# Stride Directed Prefetching in Scalar Processors

**John W. C. Fu**

Intel Corporation
MPG Architecture and Planning
1900 Praire City Road, Folsom, CA 95630
jfu@pcocd2.intel.com

**Janak H. Patel and Bob L. Janssens**

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801
patel@crhc.uiuc.edu, blj@crhc.uiuc.edu

## Abstract

*The execution of numerically intensive programs presents a challenge to memory system designers. Numerical program execution can be accelerated by pipelined arithmetic units, but to be effective, must be supported by high speed memory access. A cache memory is a well known hardware mechanism used to reduce the average memory access latency. Numerical programs, however, often have poor cache performance. Stride directed prefetching has been proposed to improve the cache performance of numerical programs executing on a vector processor. This paper shows how this approach can be extended to a scalar processor by using a simple hardware mechanism, called a stride prediction table (SPT), to calculate the stride distances of array accesses made from within the loop body of a program. The results using selected programs from the PERFECT and SPEC benchmarks show that stride directed prefetching on a scalar processor can significantly reduce the cache miss rate of particular programs and a SPT need only a small number of entries to be effective.*

## 1. Introduction

The execution of numerically intensive programs presents a challenge to memory system designers. Numerical program execution can be accelerated by pipelined arithmetic units, but to be effective, must be supported by high speed memory access. A cache memory is a well known hardware mechanism used to reduce the average memory access latency [Smit82]. Numerical programs, however, often have poor cache performance because of large data sets, little data reuse and interaction between the vector stride distance and the selected cache block size [FuPa91].

Vector processors do not typically employ cache memories because of poor numerical program cache performance. Numerical programs executed on a scalar processor show better cache performance (lower miss ratio) but can still suffer high miss rates [Fu92]. As a simple example, consider the matrix multiple $C = A \times B$. Table 1 shows the cache miss ratio for the scalar execution of the matrix multiply for matrix sizes of $100 \times 100$. For comparison purposes the corresponding vector execution is also shown. The results were obtained using trace driven simulation of a 4 Kbyte cache with block sizes of 8, 16, 32 and 64 bytes. The traces are from executions on an Alliant FX/80. Each trace is for single processor execution where the scalar and vector versions are generated using compiler optimizations. Two miss ratios are shown for each execution; *ALL* means that all memory data references are simulated and *MATRIX* means that only references to matrix data (data size of 8 bytes) are simulated. There are 19 and 2.2 million references for scalar and vector executions respectively but only 4 and 2 million of these references are to matrix data. Note that the vector miss ratios are computed relative to the number of vector accesses and not the number of vector referencing instructions. For example, a vector instruction may load 32 elements but this is counted as 32 vector accesses.

The results show that scalar execution results in better cache performance (lower miss ratio). For *ALL* memory references, accesses to the index variables contribute significantly to the hit ratio. Ignoring these references in scalar execution (*MATRIX*), the cache miss ratio is much worse though still lower than the vector execution miss ratio. It is the cache performance of these references that needs to be improved as these are the references that are accessed for high speed arithmetic execution. Scalar execution results in about twice the number of matrix accesses. One reason is that the register space for vector processing is higher than for scalar processing.

| blk size | SCALAR | | VECTOR | |
|---|---|---|---|---|
| | *ALL* | *MATRIX* | *ALL* | *MATRIX* |
| 8 | 0.066 | 0.303 | 0.549 | 0.588 |
| 16 | 0.061 | 0.270 | 0.490 | 0.523 |
| 32 | 0.066 | 0.264 | 0.464 | 0.493 |
| 64 | 0.066 | 0.258 | 0.452 | 0.485 |

**Table 1: Scalar, vector matrix multiply miss ratios**

This research was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Airforce) under contract N00014-84-C-0149 and in part by NASA contract NAG 1-613.

To improve numerical program cache performance on a vector processor, [FuPa91] proposed the use of stride directed prefetching. The schemes proposed use the relationship between the vector stride distance and the cache block size to direct prefetching. It was suggested that this method can be extended to scalar processors, where the stride is not explicitly known, by using a history table to calculate the stride distance of array or vector accesses made from within the body of program loops. This paper shows how a simple hardware mechanism, called a stride prediction table (SPT), can be used to calculate the stride distance on a scalar processor and how this stride can be used to direct prefetching.

The rest of this paper is organized as follows. Section 2 describes how the stride distance can be predicted and Section 3 describes the trace driven simulation method used. Results with selected programs from the PERFECT and SPEC benchmarks, presented in Sections 4 and 5, show that SPT directed prefetching can significantly reduce the cache miss rate of some programs. However, the results are sensitive to the characteristics of the program and when prefetching is initiated. Programs with deep loops that are highly vectorizable yield the best performance improvement. A measure for prefetch overhead is introduced in Section 6. This shows that SPT directed prefetching is accurate with low overhead for highly vectorizable programs, but for other programs the overhead cost of removing cache misses, using this scheme, can be significant. Results in Section 7 show that a SPT need only have a small number of entries to be effective. Concluding remarks are given in Section 8.

