

# Algorithmic Foundations for a Parallel Vector Access Memory System

Binu K. Mathew, Sally A. McKee, John B. Carter, Al Davis  
Department of Computer Science  
University of Utah  
Salt Lake City, UT 84112

{mbinu | sam | retrac | ald}@cs.utah.edu

## Abstract

*This paper presents mathematical foundations for the design of a memory controller subcomponent that helps to bridge the processor/memory performance gap for applications with strided access patterns. The Parallel Vector Access (PVA) unit exploits the regularity of vectors or streams to access them efficiently in parallel on a multi-bank SDRAM memory system. The PVA unit performs scatter/gather operations so that only the elements accessed by the application are transmitted across the system bus. Vector operations are broadcast in parallel to all memory banks, each of which implements an efficient algorithm to determine which vector elements it holds. Earlier performance evaluations have demonstrated that our PVA implementation loads elements up to 32.8 times faster than a conventional memory system and 3.3 times faster than a pipelined vector unit, without hurting the performance of normal cache-line fills. Here we present the underlying PVA algorithms for both word interleaved and cache-line interleaved memory systems.*

## 1. Introduction

Processor speeds are increasing much faster than memory speeds, and thus memory latency and bandwidth limitations prevent many applications from making effective use of the tremendous computing power of modern microprocessors. The traditional approach to addressing this mismatch has been to structure memory hierarchically by adding several levels of cache between the processor and the DRAM store. Caches reduce average load/store latency by exploiting spatial and temporal locality, but they may not improve the performance of irregular applications that lack this locality. In fact, the cache hierarchy may exacerbate the problem by loading and storing entire data cache lines, of which the application uses

only a small portion. Moreover, caches do not solve the bandwidth mismatch on the cache-fill path. In cases where system-bus bandwidth is the limiting bottleneck, better memory system performance depends on utilizing this resource more efficiently.

Fortunately, several classes of applications that suffer from poor cache locality have predictable access patterns [1, 9, 2]. For instance, many data structures used in scientific and multimedia computations are accessed in a regular pattern that can be described by the base address, stride, and length. We refer to data accessed in this way as *vectors*. The work described here is predicated on our belief that memory system performance can be improved by explicitly informing the memory system about these vector accesses.

This paper presents algorithms to efficiently load and store base-stride vectors by operating multiple memory banks in parallel. We refer to the architectural mechanisms used for this purpose as *Parallel Vector Access*, or *PVA*, mechanisms. Our PVA algorithms have been incorporated into the design of a hardware PVA unit that functions as a subcomponent of an intelligent memory controller [3]. For unstructured accesses, this PVA mechanism performs normal cache-line fills efficiently, and for non-unit stride vectors, it performs parallel scatter/gather operations much like those of a vector supercomputer.

We have implemented a Verilog model of the unit, and have validated the PVA's design via gate-level synthesis and emulation on an IKOS FPGA emulation system. These low-level experiments provided timing and complexity estimates for an ASIC incorporating a PVA. We then used the Verilog functional model to analyze PVA performance for a suite of benchmark kernels for which we varied the number and type of streams, relative alignment, and stride, and we compared this performance to that of three other memory system models. For these kernels, the PVA-based SDRAM memory system fills normal (unit-stride) cache lines as fast as and up to 8% faster than a conventional, cache-line interleaved memory system optimized for line fills. For larger strides, our PVA loads elements up to 32.8 times faster than the conventional memory system and up to 3.3 times faster than a pipelined, centralized memory access unit that can gather sparse vector data by issuing (up to) one SDRAM access per cycle. At the cost of a modest increase in hardware complexity, our PVA calculates DRAM addresses for gather operations from two to five times faster than another recently published design with similar

goals [5]. Furthermore, exploiting this mechanism requires no modifications to the processor, system bus, or on-chip memory hierarchy. Complete architectural details and experimental results are reported elsewhere [12].

In the next section, we present background information about the organization and component properties of the high performance memory system targeted by our PVA mechanism. The following sections define our terminology and outline the operations performed by the PVA unit, then provide a detailed analysis of an initial implementation of that functionality. We then refine the operation of the PVA mechanisms, and describe efficient hardware designs for realizing them. We discuss issues of integrating the PVA into a high performance uniprocessor system and survey related work. Finally, we discuss ongoing work within this project.

## 2. Memory System Characteristics

This section describes the basic organization and component composition of the main memory system containing the PVA. Figure 1 illustrates the high-level organization. The processor issues memory requests via the system bus, where they are seen by the *Vector Command Unit* (VCU). The VCU is responsible for loading the data requested by the processor from the bank(s) where it is stored, and returning it to the processor. The memory system that our work targets returns requested base-stride data as cache lines that can be stored in the processor’s normal cache hierarchy [3]. However, our PVA unit is equally applicable to systems that store the requested base-stride data in vector registers [5, 7] or stream buffers [13]. For the sake of clarity, we refer to the chunks of base-stride data manipulated by the memory controller as *memory vectors*, while we refer to the base-stride data structures manipulated by the program as *application vectors*.

We first review the basic operation of dynamic memory devices, and then we then present the logical organization of the memory system and describe the vector operations that our memory controller must perform.

### 2.1 Backend Basics

Since DRAM storage cell arrays are typically rectangular, a data access sequence consists of a row access (indicated by a RAS, or *row address strobe*, signal) followed by one or more column accesses (indicated by a CAS, or *column address strobe*, signal). During the RAS, the row address is presented to the DRAM. Data in the storage cells of the decoded row are moved into a bank of *sense amplifiers* (also called a *page buffer*) that serves as a row cache. During the CAS, the column address is decoded and the selected data are read from the sense amps. Once the sense amps are *precharged* and the selected page (row) is loaded, the page remains charged long enough for many columns to be accessed. Before reading or writing data from a different row, the current row must be written back to memory (i.e., the row is *closed*), and the sense amps must again be precharged before the next row can be loaded. The time required to close a row accounts for the difference between advertised DRAM access and cycle times.

