PARALLEL VECTOR ACCESS: A TECHNIQUE FOR IMPROVING MEMORY SYSTEM PERFORMANCE

by

Binu K. Mathew

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

May 2000

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Binu K. Mathew

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Al Davis
	Sally A. McKee
	John B. Carter

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Binu K. Mathew in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Al Davis
Chair, Supervisory Committee

Approved for the Major Department

Robert Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Parallel Vector Access (PVA) is a technique that exploits the regularity of vector or stream accesses to perform them efficiently in parallel on a multibank memory system. The performance of vector applications may be improved by a memory controller that performs scatter/gather operations, so that only the vector or stream elements that are accessed by the application are transmitted across the system bus. These scatter/gather operations can be accelerated by broadcasting vector operations in parallel to all memory banks, each of which implements an algorithm to determine which elements of the requested vector it contains. This thesis presents the mathematical foundations behind one such algorithm for efficient parallel access of base-stride vectors on both word interleaved and cache-line interleaved memory systems. The design of a memory controller subcomponent that uses the PVA algorithm to improve the performance of applications with strided access patterns is described. The hardware implementation issues behind such a memory controller are investigated. Gate level simulation on vector kernels demonstrates that this parallel approach allows the PVA to load elements up to 32.8 times faster than a conventional SDRAM memory system and 3.3 times faster than a pipelined vector unit, without hurting normal cache-line fill performance.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Al Davis, for starting me off on the mathematical basis of this work and for gently correcting the naivete of a novice hardware designer on numerous occasions. I am grateful to Dr. John Carter and Dr. Sally McKee for their constant support and encouragement, for helpful discussions, for providing references and for help with writing. Discussions with Mike Parker and Lambert Schaelicke on aspects of the PVA design and its evaluation proved invaluable. Dr. Ganesh Gopalakrishnan helped with the IKOS equipment and the Verilog model. Dr. Wilson Hsieh, Lixin Zhang, and the other members of the Impulse and Avalanche projects helped shape this work, and Gordon Kindlmann helped with the figures. Jesus Corbal of the Computer Architecture group at the Universitat Politecnica de Catalunya (UPC) in Barcelona helped review the PVA algorithm.

CHAPTER 1

INTRODUCTION

Processor speeds are increasing much faster than memory speeds, so memory latency and bandwidth limitations prevent many applications from making effective use of the tremendous computing power of modern microprocessors. The traditional approach to solving this mismatch has been to structure memory hierarchically by adding several levels of fast cache memory between the processor and the real memory. The fast cache memory improves overall performance by taking advantage of spatial and temporal locality to reduce average load/store latency. However caches may not improve the performance of irregular applications that have poor locality. They might in fact exacerbate the problem by loading and storing entire cachelines even when the application uses only a few of the memory words in a cacheline. Moreover caches do not solve the bandwidth mismatch on the cachefill path. In cases where system bus bandwidth is the bottleneck, memory system performance can be improved only by utilizing this resource more efficiently. This thesis introduces a memory controller architecture that helps irregular applications with strided accesses achieve performance similar to those that access only dense cache-lines.

Several classes of applications that suffer from poor cache locality have predictable access patterns. Programs that operate on large multidimensional arrays are an example of this class of applications. Though modern processors generate memory operations at multiple granularities, they are filtered through the cache such that the real memory accesses are done by the cache controllers at a cacheline grain size. Hence, memory operations are seen at the DRAM at the granularity of the lowest level cacheline size. If an access to an array element misses in the cache, it will be seen by the DRAM as a cache line access. Conceptually, a cacheline request can be considered a fixed length

vector, and a memory controller serves requests to load or store fixed length vectors. Sequences of memory words that a memory controller knows how to load or store will be henceforth referred to as memory vectors. Traditionally, a memory vector has been the same as a cacheline., i.e., memory controllers know only about unit-stride vectors of length equal to the number of words in a a cacheline. When applications access their data elements in the same order as a memory vector, performance improves due to good cache and bus bandwidth utilization. When the sequence of memory elements accessed by an application belong to different memory vectors performance suffers. An example of the former case is when an application accesses an array stored in row major order along a row of the array. An example of the latter case is when the same array is accessed along a column or a diagonal. Performance loss in the latter case is due to two phenomena.

- Poor cache utilization: The application uses only some elements of a memory vector, but the whole vector occupies space in the cache. This increases the number of conflict and capacity misses.

- Poor system bus utilization: The application uses only some elements of a memory vector, but the whole vector is transferred across the system bus. Hence, the bandwidth is reduced as far as the application is concerned.

Henceforth, let us call a sequence of memory locations accessed by an application that occupies the same number of memory words as a cacheline an application vector. The performance of irregular applications suffers due to poor cache and system bus utilization because their application vectors do not match memory vectors. Memory vectors are currently unit stride vectors while application vectors may have some other pattern depending on the nature of the application. If a traditional memory controller can be extended to understand application vectors that have patterns other than unit stride, then applications with such patterns can benefit from better cache and system bus utilization.

Some of the common patterns for application vectors are:

- The elements of the application vector are accessed using nonunit, but constant stride. This occurs when an application accesses array elements along the column

of an array, accesses particular fields of an array of records, etc. We will refer to this pattern as base-stride access.

- The elements of the application vector are accessed indirectly using offsets or addresses contained in another vector. The latter case is common in sparse matrix computations. We will refer to this pattern as vector-indirect access.

- The application vector for Fast Fourier Transform algorithms corresponds to a bit-reversal of the address of consecutive elements in an array that contains the data.

- Linked data structures.

This thesis introduces a memory controller architecture that understands application vectors that follow a base-stride pattern in addition to the traditional unit stride pattern. The architectural features and algorithms used for this purpose will be collectively referred to as Parallel Vector Access or PVA. This thesis discusses the problems which cause base-stride access to perform poorly on DRAM memory systems and introduces architectural features that enable the memory controller to efficiently load and store base-stride vectors by operating multiple memory banks in parallel. It also provides suggestions on how other common application vectors might be handled.

The domain of applicability extends from traditional scientific vector processing to the realm of desktop computing. Several new instruction set extensions (e.g., Intel's MMX for the Pentium [18], AMD's 3DNow! for the K6-2 [1], MIPS's MDMX [29], Sun's VIS for the UltraSPARC [41], and Motorola's AltiVec for the PowerPC [32]) bring stream and vector processing to the domain of desktop computing. The results for some applications that use these vector extensions are quite promising [42, 40], even though the extensions do little to address memory system performance. All these extensions will benefit from a memory controller that understands application vectors.

The organization of the main memory system assumed in the rest of this thesis is shown in Figure 1.1. The data paths are not shown in the figure. This thesis assumes that the processor has some means of communicating information about application vectors to the memory controller. Some indications of how this can be achieved can be found

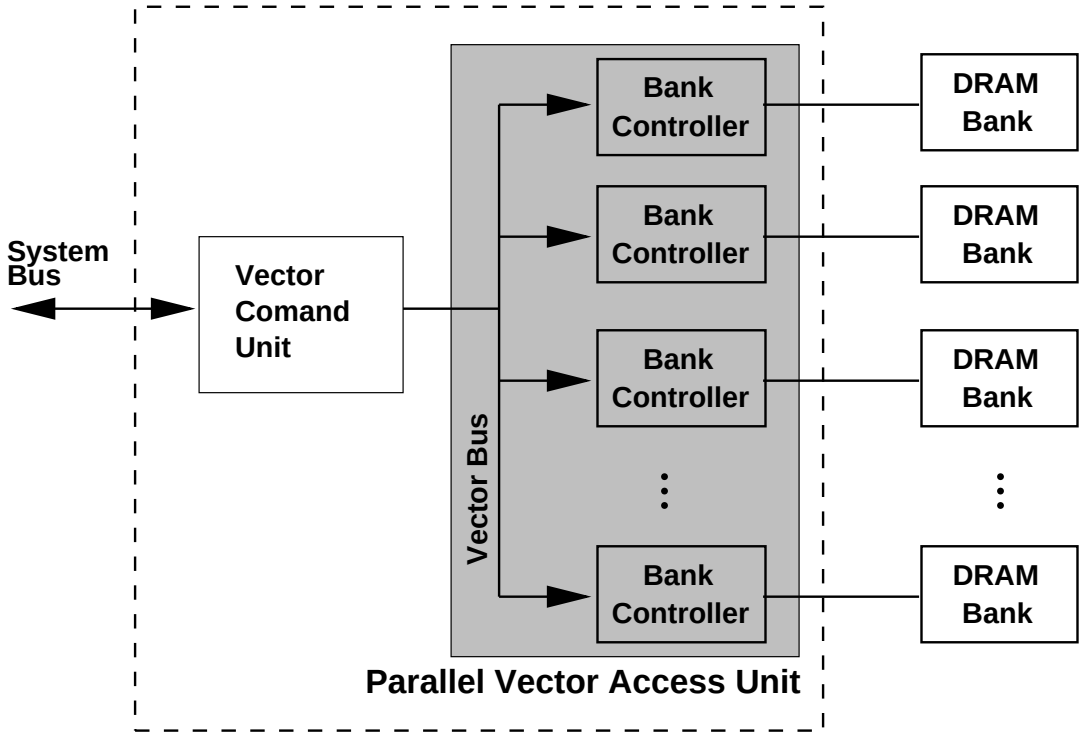


Figure 1.1. Memory System Organization

in Section 3.2. The rest of this thesis assumes that the PVA unit receives vector requests from the Vector Command Unit and returns results to it. The communication between the Vector Command Unit and the processor over the system bus are not relevant to the ideas discussed here.

To put the discussion of the PVA in the appropriate context, Chapter 2 provides a brief background of current memory technologies. Chapter 3 discusses related work in this area. Chapter 4 introduces the performance problems faced by a serial vector access method and derives PVA algorithms for parallel base-stride access. It sets the background for Chapter 5, which describes the implementation architecture. Chapter 6 describes the experiments we did to evaluate the PVA's performance and analyzes their results. Chapter 8 concludes the results of this work and presents some directions for future research.

CHAPTER 2

MEMORY TECHNOLOGY BACKGROUND

The memory technology used for high performance vector processors has traditionally been SRAM, and DRAM has been employed on almost all other machines. A significant amount of memory system research, including the work presented in this thesis, attempts to close the gap between DRAM and SRAM performance. To fully understand the PVA design presented later in this thesis, it is necessary to have sufficient background in the technology and tradeoffs of SRAM and DRAM variants.

A major portion (often more than half) of the cost of a vector supercomputer consists of the cost of the memory system [20][14]. Architects of DRAM based computer systems have long been exploring ways to make the performance of DRAM match that of SRAM without paying the vastly greater cost. The importance of this effort becomes obvious when we consider that the largest DRAM chip available from a major manufacturer in 1999 was 256 Mbits while the largest SRAM part available from the same manufacturer was 4 Mbits [30][31]. In addition, the 4-Mbit SRAM chip is priced much higher than an SDRAM chip with 64 times the capacity. The 4-Mbit SRAM part has a cycle time of 10ns (max). The 256-Mbit SDRAM part too is capable of operating with a clock cycle time of 10ns. What sets the SRAM's performance apart from that of the SDRAM is that while the SRAM always has a fixed latency of 10ns, the SDRAM might take several clock cycles to access data. Theoretically, it is possible to apply one address to an SDRAM every cycle, since it internally pipelines accesses. If this were practically possible, then the 256 M bit SDRAM part might be able to deliver performance close to that of the 4-Mbit SRAM part at a fraction of the cost. The current trends in DRAM technology can all be considered as interface modifications that are geared towards exploiting this ability to pipeline accesses to the maximum.

The RMC2 memory controller and the Alpha 21174 memory controller described in Section 3.1 are examples of advanced memory controllers that exploit this trend to reduce memory system latency. Both try to hide row open and precharge latencies of DRAM as much as possible by exploiting row hits within a stream of accesses. The PVA unit described in this thesis also uses sophisticated techniques to achieve the same objective. To understand how these advanced memory controllers work it is necessary to understand how modern DRAM and SRAM chips work. This chapter provides a brief introduction to current memory technologies.

2.1 Static RAM (SRAM)

In SRAM every bit corresponds to a six transistor cell. For optimal layout the cells are often organized as a square matrix. Figure 2.1 shows the internal organization of a typical SRAM chip. The address bits $A_m ... A_o$, consist of the row address (bits $A_o ... A_n$) and the column address (bits $A_{n+1} ... A_m$) . The row decoder uses the row address to select an entire row within the memory array and portion of this row is selected by the column address decoder using the column address. Control signals like Chip Select,

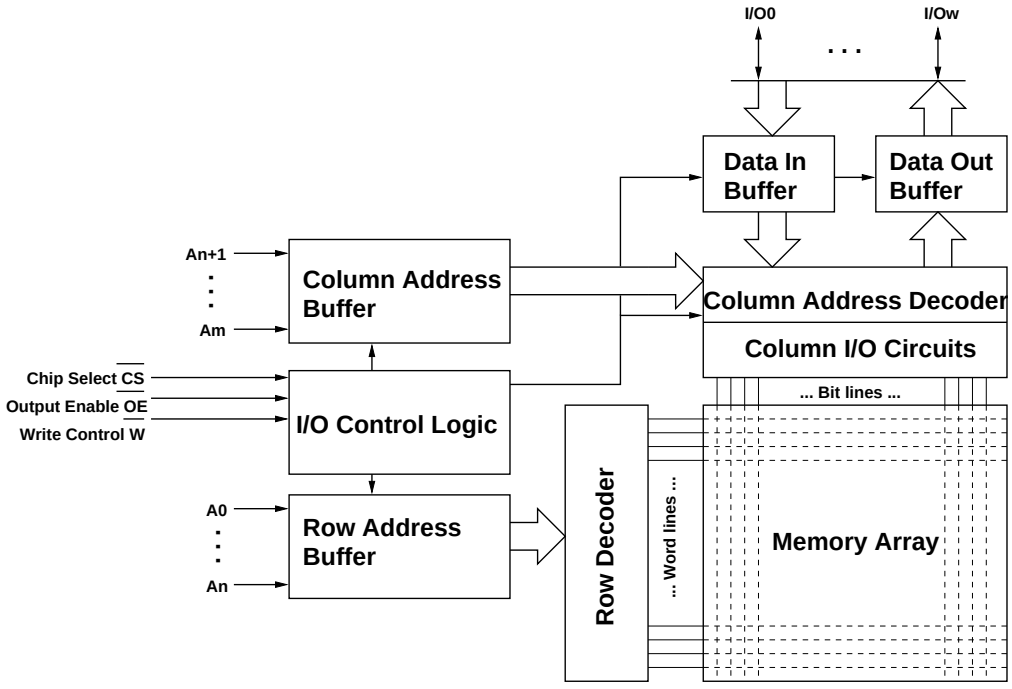


Figure 2.1. Block Diagram of SRAM

Output Enable and Write Control specify the operation to be done.

