Design of a Parallel Vector Access Unit for SDRAM Memory Systems

Binu K. Mathew, Sally A. McKee, John B. Carter, Al Davis

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

Draft. Do not distribute [1]

UUCS-99-006

Department of Computer Science
3190 Merrill Engineering Building
University of Utah
Salt Lake City, UT 84112

February 4, 2000

Abstract

Parallel Vector Access is a technique that exploits the regularity of vector or stream accesses to perform them efficiently in parallel on a multi-bank memory system. The performance of applications that have vector accesses may be improved using 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 speeded up by broadcasting vector operations to all banks of memory in parallel, each of which implements an algorithm to determine which elements of the requested vector they contain. 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 Parallel Vector Access (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. The the performance of such a memory controller on vector kernels is studied by gate level simulation and the results analyzed. Because of the parallel approach, the PVA is able to load elements up to 32.8 times faster than a conventional memory system and 3.3 times faster than a pipelined vector unit, without hurting normal cache line fill performance.

Keywords: memory architecture, memory latency, memory bandwidth, bus utilization, cache efficiency

[1]This effort was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the US Government.

Technical Areas: Architecture, Memory Systems, Hardware Design, Vector Processing

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 be able to 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.

Several applications that suffer from poor cache locality have predictable access patterns. Programs that operate on large multi-dimensional arrays are an example of this class of applications. Though modern processors generate memory operations at several granularities, such operations are filtered through the cache and the real memory accesses are done by the cache controllers at cacheline grain size. Hence, memory operations are seen at the DRAM end 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 accesses. Conceptually a cacheline request can be considered a fixed length vector, and a memory controller serves requests to load or store fixed length vectors. When applications access their array 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. The former case happens 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 reasons.

- Poor cache utilization: The application uses only some elements of a memory vector, but the whole vector occupies space in the cache. The amount of data the cache can handle is therefore reduced.
- 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 amount of usable data that can be transferred across the bus is reduced.

Henceforth let us call a sequence of array elements accessed by an application that occupies the same number of memory words as a cacheline an application vector. The problem of irregular applications is that their performance 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:

- Non-unit, but constant stride which 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.

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. It discusses architectural features that enable the memory controller to efficiently load and store base-stride vectors by operating multiple memory banks in parallel. In closing, it also provides some suggestions on how other common application vectors can be handled.

The domain of applicability extends from the traditional scientific vector processing to the realm of desktop computing. Several new instruction set extensions (e.g., Intel's MMX for the Pentium [13], AMD's 3DNow! for the K6-2 [1], MIPS's MDMX [24], Sun's VIS for the UltraSPARC [35], and Motorola's AltiVec for the PowerPC [27]) bring stream and vector processing to the domain of desktop computing. The results for some applications that use these vector extensions are quite promising [36, 34], 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 architectural features and algorithms used for this purpose will be collectively referred to as PARALLEL VECTOR ACCESS or PVA. The organization of the main memory system assumed in the rest of this thesis is shown in figure 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 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 later 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 PVA algorithms for parallel base-stride access and sets the background for chapter 5 that describes the implementation architecture. Chapter 6

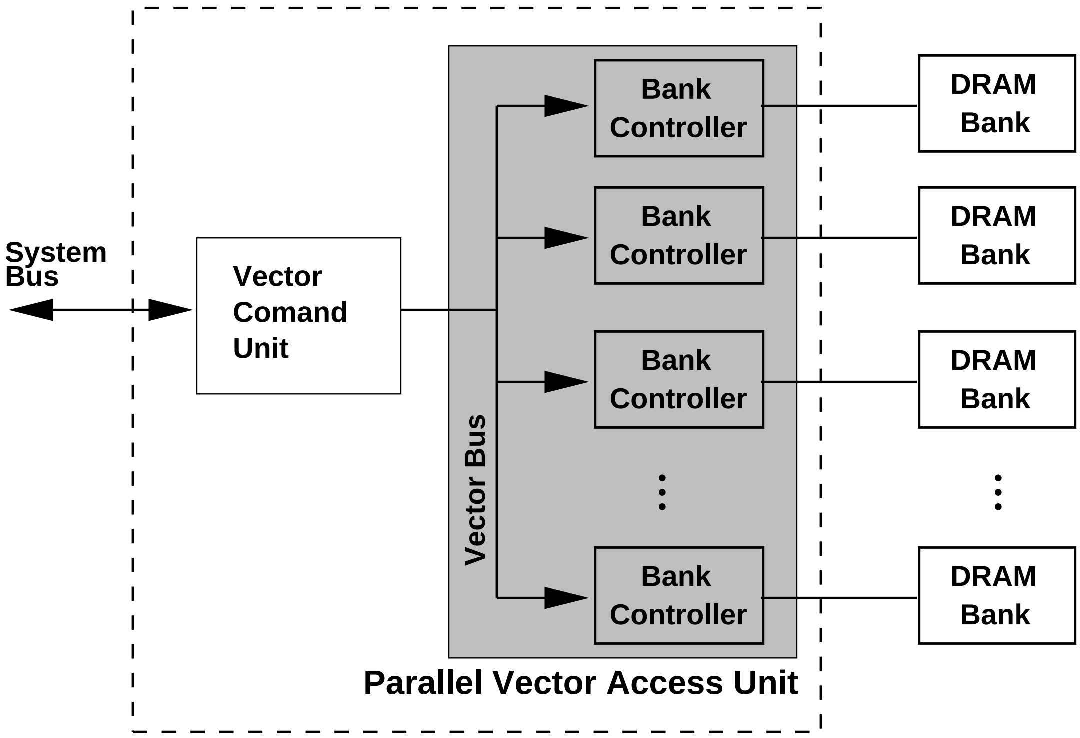


Figure 1: Memory System Organization

describes the experiments that were done and analyzes their results. Chapter 8 concludes the results of this work and presents some directions for future research.

2 Memory Technology Background

The memory technology used for high performance vector processors has traditionally been SRAM while DRAM has been more common on almost all other macines. A major portion (often more than half) of the cost of a vector super computer consists of the cost of the memory system [15][10]. The reason for this preference of DRAM over SRAM 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 is 4 Mbits [25][26]. 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 at 100 Mhz, i.e. with a clock cycle time of 10ns. What sets the SRAMs 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 towords exploiting this ability to pipeline accesses to the maximum. RAMBUS, SLDRAM and the Alpha 21174 memory controller discussed later in this chapter are all examples of this trend. All these memory systems try to hide RAS and pre-charge latencies of DRAM as much as possible by exploiting row hits within a stream of accesses. To understand how these new memory technologies work let us look at how SRAM and DRAM are organized.

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

2.2 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 yielding much greater densities for DRAM over SRAM. Figure 3 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