## 2. Stride Prediction and Directed Prefetching

Consider the matrix multiply previously discussed and shown in a typical triple nested loop form in Figure 1. The sequence on the right is a possible set of assembly level instructions for the loop body where the $i,j,k$ indices are the effective memory addresses of the vector elements. Consider loading vector $B$ at instruction address $W$. In the first loop iteration an element of $B$, at memory address $b(1,1)$, is loaded into the floating point register $fp$ and $b(2,1)$ is loaded into $fp$ in the second iteration. In the third iteration the element at address $b(2,1)+stride$ is loaded, where stride $=b(2,1)-b(1,1)$. For each $i^{th}$ iteration of loop $K$, the address of the element of $B$ accessed in the $i+1^{th}$ iteration by instruction $W$, can be predicted using the previous element address and the calculated stride. Only when the outer loop index is incremented can this become untrue.

```
DO I = 1,L                          .
 DO J = 1, M                        .        W: fp ← b(k,j)
  DO K = 1, N                       .        X: fp ← fp×a(i,k)
                                    .        Y: fp ← fp+c(i,j)
   C[I,J] = [CI,J] + A[I,K] * B[K,J]        Z: fp → c(i,j)
                                    .
CONTINUE                            .
                                    .
```

**Figure 1: Matrix multiply.**

A Stride Prediction Table (SPT) consists of 3 fields: the instruction address (IA), the effective memory address, (MA) and a valid bit (V). The IA is the memory address of a memory referencing instruction, and MA is the corresponding memory address of the data being referenced. For example, in Figure 1, IA is $W$ and $MA$ is $b(k,j)$ for the memory referencing instruction that loads an element of $B$. The valid bit denotes a valid IA and MA entry in the SPT.

Figure 2 shows the basic SPT organization. Let $IA_{cur}$ and $MA_{cur}$ be the current instruction and data memory address pair being processed and $IA_{spt}$ and $MA_{spt}$ be some memory address pair held in the SPT. Each memory access made by the processor presents a $IA_{cur}$ and $MA_{cur}$ address pair to the SPT. The address $IA_{cur}$ is used to access the SPT and if the SPT entry is valid the corresponding $IA_{spt}$ and $MA_{spt}$ are made available. The stride is calculated as $MA_{cur}-MA_{spt}$ and the prefetch address is $MA_{cur}+stride$. This is a spt hit and a prefetch attempt can be made if the *stride* is not zero. The $MA$ field of the SPT entry causing the spt hit is updated with $MA_{cur}$. If the SPT entry is not valid, the $IA_{cur}$ and $MA_{cur}$ address pair is written into the SPT and the valid bit set. This is a spt miss.

The SPT is a cache of memory data addresses previously referenced and as such can be implemented using any of the known cache organizations. This paper does not investigate SPT implementation in detail; however, later results show that the SPT can be quite small and simple to be effective. Only two SPT organizations are considered: a SPT with an infinite number of entries and a SPT with a finite number of entries using direct mapping.

Additional fields can be used in the SPT to minimize incorrect stride prediction. For example, a previous stride field can be used to prevent prefetches when the current stride does not match the previous calculated stride. Using this field, prefetches will not be initiated when an outer loop index is incremented or when the stride is random. The results presented in Section 7 show that this only occurs for a small fraction of the references for two of the programs where stride directed prefetching has the most benefit. This does not preclude significant stride changes in other programs. However, for these programs this form of prefetching is not likely to be effective because stride changes will limit the number of prefetches.

[1] The actual stride distance of the elements of B depends on the language. For fortran, matrix data is mapped column major such that the elements of B will be in consecutive memory locations, but for C the stride will be of size N for the example in Figure 1.

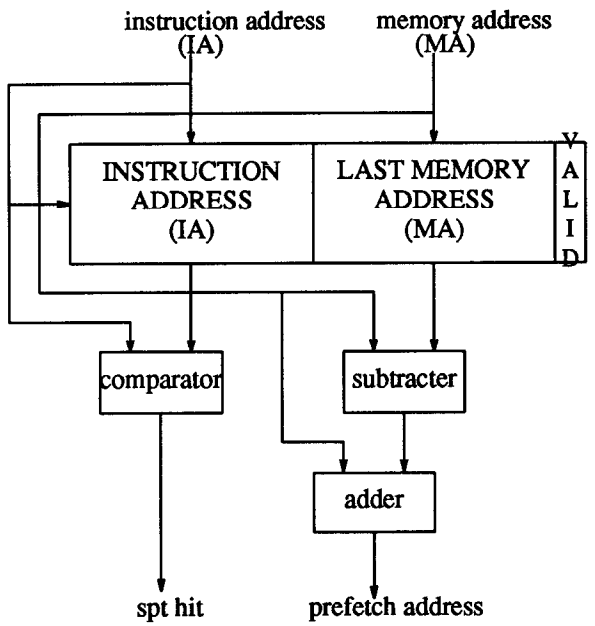


**Figure 2: Stride Prediction Table.**

Prefetching is not a new idea; previous studies include [Smit82, Przy90]. Prefetch schemes differ in how prefetch addresses are determined, when prefetches are initiated and where the prefetch data is loaded. In this paper a prefetch can be initiated only if a valid stride has been calculated. Results are shown for prefetch initiation under three demand access[2] conditions of: demand hit, demand miss and prefetch all (demand hit or miss). If the prefetch access is a miss, the prefetch data is loaded only into the cache. The unit of prefetch is a memory block. Prefetch blocks are treated like demand access blocks in the cache i.e. they can replace demand blocks and can be replaced by demand blocks. A similar scheme is independently proposed in [BaCh91, Skle92]. Unlike the scheme described in this paper, the scheme in [BaCh91] relies on instruction stream prediction using branch prediction and a lookahead program counter to initiate prefetches.