2.2 Dynamic RAM (DRAM)

In DRAM, every bit corresponds to a single transistor cell, which is implemented as a simple capacitive charge well. DRAM cells are more compact than SRAM cells, yielding much greater densities for DRAM over SRAM. Figure 2.2 shows the internal organization of a typical DRAM chip. Like SRAM, the cells are organized as a matrix. But unlike SRAM, the address bits are multiplexed. The row and column decoders work like their counterparts in SRAM. The fundamental difference is that an analog sense amplifier is used to read the contents of a cell and the charge in the cell is drained after each read. The charge leaks over time and needs to be restored periodically and also after each read. For these reasons access to a DRAM word involves a more complex procedure than access to an SRAM word. First the row address is applied (RAS — Row Address Strobe) and the row decoder selects the appropriate row and then the column address is applied (CAS — Column Address Strobe). In the case of reads, after the data

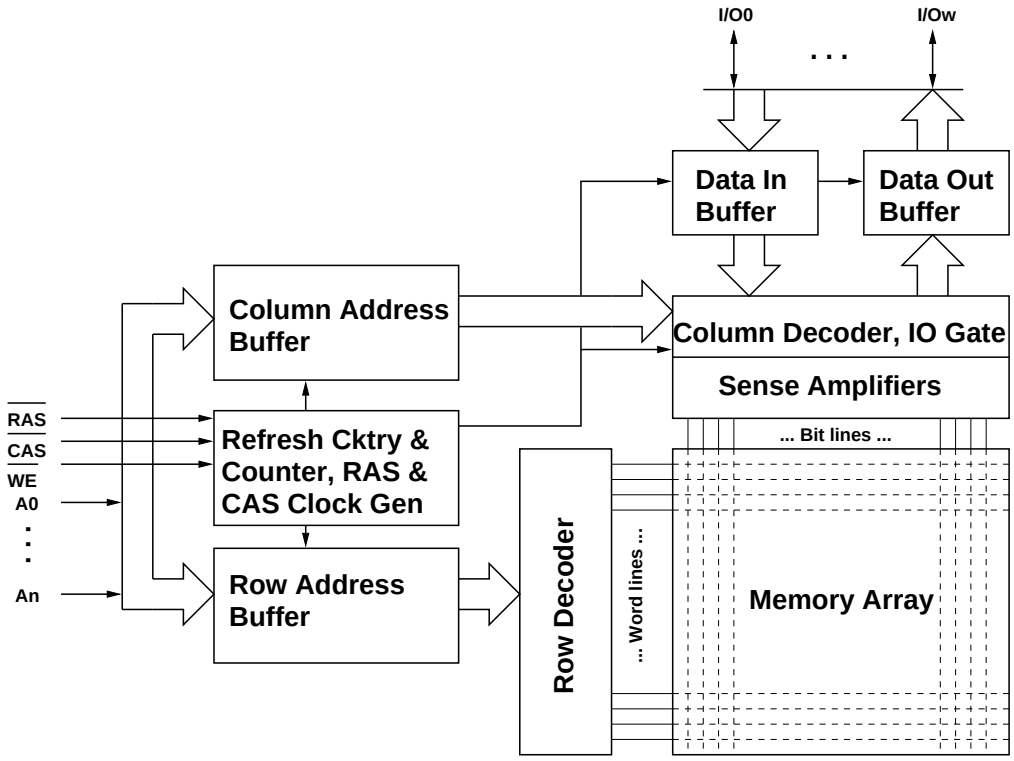


Figure 2.2. Conventional DRAM

has been read, a precharge operation is done to restore the charge to the row that was read. Refresh circuitry is used to periodically (typically every 64ms) refresh the contents of each row.

2.3 New DRAM Variants

Most manufacturers currently consider traditional DRAM and its simpler variants like Fast Page Mode DRAM and EDO DRAM as end of life products. This section therefore emphasizes more sophisticated DRAM technologies like SDRAM, SLDRAM and RAMBUS.

2.3.1 Fast Page Mode DRAM (FPM DRAM)

Fast page mode DRAM is a minor improvement over the conventional DRAM described in Section 2.2, in that it allows multiple CAS cycles following a RAS cycle. This is good for accessing sequential data within a row. The sense amplifiers hold the data corresponding to the currently open page (row) within the memory array, and such data can be accessed quickly. When all accesses to the row are completed, a final precharge operation is issued to close the row.

2.3.2 Extended Data Out DRAM (EDO DRAM)

EDO DRAM has an additional latch that stores the data while the row is being precharged, permitting overlap of reading the data off the bus and precharging the row. It also permits the data to remain valid longer.

2.3.3 Synchronous DRAM (SDRAM)

Though SDRAM uses a core similar to a traditional DRAM core, it is fundamentally different in that it synchronizes its operation to a system clock. While RAS and CAS are asynchronous signals in the case of conventional DRAM, it is more appropriate to consider these as commands issued to an SDRAM chip at the edge of the clock. SDRAM internally pipelines its operation. Though SDRAM has several timing constraints (e.g., a RAS following a precharge must be issued only after a specified precharge delay), because of the pipelined operation a CAS can be issued each cycle. SDRAMS are internally organized as several banks (typically four) and operations to different banks

can be overlapped. A smart memory controller can use these features to ensure better performance by issuing optimal sequences of operations and by managing open rows more effectively. Since the PVA implementation described in Chapter 5 is based on SDRAM, the remainder of this section provides more detail on the operations required to access data on SDRAM, and the factors that afftect SDRAM memory system performance.

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 SDRAM. Like in the case of conventional DRAM, data in the storage cells of the decoded row are moved into a bank of sense amplifiers (also called a page buffer), which serves as a row cache. During the CAS, the column address is decoded and the selected data is 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 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 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. 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.

Memory system designers typically try to optimize memory system performance by choosing interleaving and row management policies that work well for their intended

workloads. For example, independent SDRAM banks can be interleaved at almost any granularity, from one-word to one-row (typically 512-2048 bytes) wide. The nonuniform nature of SDRAM accesses causes the choice of interleaving scheme to affect the memory performance of programs with some access patterns.

2.3.4 Synchronous Link DRAM (SLDRAM)

SLDRAM is an open-standard, high-performance DRAM technology that follows an evolutionary approach from SDRAM to Dual Data Rate DRAM (DDR), i.e., DRAM that transfers data on both edges of the clock [12]. SLDRAM uses a multidrop bus that connects a memory controller and up to eight SLDRAM devices. The controller sends commands to SLDRAM devices over a portion of the bus called the CommandLink that operates on both edges of the clock. Data is transferred over the 18-bit DataLink portion of the bus and may be synchronized with one of two possible clock signals. Two clocks are used to minimize the gap required when control of the DataLink is transferred from one device to another. Like SDRAM, all internal operations are pipelined.

2.3.5 Direct Rambus DRAM (DRDRAM)

DRDRAM represents a significant advance in DRAM technology in terms of both the semantic level and the electrical characteristics of the DRAM interface [36][35]. Like SDRAM, the interface is synchronous. However, to minimize clock to data skew, DRDRAM uses a source synchronous timing model. DRDRAM sends clock and data in parallel and there are two separate clocks called ClockToMaster and ClockFromMaster. Data sent by a DRDRAM to the controller is synchronous with ClockToMaster, whereas data sent from the memory controller to DRDRAM is synchronous with ClockFromMaster. Data is transferred on both edges of the clock, permitting transfer rates of 600MHz or 800MHz which corresponds to a sustained bandwidth of 1.6GB/s. DRDRAM has a 16-bit data bus and separate row and column control buses. The core is organized as 32 banks and four transactions can take place simultaneously. All operations are internally pipelined. The unit of data transfer is a dual-oct or 16 bytes. Each transfer takes four clock cycles over the 16-bit data bus.

CHAPTER 3

RELATED WORK

Most high performance vector memory system research in the past two decades targets memories composed of SRAM devices, which have a uniform access time and are faster than DRAM parts, but which increase the memory cost beyond what is reasonable for commodity systems [15][3][39]. In many cases it may not be straightforward to extend these techniques to DRAM memory systems because of their non-uniform access times. More importantly, those techniques do not take advantage of the ability of modern parts like SDRAM, Direct Rambus and SyncLink to overlap commands to different internal banks. For instance, techniques like address skewing complicate the address arithmetic for each bank too much to be viable in an access-ordering memory controller for dynamic memory components [5]. In this chapter we limit our evaluation of related work to that which deals with vector accesses, especially those that load vectors from DRAM.

3.1 Access Scheduling and Access Ordering Systems

Moyer defines access scheduling as those techniques that reduce load/store interlock delay by overlapping computation with memory latency [33]. Access scheduling techniques attempt to separate the execution of a load/store instruction from the execution of the instruction that produces/consumes the data, thereby reducing the delays that the processor sees for memory requests. In contrast, Moyer defines access ordering to be any technique that changes the order of memory requests to increase memory system performance. He then presents compiler algorithms that optimize access ordering by unrolling loops and grouping accesses to “streams” so that the cost of each DRAM page miss can be amortized over several references to the same page [33].

The DEC Alpha 21174 memory controller implements a relatively 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) [38]. This memory controller is an ASIC used in the DIGITAL Personal Workstation . It interfaces with a set of SDRAM DIMMs (Dual Inline Memory Module) and the 128 bit system bus on the workstation. What sets it apart from traditional memory controllers is its ability to reduce memory latency by exploiting open rows on the SDRAM devices. The 21174 memory controller manages the open rows on its SDRAM devices as a cache. It uses a 4-bit predictor per DIMM to track whether accesses hit or miss the most recent row in each row buffer. At the end of an access, if a row-hit is predicted the row is left open, and if a row-miss is predicted the row is closed. Associated with each predictor is a 16-bit precharge policy register. This register is set by software to indicate whether the row should be left open or precharged for each possible value of the four bit history. For McCalpin's STREAM benchmark [23], this simple adaptive hot-row management policy yields best-case improvements in memory latency and bandwidth of 23% and 7%, respectively.

The RMC2 memory controller from RAMBUS, Inc., uses timing and logical constraints to skip precharge cycles when possible, to improve channel bandwidth [37]. It has many features similar to the SDRAM interface of the PVA unit described in this thesis, although the PVA work predates the RMC2 controller. The RMC2 controller permits up to seven outstanding transactions, permits both open-page and closed-page policies and automatically keeps a page open at the completion of a transaction if another issued transaction hits on the same page. It is designed to accept and start one transaction each clock cycle. However, its heuristics are not as extensive as those of the PVA and it does not reorder transactions.

Lee mimics Cray instructions on the Intel i860XR using a purely software approach. He treats the cache as a pseudo "vector register" by reading vector elements in blocks (using noncaching load instructions) and then writing them to a preallocated portion of the cache [22]. The benefits of these optimizations can be dramatic: loading a single vector via Moyer's and Lee's schemes on a node of an iPSC/860 yields performance

improvements between about 40% and 450%, depending on the stride of the vector [25]. 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 [43], and then extend the work for multiple vectors [44]. 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 vector computers whose memory systems are composed of SRAM components (with uniform access time).

The system most similar to the PVA design presented in this thesis is the Command Vector Memory System [7] (CVMS). The CVMS exploits parallelism and locality of reference to improve the effective bandwidth of 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, as described in Chapters 4 and 5.

The Command Vector Memory System's hardware scheme for computing vector subcommands is based on earlier access-scheduling work for vector multiprocessors [34]. Although the full details of their subcommand-generation algorithm have not yet been published, the authors state that for strides that are not powers of two, 15 memory cycles are required to generate the subcommands [7]. The scheme that we have implemented and simulated in Verilog is substantially faster, requiring at most five memory cycles to generate subcommands for strides that are not powers of two. Both designs process power-of-two strides in only two cycles. Their system relies on a crossbar interconnect,

and the details of how vector data are merged from the various section controllers have not yet been published. Our design is based on a 128-bit bus that connects the bank controllers to the main memory controller, and vector data is merged on this bus by alternately driving each 64-bit half. Furthermore, the Command Vector Memory System is specifically designed for out-of-order vector machines, where vector data are loaded into vector registers. Our system delivers the vector data in cacheline- sized chunks intended for the on-chip L2 cache, but could easily be adapted to interact with dedicated vector registers.

The Stream Memory Controller (SMC) of McKee et al. [26] combines programmable stream buffers and prefetching in a memory controller with intelligent DRAM scheduling. Vector data bypass the cache in this system, but the underlying access-ordering concepts can be adapted to systems that cache vectors. The SMC dynamically reorders stream/vector accesses and issues them serially to exploit: a) parallelism across dual banks of fast-page mode DRAM, and b) locality of reference within DRAM page buffers. For most alignments and strides on uniprocessor systems, simple ordering schemes perform competitively with sophisticated ones [24].

3.2 Detecting Vectors

Another issue to consider when designing a vector access unit is how to detect the vector accesses (streams). At one end of the design spectrum, the application programmer may be required to identify vectors. Alternatively, the compiler could identify the vector accesses and specify them to the memory controller. One simple and efficient means of recognizing vectors uses Benitez and Davidson's compiler algorithm to detect streams, which is similar in complexity to strength reduction [2]. Vectorizing compilers can also provide the needed vector parameters, and can perform extensive loop restructuring and other optimizations to maximize vector performance [45]. At the other end of the spectrum lie hardware vector or stream detection schemes, which may be implemented via reference prediction tables [6]. The PVA unit described in this thesis was designed in the context of the Impulse memory controller which provides yet other ways of using vectors [4]. Impulse supports multiple views of the same data. A region of memory

may be remapped through a shadow address space, which effects an additional step of address translation. One possible shadow space is a strided view of some other unit stride region of memory. When the processor accesses data in the shadow space, the memory controller does scatter/gather accesses from the real memory region that backs the shadow address region and compacts the strided data into dense cache lines. Shadow spaces may be configured in the memory controller either directly by the programmer or by a smart compiler. Either way, when the PVA unit is used with an advanced memory controller like Impulse, there is an efficient mechanism by which the PVA can be informed about vector accesses and can return dense cachelines to the processor.

3.3 Memory Interleaving Schemes

Numerous studies have explored the use of specialized addressing schemes that tend to avoid memory bank conflicts for commonly observed access patterns on vector machines. XOR-tree based schemes and interleave methods that use $2^k \pm 1$ modules are typical examples [10]. While such schemes are suitable for uniform-access components like SRAM access ordering for nonuniform access memory components like SDRAM require performing address arithmetic which gets complicated when skewing schemes are used. Moreover, Hsu and Smith demonstrate that it is useful to take advantage of spatial locality while using such components [14]. Their study concentrated on interleaving schemes for paged DRAM memory in vector machines and did not cover any access ordering scheme. Their study indicated that cache-line interleaving and block-interleaving are much superior to low-order interleaving for many vector applications. Results from their study showed that cache-line interleaving has performance nearly the same as block-interleaving for a moderate number (16-64) banks, beyond which block-interleaving performed better. It is possible that low-order interleaving may perform better when used along with access ordering and scheduling techniques. Like address skewing techniques, block interleaving has the undesireable property that it complicates address arithmetic.

3.4 Scheduling Theory

Assuming that the memory system has multiple outstanding addresses that need to be accessed, it may be necessary to reorder the sequence of addresses to optimize overall

performance. Whole bodies of literature exist on scheduling tasks in various domains [13]. Some of the work in scheduling theory can serve as starting points for implementing access reordering systems. The optimal scheduling problem has been proven to be NP complete and many of the approaches discussed in this section always generate an optimal solution if one or more optimal solutions exist [11]. In general the algorithms in this area are too complex to be implemented in fast hardware.

3.4.1 Online Algorithms

Online algorithms try to make decisions using incomplete information, often by trying to approximate an optimal off-line algorithm [19]. Good examples are OS page replacement algorithms and what is known as the Ski Rental problem in the literature [19]. There are variants like deterministic online algorithms and randomized online algorithms.

3.4.2 Rate Monotonic Scheduling

Rate Monotonic Scheduling (RMS) is often used to analyze the schedulability of real-time tasks [46][21]. Each task is characterized by its processing time P_i and repeat interval T_i. The release time of a task is the time at which it is given to the scheduling algorithm. The task has an implicit deadline equal to the release time plus the repeat interval, since RMS does not permit two instances of the same task to be active at the same time. RMS theory uses the resource (often processor) utilization factors of the tasks to assure schedulability. For example, RMS theory can guarantee that if the total processor utilization of a set of tasks is 69% or less, then they can be scheduled. Though it is very useful for real-time OS schedulers, the dependency on the repeat interval (which is not known in the case of memory access streams) makes RMS unsuitable for use in memory access ordering hardware.

3.4.3 Non-preemptive Earliest Deadline First Scheduling

Unlike Rate Monotonic Scheduling the Non-preemptive Earliest Deadline First Scheduling (EDF) algorithm is capable of scheduling tasks whose deadline is not the same as the sum of the release time and repeat interval. It works as follows:

Given n tasks $T_1, T_2, ...T_n$ arranged in order of their deadlines $D_1, D_2, ...D_n$ and having

execution times of $E_1, E_2, ...E_n$ respectively:

1. Schedule T_n in the interval $[D_n - E_n, D_n]$

2. While more tasks remain to be scheduled do
   Schedule task with latest deadline as late as possible

3. Move tasks forward as much as possible in time maintaining their order.

The preemptive version of this algorithm is provably optimal [21]. The non-preemptive version is more amenable to hardware implementation.

3.5 Operations Research and the Binate Covering Problem

The Transportation Problem from Operations Research (OR) and the Binate Covering Problem (BCP) often discussed in logic minimization literature are very similar in nature. They are both used for solving optimization problems that involve complex sets of choices. We can consider memory access scheduling under conditions when the memory controller is heavily loaded as a problem of choice. Under such conditions, each request has two different penalties associated with it — the number of cycles required to complete the access before the processor stalls on it, and the cycles required to complete the access assuming that a processor stall cannot be avoided. Both the OR approach and BCP can be applied to generate an optimal memory schedule that minimizes total execution time. Both algorithms generate a provably optimal solution, when one or more optimal solutions exist. The approach followed in both these algorithms is as follows.