The order in which a DRAM handles requests affects memory performance at many levels. Within an SDRAM bank, consecutive

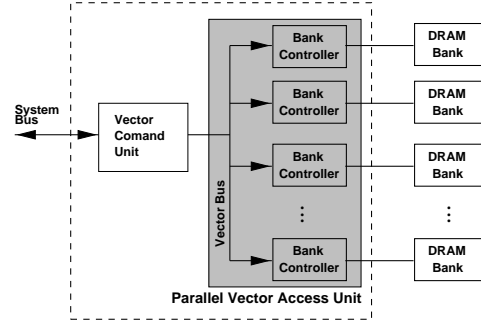


Figure 1. Organization of SDRAM Memory System

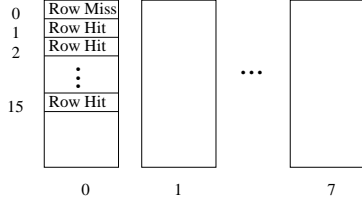
accesses to the current row—called *page hits* or *row hits*—require only a CAS, allowing data to be accessed at the maximum frequency. In systems with interleaved banks (either internal banks, or multiple devices), accesses to different banks can be performed in parallel, but successive accesses to the same bank must be serialized. In addition, the order of reads with respect to writes affects bus utilization: every time the memory controller switches between reading and writing, a *bus turnaround delay* must be introduced to allow data traveling in the opposite direction to clear.

Memory system designers typically try to optimize performance by choosing advantageous *interleaving* and *row management* policies for their intended workloads. Independent SDRAM banks can be interleaved at almost any granularity, from one-word to one-row (typically 512-2048 bytes) wide, and the non-uniform costs of DRAM accesses cause the choice of interleaving scheme to affect the memory performance of programs with certain access patterns. Similarly, consecutive accesses to the same row can be performed more quickly than accesses to different rows, since the latter require the first row to be written back (closed) and the new row to be read (opened). The choice of when to close a row (immediately after each access, only after a request to a new row arrives, or perhaps some combination of these *open* and *closed* page policies) therefore impacts performance. Our analysis of the PVA system [12] examined a variety of interleaving schemes and row management policies, in addition to various implementation choices.

### 2.2 Memory System Operation

For simplicity, data paths are omitted from the organization depicted in Figure 1. We assume that the processor has some means of communicating information about application vectors to the memory controller, and that the PVA unit merely receives memory-vector requests from the VCU and returns results to it. The details of the communication between the VCU and CPU over the system bus are not relevant to the ideas discussed here. This communication mechanism can take many forms, and some of the possible options are discussed in our earlier reports [12, 19].

Processing a base-stride application vector involves gathering strided words from memory for a read, and scattering the contents of a dense memory vector to strided words in memory for a write. For example, to fetch consecutive column elements from an array stored in row-major order, the PVA would need to fetch every  $N$  words from memory, where  $N$  is the width of the array.



**Figure 2.** Cache-line fill timing

SDRAM’s non-uniform delays make such scatter/gather operations significantly more expensive than they are for SRAM. Consider the eight-bank, cacheline-interleaved SDRAM memory system illustrated in Figures 2 and 3.

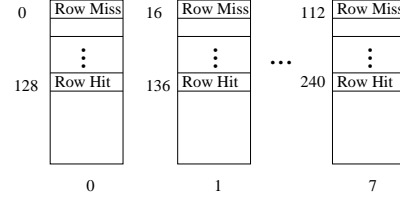
Figure 2 illustrates the operations required to read a 16-word cacheline when all of the internal banks are initially closed. First the row address is sent to the SDRAM, after which a delay equivalent to the RAS latency must be observed (two cycles). Then the column address is sent to the SDRAM, after which a delay equivalent to the CAS latency must be observed (two cycles). After the appropriate delay, the first word of data appears on the SDRAM data bus. Assuming that the SDRAM has a burst length of 16, it sends out one word of data on each of the next 15 cycles.

Figure 3 illustrates what happens when a simple vector unit uses the same memory system to gather a memory vector with a large stride. In this case, we assume that the desired vector elements are spread evenly across the eight banks. As will be shown in Section 4, the distribution across banks is dependent on the vector stride. To load the vector, the memory system first determines the address for the next element. It performs a RAS and CAS, as above, and after the appropriate delay, the data appears on the SDRAM data bus. The controller then generates the next element’s address by adding the stride to the current address, and repeats this operation sequence until the entire memory vector has been read.

This example illustrates that when each word is accessed individually, a serial gather operation incurs a latency much larger than that of a cacheline fill. Fortunately, the latency of such a gather can be reduced by:

- issuing addresses to SDRAMs in parallel to overlap several RAS/CAS latencies for better throughput, and
- hiding or avoiding latencies by re-ordering the sequence of operations, and making good decisions on whether to close rows or to keep them open.

To implement these optimizations, we need a method by which each bank controller can determine the addresses of vector elements in its DRAM without sequentially expanding the entire vector. The remainder of this paper describes one such method. Using this algorithm, the PVA unit shown in Figure 1 parallelizes scatter/gather operations by broadcasting a vector command to a collection of bank controllers (BCs), each of which determines independently which elements of the vector (if any) reside in the DRAM it manages. The primary advantage of our PVA over sim-



**Figure 3.** Strided memory vector access timing

ilar designs is the efficiency of our hardware algorithms for computing the subvector for which each bank is responsible.

### 3. Terminology

Three functions implement the core of our scheduling scheme, *DecodeBank()*, *FirstHit()*, and *NextHit()*. Before deriving their implementations, we first define some terminology. Division denotes integer division. References to vectors denote memory vectors, unless otherwise stated.

- Vectors are represented by the tuple  $V = \langle B, S, L \rangle$ , where  $V.B$  is the base address,  $V.S$  is the sequence stride, and  $V.L$  is the sequence length.
- $V[i]$  is the  $i^{th}$  element in the vector  $V$ . For example, vector  $V = \langle A, 4, 3 \rangle$  designates elements  $A[0]$ ,  $A[4]$ , and  $A[8]$ , where  $V[0] = A[0]$ ,  $V[1] = A[4]$ , etc.
- Let  $M$  be the number of memory banks, such that  $M = 2^m$ .
- Let  $N$  be the number of consecutive words of memory exported by each of the  $M$  banks, i.e., the bank interleave factor, such that  $N = 2^n$ . We refer to these  $N$  consecutive words as a *block*.
- *DecodeBank(addr)* returns the bank number for an address  $addr$ , and can be implemented by the bit-select operation  $(addr \gg n) \bmod M$ .
- *FirstHit(V, b)* takes a vector  $V$  and a bank  $b$  and returns either the index of the first element of  $V$  that hits in  $b$  or a value indicating that no elements hit.
- *NextHit(S)* returns an increment  $\delta$  such that if a bank holds  $V[n]$ , it also holds  $V[n + \delta]$ .
- Let  $d$  be the distance between a given bank  $b$  and the bank holding the vector’s base address,  $V.B$ . Then  $d = (\text{DecodeBank}(V.B) - b + M) \bmod M$ . This is implemented as an  $m$  bit unsigned subtraction that ignores underflow.
- Let  $\Delta b$  represent the number of banks skipped between any two consecutive elements  $V[i]$  and  $V[i+1]$ .  $\Delta b = (V.S \bmod NM) / N$ .
- Let  $\theta$  be the block offset of  $V$ ’s first element, thus  $\theta = V.B \bmod N$ .
- Let  $\Delta\theta$  be the difference in offset within their respective blocks between any two consecutive elements  $V[i]$  and  $V[i+1]$ , i.e.,  $\Delta\theta = (V.S \bmod NM) \bmod N$ . The elements may reside within the same block if the stride is small compared to  $N$ , or they may reside in separate blocks on different banks.

