(J,I), 2 = X(J), 3 = Y(I) (load), 4 = Y(I) (store)
call trace(1,0,1)
call trace(N,1,1)
call trace(1,0,2)
call trace(1,1,3)
call trace(1,1,4)

DO I=1,P
  reg = Y(I)
  DO J=1,P
    reg = reg + A(J,I)*X(J)
  ENDDO
  Y(I) = reg
ENDDO
```

Because no formal calculus engine is available within Sage++ to compare coefficients expressions (therefore `2*N` and `N*2` could be considered distinct values), the comparison was based on the numerical values obtained prior to loop entry at run-time. In some cases, it is possible that formal calculus may still not be sufficient to find out that some coefficients have identical numerical values. This can potentially increase the size of the *stride table*, but considering the small optimal size found, even doubling the *stride table* size raises no major difficulty.

The most troublesome issue was to map source code references to objet code load/store so as to determine the *stride table* entry of a load/store instruction. For that purpose, we further instrument the loops using Sage++ to obtain array boundaries. These informations are passed to the object code tracer (we used the Spa package [6]). Passing informations from the source code to the tracer was done through files because the traced code process is not visible from the analyzer part of the Spa package (`spanner`). To perform the mapping without altering the object code trace, the code is instrumented twice differently. In one case, only the instrumentation mentioned above is inserted (i.e., prior to the loop nest), in the other code, a trace call is inserted after every array reference. This second code is *not* used to obtain the traces, but to allow the identification of the references in the non-instrumented code through a comparison of the references each loop nest. The mapping is done within the tracer, based on the array being accessed (this is determined by a comparison with the array bounds provided by the traced program) and the reference stride sequence. Consequently, mapping may not work properly for non-linear references, but only the latters are of interest in this study. For the mapping to work properly, we had to restrict to `-O2` optimizations on Sun Sparcstations (the default optimization level), otherwise loop unrolling is used and mapping may be inaccurate.

Considering our scheme relies on the predictability of loop nest reference patterns, some loops could not be instrumented. Loops with I/O statements were not instrumented. More important, loops with `CALL` statements were not instrumented since the prefetch distance could not be determined in this case (if numerous references take place in the called subroutine, data prefetched in the calling loop would probably be flushed before being used). However, the main limitation proved to be induction variables. Since induction variable expansion is not yet implemented in Sage++, the corresponding loops could not be instrumented. Finally, a last class of code constructs limited our analysis: in some dusty-deck codes, loops are not implemented as such, but with gotos and counters. Of course, the absence of loop indices made it impossible to deal with such constructs.

Finally, *streaming prefetch* also required that branches corresponding to loop branches be flagged. This information can be obtained trivially if one has access to the complete compiler chain. However, in our case, we had no way to pass this information from the source code to the object code. Therefore, branches were analyzed on-the-fly in the tracer and flagged accordingly (in most cases, the analysis basically summed up to check whether they were backward branches).

Simulations

The simulator includes two parts: the cache operations and the branch predictions. Therefore, both load/store and branch instructions were passed to the simulator. To come as close as possible to the conditions of a superscalar processor, we assumed one load/store request is sent to the cache every cycle, without disruption (due to branches or else).

Benchmarks and traces

Seven benchmarks from the Perfect Club Suite [2] were used. For each benchmark, a 50 million-entry trace was extracted. In our case, source-code trace would have been the easiest solution, but because hardware implementation issues had to be finely studied, we decided against it and extracted objet code traces to get all load/store references.