Given a set of n possible partial solutions $\{S_{0,}S_1, S_2, ...S_{n-1}\}$ each of which satisfy some subset of a set of constraints, to find an optimal solution that satisfies all the constraints:

1. Assume S_0 is included in the final solution.

2. Recursively solve the partial problem using the partial solution set $\{S_1, S_2, ...S_{n-1}\}$ and the set of constraints not already satisfied by S_0.

3. Assume S_0 is excluded from the final solution.

4. Recursively solve the partial problem using the partial solution set $\{S_1, S_2, ...S_{n-1}\}$ and the set of original constraints.

5. Choose the solution with the best cost.

Heuristic techniques or dynamic programming may be used to optimize these algorithms. But they usually involve large matrix manipulations and large amount of integer arithmetic which are unsuitable for fast hardware implementation.

CHAPTER 4

PARALLEL VECTOR ACCESS ALGORITHMS

As explained in Chapter 1, base-stride is a common and important type of application vector. This chapter discusses the problems associated with serial vector access and explains algorithms that a multibank memory system can use for parallelizing the base-stride type of memory accesses.

4.1 The Performance Problem with Serial Vector Access

Processing a base-stride application vector involves gathering strided words from memory into a dense cache line for a read operation, and scattering the contents of a dense cache line to strided words in memory for a write operation. The non-uniform delays of SDRAM explained in Section 2.3.3 make such scatter/gather operations significantly more expensive than they are for SRAM. Consider the 8-bank, cacheline-interleaved SDRAM memory system illustrated in Figures 4.1 and 4.2.

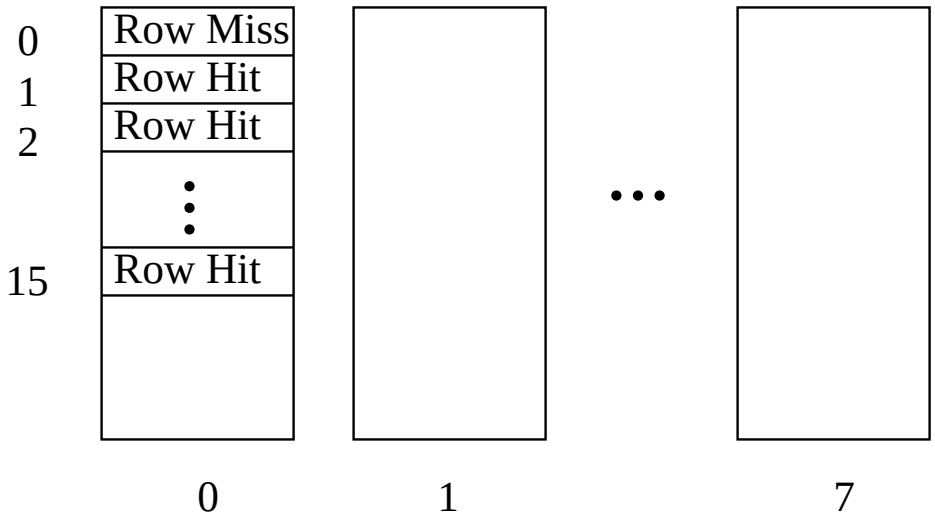


Figure 4.1. Cache-line Fill Timing

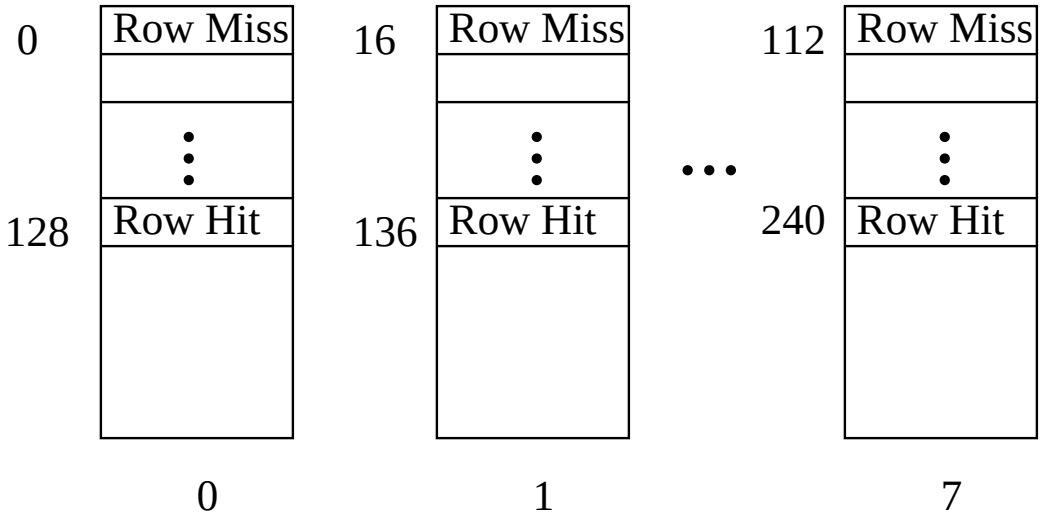


Figure 4.2. Strided Access Timing

Figure 4.1 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 (2 cycles). Then the column address is sent to the SDRAM, after which a delay equivalent to the CAS latency must be observed (2 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 4.2 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. 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 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, repeating 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 chapter describes such methods.

4.2 Parallel Access to Base-Stride Vectors

The PVA unit shown in Figure 1.1 parallelizes base-stride accesses by broadcasting a vector command to a collection of bank controllers (BCs), each of which determines independently, and in tandem with the other BCs, which elements of the vector (if any) reside in the DRAM it manages. This broadcast approach to gather sparse data is potentially much more efficient than the straightforward alternative of having a centralized vector controller issue the stream of addresses, one per cycle, that correspond to the vector elements. However, to realize this performance potential we need a method by which each bank controller can determine the addresses of the elements that reside on its DRAM without sequentially expanding the entire vector. The primary advantage of the PVA over similar designs is the efficiency of our hardware algorithms for computing the sub-vector of each bank.

4.2.1 Terminology

Three functions implement the core of our scheduling scheme. They are *Decode-Bank(), 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 $V =< B, S, L >$, 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 $V =< A, 4, 3 >$

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 δ 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, $V.B$. If $DecodeBank(V.B) \geq b$, $d = DecodeBank(V.B) - b$. Otherwise, $d = DecodeBank(V.B) + M - b$.
- Let $\triangle b$ represent the number of banks skipped between any two consecutive elements $V[i]$ and $V[i + 1]$. $\triangle b = (V.S \bmod NM)/N$.
- Let θ be the block offset of V's first element, thus $\theta = V.B \bmod N$.
- Let $\triangle\theta$ be the difference in offset within their respective blocks between any two consecutive elements $V[i]$ and $V[i + 1]$, i.e., $\triangle\theta = (V.S \bmod NM) \bmod N$. 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.

4.2.2 Basic PVA Design

This section describes a PVA Bank Controller design and a recursive $FirstHit()$ algorithm for block interleaving schemes.

The basic PVA design works as follows. To perform a vector gather, the VCU issues

a vector command of the form $V =< B, S, L >$ 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.

4.2.2.1 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** $\triangle\theta = 0$

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

Case 1.1: $V.L > d$ **and** $\triangle b$ **divides** d

In this case, $FirstHit(V, b) = d/\triangle b$.

Case 1.2: $V.L \leq d$ **or** $\triangle b$ **does not divide** d

For all other $FirstHit(V, b) = no\, hit$.

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

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

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

In this case the $\triangle\theta s$ 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 $\triangle\theta$ on $FirstHit(V, b)$ and use the same procedure as in Case 1.

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

Whenever the running sum of the $\triangle\theta$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 $\theta = 0, \; \triangle\theta = 0, \; and \triangle b = 2$.

   This vector hits the repeating sequence of banks $0, 2, 4, 6, 0, 2, 4, 6, \ldots$

2. Let $B = 0$, $S = 9$, and $L = 4$.

   This is Case 2.1, with $\theta = 0, \; \triangle\theta = 1, \; and \triangle b = 2$.

   This vector hits banks $0, 2, 4, 6$.

3. Let $B = 0$, $S = 9$, and $L = 10$.

   This is Case 2.2, with $\theta = 0, \; \triangle\theta = 1, \; and \triangle b = 2$.

   Note that when the cumulative effect of $\triangle\theta$ 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.

4.2.2.2 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. Figures 4.3 and 4.4 provide a graphical analogy to lend intuition to the derivation.

Figure 4.3 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. Note that the banks repeat after NM words. Figure 4.4 shows a section of the address line along with a vector V with base address B and stride S.

Let $S_0 = V.S \; mod \; NM$. Bank access pattern 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 θ 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 which ever bank the address B decodes to. Also, note that b_1 is not necessarily the bank

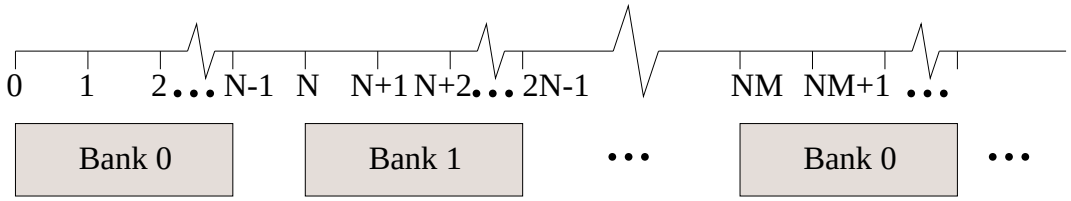


Figure 4.3. Address Line

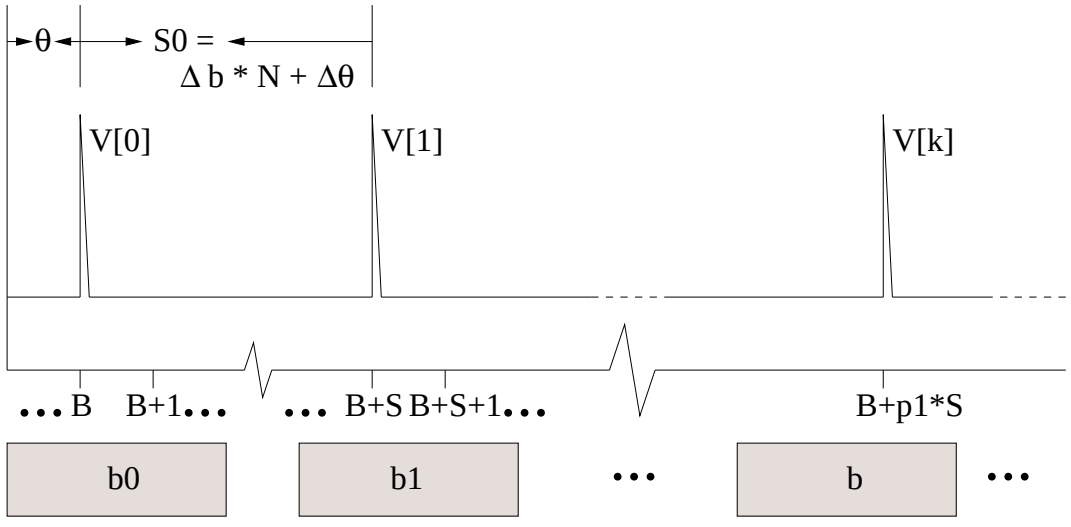


Figure 4.4. Impulse Train

succeeding bank b_0, but is which ever bank the address $B + S$ happens to decode to.

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. Because of the difference in the periodicity of the banks and the impulse train, the impulse train might go past bank b several times before it actually enters a block of words belonging to bank b. Note that 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. However, when $S > N$, the problem is more difficult. The general approach our algorithm takes is to split the original problem into similar subproblems with smaller and smaller values for S_0,S_1,... S_n, until we reach a value $S_n < N$. At this point, we solve the inequation trivially, and we propagate the values up to the larger subproblems. 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_1th 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 $\theta + p_1 S_0 - p_2 NM - dN$. For a hit to happen, this distance should be less than N. Expressing this as an inequality, we get:

$$0 \leq \theta + p_1 S_0 - p_2 NM - dN < N \tag{0}$$

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

$$0 \leq \gamma + p_1 S_0 - p_2 NM < N \tag{1}$$
$$-\gamma \leq p_1 S_0 - p_2 NM < N - \gamma$$
$$\gamma \geq p_2 NM - p_1 S_0 > \gamma - N$$
$$S_0 + \gamma \geq p_2 NM - (p_1 - 1) S_0 > S_0 + \gamma - N \tag{2}$$

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

$$S_0 > S_0 + \gamma - N \tag{3}$$

we can substitute (2) with:

$$S_0 + \gamma \geq p_2 NM \bmod S_0 > S_0 + \gamma - N \tag{4}$$

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

$$\theta < N \tag{5}$$

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

$N > \gamma$

$N > \theta - dN$

$(d+1)N > \theta$

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 $p_1 = p_2 NM/S_0$ or $p_1 = 1 + p_2 NM/S_0$, and one of these two values will satisfy (2).

Recall that $p_2 NM \bmod S_0 = p_2(NM \bmod S_0) \bmod S_0$. Substituting $S_1 = NM \bmod S_0$, 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:

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

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_is 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_is 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 $NextHit()$.

4.2.2.3 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.

4.2.3 Improved PVA Design

We can convert all possible cases of $FirstHit()$ to either of the two simple cases from Section 4.2.2.1 by changing the way we view memory. Consider the memory system shown in Figure 4.5, which consists of two banks of RAM each of which is eight words wide (i.e., $M = 2$ and $N = 8$). Recall that it is difficult for each bank to quickly calculate

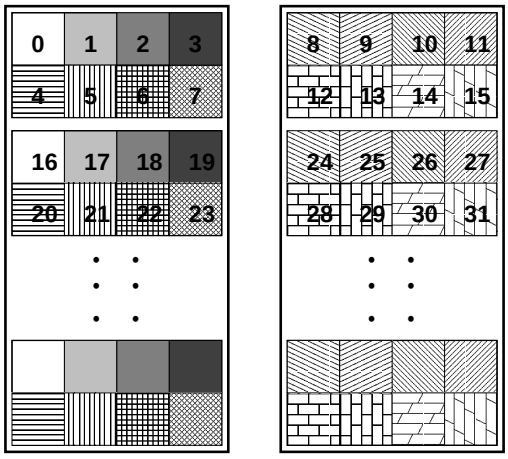


Figure 4.5. Physical View of Memory

$FirstHit()$ when the stride of a given vector is not an integer multiple of N (and thus $\triangle\theta$ is non-zero).

If, instead, we treat this memory system as if it had 16 one-word wide banks (i.e., $M = 16$ and $N = 1$), then $\triangle\theta$ will always be zero. To see this, recall that $\triangle\theta$ 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 4.2.2.1. Figure 4.6 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 4.2.2.1 (since $\triangle\theta = 0$ when $N = 1$). This logical transformation requires us to build NM copies of the $FirstHit()$ logic, whereas we required only M copies in

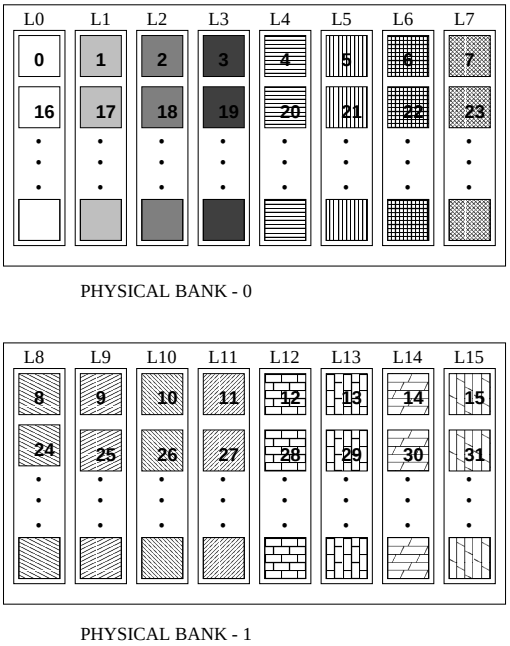


Figure 4.6. Logical View of Memory

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 4.2.1 *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 $b_o = DecodeBank(V.B)$. 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 :

$n * S = q_d * M + d$

$n * (q_s M + S_m) = q_d * M + d$

$n * S_m = (q_d - n * q_s)M + d$

Therefore $(n * S_m) \bmod M = d$.

