A Dynamic Storage Scheme for Conflict-Free Vector Access

D. T. Harper III* and D. A. Linebarger

Erik Jonsson School of Engineering and Computer Science
The University of Texas at Dallas, Mail Stop MP-32
Richardson, Texas 75083 USA
(214) 690–2893
dth@engc1.dal.utexas.edu

Abstract

Previous investigations into data storage schemes have focused on finding a storage scheme that permits conflict-free access for a set of frequently encountered access patterns. This paper considers an alternative approach. Rather than forcing a single storage scheme to be used for all access patterns, conflict-free accesses of any constant stride can be made by selecting a storage scheme for each vector based on the accessing patterns used with that vector.

By factoring the stride into two components, one a power of 2 and the other relatively prime to 2, a storage scheme can be synthesized which allows conflict-free access to the vector using the specified stride. All such schemes are based on a variation of the row rotation mechanism proposed by Budnik and Kuck[1]. Each storage scheme is based on two parameters, one describes the type of rotation to perform and the other describes the amount of memory to be rotated as a single block. Hardware required to implement this storage scheme is efficient.

The performance of the memory under access strides other than the stride used to specify the storage scheme is also considered. This models a vector being accessed with multiple strides, in particular the row/column access of a matrix, and situations when the stride can not be determined prior to initializing the vector. Simulation results show that if a single buffer is added to each memory port then the average performance of the dynamic scheme surpasses that of the interleaved scheme for arbitrary stride accesses.

For dynamic storage schemes to be effective, the compiler must be able to detect information about the stride of vector accesses. In general, this is within the capabilities of current vectorizing compilers. Dynamic storage schemes also may allow more flexibility in program transformations performed by vectorizing compilers during optimization.

Introduction

The performance of heavily pipelined vector supercomputers is highly dependent upon the architecture's ability to rapidly move data in and out of main memory. Because processor speeds are substantially higher than memory speeds, it has been necessary to develop architectural features to support parallelism in the memory subsystems. These features include superword accesses and parallel memory modules. This paper considers a problem associated with the use of parallel memory modules.

The topic of vector accesses in parallel memory systems has been studied by many previous investigators. Most have centered their investigation on the question of the mapping between addresses and physical storage locations in the memory. The algorithm used to describe this mapping is often referred to as the *storage scheme* of the system. The simplest storage scheme is the low-order interleaved (also referred to simply as interleaved) scheme which maps address a into memory module $a \bmod N$, where N is the number of memory modules in the system. This scheme, shown in figure 1, has been successfully implemented in many high-performance systems. In the following discussion, memory is viewed as a two dimensional matrix consisting of *rows* and *modules*. The term *module address* refers to the module number into which the address is mapped and the term *row address* refers to the position of the storage location relative to the beginning of the module. For example, in figure 1 the row address of element 9 is 2 (addressing begins at 0) and the module address of 9 is 1.

M_0	M_1	M_2	M_3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

Figure 1: Low-order interleaved storage scheme.

The rationale for using the interleaved scheme is two-fold:

1. A parallel memory system allows concurrent access to multiple data items by placing items which can be used in parallel in different modules. If this can be done such that N items can be accessed in a single memory cycle, a *conflict-free* access is said to have occurred. The assumption is made that addresses can be presented to all of the modules in parallel and that, after a delay equal to the cycle time of the memory, data can be removed in parallel from all of the modules. Previous work discussing features of interconnection networks capable of this has been presented by several authors [2] [3][4]. One of the most common accessing patterns, particularly in vector operations, is sequential access, in which the first datum needed is found at address a, the next at address $a + 1$, *etc.* In this case, the interleaved scheme permits conflict-free access.
2. The interleaved scheme is a simple mapping. Parallel memory systems usually use an address composed of two fields. One field,

* This work was supported by Texas Advanced Research Program grant number 10897-701.

the module address, determines which module contains the physical storage location and the other field, the row address, determines the offset of the storage location within the module. Each of these fields must be computed from the original address. When N is of the form $N = 2^n$, for some non-negative integer n, this computation is trivial for the interleaved scheme. The n low-order bits of the address form the module address field and the remaining high-order bits form the row address.

