Module Partitioning and Interlaced Data Placement Schemes to Reduce Conflicts in Interleaved Memories

Lizyamma Kurian and Bermjae Choi
Computer Science and Engineering
University of South Florida
Tampa, FL 33620
kurian@sol.csee.usf.edu

Paul T. Hulina and Lee D. Coraor
Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
pth@ecl.psu.edu

Abstract

In interleaved memories, interference between concurrently active vector streams results in memory bank conflicts and reduced bandwidth. In this paper, we present two schemes for reducing inter-vector interference. First, we propose a memory module partitioning technique in which disjoint access sets are created for each of the concurrent vectors. Various properties of the involved address mapping are presented. Then we present an interlaced data placement scheme, where the simultaneously accessed vectors are interlaced and stored to the memory. Performance of the two schemes are evaluated by trace driven simulation. It is observed that the schemes have significant merit in reducing the interference in interleaved memories and increasing the effective memory bandwidth. The schemes are applicable to memory systems for superscalar processors, vector supercomputers and parallel processors.

Keywords: Address Distribution, Conflict-free access, Vector Interference, Memory Storage Patterns, Memory Bank Conflicts, Memory Bandwidth.

1 Introduction

Processor speeds have increased tremendously in the past few years, but memory systems have barely kept pace, widening the speed disparity between processors and memory systems. Standard memory technology is not scaling with current processor technology [12]. The memory latency in the first VAX microprocessor was 2-3 processor cycles, while in a recent one, it is 60 processor cycles [12]. While effective use of memory has been an important focus for computer systems development for thirty years, advances in memory performance greatly lag the rate of increase in system processing speed and the rate of increase of application memory use [12]. Despite efforts from the computer architecture community and researchers in semiconductor technology, the disparity between processor and memory speeds is growing.

Multiple instruction-issue processors exploiting the fine-grain parallelism in the instruction stream are emerging to be very popular. By the middle of this decade, we expect processors that attempt to issue about ten instructions per cycle to be within the realm of possibility [18]. As the instruction issuing capabilities of processors are improved to allow the issue of several instructions per cycle, the bandwidth provided by existing memory systems is posing a dangerous and critical bottleneck. The bandwidth problem in supercomputers such as the Cray machines was typically solved by using memory interleaving, but interleaving has been almost limited to vector processors. Bandwidth was not a major concern in processors issuing one or two instructions per cycle and no major research efforts had gone into designing high bandwidth memories for scalar processors except [18]. More recently, there has been a higher research emphasis on high bandwidth memory systems for modern multiple-issue processors [22] [11].

1.1 Background and Significance

The bandwidth crisis that arises in pipelined processors issuing multiple instructions per cycle is investigated in this section. The peak instruction issue of a superscalar processor is limited to $\frac{BW_S}{f_M}$ instructions per cycle, where BW_S is the bandwidth that the data memory can supply, and f_M is the fraction of all instructions that are loads and stores [18]. In Cmelik et al.'s instruction stream characterization [3], for SPEC floating point programs, 46% of all instructions on the MIPS and 36.8% of all instructions on the SPARC are memory reference instructions. Some results from our own profiling studies [10] using the SPEC benchmarks, on the DEC 5000 workstation with C compiler CC 2.1 and Fortran compiler version 3, are presented in Table 1. The instruction issue bound has been calculated assuming a peak bandwidth of one element per cycle.

One may observe that in the integer programs approximately 25% of all instructions are memory reference instructions and in the floating point programs, approximately 50% of all instructions are memory references. That means, if the memory system is capable of supplying only one data element in each cycle, a superscalar processor will not be able to issue more than two instructions every cycle for the SPEC floating point programs. The floating point benchmarks, typically thought of as number crunching programs are seen to place more demands on the memory system than on the floating point computation units. We believe that memory bandwidth is the fundamental bottleneck in scalar processors performing number crunching and that techniques to increase memory bandwidth are very important in modern multiple-issue processors.

Sohi and Franklin [18] suggested that caches for future superscalar processors would need to be interleaved. In this paper, we propose two schemes to improve the bandwidth provided by interleaved memories. The schemes can be applied for any architecture that uses a memory system with interleaved memory modules in any layer of the hierarchy.

Interleaved memories often do not provide their peak bandwidth due to inter-vector and intra-vector conflicts. Memory bank conflicts occur when a new request is submitted to a memory bank that is servicing a prior request and such bank conflicts result in the actual bandwidth supplied being significantly lower than the peak bandwidth. Memory contention in parallel and interleaved memories has been extensively studied [2] [5] [13] [4] [15] [20] [14] [19]. Unfortunately, most of the aforementioned research concentrated on memory conflicts for a single vector access stream, for various access patterns and strides.

Table 1: Percentage of loads and stores and instruction issue bound for SPEC programs assuming peak memory bandwidth of one element per cycle

Benchmark	% load	% store	Issue bound
doduc	26.6	9.3	2.7
nasa7	37.5	14.3	1.9
matrix300	37.0	18.5	1.8
fpppp	43.8	16.7	1.6
tomcatv	43.6	12.3	1.8
Mean(floating point)	37.7	14.2	1.9
eqntott	20.1	0.8	4.8
li	23.5	13.3	2.7
espresso	18.7	2.2	4.7
Mean(integer)	20.8	5.4	4.1

In most practical applications, several vectors are accessed concurrently, but there have been only a few studies dealing with multiple vector access streams [2] [13] [14]. This paper deals with high-bandwidth interleaved memories in the practical and realistic scenario of multiple access streams.