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 $V.S_1 = (V.S \bmod M)$, will also hit b for $V_1[n]$.

Lemma 4.2.1 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 σ is odd. Using this notation,

s is the number of least significant zeroes in S's binary representation [7].

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 4.2.2 *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, $(n * S)\, mod\, M = d$. Therefore, for some integer q:

$n * S = q * M + d$

$n * \sigma * 2^s = q * M + d = q * 2^m + d$

$d = n * \sigma * 2^s - q * 2^m = 2^s(n * \sigma - q2^{m-s})$

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

Lemma 4.2.2 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^{th} 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 4.2.1 and 4.2.2 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:

$(K_i * S)\, mod\, M = d$

$K_i * \sigma * 2^s = (q_i * 2^m + i * 2^s)$ 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 σ evenly divides $q_1 * 2^{m-s} + 1$.

Theorem 4.2.3 $FirstHit(V, b) = K_i = (K_1 * i) \, mod \, 2^{m-s}$.

Proof: By induction.

Basis: $K_1 = K_1 \, mod \, 2^{m-s}$. This is equivalent to proving that $K_1 < 2^{m-s}$. By lemma 4.2.2, 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 $K_r = \frac{(q_r * 2^{m-s} + r)}{\sigma} = (K_1 * r) \, mod \, 2^{m-s}$, where q_r is the least integer such that σ 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 σ 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 $K_{r+1} = (K_1 * (r + 1)) \, mod \, 2^{m-s}$.

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

For the above induction to work, we must prove that the least integer q_{r+1}, such that σ 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 σ evenly divides $x = q_l * 2^{m-s} + (r+1)$. Since σ 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 σ 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 4.2.4 $NextHit(S) = \delta = 2^{m-s}$.

Proof: Let the bank at distance $d = i * 2^s$ have a hit. Then $K_i * S = q_i * M + d$, where q_i is the least integer that satisfies this equation. Since there is a hit at vector index $K_i + \delta$ on the same bank, we have $(K_i + \delta) * S = q_j * M + d$, where q_j is the least integer that satisfies this equation. Subtracting the two equations, we get:
$(K_i + \delta) * S - K_i * S = \delta * S = \delta * \sigma * 2^s = (q_j - q_i) * M = (q_j - q_i) * 2^m$.
$\delta = \frac{(q_j - q_i) * 2^{m-s}}{\sigma}$.

Recall that σ is an odd number. For σ to divide a multiple of a power of two, it must be that $(q_j - q_i) = n * \sigma$. Since q_j and q_i are the least integers that satisfy their respective equations, it must be the case that $n = q_j - q_i = 1$. Therefore, $\delta = 2^{m-s}$.

4.3 Implementation Strategies for FirstHit() and NextHit()

Using Theorems 4.2.3 and 4.2.4 and given b, M, $V.S$ mod M, and $V.B \bmod M$ as inputs, each Bank Controller can independently determine the subvector 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., δ). 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 $d = (b - b_0) \, mod \, M$ 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 $(K_1 * i) \, mod \, 2^{m-s}$. In the latter case, this only involves selecting the least significant $m-s$ bits of the product, $K_1 * (d \gg s)$). 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.

4.4 Some Practical Issues

An intelligent memory controller that is aware of application vectors needs to be aware of the current application's page mappings in the presence of virtual memory. Scaling the memory system is more complicated than for a traditional memory system. These issues are discussed in this section.

4.4.1 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.

4.4.2 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 add a small amount of logic to then calculate K_i. The complexity of the PLA in this design increases approximately linearly with the number of banks, and the rest of the hardware remains unchanged.

4.4.3 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. Performance will be optimal if each frequently accessed, large data structure fits entirely in one super-page. In that 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 super-page, then the memory controller can split a single vector operation into multiple vector operations such that each subvector is contained in a single super-page. 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 lower bound by rounding the stride size up to the next power of two and using this new stride to compute a minimum number of elements that reside in the region between the first element in the vector and the end of the super-page. Effectively, this turns an integer division into a shift.

CHAPTER 5

IMPLEMENTATION

The design space for a PVA unit is enormous: the type of DRAM, the number of banks, the interleave factor, and the implementation strategy for FirstHit() can all be varied to trade hardware complexity for performance. For instance, lower-cost solutions might let a set of banks share bank controllers and BC buses, multiplexing the use of these resources. To demonstrate the feasibility of our approach and to derive timing and hardware complexity estimates, we have developed and synthesized a Verilog model of a prototype design representing one point in this large design space. The number of banks was chosen as 16 since this is the maximum number of banks that would be reasonable for a uniprocessor system of reasonable complexity. Word interleaving was chosen to simplify the design and also because of the intuition (substantiated by later experiments not reported in this thesis), that word interleaving would improve the performance of irregular applications. The rest of the design parameters were based on the target processor.

We produced an initial FPGA implementation of our PVA design on an IKOS Hermes emulator with 64 Xi-4000 FPGAs, and then used this implementation to derive timing estimates [17]. The timing estimates thus obtained were used to further modify and optimize the design. Only software simulation of the Verilog model was done for the final design. The software simulation used the latencies derived from the synthesized version tested on the hardware. Software simulation was done instead of full design emulation because of the inordinate amount of time and effort required to push the design through the whole tool-path before it can be mapped on to the hardware emulator, and also because some of the tools turned out to have bugs.

5.1 Parameters of the Prototype Implementation

The prototype implementation of the PVA is designed to be incorporated into an adaptable memory controller for the MIPS R10000 processor. Many of the parameters of the system are thus dictated by those of the target processor. Our implementation has 16 banks of word-interleaved SDRAM (32-bit wide) with a dedicated bank controller for each bank. We drive Micron 256 Mbit 16-bit wide SDRAM parts, each of which has four internal banks, and thus four independent row or page buffers [28]. Our PVA unit design assumes an L2 cache line of 128 bytes, and therefore operates on vector commands of 32 single-word elements. Figure 5.1 shows the implementation architecture of our PVA unit. We first describe the overall working of the system, the implementation of the Vector Bus, and the BCs, and then show how the controllers work in tandem.

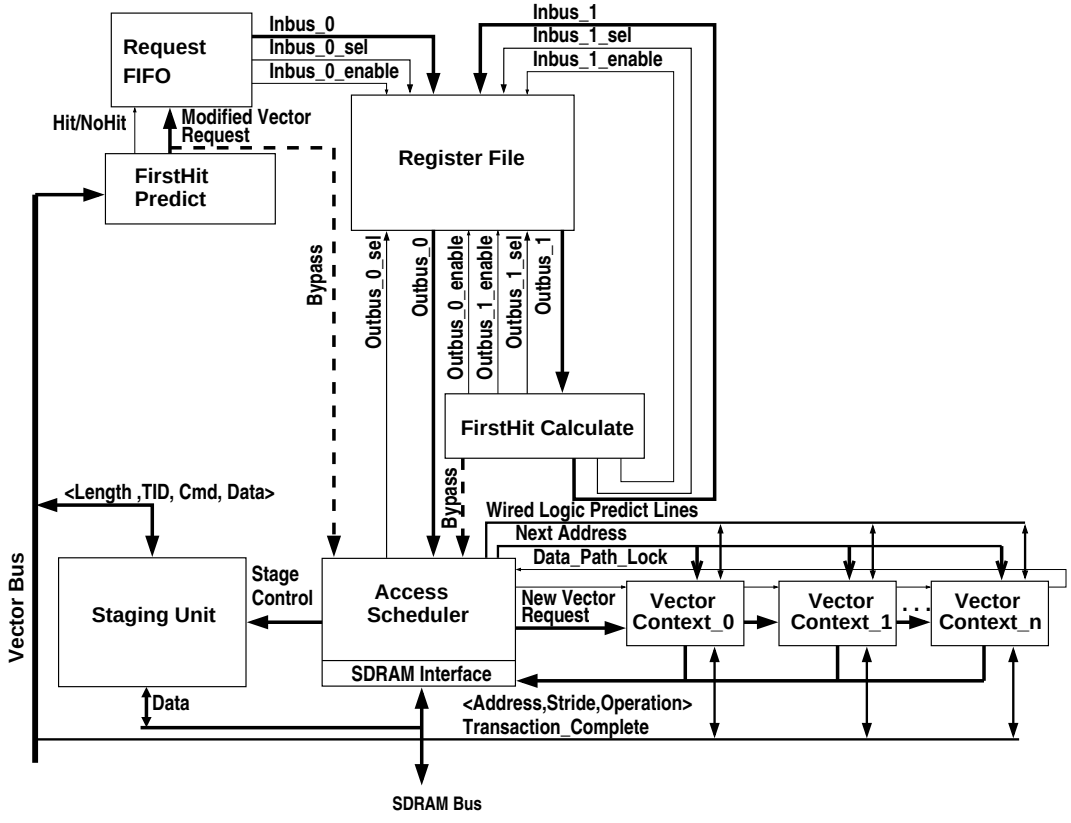


Figure 5.1. Bank Controller Internal Organization

5.2 Implementation Architecture

The Vector Command Unit (VCU — shown earlier in Figure 1.1) , broadcasts a vector command on the Vector Bus. When each Bank Controller receives this command, its Firsthit Predict Unit quickly determines if the given vector hits that bank or not. If a hit occurs, the command is queued in the Request Fifo. The FirstHit Calculate unit does any additional computation that might be required for this particular vector and the vector command is issued to the Access Scheduler. The Access Scheduler stores the command in one of its Vector Contexts and issues memory requests to the SDRAM interface. When the vector command is completed, the staging units help to send out any data they have accumulated for a vector read. The following sections provide more detail on each of these components.

5.2.1 Vector Bus

As illustrated in Figure 1.1, the bank controllers communicate with the rest of the memory controller via a shared, split-transaction Vector Bus that multiplexes requests and data. During a vector request cycle, it supports a 32-bit address, a 32-bit stride, a 3-bit transaction ID, a 2-bit command, and some control information. During a data cycle, it supports 64 bits of data. The MIPS R10000 processor has a 64-bit system bus, and thus the PVA unit can send or receive a data word directly on this bus every cycle. No intermediate unit is needed to merge data collected by multiple bank controllers: when read data is returned to the processor, the BCs take turns driving their part of the cache line onto the system bus. Electrical limitations require a turn-around cycle whenever bus ownership changes. To avoid these delay cycles, we use a 128-bit BC bus, driving alternate 64-bit halves every other data cycle. In addition to the 128 multiplexed lines, the BC bus includes eight transaction_complete indication lines shared by all BCs.

5.2.2 Bank Controllers

For a given vector read or write command, each Bank Controller is responsible for identifying and accessing the (possibly null) subvector that resides in its bank. The architecture of this component, shown in Figure 5.1, consists of:

1. a FirstHit predictor that determines whether elements of a given vector request hit in this bank. If there is a hit and the stride is a power of two, this subcomponent also performs the FirstHit() address calculation;
2. a Request FIFO that queues vector requests for service;
3. a Register File that provides storage for the vector requests in the Request FIFO;
4. a FirstHit Calculation module that determines the address of the first element that hits this bank when the stride is not a power of two;
5. an Access Scheduler that drives the SDRAM, reordering read, write, bank activate, and precharge operations to maximize performance;
6. a set of Vector Contexts within the Access Scheduler to represent the vector requests currently being serviced;
7. a Scheduling Policy Module within each Vector Context to dictate the scheduling policy; and
8. a Staging Unit that consists of
   (a) a Read Staging Unit to store read-data waiting to be assembled into a cache line, and
   (b) a Write Staging Unit to store write-data waiting to be sent to the SDRAMs.

We briefly describe each of these subcomponents below. Note that we have implemented several bypass paths to reduce communication latency among some parts of the BC; these bypass paths are essential to efficient operation. The details of the bypass paths may be found in Section 5.2.3. The main modules of a BC deal with the computations required to do parallel vector access, scheduling SDRAM accesses efficiently, and staging data.

5.2.2.1 Parallelizing Logic

The parallelizing logic consists of the FirstHit Predict (FHP) module, the Register File (RF), the Request FIFO (RQF) and the FirstHit Calculate (FHC) modules. The

FHP module watches vector requests on the BC bus, determining whether or not any element of a vector request will hit this bank. If a hit is indicated, and the stride is a power of two it calculates the address and index of the first vector element that hits this SDRAM bank. It then signals the RQF to queue the request, the calculated address, and the firsthit index. If the stride is a power of two, the request queued by the RQF has an "address calculation complete" (ACC) flag set to indicate that address calculation has been completed and the FHC module may be bypassed. The RF subcomponent provides intermediate storage for vector requests not yet assigned to vector contexts. It contains as many entries as the number of outstanding transactions permitted by the BC bus, eight in our implementation. The Request FIFO (RQF) module implements the state machine and tail pointer required to maintain the Register File as if it were a queue. Requests written into the Register File whose ACC flag were not set by the FHP require further processing. The FHC module computes the firsthit address for vector requests whose stride is not a power of two. It maintains a pointer (workptr) into the Register File and scans the ACC flag of newly queued requests. For requests whose ACC flag is zero because the stride is not a power of two, the FHC multiplies the firsthit index previously calculated by the FHP by the stride and adds it to the base address to generate the firsthit address, and writes the modified address back into the register file with the ACC flag set. Since this calculation requires a multiply and add, it incurs a two-cycle delay. When the scheduler is busy, this delay is completely hidden, since the FHC module works in parallel with the scheduler. When the Access Scheduler (SCHED) sees the ACC bit set for the entry at the head of the RQF it knows that there is a vector request ready for issue.

5.2.2.2 Access Scheduler

The Access Scheduler (SCHED), along with its subcomponents, the Vector Contexts (VCs) and Scheduling Policy Unit (SPU) modules, is responsible for: (i) expanding the series of addresses corresponding to a vector request, (ii) ordering the stream of read, write, bank activate, and precharge operations so that multiple vector requests can be issued optimally, (iii) making row activate/precharge decisions, and (iv) driving the SDRAM. The SCHED module decides when to keep a row open, while reordering

decisions are made by the SPUs contained within the SCHED's Vector Contexts (we implement four VCs in the current design).

Each Vector Context (VC) can hold a vector request whose accesses are ready to be issued to the SDRAM. It determines the series of addresses required to fetch a particular vector via a series of shifts and adds, as described in Chapter 4, and issues the reads, writes, and precharge operations in cooperation with other VCs. The VCs share a data-path to the Access Scheduler that is used to send it the highest priority pending SDRAM operation required by any of the VCs. The VCs arbitrate for this data-path such that at most one of them can access it in any cycle, where the oldest pending operations have highest priority. Vector operations are injected into VC_0. Whenever a vector operation completes, at most one per cycle, any other pending operations "shift right" into the next higher numbered free VC (if any). To give the oldest pending operations higher priority, we daisy-chain the access scheduler requests from VC_N to VC_0 such that a lower numbered VC can place a request on the shared AC data-path if and only if no higher numbered VC wishes to do so.

The VCs attempt to minimize precharge overhead by giving accesses that hit in an open internal bank priority over requests that need to access a different internal bank on the same SDRAM module, as follows. One bank_hit_predict, bank_more_hit_predict, and bank_close_predict line per internal bank are used to coordinate this operation. The AS broadcasts the address of the current row of each open internal bank to the VCs. When a VC determines that it has a pending request that would hit in an open row, it drives the shared line corresponding to the internal bank of the open row to tell the AS not to close the row – in other words, we implement a wired OR operation. Similarly VCs that have a pending request that misses in the internal bank use the bank_close_predict line to tell the AS to close the row. Scheduling Policy Units (SPUs) within each of the VCs decide together which VC can issue an operation during the current cycle. This decision is based on their collective state as observed on the bank_hit_predict, bank_more_hit_predict, and bank_close_predict lines. Separate SPUs are used to isolate the scheduling heuristics within submodules so we can experiment with various scheduling policies without making significant changes to the rest of the BC.

