Design of a Parallel Vector Access Unit for SDRAM Memory Systems

Binu K. Mathew, Sally A. McKee, John B. Carter, Al Davis
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
{mbinu | sam | retrac | ald}@cs.utah.edu

Abstract

We are attacking the memory bottleneck by building a "smart" memory controller that improves effective memory bandwidth, bus utilization, and cache efficiency by letting applications dictate how their data is accessed and cached. This paper describes a Parallel Vector Access unit *(PVA), the vector memory subsystem that efficiently "gathers" sparse, strided data structures in parallel on a multi-bank SDRAM memory. We have validated our PVA design via gate-level simulation, and have evaluated its performance via functional simulation and formal analysis. On unit-stride vectors, PVA performance equals or exceeds that of an SDRAM system optimized for cache line fills. On vectors with larger strides, the PVA is up to 32.8 times faster. Our design is up to 3.3 times faster than a pipelined, serial SDRAM memory system that gathers sparse vector data, and the gathering mechanism is two to five times faster than in other PVAs with similar goals. Our PVA only slightly increases hardware complexity with respect to these other systems, and the scalable design is appropriate for a range of computing platforms, from vector supercomputers to commodity PCs.*

1. Introduction

Processor speeds are increasing much faster than memory speeds, and this disparity prevents many applications from making effective use of the tremendous computing power of modern microprocessors. In the Impulse project, we are attacking the memory bottleneck by designing and building a "smart" memory controller [3]. The Impulse memory system can significantly improve the performance of applications with predictable access patterns but poor spatial or temporal locality [3]. Impulse supports an optional extra address translation stage allowing applications to control how their data is accessed and cached. For instance, on a conventional memory system, traversing rows of a FORTRAN matrix wastes bus bandwidth: the cache line fills transfer unneeded data and evict other useful data. Impulse *gathers* sparse "vector" elements into dense cache lines, much like the scatter/gather operations supported by the load-store units of vector supercomputers.

Several new instruction set extensions (e.g., Intel's MMX for the Pentium [10], AMD's 3DNow! for the K6-2 [1], MIPS's MDMX [18], Sun's VIS for the UltraSPARC [22], and Motorola's AltiVec for the PowerPC [19]) bring stream and vector processing to the domain of desktop computing. Results for some applications that use these extensions are promising [21, 23], even though the extensions do little to address memory system performance. Impulse can boost the benefit of these vector extensions by optimizing the cache and bus utilization of sparse data accesses.

In this paper we describe a vector memory subsystem that implements both conventional cache line fills and vector-style scatter/gather operations efficiently. Our design incorporates three complementary optimizations for non-unit stride vector requests:

1. **We improve memory locality via remapping**. Rather than perform a series of regular cache line fills for non-unit stride vectors, our system gathers only the desired elements into dense cache lines.
2. **We increase throughput with parallelism**. To mitigate the relatively high latency of SDRAM, we operate multiple banks simultaneously, with components working on independent parts of a vector request. Encoding many individual requests in a compound *vector command* enables this parallelism and reduces communication within the memory controller.

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.

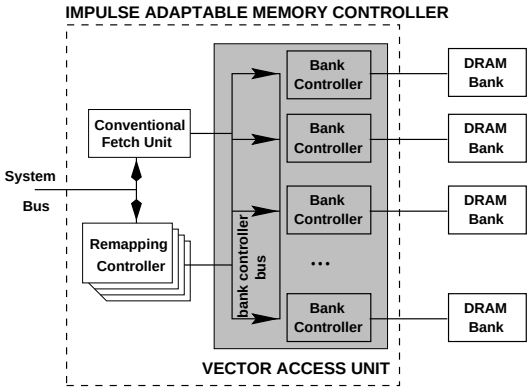


Figure 1. Memory subsystem overview. The configurable remapping controllers broadcast "vector commands" to all bank controllers. The controllers gather data elements from the SDRAMs into staging units, from which the vector is transferred to the CPU chip.

3. **We exploit SDRAM's non-uniform access characteristics**. We minimize observed precharge latencies and row access delays by overlapping these with other memory activity, and by trying to issue vector references in an order that hits the current row buffers.