## 4. Basic PVA Design

This section describes a PVA Bank Controller design and a recursive *FirstHit()* algorithm for block interleaving schemes. To perform a vector gather, the VCU issues a vector command of the form  $V = \langle B, S, L \rangle$  on the vector bus. In response, each BC performs the following operations:

1. Compute  $i = FirstHit(V, b)$ , where  $b$  is the number of this BC's bank. If there is no hit, break.
2. Compute  $\delta = NextHit(V, S)$ .
3. Compute  $Addr = V.B + V.S * i$ .
4. While the end of the vector has not been reached, do:
  - (a) Access the memory location  $Addr$ .
  - (b) Compute the next  $Addr = Addr + V.S * \delta$ .

The key to performing these operations quickly is an efficient *FirstHit()* algorithm.

### 4.1 *FirstHit(V, b)* Functionality

To build intuition behind the operation of the *FirstHit(V, b)* calculation, we break it into three cases, and then illustrate these with examples.

**Case 0:**  $DecodeBank(V, B) = b$

In this case,  $V[0]$  resides in bank  $b$ , so  $FirstHit(V, b) = 0$ .

**Case 1:**  $DecodeBank(V, B) \neq b$  and  $\Delta\theta = 0$

When the vector stride is an integer multiple of  $N$ , the vector always hits at the same offset ( $\theta$ ) within the blocks on the different banks. If  $V[0]$  resides in bank  $b'$ , then  $V[1]$  hits in bank  $(b' + \Delta b) \bmod M$ ,  $V[2]$  in  $(b' + 2\Delta b) \bmod M$ , and so on. By the properties of modulo arithmetic,  $(b' + n * \Delta b) \bmod M$  represents a repeating sequence with a period of at most  $M$  for  $n = 0, 1, \dots, \infty$ . Hence, when  $\Delta\theta = 0$ , the implementation of *FirstHit(V, b)* takes one of two forms:

**Case 1.1:**  $V.L > d$  and  $\Delta b$  divides  $d$   
In this case,  $FirstHit(V, b) = d / \Delta b$ .

**Case 1.2:**  $V.L \leq d$  or  $\Delta b$  does not divide  $d$   
For all other  $FirstHit(V, b) = no\ hit$ .

**Case 2:**  $DecodeBank(V, B) \neq b$  and  $N > \Delta\theta > 0$

If  $V[0]$  is contained in bank  $b'$  at an offset of  $\theta$ , then  $V[1]$  is in bank  $(b' + \Delta b + (\theta + \Delta\theta)/N) \bmod M$ ,  $V[2]$  in  $(b' + 2\Delta b + (\theta + 2\Delta\theta)/N) \bmod M$ , etc. and  $V[i]$  in  $(b' + i\Delta b + (\theta + i\Delta\theta)/N) \bmod M$ . There are two sub-cases.

**Case 2.1:**  $\theta + (V.L - 1) * \Delta\theta < N$

In this case the  $\Delta\theta$ s never sum to  $N$ . So the sequence of banks in Case 2.1 is the same as for case 1 and we may ignore the effect of  $\Delta\theta$  on *FirstHit(V, b)* and use the same procedure as in Case 1.

**Case 2.2:**  $\theta + (V.L - 1) * \Delta\theta \geq N$

Whenever the running sum of the  $\Delta\theta$ s reaches  $N$ , the bank as calculated by Cases 1 and 2.1 needs to be incremented. This increment can cause the calculation to shift between multiple cyclic sequences.

The following examples illustrate the more interesting cases for  $M = 8$  and  $N = 4$ .

1. Let  $B = 0, S = 8$ , and  $L = 16$ .  
This is Case 1, with  $\theta = 0, \Delta\theta = 0$ , and  $\Delta b = 2$ .  
This vector hits the repeating sequence of banks 0, 2, 4, 6, 0, 2, 4, 6, ...
2. Let  $B = 0, S = 9$ , and  $L = 4$ .  
This is Case 2.1, with  $\theta = 0, \Delta\theta = 1$ , and  $\Delta b = 2$ .  
This vector hits banks 0, 2, 4, 6.
3. Let  $B = 0, S = 9$ , and  $L = 16$ .  
This is Case 2.2, with  $\theta = 0, \Delta\theta = 1$ , and  $\Delta b = 2$ .

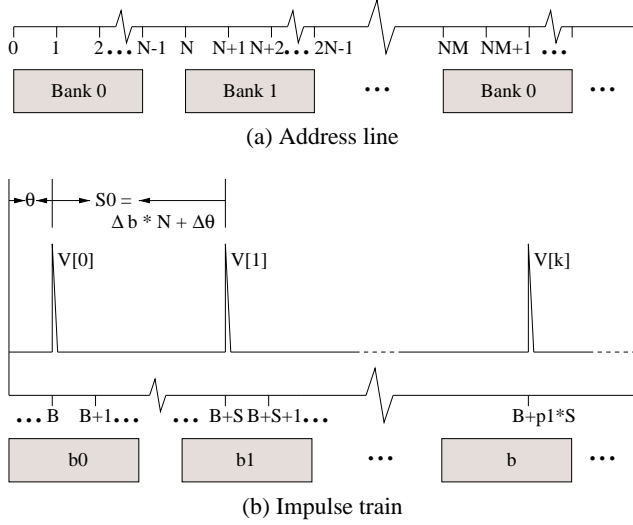
Note that when the cumulative effect of  $\Delta\theta$  exceeds  $N$ , there is a shift from the initial sequence of 0, 2, 4, 6 to the sequence 1, 3, 5, 7. For some values of  $B, S$  and  $L$ , the banks hit by the vector may cycle through several such sequences or may have multiple sequences interleaved. It is this case that complicates the definition of the *FirstHit()* algorithm.