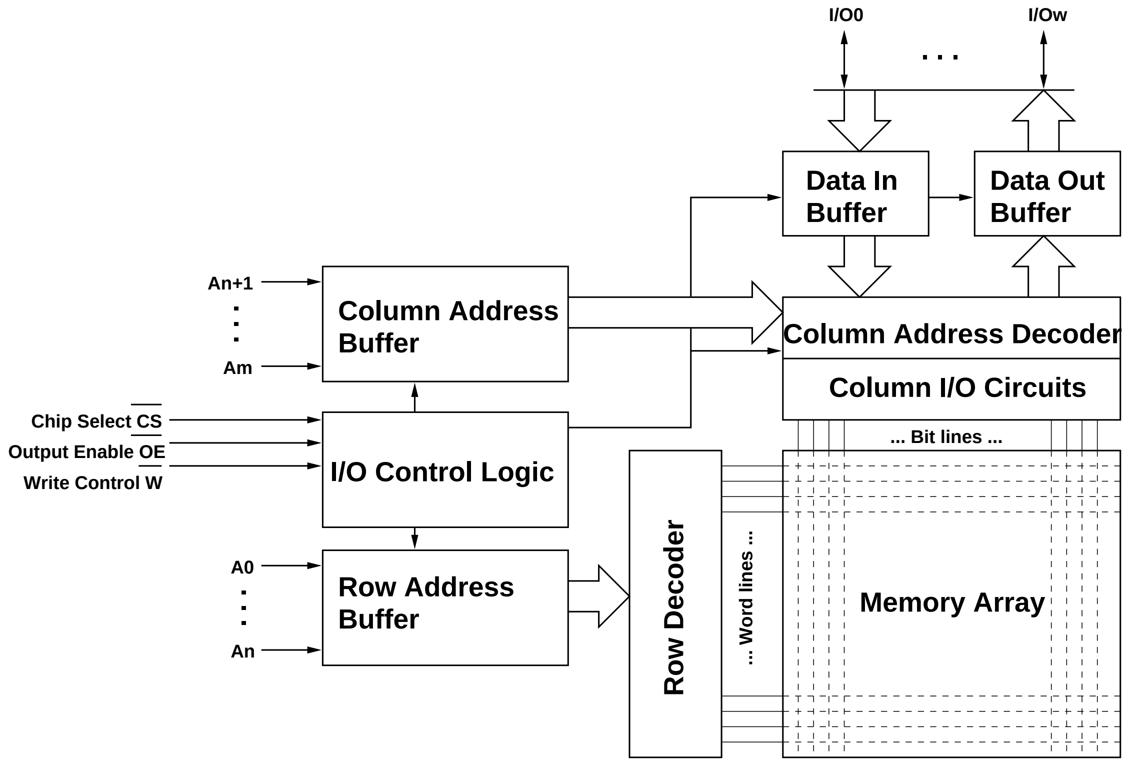


Figure 2: Block Diagram of SRAM

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 has been read, a pre-charge 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.

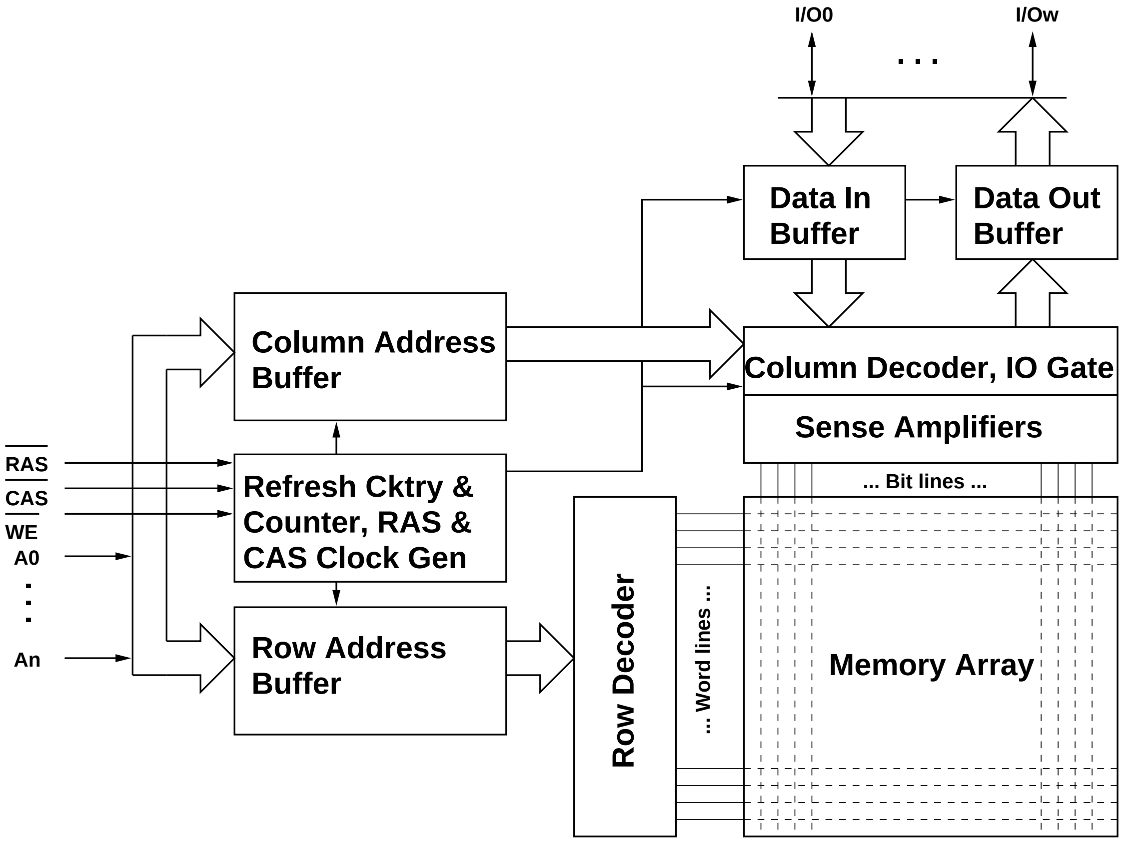


Figure 3: Conventional DRAM

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 explained 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 this data can be accessed relatively fast before issuing a final pre-charge operation 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 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 on 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.

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) [8]. SLDRAM uses a multi-drop bus that connects a memory controller and up to eight SLDRAM devices. The controller sends commands to SLDRAM deices 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 called DCLK0 and DCLK1. Two clocks are used to minimize the gap required when control of the DataLink is transferred from one device to another. Like in the case of 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 [31][30]. The interface between the DRDRAM and the RAMBUS memory controller is called a RAMBUS channel. Like in the case of SDRAM this 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 send by DRDRAM to the controller is synchronous with ClockToMaster and data send from the memory controller to DRDRAM is synchronous with ClockFromMaster. It transfers data on both edges of the clock permitting transfer rates of 600MHz or 800MHz and is capable of a sustained bandwidth of 1.6GB/s. The core operates at one eighth of the data frequency. DRDRAM has a 16 bit data buse 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.

2.3.6 RAMBUS DRAM (RDRAM)

Architecturally, RDRAM is like a more primitive version of DRDRAM described in the previous section except that It operates the channel at 300MHz and can achieve only a peak data rate of 600MB/s using a byte wide multiplexed address/data bus.

2.4 Advanced Memory Controllers

This section describes two memory controllers that represent the state of the art. The Alpha 21174 controller is in production at the time of this writing while the RAMBUS RMC2 controller is still in the early stages of design.

2.4.1 The Alpha 21174 Memory Controller

The Alpha 21174 memory controller is an ASIC used in DIGITAL Personal Workstations [33]. It interfaces with a set of SDRAM DIMMs 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 predictor for each DIMM to decide whether to close the SDRAM row after each access. If a hit is predicted the row is left open and if a miss is predicted the row is closed. To predict row hits and misses it uses a four bit history for each DIMM to record hits and misses. 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. The adaptive hot-row management provided a 23% improvement in measured best-case memory latency and a 7% improvement in measured bandwidth for McCalpin's STREAM benchmark [18].

2.5 RAMBUS Memory Controller (RMC2)

The RMC2 memory controller from RAMBUS Inc has many features similar to the SDRAM interface of the PVA unit described in this thesis though the PVA work pre-dates the RMC2 controller [32]. RMC2 is a constraint based memory controller in that it allows for both the logical and timing constraints of a RAMBUS memory system and provides optimal channel bandwidth possible without reordering transactions. It 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.

3 Related Work

There has been a tremendous amount of research on optimizing memory system performance. Most of this work 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. In many cases it may not be straightforward to extend these techniques to DRAM memory systems because of their non-uniform access time. 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. 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 [28]. Access scheduling techniques attempt to separate the execution of a load/store instruction from the execution of the instruction that produces/consumes its operand, 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 [28].

The DEC Alpha 21174 memory controller described in section 2.4.1 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). A four-bit predictor tracks whether accesses hit or miss the most recent row in each row buffer, and the controller leaves a row open only when a hit is predicted [33]. For McCalpin's STREAM benchmark [18], this simple policy yields best-case improvements in memory latency and bandwidth of 23% and 7%, respectively.