Unfortunately, interleaved schemes do not perform as well when other accessing patterns are encountered. The *stride* of an access is defined to be the distance between successive addresses; this paper only considers access patterns with constant strides. The sequential access pattern is said to be a stride 1 access. A stride 2 access references alternate addresses, stride 3 accesses reference every third address, *etc.* The interleaved scheme provides conflict-free access as long as the stride, S, is relatively prime to N. If S and N are not relatively prime, *collisions* occur in the memory when a reference is mapped to a module which is busy processing (reading or writing) a previous reference. This can cause substantial degradation in the performance of the system [5].

One solution to this problem is to *skew* the storage scheme of the system. This was proposed by Budnik and Kuck [1] and maps addresses to physical locations as shown in figure 2. Several authors have proposed storage schemes which are variants on figure 2. These schemes are collectively referred to as skewing schemes in the literature. Shapiro [6] presents a classification of these schemes and several investigations have considered properties of such schemes [7] [8]. In this paper, the term skewed storage scheme refers to the scheme shown in figure 2.

M_0	M_1	M_2	M_3
0	1	2	3
7	4	5	6
10	11	8	9
13	14	15	12
16	17	18	19
23	20	21	22
26	27	24	25
29	30	31	28

Figure 2: Skewed storage scheme

In figure 2, address a is mapped to module $(a + \lfloor \frac{a}{N} \rfloor) \bmod N$. As noted in [1], this mapping allows conflict-free access for strides which were not conflict-free in the interleaved scheme. If N is prime, a larger set of conflict-free patterns results than when N is composite. This fact was used in the design of the Burroughs Scientific Processor [9]. However, the use of a prime number of modules has several disadvantages, the primary one being that the address to storage location mapping becomes computationally expensive [10][11]. To deal with these problems a different approach to achieving conflict-free accesses is proposed.

Dynamic Storage Schemes

To date, investigations into storage schemes have discussed the use of a single scheme for all memory accesses. For a system where the number of memory modules in the system is given by $N = 2^n$, no single scheme has been found which allows conflict-free access to vectors for all strides. For this reason, it is proposed that multiple storage schemes be used within a single system. The scheme to be used with each vector is chosen dynamically, and is based on the the system's perception of the vector's access patterns. Deciding what stride will be used to access a vector can be done by current vectorizing compilers. The following sections show how to synthesize a storage scheme for conflict-free accesses to the vector once the access stride is known.

To achieve this goal, several problems must be solved. A family of storage schemes, which collectively permit a vector to be accessed with both stride 1 (to initially load the vector) and the stride required to operate on the specified subset of the vector, must be identified. Second, a mapping must be found which determines the appropriate storage scheme to use for each stride. Finally, hardware support must be provided for the use of multiple storage schemes. In concept, the use of multiple storage schemes should not be seen as a disadvantage. Several vector architectures force the stride of an access to be defined in a processor register before a vector reference occurs. The specification of a storage scheme, discussed in detail later, is compact and can be done concurrently.

Conflict-Free Storage Schemes

The goal of this section is to develop a family of stride-dependent or dynamic storage schemes which allow conflict-free access to a vector using any predetermined stride. Let S be the stride used to access the vector and let $N = 2^n$ be the number of memory modules in the system.

Development of the storage scheme begins by considering the interleaved scheme as shown in figure 1. It was seen that the performance of the interleaved scheme degraded when S was not relatively prime to N. In order to see the relationship between S and N, it is useful to consider a particular factorization of S.

Let $S = \sigma 2^s$ for $(\sigma, 2) = 1$; the notation $(\sigma, 2)$ represents the greatest common factor of σ and 2. This factorization exists and is unique for all integers $S > 0$. For the moment, assume that $\sigma = 1$. This means that S is of the form $S = 2^s$. The situation for $\sigma > 1$ will be discussed later.

If $S = 2^s$, it is useful to partition the set of strides into two categories; strides which have at least one access per row and strides which have fewer than one access per row (accesses do not occur in every row). The first case occurs when $S \leq N$, or when $s \leq n$. The second case occurs when $s > n$.