1.2 Objectives

Oed and Lange [13] and Raghavan and Hayes [14] presented vector data placement schemes to eliminate memory contention due to intervector interferences in vector processors. But in their scheme, the probability of finding a placement for conflict-free access is very low, as the number of vectors and the number of different strides increase. Our objective in this paper is to present some new schemes for eliminating/reducing memory contention during multiple vector accesses. We present a module partitioning technique in which we create disjoint access sets for each vector by partitioning the modules into sections for each vector. We also present an interlaced data placement scheme which is suitable for multiple vectors with unit stride.

The rest of this paper is organized as follows. Section 2 describes a module partitioning approach designed to eliminate inter-vector interferences. In section 3, an interlaced data placement scheme suitable for multiple concurrent vectors with unit stride, is presented. In section 4, we present a performance evaluation of the schemes. Section 5 offers concluding remarks.

2 A Module Partitioning Approach

Most applications involve simultaneous access of several vectors. If several vectors are to be accessed simultaneously, and if the different arrays are placed aligned to the same start bank (as most compilers would ordinarily do), it would result in high memory contention. Oed and Lange [13] and Raghavan and Hayes [14] proposed a data placement scheme to solve this problem. Let m represent the degree of memory interleaving and n_c be the memory latency in terms of processor cycles. Lemma 2 in [14] states the following. Let d_i and d_j be the strides of two streams with non-disjoint access sets and let $f = gcd(m, d_i, d_j)$ (gcd stands for greatest common divisor.) Then the start bank assignment, $b_i = 0$, $b_j = (n_c + l)d_i$ where $l \geq 0$ will lead to conflict free assignment, if and only if

$$\gcd(\frac{m}{f}, \frac{d_i - d_j}{f}) \geq 2n_c + l. \qquad (1)$$

When there are more than two vectors, this condition has to be satisfied for all pairs of vectors, and it should be possible to find distinct start banks for all the vectors. Thus the basic idea is to shift the start banks of the different vectors, and in many cases, such data placement can eliminate bank conflicts.

Such a data placement approach works very well if all the strides being used are the same. If all the streams have the same stride, even if there are conflicts, there is only one conflict and after the first conflict, one stream will lag behind the other stream and will not conflict again [2]. But if there are several vectors and they have different strides, as Raghavan and Hayes [14] point out, the number of conditions to meet increases and it becomes increasingly hard to meet all of them. We propose a *module partitioning* approach to solve this problem.

Theorem 1: In an m-way interleaved memory system with access time n_c, k vectors with strides $d_1, d_2, d_3, ...d_k$ may be accessed conflict-free if m can be partitioned into k disjoint sets with the cardinality of each set being greater than or equal to n_c, and the cardinality of set i being prime with respect to d_i.

Proof: The proof is very straightforward. Within each partition, the vector is interleaved in low order fashion. Since the cardinality of set i is coprime to d_i, there will be no inter-vector conflicts within vector i. The k sets are disjoint; therefore inter-vector interference cannot arise between disjoint vectors. □

The procedure for distributing the vector may be described as follows. For each vector j, find a set S_j as the set of all numbers s such that $n_c \leq s \leq m$ and such that s is prime with respect to the stride of vector j. For vectors $1, 2, \ldots, k$, we now have the sets S_1, S_2, S_3,. . . ,S_k. Then, find p_1, p_2, p_3,. . . ,p_k from sets S_1, S_2, S_3,. . . ,S_k respectively such that $\sum p_i = m$. If all arrays are of same size, make p_1, p_2, p_3,. . . ,p_k as close to each other as possible to avoid wasting space. If any vector j is larger than the others, allot larger p_j to that array. Now distribute vector 1 in the first p_1 modules, vector 2 in the next p_2 modules and so on. Basically the memory is being partitioned so that all vectors have disjoint access sets.

Consider the example used by Raghavan and Hayes [14]. Three vector streams A, B, and C with strides $d_A = 1$, $d_B = 5$, and $d_C = 9$ are being accessed simultaneously in an interleaved memory system with $m = 12$ and $n_c = 2$. They find that no data placement can avoid memory conflicts for such a condition. Now let us try our approach. First let us find S_A, S_B and S_C, the set of numbers lying between n_c and m and prime with respect to the stride. $S_A = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$, $S_B = \{2, 3, 4, 6, 7, 8, 9, 11, 12\}$, and $S_C = \{2, 4, 5, 7, 8, 10, 11\}$. Now we have to find p_A, p_B and p_C from sets S_A, S_B and S_C such that they add to exactly 12. A few choices are $\{2, 2, 8\}, \{2, 3, 7\}, \{3, 4, 5\}, \{4, 4, 4\}$. Let us choose (4,4,4) based on the assumption that the vectors are of the same dimension. The resulting placement is shown in Figure 1.

Raghavan and Hayes illustrate the conflicts in the memory for three vectors A, B and C for $m = 12$ and $n_c = 2$. (Figure 2 is reproduced from their illustration.) Figure 3 illustrates the same problem being handled in our approach without any conflicts. The module partitioning approach would work satisfactorily up to a latency of 4. Thus our approach leads to a solution for latency 4 where as the Raghavan and Hayes approach could not handle even a latency of 2.

Now we examine a few properties of the address mapping in the module partitioning scheme.

