Appears in the Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37), December 2004

Cache Refill/Access Decoupling for Vector Machines

Christopher Batten, Ronny Krashinsky, Steve Gerding, and Krste Asanović
MIT Computer Science and Artificial Intelligence Laboratory*
The Stata Center, 32 Vassar Street, Cambridge, MA 02139
{cbatten|ronny|sgerding|krste}@csail.mit.edu

Abstract

Vector processors often use a cache to exploit temporal locality and reduce memory bandwidth demands, but then require expensive logic to track large numbers of outstanding cache misses to sustain peak bandwidth from memory. We present refill/access decoupling, which augments the vector processor with a Vector Refill Unit (VRU) to quickly pre-execute vector memory commands and issue any needed cache line refills ahead of regular execution. The VRU reduces costs by eliminating much of the outstanding miss state required in traditional vector architectures and by using the cache itself as a cost-effective prefetch buffer. We also introduce vector segment accesses, a new class of vector memory instructions that efficiently encode two-dimensional access patterns. Segments reduce address bandwidth demands and enable more efficient refill/access decoupling by increasing the information contained in each vector memory command. Our results show that refill/access decoupling is able to achieve better performance with less resources than more traditional decoupling methods. Even with a small cache and memory latencies as long as 800 cycles, refill/access decoupling can sustain several kilobytes of in-flight data with minimal access management state and no need for expensive reserved element buffering.

1. Introduction

Abundant data parallelism is found in a wide range of important compute-intensive applications, from scientific supercomputing to mobile media processing. Vector architectures [23] exploit this data parallelism by arraying multiple clusters of functional units and register files to provide huge computational throughput at low power and low cost. Memory bandwidth is much more expensive to provide, and consequently often limits application performance.

Earlier vector supercomputers [23] relied on interleaved SRAM memory banks to provide high bandwidth at moderate latency, but modern vector supercomputers [6, 17] now implement main memory with the same commodity DRAM parts as other sectors of the computing industry, for improved cost, power, and density. As with conventional scalar processors, designers of all classes of vector machine, from vector supercomputers [6] to vector microprocessors [8], are motivated to add vector data caches to improve the effective bandwidth and latency of a DRAM main memory.

For vector codes with temporal locality, caches can provide a significant boost in throughput and a reduction in the energy consumed for off-chip accesses. But, even for codes with significant reuse, it is important that the cache not impede the flow of data from main memory. Existing non-blocking vector cache designs are complex since they must track all primary misses and merge secondary misses using replay queues [1, 11]. This complexity also pervades the vector unit itself, as element storage must be reserved for all pending data accesses in either vector registers [10, 18] or a decoupled load data queue [9].

In this paper, we introduce *vector refill/access decoupling*, which is a simple and inexpensive mechanism to sustain high memory bandwidths in a cached vector machine. A scalar unit runs ahead queuing compactly encoded instructions for the vector units, and a vector refill unit quickly pre-executes vector memory instructions to detect which of the lines they will touch are not in cache and should be prefetched. This exact non-speculative hardware prefetch tries to ensure that when the vector memory instruction is eventually executed, it will find data already in cache. The cache can be simpler because we do not need to track and merge large numbers of pending misses. The vector unit is also simpler as less buffering is needed for pending element accesses. Vector refill/access decoupling is a microarchitectural technique that is invisible to software, and so can be used with any existing vector architecture.

We also propose new *vector segment memory instructions*, which encode common two-dimensional vector access patterns such as array-of-structures and loop raking. Vector segment instructions improve performance by converting some strided accesses into contiguous bursts, and improve the efficiency of refill/access decoupling by reducing the number of cache tag probes required by the refill engine.

We describe an implementation of these techniques within the context of the SCALE vector-thread processor [19] and provide an evaluation over a range of scientific and embedded kernels. Our results show an improvement in performance and a dramatic reduction in the hardware resources required to sustain high throughput in long latency memory systems.

2. Memory Bandwidth-Delay Products

The *bandwidth-delay product* for a memory system is the peak memory bandwidth, B, in bytes per processor cycle multiplied by the round-trip access latency, L, in processor cycles. To saturate a given memory system, a processor must support at least $(B/b) \times L$ independent b-byte elements in flight simultaneously. The relative latency of DRAM has been growing, while at the same time DRAM bandwidth has been rapidly improving through advances such as pipelined accesses, multiple interleaved banks, and high-speed double-data-rate interfaces. These trends combine to yield large and growing

*This work was partly funded by NSF CAREER Award CCR-0093354 and by the Cambridge-MIT Institute.

bandwidth-delay products. For example, a current Pentium-4 desktop has around two bytes per processor cycle of main memory bandwidth with around 400 cycles of latency, representing around 200 independent 4-byte elements. A Cray X1 vector unit has around 32 bytes per cycle of global memory bandwidth with up to 800 cycles of latency across a large multiprocessor machine, representing around 1,600 independent 8-byte words [6].

Each in-flight memory request has an associated hardware cost. The *access management state* is the information required to track an in-flight access, and the *reserved element data buffering* is the storage space into which the memory system will write returned data. In practice, it is impossible to stall a deeply pipelined memory system, and so element buffering must be reserved at request initiation time. The cost of supporting full memory throughput grows linearly with the bandwidth-delay product, and it is often this control overhead rather than raw memory bandwidth that limits memory system performance.

Vector machines are successful at saturating large bandwidth-delay product memory systems because they exploit *structured memory access patterns*, groups of independent memory accesses whose addresses form a simple pattern and therefore are known well in advance. Unit-stride and strided vector memory instructions compactly encode hundreds of individual element accesses in a single vector instruction [17, 23], and help amortize the cost of access management state across multiple elements.

At first, it may seem that a cache also exploits structured access patterns by fetching full cache lines (and thus many elements) for each request, and so a large bandwidth-delay memory system can be saturated by only tracking a relatively small number of outstanding cache line misses. Although each cache miss brings in many data elements, the processor cannot issue the request that will generate the *next* miss until it has issued all preceding requests. If these intervening requests access cache lines that are still in flight, the requests must be buffered up until the line returns, and this buffering grows with the bandwidth-delay product. Any request which misses in the cache has *twice* the usual reserved element data storage: a register is reserved in the processor itself plus buffering is reserved in the cache.

Data caches amplify memory bandwidth, and in doing so they can increase the *effective* bandwidth-delay product. For example, consider a non-blocking cache with four times more bandwidth than main memory, and an application which accesses a stream of data and reuses each operand three times. The processor must now support three times the number of outstanding elements to attain peak memory throughput. With reuse, the bandwidth-delay product of a cached memory system can be as large as the cache bandwidth multiplied by the main memory latency.

3. Vector Refill/Access Decoupling

An alternative to buffering processor requests after they are issued is to *prefetch* cache data ahead of time so that the processor sees only hits. A successful prefetching scheme can dramatically improve the cost-performance of a large bandwidth-delay memory system. The goals of a prefetching scheme are to fetch only the required data, to do so before the processor requires the data, and to not evict useful data from the cache. In this section, we introduce the refill/access decoupling scheme and describe how it achieves these goals.

Figure 1 shows our baseline system, a cached decoupled vector machine (DVM) broadly similar to previous decoupled vector architectures [9, 25]. DVM includes three components: main memory, a non-blocking cache, and a decoupled vector processor. In this paper, we treat main memory as a simple pipeline with variable bandwidth and latency. The non-blocking cache includes a tag array, data array, and miss status handling registers (MSHRs). The decoupled vector processor contains a control processor (CP), vector execution unit (VEU), vector load unit (VLU), and vector store unit (VSU). A decoupled vector machine with refill/access decoupling (DVMR) extends DVM with a vector refill unit (VRU).

3.1. Non-Blocking Cache

The non-blocking cache shown in Figure 1 is similar to other non-blocking caches found both in scalar [20] and vector machines [1, 11]. The cache supports both *primary misses* and *secondary misses*. A primary miss is the first miss to a cache line and causes a refill request to be sent to main memory, while secondary misses are accesses which miss in the cache but are for the same cache line as an earlier primary miss. Misses are tracked in the MSHR using *primary miss tags* and *replay queues*. Primary miss tags hold the address of an in-flight refill request and are searched on every miss to determine if it is a secondary miss. A primary miss allocates a new primary miss tag to hold the address, issues a refill request, performs an eviction if needed, and adds a new replay queue entry to a linked list next to the primary miss tag. Replay queue entries contain information about the access including the corresponding cache line offset, the byte width, the destination register for loads, and pending data for stores. A secondary miss adds a new replay queue entry to the appropriate replay queue, but does not send an additional refill request to main memory.