Case 1, $s \leq n$:

The following discussion assumes an initial access to the first element of a vector. This element is stored in module 0. No loss of generality results from this assumption.

Assuming an interleaved scheme, the sequence of module addresses is: $0, 2^s, ..., 2^{n-s-1}2^s$. After 2^{n-s} references the sequence repeats since $2^{n-s}2^s$ is congruent modulo N to 0. In the interleaved scheme the $(2^{n-s})^{th}$ access falls in module 0 of the row following the initial row. This causes a collision with the 0^{th} access. However, if the second row is skewed relative to the initial row then rather than referencing module 0, the $(2^{n-s})^{th}$ access references module 1. Skewing the second row relative to the first allows $2 \cdot 2^{n-s}$ accesses to be made before a conflict occurs. Each reference in the second 2^{n-s} accesses goes to the module adjacent to the module of the corresponding reference in the first 2^{n-s} accesses. To permit N accesses to occur before a conflict occurs (the condition for conflict-free accesses), it is necessary to skew 2^s rows relative to each other. Therefore, a conflict-free storage scheme for any stride $S = 2^s$, $s \leq n$, can be generated by circularly rotating each row. Row r should be rotated $r \bmod 2^s$ places relative to its state in the interleaved scheme. Figure 3 shows an example of a dynamic storage scheme for $N = 8$ and $S = 4$.

Case 2, $s > n$:

In this case the situation is somewhat different: there is a maximum of one access per row. For convenience and without loss of generality, it is assumed that the initial module address is again 0. The sequence of modules referenced is: $0, 2^s, 2 \cdot 2^s, ...$ Since $2^s = k2^n = kN$, it is clear that all references fall in the same module and maximum performance degradation occurs. Again, the technique of row rotation is used to distribute references over all of the modules so that conflict-free accesses can be made. However, the rotation pattern is different from the previous case. This time, blocks of contiguous rows will be rotated relative to preceding blocks. In particular, blocks of 2^{s-n} rows are rotated as single entities. Note that 2^{s-n} is exactly the number of rows between consecutive elements required in the stride S access. This rotation pattern serves to place consecutive accesses in adjacent modules. Over a period

M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	1	2	3	4	5	6	7
15	8	9	10	11	12	13	14
22	23	16	17	18	19	20	21
29	30	31	24	25	26	27	28
32	33	34	35	36	37	38	39
47	40	41	42	43	44	45	46
54	55	48	49	50	51	52	53

Figure 3: Dynamic storage scheme for $N = 8$ and $S = 4$.

M_0	M_1	M_2	M_3
0	1	2	3
4	5	6	7
11	8	9	10
15	12	13	14
18	19	16	17
22	23	20	21
25	26	27	24
29	30	31	28
32	33	34	35

Figure 4: Dynamic storage scheme for $N = 4$ and $S = 8$.

of N accesses each module is referenced exactly once and conflict-free access is obtained. An example of a 4 memory scheme configured for a stride 8 access is shown in figure 4.

The schemes described in cases 1 and 2 provide for conflict-free accessing for any stride of the form $S = 2^s$.

Now the restriction placed on σ is removed and strides of the form $S = \sigma 2^s, (\sigma, 2) = 1$, are discussed. Assuming the schemes described in cases 1 and 2 have been applied, consider the sequence of module addresses generated by an access of stride $\hat{S} = 2^s$. Let this sequence be indicated by $\hat{M} = \hat{a_0}, \hat{a_1}, \ldots$ This sequence is periodic with a period of length N and each period contains exactly one address equal to i, $0 \leq i < N$. If an access is made from the same initial address but with a stride $S = \sigma\hat{S} = \sigma 2^s$, then the sequence of module addresses for this access, M, is formed by selecting every σ^{th} element from $\hat{M}$. That is, the i^{th} element of M is given by $a_i = \hat{a_{i_\sigma}}$. Because σ is relatively prime to N, any N consecutive elements of M will reference N different memory modules. Therefore, stride S accesses are conflict-free.