Figure 1 illustrates how the components of the Impulse memory controller interact. A small set of *remapping controllers* support three types of scatter/gather operations: *base-stride*, *vector indirect*, and *matrix inversion*. Applications configure the memory controller so that requests to certain physical address regions trigger scatter/gather operations (interface and programming details are presented elsewhere [3]). When the processor issues a load falling into such a region, its remapping controller sees the load and broadcasts the appropriate scatter-gather vector command to all bank controller (BC) units via the bank controller bus. In parallel, each BC determines which parts of the command it must perform locally and then "gathers" the corresponding vector elements into a staging unit. When all BCs have fetched their elements, they signal the remapping controller, which constructs the complete vector from the data in the staging units. In this way, a single cache line fill can load data from a set of sparse addresses, e.g., the row elements of a FORTRAN array.

We validate the PVA design via gate-level synthesis and simulation, and have evaluated its performance via functional simulation and formal analysis. For the kernels we study, our PVA-based memory system fills normal (unit stride) cache lines as fast as and up to 8% faster than a conventional, cache-line interleaved memory system optimized for line fills. For larger strides, it loads elements up to 32.8 times faster than the conventional memory system. Our system is up to 3.3 times faster than a pipelined, centralized memory access unit that gathers sparse vector data by issuing (up to) one SDRAM access per cycle. Compared to other parallel vector access units with similar goals [5], gather operations are two to five times faster. This improved parallel access algorithm only modestly increases hardware complexity. By localizing all architectural changes within the memory controller, we require no modifications to the processor, system bus, or on-chip memory hierarchy. This scalable solution is applicable to a range of computing platforms, from vector computers with DRAM memories to commodity personal computers.

2. Related Work

We limit our discussion to work that addresses loading vectors from DRAM. Moyer defines *access scheduling* and *access ordering* to be techniques that reduce load/store interlock delays by overlapping computation with memory latency, and that change the order of memory requests to increase performance, respectively. [20]. Access scheduling attempts to separate the execution of a load/store instruction from that of the instruction that produces/consumes its operand, thereby reducing the processor's observed memory delays. Moyer applies both concepts to compiler algorithms that optimize inner loops, unrolling and grouping stream accesses to amortize the cost of each DRAM page miss over several references to the open page.

Lee mimics Cray instructions on the Intel i860XR using another software approach, treating the cache as a pseudo "vector register" by reading vector elements in blocks (using non-caching loads) and then writing them to a preallocated portion of cache [11]. Loading a single vector via Moyer's and Lee's schemes on an iPSC/860 node improves performance by 40-450%, depending on the stride [15].

Valero et al. dynamically avoid bank conflicts in vector processors by accessing vector elements out of order. They analyze this system first for single vectors [24], and then extend the design to multiple vectors [25]. del Corral and Llaberia analyze a related hardware scheme for avoiding bank conflicts among multiple vectors in complex memories [7]. These schemes focus on vector computers with (uniform access time) SRAM memory components.

The PVA component presented herein is similar to Corbal et al.'s Command Vector Memory System [6] (CVMS), which exploits parallelism and locality of reference to improve effective bandwidth for out-of-order vector processors with dual-banked SDRAM memories. Instead of sending individual requests to individual devices, the CVMS broadcasts commands requesting multiple independent words, a design idea we adopt. Section controllers receive the broadcasts, compute subcommands for the portion

of the data for which they are responsible, and then issue the addresses to SDRAMs under their control. The memory subsystem orders requests to each dual-banked device, attempting to overlap precharges to each internal bank with accesses to the other. Simulation results demonstrate performance improvements of 15-54% over a serial controller. Our bank controllers behaviorally resemble CVMS section controllers, but our hardware design and parallel access algorithm (see Section 4.3) differ substantially.

The Stream Memory Controller (SMC) of McKee *et al.* [14] combines programmable stream buffers and prefetching in a memory controller with intelligent DRAM scheduling. Vector data bypass the cache in this system, but the underlying access-ordering concepts can be adapted to systems that cache vectors. The SMC dynamically reorders stream/vector accesses and issues them serially to exploit: a) parallelism across dual banks of fast-page mode DRAM, and b) locality of reference within DRAM page buffers. For most alignments and strides on uniprocessor systems, simple ordering schemes perform competitively with sophisticated ones [13].