### 4.2 Derivation of *FirstHit(V, b)*

In this section we use the insights from the case-by-case examination of *FirstHit()* to derive a generic algorithm for all cases. Figure 4 provides a graphical analogy to lend intuition to the derivation. Figure 4(a) depicts the address space as if it were a number line. The bank that a set of  $N$  consecutive words belong to appears below the address line as a gray rectangle. The banks repeat after  $NM$  words. Figure 4(b) shows a section of the address line along with a vector  $V$  with base address  $B$  and stride  $S$ .

Let  $S_0 = V.S \bmod NM$ . Bank access patterns with strides over  $NM$  are equivalent to this factored stride. The figure therefore shows  $V$  as an impulse train of period  $S_0$ . The first impulse starts at offset  $\theta$  from the edge of the bank containing  $B$ , denoted here by  $b_0$ . The bank hit next is denoted by  $b_1$ . Note that  $b_0$  is not the same as bank 0, but is the bank to which address  $B$  decodes. Also, note that  $b_1$  is not necessarily the bank succeeding bank  $b_0$ , but is the bank to which  $B + S$  happens to decode.

The problem of *FirstHit(V, b)* is that of finding the integral period  $p_1$  of the impulse train during which the impulse is within a block of  $N$  words belonging to bank  $b$ . The difference in the periodicities of the banks and the impulse train may cause the train to pass bank  $b$  several times before it actually enters a block of words belonging to  $b$ . For values of  $S$  less than  $N$ , then the problem can be solved trivially by dividing the distance between  $b$  and  $b_0$  on the address line by  $S_0$ , but when  $S > N$ , the problem becomes more difficult. Our algorithm splits the original problem into similar subproblems with smaller and smaller values for  $S_0, S_1, \dots, S_n$ , until we reach a value  $S_n < N$ . At this point, we solve the in-equation trivially, and we propagate the values up to the larger sub-



**Figure 4.** Visualizing *FirstHit()*

problems. This process continues until we have solved the original problem.

Furthering the analogy, let  $p_1$  be the impulse train's least integral period during which it enters a region belonging to bank  $b$  after passing by bank  $b$ ,  $p_2$  times without hitting that bank. Thus  $p_1$  is the least and  $p_2$  the greatest integer of its kind. The position of the impulse on its  $p_1$ th interval will be  $\theta + p_1 * S_0$ . The relative distance between it and the beginning of a block belonging to bank  $b$  after allowing for the fact that the impulse train passed by  $p_2$  times and that the modulo distance between  $b$  and  $b_0$  is  $d * N$  is  $\theta + p_1 S_0 - p_2 N M - d N$ . For a hit to happen, this distance should be less than  $N$ . Expressing this as an inequality, we get:

$$0 \leq \theta + p_1 S_0 - p_2 N M - d N < N \quad (0)$$

Let  $\gamma = \theta - d N$ . We need to find  $p_1$  and  $p_2$  such that:

$$\begin{aligned} 0 &\leq \gamma + p_1 S_0 - p_2 N M < N & (1) \\ -\gamma &\leq p_1 S_0 - p_2 N M < N - \gamma \\ \gamma &\geq p_2 N M - p_1 S_0 > \gamma - N \\ S_0 + \gamma &\geq p_2 N M - (p_1 - 1) S_0 > S_0 + \gamma - N & (2) \end{aligned}$$

Since we have two variables to solve for in one inequality, we need to eliminate one. We solve for  $p_1$  and  $p_2$  one at a time. If

$$S_0 > S_0 + \gamma - N \quad (3)$$

we can substitute (2) with:

$$S_0 + \gamma \geq p_2 N M \bmod S_0 > S_0 + \gamma - N \quad (4)$$

Recall that  $\theta = V.B \bmod N$ . Therefore:

$$\theta < N \quad (5)$$

Inequality (3) is satisfied if  $0 > \gamma - N$ , which is true iff:

$$\begin{aligned} N &> \gamma \\ N &> \theta - d N \\ (d + 1) N &> \theta \end{aligned}$$

Therefore, since  $N > \theta$  by inequality (5), inequality (3) must be true. We can now solve for  $p_2$  using the simplified inequality (4), and then we can use the value of  $p_2$  to solve for  $p_1$ . In other words, after solving for  $p_2$ , we set  $p_1 = p_2 N M / S_0$  or  $p_1 = 1 + p_2 N M / S_0$ , and one of these two values will satisfy (2).

Recall that  $p_2 N M \bmod S_0 = p_2 (N M \bmod S_0) \bmod S_0$ . Substituting  $S_1 = N M \bmod S_0$ , we need to solve  $S + \gamma \geq p_2 S_1 \bmod S_0 > S_0 + \gamma - N$ , for which we need to find  $p_3$  such that:

$$\begin{aligned} S_0 + \gamma &\geq p_2 S_1 - p_3 S_0 > S_0 + \gamma - N \\ -S_0 - \gamma &\leq p_3 S_0 - p_2 S_1 < -S_0 - \gamma + N \\ -\gamma &\leq (p_3 + 1) S_0 - p_2 S_1 < -\gamma + N \\ 0 &\leq \gamma + (p_3 + 1) S_0 - p_2 S_1 < N & (6) \end{aligned}$$

Notice that (6) is of the same format as (1). At this point the same algorithm can be recursively applied. Recursive application terminates whenever we have an  $S_i$  such that  $S_i < N$  at steps (1) or (6). When the subproblem is solved, the value of the  $p_i$ s may be propagated back to solve the preceding subproblem, and so on until the value of  $p_1$  is finally obtained.

Since each  $S_i = S_{i-2} \bmod S_{i-1}$ , the  $S_i$ s reduce monotonically. The algorithm will therefore always terminate when there is a hit. A simpler version of this algorithm with  $\gamma = \theta$  can be used for *NextHit()*.

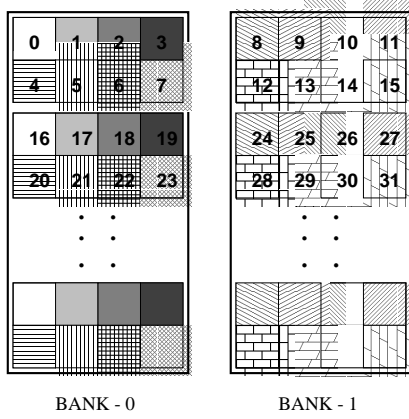
### 4.3 Efficiency

The recursive nature of the *FirstHit()* algorithm derived above is not an impediment to hardware implementation, since the algorithm terminates at the second level for most inputs corresponding to memory systems with reasonable values of  $N$  and  $M$ . The recursion can be unraveled by replicating the data path, which is equivalent to "unrolling" the algorithm once.