M0	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11
A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3	C4
A5	A6	A7	A8	B5	B6	B7	B8	C5	C6	C7	C8
A9	A10	A11	A12	B9	B10	B11	B12	C9	C10	C11	C12
A13	A14	A15	A16	B13	B14	B15	B16	C13	C14	C15	C16
A17	A18	A19	A20	B17	B18	B19	B20	C17	C18	C19	C20
A21	A22	A23	A24	B21	B22	B23	B24	C21	C22	C23	C24
A25	A26	A27	A28	B25	B26	B27	B28	C25	C26	C27	C28

Figure 1: Module Partitioning for three vectors with strides 1,5,9. ($n_c = 2$, $m = 12$)

Property 1.1: If α is the size of a data element, the ith element of the jth vector would be located at the address

$$A_j(i) = A_j(0) + \alpha * \{(i-1) + \lfloor \frac{(i-1)}{p_j} \rfloor * (m - p_j)\} \quad (2)$$

where $A_j(0)$ is the start address of the jth vector, p_j is the size of the partition for vector j and m is the degree of interleaving.

Proof: If no module partitioning is done

$$A_j(i) \quad = \quad A_j(0) + \alpha * (i-1). \quad (3)$$

When vector partitioning is done, the row number of the ith element will be $\lfloor \frac{(i-1)}{p_j} \rfloor$.

And from the perspective of vector j, each row has $m - p_j$ hole modules. Hence the ith element would be located at the address

$$A_j(i) = A_j(0) + \alpha * \{(i-1) + \lfloor \frac{(i-1)}{p_j} \rfloor * (m - p_j)\}.$$

□

Illustration: The address of C(19) in the placement in Figure 1 may be found out as follows. Assume the first location in $M0$ is $x0000$.

$$\begin{aligned} \text{address } C(19) &= x0032 + 4 * \{18 + \lfloor \frac{18}{4} \rfloor * 8 \\ &= x0232 \text{ (would fall in M10)} \end{aligned}$$

Property 1.2: Module partitioning transforms a vector of stride d_j to a vector with a stride pattern that repeats in a cycle of p_j elements. The p_j elements of the stride pattern would be

$$\begin{aligned} s_k \quad &= \quad d_j + \{\lfloor (k+1) * d_j p_j \rfloor - \lfloor \frac{k * d_j}{p_j} \rfloor\} * (m - p_j) \\ &\mid \quad k = 0, 1, 2, \ldots, p_j - 1 \end{aligned} \quad (4)$$

Proof: From Property 1.1, the ith element of the jth vector would be located at the address

$$A_j(i) = A_j(0) + \alpha * \{(i-1) + \lfloor \frac{(i-1)}{p_j} \rfloor * (m - p_j)\}.$$

Since the stride is d_j, the next element would be located at

$$\begin{aligned} A_j(i + d_j) \quad &= \quad A_j(0) + \alpha * \{(i - 1 + d_j) + \\ &\lfloor \frac{(i - 1 + d_j)}{p_j} \rfloor * (m - p_j)\} \end{aligned}$$

Hence the stride between elements i and $i + d_j$ is

$$s = d_j + \{\lfloor \frac{(d_j + i - 1)}{p_j} \rfloor - \lfloor \frac{i-1}{p_j} \rfloor\} * (m - p_j).$$

Since the vector is stored in p_j modules and the stride is d_j, if the element $A(i)$ is accessed, the element $A(i + d_j * p_j)$ will be accessed, and hence the stride pattern will repeat after $\frac{d_j * p_j}{d_j}$ or p_j elements. The new stride pattern will hence consist of the p_j elements

$$\begin{aligned} s_k \quad &= \quad d_j + \{\lfloor \frac{(k+1) * d_j}{p_j} \rfloor - \lfloor \frac{k * d_j}{p_j} \rfloor\} * (m - p_j) \\ &\mid k = 0, 1, 2, \ldots, p_j - 1 \end{aligned}$$

□

Time	M0	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11
0	A	—	B	—	C	—	—	—	—	—	—	—
1	A	x	B	—	C	—	—	B	—	—	—	—
2	B	x	A	—	—	—	—	B	—	—	—	—
3	B	C	A	A	—	B	—	—	—	—	—	—
4	—	C	—	A	A	B	—	—	—	—	x	—
5	—	—	—	B	A	A	—	—	—	—	x	—
6	—	—	—	B	—	A	A	—	B	—	C	—
7	—	B	—	—	—	—	A	x	B	—	C	—
8	—	B	—	—	—	—	B	x	A	—	—	—
9	—	—	—	—	—	—	B	C	A	A	—	B

Figure 2: The state-time diagram of a modular memory with $m = 12$ and $n_c = 2$, for three vector streams ($d_A = 1, d_B = 5, d_C = 9$) for which no data placement can prevent conflicts. x denotes a conflict. (Reproduced from [14])

Illustration: For the example in Figure 1, for vector B, the first stride is

$$s_0 = 5 + \{\lfloor \frac{(1*5)}{4} \rfloor - \lfloor \frac{0*5}{4} \rfloor\} * (8) = 13$$

The entire stride pattern for vector B could be similarly computed, and seen to be 13,13,13,21. For A and C, the patterns are 1,1,1,9 and 25,25,25,33 respectively.

Property 1.3: In a single instruction stream processor, the number of iterations required to be present in an unrolled loop, to eliminate detailed address computation within every iteration, is equal to the least common multiple (LCM) of the cardinality of the partitions i.e., $LCM(p_1, p_2, \ldots, p_k)$

Proof: The proof follows Property 2. For each vector j, the modified stride pattern consists of p_j elements. In order to eliminate detailed address computations, an integral multiple of each pattern should be contained in the unrolled loop. If there are n vectors or partitions, the minimum number of iterations in the unrolled loop will be the LCM of the p_js, or the minimum degree of unrolling required will be one less than the LCM of the cardinalities. □

Illustration: For the example in Figure 1, $p_A = p_B = p_C = 4$. Hence the LCM of the cardinality of the partitions is 4 and property 1.3 says that detailed address computation within every loop can be avoided if the loop is unrolled thrice. This could be verified in the following manner also. The stride patterns for A, B, and C contain 4 elements each, as evident from the illustration following Property 1.2. Hence if the unrolled loop contains 4 iterations, a complete pattern for each vector can be contained in the unrolled loop and detailed address computation need be performed only in the prologue to the loop.