The goal of our scheduling algorithm is to improve performance by maximizing row hits and hiding latencies by operating other internal banks while a given internal bank is being opened or precharged. A heuristic that achieves this goal is to promote row opens and precharges above read and write operations, as long as they do not conflict with the open rows being used by some other VC. This heuristic has the effect of opening rows as early as possible. When no previous VC can issue a read or write due to a conflict or a need to wait for a bank open/precharge to complete, VCs with lower priority can issue their reads or writes. Also, when an older request completes, this policy ensures that a newer request will be ready even if it uses a different internal bank, allowing multiple vector operations to be done in close succession. Another heuristic that improves performance is to do operations out of order as long as the VC whose read or write is to be issued has correct bus polarity (i.e., data travels in the same direction as the last data transfer on the bus). See Section 5.2.4 to understand why this restriction is required. The scheduling algorithm within each SPU is given below.

```
Schedule()
{
 if the VC is ready
        If bank_actv is asserted
                Do nothing this cycle and propagate the
                datapath lock since some other VC wants
                to do a bank activate/precharge
        else
         if the datapath lock could be acquired
                Issue the read/write operation and update
                the address information in the context
                with the value returned by the next
                address calculation logic in the datapath.
                Propagate 0 to the next VCs datapath lock
                input.
         else
                Do nothing this cycle and propagate the
                datapath lock
 else
 if the VC is blocked
        if bank_hit_predict for the current bank is
           not asserted and datapath lock could be
           acquired
                Issue a precharge/bank activate.
```

```
                Propagate 0 to the next VCs datapath
                lock input.
        else
                Do nothing this cycle and propagate the
                datapath lock
 else // i.e. The VC is empty
  Do nothing this cycle and propagate the datapath lock
}
```

5.2.2.3 Row Management Algorithm

To obtain better performance, the scheduling heuristics must be combined with intelligent management of open rows. If we believe that the next access will be to a different row, then closing the row immediately after it is accessed (by using an auto-precharge along with a read or write) gives the best performance. If the next access is likely to be to the same row, then it is better to leave that row open. The access scheduler decides whether to leave a row open after an access or to close it by examining the state of the bank_hit_predict, bank_morehit_predict, bank_close_predict and bank_actv_predict lines and a one bit (per internal bank) autoprecharge_predictor. The autoprecharge_predictor is set whenever the very first operation of a new vector request is issued. The predictor is set to one if the row that open last within the internal bank matches the row of the address of the first vector element irrespective of whether there is a hit or not. If sufficient information to accurately decide the best row policy is not available when the new vector request completes, the predictors value is used to decide whether the row should be closed or not. The one bit predictor is sufficient to detect most simple loops. The actual algorithm used is:

```
ManageRow()
{
 if none of the VCs have issued any operation
        send a nop to the SDRAM
 else
        Let b be the bank corresponding to the current
        operation.
        if the operation was the very first one for a
           vector context
              autoprecharge_predict[b] =
                  (last row address on bank b == row
                   address of the firsthit address)
```

```
        if the operation is a read or a write
           if the vector request is complete
              if bank_morehit_predict[b] is asserted
                 leave the row open
              else
              if(bank_close_predict[b] or
                 autoprecharge_predict[b])
                     auto precharge the row
              else
                     leave the row open
        else // Vector request not complete
          if the next address hits on the same bank or
             bank_morehit_predict[b] is asserted
                  leave the row open
          else
                  auto precharge the row
}
```

5.2.2.4 Staging Units

The Staging Units (SUs) store the data returned by the SDRAMs for a VC-generated read operation and the data provided by the memory controller for a write. In the case of a gathered vector read operation, the SUs on the participating BCs cooperate to merge vector elements into a cache line that is sent to the memory controller front end, as described in Section 5.2.1. In the case of a scattered vector write operation, the SUs at each participating BC will buffer the write data sent by the front end. Associated with each vector pending operation is a transaction_complete line on the BC bus, driven by the SUs. This line acts as a wired-or that de-asserts whenever all BCs have serviced a particular gathered vector read or scattered vector write operation. In the case of a read, when the line eventually goes low the memory controller issues a STAGE_READ command on the vector bus, indicating which pending vector read operation's data is to be read. In the case of a write, the line going low indicates to the memory controller that the corresponding data has been committed to SDRAM.

5.2.3 Bypass Paths

The description in the previous section is actually a simplified version of that in our implementation. To improve performance, we have implemented several bypass paths

that reduce communication latency among some parts of the BCs. For example, there is a bypass path from the FHP module straight to the input port of the last VC within the access schedulers window, which reduces the latency when the Request FIFO is empty and the stride is a power of two. Similarly there is a bypass path from the output of the firsthit calculate module to the input port of the last VC within the access schedulers window, which helps to reduce the latency by one cycle in cases where the stride is not a power of two but there is only one outstanding request in the bank controller. If this bypass path did not exist, then the FHC module would need to write the value back to the register file before the request becomes visible to the access scheduler. As explained earlier, the bank controller design hides latency when the controllers have multiple outstanding requests. In the case where a single request is issued to an idle bank controller, the bypass paths significantly help in reducing latency.

5.2.4 Data Hazards

Reordering reads and writes may violate consistency semantics. To maintain acceptable consistency semantics and to avoid turnaround cycles, the following restriction is required: a VC may issue a read/write only if the bus has the same polarity and no polarity reversals have occurred in any preceding (older) VC. The gist of this rule is that elements of different vectors may be issued out-of-order as long as they are not separated by a request of the opposite polarity. This policy gives rise to the following consistency semantics:

1. RAW hazards cannot happen.

2. WAW hazards may happen if two vector write requests not separated by a read happen to write different data to the same location.

We assume that the latter event is unlikely to occur in a uniprocessor machine. If the L2 cache has a write-back and write-allocate policy, then any consecutive writes to the same location will be separated by a read. If stricter consistency semantics are required, a compiler can be made to issue a dummy read to separate the two writes.

5.2.5 Timing Considerations

SDRAMs have various timing restrictions on the sequence of operations that can be performed. To maintain these timing restrictions, we use a set of small counters called restimers, each of which enforces one timing parameter by asserting a "resource available" line when the corresponding operation may be performed. The control logic of the VC window works like a scoreboard and ensures that all timing restrictions are met by letting a VC issue an operation only when all the resources it needs including restimers and the data-path can be acquired. Electrical considerations require a single cycle bus turnaround delay whenever the bus polarity is reversed, i.e., when a read is immediately followed by a write or vice versa. Precharge and row open operations are not subject to such restrictions. The SCHED units attempt to minimize turnaround cycles while reordering accesses.

5.2.6 Overall Operation

The overall operation of the PVA unit can be understood from the following example. Assume that a vector read needs to be performed with base address B and stride S and that transaction id t is free. The memory controller first issues a VEC_READ command with address B, stride S and transaction id t. The staging units of the bank controllers assert the transaction complete line for t in response to the read command. Note that the transaction complete lines are active low. The bank controllers notice this command on the bus and the first hit predict modules of each bank controller decides if its bank is going to get a hit or not. If there is a hit, it computes the firsthit index. In the case of a power of two stride the first hit predict modules also compute the firsthit address for their respective banks. The vector request gets queued in the Request FIFO. At this point the first hit calculate module detects a new entry in the Request FIFO and completes the address calculation if S was not a power of two. When the access scheduler detects an entry at the head of the FIFO that has its "address calculation complete" flag set it dequeues the entry from the FIFO and enters it into a vector context. The VC then opens the necessary banks and issues the read operations. Since all the bank controllers are working in parallel the read time is reduced. As the data comes back from each SDRAM, the corresponding staging

unit buffers the data in transaction buffer t. When each staging unit detects that all the data that hits on its bank has been collected, it de-asserts the transaction line for t. When all staging units de-assert the line, the memory controller detects that the transaction has completed and issues a STAGE_READ command for transaction id t. In response to this command the staging units that have the zeroth and first words of the data drive it on the BC bus followed by the units that have the second and third words of data and so on. and In 16 cycles all 128 bytes of the data are returned to the memory controller. To avoid electrical limitations, alternate halves of the bus are used every other cycle as explained in Section 5.2.1. The case for a vector write is similar except that the memory controller issues a STAGE_WRITE command for transaction id t first followed by 16 cycles during which it transmits 64 bits of data in each cycle. In the end it sends a VEC_WRITE command with address B, stride S and transaction id t. It may continue to issue other operations after issuing the VEC_WRITE command. When the data has been committed to SDRAM the transaction line for t will be de-asserted.

5.3 Hardware Complexity

The PVA unit's Verilog description consists of about 3600 lines of code. The results of synthesizing this unoptimized hardware prototype for the IKOS library for Xilinx FPGAs [16] are shown in Table 5.1. We used the synthesized design to measure the delay through the critical path, which is through the multiply-and-add circuit required for calculating $FirstHit()$ for non-power-of-two strides. Our multiply-and-add unit has a delay of 29.5ns. We expect that an optimized CMOS implementation will have a delay less than 20ns, making it possible to complete this operation in two cycles at 100MHz. The other critical paths are fast enough to operate at 100MHz even in our FPGA implementation. The FHP unit has a delay of 8.3ns and SCHED has a delay of 9.3ns. CMOS timing considerations are usually very different from those for FPGAs, and thus the optimization strategies differ significantly. Our FPGA delays represent an upper bound — the custom CMOS implementation will be much faster.

Table 5.1. Synthesis Summary

Type	Count
AND2	1193
D Flip-flop	1039
D Latch	32
INV	1627
MUX2	183
NAND2	5488
NOR2	843
OR2	194
XOR2	500
PULLDOWN	13
TRISTATE BUFFER	1849
On-chip RAM	2K bytes

CHAPTER 6

PERFORMANCE EVALUATION

To evaluate the performance of the hardware prototype of the PVA unit described in Chapter 5, we benchmarked it against three other memory systems using six vector style kernels. Gate level simulation on these vector kernels demonstrates the PVA loads elements up to 32.8 times faster than a conventional SDRAM memory system and 3.3 times faster than a pipelined vector unit, without hurting normal cache-line fill performance.This chapter describes the details of our benchmarks and compares the performance of the different memory systems.

6.1 Memory Systems Evaluated

We used four different memory systems in our performance evaluation. The characteristics of each are described below.

6.1.1 PVA

This is the PVA hardware prototype described in Chapter 5. The DRAM technology used is 256 Mbit, 16-bit wide SDRAM organized as 16 banks, each of which is 32 bits wide. RAS and CAS latencies are both two cycles. L2 cache line size is assumed to be 128 bytes, so the maximum length of a vector is 32 words.

6.1.2 Cache Line Interleaved Serial SDRAM

This memory system is an idealized, 16-module SDRAM system optimized for cache line fills. The memory bus is 64 bits, and L2 cache lines are 128 bytes. The SDRAMs modeled require two cycles for each of RAS and CAS, and are capable of 16-cycle bursts. We optimistically assume that precharge latencies can be overlapped with activity on other SDRAMs (and we ignore the fact that writing lines takes slightly less time than reading), thus each cache line fill takes 20 cycles (two for RAS, two for CAS, and 16

for the data burst). The number of cache lines accessed depends on the length and stride of the vectors; this system makes no attempt to gather sparse data within the memory controller.

6.1.3 Gathering Pipelined Serial SDRAM

This memory system is a 16-module, word-interleaved SDRAM system with a closed-page policy. As before, the memory bus is 64 bits, and vector commands access 32 elements (128 bytes, since the present system uses 4-byte elements). Instead of performing cache line fills, this system accesses each vector element individually. Although accesses are issued serially, we assume that the memory controller can overlap RAS latencies with activity on other banks for all but the first element accessed by each command. We optimistically assume that vector commands never cross DRAM pages, and thus DRAM pages are left open during the processing of each command. This simplification was done to make the analysis easier. The result of this assumption is to make the performance of the gathering serial SDRAM unit look slightly better than it would actually be. Precharge costs are incurred at the beginning of each vector command. This system requires more cycles to access unit-stride vectors than the cache line interleaved system we model, but because it accesses only the desired vector elements, its relative performance increases dramatically as vector stride goes up.

6.1.4 Parallel Vector Access SRAM

The performance of this memory system appears under the labels “min parallel vector access SRAM” and “max parallel vector access SRAM” in later graphs. They respectively model the minimum and maximum performance of an idealized SRAM vector memory system with the same parallel access scheme developed for our SDRAM system. Based on static RAM, this system incurs no precharge or RAS latencies: all memory accesses take a single cycle. Comparing the performance of our PVA SDRAM system to the PVA SRAM one gives us a measure of how well our system hides the extra latencies associated with dynamic RAM.

6.2 Experimental Methodology

Table 6.1 lists the kernels used to generate the results presented here. Copy, saxpy and scale are from the BLAS (Basic Linear Algebra Subprograms) [9], and tridiag is a tridiagonal gaussian elimination fragment, the fifth Livermore Loop [27]. Vaxpy denotes a "vector axpy" operation that occurs in matrix-vector multiplication by diagonals. We choose loop kernels over whole-program benchmarks for this initial study because: (i) our PVA scheduler only speeds up vector accesses, (ii) kernels allow us to examine the performance of our PVA mechanism over a larger experimental design space, and (iii) kernels are small enough to permit the detailed, gate-level simulations required to validate the design and to derive timing estimates. Performance on larger, real-world benchmarks — via functional simulation of the whole memory system or performance analysis of the hardware prototype we are building — will be necessary to demonstrate the final proof of concept for the design presented here. These studies have been left as future work.

Recall that the bus model we target allows only eight outstanding transactions. This limit prevents us from unrolling most of our loops to group multiple commands to a given vector, but we examine performance for this optimization on the two kernels that access only two vectors, copy and scale. In our experiments, we vary both the vector stride and the relative vector alignments (placement of the base addresses within memory banks, within internal banks for a given SDRAM, and within rows or pages for a given internal bank). All application-vectors are 1024 elements (32 cache lines) long, and the strides are equal throughout a given loop. In all, we have evaluated PVA performance for 240 data points (eight access patterns $\times$ six strides $\times$ five relative vector alignments) for each

Table 6.1. Kernels Used to Evaluate our Design

Kernel	Access Pattern
copy	for (i = 0; i < L * S; i += S) y[i] = x[i];
saxpy	for (i = 0; i < L * S; i += S) y[i] += a * x[i];
scale	for (i = 0; i < L * S; i += S) x[i] = a * x[i];
swap	for (i = 0; i < L * S; i += S) {reg = x[i]; x[i] = y[i]; y[i] = reg;}
tridiag	for (i = 0; i < L * S; i += S) x[i] = z[i] * (y[i] - x[i-1]);
vaxpy	for (i = 0; i < L * S; i += S) y[i] += a[i] * x[i];

of four different memory system models.

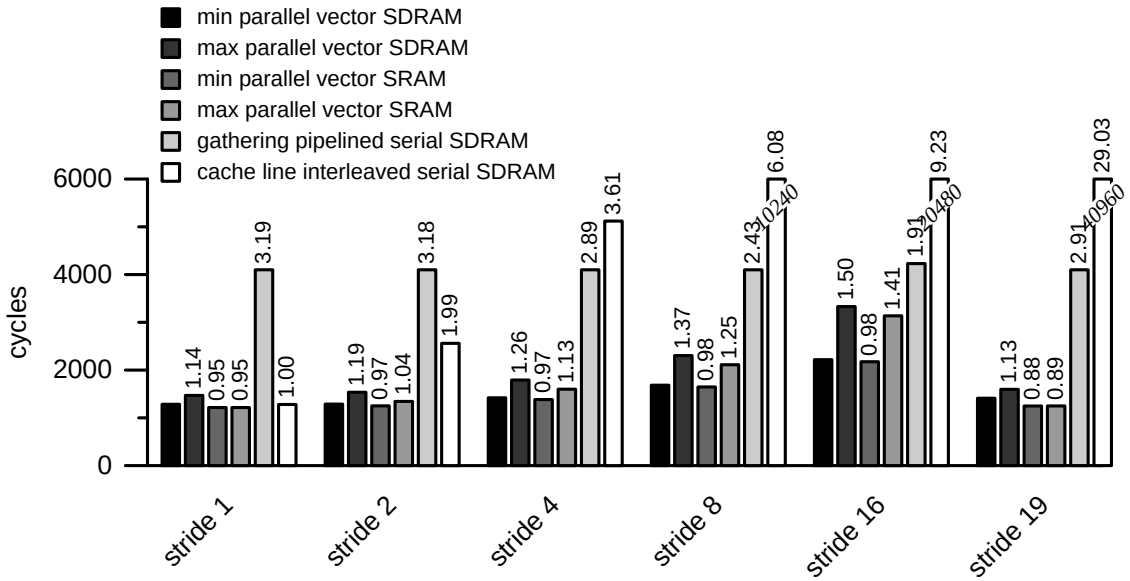
It must be emphasized that the performance evaluation makes the assumption of an infinitely fast CPU that issues memory requests as soon as possible (subject to availability of bus resources). As such, the performance numbers here represent the maximum pressure to which the memory system can be subjected. Speedup experienced by vector applications will be subject to criteria like the percentage of vectoriseable memory accesses, the issue width of the processor and the number of outstanding L2 cache misses permitted. But, in general, it is safe to assume that the faster the processor consumes data, the closer it is to the peak conditions described here. Likewise, the greater the mismatch between the processor and memory speed and data consumption rate and bus bandwidth, the better the performance of a PVA system over a traditional memory system.