3.2. Basic Decoupled Vector Processor

In a decoupled vector processor the CP runs ahead and queues vector memory or compute commands for the vector units. The VEU may include a heavily pipelined datapath or multiple parallel execution units to exploit data level compute parallelism. The VLU provides access/execute decoupling, while the VSU provides store decoupling.

It is useful to explain DVM by following a single vector load request through the system. The CP first sends a vector load command to the Vector-CmdQ. When it reaches the front, the command is broken into two pieces: the address portion is sent to the VLU-CmdQ while the register writeback portion is sent to the VEU-CmdQ. The VLU then breaks the long vector load command into multiple smaller subblocks. For each subblock it reserves an entry in the vector load data queue (VLDQ) and issues a cache request. On a hit, the data is loaded from the data array into the reserved VLDQ entry.

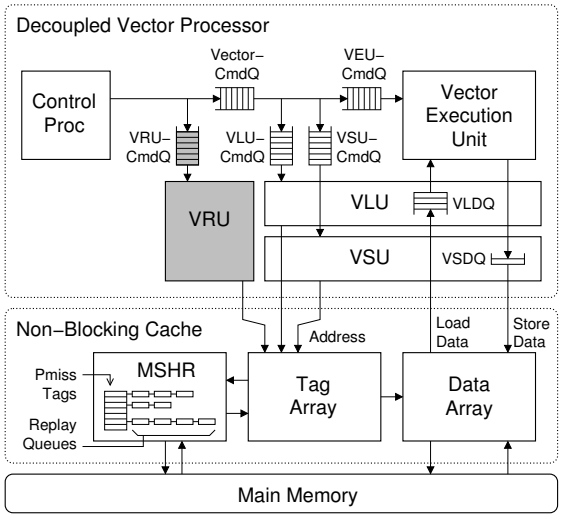


Figure 1: Basic decoupled vector machine (DVM). The vector processor is made up of several units including the control processor (CP), vector execution unit (VEU), vector load unit (VLU), and vector store unit (VSU). The non-blocking cache contains miss status handling registers (MSHR), a tag array, and a data array. Refill/access decoupling is enabled by adding the highlighted components: a vector refill unit (VRU) and corresponding command queue.

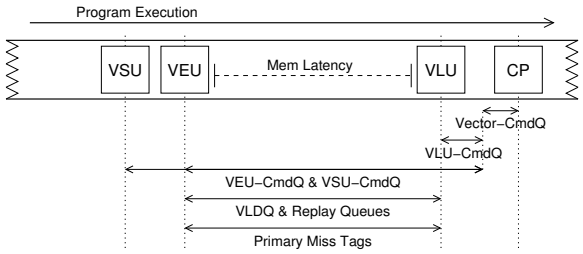


Figure 2: Queuing resources required in basic DVM. Decoupling allows the CP and VLU to run ahead of the VEU. Many large queues (represented by longer arrows) are needed to saturate memory.

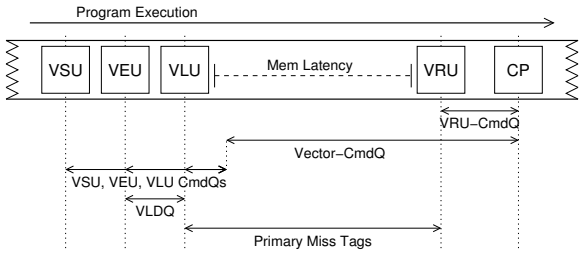


Figure 3: Queuing resources required in a DVMR. Refill/access decoupling requires fewer resources to saturate the memory system.

The VLDQ enables the VLU to run ahead performing many memory requests while the VEU trails behind and moves data in FIFO order into the architecturally visible vector registers. The VLDQ acts as a small memory reorder buffer since requests can return from the memory system out-of-order: hits may return while an earlier miss is still outstanding. The VSU trails behind both the VLU and VEU and writes results to the memory system.

Decoupling is enabled by queues which buffer data and control information for the trailing units to allow the leading units to run ahead. Figure 2 illustrates the resource requirements for DVM. The distance that the CP may run ahead is determined by the size of the various vector command queues. The distance that the VLU may run ahead is constrained by the VEU-CmdQ, VSU-CmdQ, VLDQ, replay queues, and primary miss tags. The key observation is that to tolerate increasing memory latencies these resources must all be proportional to the memory bandwidth-delay product and thus can become quite large in modern memory systems. In this system the VLDQ is the reserved data element state, while the replay queues and the primary miss tags are the access management state required to support the in-flight memory requests.

3.3. Vector Refill Unit

We enable refill/access decoupling by extending a basic decoupled vector machine with a vector refill unit (VRU). The VRU and associated VRU-CmdQ are shown highlighted in Figure 1. The VRU receives the same commands as the VLU, and it runs ahead of the VLU issuing refill requests for the associated cache-lines. The primary cost of the VRU is an additional address generator. Ideally, the VRU runs sufficiently ahead to allow the VLU to always hit in the cache. A refill request only brings data into the cache, and as such it needs a primary miss tag but no replay queue entry. It also does not need to provide any associated reserved element buffering in the processor, and refill requests which hit in cache due to reuse in the load stream are ignored. In this paper we assume that stores are non-allocating and thus the VRU need not process vector store commands.

Figure 3 illustrates the associated resource requirements for a decoupled vector machine with refill/access decoupling (DVMR). The VRU is able to run ahead with no need for replay queues. The VLU runs further behind the CP compared to the basic decoupled vector machine shown in Figure 2, and so the Vector-CmdQ must be larger. However, since the VLU accesses are typically hits the distance between the VLU and VEU is shorter and the VLDQ is smaller. With refill/access decoupling, the only resources that must grow to tolerate an increasing memory latency are the Vector-CmdQ and the number of primary miss tags. This scaling is very efficient since the Vector-CmdQ holds vector commands which each compactly encode many element operations and the primary miss tags must only track non-redundant cache line requests. The only reserved element buffering involved in scaling refill/access decoupling is the efficient storage provided by the cache itself.

A DVMR implementation will need to carefully manage the relative steady-state rates of the VLU and VRU. If the VRU is able to run too far ahead it will start to evict lines which the VLU has yet to even access, and if the VLU can outpace the VRU then the refill/access decoupling will be ineffective. In addition to the rates between the two units, an implementation must also ensure that the temporal distance

between the two units is large enough to cover the memory latency. Throttling the VLU and/or the VRU can help constrain these rates and distances to enable smoother operation.

One disadvantage of refill/access decoupling is an increase in cache request bandwidth: the VRU first probes the cache to generate a refill and then the VLU accesses the cache to actually retrieve the data. This overhead is usually low since the VRU must only probe the cache once per cache line. Another important concern with refill/access decoupling is support for unstructured load accesses (i.e. indexed vector loads). Limited replay queuing is still necessary to provide these with some degree of access/execute decoupling.

4. Vector Segment Memory Accesses

Vector unit-stride accesses move consecutive elements in memory into consecutive elements in a single vector register. Although these one-dimensional unit-stride access patterns are very common, many applications also include two-dimensional access patterns where consecutive elements in memory are moved into *different* vector registers. For example, assume that an application needs to process an array of fixed-width structures such as an array of image pixels or polygon vertices. A programmer can use multiple strided accesses to load the structures into vector registers such that the same field for all structures is in the same vector register. The following sequence of vector instructions uses three strided vector accesses (each with a stride of three) to load an array (`A`) of RGB pixel values into the vector registers `vr1`, `vr2`, and `vr3`:

```
la     r1, A
li     r2, 3
vlbst  vr1, r1, r2
addu   r1, r1, 1
vlbst  vr2, r1, r2
addu   r1, r1, 1
vlbst  vr3, r1, r2
```

In this example, the programmer has converted a two-dimensional access pattern into multiple one-dimensional access patterns. Unfortunately, these multiple strided accesses hide the spatial locality in the original higher-order access pattern and can cause poor performance in the memory system including increased bank conflicts, wasted address bandwidth, additional access management state, and wasted non-allocating store data bandwidth. A different data layout might enable the programmer to use more efficient unit-stride accesses, but reorganizing the data layout requires additional overhead and is complicated by external API constraints.