## 3. Method

The results presented in this paper use traces collected from an Encore Multimax using an inline tracing system called TRAPEDS [StFu89, Jans91]. Because of time constraints, only a limited number of programs are evaluated. The programs used are ADM, ARC2D, DYFESM and BDNA from the PERFECT benchmarks [Berr89]. These programs are also used in [FuPa91]. Two programs, MATRIX-3000 and ESPRESSO, from the SPEC benchmarks are also used. The former is used because it is expected to have good performance with SPT prefetching and the latter for the opposite reason. All the PERFECT programs and MATRIX were originally coded in Fortran but have been converted to C using f2c, a program that translates Fortran source to C source. Some modifications have been made to the converted C programs and some IO routines removed or modified to execute with TRAPEDS.

Because of disk storage costs, immediate simulation is used [BoKW90]. After each memory reference is generated the model is immediately simulated. Initially, attempts were made to execute all programs until completion, but because of the trace lengths and the need to regenerate each trace for each case, simulations are terminated, if not completed, after 1.2 billion data references. Both SPEC programs are executed until completion. For each of the programs only the data references are simulated; no instruction stream or stack references are simulated. The stack references are not simulated because they are not the focus of the SPT prediction scheme.

Some memory reference characteristics of the traces used are shown in Table 2. The DATA and STACK numbers refer to the number of references to the data and stack areas. The PERFECT benchmarks issue in excess of 1 billion data references. The SPT size is the maximum number of SPT entries required for each program execution i.e. the number of memory referencing instructions in each program. Note that the number of SPT entries required do not correlate with the number of memory references made by a program. For example, the memory referencing instructions in MATRIX is 5% of those in ESPRESSO but it generates 19 times more references. Results presented in Section 4, 5 and 6 all use an infinite number of SPT entries. Only Section 7 considers a limited number of SPT entries.

| PROGRAM | DATA | | STACK | | SPT |
|---|---|---|---|---|---|
| | ref * | % read † | ref * | % read † | entries |
| ADM | 1662 | 80.3 | 716 | 90.9 | 3874 |
| DYFESM | 1409 | 76.4 | 330 | 97.0 | 2650 |
| ARC2D | 2000 | 94.2 | 725 | 97.6 | 4716 |
| BDNA | 1973 | 90.9 | 507 | 85.1 | 5098 |
| MATRIX | 651 | 66.9 | 670 | 98.6 | 94 |
| ESPRESSO | 31 | 83.9 | 23 | 59.2 | 1745 |

* - millions of references † - % read references

**Table 2: Reference characteristics**

[2] A demand access is a memory access initiated by the processor in executing a memory referencing instruction. A prefetch access is an access initiated external to the memory referencing instruction stream.

## 4. Stride Direct Prefetch Performance

A prefetch scheme that loads data into the cache is useful only if it can reduce the number of cache misses. If this is not true the scheme is unlikely to result in improved system performance. This section presents the performance of a cache that does not prefetch (no-prefetch) and a cache that uses SPT predicted strides to guide prefetching (spt-prefetch). Whether a prefetch scheme that successfully reduces cache misses will lead to better system performance depends on factors including prefetch access timing and the available bandwidth of the memory system. These factors are not the focus of this paper.

The cache performance as a function of the block size for a no-prefetch and a spt-prefetch cache is shown in Figure 3. Each cache is 64 Kbytes and direct mapped. A no-prefetch cache loads a single memory block after a demand access miss. A spt-prefetch cache can initiate a prefetch access whenever a stride can be calculated by the SPT. A previous stride field is not used to prevent prefetches. If the prefetch access is a miss only a single block is loaded into the cache. Prefetching multiple memory blocks from a single calculated stride is not used in this paper.

The results for three prefetch initiation conditions are shown: prefetch on a demand access miss (pf_miss), prefetch on a demand access hit (pf_hit) and prefetch all (pf_all). These conditions were chosen as they are the simplest conditions to evaluate. In pf_miss, a prefetch is initiated whenever a valid stride is calculated and the demand access is a miss in the cache. If both demand and prefetch accesses miss in the cache, both addresses are sent to the memory with the demand address preceding the prefetch address. Similarly, pf_hit only initiates a prefetch access if the demand access was a hit. In this case a prefetch memory request is only issued when there is no demand memory request. Finally, pf_all initiates a prefetch access whenever a valid stride has been calculated regardless of the outcome of the demand access. In all cases the prefetch address is *demand address + calculated stride.*

The spt-prefetch cache performance is nearly always better (lower cache miss rate) than a cache with no prefetching. The improvement in cache performance with a spt-prefetch cache is most significant for ARC2D, BDNA and MATRIX. This is not surprising since these programs have characteristics that should benefit the most from this form of prefetching. Both ARC2D and BDNA have more than 90% and 80% vector references respectively and both load long vectors [FuPa91]. Programs that vectorize easily, in scalar form, have regular nested loop structures and loop invariant strides that can be captured in a SPT. Similarly, MATRIX is known to consist of regular nested loop accesses to array data similar to the loop nest in Figure 1. The programs ADM, DYFESM and ESPRESSO show less improvement in performance, but using spt-prefetch does reduce the number of cache misses with the improvements being more significant when the cache block size is small. Note that the no-prefetch cache miss rate is already low for these programs so it is harder to reduce the number of misses in these programs.