Stream detection is an important design issue for these systems. At one end of the spectrum, the application programmer may be required to identify vectors, as is currently the case in Impulse. Alternatively, the compiler can identify vector accesses and specify them to the memory controller, an approach we are pursuing. For instance, Benitez and Davidson present simple and efficient compiler algorithms (whose complexity is similar to strength reduction) to detect and optimize streams [2]. Vectorizing compilers can also provide the needed vector parameters, and can perform extensive loop restructuring and optimization to maximize vector performance [26]. At the other end of the spectrum lie hardware vector or stream detection schemes, as in reference prediction tables [4]. Any of these suffices to provide the information Impulse needs to generate vector accesses.

3. Mathematical Foundations

The Impulse remapping controller gathers strided data structures by broadcasting vector commands to a set of bank controllers (BCs), each of which determines independently and in tandem with the others which elements of the vector reside in the SDRAM it manages. This broadcast approach is potentially much more efficient than the straightforward alternative of having a centralized vector controller issue the stream of element addresses, one per cycle. Realizing this performance potential requires a method whereby each bank controller can determine the addresses of the elements that reside on its SDRAM without sequentially expanding the entire vector. The primary advantage of our PVA mechanism over similar designs is the efficiency of our hardware algorithms for computing each bank's subvector.

We first introduce the terminology used in describing these algorithms. Base-stride vector operations are represented by a tuple, $V =< B, S, L >$, where $V.B$ is the base address, $V.S$ the sequence stride, and $V.L$ the sequence length. We refer to V's i^{th} element as $V[i]$. For example, $< A, 4, 5 >$ designates vector elements $A[0]$, $A[4]$, $\ldots A[16]$. The number of banks, M, is a power of two. The PVA algorithm is based on two functions:

1. *FirstHit(V,b)* takes a vector V and a bank b and returns either the index of the first element of V that hits in b or a value indicating that no such element exists.
2. *NextHit(S)* returns an increment δ such that if a bank holds $V[n]$, it also holds $V[n+\delta]$.

Space considerations only permit a simplified explanation here. Our technical report contains complete mathematical details [12]. *FirstHit()* and first address calculation together can be evaluated in two cycles for power of two strides and at most five cycles for other strides. Their design is scalable and can be implemented in a variety of ways. This paper henceforth assumes that they are implemented as a programmable logic array (PLA). *NextHit()* is trivial to implement, and takes only a few gate delays to evaluate.

Given inputs b, M, $V.S$ mod M, and $V.B mod M$, each bank controller uses these functions to independently determine the sub-vector elements for which it is responsible. The BC for bank b performs the following operations (concurrently, where possible):

1. calculate *i=FirstHit(V,b)*; if *NoHit*, continue.
2. while i < V.L do
   access memory location V.B + i × V.S
   i += *NextHit(V.S)*

4. Vector Access Unit

The design space for a PVA mechanism is enormous: the type of DRAM, number of banks, interleave factor, and implementation strategy for *FirstHit()* can 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 developed and synthesized a Verilog model of one design point in this large space. The implementation uses 16 banks of word-interleaved SDRAM (32-bit wide). Each has a dedicated bank controller that drives 256 Mbit 16-bit wide Micron SDRAM parts, each of which contains four internal banks [17]. The current PVA 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 bank-controller bus and the BCs, and then show how the controllers work in tandem.

4.1 Bank Controller Bus

As illustrated in Figure 1, the bank controllers communicate with the rest of the memory controller via a shared, split-transaction bus (BC bus) that multiplexes requests and data. During a vector request cycle, each bus supports a 32-bit address, 32-bit stride, three-bit transaction ID, two-bit command, and some control information. During a data cycle, each supports 64 data bits. The current PVA design targets a MIPS R10000 processor with a 64-bit system bus, on which the PVA unit can send or receive one data word per cycle. No intermediate unit is needed to merge data collected by multiple BCs: 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, but to avoid these delay cycles, we use a 128-bit BC bus and drive alternate 64-bit halves every other data cycle. In addition to the 128 multiplexed lines, the BC bus includes eight shared *transaction-complete* indication lines.

4.2 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. Shown in Figure 2, the architecture of this component consists of:

1. a *FirstHit Predictor* to determines whether elements of a given vector request hit this bank. If there is a hit and the stride is a power of two, this subcomponent performs the *FirstHit* address calculation;
2. a *Request FIFO* to queue vector requests for service;
3. a *Register File* to provide storage for the Request FIFO;
4. a *FirstHit Calculate* module to determine the address of the first element hitting this bank when the stride is not a power of two;
5. an *Access Scheduler* to drive 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 current vector requests;
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. Essential to efficient operation are several bypass paths that reduce communication latency within the BC. Our technical report fleshes out details of these modules and their interactions [12]. The main modules of the BC manage the computations required for parallel vector access, the efficient scheduling of SDRAM, and the data staging.