Stride to Storage Scheme Mapping

The solution to the problem of finding an algorithm which maps each stride into a storage scheme has been outlined in the previous section. In the cases above, the proposed storage schemes can be characterized using two parameters

The first parameter, w_{max}, indicates the maximum rotation a row can have relative to the first row. For the $s \geq n$ case, this parameter is always N. For the $s < n$ case, w_{max} is given by 2^s. Another way to view this parameter is that rotations are always performed modulo w_{max}.

$$w_{max} = min(2^s, N) \tag{1}$$

The second parameter, B, specifies the number of rows to rotate as a single block. When $s < n$, B is always 1; when $s \geq n$, B is equal to 2^{s-n}.

$$B = \lceil 2^{s-n} \rceil \tag{2}$$

Knowledge of these two parameters is sufficient to generate a conflict-free storage scheme for any constant stride vector access. Because both of the parameters are functions only of n and s, determination of n and s is also sufficient to generate the appropriate storage scheme.

Hardware Support

Providing hardware support so that a dynamic storage scheme can be implemented efficiently requires a mechanism to transform addresses into physical storage locations. To perform this transformation, the relationship of s to n must be known. Since n, the base 2 logarithm of the number of memory modules, is generally considered fixed and known, it is only necessary to compute s. This computation is simple when it is noted that s is merely the exponent of the largest power of 2 which is a factor of the stride; s can be found by dividing S by 2 until the remainder is non-zero. The hardware required to do this is a shift register which is initially loaded with the stride and performs right shifts. Shifting is continued until a 1 bit is shifted out of the register. At this point the number of shifts performed is $s + 1$.

To design hardware which implements the proper address transformation, it is useful to consider equations which describe the mappings. The notation $x_{i:j}$ indicates a bit-field of x which begins with bit position i and continues through bit position j. Bit 0 is the least significant position and bit $l - 1$ is the most significant. Equation (3) computes the module address in the case where $s < n$.

$$\begin{aligned} m(a) &= \left(a \bmod N + \left\lfloor \frac{a}{N} \right\rfloor \bmod 2^s\right) \bmod N \\ &= \left(a \bmod 2^n + \left\lfloor \frac{a}{2^n} \right\rfloor \bmod 2^s\right) \bmod 2^n \\ &= (a_{0:n-1} + a_{n:l-1} \bmod 2^s) \bmod 2^n \\ &= (a_{0:n-1} + a_{n:n+s-1}) \bmod 2^n \end{aligned} \tag{3}$$

Because the modulo and division operations involve only power of two divisors, they can be implemented inexpensively by masking and shifting operations. If these operations are combined and reduced, the hardware required to evaluate equation (3) is simply an n-bit adder, as shown in figure 5.

Equation (4) computes the module address when $s \geq n$. Again, the hardware required to implement this equation is simply an n-bit adder as shown in figure 5 combined with a circuit to extract the $s : s+n-1$ field from a. Unfortunately, the field selection circuit introduces some hardware complexity. Because the address bits used to compute the address are determined by the storage scheme of the vector being referenced, provision must be made to gate any n bit field of the address to the adder. This can be implemented using an array of pass transistors $(n \times l)$ so that the address delay is minimized. Note that the field selection hardware is required for any dynamic address calculations based on the stride of the access.

$$\begin{aligned} m(a) &= \left(a \bmod N + \left\lfloor \frac{\lfloor \frac{a}{N} \rfloor}{2^{s-n}} \right\rfloor \bmod N\right) \bmod N \\ &= \left(a_{0:n-1} + \left\lfloor \frac{a_{n:l-1}}{2^{s-n}} \right\rfloor \bmod 2^n\right) \bmod 2^n \\ &= (a_{0:n-1} + a_{n+(s-n):l-1} \bmod 2^n) \bmod 2^n \\ &= (a_{0:n-1} + a_{s:s+n-1}) \bmod 2^n \end{aligned} \tag{4}$$

The efficiency of the mapping hardware allows it to be used in high-performance architectures. An alternative implementation when n and s are small is to use a ROM to perform a table lookup similar to the address mapping circuit used in the IBM RP3 architecture [12].