Of the three prefetch initiation schemes, pf_all shows the best performance, with pf_miss having the performance closest to the no-prefetch cache. This is because pf_miss and pf_all represent the least and most number of prefetch accesses initiated. Initiating a prefetch after a demand miss is a poor choice because the number of initiations is small and prefetch memory requests conflict with the demand memory requests.

Initiating a prefetch on a demand access hit does not result in good performance when the block size is 8 bytes for ARC2D and MATRIX. This is because with such a small block size there are few hits (and a large number of misses) and this limits the number of prefetch attempts. As the block size is increased the demand hit rate increases, more prefetches are initiated, and pf_hit performance approaches that of pf_all. Prefetch initiation after a demand hit is attractive as a prefetch memory request does not conflict with a demand memory request. However, when a program incurs a large number of misses this is also a condition to attempt prefetch (assuming that prefetching can reduce demand cache misses). The results indicate that spt-prefetch cache performance is very sensitive to the conditions under which prefetches should be initiated and a major factor for consideration in using such a prefetch strategy.

## 5. Prefetching or Larger Block

In comparing the no-prefetch and spt-prefetch caches, Figure 3 shows the cache performance at the same block size i.e. the effect of prefetching a single block. For a cache with a particular block size a prefetching cache loads more data than a no-prefetch cache. As with the results in [FuPa91], it is also appropriate to compare the performance of a no-prefetch and spt-prefetch cache at approximately the same load size. For example, a spt-prefetch cache with an 8 byte block is compared with a no-prefetch cache with a 16 byte block. This is reasonable because the question of interest is: given a no-prefetch cache with a particular block size $n$ and a certain performance, does increasing the block size to $2\times n$ or prefetching another $n$ bytes result in better performance? This can also be considered as comparing the accuracy of loading more sequential data (increasing the block size) with data access prediction. Figure 4 shows the relative improvement in the cache performance between a spt-prefetch cache with a load size of $2\times n$ (and a block size of $n$) and a no-prefetch cache with a block size $2\times n$ bytes. The relative improvement in cache performance is:

$$\frac{CM_{np}^{b} - CM_{pf}^{\frac{b}{2}}}{CM_{np}^{b}}$$

where $CM_{np}^{b}$ is the number of cache misses in the no-prefetch cache with a block size of $b$ bytes and $CM_{pf}^{\frac{b}{2}}$ is the number of cache misses in the prefetch cache with $\frac{b}{2}$ bytes.