The RMC2 memory controller described in section 2.5 tries to use timing and logical constraints and skips pre-charge cycles when possible to provide optimal channel bandwidth. 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 non-caching load instructions) and then writing them to a pre-allocated portion of cache [17]. 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 [20]. 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 [37], and then extend the work for multiple vectors [38]. del Corral and Llaberia analyze a related hardware scheme for avoiding bank conflicts among multiple vectors in complex memory systems [6]. 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 [5] (CVMS). The CVMS exploits parallelism and locality of reference to improve the effective bandwidth for vector accesses from out-of-order vector processors with dual-banked SDRAM memories. Rather than sending individual requests to specific devices, the CVMS broadcasts commands requesting multiple independent words, a design idea that we adopted. Section controllers receive the broadcasts, compute subcommands for the portion of the data for which they are responsible, and then issue the addresses to the memory chips under their control. The memory subsystem orders requests to each dual-banked device, attempting to overlap precharge operations to each internal SDRAM bank with access operations to the other. Simulation results demonstrate performance improvements of 15% to 54% compared to a serial memory controller. At the behavioral level, our bank controllers resemble CVMS section controllers, but the specific hardware design and parallel access algorithm is substantially different, as described in chapters 4 and 5.

The Command Vector Memory System's hardware scheme for computing the vector subcommands is based on earlier access-scheduling work for vector multiprocessors [29]. 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 [5]. 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.

Another system similar to ours is the Stream Memory Controller (SMC) of McKee, et al. [21]. The SMC combines programmable stream buffers and prefetching within a memory controller that performs intelligent DRAM scheduling. The SMC dynamically reorders vector or stream accesses to exploit parallelism among multiple banks and to exploit locality of reference within DRAM page buffers. For most vector alignments and strides on a uniprocessor system, simple ordering schemes were found to perform competitively with sophisticated ones [19].

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 [39]. At the other end of the spectrum lie hardware vector or stream detection schemes, which may be implemented via reference prediction tables [4]. 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 [cite]. Impulse supports multiple views of the same data [3]. 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 cache-lines 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. While such schemes are suitable for uniform-access components like SRAM access ordering for non-uniform access memory components like SDRAM require performing address arithmetic which gets complicated when skewing schemes are used. Moreover as Hsu and Smith demonstrate that it is useful to take advantage of spatial locality while using such components [10]. 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 [9]. Some of the work in scheduling theory can act as starting points for implementing access re-ordering 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 [cite-for-NP-completeness]. In general the algorithms in this area are too complex to be implemented fast in hardware.

3.4.1 Online Algorithms

Online algorithms try to make decision using incomplete information often by trying to approximate an optimal offline algorithm [14]. Good examples are OS page replacement algorithms and what is known as the Ski Rental problem in the literature. There are variants like deterministic online algorithms and randomized online algorithms.

3.4.2 Rate Monotonic Scheduling

Rate Monotonic Scheduling is a technique often used to analyze the schedulability of real-time tasks [40][16]. Such tasks are characterized by 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 + 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 repeat interval (which is not known in the case of memory access streams) makes RMS unsuitable for use in memory access re-ordering hardware.

3.4.3 Nonpreemptive Earliest Deadline First (EDF) Scheduling

Unlike Rate Monotonic Scheduling the 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 pre-emptive version of this algorithm is provably optimal, but the nonpre-emptive version is more amenable to hardware implementation [16].

3.5 Operations Research/Logic Minimization

The Transportation problem from Operations Research and the Binate Covering Problem (BCP) often discussed in logic minimization literature are very similar in nature and aim at generating provably optimal solutions for optimization problems that involve complex sets of choices. 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. Chose the solution with the best cost.

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

4 PVA Algorithms

As explained in Chapter 1, BASE-STRIDE is a common and important type of application vector. This chapter explains algorithms that a multi-bank memory system can use for parallelizing this type of access.

4.1 Parallel Access to Base-Stride Vectors

Processing a base-stride type of 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 PVA unit shown earlier in figure 1 parallelizes this task 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 subvector of each bank.

4.1.1 Terminology

We first introduce the terminology used in describing the PVA algorithms. BASE-STRIDE vector operations are represented as a 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. We refer to the i^{th} element in the vector V as $V[i]$. For example, vector $V = < A, 4, 5 >$ designates elements $A[0]$, $A[4]$, $A[8]$, $A[12]$, and $A[16]$ of the array A where $V[0] = A[0]$, $V[1] = A[4]$ and so on.

Let M be the number of memory banks, such that $M = 2^m$. Let N be the number of words in a cache-line, such that $N = 2^n$.

Three functions implement the crux of our scheduling scheme.

- $DecodeBank(addr)$ returns the bank number b for an address $addr$; it is implemented as a bit-select operation equivalent to $(addr \gg n) \, mod \, M$.
- $FirstHit(V, b)$ 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 that indicates that no such element exists.
- $NextHit(S)$ returns an increment δ such that if a bank holds $V[n]$, it also holds $V[n+\delta]$.

4.1.2 The Difficulty of Implementing FirstHit(V,b) for Cache-line Interleave

In this section we derive an algorithm for FirstHit(V,b) and show why it is difficult to implement FirstHit(V,b) for a cache-line interleaved memory. In a later section we will introduce a technique of converting cache-line interleave to appear like word interleave for the purpose of computing FirstHit(V,b) and an efficient implementation of FirstHit(V,b) for word interleave.

Analysis of FirstHit(V,b) Let us analyze FirstHit() on a case by case basis to understand it better so that we can find a more parallel approach for implementing it.

Case 0 : $DecodeBank(V.B) = b$

The easiest case is when DecodeBank(V.B) returns b. In other words V[0] is contained in bank b, so FirstHit() returns 0.

Definition :

Let

- $\triangle b = (V.S\, mod\, NM)/N$, the number of banks skipped between any two consecutive elements V[i] and V[i+1]
- $\triangle \theta = (V.S\, mod\, NM)\, mod\, N$ and $\theta = V.B\, mod\, N$, the difference in offest within the block between any two consecutive elements V[i] and V[i+1].
- $\theta = V.B\, mod\, N$, the offset within the block of the first element.

When $DecodeBank(V.B) \neq b$, we have two more cases to consider.

Case 1 : $\triangle\theta = 0$

In this case no matter which banks the vector V hits, the offset within the block will always be θ. If V[0] is contained in bank b' then V[1] is in bank $(b' + \triangle b)\, mod\, M$, V[2] in $(b' + 2\triangle b)\, mod\, M$ and so on. Because of the properties of modulo arithmetic $(b' + n * \triangle b)\, mod\, M$ is a repeating sequence for $n = 0, 1, ..\infty$ with a period of at most M. Hence, when $\triangle\theta = 0$ FirstHit(V,b) may be defined as:

Let

$$d = \begin{cases} DecodeBank(V.B) - b,\ if\ DecodeBank(V.B) \geq b \\ DecodeBank(V.B) + M - b,\ if\ DecodeBank(V.B) < b \end{cases}$$

$$FirstHit(V, b) = \begin{cases} d/\triangle b,\ if\ d < V.L\, and\, \triangle b\, divides\, d \\ no\, hit,\ otherwise \end{cases}$$

Case 2 : $N > \triangle\theta > 0$

If V[0] is contained in bank b' at an offest 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 add up to exceed 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 : $\theta + (V.L - 1) * \triangle\theta \geq N$

In this case, when ever $\triangle\theta$s add up to reach N, the bank as calculated by cases 1 and 2.1 need to be incremented by 1. This increment can cause the calculation to shift between multiple cyclic sequences. It is not easy to define FirstHit() in this case.

Examples:

In the following examples assume M=8 and N=4.

1. Let B=0, S = 8, L=16.

   This is case 1 with $\theta = 0$, $\triangle\theta = 0$, $\triangle b = 2$.

   The repeating sequence of banks hit by this vector is 0,2,4,6,0,2,4,6,...

2. Let B=5, S = 8, L=16.

   This is case 1 with $\theta = 1$, $\triangle\theta = 0$, $\triangle b = 2$.

   The repeating sequence of banks hit by this vector is 1,3,5,7,1,3,5,7,...