We propose *vector segment memory accesses* as a new vector mechanism which more directly captures the two-dimensional nature of many higher-order access patterns. They are similar in spirit to the stream loads and stores found in stream processors [4, 16]. As an example, the following sequence of vector instructions uses a segment access to load an array of RGB pixel values.

```
la     r1, A
vlbseg 3, vr1, r1
```

The `vlbseg` instruction has three fields: a segment length encoded as an immediate, a base destination vector register specifier, and a scalar base address register specifier. The instruction writes each element of each segment into consecutive vector registers starting with the base destination vector register. In this example, the red values would be loaded into `vr1`, the green values into `vr2`, and the blue values into `vr3`. The segment length is therefore limited by the number of vector registers. The basic vector segment concept can be extended to include segments which are offset at a constant stride enabling strided vector segment accesses. Traditional strided accesses can then be reduced both in the vector ISA and the implementation to strided segment accesses with a segment length of one.

Segment accesses are useful for more than just array-of-structures access patterns; they are appropriate any time a programmer needs to move consecutive elements in memory into different vector registers. Segments can be used to efficiently access sub-matrices in a larger matrix or to implement vector loop-raking.

5. The SCALE Decoupled Vector Machine

We evaluate vector refill/access decoupling and vector segment accesses in the context of the SCALE vector-thread processor [19] shown in Figure 4. For this work, we use SCALE as a more traditional decoupled vector machine and do not make use of its more advanced threading features. SCALE is roughly similar to the more abstract DVMR shown in Figure 1 with several key differences: the VEU contains four execution lanes with clustered functional units, the VLU and VSU support segment accesses, and the non-blocking cache is divided into four banks. In this section we first describe the SCALE decoupled vector processor and the SCALE non-blocking cache before discussing the SCALE vector refill unit.

5.1. SCALE Decoupled Vector Processor

The single-issue MIPS-RISC scalar control processor executes the control code and issues commands to the vector units. A *vector configuration* command allows the control processor to adjust the maximum vector length based on the register requirements of the application. A *vector fetch* command sends a block of compute instructions to the vector execution unit, and these may include indexed load and store operations. A *vector load* or *vector store* command specifies a structured memory access that reads data from or writes data to the vector registers in the VEU. Each lane in the VEU has four execution clusters and provides a peak per-cycle bandwidth of one load element, one store element, and four cluster operations. Cluster 0 executes indexed loads and stores, and uses a load data queue (LDQ) and store address queue (SAQ) to provide access/execute decoupling.

The VLU and VSU each process one vector command at a time using either a single unit-stride address generator or per-lane segment address generators. The address generators step through each vector command and break them into subblocks with up to four elements. The VLU manages per-lane VLDQs. For unit-stride commands, the VLU allocates VLDQ entries in parallel across lanes for each cache access,

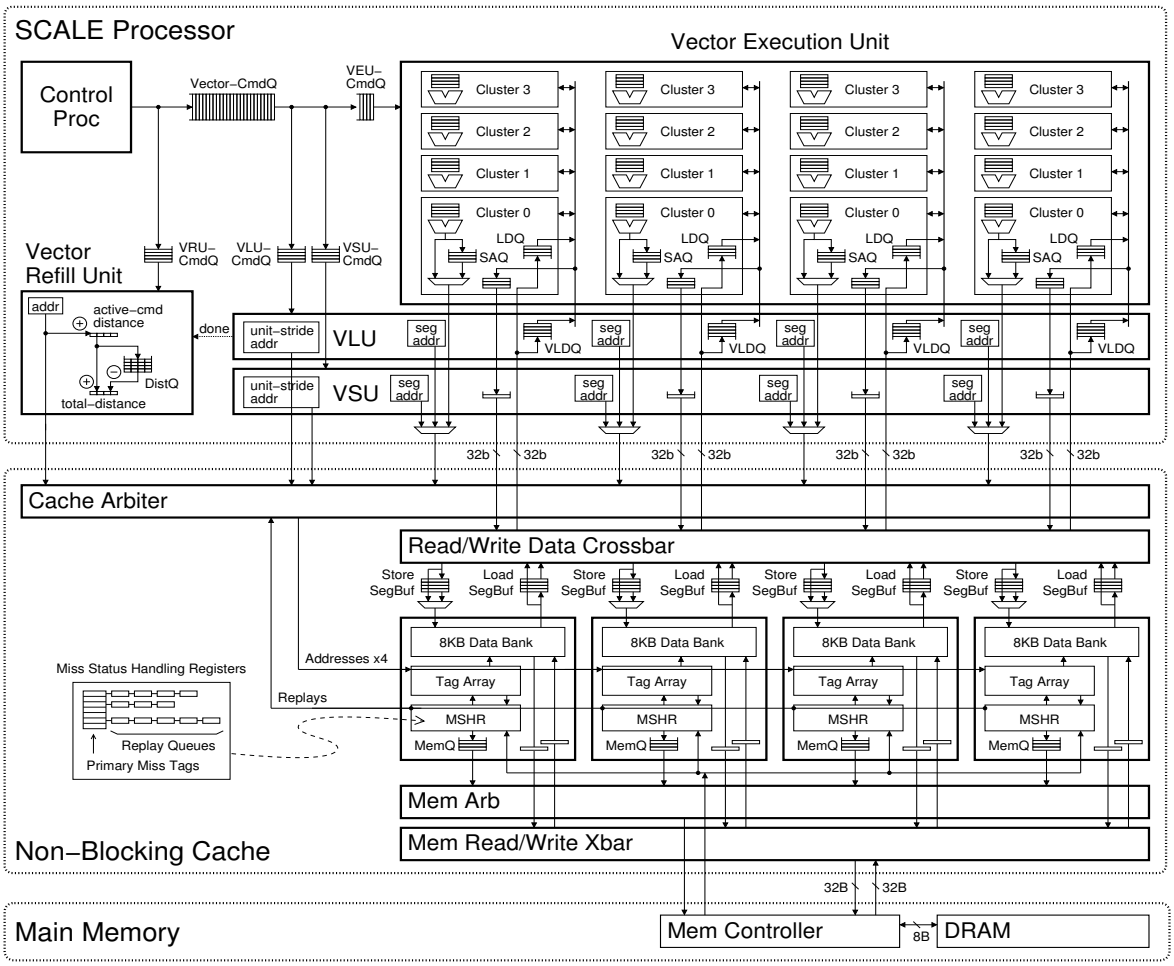


Figure 4: Microarchitecture of the SCALE decoupled vector machine. For clarity the figure omits the CP ports into the cache.

and writes the data elements into the VLDQs in parallel when the access returns. For segment commands, the VLU may allocate multiple VLDQ entries in each lane for each segment access. After the data is read from the cache, it is held in per-lane segment buffers next to the cache banks and returned to each lane's VLDQ at a rate on one element per cycle. Since both VLU and indexed lane load requests share the write-back bus into the lane, they must arbitrate for this resource. The VSU sends unit-stride store data to the cache in parallel for each access. Segment store data is written into per-lane buffers next to the cache banks at a rate of one element per cycle, and then written into the cache using a single access.

As with most vector machines, SCALE requires software memory fences to enforce ordering when there are potential memory dependencies. These are implemented by stalling the head of the Vector-CmdQ until the VEU, VLU, and VSU are idle. Importantly, this does not block the control processor and VRU from continuing to run ahead.

5.2. SCALE Non-Blocking Cache

The 32 KB non-blocking first-level unified instruction and data cache is comprised of four banks which each support one request of up to 16 B per cycle. The cache has 32 B lines and uses CAM-tags to provide 32-way set-associativity with FIFO replacement. The cache has 16 B load and store segment buffers for every lane/bank pair, and we expect that these can be efficiently implemented under the data crossbar. A priority arbiter decides which of the requesters can access each bank; thus the maximum throughput is four requests per cycle if those requests go to different banks. The cache has a two-cycle hit latency. Arbitration and tag check occur in the first cycle and data is read from or written to the data bank on the second cycle. Requests that miss in the cache still consume an unused cycle of writeback bandwidth into the corresponding requester.

As in the abstract non-blocking cache discussed in Section 3, primary misses allocate primary miss tags and both primary and secondary misses (from requesters other than the VRU) require a corresponding replay queue entry. On a miss, a refill request and the index of a victim line (if needed) are placed in the bank's memory queue (MemQ). When the entry reaches the head of the MemQ, the bank first sends the refill request to memory, then steals unused bank cycles to read the victim and issue a memory writeback request. When a refill returns from memory it is immediately written back into the data array and the replay queue is used to replay one primary

or secondary miss per cycle. Since multiple banks can be returning load replay data or load hit data to the same requester on a cycle, replays must arbitrate for writeback ports through the main cache arbiter. The bank blocks if the MemQ is full or if there are no more pending tags or replay queue entries.