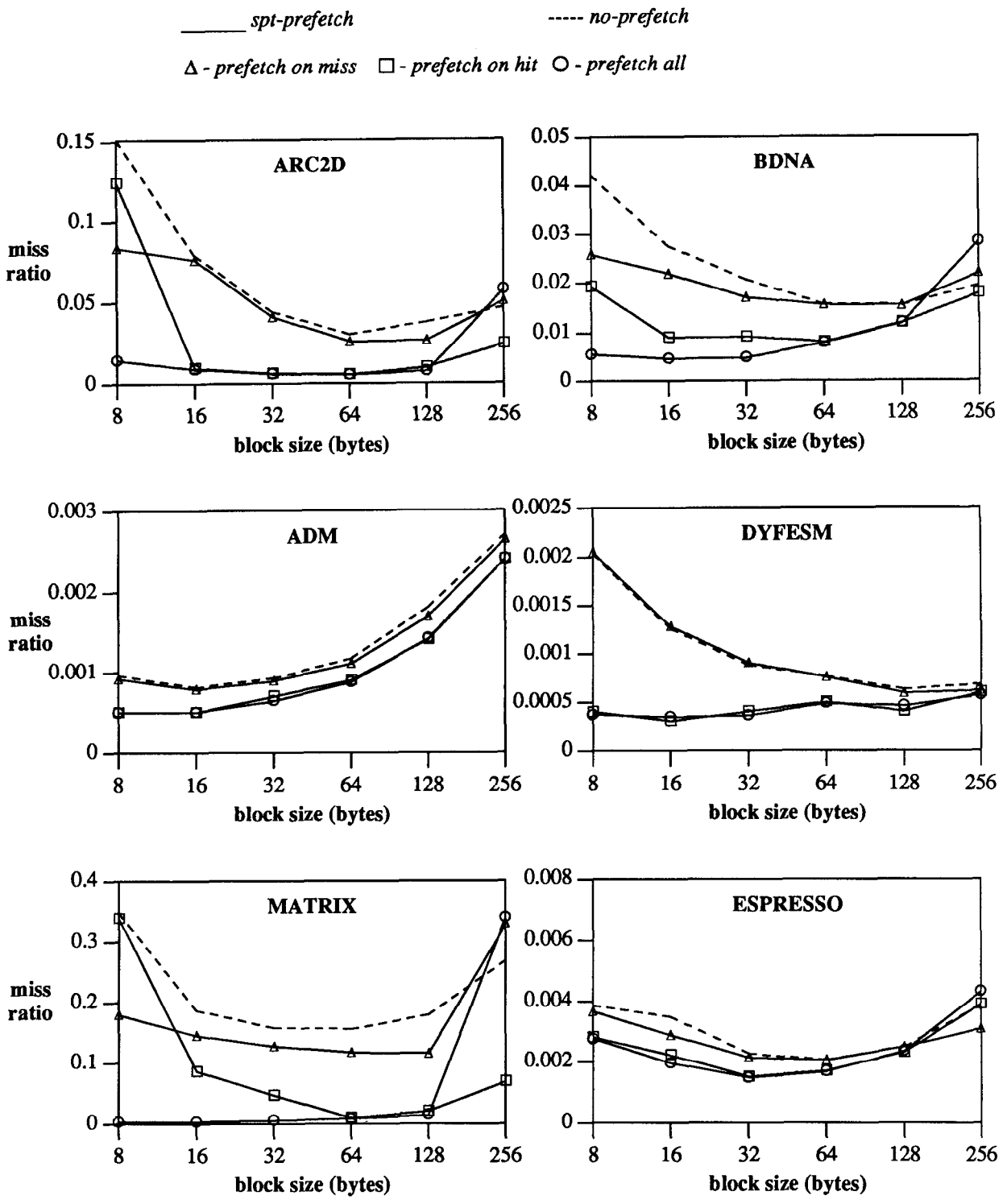


**Figure 3: SPT predicted stride prefetch performance.**

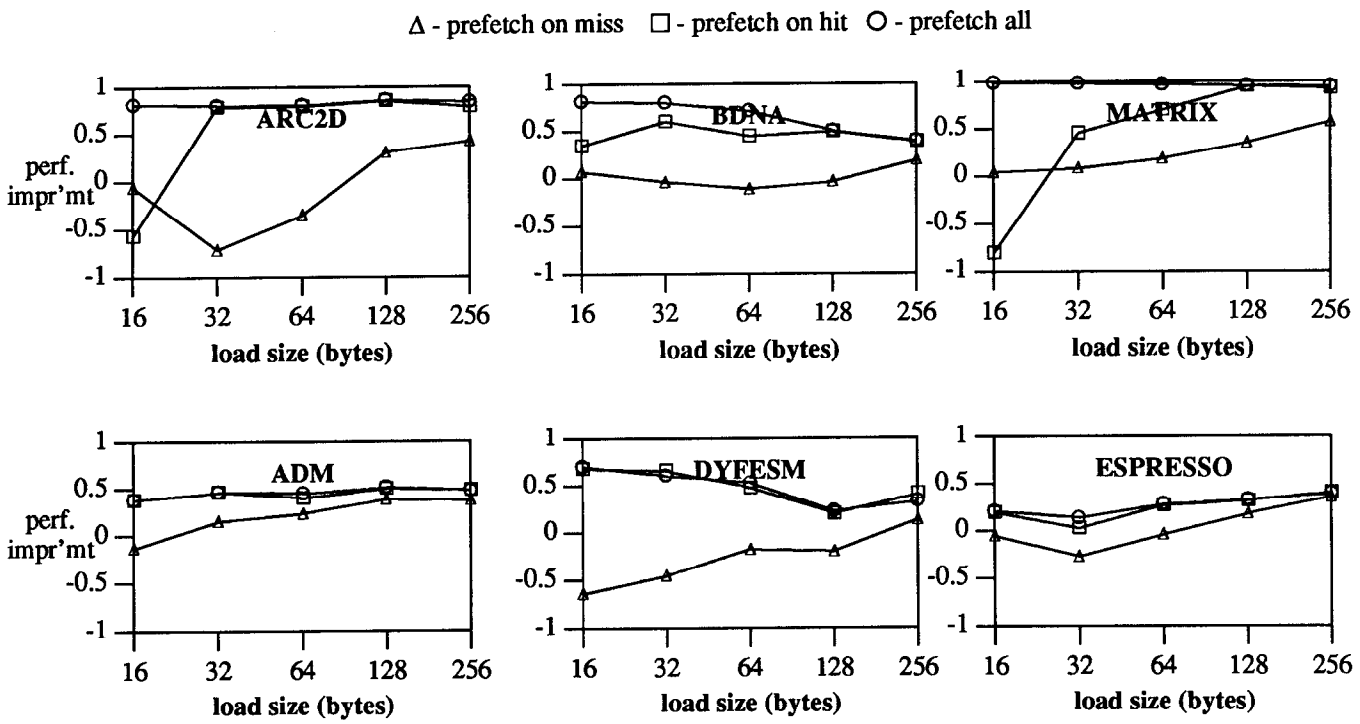


**Figure 4: Relative cache performance improvement of spt-prefetching versus larger block size.**

Hence, a no-prefetch cache with a block size of $b$ bytes is being compared with a spt-prefetch cache with a block size of $\frac{b}{2}$ that attempts to prefetch $\frac{b}{2}$ bytes whenever a valid spt stride has been calculated.

The results in Figure 4 show that for ARC2D, BDNA and MATRIX, pf_all is always significantly better than a no-prefetch with a larger block size. For MATRIX the miss rate decreases by almost 100% for all load sizes. The relative cache performance improvement is about 80% for BDNA when the load size is small (16 and 32 bytes, prefetch blocks size 8 and 16 bytes) but decreases as the load size is increased. With pf_hit the miss rate can decrease significantly but only when the prefetch cache block load size is larger than 8 blocks. For example, in ARC2D a prefetch cache with an 8 byte block and a prefetch of 1 is significantly worse than a no-prefetch cache with a 16 byte block. As previously stated this is because with a small block size of 8 bytes there is a high miss rate. For pf_miss a no-prefetch cache with a 32 byte block can be much better than a prefetch cache (ARC2D and MATRIX). This is because there are very few misses and few prefetch loads with this block size.

For the programs ADM, DYFESM and ESPRESSO the percentage improvement with a spt-prefetch cache over a no-prefetch cache with a larger block is generally less significant though both ADM and DYFESM do show good improvements with pf_hit and pf_all. However, with the same cache capacity, a spt-prefetch cache with a particularly block size will require a larger tag store than a corresponding no-prefetch cache with a larger block size.