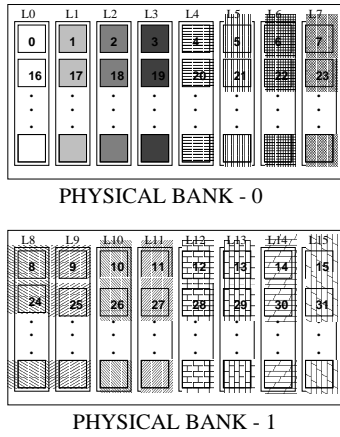
Unfortunately, this algorithm is not suitable for a *fast* hardware implementation because it contains several division and modulo operations by numbers that may not be powers of two. The problem arises from Case 2, above, when vector strides are not integer multiples of the block width  $N$ . Since this straightforward, analytical solution for *FirstHit()* does not yield a low latency implementation for cache-line interleaved memories, i.e., with  $N > 1$ , we now transform the problem to one that can be implemented in fast hardware at a reasonable cost.

## 5. Improved PVA Design

We can convert all possible cases of *FirstHit()* to either of the two simple cases from Section 4.1 by changing the way we



**Figure 5.** Physical View of Memory



**Figure 6.** Logical View of Memory

view memory. Assume that memory is physical divided into two eight-word wide banks of SDRAM (i.e.,  $M = 2$  and  $N = 8$ ), as illustrated in Figure 5. Recall that it is difficult for each bank to quickly calculate  $FirstHit()$  when the stride of a given vector is not an integer multiple of  $N$  (and thus  $\Delta\theta$  is non-zero).

If, instead, we treat this memory system as if it had sixteen one-word wide banks (i.e.,  $M = 16$  and  $N = 1$ ), then  $\Delta\theta$  will always be zero. To see this, recall that  $\Delta\theta$  is the change in block offset between each vector element and its predecessor. If, however, every block is only one-word wide, then each vector element must start at the same offset (zero) in each block. Thus, if we treat the memory system as if it were composed of many word-interleaved banks, we eliminate the tricky Case 2 from the algorithm derived in Section 4.1. Figure 6 illustrates this logical view of the memory system. In general, a memory system composed of  $M$  banks of memory each  $N$  words wide can be treated as a logical memory system composed of  $MN$  logical banks (denoted  $L_0$  to  $L_{WNM-1}$ ), each one-word wide.

We can now simplify the  $FirstHit()$  algorithm, and the resulting hardware, by giving each logical bank its own  $FirstHit()$  logic, which need only handle Case 0 and Case 1 from Section 4.1 (since  $\Delta\theta = 0$  when  $N = 1$ ). This logical transformation requires

us to build  $NM$  copies of the  $FirstHit()$  logic, whereas we required only  $M$  copies in our initial design, but the  $FirstHit()$  logic required in this case is much simpler. Our empirical observation is that the size of the  $NM$  copies of the simplified  $FirstHit()$  logic is comparable to that of the  $M$  copies of the initial design's logic.

We now derive improved algorithms for  $FirstHit()$  and  $NextHit()$  for word interleaved memory systems. For the purposes of this discussion, memory systems with  $N$  greater than one are equivalent to a memory system with  $N = 1$  where  $N$  banks sharing a common bus. The parameter  $N = 1$  is assumed in the following discussion.

Let  $b_0 = DecodeBank(V.B)$ , i.e.,  $b_0$  is the bank where the first element of  $V$  resides. Recall that  $d$  is the distance module  $M$  between some bank  $b$  and  $b_0$ .

**LEMMA 5.1.** *To find the bank access pattern of a vector  $V$  with stride  $V.S$ , it suffices to consider the bank access pattern for stride  $V.S \bmod M$ .*

**Proof:** Let  $S = q_s M + S_m$ , where  $S_m = S \bmod M$  and  $q_s$  is some integer. Let  $b_0 = DecodeBank(V.B)$ . For vector element  $V[n]$  to hit a bank at modulo distance  $d$  from  $b_0$ , it must be the case that  $(n * S) \bmod M = d$ . Therefore, for some integer  $q_d$  :

$$\begin{aligned} n * S &= q_d * M + d \\ n * (q_s M + S_m) &= q_d * M + d \\ n * S_m &= (q_d - n * q_s) M + d \\ \text{Therefore } (n * S_m) \bmod M &= d. \end{aligned}$$

Thus, if vector  $V$  with stride  $V.S$  hits bank  $b$  at distance  $d$  for  $V[n]$ , then vector  $V_1$  with stride  $V_1.S_1$ , where  $V_1.S_1 = (V.S \bmod M)$ , will also hit  $b$  for  $V_1[n]$ .

Lemma 5.1 says that only the least significant  $m$  bits of the stride  $(V.S)$  are required to find the bank access pattern of  $V$ . When element  $V[n]$  of vector  $V$  with stride  $V.S$  hits bank  $b$ , then element  $V_1[n]$  of vector  $V_1$  with stride  $V_1.S_1 = (V.S \bmod M)$  will also hit bank  $b$ . Henceforth, references to any stride  $S$  will denote only the least significant  $m$  bits of  $V.S$ .

**Definition:** Every stride  $S$  can be written as  $\sigma * 2^s$ , where  $\sigma$  is odd. Using this notation,  $s$  is the number of least significant zeroes in  $S$ 's binary representation [6].

For instance, when  $S = 6 = 3 * 2^1$ , then  $\sigma = 3$  and  $s = 1$ . When  $S = 7 = 7 * 2^0$ ,  $\sigma = 7$  and  $s = 0$ , and when  $S = 8 = 1 * 2^3$ ,  $\sigma = 1$  and  $s = 3$ .

**LEMMA 5.2.** *Vector  $V$  hits bank  $b$  iff  $d$  is some multiple of  $2^s$ .*

**Proof:** Assume that at least one element  $V[n]$  of vector  $V$  hits on bank  $b$ . For this to be true,  $(n * S) \bmod M = d$ . Therefore, for some integer  $q$ :

$$\begin{aligned}
n * S &= q * M + d \\
n * \sigma * 2^s &= q * M + d = q * 2^m + d \\
d &= n * \sigma * 2^s - q * 2^m = 2^s (n * \sigma - q 2^{m-s})
\end{aligned}$$

Thus, if some element  $n$  of vector  $V$  hits on bank  $b$ , then  $d$  is a multiple of  $2^s$ .