The SCALE cache is a write-back/no-write-allocate cache. Store data for pending store misses is held in a per-bank *pending store data buffer* and store replay queue entries contain pointers into this buffer. Secondary store misses are merged where possible. Non-allocating stores are held in the replay queues while waiting to be sent to memory and are converted to allocating stores if there is a secondary load miss to the same cache line.

5.3. SCALE Vector Refill Unit

The VRU processes vector load commands and issues up to one refill request per cycle to the cache. Since the VRU does not use replay queues, the VRU is not blocked access to the bank when there are no free replay queue entries.

The VRU has two refill modes: unit-stride and strided. It uses the unit-stride refill mode for unit-stride vector loads and segment/strided vector loads with positive strides less than the cache line size. In this mode, the VRU calculates the number of cache lines in the memory footprint once when it begins processing a command, and then issues cache-line refill requests to cover the entire region. The hardware cost for the segment/strided conversion includes a comparator to determine the short stride and a small multiplier to calculate the vector length (7 bits) times the stride (5 bits). For the strided refill mode, the VRU repeatedly issues a refill request to the cache and increments its address by the stride until it covers the entire vector. It also issues an additional request whenever a segment straddles a cache-line boundary.

Three important throttling techniques prevent the VLU and the VRU from interfering with each other. (1) Ideally, the VRU runs far enough ahead so that all of the VLU accesses hit in the cache. However, when the program is memory bandwidth limited, the VLU can always catch up with the VRU. The VLU interference wastes cache bandwidth and miss resources and can reduce overall performance. As a solution, the VLU is throttled to prevent it from having more than a small number of outstanding misses. (2) The VLU has four address generators and can therefore process segment/strided vector loads that hit in the cache at a faster rate than the VRU. Whenever the VLU finishes a vector load command before the VRU, the VRU aborts the command. (3) If the VRU is not constrained, it will run ahead and use all of the primary miss tags, causing the cache to block and preventing other requesters from issuing cache hits. The VRU is throttled to reserve one or two primary miss tags for the other requesters.

When a program is compute-limited, the VRU may run arbitrarily far ahead and evict data that it has prefetched into the cache before the VLU even accesses it. To avoid this, the VRU is also throttled based on the *distance* that it is running ahead of the VLU. To track distance, the VRU counts the number of cache-line requests that it makes for each vector load command (including hits). After finishing each command, it increments its total distance count and enqueues that command's distance count in its DistQ (see Figure 4). Whenever the VLU finishes a vector load, the VRU pops the head of the DistQ and decrements its total distance count by the count for that command. A single distance count is sufficient for programs that use the cache uniformly, but it can fail when a program has many accesses to a single set in the cache. As a solution, the VRU maintains per-set counts, and throttles based on the maximum per-set distance. To reduce the hardware cost, the distance-sets that the VRU divides its counts into may be more coarse-grained than the actual cache sets.

6. Evaluation

Several scientific and embedded kernels with varying amounts and types of access parallelism were implemented for the SCALE vector-thread architecture. We evaluate vector refill/access decoupling and vector segment accesses in terms of performance and resource utilization for three machine configurations: a basic decoupled vector machine (DVM), a decoupled vector machine with refill/access decoupling (DVMR), and a decoupled scalar machine (DSM). We use SCALE as our DVMR design, and we disable the vector refill unit to model the DVM machine. For the DSM machine, we dynamically convert vector memory accesses into scalar element accesses to model an optimistic scalar processor with four load-store units and enough decoupling resources to produce hundreds of in-flight memory accesses. We also evaluate throttling mechanisms and memory latency scaling for refill/access decoupling. Main memory is modeled with a simple pipelined magic memory that has a bandwidth of 8 bytes per processor clock cycle.

6.1. Benchmarks

Table 1 lists the benchmarks used in this evaluation. All vector code was hand-written in assembler and linked with C code compiled for the MIPS control processor using `gcc`. Results in the table are intended to approximately indicate each benchmark's peak performance given the processor, cache, and memory bandwidth constraints. They are based on a *pseudo-ideal* SCALE configuration with very large decoupling queues, VRU refill/access decoupling, and zero cycle main memory with a peak bandwidth of 8 B/cycle. The results throughout this paper are normalized to this performance.

The first eight benchmarks are custom kernels chosen to exercise various memory access patterns. `vvadd-word`, `vvadd-byte`, and `vvadd-cmplx` all perform vector-vector additions, with `vvadd-cmplx` making use of segment accesses to read and write the interleaved complex data. `vertex` is a graphics kernel which projects 3-D vertices in homogeneous coordinates onto a 2-D plane using four-element segment accesses to load and store the vertices. `fir` performs a 35-tap finite impulse response filter by multiplying vectors of input elements by successive tap coefficients while accumulating the output elements. Successive vector-loads are offset by one element such that each element is loaded 35 times. `transpose` uses strided segment loads and unit-stride stores to perform an out-of-place matrix transposition on word element data. The 512×512 data set size has a stride

Benchmark Name	Input Size	Access Pattern: Indexed Load	Indexed Store	Unit Stride Load	Unit Stride Store	Strided Load	Strided Store	Segment Load	Segment Store	Avg VL	Ops/Cycle Total	Ops/Cycle Muls	Elements/Cycle Load	Elements/Cycle Store	Cache Bytes/Cycle	Mem Bytes/Cycle Total	Mem Bytes/Cycle Load	Mem Bytes/Cycle Store
vvadd-word	250k elements			2 W	1 W					64.0	0.67	0.00	1.33	0.67	8.00	8.00	5.33	2.67
vvadd-byte	1M elements			2 B	1 B					64.0	1.88	0.00	3.76	1.88	5.64	6.57	3.76	2.81
vvadd-cmplx	125k cmplx nums							2 W[2]:*	1 W[2]:*	32.0	0.67	0.00	1.33	0.67	7.99	8.00	5.33	2.67
vertex	100k 4W vertices							1 W[4]:*	1 W[4]:*	28.0	9.55	3.82	0.96	0.96	7.64	7.65	3.82	3.82
fir	500k elements			1 H	1 H					124.0	7.36	3.58	3.61	0.10	7.42	0.43	0.20	0.22
transpose400	400×400 words				8 W			1 W[8]:1600		16.0	0.00	0.00	1.00	1.00	8.00	8.00	4.00	4.00
transpose512	512×512 words				8 W			1 W[8]:2048		16.0	0.00	0.00	0.96	0.96	7.65	7.66	3.83	3.83
idct	20k 8×8 blocks			8 H	8 H			1 H[8]:*	1 H[8]:*	12.0	8.80	1.45	0.96	0.96	3.86	3.85	1.93	1.92
rgbyiq	320×240 3B pixels							1 B[3]:*	1 B[3]:*	28.0	9.33	4.00	1.33	1.33	2.67	3.91	1.33	2.58
rgbcmyk	320×240 3B pixels				1 W			1 B[3]:*		32.0	6.80	0.00	1.20	0.40	2.80	3.00	1.20	1.80
hpg	320×240 1B pixels			3 B	1 B					63.6	10.77	3.91	2.93	0.98	3.92	2.54	1.01	1.54
fft	4096 points		√	4 H	4 H	6 H:?	4 H:?			31.4	3.25	1.00	1.50	1.08	5.16	1.13	0.44	0.69
rotate	742×768 bits			8 B			8 B:-768			15.5	9.01	0.00	0.47	0.47	0.94	7.99	0.47	7.52
dither	256×256 1B pixels	√	√	4 H	1 H	1 B:254				31.3	5.09	0.65	1.11	0.27	2.26	0.25	0.22	0.03

Table 1: Benchmark characterization. The *Access Pattern* columns display the types of memory access streams used by each benchmark. The entries are of the format $N\{B, H, W\}[n] : S$, where N indicates the number of streams of this type, B, H, or W indicates the element width (byte, half-word, or word), $[n]$ indicates the segment size in elements, and S indicates the stride in bytes between successive elements or segments. A stride of '*' indicates that the segments are consecutive, and a stride of '?' indicates that the stride changes throughout the benchmark. The *Avg VL* column displays the average vector length used by the benchmark. The per-cycle statistics are for the *pseudo-ideal* SCALE configuration that is used as a baseline for normalization throughout this paper.

which maps columns to a single set in the cache. `idct` performs an in-place 2-D 8×8 inverse discrete cosine transform on a series of blocks using the LLM algorithm. The implementation first uses segment vector loads and stores to perform a 1-D IDCT on the block rows, then uses unit-stride accesses to perform a 1-D IDCT on the columns.