6.3 Performance Results

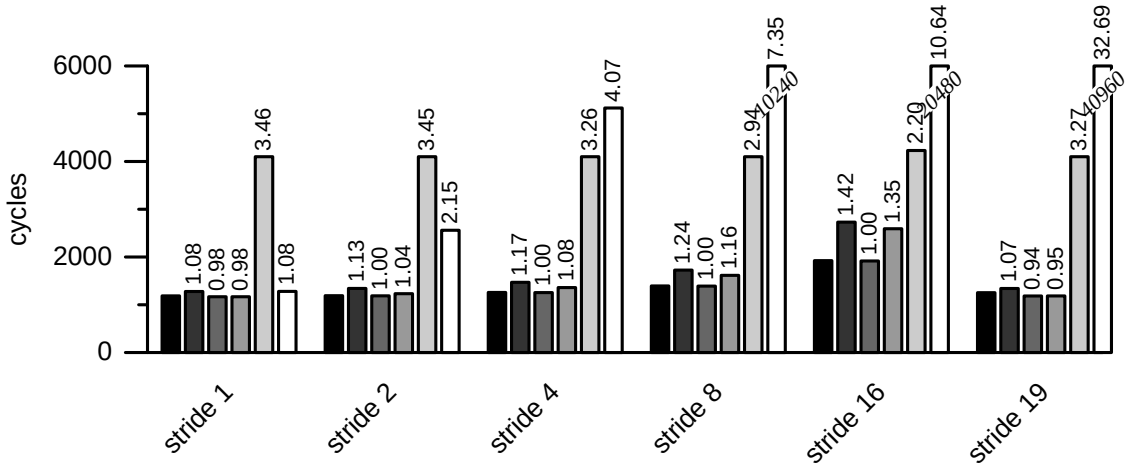
Figures 6.1, 6.2, 6.3, and 6.4 show the comparative performance for our four memory models on strides 1, 2, 4, 8, 16, and 19 for each of the kernels. Figures 6.5, 6.6 and 6.7 show comparative performance across all benchmarks for each of strides 1, 4, 8, 16 and 19. The annotations above each bar indicate execution time normalized to the minimum PVA SDRAM cycle time for each access pattern. Bars that would be off the *y* scale are drawn at the maximum *y* value and annotated with the actual number of cycles spent. For cases where the minimum equals the maximum execution time for the PVA SRAM model, we include only the former bar. The sets of bars labeled “copy2” and “scale2” represent unrolled versions of those kernels for which read and write vector commands are grouped (so the PVA unit sees two consecutive vector commands for the first vector, then two for the second, and so on). This optimization only improves performance for the PVA SDRAM systems, yielding only a slight advantage over the unoptimized versions of the same benchmark. If more outstanding transactions were allowed on the processor bus, greater unrolling would deliver larger performance improvements.

6.3.1 Explanation of Performance Trends

The performance improvement offered by the PVA has four sources:

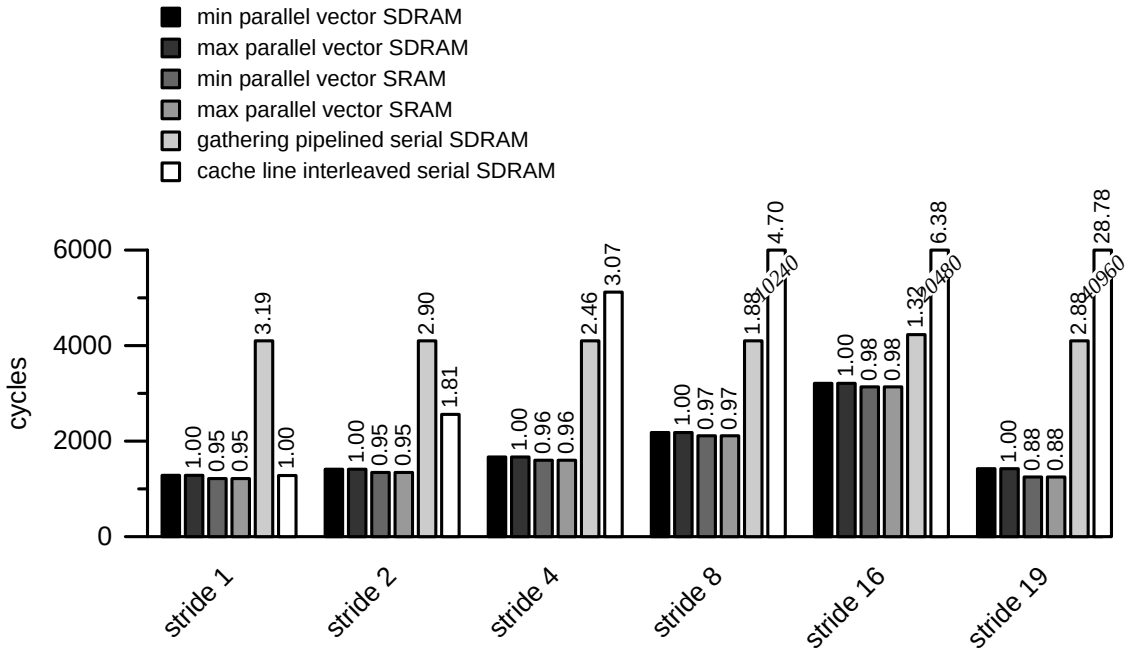


(a) Copy

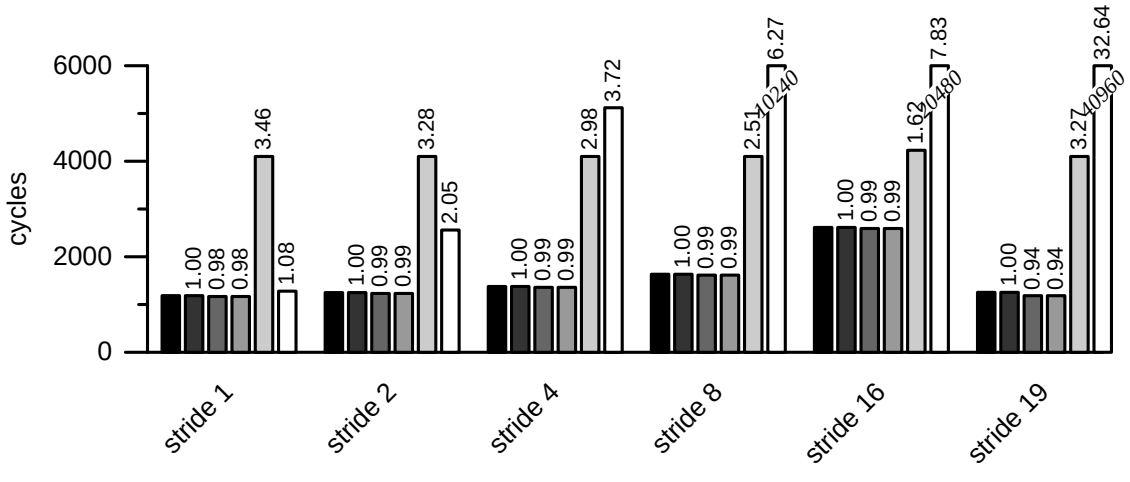


(b) Copy2

Figure 6.1. Comparative Performance with Varying Stride for the Kernels Copy and Copy2

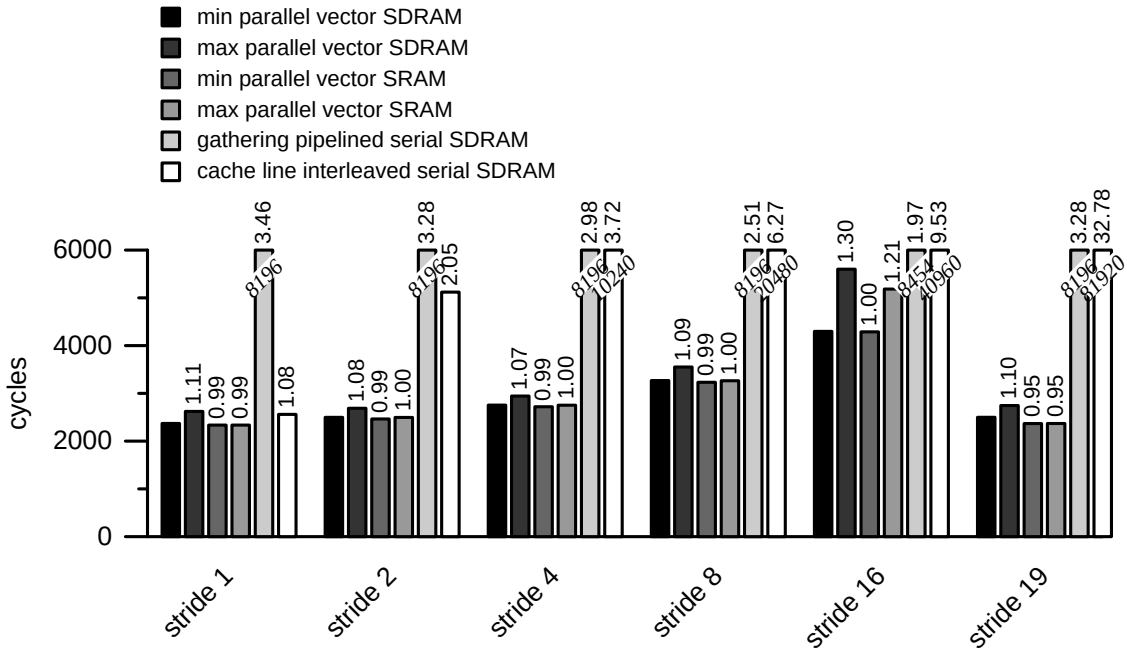


(a) Scale

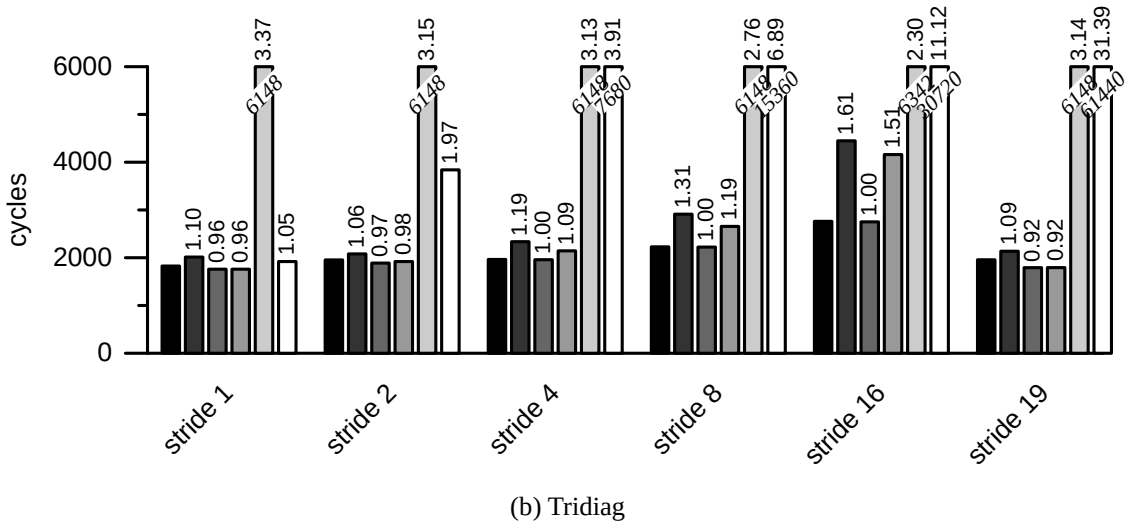


(b) Scale2

Figure 6.2. Comparative Performance with Varying Stride for the Kernels Scale and Scale2

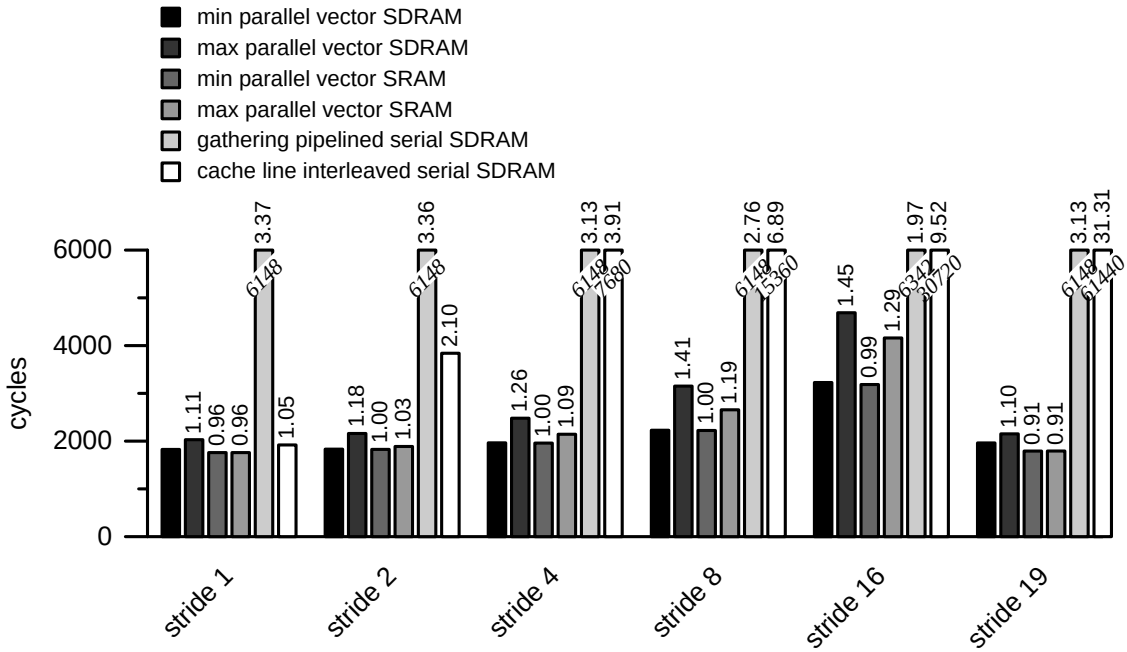


(a) Swap

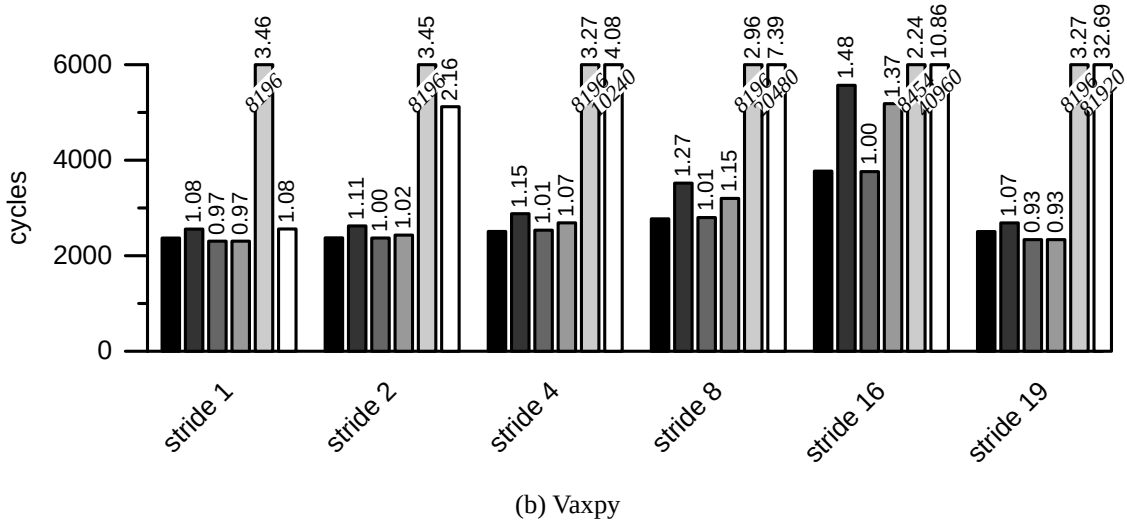


(b) Tridiag

Figure 6.3. Comparative Performance with Varying Stride for the Kernels Swap and Tridiag

