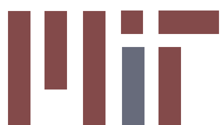
Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

Cache Refill/Access Decoupling for Vector Machines

**Christopher Batten, Ronny Krashinsky,
Steve Gerding, Krste Asanović**

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
December 8, 2004

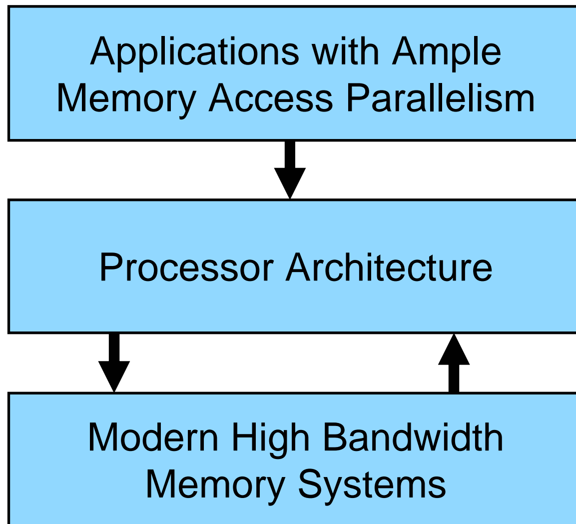


Cache Refill/Access Decoupling for Vector Machines

- **Intuition**
  - Motivation and Background
  - Cache Refill/Access Decoupling
  - Vector Segment Memory Accesses
- **Evaluation**
  - The SCALE Vector-Thread Processor
  - Selected Results

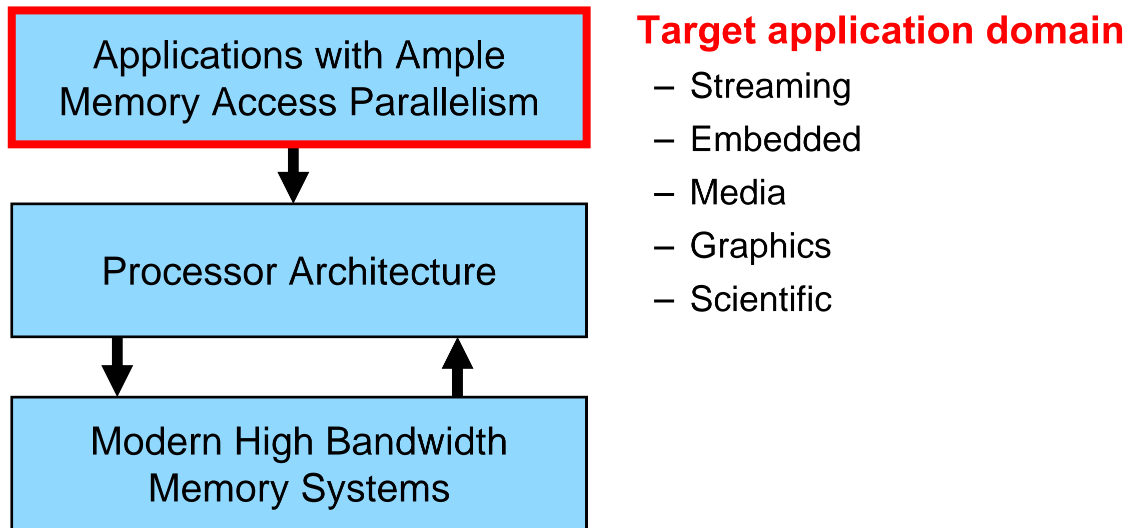
My talk will have two primary parts. First, I will give some motivation and background before discussing the two key techniques that we are proposing in this work. Namely, cache refill/access decoupling and vector segment memory accesses. In the second part of the talk, I will briefly evaluate a specific implementation of these ideas within the context of the SCALE vector-thread processor.

Turning access parallelism into performance is challenging



I would like to begin with two key observations.

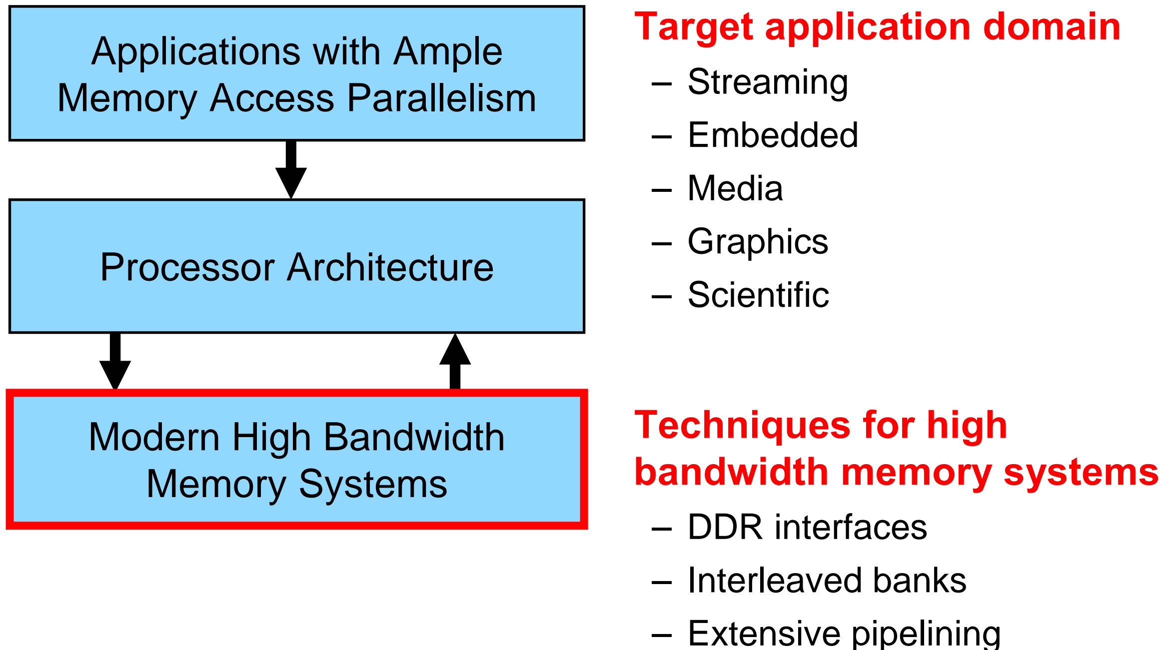
Turning access parallelism into performance is challenging



Target application domain

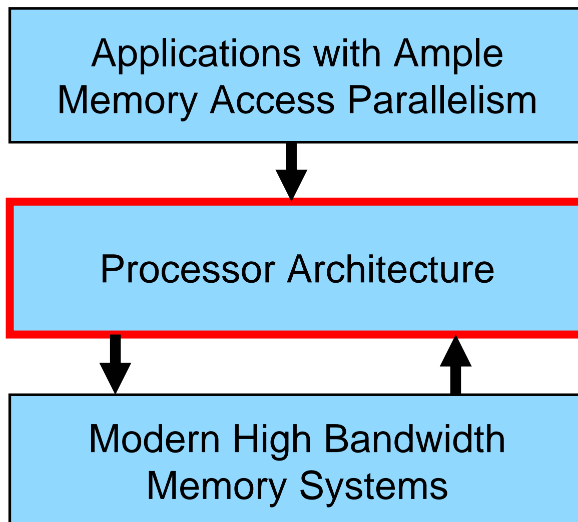
- Streaming
- Embedded
- Media
- Graphics
- Scientific

The first is that many applications have ample memory access parallelism and by this I simply mean that they have many independent memory accesses. This is especially true in many streaming, embedded, media, graphics, and scientific applications.



The second observation is that modern memory systems have relatively large bandwidths due to several reasons including high speed DDR interfaces, numerous interleaved banks, and extensive pipelining.

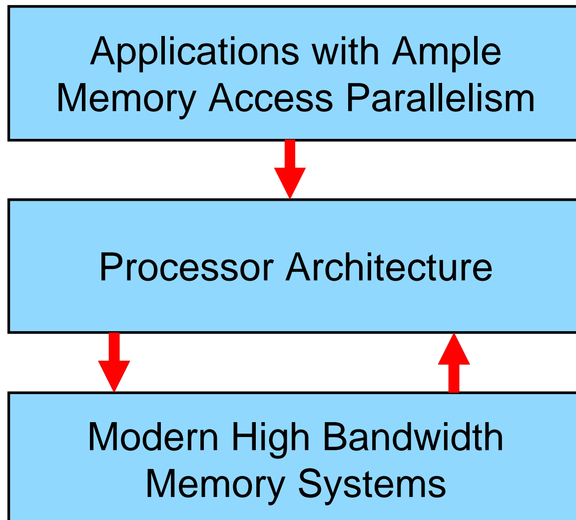
Turning access parallelism into performance is challenging



Many architectures have difficulty turning memory access parallelism into performance since they are unable to fully saturate their memory systems

Ideally, an architecture should be able to turn this memory access parallelism into performance by issuing many overlapping memory requests which saturate the memory system. Unfortunately, there are two significant challenges which make it difficult for modern architectures to achieve this goal.

Turning access parallelism into performance is challenging

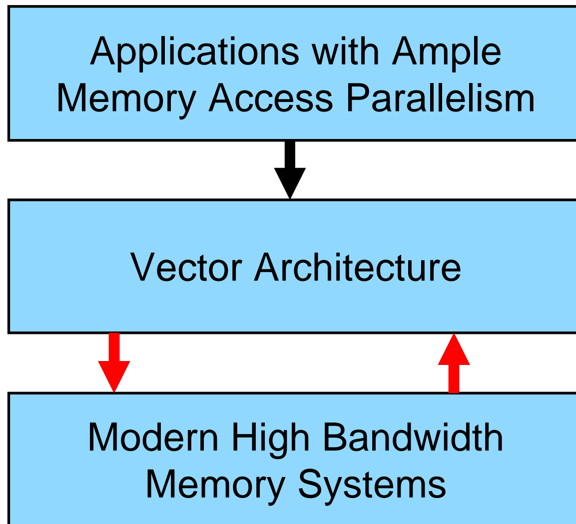


Memory access parallelism is poorly encoded in a scalar ISA

Supporting many in-flight accesses is very expensive

The first is at the application/processor interface – scalar ISAs poorly encode memory access parallelism making it difficult for architectures to exploit this parallelism. The second challenge is at the processor/memory system interface since supporting many accesses in-flight in the memory system is very expensive.