3. Let B=0, S = 9, L=4.

   This is case 2.1 with $\theta = 0$, $\triangle\theta = 1$, $\triangle b = 2$.

   The sequence of banks hit by this vector is 0,2,4,6.

4. Now consider B=0, S = 9, L=10.

   This is case 2.2 with $\theta = 0$, $\triangle\theta = 1$, $\triangle b = 2$.

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

In the next section we use the insights gained from the case by case examination of FirstHit() to derive a generic algorithm that can handle all the cases.

Deriving an Algorithm for FirstHit(V,b) In this section we derive an algorithm that can handle all the cases of FirstHit() and show why it is not a good idea to implement it in hardware. Later we present a method of transforming the problem into one which is suitable for hardware implementation.

Since we previously defined $\theta = V.B \, mod \, N$ in section 4.1.2, we have

$\theta < N$ (0)

Define $S_0 = V.S \, mod \, NM$ and $S_{-1} = NM$

Then the problem of FirstHit() is essentially that of finding the least integers p_1 and p_2 such that $0 \le \theta + p_1 S_0 - p_2 NM - dN < N$ [2]

Let $\gamma = \theta - dN$

We need to find p_1 and p_2 st, $0 \le \gamma + p_1 S_0 - p_2 NM < N$ (1)

i.e. $-\gamma \le p_1 S_0 - p_2 NM < N - \gamma$

i.e. $\gamma \ge p_2 NM - p_1 S_0 > \gamma - N$

i.e. $S_0 + \gamma \ge p_2 NM - (p_1 - 1) S_0 > S_0 + \gamma - N$ (2)

To solve for p_1 and p_2 one at a time we can substitute the above inequality with $S_0 + \gamma \ge p_2 NM \, mod \, S_0 > S_0 + \gamma - N$ if $S_0 > S_0 + \gamma - N$ (3)

(3) is satisfied if $0 > \gamma - N$. i.e. if $N > \gamma$. i.e. if $N > \theta - dN$. i.e. if $(d+1)N > \theta$. Which is true by (0).

After solving for p_2 we can 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).

But $p_2 NM \, mod \, S_0 = p_2 (NM \, mod \, S_0) \, mod \, S_0$. Substituting $S_1 = NM \, mod \, S_0$

we need to solve $S + \gamma \ge p_2 S_1 \, mod \, S_0 > S_0 + \gamma - N$

To solve $S_0 + \gamma \ge p_2 S_1 \, mod \, S_0 > S_0 + \gamma - N$ we need to find p_3 st

$S_0 + \gamma \ge p_2 S_1 - p_3 S_0 > S_0 + \gamma - N$

i.e. $-S_0 - \gamma \le p_3 S_0 - p_2 S_1 < -S_0 - \gamma + N$

[2] To visualize this situation, consider two impulse trains, one starting at $time = \theta - dN$ with period=$V.S \, mod \, NM$, and the other starting at time=0 with period=NM. The inequality is solved at the p1*th* period of the first wave if the edges of both the waves are a distance less than N apart. The analogy helps to illustrate that while it is easy to solve such inequalities in a continuous domain, it is harder to solve them in a discrete domain - i.e. the integral period of the waveforms when their edges are closer than N.

i.e. $-\gamma \leq (p_3 + 1)S_0 - p_2 S_1 < -\gamma + N$

i.e. $0 \leq \gamma + (p_3 + 1)S_0 - p_2 S_1 < N$ (4)

Notice that (4) is of the same format as (1). At this point the same algorithm can be recursively applied. Recursive application can be terminated whenever we have an S_i such that $S_i < N$ at steps (1) or (4). Since each $S_i = S_{i-1}\, mod\, S_{i-2}$ the S_is reduce monotonically. Hence the algorithm will always terminate.

A simpler version of this algorithm that has $\gamma = \theta$ can be used for NextHit(). The C code for the NextHit() function is shown below.

```
unsigned NextHit(unsigned theta, unsigned stride,
                 unsigned NM)
{
  unsigned s1, s2;
  unsigned p3_plus_1, p2,p1_minus_1, carry;

  if(stride < N)
    {
      if(theta+stride < N)
        return 1;
      p3_plus_1 = (NM-theta)/stride;
      if(p3_plus_1 &&
         ((theta + p3_plus_1 * stride) % NM < N))
        return p3_plus_1;
      return p3_plus_1+1;
    }

  if( (s1 = NM % stride) <= theta)
    return NM/stride;
  if( s1 < N )
    {
      p2 = (stride-N+theta)/s1 + 1;
    }
  else
    {
      s2 = stride % s1;
      p3_plus_1 = NextHit(theta,s2,s1);
      p2 = (p3_plus_1 * stride + theta)/s1;
    }

  carry = 1;
  if((p2 * NM) % stride <= stride-N+theta)
    {
      carry = 0;
    }

  p1_minus_1 = (p2 * NM)/stride;
  return p1_minus_1 + carry;
}
```

The recursive nature of the algorithm is not a problem for hardware implementation since the algorithm terminates at the second level for most inputs that correspond to reasonable values of N and M for memory systems. The recursion can be unravelled by inlining the algorithm once at the only recursive call site. This algorithm is not suitable for a fast hardware implementation because it contains several division and modulo operations by numbers which may not be powers of 2.

Having demonstrated that a straight forward analytical solution for FirstHit() cannot result in a low latency implementation for cache-line interleaved memory, we need to look at methods of transforming the problem to one that can result in a fast hardware implementation at a reasonable cost.

4.1.3 A Simplified Approach using Word Interleaving

It is possible to convert all possible cases of FirstHit() to the simple case 1 by changing the way we view memory. For the sake of generality, let us extend our memory interleave scheme so that each bank is W machine words wide. Assume that we have M banks each containing blocks of size $W * N$ words. Figures 4.1.3 and 5 show the physical view and logical view of a memory system with N=2, W=4 and M=2.

In the physical view we consider the system as a two bank memory system with two memory-words (a memory-word being four machine words) per block. In the logical view we can think of the same two bank system as consisting of 16 logical banks L0-L15. Each logical bank has W=1 and N=1. In general a WxNxM memory may be considered to be WxNxM logical banks denoted by L_0 to L_{WNM-1}.

Assuming that each logical bank has its own FirstHit logic, all possible vector accesses may be handled using case 1 alone. This happens because $\triangle\theta = (V.S \, mod \, NM) \; mod \, N = (V.S \, mod \, M) \, mod \, 1 = 0$ when N=1. The cost of the transformation is that we now need WNM copies of the FirstHit logic where we initially needed only M. This does not directly result in the hardware cost bloating by a factor of WN because of our empirical observation that the combined WNM copies of the logic reduce in size by a factor more than N when optimized. Although we need WN copies of some of the datapath elements of the bank controller, the datapath itself is much simpler than the one required for the algorithm from section 4.1.2. The increased hardware cost is the price paid for making the problem solvable.

In the explanations that follow we will consider only word interleaved memory organizations (i.e. W=N=1) since all other organizations may be converted to a logical equivalent with W and N equal to one. Memory systems with W or N greater than one are equivalent for the purpose of this discussion to a memory system and W=N=1 and WN banks sharing a common bus.

4.1.4 Improved Algorithms for FirstHit() and NextHit()

In section 4.1.3 we showed that it is possible to predict the bank hit sequence for a cache-line interleaved memeory system using an equivalent word interleaved memory system. In this section we derive improved algorithms for $FirstHit()$ and $NextHit()$ for word interleaving which can thus be used for cache-line interleaving as well. These algorithms permit us to access base-stride vectors in parallel without sequentially expanding the addresses of the individual vector elements. Since this section deals exclusively with word interleaving, the parameter N=1 is omited in the following discussion.