4.2.1 Parallelizing Logic

The parallelizing logic consists of the FirstHit Predict (FHP) module, the Request FIFO (RQF), the Register File (RF), and the FirstHit Calculate (FHC) modules. The FHP module watches vector requests on the BC bus and determines whether or not any element of a request will hit the bank. The FHP calculates the *FirstHit index*, the index of the first vector element in the bank. For power-of-two strides that hit, the FHP also calculates the *FirstHit address*, the bank address of the first element. The FHP then signals the RQF to queue: the request's $V =< B, S, L >$ tuple; the FirstHit index; the calculated bank address, if ready; and an *address calculation complete* (ACC) flag indicating the status of the bank address field.

The RF subcomponent contains as many entries as the number of outstanding transactions permitted by the BC bus, which is eight in this implementation. The RQF module implements the state machine and tail pointer to maintain the RF as a queue, storing vector requests in the RF entries until those requests are assigned to vector contexts. Queued requests with a cleared ACC flag require further processing: the FHC module computes the FirstHit address for these requests, whose stride is not a power of two. The FHC scans the requests between the queue head pointer, which it maintains, and the tail pointer, multiplying the stride S by the FirstHit index calculated by the FHP, and then adding that to the base address B to generate the FirstHit address. The FHC then writes this address into the register file and sets the entry's ACC flag. Since this calculation requires a multiply and add, it incurs a two-cycle delay, but the FHC works in parallel with the Access Scheduler (SCHED), so when the latter module is busy, this delay is completely hidden. When the 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.

4.2.2 Access Scheduler

The SCHED and its subcomponents, the *Vector Contexts* (VCs) and *Scheduling Policy Unit* (SPU) modules, are responsible for: (i) expanding the series of addresses in a vector request, (ii) ordering the stream of reads, writes, bank activates, and precharges so that multiple vector requests can be issued optimally, (iii) making row activate/precharge

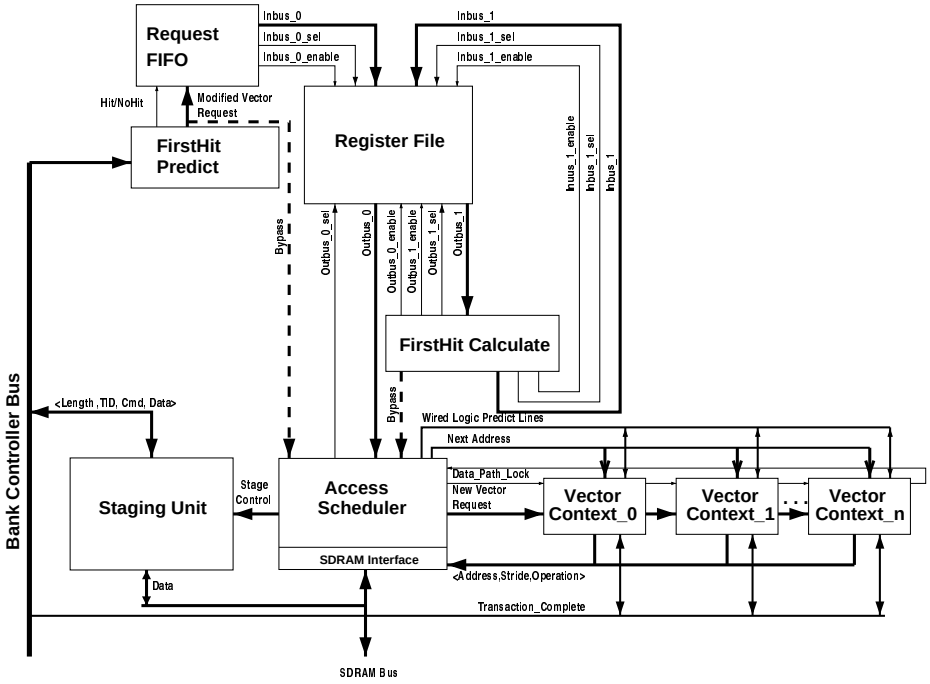


Figure 2. Bank controller internal organization

decisions, and (iv) driving the SDRAM. The SCHED module decides when to keep an SDRAM row open, and the SPUs within the SCHED's VCs reorder the accesses.