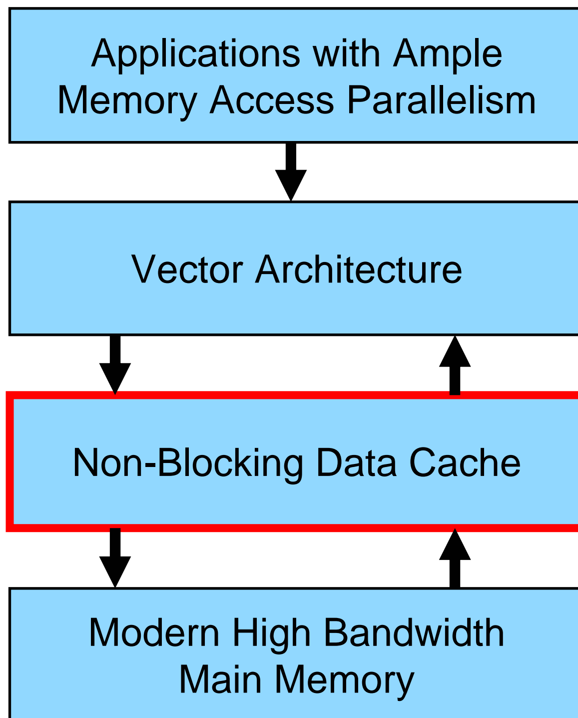
Turning access parallelism into performance is challenging



Supporting many in-flight accesses is very expensive

Our group is specifically interested in vector architectures. Vector architectures are nice since vector memory instructions better encode memory access parallelism, but even vector architectures require a great deal of hardware to track many in-flight accesses

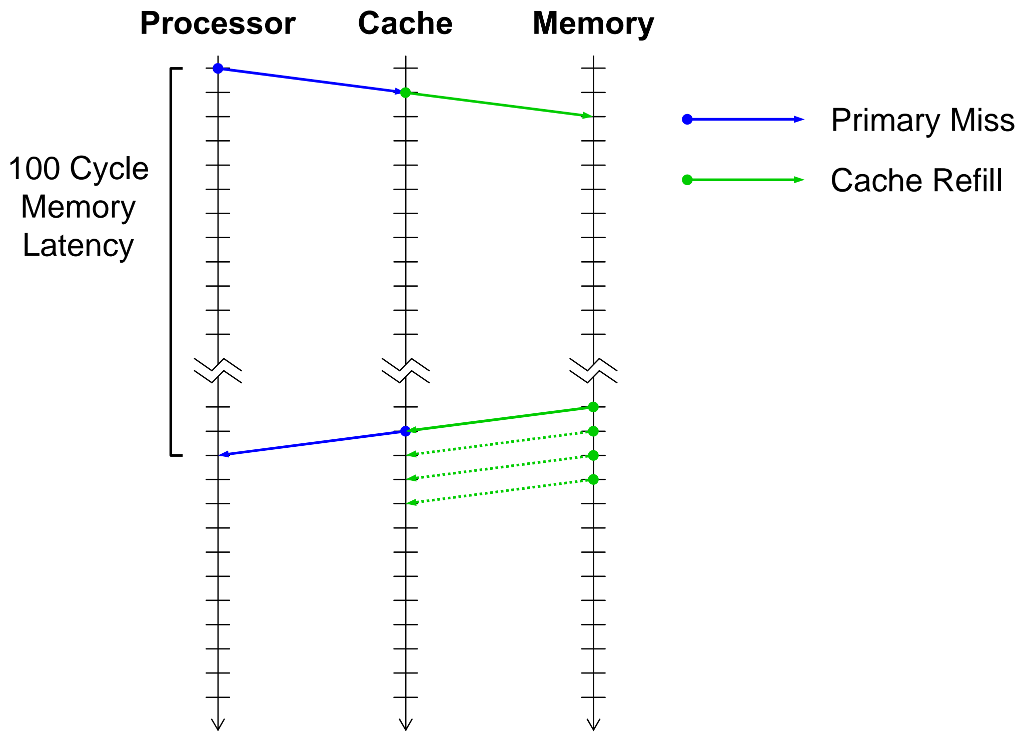
Turning access parallelism into performance is challenging



A data cache helps reduce off-chip bandwidth costs at the expense of additional on-chip hardware

Furthermore, modern vector machines often include non-blocking data caches to exploit reuse and reduce expensive off-chip bandwidth requirements. Unfortunately, these non-blocking caches have several resources which scale with the number of in-flight accesses and this increases the cost for applications which do not fit in cache or have a significant number of compulsory misses. To get a better feel for these hardware costs we first examine how many in-flight accesses are required to saturate modern memory systems.

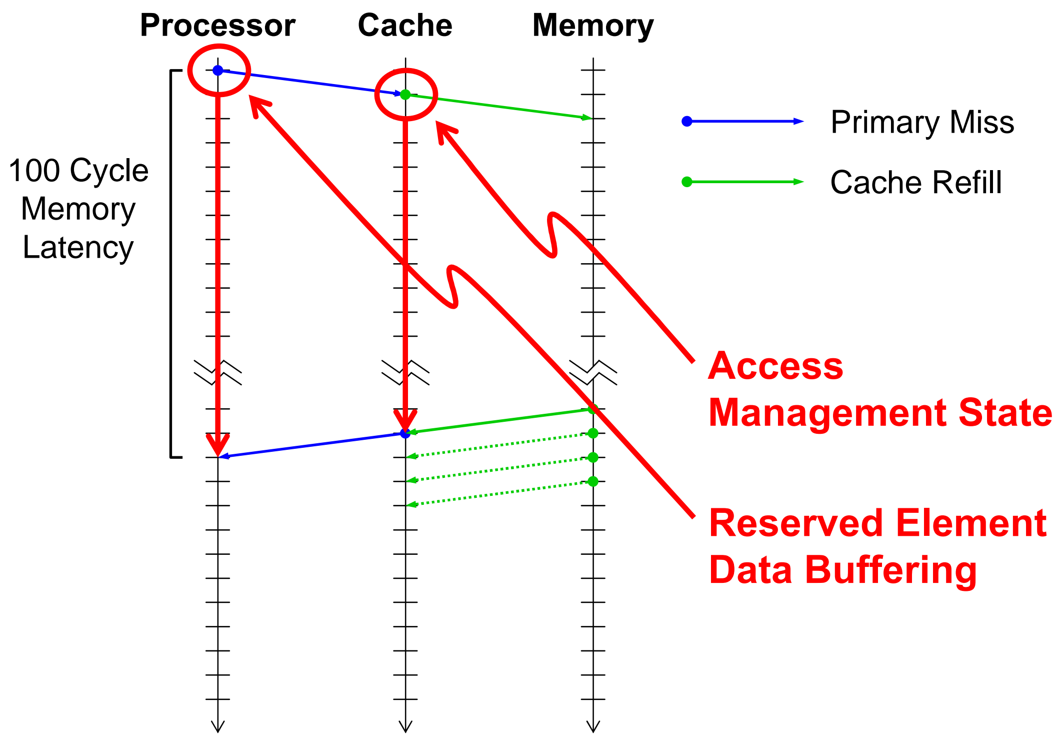
Each in-flight access has an associated hardware cost



This is a timeline of requests and responses between the processor and the cache and between the cache and main memory. Each tick represents one cycle, and we assume that the processor to cache bandwidth is two elements per cycle while the cache to main memory bandwidth is one element per cycle. The blue arrow indicates a processor load request for a single element. For this example, we assume the processor is accessing consecutive elements in memory and that these elements are not allocated in the cache.

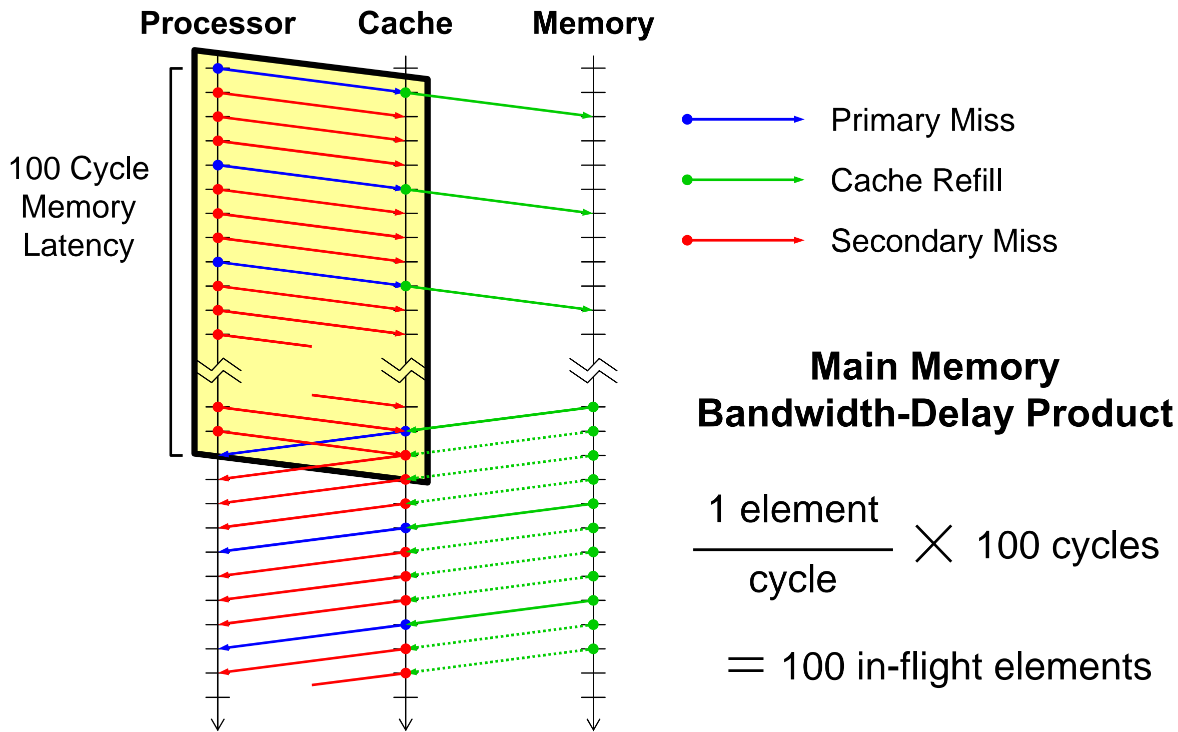
Thus the load request misses in the cache and causes a cache refill request to be issued to main memory. Some time later, main memory returns the load data as well as the rest of the cache line. We assume that the cache line is four elements. The cache then writes the returned element into the appropriate processor register.

Each in-flight access has an associated hardware cost



Each in-flight access requires two pieces of hardware. The first is some reserved element data buffering in the processor. This is some storage that the processor sets aside so that the memory system has a place to write data when it returns. We need this because we are assuming that the memory system cannot be stalled, which is a reasonable assumption with today's heavily pipelined memory systems. The second component of the hardware cost is its access management state – this is information stored by the cache about each in-flight element. For example, it includes the target register specifier so that the cache knows into which register to writeback. It is important to note that the lifetime of these resources is approximately equal to the memory latency.

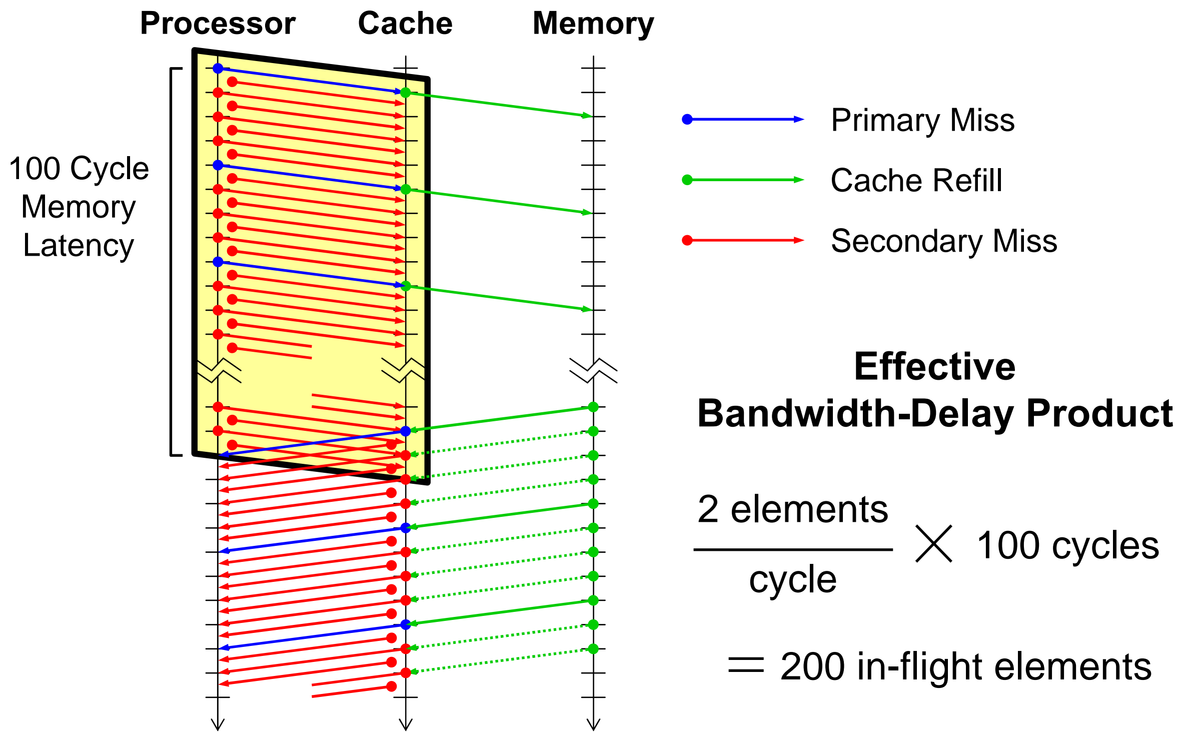
Obviously, the processor cannot wait 100 cycles to issue the next load request if we hope to saturate the memory system ...