Definitions:

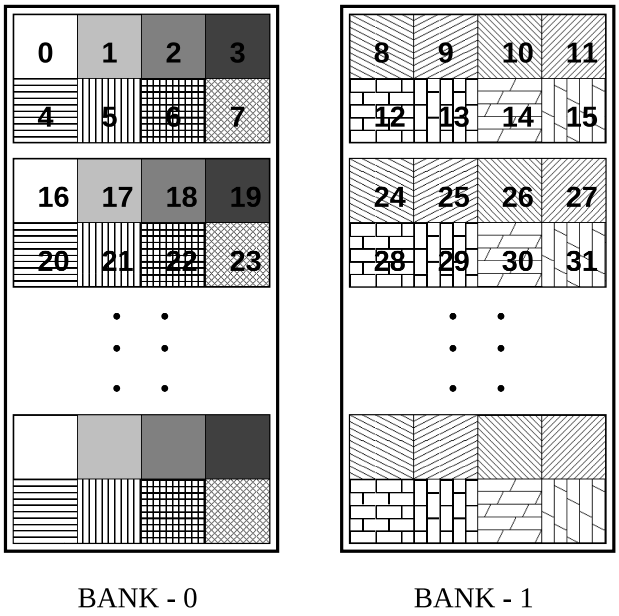


Figure 4: Physical View of Memory

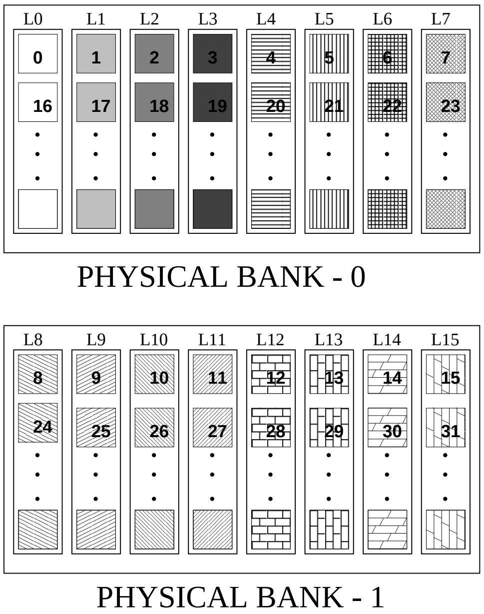


Figure 5: Logical View of Memory

Let $M = 2^m$ denote the number of word-interleaved memory banks.
Let V be a vector and let $b_0 = DecodeBank(V.B)$, i.e., b_0 is the bank where the first element of V resides.
Let d be the distance modulo M between some bank b and b_0, i.e., $d = (b - b_0) \, mod \, M$ as defined in section 4.1.2.

Lemma 4.1 *To find the bank access pattern of a vector V with stride $V.S$, it suffices to consider the bank access pattern for stride $V.S \, mod \, M$.*

Proof: Let $S = q_s M + S_m$, where $S_m = S \, mod \, M$ 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 $(n * S) \, mod \, M = d$. Therefore, for some integer q_d :
$n * S = q_d * M + d$
$n * (q_s M + S_m) = q_d + d$
$n * S_m = (q_d - n * q_s) M + d$
Therefore $(n * S_m) \, mod \, M = d$.
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 $V.S_1 = (V.S \, mod \, M)$, will also hit b for $V_1[n]$.

Explanation: In effect lemma 4.1 says that only the least significant m bits of the stride $(V.S)$ are required to find the bank access pattern of V. This is because if element $V[n]$ of vector V with stride $V.S$ hits bank b, then element $V_1[n]$ of vector V_1 with stride $V_1.S_1 = (V.S \, mod \, M)$ 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.

e.g. $S = 6 = 3 * 2^1$, $S = 7 = 7 * 2^0$, $S = 1 * 2^3$

Lemma 4.2 *Vector V hits on 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:
$n * S = q * M + d$
$n * \sigma * 2^s = q * M + d = q * 2^m + d$
$d = n * \sigma * 2^s - q * 2^m = 2^s(n * \sigma - q 2^{m-s})$
Thus, if some element n of vector V hits on bank b, then d is a multiple of 2^s.

Explanation: In effect lemma 4.2 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 guarentees that there will be a hit.

e.g. 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 4.1 and 4.2 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 4.3 $FirstHit(V, b) = K_i = (K_1 * i)\, mod\, 2^{m-s}$.

Proof: By induction.

Basis: $K_1 = K_1\, mod\, 2^{m-s}$. Note that this is equivalent to proving that $K_1 < 2^{m-s}$. By lemma 4.2, the vector will hit banks at modulo distance 0, 2^s, $2 * 2^s$, $3 * 2^s$ etc from bank b_0. Every bank that is 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, $K_i = (K_1 * i) \, mod \, 2^{m-s} \; \forall i > 0$.

Proof: That the least integer q_{r+1} such that σ evenly divides $q_{r+1} * 2^{m-s} + (r+1)$ is $(q_r + q_1)$.
If possible let there be 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 $x - y = (q_l - q_1) * 2^{m-s} + r$. But since we earlier said that $q_l < q_r + q_1$, we have found a new number $q_{r1} = q_l - q_1$ which 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 4.4 $NextHit(S) = \delta = 2^{m-s}$.

Proof: Let the bank at distance $d = i * 2^s$ have a hit.
Then: $K_i * S = q_i * M + d$.
Since there is a hit at vector index $K_i + \delta$ on the same bank, we have:
$(K_i + \delta) * S = q_j * M + d$ for some integer q_j.
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. The only way σ can divide a multiple of a power of two is if $(q_j - q_i) = \sigma$. Therefore, $\delta = 2^{m-s}$.

4.2 Implementation Strategies for FirstHit() and NextHit()

Using theorems 4.3 and 4.4, each bank controller can independently determine the sub-vector elements for which it is responsible given b, M, $V.S$ mod M, and $V.B$ mod M as inputs. Several options exist for implementing $FirstHit()$ in hardware; which one makes the 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, $V.S \, mod \, M$, and $V.B \, mod \, M$. If M is sufficiently small, an efficient PLA (programmable logic array) implementation could take $d = (b - b_0) \, mod \, S$ 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 off hardware space for latency and 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 look-up tables.

Given appropriate hardware implementations of $FirstHit()$ and $NextHit()$, the bank controller 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 $(K_1 * (d \gg s))$. 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.

4.3 Some Practical Issues

4.3.1 Scaling the Memory System

Scaling Memory System Capacity To scale the vector memory system M and N need to be kept fixed while adding DRAM chips to extend the physical address space. This may be done in several ways. One method would be to have a bank controller for each slot where memory can be added. All the bank controllers corresponding to the same physical bank number would operate in parallel and would be identical. Simple address decoding logic may be used along with the address generated by the bank controller to enable the memory's chip select signal only if the address issued belongs on a particular memory. Another method would be to use a single bank controller for multiple slots, but to maintain different current row registers in order to keep track of the current row inside the different chips which form a single physical bank.

Scaling the Number of Banks The ability to scale the PVA unit to a large number of banks depends on the implementation choice of $FirstHit$. For systems that use a PLA to compute the firsthit index, the complexity of the PLA grows as the square of the number of banks, which 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 bank controller is optimal. For very large memory systems, regardless of their interleave factor, it is best to implement a PLA that generates K_1, adding 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, the rest of the hardware remains unchanged.