Lemma 5.2 says that after the initial hit on bank  $b_0$ , every  $2^s$ th bank will have a hit. Note that the index of the vector may not necessarily follow the same sequence as that of the banks that are hit. The lemma only guarantees that there will be a hit. For example, if  $S = 12$ , and thus  $s = 2$  (because  $12 = 3 * 2^2$ ), then only every 4<sup>th</sup> bank controller may contain an element of the vector, starting with  $b_0$  and wrapping modulo  $M$ . Note that even though every  $2^s$  banks may contain an element of the vector, this does *not* mean that consecutive elements of the vector will hit every  $2^s$  banks. Rather, *some* element(s) will correspond to each such bank. For example, if  $M = 16$ , consecutive elements of a vector of stride 10 ( $s = 1$ ) hit in banks 2, 12, 6, 0, 10, 4, 14, 8, 2, etc.

Lemmas 5.1 and 5.2 let us derive extremely efficient algorithms for *FirstHit()* and *NextHit()*.

Let  $K_i$  be the smallest vector index that hits a bank  $b$  at a distance modulo  $M$  of  $d = i * 2^s$  from  $b_0$ . In particular, let  $K_1$  be the smallest vector index that hits a bank  $b$  at a distance  $d = 2^s$  from  $b_0$ . Since  $V[K_i]$  hits  $b$  we have:

$$(K_i * S) \bmod M = d$$

$K_i * \sigma * 2^s = (q_i * 2^m + i * 2^s)$  where  $q_i$  is the least integer such that  $M$  divides  $K_i * S$  producing remainder  $d$ . Therefore,

$$K_i = \frac{(q_i * 2^{m-s} + i)}{\sigma}$$

Also, by definition, for  $K_1$ , distance  $d = 1 * 2^s$ .

Therefore,

$$K_1 = \frac{(q_1 * 2^{m-s} + 1)}{\sigma}$$

where  $q_1$  is the least integer such that  $\sigma$  evenly divides  $q_1 * 2^{m-s} + 1$ .

**THEOREM 5.3.** *FirstHit*( $V, b$ ) =  $K_i = (K_1 * i) \bmod 2^{m-s}$ .

**Proof:** By induction.

**Basis:**  $K_1 = K_1 \bmod 2^{m-s}$ . This is equivalent to proving that  $K_1 < 2^{m-s}$ . By lemma 5.2, the vector will hit banks at modulo distance 0,  $2^s$ ,  $2 * 2^s$ ,  $3 * 2^s$ , ... from bank  $b_0$ . Every bank hit will be revisited within  $M/2^s = 2^{m-s}$  strides. The vector indices may not be continuous, but the change in index before  $b_0$  is revisited cannot exceed  $2^{m-s}$ . Hence  $K_1 < 2^{m-s}$ . QED.

**Induction step:** Assume that the result holds for  $i = r$ .

Then  $K_r = \frac{(q_r * 2^{m-s} + r)}{\sigma} = (K_1 * r) \bmod 2^{m-s}$ , where  $q_r$  is the least integer such that  $\sigma$  evenly divides  $q_r * 2^{m-s} + r$ .

This means that  $K_1 * r = Q_r * 2^{m-s} + K_r = Q_r * 2^{m-s} + \frac{(q_r * 2^{m-s} + r)}{\sigma}$  for some integer  $Q_r$ .

Therefore:  $K_1 * r + K_1 = Q_r * 2^{m-s} + \frac{(q_r * 2^{m-s} + r + q_1 * 2^{m-s} + 1)}{\sigma}$ , and  $K_1 * (r + 1) = Q_r * 2^{m-s} + \frac{(q_r + q_1) * 2^{m-s} + (r + 1)}{\sigma}$ . Since  $q_1$  and  $q_r$  are the least such integers, it follows that the least integer  $q_{r+1}$  such that  $\sigma$  evenly divides  $q_{r+1} * 2^{m-s} + (r + 1)$  is  $(q_r + q_1)$ .

By the definition of  $K_i$ ,  $\frac{(q_r + q_1) * 2^{m-s} + (r + 1)}{\sigma} = K_{r+1}$ .

Therefore:  $K_1 * (r + 1) = Q_r * 2^{m-s} + K_{r+1}$ , or  $K_{r+1} = (K_1 * (r + 1)) \bmod 2^{m-s}$ .

Hence, by the principle of mathematical induction,  $K_i = (K_1 * i) \bmod 2^{m-s} \forall i > 0$ .

For the above induction to work, we must prove that the least integer  $q_{r+1}$ , such that  $\sigma$  evenly divides  $q_{r+1} * 2^{m-s} + (r + 1)$ , is  $(q_r + q_1)$ .

**Proof:** Assume there exists another number  $q_l < q_r + q_1$  such that  $\sigma$  evenly divides  $x = q_l * 2^{m-s} + (r + 1)$ . Since  $\sigma$  divides  $y = q_1 * 2^{m-s} + 1$ , it should also divide  $x - y = (q_l - q_1) * 2^{m-s} + r$ . However, since we earlier said that  $q_l < q_r + q_1$ , we have found a new number  $q_{r1} = q_l - q_1$  that is less than  $q_r$  and yet satisfies the requirement that  $\sigma$  evenly divides  $q_{r1} * 2^{m-s} + r$ . This contradicts our assumption that  $q_r$  is the least such number. Hence  $q_l$  does not exist and  $q_{r+1} = q_r + q_1$ .

**THEOREM 5.4.** *NextHit*( $S$ ) =  $\delta = 2^{m-s}$ .

**Proof:** Let the bank at distance  $d = i * 2^s$  have a hit. Also, let a bank at distance  $d_1 = j * 2^s$  have a hit. If, both of these are the same bank, then by the definition of *NextHit*( $S$ ),  $j = i + \delta$ . In that case,  $d_1 - d = M$ , since every bank is at a modulo distance of  $M$  from itself. Hence we have,  $M = \delta * 2^s$ . Therefore,  $\delta = M/2^s = 2^{m-s}$ .