Thus on the cycle after the first request, the processor should make another request for the second element in the array. This request will also miss in the cache, but it is to the same cache line as the first request – and that cache line is already in flight. Thus we do not need to send a new refill request to main memory. The first miss to a given cacheline is known as a primary miss (show here in blue), while additional misses to a cache line which is already in-flight are known as secondary misses (shown here in red). The processor will continue to issue secondary misses until it gets to the next cache line and thus the next primary miss.

So to saturate the main memory bandwidth of one element per cycle we must support one hundred in-flight elements … this means the processor must have 100 elements worth of reserved element buffering and the cache must have 100 elements worth of access management state. In other words, the hardware costs are proportional to the bandwidth-delay product of main memory.

Caches increase the effective bandwidth-delay product



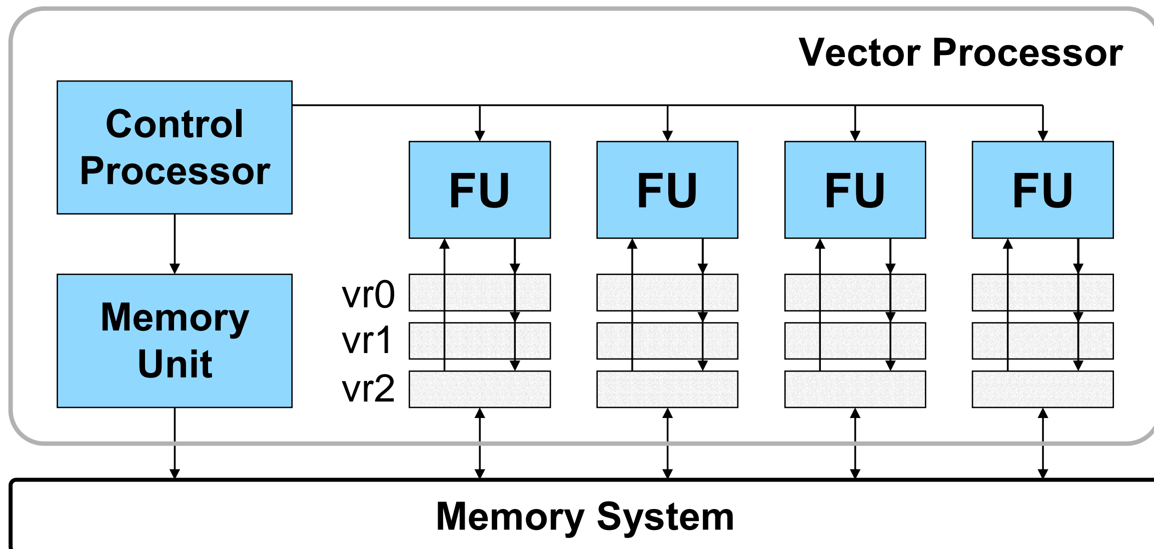
Now lets see what happens if we include reuse. Assume that the processor is loading each element in the array twice and is issuing two requests per cycle to match the cache bandwidth. You would think that this is a good thing since we should be able to exploit the reuse to amplify our memory system bandwidth. While this is true, since these requests are misses, it means that the processor and cache must track twice as many secondary misses before the processor can get to the next primary miss. Thus the processor must have twice as much reserved element data buffering and the cache must have twice as much access management state. In other words, with reuse the hardware cost can be proportional to the effective bandwidth delay product: the cache bandwidth times the access latency in cycles. Modern vector memory systems have large and growing effective bandwidth delay products and thus require expensive non-blocking caches. This brings us to the goal for this work ...

Goal For This Work

Reduce the hardware cost
of non-blocking caches in vector machines while still turning
access parallelism into performance
by saturating the memory system

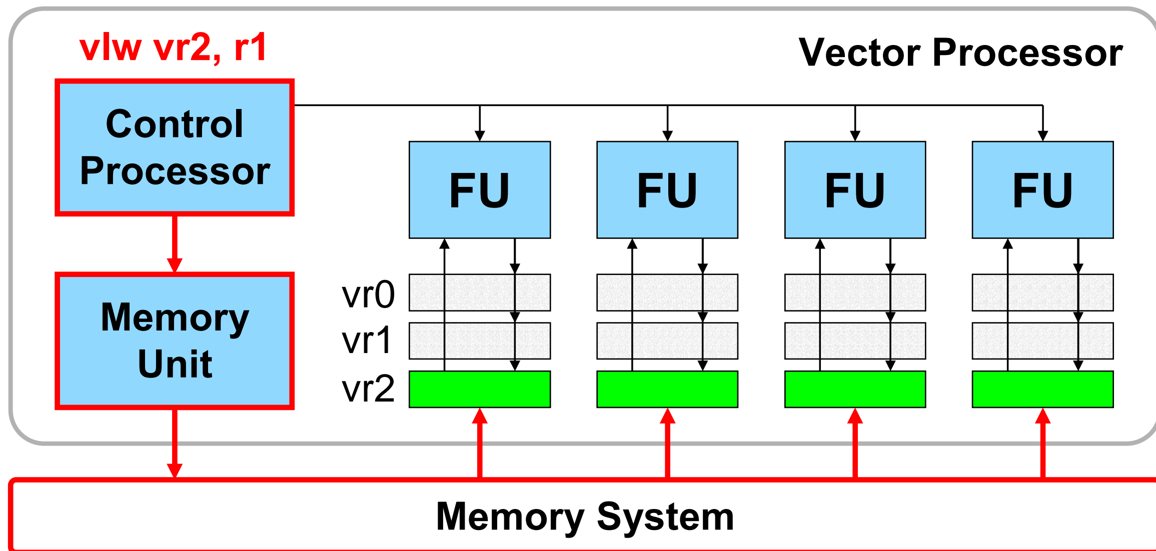
The goal is to reduce the hardware cost of non-blocking caches in vector machines while still turning access parallelism into performance by saturating the memory system.

In a basic vector machine a single vector instruction operates on a vector of data



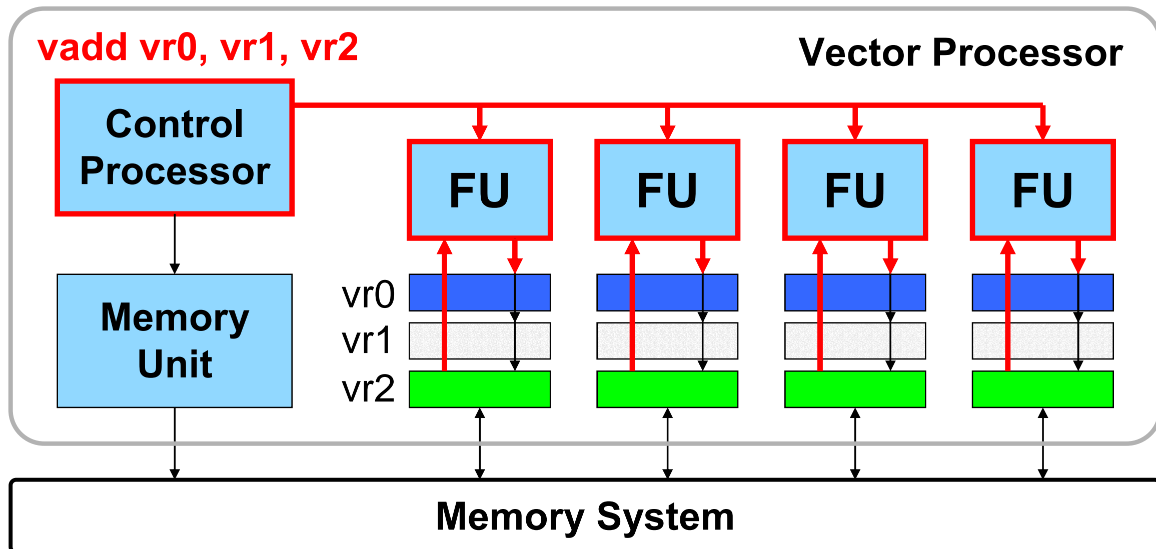
I am going to begin with a very brief refresher of how a basic vector machine works. A vector machine includes a parallel array of functional units, a vector memory unit, and a vector register file. The control processor is in charge of these units and issues commands instructing them what to do.

In a basic vector machine a single vector instruction operates on a vector of data



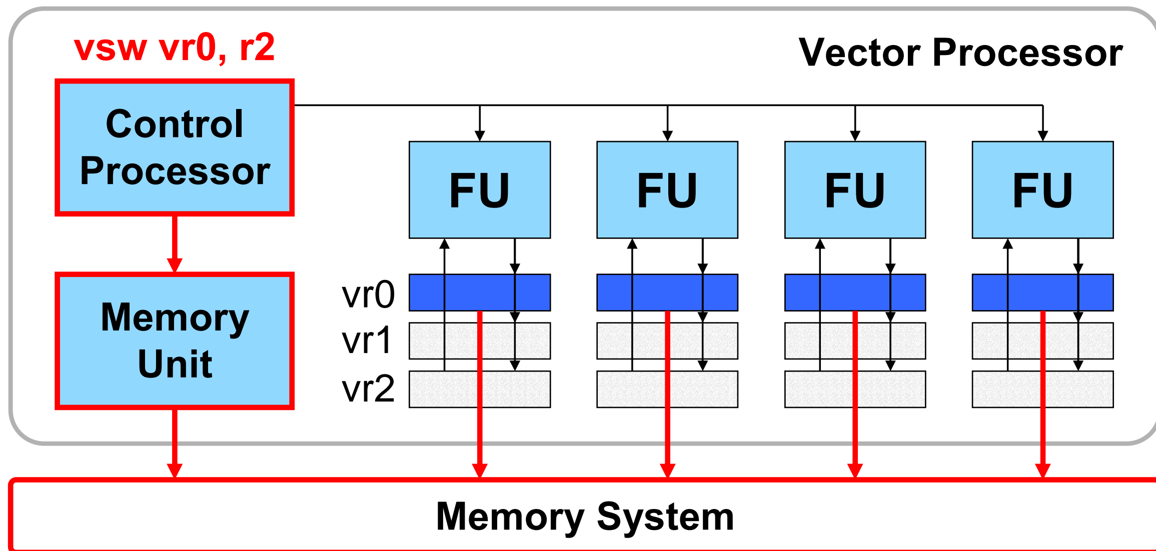
Let's consider a simple example. The control processor issues a vector load word command to the vector memory unit, which then loads a vector of data into vector register two.