The current design contains four VCs, each of which can hold a vector request whose accesses are ready to be issued to the SDRAM. The VC performs a series of shifts and adds to generate the sequence of addresses required to fetch a particular vector. These efficient calculations constitute the crux of our PVA approach, and our technical report explains their details. The VCs share the SCHED datapath, and they cooperate to issue the highest priority pending SDRAM operation required by any VC. The VCs arbitrate for this datapath such that at most one can access it in any cycle, and the oldest pending operation has highest priority. Vector operations are injected into VC_0, and whenever one completes (at most one finishes per cycle), other pending operations "shift right" into the next free VC. To give the oldest pending operation priority, we daisy-chain the SCHED requests from VC_N to VC_0 such that a lower numbered VC can place a request on the shared 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 to a different internal bank on the same SDRAM module. Three lines per internal bank — *bank_hit_predict*, *bank_more_hit_predict*, and *bank_close_predict* — coordinate this operation. The SCHED broadcasts to the VCs the the current row addresses of the open internal banks. When a VC determines that it has a pending request that would hit an open row, it drives the internal bank's shared line to tell the SCHED not to close the row – in other words, we implement a wired OR operation. Likewise, VCs that have a pending request that misses in the internal bank use the *bank_close_predict* line to tell the SCHED to close the row. The 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 the subcomponents, permitting experimentation with the scheduling policy without changing the rest of the BC.

The scheduling algorithm strives to improve performance by maximizing row hits and hiding latencies; it does this by operating other internal banks while a given internal bank is being opened or precharged. We implement a policy that promotes row opens and precharges above read and write operations, as long as the former do not conflict with the open rows in use by another VC. This heuristic opens rows as early as possible. When conflicts or open/precharge latencies prevent higher numbered VCs from issuing a read or write, a lower priority VC may issue its reads or writes. The policy ensures that when an older request completes, a new request will be ready, even if the new one uses a different internal bank. Details of the scheduling algorithm are given in our technical report [12].

4.2.3 Staging Units

The Staging Units (SUs) store data returned by the SDRAMs for a VC-generated read operation or 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 to be sent to the memory controller front end, as described in Section 4.1. In the case of a scattered vector write operation, the SUs at each participating BC buffer the write data sent by the front end.

The SUs drive a *transaction_complete* line on the BC bus to signal the completion of a pending vector operation. This line acts as a wired OR that deasserts whenever all BCs have finished a particular gathered vector read or scattered vector write operation. When the line goes low during a read, the memory controller issues a STAGE_READ command on the vector bus, indicating which pending vector read operation's data is to be read. When the line goes low during a write, the memory controller knows that the corresponding data has been committed to SDRAM.

4.2.4 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 two important consistency semantics. First, RAW hazards cannot happen. Second, 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.

4.3 Timing Considerations

SDRAMs define timing restrictions on the sequence of operations that can legally be performed. To maintain these 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 is permitted. 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 the restimers and the datapath — can be acquired. Electrical considerations require a 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. The SCHED units attempt to minimize turnaround cycles by reordering accesses.

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 1. Inner loops used to evaluate our PVA unit design.

5. Experimental Methodology

This section describes the details and rationale of how we evaluate the PVA design. The initial prototype uses a word-interleaved organization, since block-interleaving complicates address arithmetic and increases the hardware complexity of the memory controller. Our design can be extended for block-interleaved memories, but we have yet to perform price/performance analyses of this design space. Note that Hsu and Smith study interleaving schemes for fast-page mode DRAM memories in vector machines [9], finding cache-line interleaving and block interleaving superior to low-order interleaving for many vector applications. The systems they examine perform no dynamic access ordering to increase locality, though, and their results thus favor organizations that increase spatial locality within the DRAM page buffers. It remains to be seen whether low-order interleaving becomes more attractive in conjunction with access ordering and scheduling techniques, but our initial results are encouraging.