## 6. Implementation Strategies for *FirstHit()* and *NextHit()*

Using Theorems 5.3 and 5.4 and given  $b, M, V.S \bmod M$ , and  $V.B \bmod M$  as inputs, each Bank Controller can independently determine the sub-vector elements for which it is responsible. Several options exist for implementing *FirstHit()* in hardware; which makes most sense depends on the parameters of the memory organization. Note that the values of  $K_i$  can be calculated in advance for every legal combination of  $M, V.S \bmod M$ , and  $V.B \bmod M$ . If  $M$  is sufficiently small, an efficient PLA (programmable logic array) implementation could take  $d = (b - b_0) \bmod S$  and  $V.S$  as inputs and return  $K_i$ . Larger configurations could use a PLA that takes  $S$  and returns the corresponding  $K_1$  value, and then multiply  $K_1$  by a small integer  $i$  to generate  $K_i$ . Block-interleaved systems with small interleave factor  $N$  could use  $N$  copies of the *FirstHit()* logic (with either of the above organizations), or could include one instance of the *FirstHit()* logic to compute  $K_i$  for the first hit within the block, and then use an adder to generate each subsequent  $K_{i+1}$ . The various designs trade hardware

space for lower latency and greater parallelism.  $NextHit()$  can be implemented using a small PLA that takes  $S$  as input and returns  $2^{m-s}$  (i.e.,  $\delta$ ). Optionally, this value may be encoded as part of the  $FirstHit()$  PLA. In general, most of the variables used to explain the functional operation of these components will never be calculated explicitly; instead, their values will be compiled into the circuitry in the form of lookup tables.

Given appropriate hardware implementations of  $FirstHit()$  and  $NextHit()$ , the BC for bank  $b$  performs the following operations (concurrently, where possible):

1. Calculate  $b_0 = DecodeBank(V.B)$  via a simple bit-select operation.
2. Find  $NextHit(S) = \delta = 2^{m-s}$  via a PLA lookup.
3. Calculate  $d = (b - b_0) \bmod M$  via an integer subtraction-without-underflow operation.
4. Determine whether or not  $d$  is a multiple of  $2^s$  via a table lookup. If it is, return the  $K_1$  or  $K_i$  value corresponding to stride  $V.S$ . If not, return a “no hit” value, to indicate that  $b$  does not contain any elements of  $V$ .
5. If  $b$  contains elements of  $V$ ,  $FirstHit(V, b)$  can either be determined via the PLA lookup in the previous step or be computed from  $K_1$  as  $(K_1 * i) \bmod 2^{m-s}$ . In the latter case, this only involves selecting the least significant  $m - s$  bits of  $K_1 * (d \gg s)$ . If  $S$  is a power of two, this is simply a shift and mask. For other strides, this requires a small integer multiply and mask.
6. Issue the address  $addr = V.B + V.S * FirstHit(V, b)$ .
7. Repeatedly calculate and issue the address  $addr = addr + (V.S \ll (m - s))$  using a shift and add.

## 7. Some Practical Issues

This section discusses issues of scaling the PVA memory system and interactions with virtual memory.

### 7.1 Scaling Memory System Capacity

To scale the vector memory system, we hold  $M$  and  $N$  fixed while adding DRAM chips to extend the physical address space. This may be done in several ways. One method would use a Bank Controller for each slot where memory could be added. All BCs corresponding to the same physical bank number would operate in parallel and would be identical. Simple address decoding logic would be used along with the address generated by the BC to enable the memory’s chip select signal only when the address issued resides in that particular memory. Another method would use a single BC for multiple slots, but maintain different current-row registers to track the hot rows inside the different chips forming a single physical bank.

### 7.2 Scaling the Number of Banks

The ability to scale the PVA unit to a large number of banks depends on the implementation of  $FirstHit()$ . For systems that

use a PLA to compute the index of the first element of  $V$  that hits in  $b$ , the complexity of the PLA grows as the square of the number of banks. This limits the effective size of such a design to around 16 banks. For systems with a small number of banks interleaved at block-size  $N$ , replicating the  $FirstHit()$  logic  $N$  times in each BC is optimal. For very large memory systems, regardless of their interleaving factor, it is best to implement a PLA that generates  $K_1$  and to add a small amount of logic to then calculate  $K_i$ . Complexity of the PLA in this design increases approximately linearly with the number of banks. The remaining hardware is unchanged.

## 7.3 Interaction with the Paging Scheme

The opportunity to do parallel fetches for long vectors exists only when a significant part of the vector is contiguous in physical memory. Optimal performance requires each frequently accessed, large data structure to fit within one superpage, in which case the memory controller can issue vector operations that are long enough to gather/scatter a whole cache line on the vector bus. If the structure cannot reside in one superpage, then the memory controller can split a single vector operation into multiple operations such that each sub-vector resides in a single superpage. For a given vector  $V$ , the memory controller could find the page offset of  $V.B$  and divide it by  $V.S$  to determine the number of elements in the page, after which it would issue a single vector bus operation for those elements. Unfortunately, the division required for this exact solution is expensive in hardware. A more reasonable approach is to compute a lower bound on the number of elements in the page and issue a vector bus operation for that many elements. We can calculate this bound by rounding the stride up to the next power of two and using this to compute a minimum number of elements residing between the first element in the vector and the end of the (super)page. This converts an integer division to a shift.

## 8. Related Work

In this section we limit our discussion to related work that pertains to *vector* accesses, and especially to loading vector data from DRAM. In his studies of compile-time memory optimizations for streams, Moyer defines *access scheduling* as those techniques that reduce load/store interlock delay by overlapping computation with memory latency [14]. In contrast, Moyer defines *access ordering* to be any technique that changes the order of memory requests to increase memory system performance. He develops compiler algorithms that unroll loops and group accesses within each stream to amortize the cost of each DRAM page miss over several references to that page [14].

The Stream Memory Controller built at the University of Virginia extends Moyer’s work to implement access ordering in hardware dynamically at run time [13]. The SMC reorders stream accesses to avoid bank conflicts and bus turnarounds, and to exploit locality of reference within the row buffers of fast-page mode DRAM components. The simple reordering scheme used in this proof-of-concept serial, gathering memory controller targets a system with only two interleaved banks, and thus the SMC as implemented cannot fully exploit the parallelism in a system with many banks. Like the SMC, our PVA unit strives to exploit bank parallelism and locality of reference while avoiding bus turnaround delays, but it does so via very different mechanisms.



A few commercial memory controllers use prediction techniques to attempt to manage DRAM resources intelligently. The Alpha 21174 memory controller implements a simple access scheduling mechanism for an environment in which nothing is known about future access patterns (and all accesses are treated as random cache-line fills) [17]. A four-bit predictor tracks whether accesses hit or miss the most recent row in each row buffer, and the controller leaves a row open only when a hit is predicted. Similarly, Rambus's RMC2 memory controller attempts to provide optimal channel bandwidth by using timing and logical constraints to determine when precharge cycles may be skipped [16].

Valero *et al.* propose efficient hardware to dynamically avoid bank conflicts in vector processors by accessing vector elements out of order. They analyze this system first for single vectors [18], and then extend the work for multiple vectors [15]. del Corral and Llaberia analyze a related hardware scheme for avoiding bank conflicts among multiple vectors in complex memory systems [8]. These access-scheduling schemes focus on memory systems composed of SRAM components (with uniform access time), and thus they do not address precharge delays and open-row policies.