The last six benchmarks are from the EEMBC consumer, telecom, and office suites. `rgbyiq` and `rgbcmyk` are RGB color conversion kernels which use three-element segment accesses to efficiently load input pixels. `rgbyiq` also uses segment accesses to store the converted pixels, but `rgbcmyk` uses a packed word store to write its output data. `hpg` is a two dimensional gray-scale convolution over one byte pixels with a 3×3 high-pass filter. Each output pixel is a function of nine input pixels, but the kernel is optimized to load each input pixel three times and hold intermediate input row computations in registers as it produces successive output rows. `fft` is a radix-2 Fast Fourier Transform based on a decimation-in-time Cooley-Tukey algorithm. The algorithm inverts the inner loops after the first few butterfly stages to maximize vector lengths, resulting in a complex mix of strided and unit-stride accesses. The benchmark performs the initial bit-reversal using unit-stride loads and indexed stores. `rotate` turns a binary image by 90^o. It uses 8 unit-stride byte loads to feed an 8-bit×8-bit block rotation in registers, and then uses 8 strided byte stores to output the block. `dither` performs Floyd-Steinberg dithering which takes grey-scale byte pixels as input and outputs a binary image. The benchmark uses a strided byte access to load the input image and indexed loads and stores to read-modify-write the bit-packed output image. The dithering error buffer is updated with unit-stride accesses.

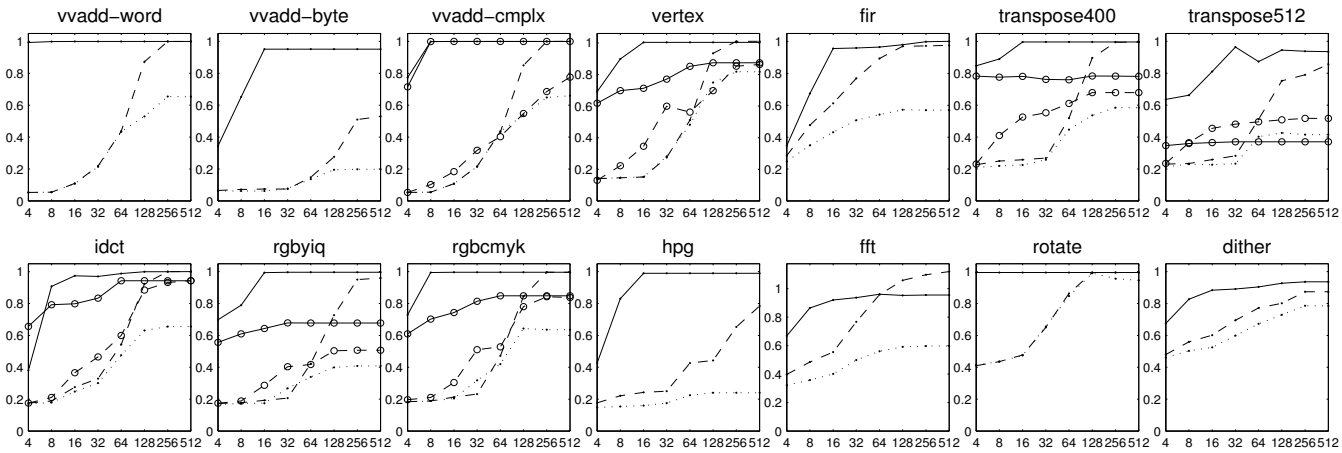
Table 1 shows that the kernels which operate on word data drive memory at or near the limit of 8 B/cycle. `vvadd-byte` and `fir` approach the limit of 4 load elements per cycle; and `vertex`, `fir`, `rgbyiq`, and `hpg` approach the limit of 4 multiplies per cycle. Although input data sets were chosen to be larger than the 32 KB cache, several benchmarks are able to use the cache to exploit significant temporal locality. In particular, `fir`, `dither`, and `fft` have cache-to-memory bandwidth amplifications of 17×, 9×, and 4.6× respectively. Several benchmarks have higher memory bandwidth than cache bandwidth due to the non-allocating store policy and insufficient spatial-temporal locality for store writeback merging.

6.2. Reserved Element Buffering and Access Management State

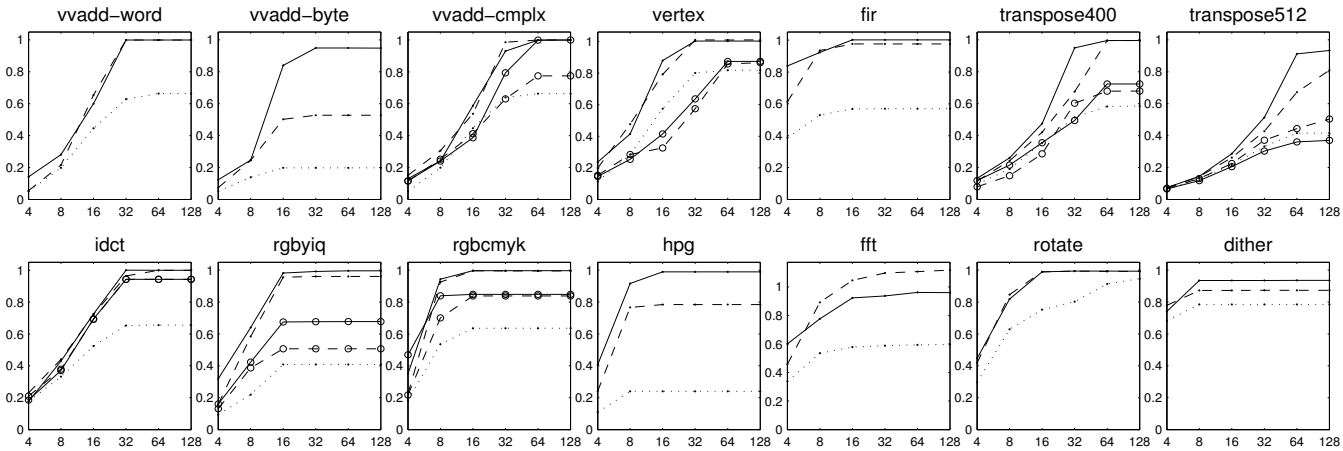
Figure 5 presents three limit studies in which we restrict either: (a) the reserved element buffering provided by the VLDQs, (b) the number of primary miss tags, or (c) the number of replay queue entries. To isolate the requirements for each of these resources, all of the other decoupling queue sizes are set to very large values, and in each of the three studies the other two resources are unconstrained. For these limit studies, the cache uses LRU replacement, and the DVMR distance throttling uses 32 distance-sets each limited to a distance of 24. Memory has a latency of 100 cycles.

The overall trend in Figure 5 shows that the DVMR machine is almost always able to achieve peak performance with drastically fewer reserved element buffering and access management resources compared to the DVM and DSM machines. All the machines must use primary miss tags to track many in-flight cache lines. However, in order to generate these misses, DVM and DSM also require 10s–100s of replay queue entries and reserved element buffering registers. DVMR mitigates the need for these by decoupling refill requests so that demand accesses hit in the cache.