In a basic vector machine a single vector instruction operates on a vector of data



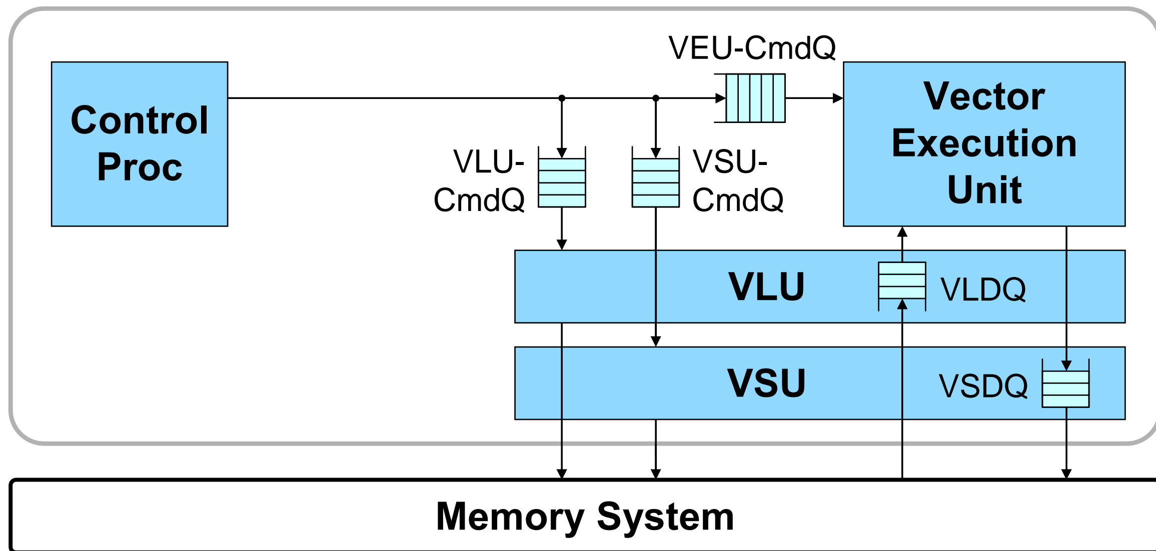
The control processor might then issue a vector add command to the functional units which perform the add and write the result into vector register zero.

In a basic vector machine a single vector instruction operates on a vector of data

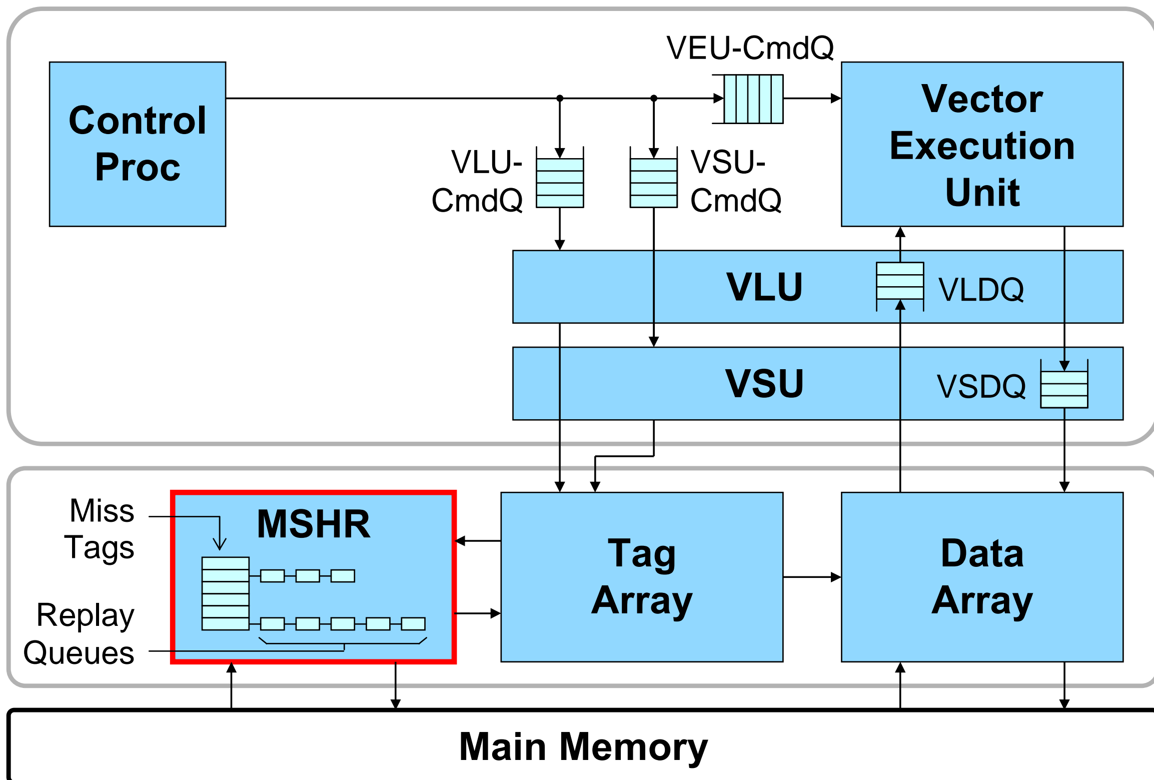


Finally, the control processor issues a vector store word command to the memory unit which moves the data from vector register zero into the memory system. Modern vector machines often include some decoupling between these units as an inexpensive way to tolerate various system latencies.

In a decoupled vector machine the vector units are connected by queues

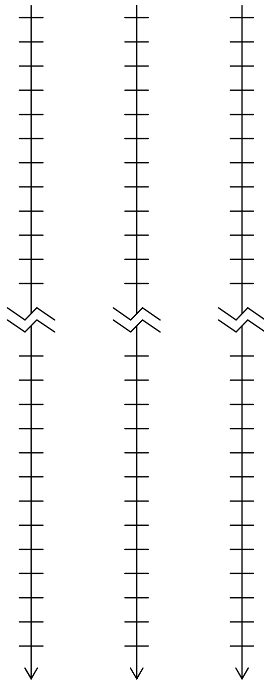


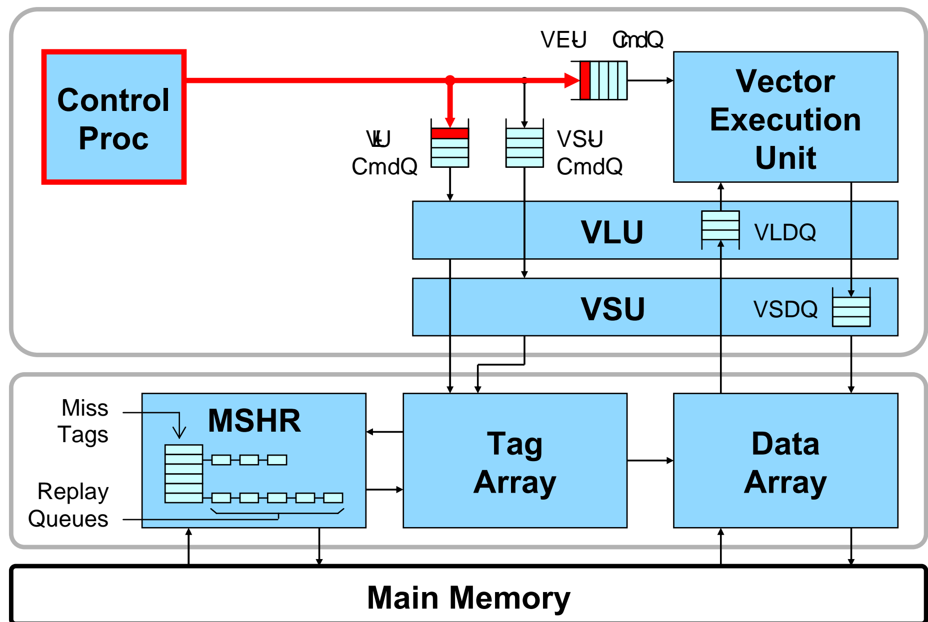
This is a figure of a basic decoupled vector machine and is similar to the previous figure, except that the units are connected by decoupling queues. Additionally, the parallel functional units have been grouped into a vector execution unit, and the memory unit has been divided into a separate vector load unit and vector store unit.



As I mentioned earlier, modern vector machines often include non-blocking data caches to act as bandwidth amplifiers. Shown here is a basic non-blocking cache. In addition to the standard tag and data arrays, the cache includes extra state to track in-flight accesses. This states is located in the miss status handling registers and is composed of two structures: a set of miss tags and one replay queue per miss tag. The function of these structures will become clear as we go through an example.

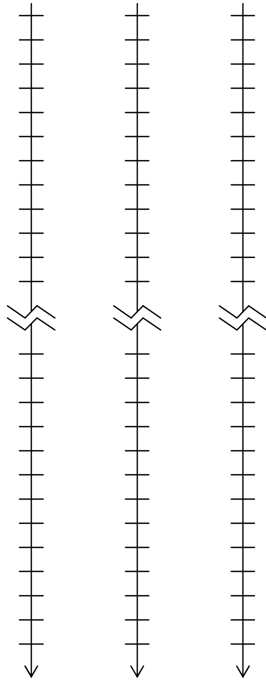
Control processor issues a vector load command to vector units

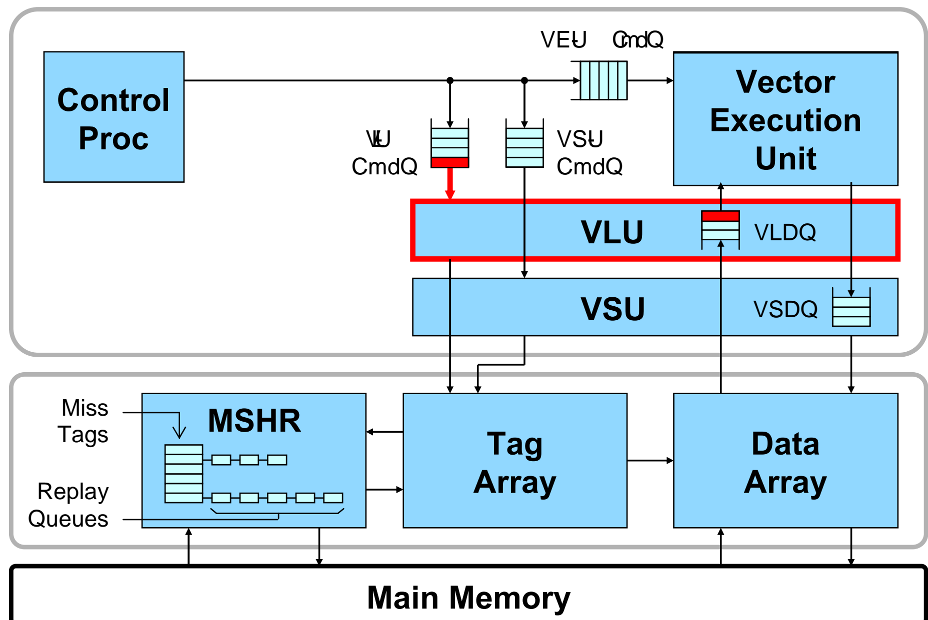




Our example will follow a vector load command through the system. The control processor begins by sending the address portion of the command to the vector load unit and the register writeback portion to the vector execution unit.