Time	M0	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11
0	A1	—	—	—	B1	—	—	—	C1	—	—	—
1	A1	A2	—	—	B1	B6	—	—	C1	C10	—	—
2	—	A2	A3	—	—	B6	B11	—	—	C10	C19	—
3	—	—	A3	A4	—	—	B11	B16	—	—	C19	C28
4	A5	—	—	A4	B21	—	—	B16	C37	—	—	C28
5	A5	A6	—	—	B21	B26	—	—	C37	C46	—	—
6	—	A6	A7	—	—	B26	B31	—	—	C46	C55	—
7	—	—	A7	A8	—	—	B31	B36	—	—	C55	C64
8	A9	—	—	A8	B41	—	—	B36	C73	—	—	C64
9	A9	A10	—	—	B41	B46	—	—	C73	C82	—	—

Figure 3: The state-time diagram of a modular memory with $m = 12$ and $n_c = 2$, for three vector streams ($d_A = 1, d_B = 5, d_C = 9$) when module partitioning is applied. The access is conflict-free.

Property 1.4: If m is a power of two, say 2^l, the number of vectors k is a power of two, say 2^v, the strides of the vectors are prime with respect to $\frac{m}{k}$, and p is the smallest integer such that $2^p \geq$ dimension of the vectors, then module partitioning is equivalent to the address transformation

$$\begin{aligned} &< A_t \ldots A_{p+v} >< A_{p+v-1} \ldots A_p >< A_p \ldots A_0 > \\ &\Longleftrightarrow < A_t \ldots A_{p+v} >< A_{p-1} \ldots A_{l-v} > \\ &< A_{p+v-1} \ldots A_p >< A_{l-v-1} \ldots A_0 > \end{aligned}$$

where t is the total number of bits in the word address. The bank number would be the concatenation of the v-bit vector number, $< A_{p+v-1} \ldots A_{p+1} A_p >$ and the lower order $l - v$ address bits, $< A_{l-v-1} \ldots A_1 A_0 >$.

Proof: The cardinality of the partitions would be $\frac{m}{k}$ which is equal to $\frac{2^l}{2^v} = 2^{l-v}$. Hence the memory consists of 2^v partitions of 2^{l-v} modules each, interleaved in a low-order fashion within the partition. The lower p bits of the address would denote the offset of the vector element from the start address. Align the first element of the first vector in such a way that bits $p, p+1, \ldots, p+v-1$ from the least significant edge are zeroes. Then the other vectors can be placed in such a way that bits $p, p+1, \ldots, p+v-1$ represent the vector number in the conventional (unpartitioned) address. Hence the address after partitioning would be the concatenation of the higher order $t - v - p$ bits, the $(p - l + v)$ bits of the vector element offset, the vector number and the lower order $l - v$ bits of the address. The bank number would then be the concatenation of the v-bit vector number and the lower $l - v$ bits of the address.

Actually the bits for the vector number part of the original address need not be contiguous to the lower p bits. If they are contiguous, the vectors would be in continuous locations after module partitioning. Even otherwise, they can be accessed in a conflict-free fashion; the only problem would be that memory would be fragmented. □

Illustration: Consider four vectors with strides 3,5,7,9 placed in an interleaved memory with 16 modules. All the four partitions could be of cardinality 4. Let us assume that the dimension of the vectors are 1000 each. Hence $p = 10$. As evident, $l = 4, k = 2, v = 2$. In a conventional memory assignment, it is very likely that the vectors are located at addresses x, x+1000, x+2000, x+3000 (considering word addressing). The address bit pattern would consist of higher order t bits, 2 v bits and 10 p bits. Module partitioned addresses can be obtained by concatenating the higher order $t-v-p$ bits, highest 8 of the 10 p bits, the 2 v bits and the lowest 2 p bits. The bank number would be the concatenation of the 2 v bits and the lowest 2 p bits.

Property 1.5: Given n_c, and the strides of the concurrent vectors d_i, the minimum degree of interleaving required to achieve conflict free access is

$$M_{\min} = \sum pm_i \quad (5)$$

where pm_i is the smallest integer greater than or equal to n_c, and prime with respect to d_i, for every stride.

Proof: The proof directly follows from the fact that in sequential interleaving, bandwidth is $\frac{m}{gcd(m,d_i)}$. Hence pm_i has to be prime with respect to d_i. If n_c is the latency, the memory has to be at least n_c-way interleaved, if one access has to be over before the next one arrives. Hence

$$M_{\min} = \sum pm_i$$

□

Illustration: To obtain the minimum degree of interleaving required for the 3 vectors in our example, for the case with latency 2, we obtain $pm_A = 2$, $pm_B = 2$, and $pm_C = 2$, and hence $M_{\min} = 6$. In effect, our approach results in a solution with 6-way interleaving where Raghavan and Hayes' data placement could not avoid conflicts even with a 12-way interleaving.

Property 1.6 If all p_js are equal, all memory modules will be accessed with the same frequency, irrespective of the stride.