Corbal *et al.* extend these hardware vector access designs to a Command Vector Memory System [6] (CVMS) for out-of-order vector processors with SDRAM memories. The CVMS exploits parallelism and locality of reference to improve the effective bandwidth for vector accesses from out-of-order vector processors with dual-banked SDRAM memories. Rather than sending individual requests to specific devices, the CVMS broadcasts commands requesting multiple independent words, a design idea that we adopted. Section controllers receive the broadcasts, compute subcommands for the portion of the data for which they are responsible, and then issue the addresses to the memory chips under their control. The memory subsystem orders requests to each dual-banked device, attempting to overlap precharge operations to each internal SDRAM bank with access operations to the other. Simulation results demonstrate performance improvements of 15% to 54% compared to a serial memory controller. At the behavioral level, our bank controllers resemble CVMS section controllers, but the specific hardware design and parallel access algorithm is substantially different. For instance, Corbal *et al.* state that for strides that are not powers of two, 15 memory cycles are required to generate the subcommands [6]. In contrast, the PVA requires at most five memory cycles to generate subcommands for strides that are not powers of two. Further architectural differences are discussed in our other reports [12, 11].

Hsu and Smith study interleaving schemes for page-mode DRAM memory in vector machines [10]. They demonstrate the benefits of exploiting spatial locality within dynamic memory components, finding that cache-line interleaving and block-interleaving are superior to low-order interleaving for many vector applications. In their experiments, cache-line interleaving performances nearly identically to block-interleaving for a moderate number of banks (16-64), beyond which block-interleaving performs better. Unfortunately, block interleaving complicates address arithmetic. It's possible that low-order interleaving could outperform block interleaving when combined with access ordering and scheduling techniques, but we have not yet tested this hypothesis.

Generating communication sets and local addresses on individual nodes involved in a parallel computation represents a more complicated version of the PVA problem of identifying which elements reside where within a bank. Chatterjee *et al.* demonstrate an HPF array storage scheme with a *cyclic(k)* for which local address generation can be characterized by a finite state machine with at most  $k$  states, and they present fast algorithms for computing these state machines [4]. Their algorithms are implemented in software, and thus can afford to be much more complex than those implemented in the PVA. Nonetheless, their experimental results demonstrate that their FSM solution requires acceptable preprocessing time and exhibits little run-time overhead.

## 9. Conclusion

This paper presents the mathematical foundations for a *Parallel Vector Access* that can efficiently gather sparse data from a banked SDRAM memory system. Earlier performance evaluations [12] showed that the resulting PVA design gathers sparse vectors up to 32.8 times faster than a conventional memory system and 3.3 times faster than a pipelined vector unit, without affecting the performance of normal cache-line fills.

More specifically, we have presented a series of algorithms that allow independent SDRAM bank controllers to determine in a small number of cycles which elements (if any) from a base-stride vector reside in their bank. These algorithms do not require a serial expansion of the vector into its component units, and can be efficiently implemented in hardware. We showed that treating each bank of SDRAM as if it were logically an array of one-word wide SDRAM banks greatly simplifies the algorithm, which in turn simplifies and improves the performance of the hardware used to implement it.

We are extending our PVA work in a number of ways. We are refining our implementation model and evaluating it in a detailed architecture-level simulator to determine its impact on large applications. We are extending its design to handle vector-indirect scatter-gather operations, in addition to base-stride operations. Finally, we are investigating the potential value of vector scatter-gather support in the context of emerging multimedia instructions sets (e.g., MMX).

## Acknowledgments

The authors thank Ganesh Gopalakrishnan for his contributions to the early stages of this project. Mateo Valero and members of the UPC Computer Science Department generously provided computing resources during preparation of our initial submission. This work was supported in part by the National Science Foundation (NSF) under Grant Number CDA-96-23614, and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) agreement F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views, findings, and conclusions or recommendations contained herein are those of the authors, and should not be interpreted as necessarily representing the official views, policies or endorsements, either express or implied, of DARPA, AFRL, the NSF, or the U.S. Government.

## 10. REFERENCES

- [1] T. Austin, D. Pnevmatikatos, and G. Sohi. Fast address calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 158–167, 1995.
- [2] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboem, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 54–63, 1999.
- [3] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.
- [4] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, Apr. 1995.
- [5] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 68–77, Oct. 1998.
- [6] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. Technical Report UPC-DAC-1999-5, Universitat Politècnica de Catalunya, Jan. 1999.
- [7] Cray Research, Inc. *CRAY T3D System Architecture Overview*, hr-04033 edition, Sept. 1993.
- [8] A. del Corral and J. Llberia. Access order to avoid inter-vector conflicts in complex memory systems. In *Proceedings of the Ninth ACM/IEEE International Parallel Processing Symposium (IPPS)*, 1995.
- [9] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 196–203, 1997.
- [10] W. Hsu and J. Smith. Performance of cached DRAM organizations in vector supercomputers. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 327–336, May 1993.
- [11] B. Mathew. Parallel vector access: A technique for improving memory system performance. Master’s thesis, University of Utah Department of Computer Science, Jan. 2000.
- [12] B. Mathew, S. McKee, J. Carter, and A. Davis. Design of a parallel vector access unit for sdram memories. In *Proceedings of the Sixth Annual Symposium on High Performance Computer Architecture*, pages 39–48, Jan. 2000.
- [13] S. McKee, et al. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the 1996 International Conference on Supercomputing*, May 1996.
- [14] S. Moyer. *Access Ordering Algorithms and Effective Memory Bandwidth*. PhD thesis, University of Virginia, May 1993.
- [15] M. Peiron, M. Valero, E. Ayguade, and T. Lang. Vector multiprocessors with arbitrated memory access. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.
- [16] Rambus, Inc. RMC2 Data Sheet Advance Information. [http://www.rambus.com/developer/downloads/rmc2\\_overview.pdf](http://www.rambus.com/developer/downloads/rmc2_overview.pdf), August 1999.
- [17] R. Schumann. Design of the 21174 memory controller for DIGITAL personal workstations. *Digital Technical Journal*, 9(2), Jan. 1997.
- [18] M. Valero, T. Lang, J. Llberia, M. Peiron, E. Ayguade, and J. Navarro. Increasing the number of strides for conflict-free vector access. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 372–381, Gold Coast, Australia, 1992.
- [19] L. Zhang, J. Carter, W. Hsieh, and S. McKee. Memory system support for image processing. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.