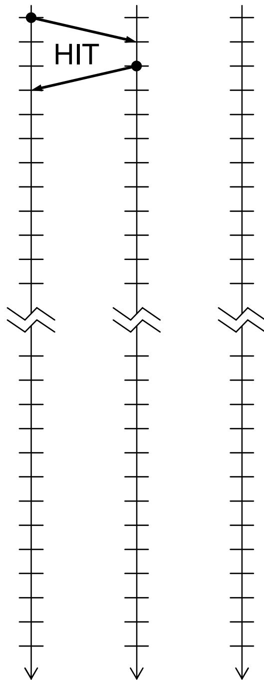
Vector load unit reserves storage in the vector load data queue

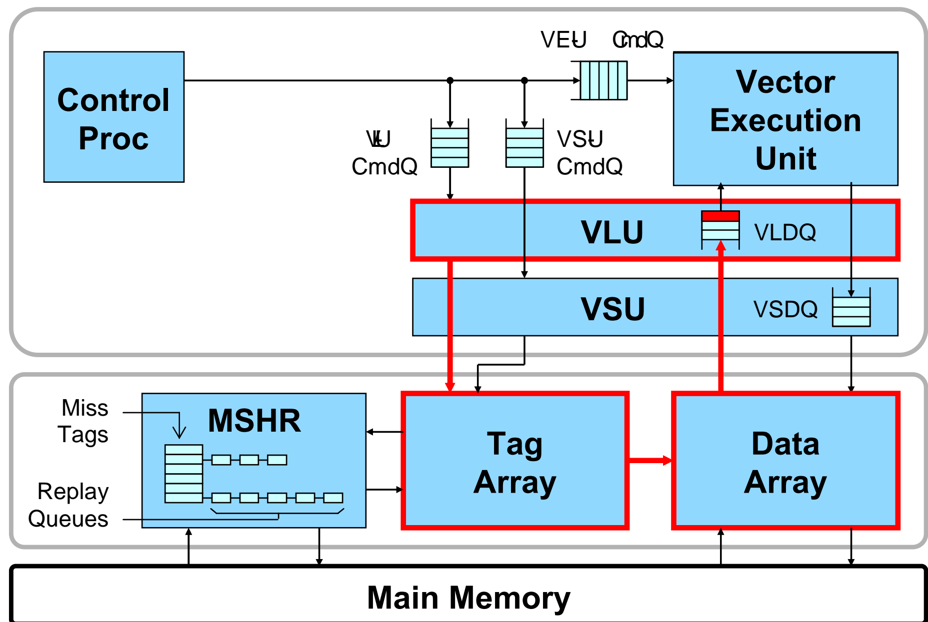




The vector load unit then breaks up the long vector accesses into smaller memory requests. For each request, the vector load unit first allocates a slot in the vector load data queue or VLDQ and then issues this request to the memory system.

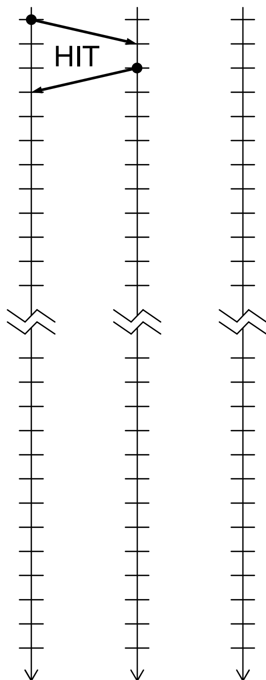
If request is a hit, then data is written into the VLDQ

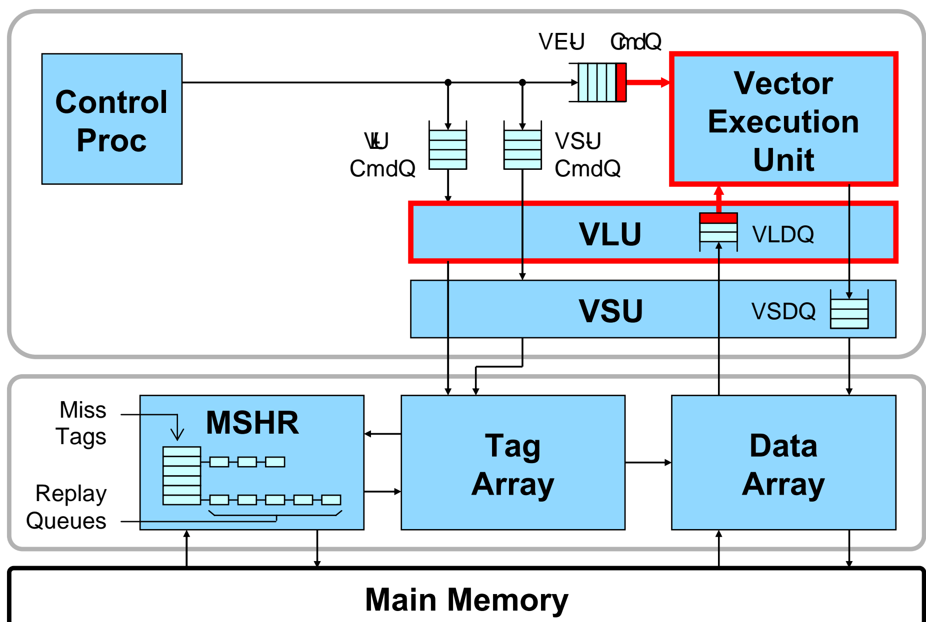




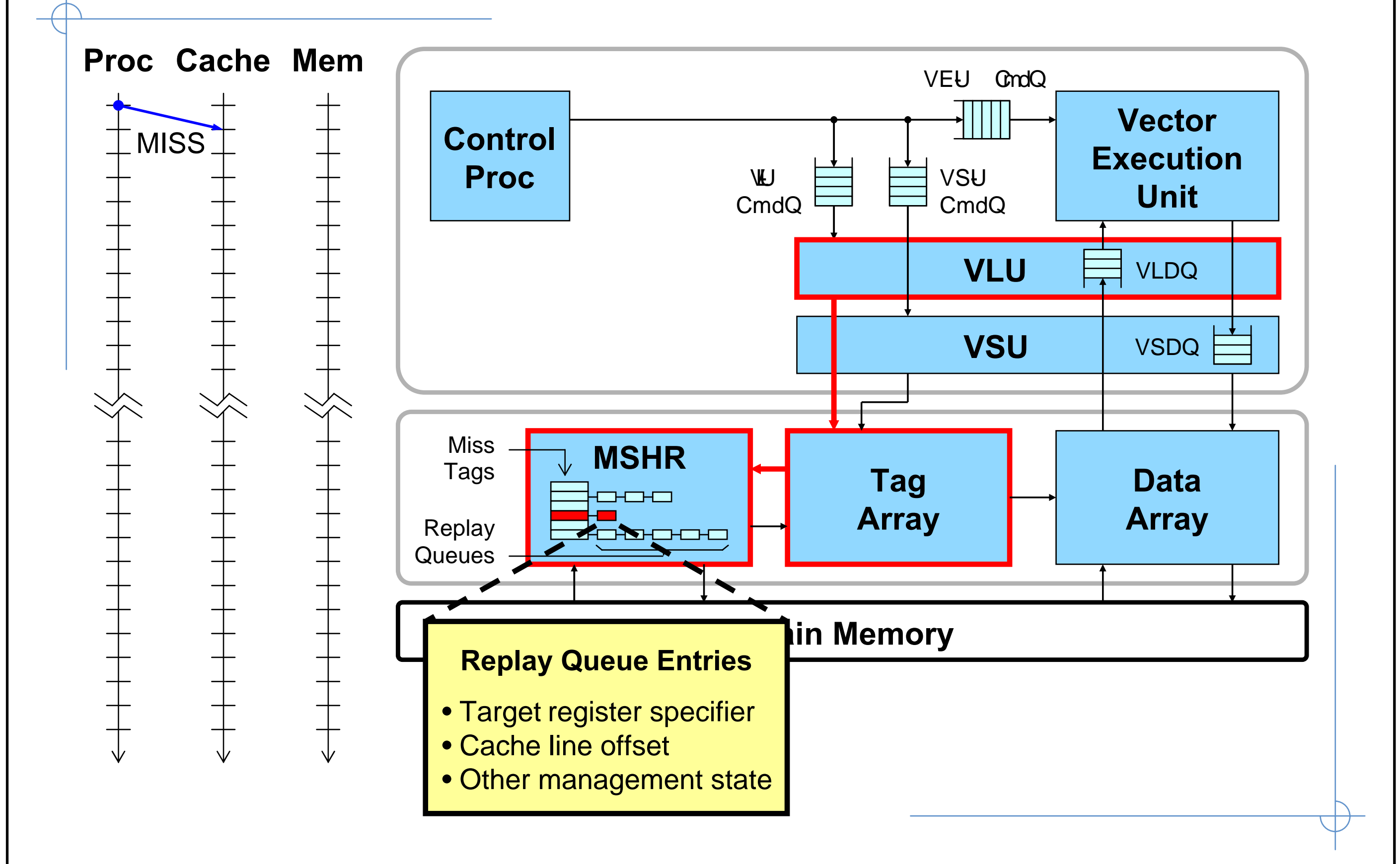
If the request is a hit, then the cache immediately writes the data into the appropriate VLDQ slot.

VEU executes writeback command to move data into architectural register





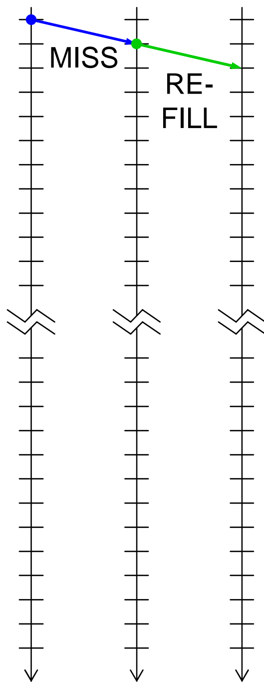
The vector execution unit executes the writeback command and moves the data from the VLDQ into an architecturally visible vector register.

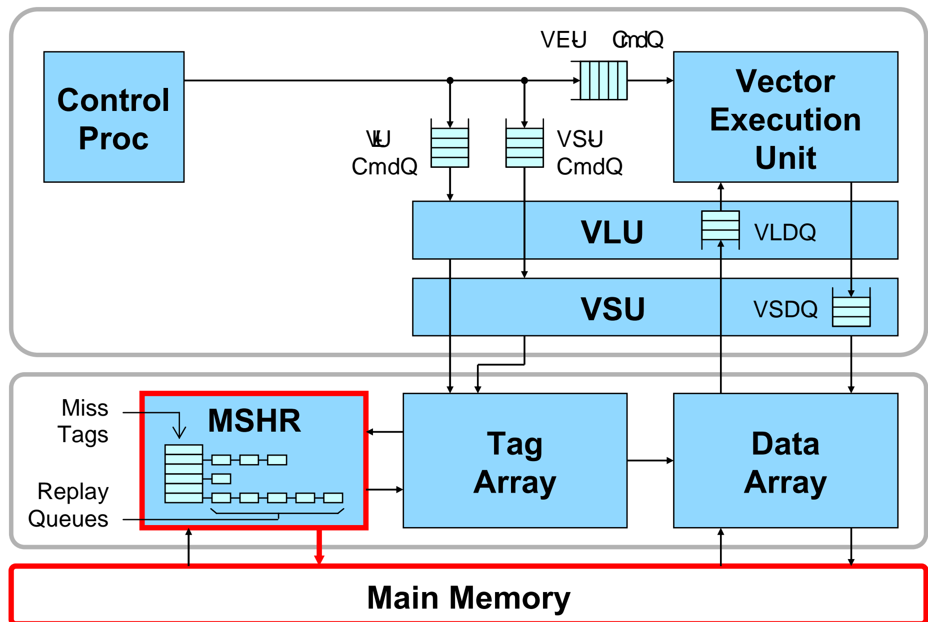


Now lets examine what happens if the VLU request misses in the cache. The cache is then going to allocate a new miss tag and replay queue entry. There is one miss tag for each cache line which is in-flight, and the miss tag simply contains the address of that in-flight cache line. There is one replay queue entry for each pending access. The entry contains various management state including the target register specifier and the cache line offset for the access.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

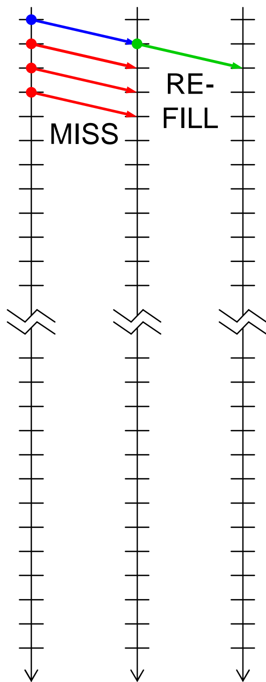
On a primary miss, cache allocates a new miss tag and replay queue entry