Proof: Since p_j is prime with respect to d_j, the requests from vector j are uniformly distributed within all the p_j modules. Therefore, the frequency of accessing a particular module is $\frac{1}{p_j}$. Hence if all p_js are equal, all the modules will be accessed at the same frequency. □

Illustration: In the module partitioning in Figure 3, $p_A = p_B = p_C$, and there is uniform distribution of requests. One may observe that in Figure 2, several modules are not being used when conflicts are occurring in some of the modules, or in other words, even when the system is underutilized, there are conflicts.

The module partitioning technique suffers from the following drawbacks. One disadvantage is that this mapping scheme might result in inefficient memory utilization or memory fragmentation. Another disadvantage is that when a single vector is being accessed, the memory system could support a higher n_c, if module partitioning was not done. Yet another problem is that the access patterns have to be known *a-priori*. Above all, there is increased address computation. But the scheme can result in conflict-free accesses in several cases, where existing techniques cannot guarantee peak-bandwidth.

Property 1.7: Let $p_i \mid_{i=1,\dots,n}$ be the cardinality of each set in which the n vectors are placed, and $DIM_i \mid_{i=1,2,\dots,n}$ be the dimensions of the n vectors. There will be no memory fragmentation if $p_1 : p_2 : p_3 : \dots p_n = DIM_1 : DIM_2 : \dots : DIM_n$

Proof: The number of rows consumed by vector i would be $\frac{DIM_i}{p_i}$. There will be no memory fragmentation if all the vectors consume identical number of rows, or if,

$$\frac{DIM_1}{p_1} = \frac{DIM_2}{p_2} = \dots = \frac{DIM_n}{p_n}$$

(It is assumed that the vectors are allocated in contiguous locations in the memory. If non-contiguous allocation is done, there will be fragmentation anyway.) □

Illustration: In the example in Figure 1, we chose (4,4,4) assuming that the vectors are of equal dimensions. If they have different dimensions, it is advantageous from a memory utilization standpoint to have p_A, p_B and p_C proportional to the array sizes. If the array dimensions have an increasing ratio, we could adopt the (2,3,7) combination.

2.1 Comments

In module partitioning, conflicts can be avoided only if $M_{\min} \leq M$. If conflicts cannot be eliminated, the best alternative would be to provide maximum possible bandwidth. For this purpose, form sets $S_A{}^*$, $S_B{}^*$ and $S_C{}^*$ along with sets S_A, S_B and S_C, which include numbers smaller than n_c and prime with respect to the stride of the respective vector if possible. If numbers that are prime with respect to the stride cannot be found, try to maximize the greatest common divisor with the stride. Now find p_A, p_B and p_C from sets S_j and $S_j{}^*$, attempting to maximise the number of p_js chosen from sets S_j, and minimising the number of p_js chosen from sets $S_j{}^*$.

Consider application of module partitioning for a matrix multiplication problem. Generally an $(n \times n)$ matrix needs to be accessed with the set of strides $\{1, n, n+1\}$. Form a set S_{matrix} with numbers greater than n_c and smaller than M and prime with respect to the strides in the set of strides. Choose p_js for the multiplicand, multiplier and product matrices from these sets. For example, if we have two 64×64 matrices to multiply, and we have an 8-way interleaved memory with latency $n_c = 2$,

$$Set\ of\ strides \quad = \quad \{1, 64, 65\}$$

and

$$S_{matrix} \quad = \quad \{3, 7\}$$

In this problem, $M_{\min}$ is 9, and it is not possible to entirely avoid conflicts with 8-way interleaving. If we had 9 memory modules, the multiplicand, multiplier and product matrices could be accessed conflict-free at the same time. Since only 8 memory modules are available, a feasible solution would be to have 3 modules each for the multiplicand and multiplier, but just 2 for the product, based on the fact that elements of the product matrix are accessed with a lower frequency than the elements of the other two matrices.

Past research [1][9] has shown that magic square or perfect latin square mapping would allow to place matrices such that accessing of rows, columns, diagonals and sub-matrices can be performed without conflicts. These schemes are designed to provide concurrent access of all elements of one column or row or diagonal at the same time, and needs N-way interleaving if the matrix is $N \times N$. Our objective in module partitioning is to provide peak bandwidth for the existing configuration. For the case where as many modules as the dimensions of the matrix are available (as in large shared memory multiprocessor systems), the latin square approach may be desirable, whereas, if the degree of interleaving is very small compared to the dimensions of the matrix, module partitioning is a good approach yielding peak bandwidth. In the SPEC benchmark *tomcatv*, $N \times N$ matrices are used, with N=33, 65, 129 and 257. If the memory system is interleaved to a low degree such as 8 or 16, the latin square approach cannot be used, but module partitioning can be.

One major argument against module partitioning would be that module partitioning will work well only when the number of vectors is relatively smaller than the degree of interleaving. If the number of vectors is greater than the degree of interleaving, several clusters of vectors could be formed and each cluster partitioned into the m memory modules. If the degree of interleaving is m no one expects more than m accesses to be proceeding simultaneously, and it is not difficult to devise a module partitioning scheme such that m accesses proceed simultaneously.

3 An Interlaced Data Placement Approach

In this section, we introduce an interlaced data placement approach primarily suited for multiple vectors with unit stride, for cases where Raghavan and Hayes' data placement approach would not work, and the degree of interleaving is less than the minimum required for module partitioning. The module partitioning approach could be applied here, with the objective of minimizing conflicts (as described in section 2.1), even if elimination of conflicts is not possible. But for the special case of unit stride, there is a simpler solution involving simpler address arithmetic than module partitioning. The idea in our interleaved data placement scheme is to interleave the vectors exactly as they would be interlaced during access i.e. to approximate an optimal memory system.