Multiple Stride Accesses

To this point, it has been tacitly assumed that a vector is only accessed with a single stride. Examination of applications programs from a variety of disciplines has shown that while vectors are not

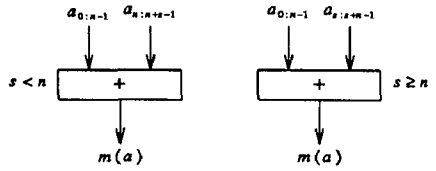

Figure 5: Address mapping hardware.

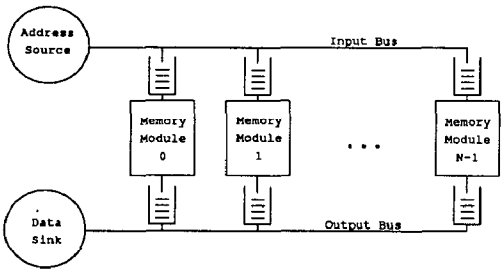

Figure 6: Architectural model.

always accessed with a single stride, single stride accesses are by far the most common case. The primary exception to this is the row/column accessing patterns found in matrix manipulation routines. However, since operations on matrices compose a substantial portion of numerical computing it is important to determine how dynamic storage schemes perform on multiple stride accesses. Three questions are considered.

1. What form should the storage scheme take when a vector is accessed with more than one stride?
2. How well does the memory system perform under multiple stride accesses?
3. Array reshaping is a simpler way of reducing conflicts. Why should dynamic storage schemes be used instead?

Consider the first and second questions which are closely related. Let a vector be accessed with two strides, $S_1 = \sigma_1 2^{s_1}$ and $S_2 = \sigma_2 2^{s_2}$. The vector should be stored as if it were to be accessed with a stride of $S_{stored} = 2^{max(s_1,s_2)}$. The argument for this is straight-forward. Assume, without loss of generality that $2^{s_1} < 2^{s_2}$. In this case, the vector should be stored so conflict-free accesses can be made for S_2. Although conflict-free access for S_1 does not occur, all modules are referenced during the S_1 access and buffers, to be discussed, can be used to maintain system throughput. Suppose instead, the vector is stored using stride $S_{stored} = 2^{s_1}$. S_2 is now an even multiple of S_{stored}; this means that not all memory modules will be referenced. The number of modules referenced is given by $N/2^{s_2-s_1}$. Reducing the number of modules referenced may severely degrade the performance of the memory system.

More quantitative results have been obtained using simulation. Figure 6 shows the architectural model used in the simulations.

Addresses are presented to the memory system sequentially on the input bus. If the referenced module is not busy, the address is latched and the bus is released for subsequent address transfers. If the memory is busy, the bus blocks until the module becomes available. Data read from the module is transferred to the processor over the output bus. Data is guaranteed to return to the processor in the order requested by tagging addresses as they cross the input bus and then ordering access to the output bus based on these tags. This technique is simple and avoids the need for centralized bus control. Time is normalized to the bus transfer time and it is assumed that the memory cycle time is equal to the N. If the elements of the vector are distributed optimally over the modules then the cycle time of each module is overlapped with address transfers to the other modules and the memory system as a whole operates with an effective cycle time of 1.0. This is shown for $N = 4$ and $S = 1$ in figure 7.

Figure 8 shows the performance of an $N = 8$ module system in which the vector has been stored to optimize stride 16 accesses ($S_{stored} = 16$). The figure shows the performance of accesses as the

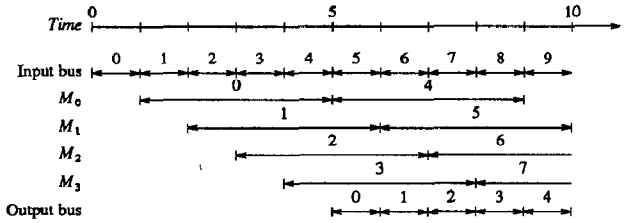

Figure 7: Memory system timing.