## 6. Prefetch Overhead

The previous sections showed that, for most of the programs, spt-prefetch can reduce the number of misses in a cache. However, the results do not indicate how accurate, or the overhead cost to the cache, of the prefetch loads. A prefetch block creates overhead if it is loaded into the cache but never referenced, replaces some actively used block or replaces some prefetched block before it is used. Conversely, a prefetch block creates no overhead only if it removes the demand access miss that loads the memory block into the cache and does not replace useful demand accessed or prefetched data. A measure for the prefetch overhead is discussed in this section.

Let $CM_{np}$ be the number of misses in a no-prefetch cache, $CM_{pf}$ be the number of misses in a spt-prefetch cache and $PF$ the number of prefetch loads. If $PF = 0$ there is no prefetch. Let $PF_0 = (PF + CM_{pf}) - CM_{np}$ be the number of blocks prefetched but never referenced. If the cache has infinite capacity and prefetching is 100% accurate, $PF_0 = 0$ and $CM_{np} = PF + CM_{pf}$. If $PF_0 > 0$, this is the

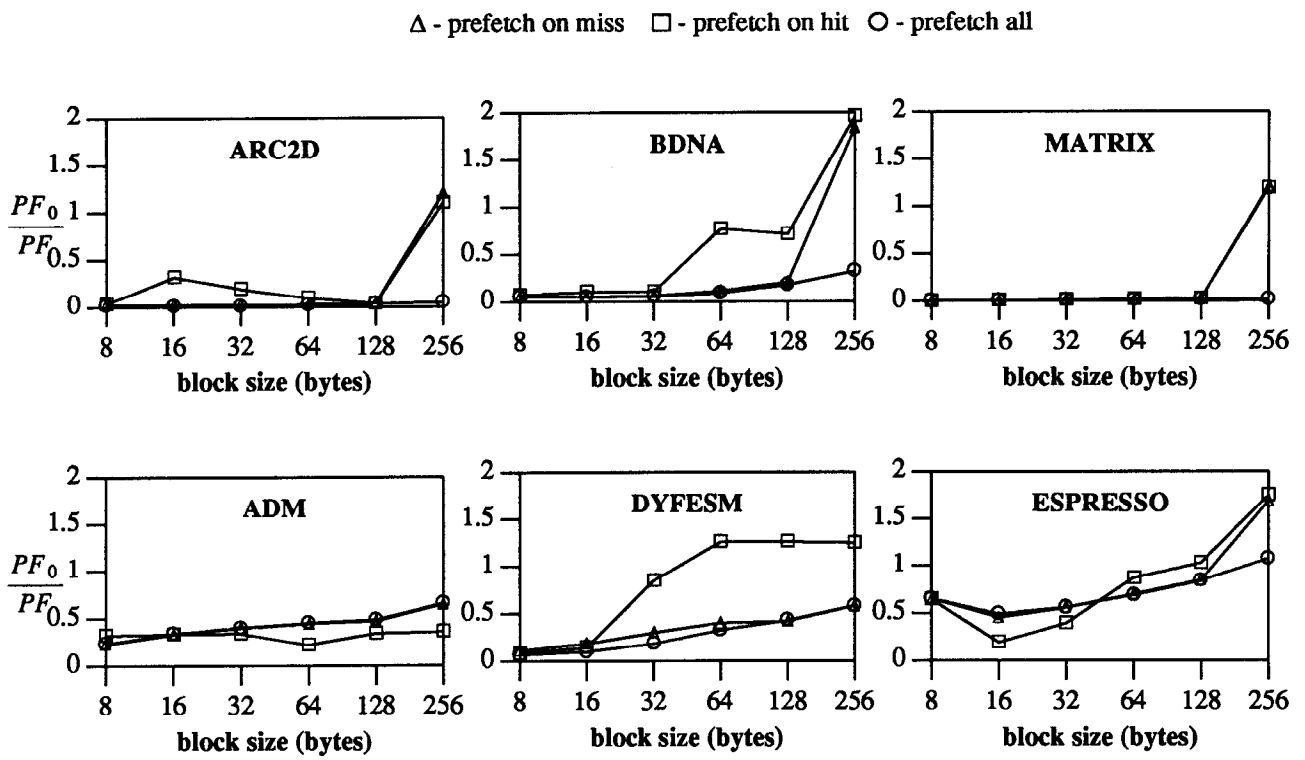


**Figure 5: Prefetch Overhead.**

number of prefetch blocks loaded into the cache that is never referenced i.e. the prefetch overhead cost for an infinite cache.

For a finite cache, $PF_0$ accounts for prefetch blocks that replaced useful demand blocks, replaced other useful prefetch blocks and useful prefetch blocks replaced by demand blocks or other prefetch blocks. The ratio $\frac{PF_0}{PF}$ is the fraction of the blocks loaded into the cache that does not reduce the misses in a no-prefetch cache and is used as a measure of the prefetch overhead cost. For example, let $CM_{np}=20$ misses and $PL=10$ prefetch loads. If 5 of the prefetch loads remove cache misses without causing additional useful blocks to be replaced, 2 prefetch loads replaced useful blocks but are not used and 3 prefetch loads have no effect on the cache miss rate, $CM_{pf}=17$ and $\frac{PF_0}{PF}=\frac{(17+10)-20}{10}=0.7$. This overhead factor of 0.7 indicates that the aggregate effect of the prefetch loads is 30% effective. Note that 5 prefetch loads were successful but because 2 prefetch loads were not used and replaced useful data the sum effect is only 3 misses were removed. The limit values for prefetch overhead are:

$$\frac{PF_0}{PF}\begin{cases}=0 & \textit{prefetched blocks used; no useful data replaced}\\ =1 & \textit{prefetched blocks not used; no effect on misses}\\ =2 & \textit{prefetched blocks not used; replaced useful data}\end{cases}$$

Figure 5 shows the results of $\frac{PF_0}{PF}$ for the three prefetch initiation schemes, pf_miss, pf_hit and pf_all, as a function of the block size. Recall that each time a prefetch is initiated only a single prefetch access is made and at most a single block loaded into a cache, if the prefetch access is a miss.