M0	M1	M2	M3
a1	b1	c1	a2
b2	c2	a3	b3
c3	a4	b4	c4
a5	b5	c5	a6
b6	c6	a7	b7
c7	a8	b8	c8

Figure 4: Interlaced data placement for two vectors performing the computation $c(i) = a(i) * b(i)$ ($n_c = 2$, $m = 4$)

An optimal memory organization is one in which the memory elements are distributed in just the sequence as they are accessed. For instance, if the computation is $c(i) = a(i) * b(i)$ during access, the data elements of a, b and c vectors are interlaced during access. If they could be stored in the same pattern, a conventional low-order sequential interleaved memory guarantees peak bandwidth. In other words, it acts as an optimal memory in a perfect pipelined fashion. The interlaced data placement for a sample computation is illustrated in Figure 4. Now we illustrate various properties of the Interlaced Placement approach.

Property 2.1: In interlaced data placement scheme, with n concurrent vectors, the ith element of the jth vector would be located at

$$A_j(i) \quad = \quad A_j(0) + (i-1) * n * \alpha. \tag{6}$$

where $A_j(0)$ is the start address of the jth vector, and α is the size of an element. Once the address transformation is performed, the new address is distributed as low order sequential interleaving. Hence the bank number would be $(A_j(i) \bmod \mathrm{m})$, as in conventional sequential interleaving.

Property 2.2: For single vector access, the bandwidth available from an interlaced data placement scheme with n vectors is

$$BW \quad = \quad \frac{M}{\gcd(m,n)} \tag{7}$$

Accessing of individual vectors is important because, during initializing and printing, each vector is accessed in a sequential fashion. The number of vectors and degree of interleaving should be prime with respect to each other, for efficient accessing of each individual vector. If the number of vectors and degree of interleaving of modules are coprime, each individual vector would be accessible at peak bandwidth. If the number of interlaced vectors n and the degree of interleaving m are the same, all elements of each vector would lie in the same module. If absolute prevention of conflicts is not possible, one is generally interested in minimization of conflicts and the conflicts are minimum when the greatest common divisor of the number of vectors and the degree of interleaving is the smallest possible.

Property 2.3 If the vectors are stored in exactly the same sequence as they are accessed, the interlaced sequence of the n vectors is available at peak bandwidth even if n is not prime with respect to m.

Property 2.4 If an interlaced placement avoids conflicts for stride i, it also avoids conflicts for stride $(k * m) + i$, where k is any integer and m is the degree of interleaving.

Property 2.5 When the number of vectors being accessed is equal to m, interlaced placement results in each vector being contained entirely in one module. i.e. the same storage pattern that module partitioning with cardinality of the partitions being unity would result.

The main advantage of interleaved placement is that it results in minimal increased complexity in address generation. The main disadvantage is that if vectors are being reused in different routines or loops in different fashions, then the optimal pattern for each routine would be different, and it may be necessary to reallocate data with each loop. Also, often data structures are defined in some routine, and they may be used in different routines, hence it may be difficult to arrive at a suitable placement at compile time. Nevertheless, interlacing results in improved performance in several cases, as illustrated in section 4.

4 Performance Evaluation

Preliminary evaluation of the benefits of the module partitioning and interlaced data placement schemes are done by evaluating the techniques in a pipelined memory system for popular kernels. The experiments are similar to experiments performed by Oed and Lange [13]. We use one particular storage pattern and test it for a variety of strides and stride patterns, to evaluate the robustness of the technique. A simulator for a processor with the MIPS instruction set [8] is used.

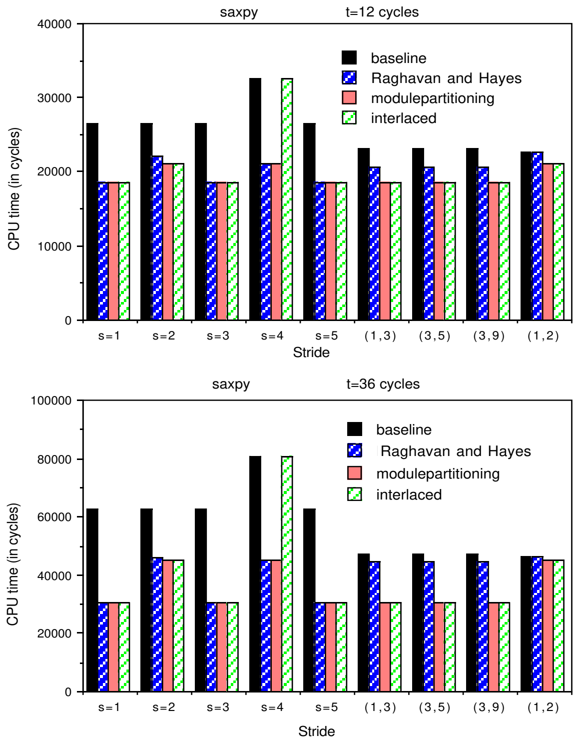

Figure 5: Execution time for saxpy for different strides

Experiments are performed for the triad used by Oed and Lange [13], and for saxpy.The triad used by Oed and Lange is

```
      DO 1 I=1,n*INC, INC
1     A(I) = B(I) + C(I) * D(I)
```

By INC, the increment or the stride of a Fortran loop is denoted. The triad is executed for several strides. Then the loop is modified to have unequal strides for the four vectors and the experiment is repeated for several combinations of strides. The other loop used for experimentation is saxpy. Experiments are performed for several equal and unequal strides for saxpy also.