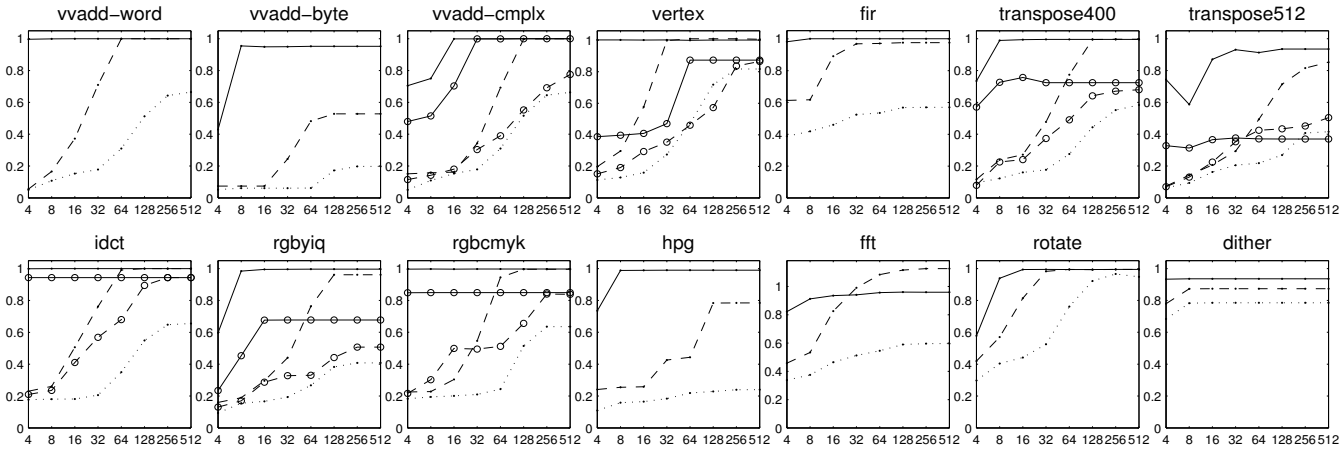
With the 100 cycle memory latency, `vvadd-word` must sustain 533 B of in-flight load data to saturate its peak load bandwidth of 5.33 B/cycle (Table 1). This is equivalent to about 17 cache lines, and the benchmark reaches its peak performance with 32 primary miss tags. DVM accesses four word elements with each cache access, and so requires two replay queue entries per primary miss tag (64 total) and 256 VLDQ registers to provide reserved element buffering for these in-flight accesses. In stark contrast, DVMR does not require any VLDQ reserved elements or replay queue entries to generate the memory requests. DSM issues independent accesses for each of the eight words in a cache line and so



(a) VLDQ entries (reserved element buffering).



(b) Primary miss tags (access management state).



(c) Replay queue entries (access management state).

Figure 5: Performance scaling with increasing reserved element buffering and access management resources. In (a), the x-axis indicates the total number of VLDQ entries across all four lanes, and in (b) and (c) the x-axis indicates the total number of primary miss tags and replay queue entries across all four cache banks. For all plots, the y-axis indicates performance relative to the *pseudo-ideal* configuration whose absolute performance is given in Table 1. DVMR is shown with solid lines, DVM with dashed lines, and DSM with dotted lines. The lines with circle markers indicate segment vector-memory accesses converted into strided accesses.

requires both 256 replay-queue entries and 256 reserved element buffering registers to achieve its peak performance. However, even with these resources it is still unable to saturate the memory bandwidth because, (1) each cache miss wastes a cycle of writeback bandwidth into the lanes, and (2) the lanes have bank conflicts when they issue scalar loads and stores. The other load memory bandwidth limited benchmarks share similarities with `vvadd-word` in their performance scaling.

The `vvadd-byte` kernel makes it even more difficult for DVM and DSM to drive memory since each load accesses four times less data than in `vvadd-word`. Thus, to generate a given amount of in-flight load data, they require four times more replay queue entries and VLDQ registers. Furthermore, the wasted lane writeback bandwidth and cache access bandwidth caused by misses are much more critical for `vvadd-byte`. DVM's cache misses waste half of the critical four elements per cycle lane writeback bandwidth, and so its peak performance is only half of that for DVMR. To reach the peak load bandwidth, DVMR uses 16 VLDQ registers (4 per lane) to cover the cache hit latency without stalling the VLU. The `rgbyiq` and `rgbcmyk` benchmarks have similar characteristics to `vvadd-byte` since they operate on streams of byte elements, but their peak throughput is limited by the processor's compute resources.

Refill/access decoupling can provide a huge benefit for access patterns with temporal reuse. The `hpg` benchmark must only sustain 1 B/cycle of load bandwidth to reach its peak throughput, and so requires 100 B of in-flight data. However, because the benchmark reads each input byte three times, DVM and DSM each require 300 reserved element buffering registers for the in-flight data. DVM accesses each 32 B cache line 24 times with four-element vector-loads, and thus requires 72 replay queue entries to sustain the three in-flight cache misses; and DSM needs four times this amount. DVMR is able to use dramatically fewer resources by prefetching each cache line with a single refill request and then discarding the two subsequent redundant requests.

Refill/access decoupling can also provide a large benefit for programs that only have occasional cache misses. To achieve peak performance, `rotate`, `dither`, and `fir` must only sustain an average load bandwidth of one cache line every 68, 145, and 160 cycles respectively. However, to avoid stalling during the 100 cycles that it takes to service these occasional cache misses, DVM must use on the order of 100 VLDQ registers as reserved storage for secondary misses (`fir` and `rotate`) or to hold hit-under-miss data (`dither`). DVMR attains high performance using drastically fewer resources by generating the cache misses long before the data is needed.

For benchmarks which use segment vector-memory accesses, we evaluate their effect by dynamically converting each n-element segment access into n strided vector-memory accesses. Figure 5 shows that several benchmarks rely on segment vector-memory accesses to attain peak performance, even with unlimited decoupling resources. Segments can improve performance over strided accesses by reducing the number of cache accesses and bank conflicts. For example, without segments, the number of bank conflicts increases by 2.8× for `rgbcmyk`, 3.4× for `vertex`, 7.5× for `rgbyiq`, and 22× for `transpose400`. Segments also improve locality in the store stream, and thereby alleviate the need to merge independent non-allocating stores in order to optimize memory bandwidth and store buffering resources.

Benchmark Name	Vector-CmdQ (size)			Total-Distance (cache lines)			Set-Distance-24 (sets)				
	1024	256	64	768	100	24	2	4	8	16	32
`vvadd-word`	0.03	0.03	1.00	1.00	1.00	0.32	0.63	1.00	1.00	1.00	1.00
`vvadd-byte`	0.02	0.95	0.90	0.96	1.00	0.48	0.92	0.94	0.98	0.98	0.92
`vvadd-cmplx`	0.04	0.04	0.96	1.00	1.00	0.32	0.63	1.00	1.00	1.00	1.00
`vertex`	0.08	0.40	0.97	0.99	1.00	0.44	0.80	1.00	1.00	1.00	0.99
`idct`	0.15	0.98	0.88	1.00	0.97	0.65	0.81	0.96	0.97	0.99	1.00
`fir`	0.99	1.00	0.99	1.00	0.97	0.79	0.84	0.92	0.99	1.00	0.99
`transpose400`	0.12	1.00	0.96	1.00	0.99	0.30	0.31	0.78	0.99	1.00	1.00
`transpose512`	0.25	0.25	0.25	0.25	0.25	0.98	1.00	1.00	1.00	1.00	1.00
`rgbyiq`	1.00	1.00	1.00	1.00	1.00	0.98	1.00	1.00	1.00	1.00	1.00
`rgbcmyk`	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
`hpg`	0.19	0.26	0.98	0.19	0.99	0.46	0.63	0.97	1.00	0.52	0.19
`fft`	1.00	0.98	0.96	0.97	0.98	0.85	0.94	0.97	0.97	0.97	0.97
`rotate`	0.99	0.98	0.94	0.99	0.99	0.94	0.96	0.98	1.00	1.00	0.99
`dither`	0.98	0.96	0.96	1.00	0.95	0.61	0.71	0.87	0.93	0.99	1.00

Table 2: Performance for various refill distance throttling mechanisms. For each benchmark, the performance values are normalized to the best performance for that benchmark attained using any of the mechanisms.

Although DVMR always benefits from segment accesses, DVM is better off with strided accesses when reserved element buffering resources are limited. This is because a strided pattern is able to use the available VLDQ registers to generate more in-flight cache misses by first loading only one element in each segment, effectively prefetching the segment before the subsequent strided accesses return to load the neighboring elements.

6.3. Refill Distance Throttling

We next evaluate mechanisms for throttling the distance that the refill unit runs ahead of the VLU. Table 2 presents results for several throttling mechanisms at a memory latency of 400 cycles. The first group of columns evaluate having no explicit throttling other than the size of the *Vector-CmdQ*. With a size of 1024, many of the benchmarks have dismal performance as the VRU evicts prefetched data from the cache before the VLU has a chance to access it. Simply reducing the queue size to 64 turns out to be an effective throttling technique. However, doing so can also over-constrain the refill unit, as seen with `vvadd-byte` and `idct`. In general, the command queue size is not a robust technique for throttling the refill unit since the distance that it can run ahead depends on both the vector-length and the percentage of queued commands that are vector-loads.