As expected the prefetch overheads for ARC2D, BDNA and MATRIX are very low. For block sizes of 128 bytes and less the prefetch overhead for pf_all and pf_hit for ARC2D and MATRIX are only 3 to 4%. The overhead for BDNA is slightly higher ranging from 5% for 8 bytes to about 20% with 128 bytes. For 256 bytes, both pf_hit and pf_all have an overhead greater than one. As the block size is increased, the number of blocks in the cache is decreased and this increases the potential for conflicts in the cache. The result is that prefetch blocks either replace useful data or are being replaced before being referenced.

The programs ADM, DYFESM and ESPRESSO show one of the problems in prefetching into a cache. The performance in Figure 5 shows that prefetching does reduce the cache miss ratio but the overhead result shows that a high percentage of the prefetch loads do not contribute to this reduction. For example, in ESPRESSO (block size 8 bytes) 60% of the prefetch loads do not get used. This means that to achieve the desired cache performance the number of prefetch memory requests made is nearly twice the number of demand memory requests removed. For some systems this may not be a good tradeoff.

## 7. Finite SPT Entries

The previous sections assumed a SPT with an infinite number of entries. This section presents some cache performance results with the SPT limited to 64 to 1K entries. Figure 6 shows the cache performance of the prefetch cache as a function of the block size for different SPT sizes for ARC2D and BDNA. Table 3 shows the fraction of memory references that hit in the spt (*spt hit*), the fraction of spt hits that cause a stride change (*stride change*), and the fraction of references that result in a prefetch attempt (*prefetch attempts*). A spt hit means the instruction address was previously loaded into the SPT but a prefetch is attempted only if the stride is greater than zero. Recall that the maximum number of SPT entries required for each program is shown in Table 1. For all SPT sizes, the SPT is organized as a direct mapped cache.

Not surprisingly, a smaller SPT size reduces the effectiveness of the prefetch by reducing the number of hits to the spt and the number of prefetches attempted. However, the results also show that a SPT can be quite small to be effective. A SPT with 1K entries result in a performance quite close to a infinite SPT particularly for ARC2D. Even when 256 entries are used, over 90% of the memory references are hitting in the SPT. This is because the SPT only needs to be large enough to hold the memory instructions of the most executed loop to be effective. Furthermore, as with an instruction cache, the sequential nature of instruction addresses means direct mapping is quite effective. As the program moves from one loop to another loop, the SPT is loaded with *IA* and *MA* pairs and only 1 loop traversal is necessary for the SPT to become effective.

| entries | ARC2D | | | BDNA | | |
|---|---|---|---|---|---|---|
| | spt hit | stride change | prefetch attempts | spt hit | stride change | prefetch attempts |
| 64 | 0.578 | 0.0004 | 0.135 | 0.107 | 0.0126 | 0.017 |
| 128 | 0.774 | 0.0003 | 0.167 | 0.280 | 0.0080 | 0.045 |
| 256 | 0.902 | 0.0003 | 0.202 | 0.544 | 0.0047 | 0.067 |
| 512 | 0.976 | 0.0003 | 0.217 | 0.719 | 0.0043 | 0.085 |
| 1K | 0.999 | 0.0001 | 0.221 | 0.846 | 0.0072 | 0.096 |
| ∞ | 1.000 | 0.0001 | 0.221 | 1.000 | 0.0070 | 0.104 |

**Table 3: SPT hit, stride change and prefetch attempts.**

BDNA has a lower spt hit rate than ARC2D indicating that it has longer loops. With 1K entries, only 85% of the references are hitting in the SPT and this reduces to 10% when the SPT is restricted to 64 entries. The number of prefetch attempts made by BDNA is substantially lower than ARC2D. For example, with 64 SPT entries, 24% of the references that hit in the SPT for ARC2D attempt a prefetch but only 16% of the spt hits in BDNA attempt a prefetch. Since a prefetch is always attempted after a spt hit, unless the calculated stride is a zero, this indicates that BDNA has more accesses with zero strides than ARC2D.

A stride change occurs whenever the calculated stride is not the same as the stride previously calculated. In the simulation, a stride change does not prevent a prefetch attempt but a count is maintained of how often this occurs. The fraction of the references that hit in the spt and cause the stride to change is shown in Table 3. For an infinite SPT both ARC2D and BDNA show that less than 1% of the references cause the stride to change. ARC2D shows significantly less stride changes than BDNA. When the SPT size is restricted the fraction of the references causing stride changes increase. This increase is because the effect of limiting SPT entries is to reduce the number of spt hits. Since there are more non stride changing spt hits, these are more likely to be removed, and stride changing spt hits become a larger fraction of total spt hits. For example, limiting the SPT to 64 entries, the fraction of stride changing spt hits for BDNA increases to 12%.