Table 1 lists the kernels used to generate the results presented here. `copy`, `saxpy` and `scale` are from the BLAS (Basic Linear Algebra Subprograms) benchmark suite [8], and `tridiag` is a tridiagonal gaussian elimination fragment, the fifth Livermore Loop [16]. `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

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 2. Complexity of the synthesized bank controller.

estimates. Performance on larger, real-world benchmarks — via functional simulation of the whole Impulse 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, but these results are not yet available.

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 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. We present highlights of these results in the following section; details may be found in our technical report [12].

6. Results

This section presents timing and complexity results from synthesizing the PVA and comparative performance results for our suite of benchmark kernels.

6.1 Synthesis Results

Our end goal is to fabricate a CMOS ASIC of the Impulse memory controller, but we are first validating pieces of the larger design using FPGA (field programmable gate array) technology. We produce an FPGA implementation on an IKOS Hermes emulator with 64 Xi-4000 FPGAs, and then use this implementation to derive timing estimates. The PVA's Verilog description consists of 3600 lines of code. The types and numbers of components in the synthesized bank controller are given in Table 2. We expect that the custom CMOS implementation to be much more efficient than the FPGA implementation.

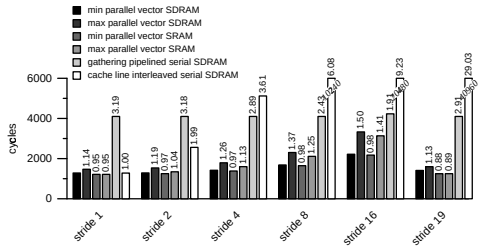
We used the synthesized design to measure delay through the critical path — the multiply-and-add circuit required to calculate *FirstHit()* for non-power-of-two strides. Our multiply-and-add unit has a delay of 29.5ns. We expect that an optimized CMOS implementation will have a delay less than 20ns, making it possible to complete this operation in two cycles at 100MHz. Other paths are fast enough to operate at 100MHz even in the FPGA implementation. The FHP unit has a delay of 8.3ns and SCHED has a delay of 9.3ns. CMOS timing considerations are very different from those for FPGAs, and thus the optimization strategies differ significantly. These FPGA delays represent an upper bound — the custom CMOS version will be much faster.

6.2 Performance Results

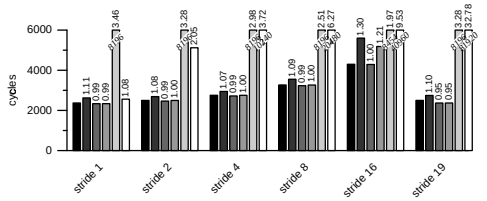
We compare the performance of the PVA functional model to three other memory systems. Figure 3(a)-(c) show the comparative performance for our four memory models on strides 1, 2, 4, 8, 16, and 19 for the `copy`, `swap`, and `vaxpy` kernels, and Figure 3(d)-(f) show comparative performance across all benchmarks for strides 1, 4, and 16. 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. The sets of bars labeled "copy2" and "scale2" represent unrolled kernels in which read and write vector commands are grouped (so the PVA sees two consecutive vector commands for the first vector, then two for the second, and so on). This optimization only improves performance for the PVA SDRAM systems, yielding 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 improvements.

The bars labeled "cache line interleaved serial SDRAM" model the back end of 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.

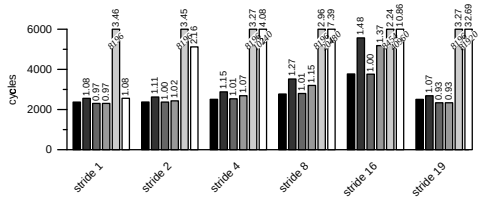
The bars labeled "gathering pipelined serial SDRAM" model the back end of a 16-module, word-interleaved



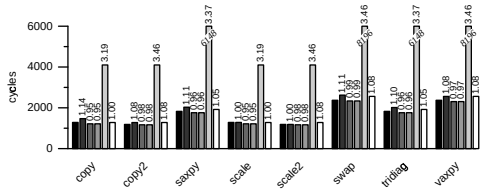
(a) *copy*



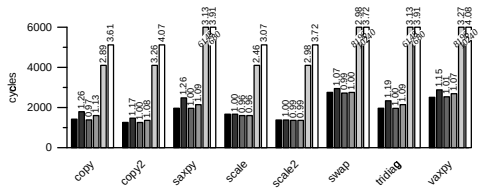
(b) *swap*



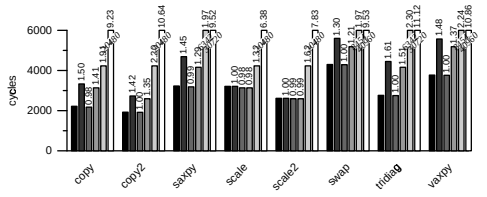
(c) *vaxpy*



(d) *stride 1*



(e) *stride 4*



(f) *stride 16*

Figure 3. Comparative performance for four memory back ends.