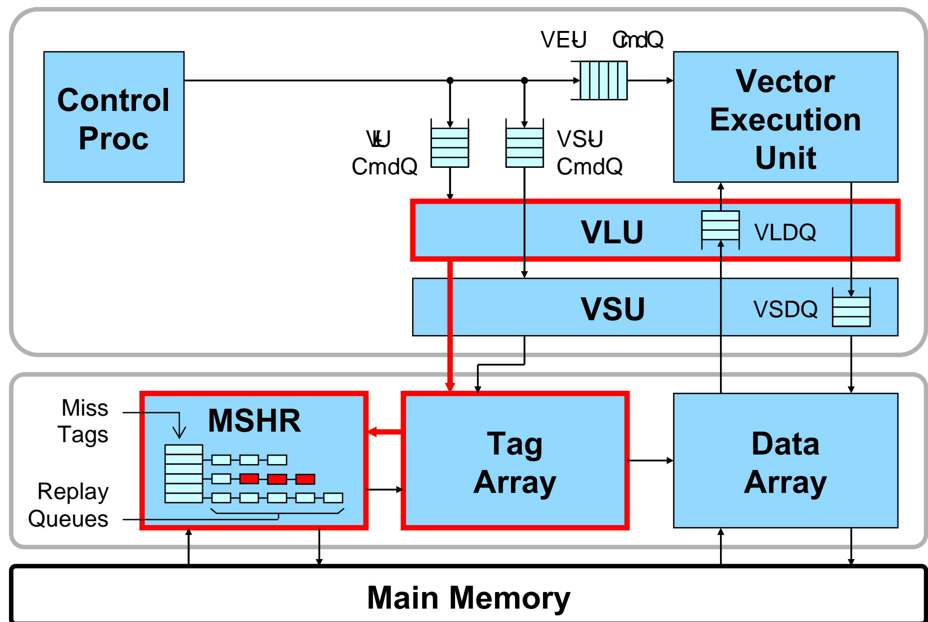




Once the cache has allocated a miss tag and replay queue entry, the cache issues a refill request to main memory.

On a secondary miss, cache just allocates a new replay queue entry

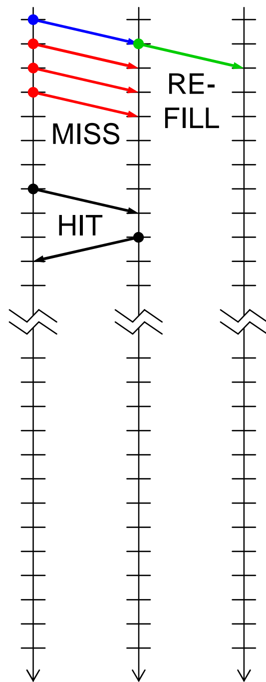


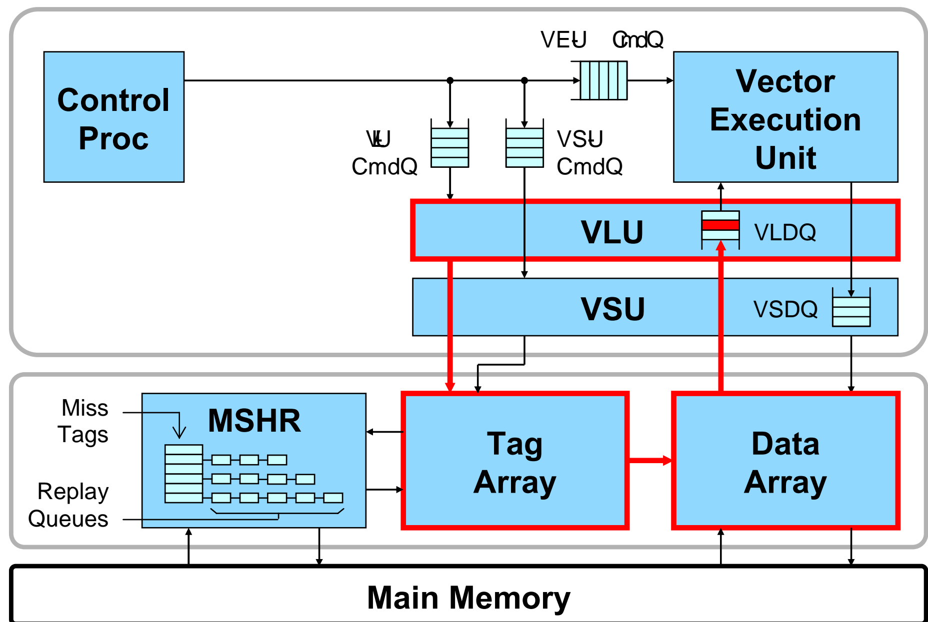


While the refill is in-flight the cache can continue to issue more requests some of which may be secondary misses. These misses are to a cache line which is already in-flight, and therefore they need only be allocated replay queue entries.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 3*7th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

Processor is free to continue issuing requests which may hit in the cache

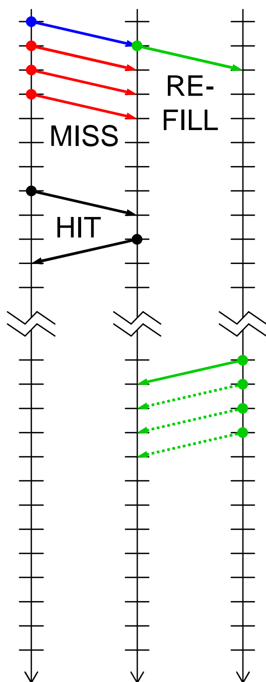


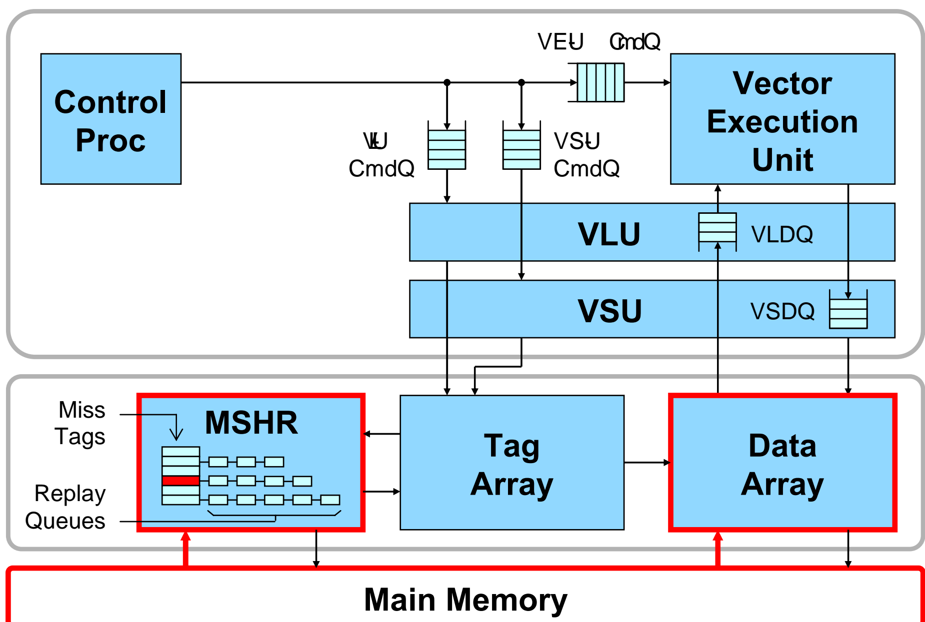


The processor can also get hits to a different cache line while the refill is still in-flight. The VLDQ acts as a small memory reorder buffer since the memory system can write the VLDQ out of order, but the vector execution unit pops data off the VLDQ in order.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

When the refill returns from memory, the cache replays each pending access

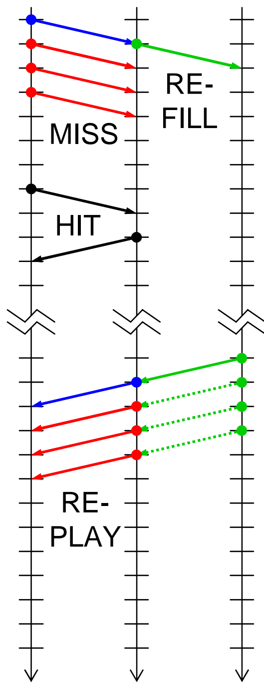


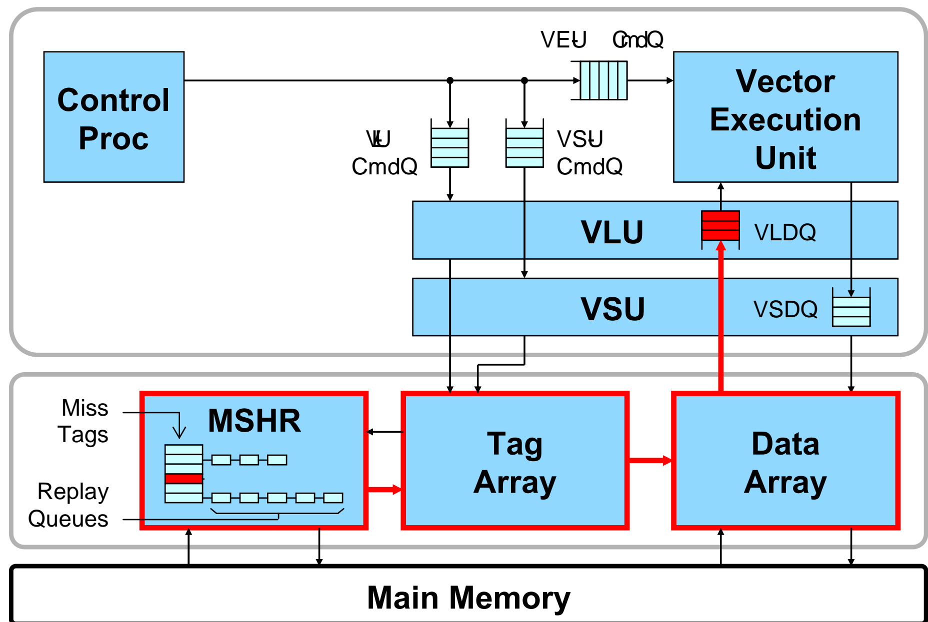


Eventually main memory returns the data to the cache ...

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

When the refill returns from memory, the cache replays each pending access





... and the cache replays the accesses in the corresponding replay queue, sending the data back to the VLDQ.