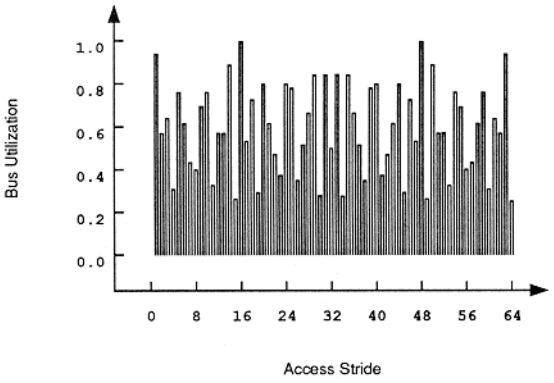

Figure 8: Memory system performance, $N = 8$, $S_{stored} = 16$.

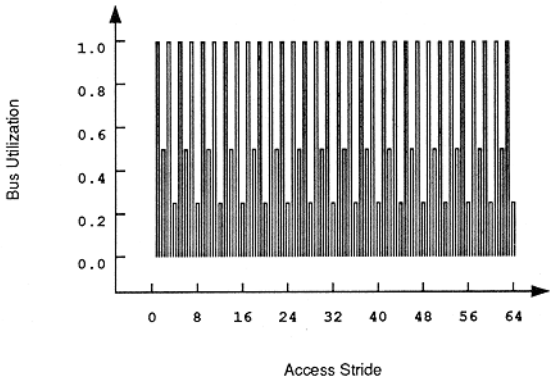

Figure 9: Memory system performance, $N = 8$, Interleaved.

stride is varied from 1 to $N \cdot S_{stored}$; the periodic nature of the storage scheme causes all strides to be congruent modulo $N \cdot S_{stored}$. Utilization of the data bus is used as the measure of memory system performance. Data bus utilization is calculated by:

$$U_{data\ bus} = \frac{L}{t_{stop} - t_{start}} \tag{5}$$

where L is the number of elements accessed in the vector, t_{start} is the time the address of the first element is loaded onto the address bus, and t_{stop} is the time the last element is removed from the data bus. This measure reflects both memory throughput and latency. Because the bus speed is matched to the memory, $1/U_{databus}$ is the effective cycle time of the memory. The effective cycle time is a measure of the memory throughput. By measuring time from the initial address bus cycle to the final data bus cycle, increased latency introduced by buffering (to be discussed later) is also measured.

Comparison of figure 8 with similar data for an interleaved storage scheme (figure 9) shows the interleaved scheme performs 16% better if a uniform distribution of strides is assumed.

A more detailed investigation finds that this is due to transient non-uniformities in the address sequence. In these cases, the number of references to each module is uniform, but the references to a given module are not spaced evenly in time. This effect was noted by Harper and Jump [10] who proposed the use of queues at the input and output of the memory modules to buffer the transients in the address stream. Figure 10 shows the performance of the dynamic scheme if one additional register is added to the input and output of each module (the parameter q indicates the number of buffers on each memory port).

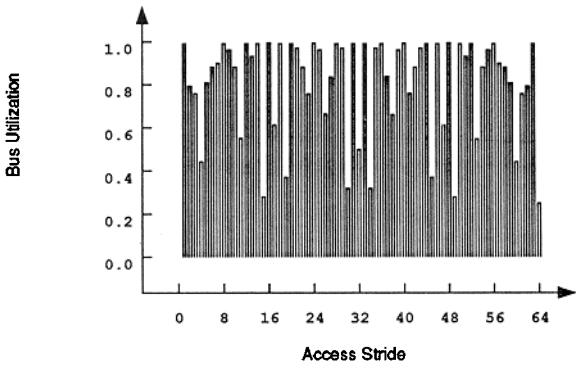

Figure 10: Memory system performance, $N = 8$, $S_{stored} = 16$, $q = 2$.

The dynamic scheme now outperforms the interleaved scheme by 18% and additional performance can be gained by adding more buffers as shown in table A, however only small performance gains are realized by adding buffers beyond two or three. The left column of table A is the number of buffers present at each memory port. The next column to the right shows memory performance averaged over all strides. Next is the change in performance due to the addition of another buffer. The right column shows the performance change as a percentage. Simulation data indicates that the performance gain per additional buffer is relatively insensitive to both N and S_{stored}. It should be noted that the relative performances are dependent on S_{stored} and the particular strides used. The dynamic scheme takes advantage of the fact that the stride of the access is known and optimizes performance for that access. This means that the assumption of uniform stride distribution produces a very conservative estimate of the performance of the dynamic storage scheme.