The *Total-Distance* and *Set-Distance-24* results in Table 2 evaluate directly throttling the distance that the refill unit is allowed to run ahead of the VLU. These schemes use the distance counting mechanism described in Section 5.3 with the *Vector-CmdQ* size set to 1024. When only one count is used to limit the VRU distance to 768 cache lines (75% of the cache) or 100 cache lines (the memory bandwidth delay product), most benchmarks perform very well. However, the refill unit still runs too far ahead for `transpose512` since its accesses tend to map to a single set in the cache.

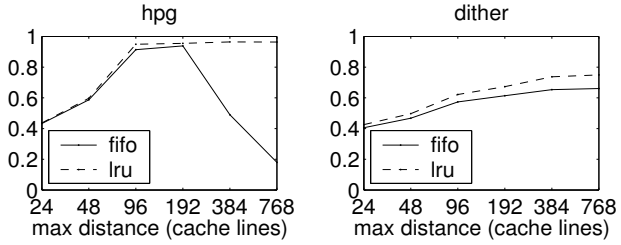


Figure 6: Cache replacement policy interaction with refill distance throttling. The refill unit uses the *Set-Distance-24* throttling mechanism, and the x-axis increases the maximum refill distance by varying the number of distance-sets from 1 to 32. The y-axis shows performance relative to the *pseudo-ideal* SCALE configuration. The results are for a memory latency of 400 cycles.

Memory latency (cycles):	4	25	50	100	200	400	800
SCALE Decoupled Vector Processor							
VEU/VLU/VSU/VRU-CmdQs	8						
VLDQ/LDQ/SAQ	32 *(8)*						
Vector-CmdQ	4	8	16	32	64	128	256
Throttle distance per set	24						
Distance-sets	1	1	1	2	4	8	16
Non-Blocking Cache: 32 KB, 32-way set-assoc., FIFO replacement							
Replay queue entries	32 *(8)*						
Pending store buffer entries	32 *(8)*						
Primary miss tags	16*(4)*	32*(8)*	64*(16)*	64*(16)*	128*(32)*	256*(64)*	512*(128)*

Table 3: Restricted configuration resource parameters with scaling memory latency. Per-lane and per-bank parameters are shown in parentheses.

Breaking the distance count into sets allows the VRU to run further ahead when access patterns use the cache uniformly, while still not exceeding the capacity when access patterns hit a single set. The *Set-Distance-24* results in Table 2 use a varying granularity of distance-sets for the bookkeeping while always limiting the maximum distance in any set to 24. With a memory latency of 400 cycles, only around 4–8 distance-sets are needed.

The VRU distance throttling assumes that the cache eviction order follows its refill probe pattern, but a FIFO replacement policy can violate this assumption and cause refill/access decoupling to breakdown. This occurs when a refill probe hits a cache line which was brought into the cache long ago and is up for eviction soon. This is a problem for the `hpg` benchmark which has distant cache line reuse as it processes columns of the input image. Figure 6 shows that its performance initially increases as the refill unit is allowed to run further ahead, but then falls off dramatically when it goes far enough to evict lines before they are reused. `dither` frequently reuses its pixel error buffers, but they are periodically evicted from the cache, causing the VLU to get a miss even though the corresponding refill unit probe got a hit. Figure 6 shows how an LRU replacement policy fixes these breakdowns in refill/access decoupling. To simplify the cache hardware, an approximation of LRU would likely be sufficient.

6.4. Memory Latency Scaling

We now evaluate a DVMR configuration with reasonable decoupling queue sizes, and show how performance scales with memory latency and processor frequency. Table 3 shows the configuration parameters, and Figure 7a shows how performance scales as the latency of the 8 B/cycle main memory increases from 4 to 800 cycles. To tolerate the increasing memory bandwidth-delay product, only the Vector-CmdQ size, the number of primary miss tags, and the distance-sets were scaled linearly with the memory latency. We found it unnecessary to scale the number of primary miss tags per bank between the 50 and 100 cycle latency configurations because applications tend to use these hard-partitioned cache bank resources more uniformly as their number increases.

As shown in Figure 7a, eight of the benchmarks are able to maintain essentially full throughput as the memory latency increases. This means that the amount of in-flight data reaches each benchmark's achievable memory bandwidth (Table 1) multiplied by 800 cycles—up to 6.25 KB. The performance for benchmarks which lack sufficient structured access parallelism (`dither` and `fft`) tapers as the memory latency increases beyond their latency tolerance. `hpg` performance drops off due to the cache's FIFO eviction policy, and `transpose512` can not tolerate the long latencies since the refill distance is limited to the size of one set.

In Figure 7b, we evaluate the effect of increasing the processor clock frequency by $4\times$ such that the memory latency also increases by $4\times$ and the bandwidth becomes 2 B/cycle. We retain the same parameters listed in Table 3 for operating points with equal bandwidth-delay products. The results show that the benchmarks which were limited by memory at 8 B/cycle still attain the same performance, while those which were previously limited by compute or cache bandwidth improve with the increased frequency.

6.5. Cache Address Bandwidth

The cache probes by the VRU increase energy through additional tag checks, although this is partially mitigated by vector cache accesses which amortize address bandwidth over many elements. For example, `idct` running on DVM without segments requires only 0.63 cache accesses per load element due to efficient unit-stride vector accesses. Adding segments reduce this to 0.19 accesses per load element. DVMR with segments results in an increase to 0.28 accesses per load element, but this is still far more energy efficient than DSM (which requires one access for every load element).

VRU cache probes can also impact performance by increasing the number of bank conflicts. Section 6.2 illustrated that for applications with many misses, any performance impact due to bank conflicts is drastically outweighed by the performance benefit of refill/access decoupling. Bank conflicts are more of a concern for applications which always hit in cache. Several of the benchmarks were tested with datasets that completely fit in cache for the DVM and DVMR configurations. The performance overhead of DVMR ranged from less than 2% for `vvadd-byte`, `vertex`, and `idct` to approximately 10% for `vvadd-word` and `transpose` for certain matrix dimensions. It is important to remember that for applications with even a few misses, refill/access decoupling quickly mitigates this performance overhead. Decreasing the VRU priority in the cache arbiter can also drastically

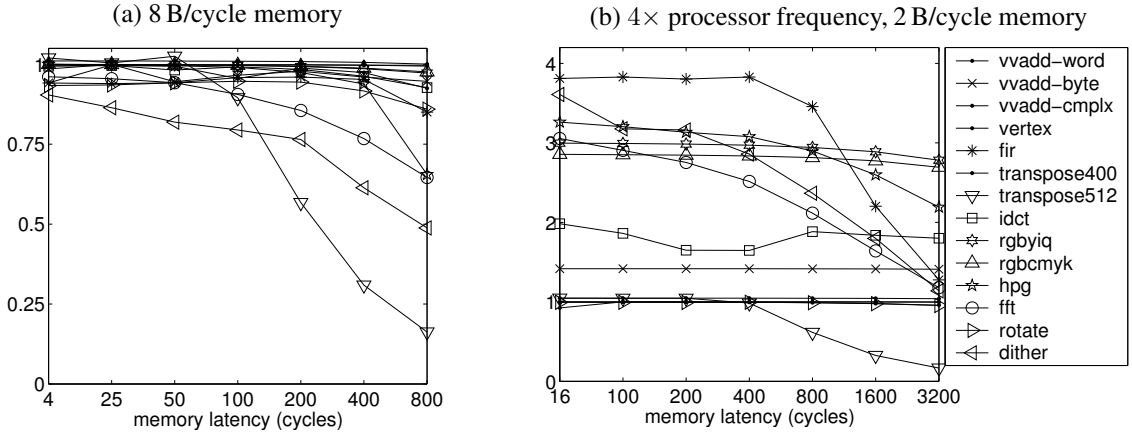


Figure 7: Performance scaling with increasing memory latency. The y-axis shows performance relative to the *pseudo-ideal* SCALE configuration (with 8 B/cycle memory). In (b), the processor frequency is increased by $4\times$ while retaining the same memory system.

reduce this performance overhead, but this would decrease the effectiveness of refill/access decoupling. An interesting future research direction is to investigate adaptive priority schemes which adjust the VRU priority based on miss rate.

7. Related Work

There have been many proposals to improve performance of large bandwidth-delay memory systems. We describe techniques for both scalar and vector machines.