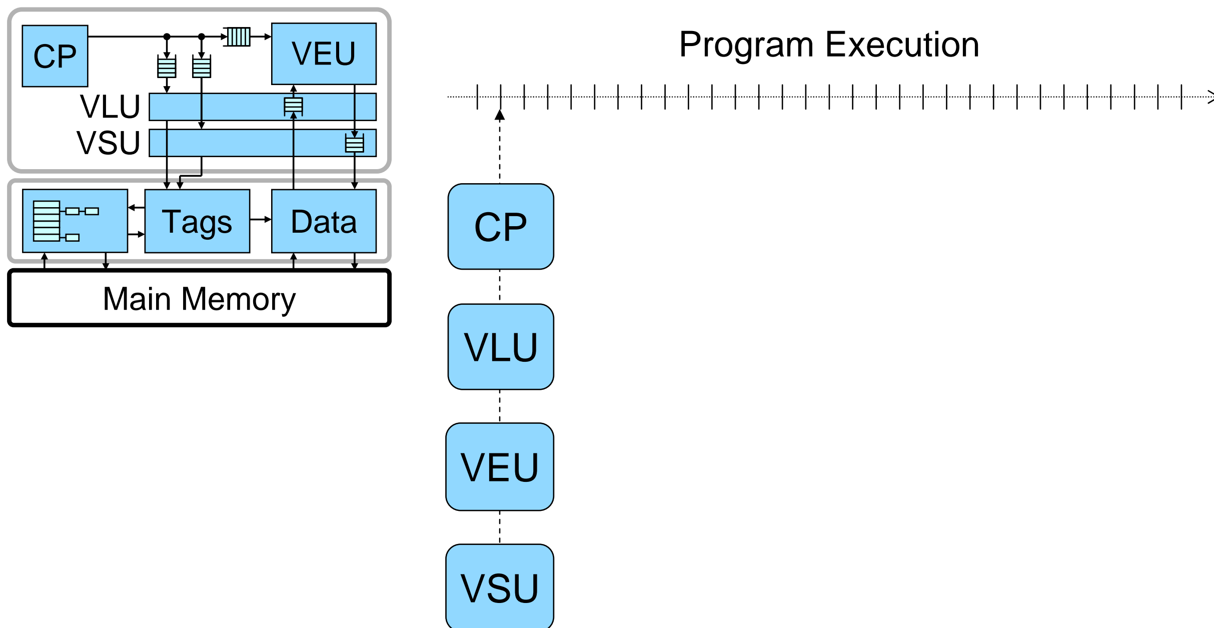
Expensive hardware is required to support many in-flight accesses

Proc Cache Mem

Control Proc
VEU CmdQ
VLU CmdQ
VSU CmdQ
Vector Execution Unit
VLU
VLDQ
VSU
VSDQ
Miss Tags
Replay Queues
MSHR
Tag Array
Data Array
Main Memory

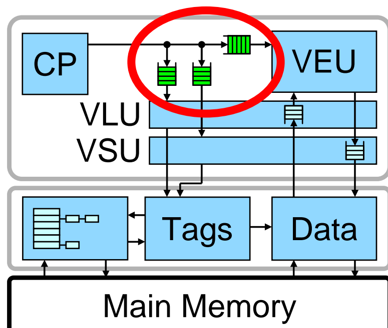
The key point is that a decoupled vector machine needs a great deal of hardware to saturate modern memory systems. This hardware includes the command queues, the VLDQ, the miss tags, and the replay queues. Reuse makes the situation even worse, since the system needs to handle even more in-flight accesses. To better understand how these resources scale we are going to take a look at a decoupling diagram.

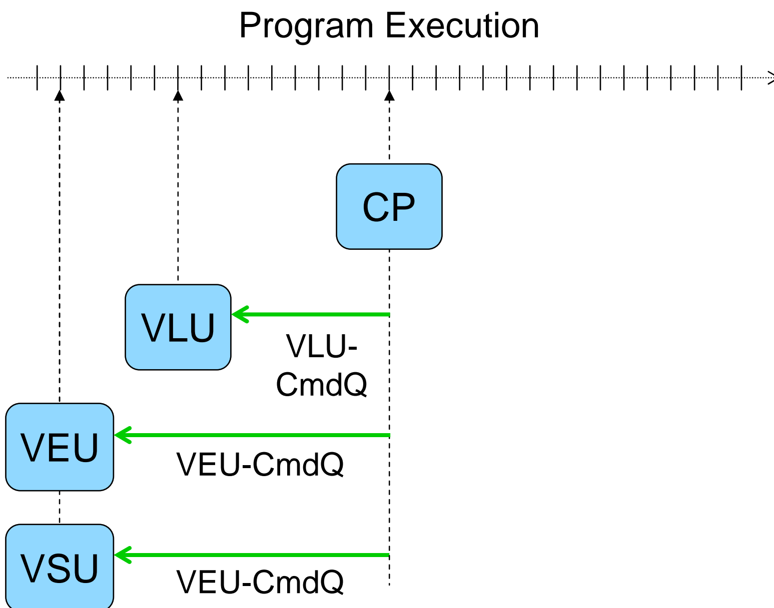
Effective decoupling requires command and data queuing



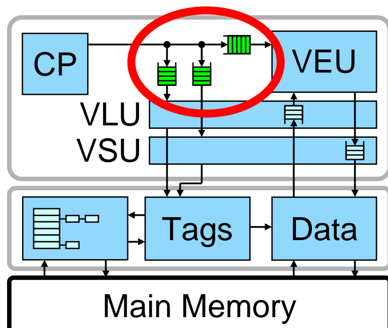
This diagram shows the decoupling between the four units in the vector machine. The horizontal position of each unit indicates which command or instruction that unit is working on. At the very beginning all four units start together - the control processor then runs ahead queuing up commands for the other units.

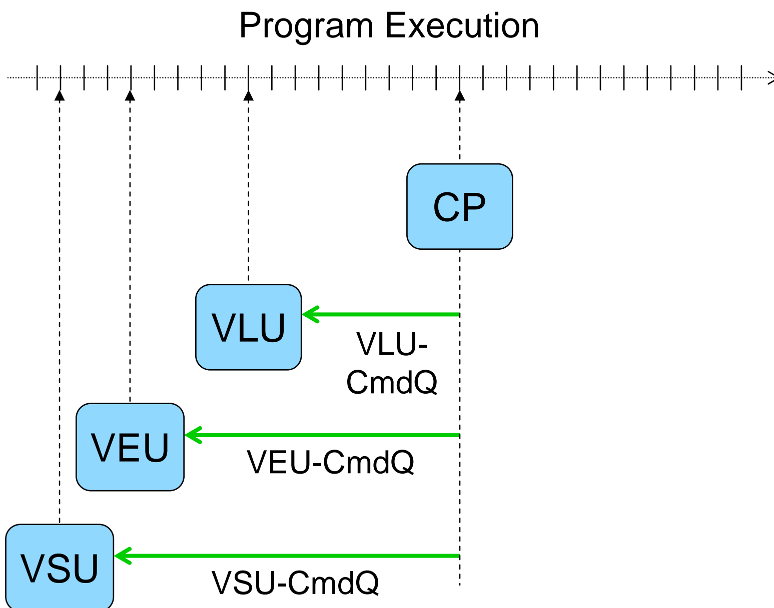
Effective decoupling requires command and data queuing

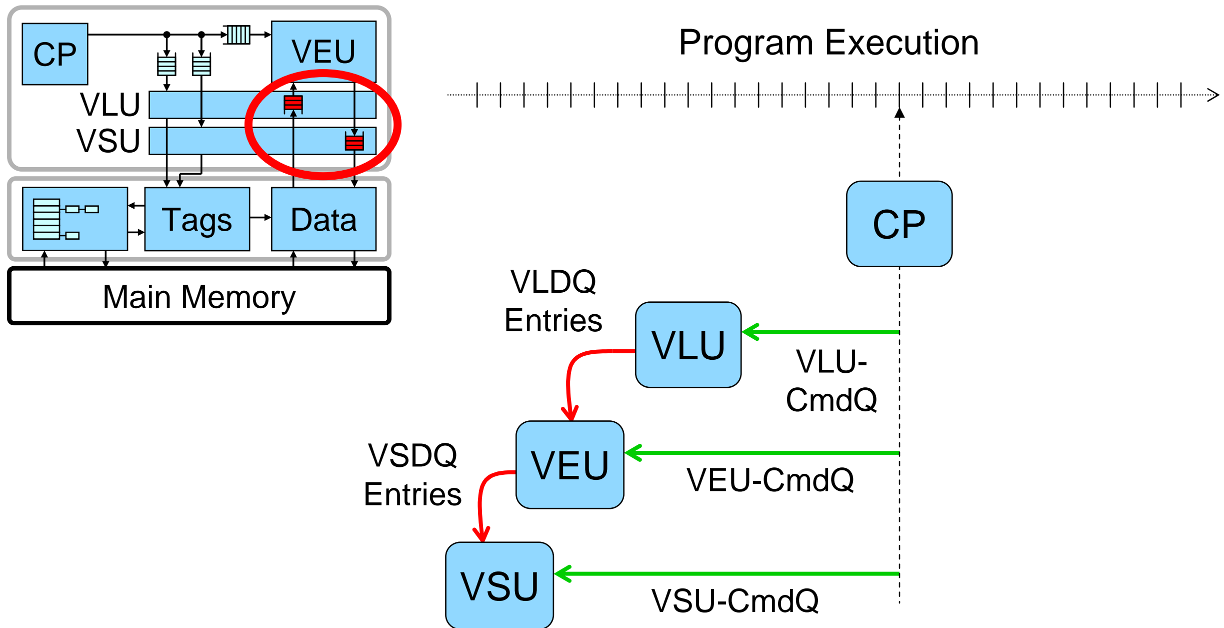




Effective decoupling requires command and data queuing

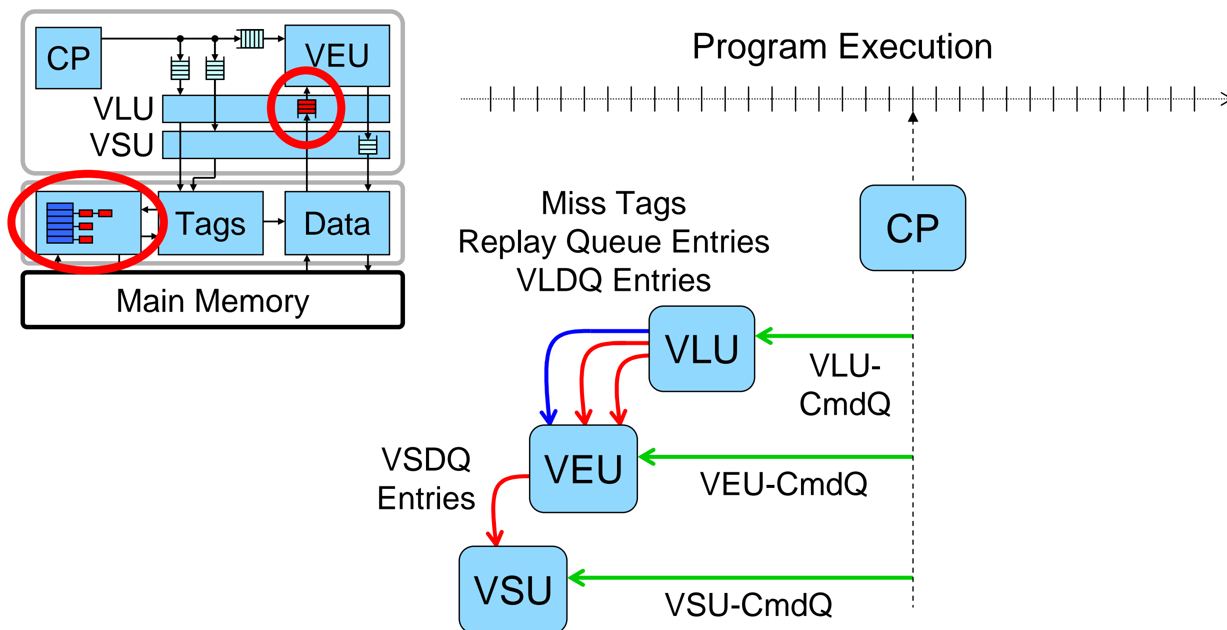






The system also includes data queues to enable decoupling between the units. The vector load data queue decouples the vector load unit and the vector execution unit, while the vector store data queue decouples the vector execution unit and the vector store unit.