# Buffers	$\overline{U}_{databus}$	$\Delta\overline{U}$	% Δ
Interleaved	0.685	-	-
1	0.592	-	-
2	0.808	0.216	36.5
3	0.891	0.083	10.3
4	0.923	0.032	3.3
5	0.938	0.015	1.6
∞	0.968	-	-

Table A: Data bus utilization vs. buffer length, $S_{stored} = 16$.

Using dynamic storage schemes is a better alternative than array reshaping for several reasons. First, array reshaping by embedding an array inside a larger array wastes space and may not be possible for large data structures. Also, the "gaps" in the new matrix which do not contain data from the original matrix may inhibit vectorization due to non-constant stride considerations. Second, using a dynamic storage scheme permits *run-time* decisions to be made about access strides. The choice of storage scheme to be used may be deferred until the data in the vector is defined. For loop structures which have data dependent indices, this can be a major advantage.

An alternative storage scheme might also be considered which, rather than adding the two address fields together to compute the module address, the module address is computed simply by using different bits of the address to determine the module number. The storage scheme achieved using this technique suffers when multiple stride accesses are made to a vector, particularly in the common row/column type access. Figure 11 shows simulation data obtained to compare the two schemes in two multiple stride cases. The first type of access assumes that all strides are equally likely and performance is measured as an unweighted average of the performance over all strides. This data is labeled using a $\times$ and is equal for both schemes. The second case models the row/column access of a matrix where the strides used are 1 and 2^s (the size of the matrix). This data is labeled with a $+$ for the proposed scheme with the adder and a Δ for the scheme without the adder. The dotted line indicates buffering of size 1, the solid line is for size two buffers. The figure plots bus utilization as a function of the access stride used. It is assumed that both storage schemes are optimized for the matrix being accessed.

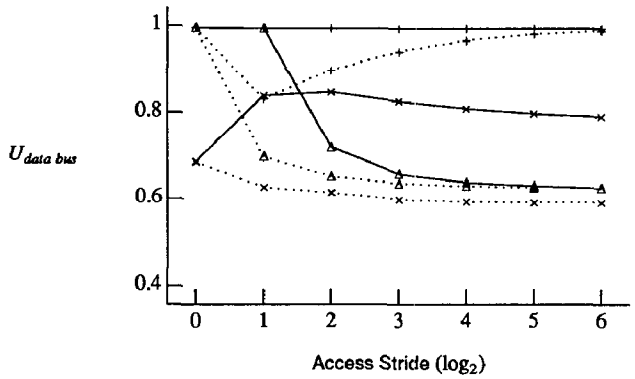

Figure 11: Comparison of dynamic storage schemes, $N = 4$.

When a single buffer is used, the performance of the averaged case is equal for both schemes. However, when row and column accesses are considered, the proposed scheme is far superior due to its increased performance on stride 1 accesses. If a single additional buffer is added, the performance of the proposed scheme is substantially improved while the performance of the simple scheme remains constant.

Compiler - Architecture Interaction

One premise of this work is that a vectorizing compiler can determine the stride of a vector access. Discussions with several people actively involved with vectorizing compiler projects have indicated that determination of the access stride is generally within the capabilities of the compiler. Because the factorization of stride and the mapping configuration are done in hardware, there is little program overhead in initiating a vector access.

The dynamic storage scheme also provides advantages to vectorizing compilers performing loop optimization. Cowell and Thompson [13] and others have described methods of transforming loop structures in programs into vector operations. In the process of doing so the stride of the vector access may change from the stride specified by the programmer. If the dynamic storage scheme is used in the memory system then the transformation will not cause memory conflicts to occur because the storage scheme is simply optimized for the new stride. This can be particularly advantageous when transformations such as loop interchange and strip mining are used. The fact that any stride can be conflict-free also frees the applications programmer from coding information about the memory architecture into the program. Instead the programmer is able to concentrate on a natural expression of the algorithm. For example, FORTRAN programmers do not need to force column-major matrix accesses into their algorithms. Row-major accesses will perform equally well.