4.3.2 Interaction with the Paging Scheme

The opportunity to do parallel fetches for long vectors is present only when a significant part of the vector is continuous in physical memory. Performance will be optimal if each large data-structure that is frequently accessed fits entirely in one super-page. In that case the memory controller can issue longer vector operations on the vector bus. If the data-structure cannot be contained in a single superpage then the memory controller can split a single vector operation into multiple vector operations such that each sub-vector is contained on a single superpage. Given a vector V, one way of doing this would be to find the distance of V.B from the page boundary and divide it by the stride to find how many elements lie on the page and then issue a single vector bus operation for those many elements. However, this operation involves a division and is expensive. A more reasonable approach is to compute a lower bound on the number of vector elements that lie on a page and issue a vector bus operation for those many elements. An algorithm that does this is given below. It assumes that the memory controller has access to the page table and the function mmc_tlb_lookup(vaddress) returns the physical address corresponding to virtual address vaddress and the size of the superpage it is contained in. It is assumed that the size of a superpage is always a power of 2.

```
SplitVector(V)
{
 shift_val = index of most significant power
             of 2 in V.S;
 base = V.B;
 length = V.L;

 while(length > 0)
   {
     (phys_address,page_size) = mmc_tlb_lookup(base);
     lower_bound =
                  (page_size - terminate(phys_address)
                   + 1) >> shift_val;
     // terminate(phys_address) returns least
     // significant n bits of
     // phys_address where page_size == 2^n

     issue on vector bus <base, V.S, lowerbound>

     // While banks are busy operating on the
     // vector we issued compute new base address
     length = length - lowerbound;
     base = base + V.S * lowerbound;
   }
}
```

The operation $lower\_bound = (page\_size - terminate(phys\_address) + 1) \gg shift\_val$ actually just inverts the least significant n bits (assuming page_size is 2^n) of phys_address, adds one to it and shifts the value. In effect this algorithm replaces the division in the exact approach with a fast operation, issues the vector operation and then does a multiply, TLB lookup, etc. while the memory is busy executing the previously issued vector operation. The effectiveness of parallel vector access will depend greatly on how effectively the system is able to create and manage super-pages.

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.

We produced an initial FPGA implementation on an IKOS Hermes emulator with 64 Xi-4000 FPGAs, and then used this implementation to derive timing estimates [12]. During the later stages only software simulation of the Verilog model was done. The software simulation used the latencies derived from the synthesized version tested on the hardware. During the later stages, the full design was not emulated because of the inordinate amount of time and effort required to push the design through the whole toolpath before it can be mapped on to the hardware emulator and also because some of the tools turned out to have bugs. The PVA unit's Verilog description consists of about 3600 lines of code. Details of the hardware complexity may be found in section 1

5.1 Parameters of the Prototype Implementation

The prototype implementation of the PVA is designed to be incorporated into an adaptable memory controller [ref:Impulse] 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 [23]. Our PVA unit design assumes an L2 cache line of 128 bytes, and therefore operates on vector commands of 32 single-word elements. We first describe the implementation of the Vector Bus (shown earlier in figure 1) and the BCs, and then show how the controllers work in tandem.

5.2 Implementation Architecture

5.2.1 Vector Bus

As illustrated in Figure 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 three-bit transaction ID, a two-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 (BC) is responsible for identifying and accessing the (possibly null) subvector that resides in its bank. The architecture of this component, shown in figure 6, 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 (i) a *Read Staging Unit* to store read-data waiting to be assembled into a cache line, and (ii) 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 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.

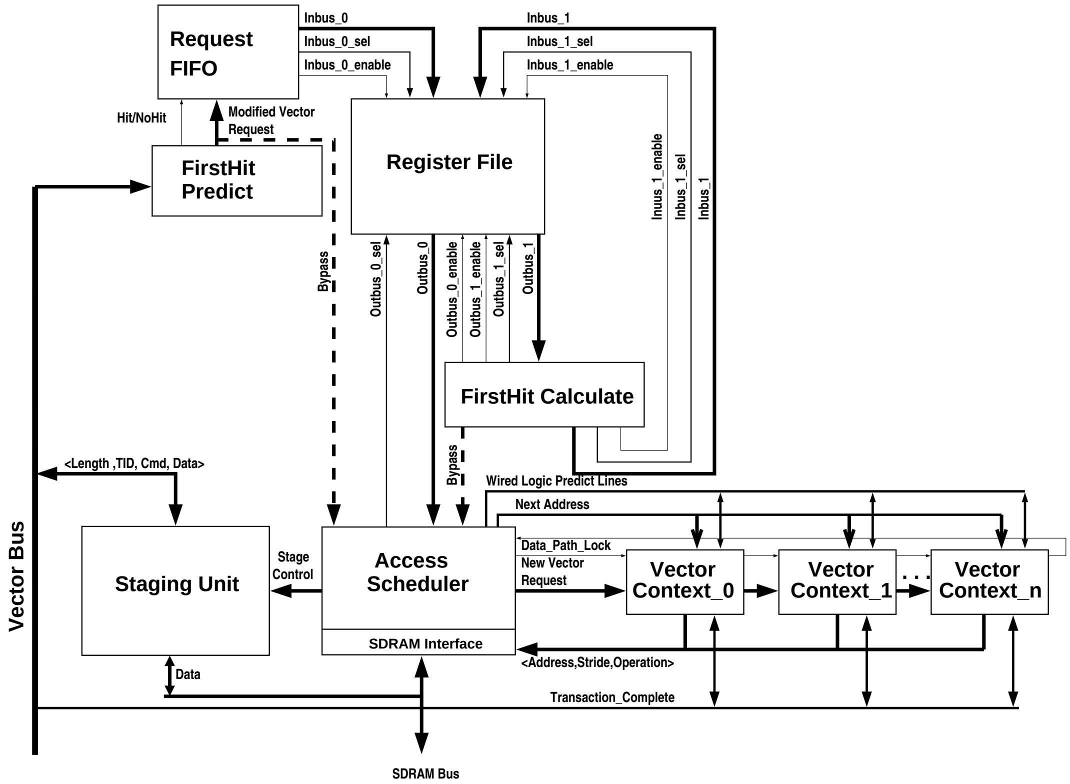


Figure 6: Bank controller internal organization

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. 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.

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 datapath 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 datapath 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 datapath 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 sub-modules 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 know 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
}
```

Row Management Agorithm To obtain better performance this heuristic has to be combined with intelligent management of open rows. If we believe that the next access will be to another row, then closing the row immediately after it is accessed (by using an autoprecharge 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
}
```

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 deasserts 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 2. 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 1 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 have to write the value back to the register file before the request becomes visible to the access scheduler. As explained before 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 datapath can be acquired. Electrical considerations require one-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 folowing 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 deasserts the transaction line for t. When all staging units deassert 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 commited to SDRAM the transcation line for t will be deasserted.

5.3 Hardware Complexity

The results of synthesizing our unoptimized hardware prototype for the IKOS library for Xilinx FPGAs are shown in table 1 [11]. 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 a 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.

6 Performance Evaluation

To evaluate the performance of the hardware prototype of the PVA unit described in 5, we benchmarked it against three other memory systems using several vector style kernels. 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.

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.

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.

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,

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

Table 1: Synthesis summary

and thus DRAM pages are left open during the processing of each command. 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 only accesses the desired vector elements, its relative performance increases dramatically as vector stride goes up.

Parallel Vector Access SRAM: This memory system appearsunder 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 2 lists the kernels used to generate the results presented here. *copy*, *saxpy* and *scale* are from the BLAS (Basic Linear Algebra Subprograms) [7], and tridiag is a tridiagonal gaussian elimination fragment, the fifth Livermore Loop [22]. *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 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 the memory system can be submitted to. Speed up experienced by vector applications will be subject to several criteria like the percentage of vectoriseable memory accesses, the issue width of the processor, number of outstanding L2 cache misses permitted etc. 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 and 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 7 and 8 show the comparative performance for our four memory models on strides 1, 2, 4, 8, 16, and 19 for each of the kernels. Figures 9 and 10 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

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];

Table 2: Kernels used to evaluate our design

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 is because of three reasons :

1. Fewer accesses to SDRAM since the memory controller loads or stores individual words lines rather than whole cache lines.
2. Better SDRAM bandwidth by operating multiple SDRAM banks in parallel.
3. Lower latency by smart scheduling policy for SDRAM banks.
4. Better utilization of bus bandwidth by compacting vector elements into cache-lines.