While stride change detection can be used to prevent unnecessary prefetches, the results in Section 4 and 5 indicate that limited prefetching may provide little benefit. It may be more prudent to simply not use prefetching for programs with a large number of stride changes. One possible approach is to identify such programs before run time. For example, compiler profiling could be used to determine if a program is likely to cause a large number of stride changes.

## 8. Conclusions

This paper has presented a method for predicting the stride of array or vector accesses made from within the loop body of a numerical program executing on a scalar processor. Using these strides to direct prefetching, into the cache, the results show that for programs that can be highly vectorized this method can significantly reduce the number of cache misses with low overhead. However, for other programs the cache miss ratio can be reduced but generally with more significant overhead. Stride directed prefetching may not be appropriate for these programs. The results also show that a SPT can have a small number of entries to be effective. The SPT only needs to be large enough to capture the memory referencing instructions of the longest program loop.

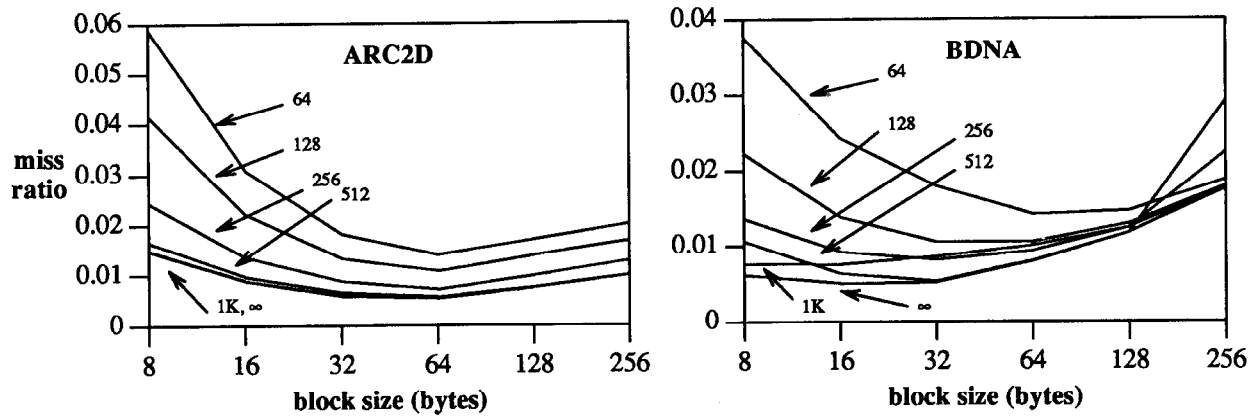


**Figure 6: Cache performance for various SPT sizes.**

A number of questions remain unanswered. The problem of system interaction and overhead needs to be further considered, as does the engineering aspects of adapting and organizing a cache for prefetching. The prefetch scheme is very successful for particular programs and poor for others. This provides an opportunity to identify such programs at compile time. For example, compiler profiling can be used to determine whether prefetching is likely to be profitable, for a particular program, by computing its loop characteristics.


## ACKNOWLEDGEMENTS

The Authors would like to thank A. N. L. Reddy of IBM Almaden, Hoichi Cheong and Bob Dimpsey of IBM Austin and Nancy Warter of University of Illinois for many useful discussions during this research. We would also like to acknowlege the useful constructive comments from the referees, particularly that of referee D.


## REFERENCES


[BaCh91] J-L Baer and T-F Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proc. Supercomputing 1991*, 1991.

[BoKW90] A. Borg, R. E. Kessler, D. W. Wall, "Generation and Analysis of Very Long Address Traces", *Proc. 17th. Int'l. Symp. on Comp. Arch.*, pp. 270-279, May 1990.

[Berr89] M. Berry, et. al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Int'l. Journal for Supercomputer Applications*, Fall 1989.

[Fu92] J. W. C. Fu, "Performance Evaluation of Memory Systems for High Speed Computers," Tech. Rpt. CRHC-92-10, Center for Reliable and High Performance Computing, University of Illinois, July 1992.

[FuPa91] J. W. C. Fu and J. H. Patel, "Prefetching in Multiprocessor Vector Cache Memories", *Proc. of 18th. Int'l Symp. on Comp. Arch.*, pp. 54-63.

[Jans91] B. L. Janssens, "Generation of Multiprocessor Address Traces and their use in the Performance Analysis of Cache-based Error Recovery Methods," Tech. Rpt. CRHC-91-10, Center for Reliable and High Performance Computing, University of Illinois, May 1991.

[Przy90] S. Przybylski, "The Performance Impact of Block Size and Fetch Strategies," *Proc. 17th. Ann. Int'l. Symp. on Comp. Arch.*, pp 160-169, June 1990.

[Skle92] I. Sklenar, "Prefetch Unit for Vector Operations on Scalar Computers," *Proc. 19th. Ann. Int'l. Symp. on Comp. Arch. (poster)*, pp 430, June 1992.

[Smit82] A. J. Smith, "Cache Memories," *ACM Comp. Surveys*, vol 18, no. 3, pp 473-530, Sept. 1982.

[StFu89] C. B. Stunkel and W. K. Fuchs, "TRAPEDS: Producing traces for multicomputers via execution driven simulation", *Proc. ACM Sigmetrics Conf. on Measurement and Modeling of Compter Systems*,pp. 70-78, May 1989.