Superscalar processors use register renaming and out-of-order execution to overlap multiple accesses to a non-blocking cache [15]. In-flight access management state includes the instruction window and the memory reorder queue, in addition to the cache's primary miss tags and replay queue. Reserved element buffering is provided by the multiported renamed register file. These structures are very expensive, and so current superscalars are typically limited to less than a hundred elements in flight.

Scalar processors are usually augmented with various software or hardware prefetching schemes, to help saturate large bandwidth-delay memory systems [30]. Effective **software prefetching** for scalar machines requires intimate and implementation-specific details of the underlying memory hierarchy and memory pipeline timing, and requires complex compiler algorithms to avoid insertion of excess prefetch instructions [24]. The resulting prefetch schedules are not performance-portable across different microarchitectures.

Hardware prefetching usually falls into one of two categories: miss-based prefetching and runahead prefetching. **Miss-based prefetching** schemes, including stream buffers [14, 22], attempt to predict future cache misses based on past cache miss address patterns. Stream buffers are more complex than refill/access decoupling as they require stride predictors, and usually have separate line storage buffers to prevent cache pollution on prefetch mispredictions. Miss-based prefetchers are only effective on longer vectors as they require one or more misses to detect a pattern and so are slower in starting to prefetch and waste memory bandwidth on mispredictions past the end of the vector. In contrast, refill/access decoupling is effective in situations where there are many short vector accesses as it eliminates speculation, begins fetching at an earlier point in the stream, and preloads only the exact data required. Also, refill/access decoupling can handle any number of independent streams as it does not maintain any per-stream state. Stream buffers, however, have the advantage of avoiding tag checks if the data is already in cache whereas the VRU must probe the tags once per cache line. Stream buffers can also prefetch scalar accesses whereas refill/access decoupling only prefetches vectors.

Perhaps the previous techniques most similar in spirit to refill/access decoupling are the various forms of scalar **runahead processing**. Baer and Chen [2] described a preloading scheme that predicts the path a program will take and the addresses that will be accessed at each program counter value, then tries to preload any missing cache lines. The look-ahead PC is throttled to not run ahead by more than a fixed number of cycles, partly to reduce cache pollution as in our decoupled refill/access decoupling scheme but also to limit the amount by which instruction stream prediction can go astray. The run-ahead processor [7] and later out-of-order variants [21] provide alternative implementations of the same general concept, but only switch into run-ahead after a cache miss. These scalar runahead schemes require complex hardware and are less effective than vector refill/access decoupling, which requires no speculation and amortizes prefetch cache probes over multiple elements.

Decoupled architectures were initially developed to tolerate memory latency in scalar machines by separating address generation and memory access from compute operations [26, 27]. Decoupled architectures are able to generate large numbers of in-flight elements but require extensive reserved element buffering in the decoupling queues, and also primary miss tags and replay queues to track pending element accesses if a non-blocking cache is added.

Several early **vector machines** designed around existing scalar architectures had caches. Both the IBM 3090 [28] and the VAX vector machines [3] would prefetch lines into the cache when cache misses were encountered during a unit-stride access, but did not prefetch past the current instruction. Fu and Patel [12] extended this by adding a strided prefetch for misses on strided vector operations. The Taran-

tula design focused on providing high bandwidth to a large level 2 cache [8]. A software vector prefetch instruction was added (apparently because regular vector instructions could not saturate main memory bandwidth [8]), which shares with scalar software prefetch the unfortunate side effect of exposing many implementation-specific details of the memory system to software. The Cray SV1 added a vector data cache with single word lines to avoid wasting bandwidth for non unit-stride accesses, but then required hundreds of pending tags to track enough element misses to saturate the memory bandwidth [5, 11]. The newer Cray X1 design employs 32 byte lines to reduce address bandwidth and tag overhead, and has pooled replay queues implemented with hardware linked lists [1]. The CODE architecture [18] is an alternative implementation of vector decoupling which uses vector register renaming to provide buffer space for decoupled loads. Fully out-of-order vector machines [8, 10] also use renamed vector registers to buffer outstanding requests.

The Imagine stream processor [16] is similar to a vector machine but adds stream load/store instructions, similar to the vector segment load/store instructions introduced in this work. Imagine stream load/stores, however, are translated into strided accesses at the memory controller which then tries to dynamically reaggregate these into unit-stride bursts for the DRAM. As with earlier vector supercomputers [29], Imagine has a two-level hierarchical vector (stream) register file, and uses the extra capacity to provide reserved element buffering for memory traffic as well as to capture foreground register spills. A recently proposed extension [13] would allow more flexible access to the stream register file to improve its ability to exploit some forms of temporal locality, but this adds interconnect similar to that needed for a more flexible banked cache and the programming model appears to be considerably more complex than for regular memory accesses.

The RSVP stream processor [4] has decoupled stream load/store engines which feed data directly into computations rather than into intermediate vector registers. Reserved element buffering is provided by FIFO buffers in the streaming engines. To exploit some limited forms of temporal locality, limited stream indexing is provided to access neighboring elements in a stream.

8. Conclusion

We have described the considerable costs involved in supporting large numbers of in-flight memory accesses, even for vector architectures running applications that have large quantities of structured memory access parallelism. Cache refill/access decoupling dramatically reduces the cost of saturating a high-bandwidth cached memory system by pre-executing vector memory instructions to fetch data into the cache just before it is needed, avoiding most per-element costs. We also introduced new vector segment instructions, which encode common two-dimensional access patterns in a form that improves memory performance and reduces the overhead of refill/access decoupling. Together these techniques enable high-performance and low-cost vector memory systems to support the demands of modern vector architectures.

References

[1] D. Abts, S. Scott, and D. J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *IPDPS*, Apr 2003.

[2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *ICS*, 1991.

[3] D. Bhandarkar and R. Brunner. VAX vector architecture. In *ISCA-17*, 1990.

[4] S. Ciricescu et al. The reconfigurable streaming vector processor (RSVP). In *MICRO-36*, pages 141–150, Dec 2003.

[5] Cray. Vector and scalar data cache for a vector multiprocessor. U.S. Patent 6,665,774, Dec 2003.

[6] D. H. Brown Associates, Inc. Cray launches X1 for extreme supercomputing, Nov 2002.

[7] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. ICS*, pages 68–75, Jul 1997.

[8] R. Espasa et al. Tarantula: a vector extension to the Alpha architecture. In *ISCA-29*, 2002.

[9] R. Espasa and M. Valero. Decoupled vector architectures. In *HPCA-2*, Feb 1996.

[10] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *MICRO-30*, Dec 1997.

[11] G. Faanes. A CMOS vector processor with a custom streaming cache. In *Proc. Hot Chips 10*, Aug 1998.

[12] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. In *ISCA-18*, May 1991.

[13] N. Jayasena et al. Stream register files with indexed access. In *HPCA-10*, page 60, Feb 2004.

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990.

[15] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, Mar/Apr 1999.

[16] B. Khailany et al. Imagine: Media processing with streams. *IEEE Micro*, Mar/Apr 2001.

[17] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh. A hardware overview of SX-6 and SX-7 supercomputer. *NEC Research & Development Journal*, 44(1):2–7, Jan 2003.

[18] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *ISCA-30*, Jun 2003.

[19] R. Krashinsky et al. The vector-thread architecture. In *ISCA-31*, Jun 2004.

[20] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, pages 81–87, May 1981.

[21] O. Mutlu et al. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, Nov/Dec 2003.

[22] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA-21*, pages 24–33, 1994.

[23] R. M. Russel. The Cray-1 Computer System. *CACM*, 21(1):63–72, Jan 1978.

[24] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. In *ISCA-24*, pages 264–273, 1997.

[25] S. Scott. Supercomputing past, present, and future. ICS-18 Keynote Address, Jun 2004.

[26] J. E. Smith. Decoupled access/execute computer architecture. In *ISCA-9*, 1982.

[27] J. E. Smith. Dynamic instruction scheduling and the Astronautics ZS-1. *IEEE Computer*, 22(7):21–35, 1989.

[28] S. G. Tucker. The IBM 3090 system: An Overview. *IBM Systems Journal*, 25(1):4–19, 1986.

[29] T. Watanabe. Architecture and performance of NEC supercomputer SX system. *Parallel Computing*, 5:247–255, 1987.

[30] S. P. V. Wiel and D. J. Lilja. When caches aren't enough: Data prefetching techniques. *IEEE Computer*, 30(7):23–30, Jul 1997.