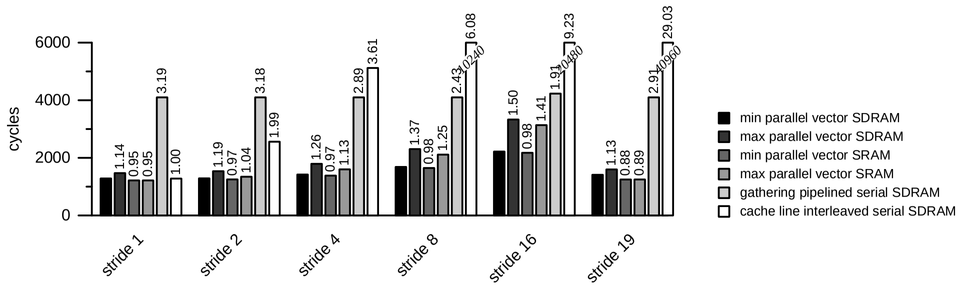
For unit-stride access patterns (dense vectors or cache-line fills), our PVA unit performs about the same as a cache-line interleaved system that performs only line fills. As shown in figure 9 (a), normalized execution time for the latter system is between 100% (for *copy* and *scale)* and 109% (for *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 the 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.

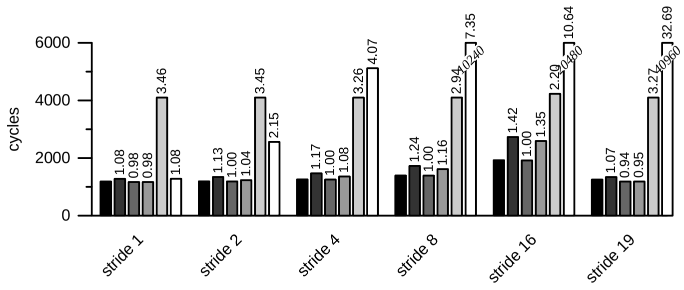
It is not possible to isolate the effect of each of 1, 2 and 4 because the amount of parallelism changes with the stride. So it is not possible to vary the stride and the degree of parallelism independently.

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 7 (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 till 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. Performances 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.

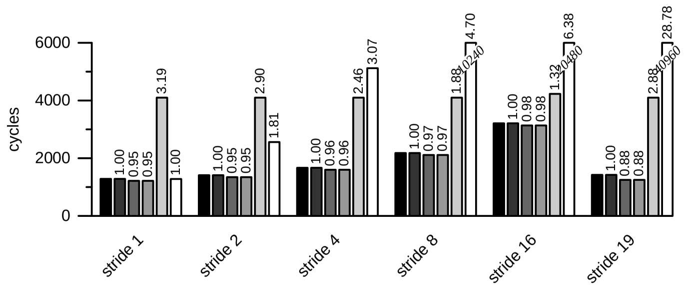
Some relative vector alignments are more advantageous than others, as evidenced by the variations in the SDRAM PVA performance in figure 11 (a). The SRAM version of the PVA system in figure 11 (b) shows similar trends for the various combinations of vector



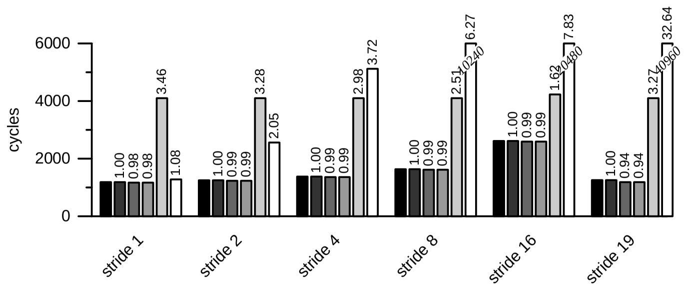
(a) Copy



(b) Copy2

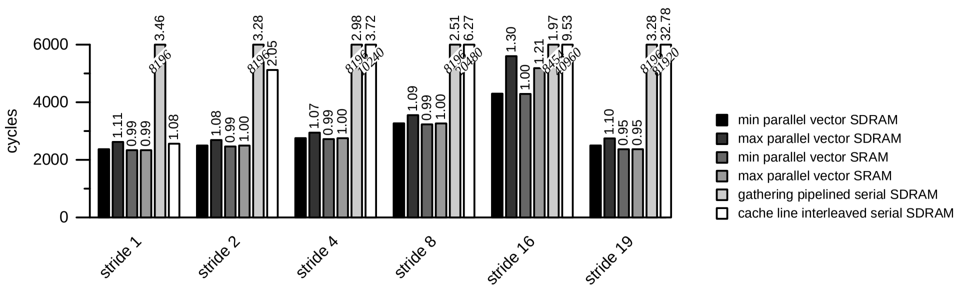


(c) Scale

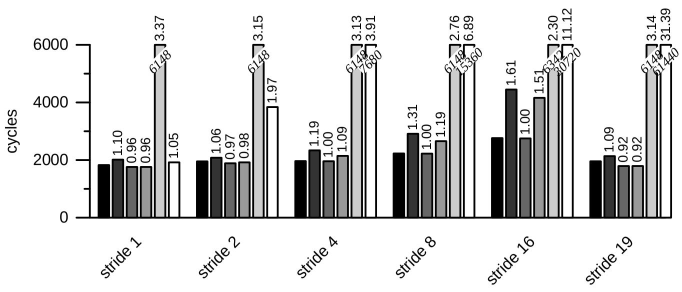


(d) Scale2

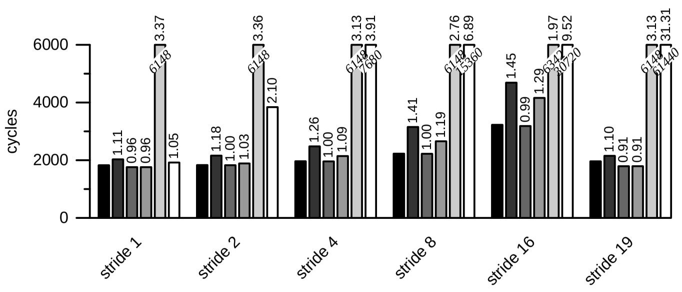
Figure 7: Comparative performance with varying stride



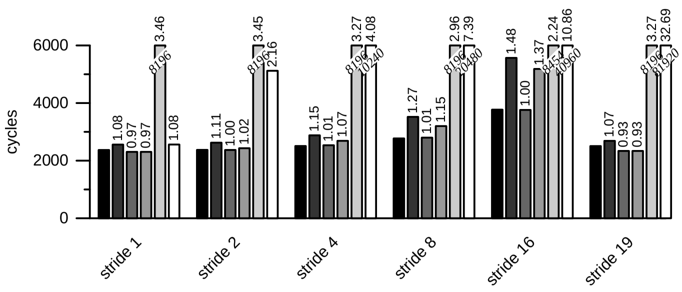
(a) Swap



(b) Tridiag

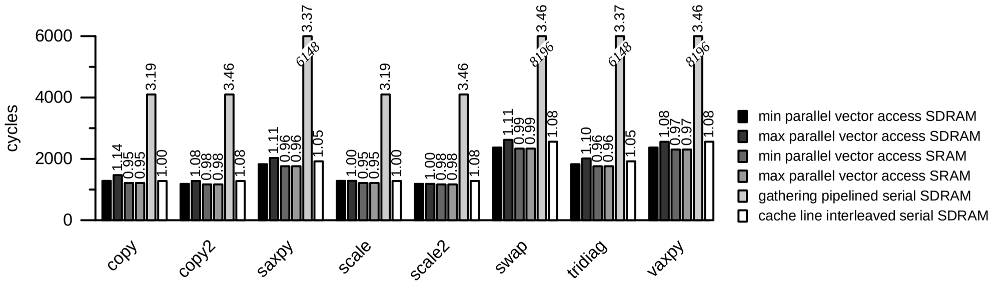


(c) Saxpy

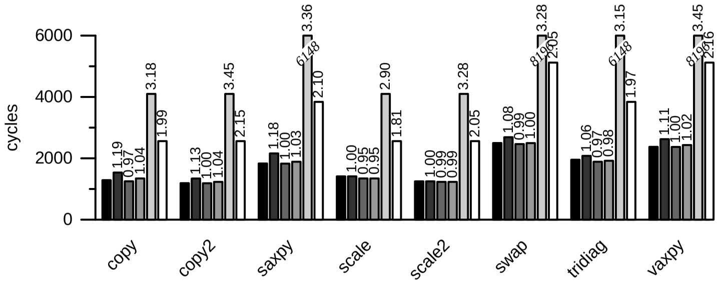


(d) Vaxpy

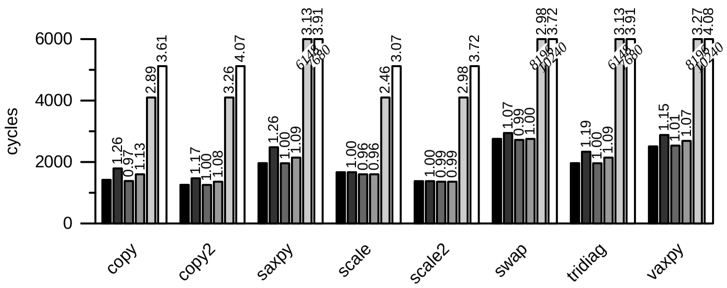
Figure 8: Comparative performance with varying stride - continued