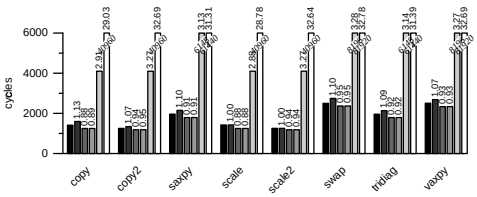
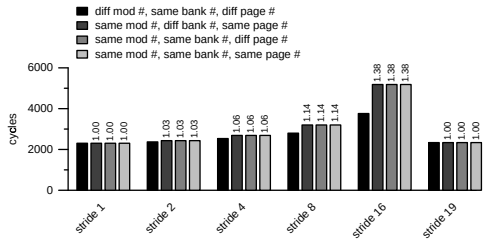


Figure 4. Comparative performance for a prime stride (19).

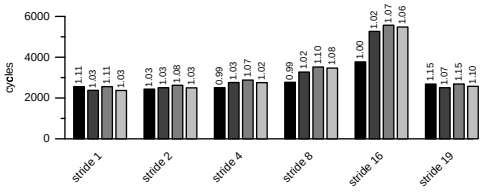
SDRAM system with a closed-page policy and pipelined precharge. As before, the memory bus is 64 bits, and vector commands access 32 elements (128 bytes, since the present system uses 4-byte elements). Instead of performing cache line fills, this system accesses each vector element individually. Although accesses are issued serially, we assume that the memory controller can overlap RAS latencies with activity on other banks for all but the first element accessed by each command. We optimistically assume that vector commands never cross DRAM pages, and thus DRAM pages are left open during the processing of each command. 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.

The bars labeled “min parallel vector access SRAM” and “max parallel vector access SRAM” model the performance of an idealized SRAM vector memory system with the same parallel access scheme but with no precharge or RAS latencies. Comparing PVA SDRAM and PVA SRAM system performances gives a measure of how well our system hides the extra latencies associated with dynamic RAM.

For unit-stride access patterns (dense vectors or cache-line fills), the PVA performs about the same as a cache-line interleaved system that performs only line fills. As shown in Figure 3, normalized execution time for the latter system is between 100% (for `copy` and `scale`) and 108% (for `copy2`, and `scale2`, `vaxpy`, `swap`) of the PVA’s minimum execution time for our kernels. 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 reaches 1112% (for `tridiag`). Figure 3(a), (b), and (c) demonstrate that performance shows similar trends for each benchmark kernel. Figure 3(d), (e), and (f) show performance trends for a given vector stride. Figure 4 shows performance results for vectors with large strides that still hit all the memory banks. Performances for both our SDRAM PVA system and the SRAM PVA sys-



(a) *SRAM PVA*



(b) *SDRAM PVA*

Figure 5. 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).

tem 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 5(b). The SRAM version of the PVA system in Figure 5(a) shows similar trends for the various combinations of vector stride and relative alignments, although its performance is slightly more robust. For small strides that hit more than two SDRAM banks, the minimum and maximum execution times for the 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.

The results highlighted here are representative of those for all our experiments [12]. On dense data, the SDRAM PVA performs like an SDRAM system optimized for cache-line fills. In general, it performs much like an SRAM vector memory system at a fraction of the cost.

7. Discussion

In this paper, we have described the design of a Parallel Vector Access unit (PVA) for the Impulse smart memory controller. The PVA employs a novel parallel access algorithm that allows a collection of bank controllers to determine in tandem which parts of a vector command are located on their SDRAMs. The BCs optimize low-level access to their SDRAMs to maximize the frequency of open-row hits and overlap accesses to independent banks as much as possible. As a result, the Impulse memory controller always performs no worse than 1% slower (and up to 8% faster) than a memory system optimized for normal cache line fills on unit-stride accesses. For vector-style accesses, the PVA delivers data up to 32.8 times faster than a conventional memory controller and up to 3.3 times faster than alternative vector access units, for a modest increase in hardware complexity. We are integrating the PVA into the full Impulse simulation environment, so that we can evaluate the performance improvements across whole applications.