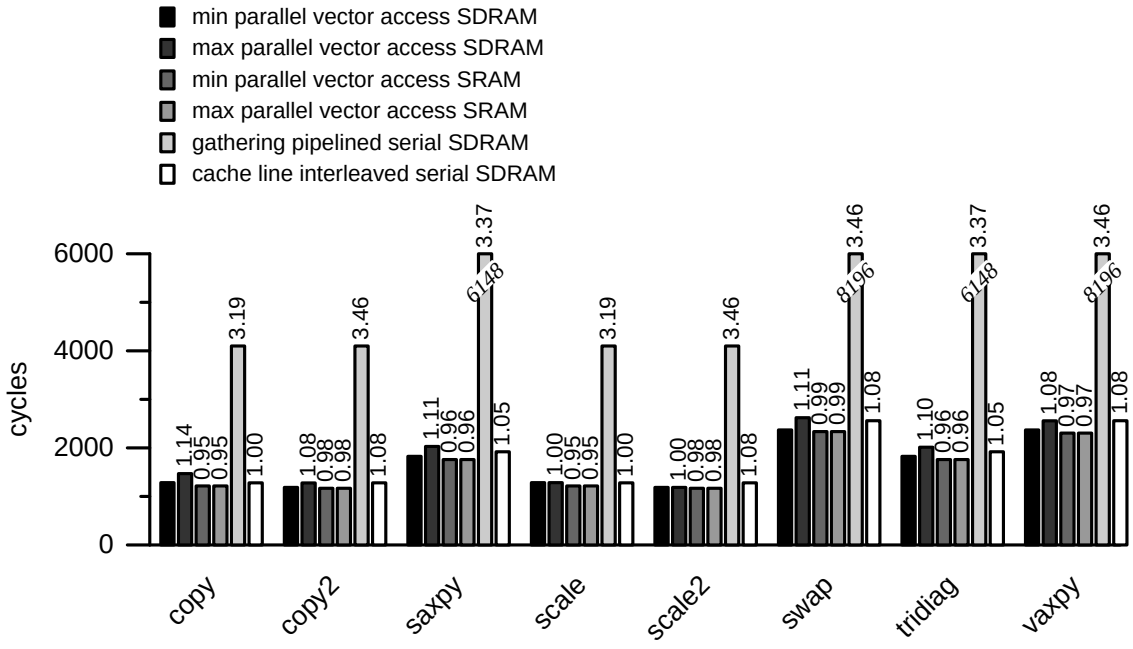


(a) Saxpy

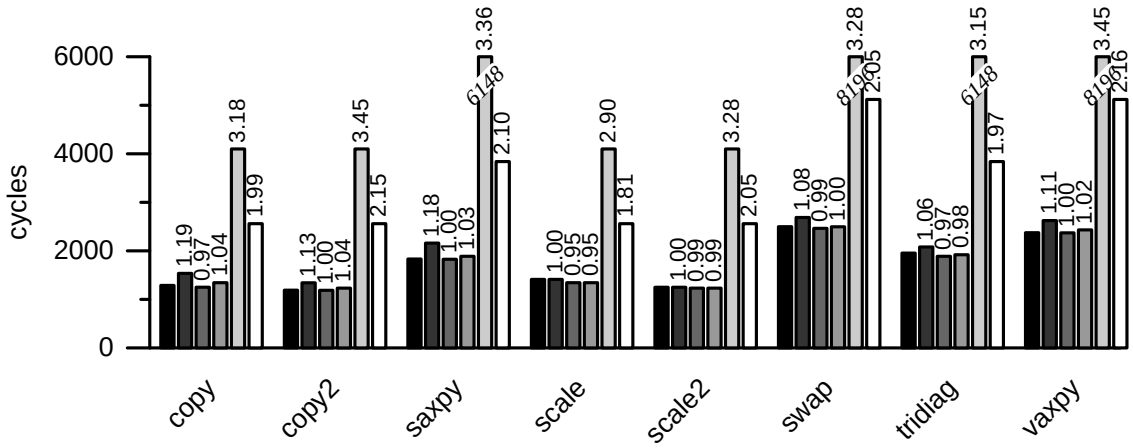


(b) Vaxpy

Figure 6.4. Comparative Performance with Varying Stride for the Kernels Saxpy and Vaxpy

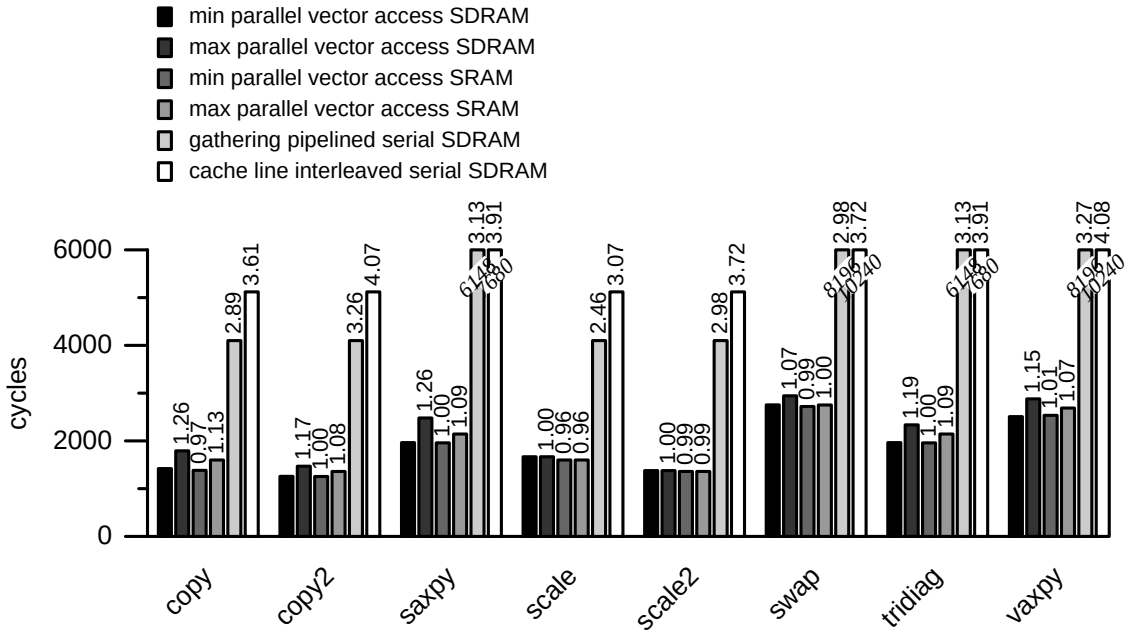


(a) Stride 1

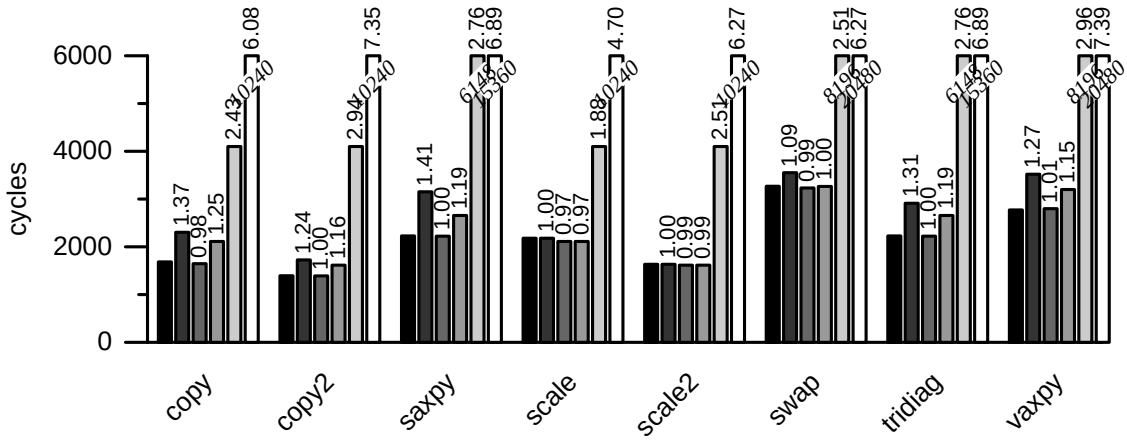


(b) Stride 2

Figure 6.5. Comparative Performance of All Kernels for Strides 1 and 2

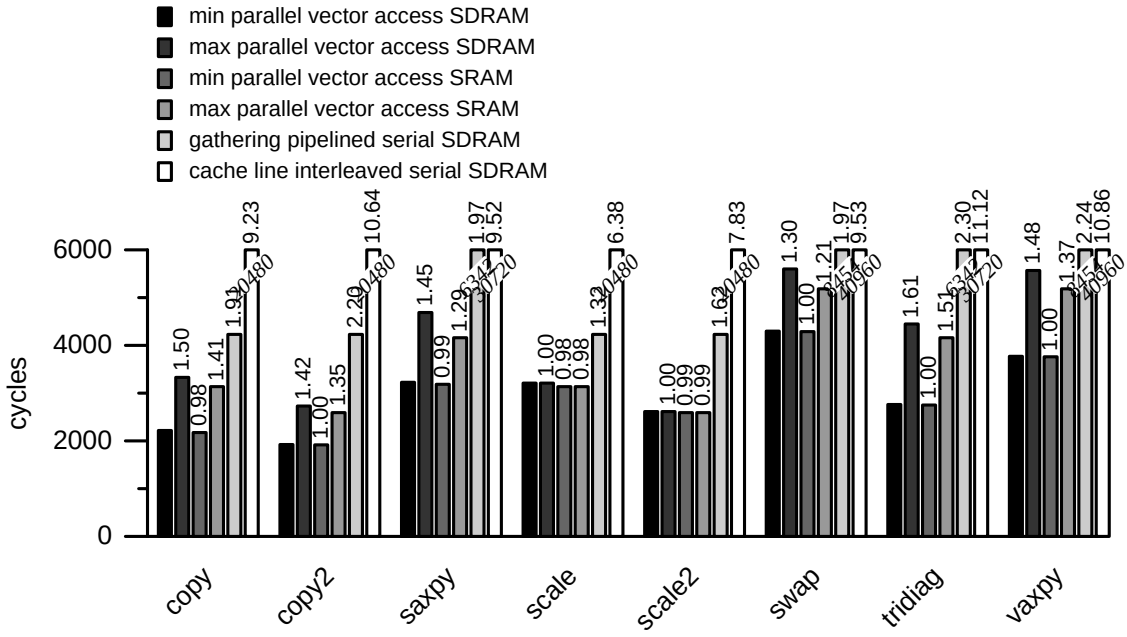


(a) Stride 4

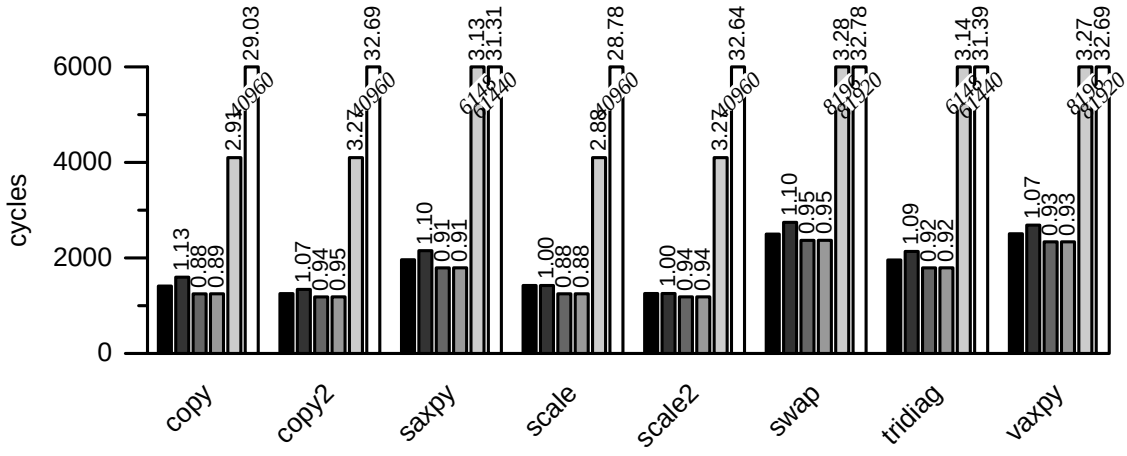


(b) Stride 8

Figure 6.6. Comparative Performance of All Kernels for Strides 4 and 8



(a) Stride 16



(b) Stride 19

Figure 6.7. Comparative Performance of all Kernels for Strides 16 and 19

1. There are fewer accesses to SDRAM, since the memory controller loads or stores individual words lines rather than whole cache lines.

2. SDRAM bandwidth is improved by operating multiple SDRAM banks in parallel.

3. Latency is reduced by employing a smart scheduling policy for accessing the SDRAM banks.

4. Bus bandwidth is better utilized by compacting vector elements into cache-lines.

For unit-stride access patterns (dense vectors or cache line fills), our PVA unit performs about the same as a cache line interleaved system that performs only line fills. As shown in Figure 6.5 (a), normalized execution time for the latter system is between 100% (for copy and scale) and 109% (for copy2, scale2, swap and vaxpy) of the PVA unit's minimum execution time for our kernels. The PVA is able to outperform the cache-line interleaved system because of its smart scheduling policy.

As stride increases, the relative performance of a cache-line interleaved system falls off rapidly. At stride four, normalized execution time rises to between 307% (for scale) and 408% (for vaxpy) of the PVA system's, and at stride 16, normalized execution time rises to between 638% (for scale) to 1112% (for tridiag). At a prime stride like 19, execution time rises to between 2878% (for scale) to 3278% (for swap). The PVA's better performance is mainly due to reasons 1, 2 and 4.

It is not possible to isolate the effect of each source of performance improvement because the amount of parallelism changes with the stride. Thus, it is not possible to vary the stride and the degree of parallelism independently.

As explained in Chapter 4, for a stride of $S = \sigma * 2^s$, every $2^s th$ bank will have a hit. Thus the degree of parallelism available is $M/2^s$. To see the effect of stride and available parallelism on memory latency, observe the results of the scale kernel in Figure 6.1 (c). Since this particular benchmark reads and writes to just one vector, it is independent of the effects of relative vector alignment. This figure shows latency gradually increasing with stride until stride 19 is reached. Note that $19 = 19 * 2^0$. Hence, the degree of parallelism is maximum, and the PVA is able to operate all 16 of its banks in parallel

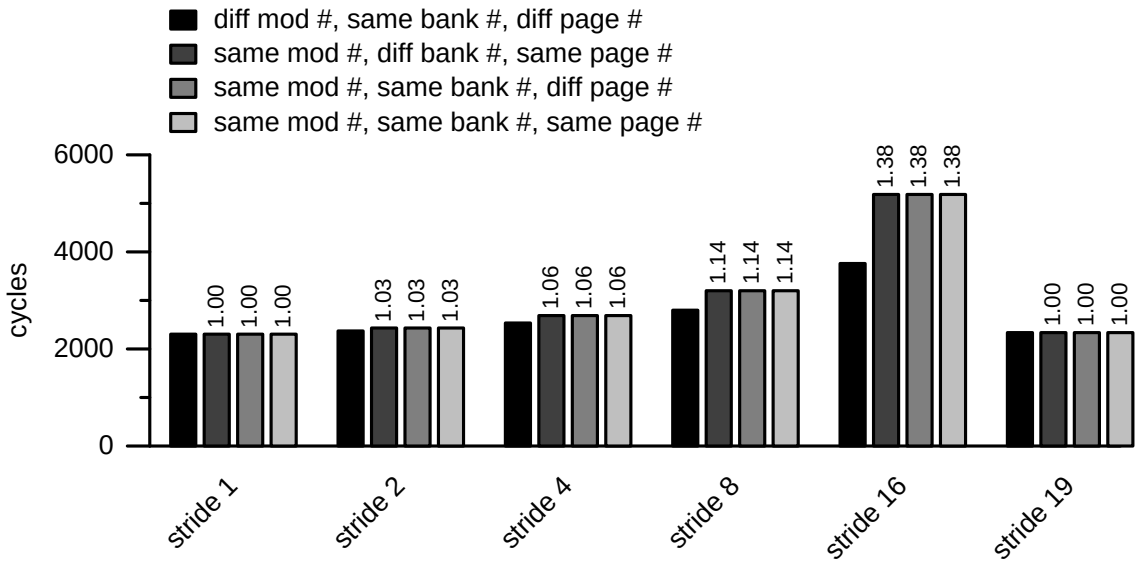
even though traditional memory systems perform poorly on prime number strides like 19. Performance for both our SDRAM PVA system and the SRAM PVA system for stride 19 are similar to the corresponding results for unit-stride access patterns. In contrast, the serial gathering SDRAM and the cache-line interleaved systems yield performances much more like those for stride 16.