Conclusions

Results presented in this paper show how memory conflicts can be eliminated during vector accesses if the stride of the access is known. This is achieved by designing a dynamic storage scheme. The scheme

does not rely on having a prime number of memory modules in the system as do many other proposed schemes.

Previous investigations into storage schemes have identified a set of commonly used vector access patterns and then attempted to synthesize a storage scheme which permitted conflict-free access to these patterns. The dynamic storage scheme presented here is a departure from the previous approach. Rather than design a single scheme for all access patterns, the storage scheme for each vector is adapted to the pattern which will be used with that vector. Taking this approach allows any vector to be accessed conflict-free.

To synthesize a storage scheme which allows conflict-free access to a vector, the stride of the access is factored into two components. One of the factors is a power of 2, the other is relatively prime to 2. Based on the value of the first factor, two parameters are chosen. These parameters are sufficient to specify the storage scheme. Hardware was presented to implement the stride factoring and the address transformation. The expense of the hardware was shown to be small.

Accesses with multiple strides to the same vector were also considered. Through the use of one or two additional buffers at each memory port it was shown that the dynamic storage scheme performs substantially better than interleaved memory. This is true even when accesses are made with strides other than those for which the memory scheme has been optimized.

Finally, it was noted that the use of dynamic storage schemes relies on the compiler to determine the stride of the vector access. This is within the capabilities of current compilers. It was also noted that the use of dynamic storage schemes will allow a compiler more flexibility in its choice of program transformations for code optimization because it no longer must maintain stride 1 access patterns or face memory performance degradation.

Because the additional hardware cost of dynamic storage schemes is so small, the use of dynamic storage schemes is an attractive method of boosting memory bandwidth during vector accesses. The hardware implementation is a simple extension of memory architectures found on current high-performance machines but memory performance is significantly improved relative to conventional interleaving.

References

[1] P. Budnik and D. Kuck, "The organization and use of parallel memories," *IEEE Trans. Computers*, vol. C-20, no. 12, pp. 1566–1569, December 1971.

[2] D. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Computers*, vol. C-24, no. 12, pp. 1145–1155, December 1975.

[3] K. Batcher, "The multidimensional access memory in STARAN," *IEEE Trans. Computers*, vol. C-26, pp. 174–177, February 1977.

[4] R. Swanson, "Interconnections for parallel memories to unscramble p-ordered vectors," *IEEE Trans. Computers*, vol. C-23, pp. 1105–1115, November 1974.

[5] W. Oed and O. Lange, "On the effective bandwidth of interleaved memories in vector processing systems," *IEEE Trans. Computers*, vol. C-34, no. 10, pp. 949–957, October 1985.

[6] H. Shapiro, "Theoretical limitations on the efficient use of parallel memories," *IEEE Trans. Computers*, vol. C-27, no. 5, pp. 421–428, May 1978.

[7] H. Wijshoff and J. van Leeuwen, "The structure of periodic storage schemes for parallel memories," *IEEE Trans. Computers*, vol. C-34, no. 6, pp. 501–505, June 1985.

[8] H. Wijshoff and J. van Leeuwen, "On linear skewing schemes and d-ordered vectors," *IEEE Trans. Computers*, vol. C-36, no. 2, pp. 233–239, February 1987.

[9] D. Lawrie and C. Vora, "The prime memory system for array access," *IEEE Trans. Computers*, vol. C-31, no. 5, pp. 435–442, May 1982.

[10] D. T. Harper III and J. R. Jump, "Vector access performance in parallel memories using a skewed storage scheme," *IEEE Trans. Computers*, vol. C-36, no. 12, pp. 1440–1449, 1987.

[11] A. Ranade, "Interconnection networks and parallel memory organizations for array processing," *Int. Conf. on Parallel. Proc.*, pp. 41–47, August 1985.

[12] A. Norton and E. Melton, "A class of boolean linear transformations for conflict-free power-of-two stride access," *Int. Conf. on Parallel. Proc.*, pp. 247–254, 1987.

[13] W. R. Cowell and C. P. Thompson, "Transforming Fortran DO Loops to Improve Performance on Vector Architectures," *ACM Transactions on Mathematical Software*, vol. 12, pp. 324–353, December 1986.