4 Performance Evaluation

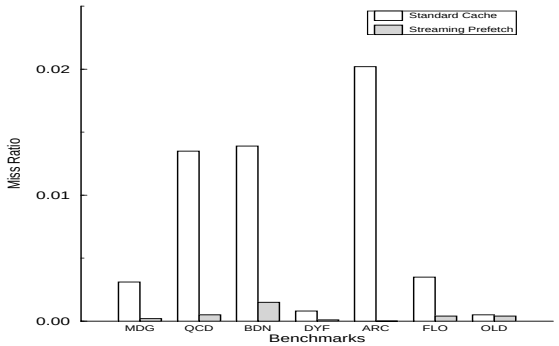

Figure 6: *Miss Ratio.*

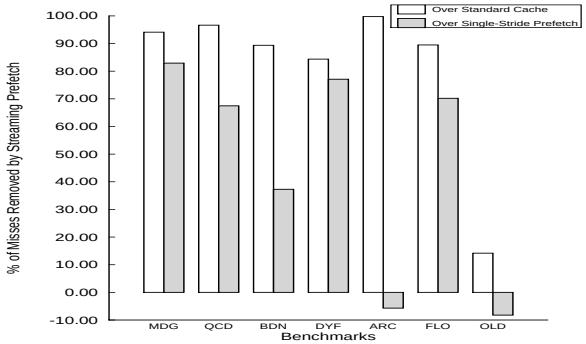

Figure 7: *Fraction of Misses Removed.*

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 relatively standard for current or soon to appear processors. In such caches, compulsory misses will correspond to a more important share of all misses than in small direct-mapped caches. Consequently, *streaming prefetch* is relatively targeted towards such caches though it can be applied to small caches as well. In Figure 6, 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 all miss ratios down to 0.15% or less. Figure 7 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 comes 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.

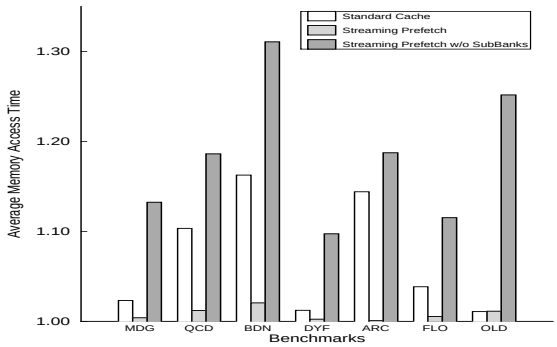

Figure 8: *Average Memory Access Time.*

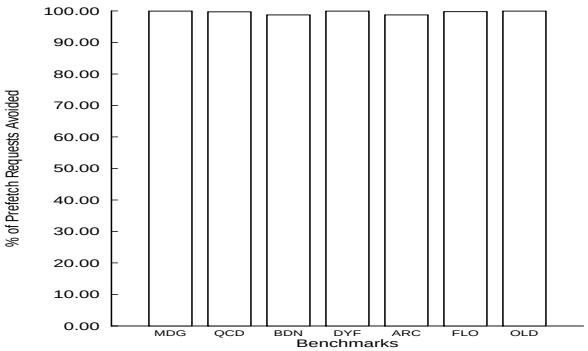

Figure 9: *Fraction of Prefetch Requests Avoided.*

The second aspect of *streaming prefetch*, the efficiency of the dedicated hardware support used to minimize cache perturbations is shown in Figure 8. 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. It is our belief that precise simulations of many prefetch schemes not implemented with the proper hardware support would degrade performance of superscalar processors (i.e., assuming a cache request every cycle), mainly because of cache stalls due to prefetch checks and prefetch reloads. Also, the mechanism used to limit the number of prefetch requests was highly effective at limiting cache checks as can be seen in Figure 9.

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 10. 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 very 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

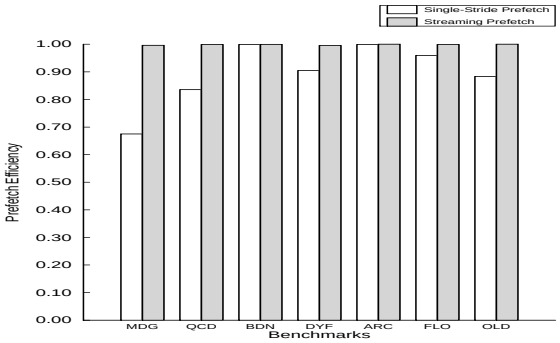

Figure 10: *Prefetch Efficiency.*

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] G. Irlam. *SPA package*, 1991.

[7] T-F. Chen J-L. Baer. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proceedings of IEEE Supercomputing*, 1991.

[8] Y. Jegou and O. Temam. Speculative Prefetching. In *Proceedings of the ACM International Conference on Supercomputing*, 1993.

[9] 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.

[10] 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.

[11] Dick Pountain. A different Kind of RISC. *BYTE*, August 1994.

[12] Alan Smith. Cache Memories. *Computing Surveys*, 14(3), September 1982.

[13] 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.

[14] O. Temam and N. Drach. Software Assistance for Data Caches. In *1st Symposium on High Performance Computer Architectures*, January 1995.