Some relative vector alignments are more advantageous than others, as evidenced by the variations in the SDRAM PVA performance in Figure 6.8 (a). The SRAM version of the PVA system in Figure 6.8 (b) shows similar trends for the various combinations of vector stride and relative alignments, although its performance is slightly more robust. For small strides that hit more than two SDRAM banks, the minimum and maximum execution times for our PVA system differ only by a few percent. For strides that hit one or two of the SDRAM components, though, relative alignment has a larger impact on overall execution time. Such strides have a lot of operational overhead (SDRAM RAS/CAS latencies and precharge latencies) that cannot be overlapped with other operations and thereby hidden.

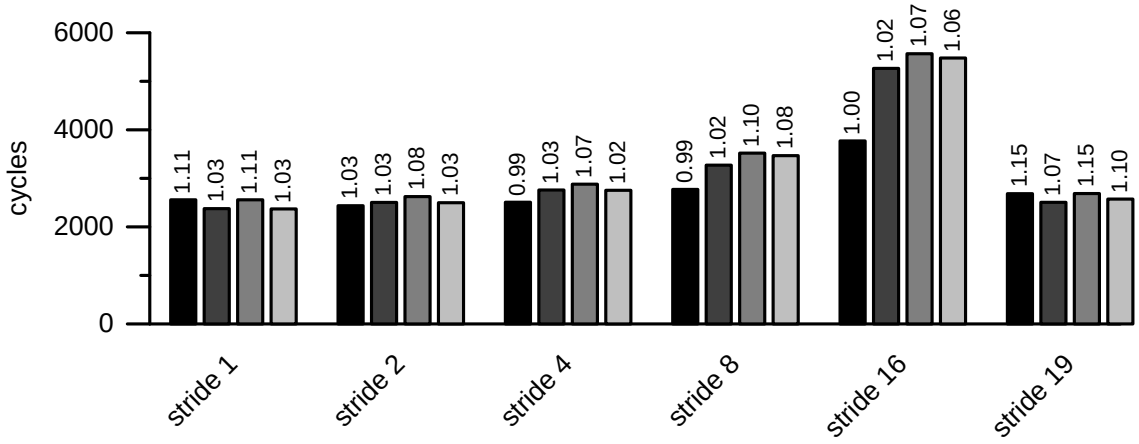
The key point to be noticed in Figure 6.8 (b) is that the PVA SDRAM unit is able to perform remarkably close to PVA SRAM. In that figure, the PVA mechanism is able to use SDRAM to achieve a performance equivalent to that of SRAM or in the worst case at most 15% slower. This is proof that the scheduling heuristics built into the PVA are extremely successful in hiding the overhead cycles associated with using SDRAM instead of SRAM. Figure 6.8 (b) illustrates that in two cases the SDRAM PVA unit outperforms SRAM. This result is an artifact of slight implementation differences between both the units that cause an additional 27 cycle delay for each experiment while running the kernels on the SRAM PVA unit. In reality, if both PVA units were identical, SDRAM would come close to the performance of SRAM, but would not outperform it.

6.3.2 The Importance of Odd Strides

As explained in Section 3.1, the major advantage of our PVA over the method used in the Command Vector Memory System (CVMS) is that our system is able to perform the firsthit address calculation for nonpower of two strides in three cycles, while CVMS takes up to 15 cycles [7]. This case is important because programs tend to have array



(a) SRAM PVA. Bars are annotated with normalized execution time with respect to the leftmost bar.



(b) SDRAM PVA. Bars are annotated with the normalized execution time with respect to the corresponding bar from (a).

Figure 6.8. Comparative Performance of the Vaxpy Kernel using SRAM and SDRAM

sizes that are powers of two. Even though both the PVA and CVMS evaluate this case in two cycles, performance still suffers. For power of two strides that are larger than the number of banks, all the hits will be on the same bank. Vector accesses will thus be serialized. Performance may be improved by padding the array width by an additional word, thereby making the width an odd number. In that case, parallelism is maximized because odd strides hit all banks. CVMS takes up to 15 cycles to evaluate this case while our PVA can perform the address computation in three cycles. Hence, our PVA algorithm can boost the performance of applications with power of two strides significantly.

CHAPTER 7

CONCLUSION

We have developed an efficient algorithm to implement parallel access to base-stride vectors and designed and simulated a hardware prototype that implements the algorithm. The prototype demonstrated the feasibility of implementing our PVA algorithm in hardware, while our performance analysis indicates that significant performance improvements are possible when using this technique. The performance of the PVA unit varied from the same as that of a cache-line oriented memory system for unit accesses to a maximum of 32.8 times faster for strided accesses. Studies of how this scheme will interact with virtual memory and functional simulation of full-program benchmarks need to be done.

It is interesting to note that industry seems to have started using approaches similar to those described in this thesis. In particular, the RMC2 "constraint-based" memory controller from Rambus, Inc., uses constraints similar to the PVA back-end and uses a simple rule to skip unnecessary precharge operations. Our work predates the RMC2 controller documentation. As such, the design of the RMC2 controller can be considered an independent validation of some of the design concepts presented in this thesis.

The general technique of making the memory controller aware of application vectors can be carried forward to common patterns other than base-stride. For example, the PVA unit described here can be extended to handle vector indirect scatter-gather operations by performing the operation in two phases: (i) loading the indirection vector into the appropriate bank controllers and then (ii) loading the appropriate vector elements. Loading the indirection vector is simply a unit-stride vector load operation. After the indirection vector is loaded, its contents can be broadcast across the vector bus. Each bank controller can easily determine which elements of the vector reside in its SDRAM by snooping this

broadcast and performing a simple bit-mask operation on each address broadcast (two per cycle). Then, each bank controller can perform its part of the vector indirect gather operation in parallel, and the result can be coalesced from the staging units in much the same way as is now done for strided accesses. Another example is handling the bit-reversal phase of a Fast Fourier Transform algorithm. The data for such algorithms is normally stored as a sequence of complex numbers, but the algorithm has to re-order the data into a form that is more amenable to later processing. This reordering phase, called bit reversal, has extremely bad cache locality for large data sets. It is quite easy to make the memory controller aware of the bit-reversed application vector pattern and let it gather/scatter sequential data into bit-reversed form. It can be done by reversing some number of low order bits of the address and using the new address to access memory, incrementing the original address and repeating the address reversal till a cache line worth of data is fetched or stored. The scatter/gather operation on bit-reversed vectors is inherently sequential for word-interleaved memory systems, but can be parallelized for block interleaved memory systems.

Though it is possible to build in and exploit knowledge of common application vectors into a memory controller (e.g., a bit-reversed application vector can be built into a memory controller for DSP and multimedia applications), it will be important to study how knowledge of application vectors can be conveyed effectively at runtime to memory controllers. However, the above examples of indirection and bit-reversed vectors seem to suggest that the kind of processing required to scatter/gather such application vectors would be quite complicated. As such, it may not be possible to implement such transformations within a general purpose framework.

In summary, the parallel vector access unit described in this thesis shows great promise for improving the memory performance of applications that use base-stride style vector access. The performance results are also an indicator of the improvements that could be obtained by raising the sematic level of the processor memory interaction by providing the memory controller with the knowledge of the memory access pattern of applications.

REFERENCES

[1] ADVANCED MICRO DEVICES. Inside 3DNow!(tm) technology. http://www.amd-.com/products/cpg/k623d/inside3d.html.

[2] BENITEZ, M., AND DAVIDSON, J. Code generation for streaming: An access/execute mechanism. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems* (Apr. 1991), pp. 132–141.

[3] BUCHER, I. Y., AND SIMMONS, M. L. Measurement of memory access contentions in multiple vector processor systems. In *Proceedings of the 1991 International Conference on Supercomputing* (Nov. 1991), p. 806.

[4] CARTER, J., HSIEH, W., STOLLER, L., SWANSON, M., ZHANG, L., BRUNVAND, E., DAVIS, A., KUO, C.-C., KURAMKOTE, R., PARKER, M., SCHAELICKE, L., , AND TATEYAMA, T. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture* (Jan. 1999), pp. 70–79.

[5] CHEN, C.-L., AND LIAO, C.-K. Analysis of vector access performance on skewed interleaved memory. In *Proceedings of the 16th Annual International Symposium on Computer Architecture* (May 1989), pp. 387–394.

[6] CHEN, T.-F. *Data Prefetching for High Performance Processors*. PhD thesis, Univ. of Washington, July 1993.

[7] CORBAL, J., ESPASA, R., AND VALERO, M. Command vector memory systems: High performance at low cost. Tech. Rep. UPC-DAC-1999-5, Universitat Politecnica de Catalunya, Jan. 1999.

[8] DEL CORRAL, A., AND LLABERIA, J. Access order to avoid inter-vector conflicts in complex memory systems. In *Proceedings of the Ninth International Parallel Processing Symposium* (1995), pp. 404–410.

[9] DONGARRA, J., DUCROZ, J., DUFF, I., AND HAMMERLING, S. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software 16*, 1 (Mar. 1990), 1–17.

[10] FLYNN, M. J. *Computer Architecture Pipelined and Parallel Processor Design*. Jones and Bartlett Publishers, Inc., 1995, ch. 7, pp. 438–443.

[11] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1978, ch. Appendix

A5, pp. 236–237.

[12] GILLINGHAM, P. SLDRAM Architectural and Functional Overview. http://www-.sldram.com/Documents/SLDRAMwhite970910.pdf, August 1997.

[13] HALL, L. A. Approximation Algorithms for Scheduling. In *Approximation Algorithms for NP-Hard Problems*, D. N. Hochbaum, Ed. PWS Publishing Company, 1997, pp. 1–43.

[14] HSU, W., AND SMITH, J. Performance of cached DRAM organizations in vector supercomputers. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May 1993), pp. 327–336.

[15] HWANG, K., AND BRIGGS, F. A. *Computer Architecture and Parallel Processing*. McGraw-Hill, Inc., 1985, ch. 4, pp. 233–324.

[16] IKOS SYSTEMS. IKOS Libraries, 1999. http://www.ikos.com/products/libraries/-index.html.

[17] IKOS SYSTEMS. VirtualLogic, 1999. http://www.ikos.com/products/vsli/index.-html.

[18] INTEL. MMX programmer's reference manual. http://developer.intel.com/drg/-mmx/Manuals/prm/prm.htm.

[19] IRANI, S., AND KARLIN, A. R. Online Computation. In *Approximation Algorithms for NP-Hard Problems*, D. N. Hochbaum, Ed. PWS Publishing Company, 1997, pp. 521–559.

[20] KONTOTHANASSIS, L. I., SUGUMAR, R. A., FAANES, G. J., SMITH, J. E., AND SCOTT, M. L. Cache performance in vector supercomputers. In *Proceedings of Supercomputing '94* (Nov. 1994), pp. 255–264.

[21] KRISHNA, C. M., AND SHIN, K. G. *Real-time Systems*. McGraw-Hill, 1997.

[22] LEE, K. *The NAS860 Library User's Manual*. NASA Ames Research Center, Mar. 1993.

[23] MCCALPIN, J. Stream: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/, 1999.

[24] MCKEE, S. *Maximizing Memory Bandwidth for Streamed Computations*. PhD thesis, School of Engineering and Applied Science, University of Virginia, May 1995.

[25] MCKEE, S., AND WULF, W. Access ordering and memory-conscious cache utilization. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture* (Jan. 1995), pp. 253–262.

[26] MCKEE, S. A., ALUWIHARE, A., CLARK, B. H., KLENKE, R. H., LANDON,

T. C., OLIVER, C. W., SALINAS, M. H., SZYMKOWIAK, A. E., WRIGHT, K. L., WULF, W. A., AND AYLOR, J. H. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the 10th ACM International Conference on Supercomputing* (May 1996), pp. 125–132.

[27] MCMAHON, F. The livermore fortran kernels: A computer test of the numerical performance range. Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

[28] MICRON TECHNOLOGY, INC. 256mb: Sdram. http://www.micron.com/mti/msp/-pdf/datasheets/256MSDRAM.pdf.

[29] MIPS TECHNOLOGIES, INC. MIPS extension for digital media with 3D. http://-www.mips.com/Documentation/isa5_tech_brf.pdf.

[30] MITSUBISHI SEMICONDUCTORS. 4194304-bit (262144-Word by 16-bit) CMOS STATIC RAM. http://www.mitsubishi-chips.com/data/datasheets/memory/mem-pdf/ds/d99019.pdf, 1998.

[31] MITSUBISHI SEMICONDUCTORS. 256M Synchronous DRAM. http://www.mits-ubishi-chips.com/data/datasheets/memory/mempdf/ds/a99005.pdf, July 1999.

[32] MOTOROLA. Altivec(tm) technology programming interface manual, rev. 0.9. http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/manuals/altivecpim.pdf, Apr. 1999.

[33] MOYER, S. *Access Ordering Algorithms and Effective Memory Bandwidth*. PhD thesis, School of Engineering and Applied Science, University of Virginia, May 1993.

[34] PEIRON, M., VALERO, M., AYGUADE, E., AND LANG, T. Vector multiprocessors with arbitrated memory access. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (June 1995), pp. 243–252.

[35] RAMBUS, INC. 256/288-Mbit Direct RDRAM (Advance Information. http://ww-w.rambus.com/developer/downloads/rdram256d.pdf, September 1998.

[36] RAMBUS, INC. Rambus technology overview. http://www.rambus.com/develope-r/downloads/TechOV.pdf, 1999. DL-0040-00.

[37] RAMBUS, INC. RMC2 Data Sheet (Advance Information. http://www.rambus.-com/developer/downloads/rmc2_overview.pdf, August 1999.

[38] SCHUMANN, R. Design of the 21174 memory controller for DIGITAL personal workstations. *Digital Technical Journal 9*, 2 (Jan. 1997).

[39] SMITH, J. E., AND TAYLOR, W. R. Characterizing memory performance in vector multiprocessors. In *Proceedings of the 1992 International Conference on Supercomputing* (1992), pp. 35–44.

[40] SUN. The VIS advantage: Benchmark results chart VIS performance. Whitepaper WPR-0012.

[41] SUN. VIS instruction set user's manual. http://www.sun.com/microelectronics/-manuals/805-1394.pdf.

[42] TYLER, J., LENT, J., MATHER, A., AND NGUYEN, H. Altivec: Bringing vector technology to the powerpc processor family. In *Proceedings of the 1999 IEEE International Performance, Computing, and Communications Conference* (Feb. 1999).

[43] VALERO, M., LANG, T., LLABERIA, J., PEIRON, M., AYGUADE, E., AND NAVARRO, J. Increasing the number of strides for conflict-free vector access. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May 1992), pp. 372–381.

[44] VALERO, M., LANG, T., PEIRON, M., AND AYGUADE, E. Conflict-free access for streams in multi-module memories. Tech. Rep. UPC-DAC-93-11, Universitat Politecnica de Catalunya, Barcelona, Spain, 1993.

[45] WOLFE, M. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, Massachusetts, 1989.

[46] ZALEWSKI, J. What Every Engineer Needs to Know about Rate-Monotonic Scheduling: A Tutorial. In *Advanced Multimicroprocessor Bus Architectures*, J. Zalewski, Ed. 1995, pp. 321–335.