The source programs are first compiled with the optimizers on and then the assembly output is manipulated to incorporate the particular data placement scheme. The program is then recompiled with the MIPS reorganizing compiler and the trace output is directly fed to the simulator. Thus manual intervention has been kept to a minimum.

The memory system is 4-way interleaved. The processor contains a 1 kbyte lock-up free cache with four Miss Status Holding Registers. The cache is set associative with 8 sets. The cache may seem to be unrealistically small, but it has been intentionally kept small so that the entire data set would not fit in the cache.

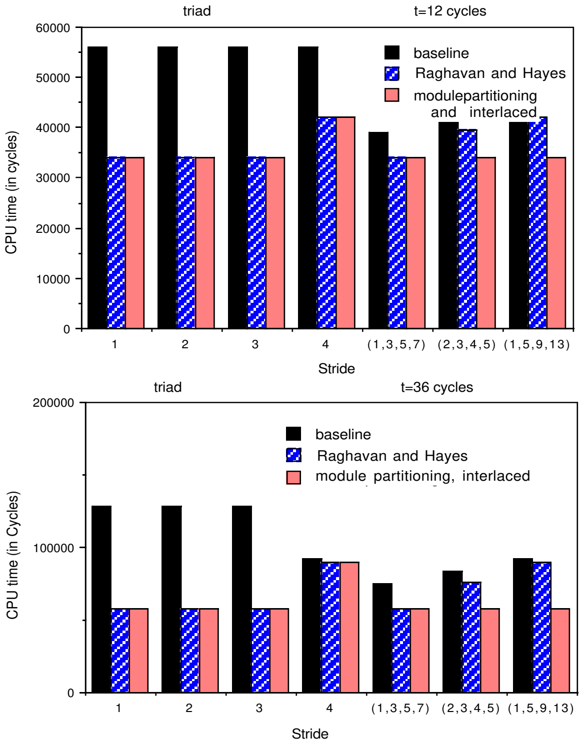

Figure 6: Execution time for the triad for different strides

The CPU execution times in processor cycles are illustrated in Figure 5 and Figure 6 for experiments with saxpy and the triad. The experiments were performed for latencies 12, 24 and 36, but only results for 12 and 36 are presented. The main impact of latency is that at lower memory latencies, factors such as instruction counts affect the CPU time, whereas at higher latencies, memory is more of a bottleneck. For module partitioning, the same partition is tested for various stride combinations. For equal strides, placement based on [14] and module partitioning yield the same performance. For unequal stride combinations, module partitioning is seen to be better than placement based on [14]. One may observe this effect in the stride combinations (1,3), (3,5), (3,9) and (1,2) for saxpy and for ((2,3,4,5) and (1,5,9,13) for the triad. The interlaced placement primarily devised for unit strides is seen to be quite robust to various strides. For saxpy, interlaced placement yields same performance as module partitioning except for stride 4. For triad, for degree of interleaving 4, module partitioning and interlaced placement resulted in the same storage pattern and so only one is illustrated in Figure 6. A notable observation is that the baseline machine configuration performance is better for a combination of strides rather than a single constant stride.

We also compute the normalized throughput of the triad for module partitioning, on a 16-way interleaved memory system, for all strides between 1 and 16. The summary of the observations are presented in Table 2 and Table 3. The results presented in these Tables can be obtained intuitively also (no simulation is required).

If the strides of all the four vectors are the same as considered in Table 2, data placement as in [13] are also equally effective. The power of module partitioning lies

Table 2: Stride versus throughput for a few partitions for the triad program

Stride	4-4-4-4	3-3-3-7	3-3-5-5
1	1	1	1
2	0.5	1	1.0
3	1	0.5	0.67
4	0.25	1	1.0
5	1	1.0	0.6
6	0.5	0.5	0.67
7	1	0.79	1.0
8	0.25	1.0	1.0
9	1	0.5	0.67
10	1	0.5	0.6
11	0.5	1.0	1.0
12	0.25	0.5	0.67
13	1	1.0	1.0
14	0.5	0.79	1.0
15	1	0.5	0.67
16	0.25	1.0	1.0

in cases with a combination of the different strides. Any combinations of the strides indicated in Table 3 would give peak throughput for the corresponding configuration. In the case of the configuration 3-3-5-5, several stride combinations not indicated in Table 3 also yield peak throughput; in the illustration, all strides with factors 3 or 5 are eliminated, whereas, actually you have to eliminate those with factors of 3 for the first two vectors and strides with factor 5, only for the last two vectors. An example is the stride combination 5,10,3,6 for vectors A,B,C, and D respectively.

5 Conclusion

Vector interference is a problem that has plagued designers of high performance computers for years. This paper presents two schemes to alleviate inter-vector interferences. First, we introduced a module partitioning technique, where the memory banks are partitioned to groups or sections for each vector in a concurrent vector access environment. The scheme results in disjoint access sets for each vector and eliminates inter-vector interferences. The number of modules in each group is prime with respect to the stride of the vector it holds and that eliminates intra-vector interferences. The scheme is successful in several cases where existing data placement techniques would not work. The paper states, proves and illustrates various properties of the approach. These properties can be used by the compiler to generate code with optimized address computations. We also presented an interlaced data placement scheme particularly suited for multiple vector accesses with unit stride. The mapping in interlaced placement involves a simpler address calculation than that in the module partitioning technique.