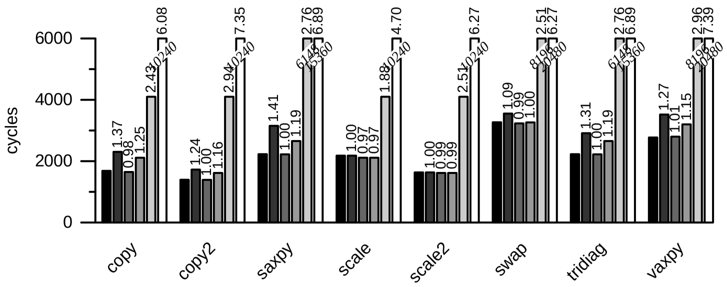
(a) Stride 1



(b) Stride 2

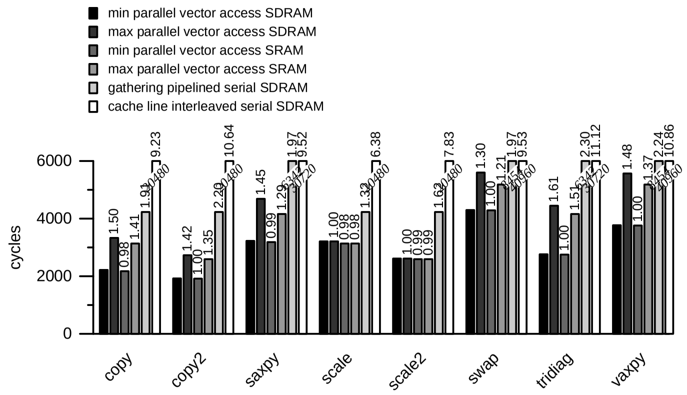


(c) Stride 4

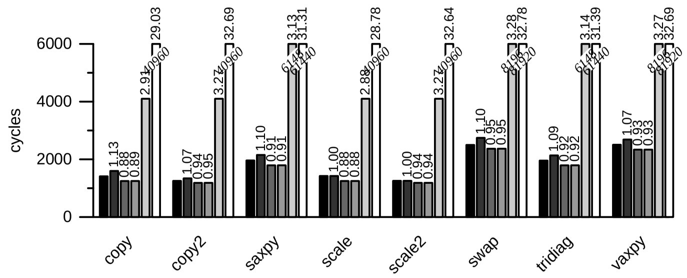


(d) Stride 8

Figure 9: Comparative performance of all kernels with fixed stride

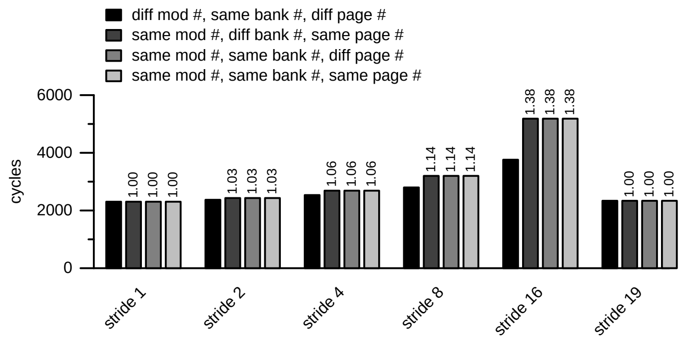


(a) Stride 16

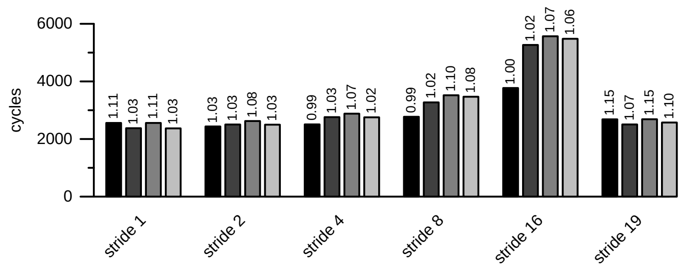


(b) Stride 19

Figure 10: Comparative performance of all kernels with fixed stride - continued



(a) SRAM PVA



(b) SDRAM PVA

Figure 11: Details of the vaxpy kernel performance on the PVA and a similar PVA SRAM system. Bars of graph (a) are annotated with normalized execution time with respect to the leftmost bar, and those of (b) with respect to the corresponding bar from (a).

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 11 (b) is that the PVA SDRAM unit is able to perform remarkably close to PVA SRAM. In that figure, it may be seen that 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. It may be seen in 11 (b) 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 will come close to the performance of SRAM, but will not outperform it.

7 Conclusion

An algorithm to implement parallel access to base-stride vectors was designed and a hardware prototype which implements the algorithm was designed and simulated. The prototype demonstrated the feasibility of implementing the PVA algorithm in hardware and the benchmarks indicate significant performance improvements 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 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 the 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 pre-dates 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 knowledge of common application vectors into a memory controller (e.g. a bit-reversed application vector for DSP environments and multi-media), it would be important to study how knowledge of application vectors can be programmed at run time into memory controllers so that they can use this apriori information to keep pace with the processor in spite of the vast speed difference between processor and memory. 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 and 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] 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.

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

[5] 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.

[6] 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).

[7] 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.

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

[9] 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.

[10] 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.

[11] IKOS Systems. IKOS Libraries, 1999. http://www.ikos.com/products/libraries/index.html.

[12] IKOS Systems. VirtualLogic, 1999. http://www.ikos.com/products/vsli/index.html.

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

[14] 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.

[15] 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.

[16] Krishna, C. M., and Shin, K. G. *Real-time Systems.* McGraw-Hill, 1997.

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

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

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

[20] 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.

[21] McKee, S. A., et al. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the 10th ACM International Conference on Supercomputing* (Philadelphia, PA, May 1996).

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

[23] Micron Technology, Inc. 256mb: Sdram. http://www.micron.com/mti/msp/pdf/datasheets/256MSDRAM.pdf.

[24] MIPS Technologies, Inc. MIPS extension for digital media with 3D. http://www.mips.com/Documentation/isa5_tech_brf.pdf.

[25] Mitsubishi Semiconductors. 4194304-bit (262144-Word by 16-bit) CMOS STATIC RAM. http://www.mitsubishi-chips.com/data/datasheets/memory/mempdf/ds/d99019.pdf, 1998.

[26] Mitsubishi Semiconductors. 256M Synchronous DRAM. http://www.mitsubishi-chips.com/data/datasheets/memory/mempdf/ds/a99005.pdf, July 1999.

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

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

[29] 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).

[30] Rambus, Inc. 256/288-Mbit Direct RDRAM (Advance Information. http://www.rambus.com/developer/downloads/rdram256d.pdf, September 1998.

[31] Rambus, Inc. Rambus technology overview. DL-0040-00, 1999. http://www.rambus.com/developer/downloads/TechOV.pdf.

[32] Rambus, Inc. RMC2 Data Sheet (Advance Information. http://www.rambus.com/developer/downloads/rmc2_overview.pdf, August 1999.

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

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

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

[36] 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).

[37] 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.

[38] 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.

[39] Wolfe, M. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, Massachusetts, 1989.

[40] 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.