Space limitations prevent us from fully addressing a number of important features of the PVA, including scalability, interoperability with virtual memory, and techniques for optimizing other kinds of scatter-gather operations. Ultimately, the scalability of our memory system depends on the implementation choice of *FirstHit()*. For systems that use a PLA to compute the firsthit index, the complexity of the PLA grows with 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 BC is optimal. For very large memory systems, regardless of their interleave factor, it is best to implement a PLA to calculate the successive vector indices within a bank. The complexity of this PLA increases approximately linearly with the number of banks, the rest of the hardware remains unchanged, and the performance is constant, irrespective of the number of banks.

Another design issue is how to handle “contiguous” data spread across disjoint physical pages. If strided vectors span multiple pages, additional address translation logic is required in the BCs. In the current evaluation, we assume the data being gathered into each dense cache line falls within a single page or superpage of physical memory. Working around the limitations of paged virtual memory is discussed in our technical report [12].

Finally, the PVA described here can be extended to handle vector-indirect scatter-gather operations by performing the gather in two phases: (i) loading the indirection vector into the BCs and then (ii) loading the vector elements. The first phase is simply a unit-stride vector load operation. After the indirection vector is loaded, its contents can be broadcast across the BC bus. Each BC determines which elements reside in its SDRAM by snooping this broadcast and performing a simple bit-mask operation on each address. Then each BC performs its part of the gather in parallel, and the result are coalesced from the staging units in much the same way as for strided accesses,

In summary, we have presented the design of a Parallel Vector Access unit that shows great promise for providing applications with poor locality with vector-machine-like memory performance. Although much work remains to be done, our experience to date indicates that such a system can significantly reduce the memory bottleneck for the kinds of applications that suffer on conventional memory systems. The next steps are to evaluate the PVA design on a suite of whole-program benchmarks and to address the issues raised above, particularly the interaction with virtual memory and supporting other scatter-gather operations.

8. Acknowledgments

Discussions with Mike Parker and Lambert Schaelicke on aspects of the PVA design and its evaluation proved invaluable. Ganesh Gopalakrishnan helped with the IKOS equipment and the Verilog model. Wilson Hsieh, Lixin Zhang, and the other members of the Impulse and Avalanche projects helped shape this work, and Gordon Kindlmann helped us with the figures.

References

[1] Advanced Micro Devices. Inside 3DNow!(tm) technology. http://www.amd.com/products/cpg/k623d/inside3d.html.

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

[3] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, , and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.

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

[5] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 68–77, Oct. 1998.

[6] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. Technical Report UPC-DAC-1999-5, Universitat Politecnica de Catalunya, Jan. 1999.

[7] A. del Corral and J. Llaberia. Access order to avoid inter-vector conflicts in complex memory systems. In *Proceedings of the Ninth International Parallel Processing Symposium*, 1995.

[8] J. Dongarra, J. DuCroz, I. Duff, and S. Hammerling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, Mar. 1990.

[9] W. Hsu and J. Smith. Performance of cached DRAM organizations in vector supercomputers. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 327–336, May 1993.

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

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

[12] B. Mathew, S. McKee, J. Carter, and A. Davis. Parallel access ordering for SDRAM memories. Technical Report UUCS-99-006, University of Utah Department of Computer Science, June 1999.

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

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

[15] S. McKee and W. Wulf. Access ordering and memory-conscious cache utilization. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 253–262, Jan. 1995.

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

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

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

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

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

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

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

[23] J. Tyler, J. Lent, A. Mather, and H. Nguyen. Altivec: Bringing vector technology to the powerpc processor family. In *Proceedings of the 1999 IEEE International Performance, Computing, and Communications Conference*, Feb. 1999.

[24] M. Valero, T. Lang, J. Llaberia, M. Peiron, E. Ayguade, and J. Navarro. Increasing the number of strides for conflict-free vector access. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 372–381, May 1992.

[25] M. Valero, T. Lang, M. Peiron, and E. Ayguade. Conflict-free access for streams in multi-module memories. Technical Report UPC-DAC-93-11, Universitat Politecnica de Catalunya, Barcelona, Spain, 1993.

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