Performance of the techniques are evaluated for a wide variety of strides and stride patterns. The performance is also compared with data placement schemes reported in literature. The techniques can be applied in vector supercomputers, superscalar processors, or any computing paradigm, if any layer in the memory hierarchy has interleaved or parallel memory modules.

Prefetching techniques have been used in superscalar processors to reduce latency. A limitation with prefetching techniques is that they are latency hiding techniques,

Table 3: Conflict-free strides and maximum n_c for a few partitions, for the triad program

Partition	Conflict-free strides	max nc
4,4,4,4	1,3,5,7,9,11,13,15, and any combinations thereof	4
3,3,5,5	1,2,4,8,14,16, and any combinations thereof	3
3,3,3,3,	1,2,4,5,7,8,10,11,13,14,16, and any combinations thereof	3

and latency can be hidden only if sufficient bandwidth is available. Unless accompanied by matching memory bandwidth, aggressive instruction issue in processors or aggressive data prefetching will not bring any net returns. Hence research on improving the bandwidth of memory systems is critical to achieving high performance from modern and future superscalar processors and that also increases the significance of this work. With the popularity of multiple issue processors, a lot of 'vector supercomputing' has shifted to workstations based on superscalar and/or superpipelined instruction issue and hence the importance given to vectors in this paper is very well justified.

Acknowledgment

This work was supported in part by the University of South Florida Research and Creative Scholarship Grant Program under Grant No. 21-08-935-RO. The authors also wish to acknowledge the discussions with Ed Wilkens. The help of Vinod Reddy and Vijay Kammila in preparing the manuscript is also acknowledged.

References

[1] M. Balakrishnan, R. Jain and C. S. Raghavendra, *On array storage for conflict-free access for parallel processors,* Proceedings 17th International Conference on Parallel Processing, 1988, pp. 103-107.

[2] T. Cheung and J. E. Smith, *A Simulation Study of the Cray X-MP memory system,* IEEE Transactions on Computers, vol. C-35, July 1986, pp. 613-622.

[3] R. F. Cmelik, S. I. Kong, D. R. Ditzel, and E. J. Kelly, *An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks,* Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 290-302.

[4] D. T. Harper III, *Address transformations to increase memory performance,* Proceedings of the 1989 International Conference on Parallel Processing, August 1989, pp. I-237-I-241.

[5] D. T. Harper III and J. R. Jump, : *Vector access performance in parallel memories using a skewed storage scheme,* IEEE Transactions on Computers, vol. C-36, December 1987, pp. 1440-1449.

[6] D. T. Harper III, *Block, multi-stride vector, and FFT accesses in parallel memory systems,* IEEE Transactions on Parallel and Distributed Systems, Vol.2. No.1, pp. 43 - 51, January 1991.

[7] D. T. Harper III and Y. Costa, *Analytical Estimation of Vector Access Performance in Parallel Memory Architectures,* IEEE Transactions on Computers, May 1993.

[8] G. Kane, *mips RISC ARCHITECTURE,* Prentice Hall, NJ, 1989.

[9] K. Kim and V. K. Prasanna, *Latin Squares for Parallel Array Access,* IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 4, April 1993, pp. 361-370.

[10] L. Kurian, *Issues in the Design of a Decoupled Architecture for a RISC environment,* Ph.D. thesis, Computer Engineering, The Pennsylvania State University, August 1993.

[11] S-M. Moon and K. Ebcioglu, *A Study on the Number of Memory Ports in Multiple Instruction Issue Machines,* Proceedings of the 26th Annual International Symposium on Microarchitecture, December, 1993, pp. 49-58.

[12] *NSF Memory System Workshop Report,* April 1993.

[13] W. Oed and O. Lange, *On the Effective Bandwidth of interleaved memories in vector processor systems,* IEEE Transactions on Computers, vol. C-34, October 1985, pp. 949-957.

[14] R. Raghavan and J. P. Hayes, *Reducing Interference Among Vector Accesses in Interleaved Memories,* IEEE Transactions on Computers, April 1993, pp. 471-483.

[15] B. R. Rau, M. S. Schlansker and D. W. L. Yen, *The Cydra 5 stride-insensitive memory system,* Proceedings of the 1989 International Conference on Parallel Processing, August 1989, pp. 242-246.

[16] A. Rogers and K. Li, *Software Support for Speculative Loads,* Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, pp. 38 - 50. 1989.

[17] A. Seznec and J. Lenfant, *Interleaved Parallel Schemes: improving memory throughput on supercomputers,* Proceedings of the 19th International Symposium on Computer Architecture, May 1992, pp. 246-255.

[18] G. S. Sohi and M. Franklin, *High Bandwidth Data Memory Systems for Superscalar Processors,* Proceedings of ASPLOS-IV, 1991, pp. 53-62.

[19] G. S. Sohi, *High Bandwidth Interleaved memories for Vector Processors,* IEEE Transactions on Computers, vol. 42, No. 1, January 1993, pp. 34-44.

[20] M. Valero, T. Lang, J. M. Llaberia, M. Peiron, E. Ayguade and J.J. Navarro, *Increasing the Number of Strides for Conflict Free Vectors,* Proceedings of the International Symposium on Computer Architecture, May 1992, pp.372-381

[21] S. Weiss, *Multiport Memory Access in Decoupled Architecture Processors,* Proceedings of the International Conference on Parallel Processing, 1991, I-373-I-376.

[22] A. Wolfe and R. Boleyn, *Two-ported Cache Alternatives for Superscalar Processors,* Proceedings of the 26th Annual International Symposium on Microarchitecture, December, 1993, pp. 41-48.