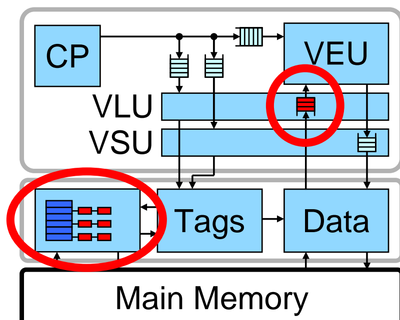
Saturating memory system with many misses requires additional queuing

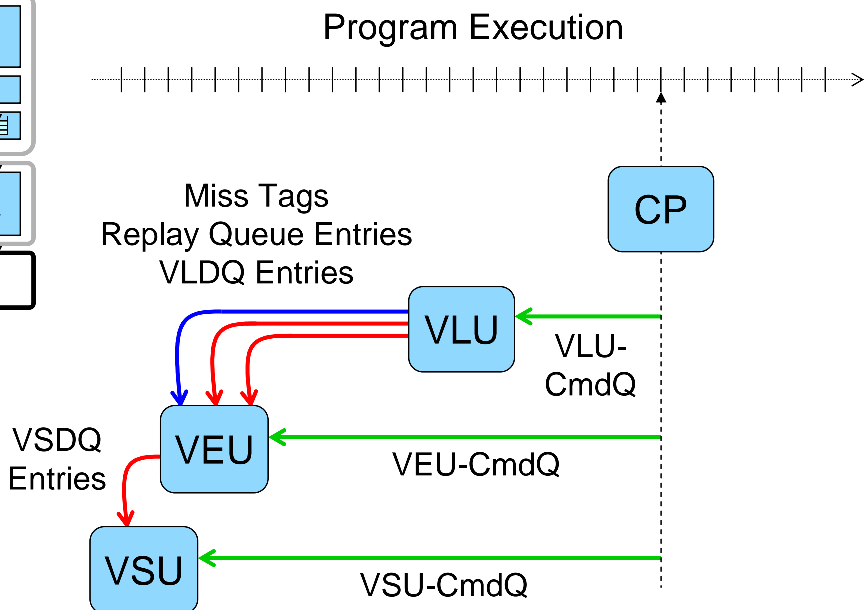


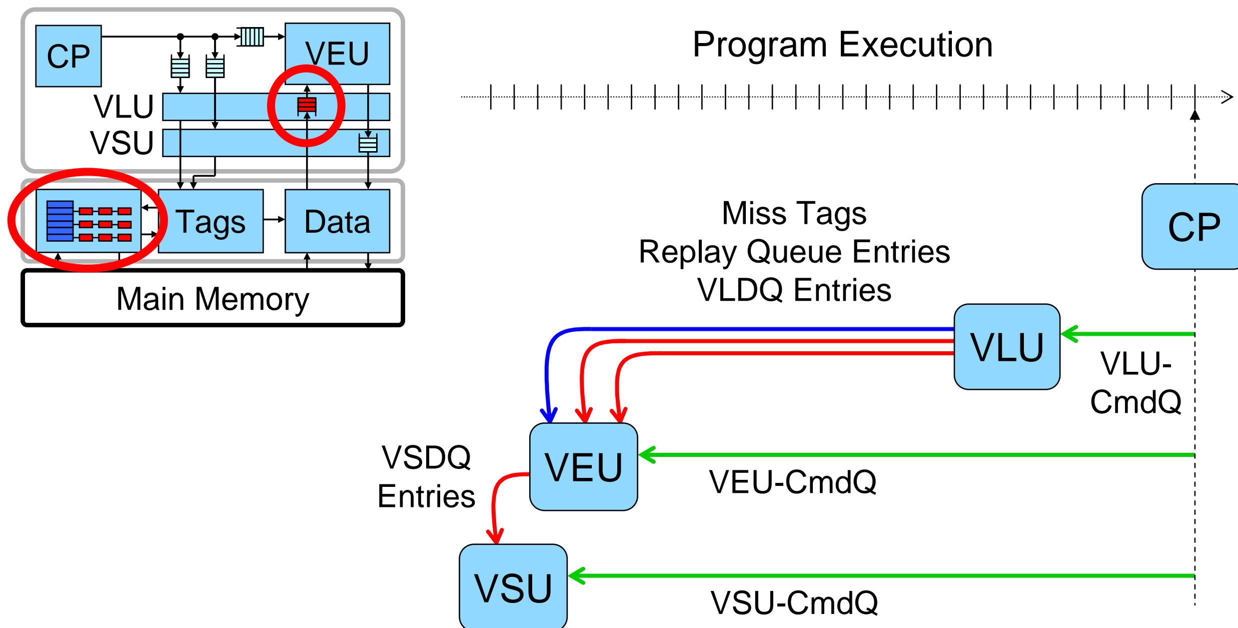
Now let's look at what happens when the VLU reaches a string of misses. The non-blocking cache will start to allocate miss tags and replay queue entries as the VLU and the control processor continue to run ahead.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

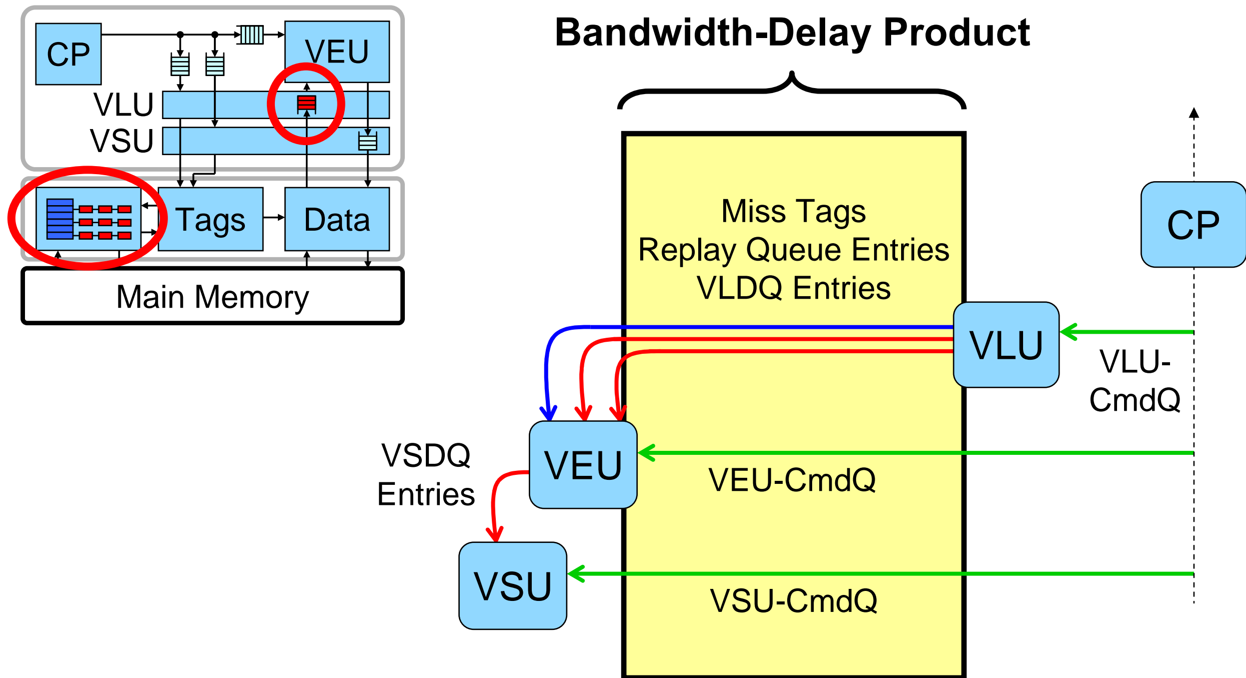
Saturating memory system with many misses requires additional queuing





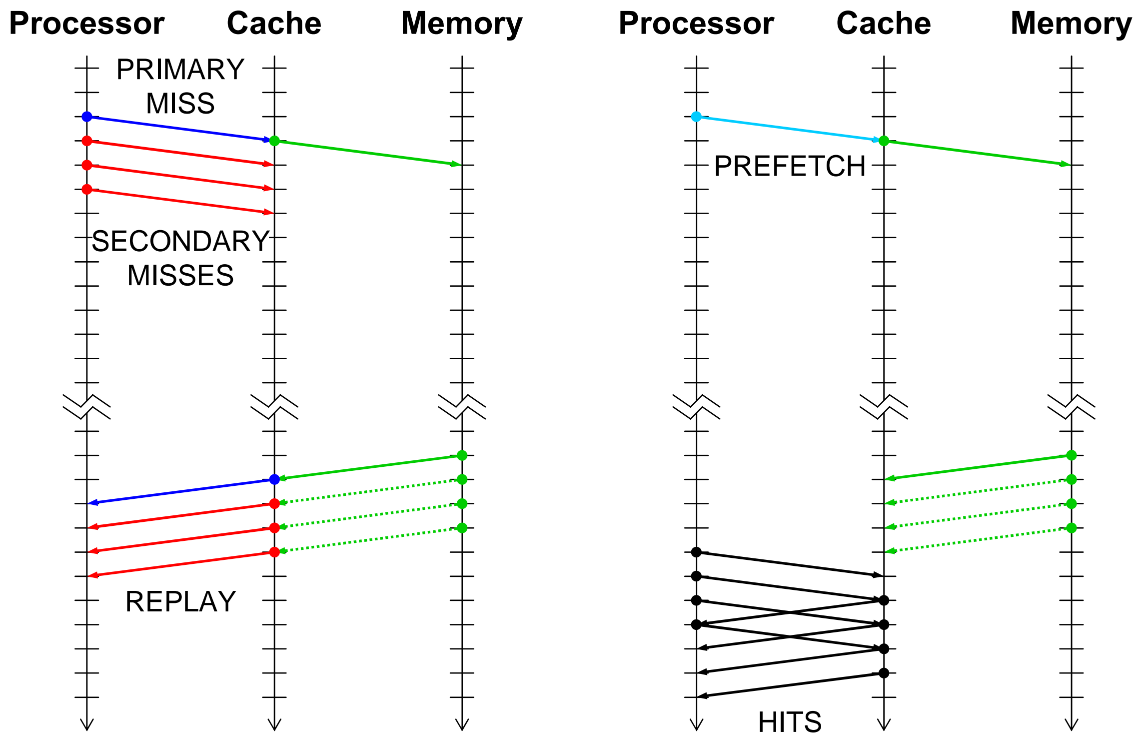


The vector execution unit is stalled waiting for the data to return.



The key point to note here is that all of these resources need to scale in order to saturate large bandwidth delay product memory systems, and these resources can be quite expensive.

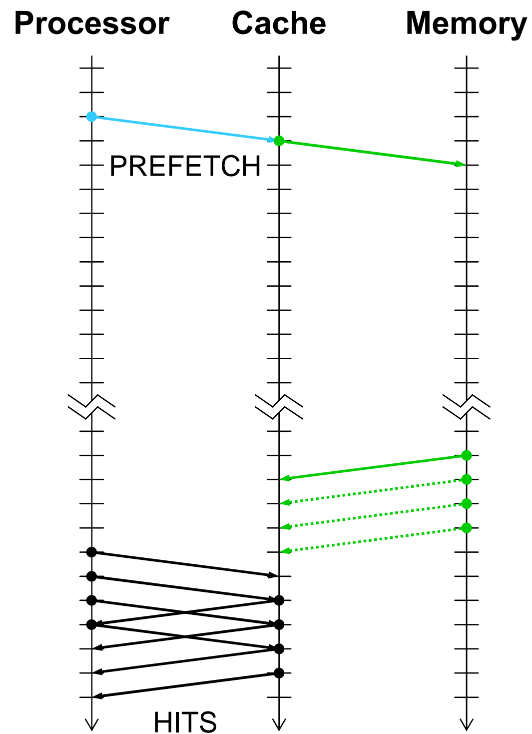
Refill/access decoupling prefetches lines into cache



We are now going to look at our first technique called refill/access decoupling. This simple technique drastically reduces the hardware cost of non-blocking data caches in vector machines. It is based on the observation that every cache miss has two parts – the refill which moves data from main memory into the cache and the access which moves data from the cache into the processor. We simply want to decouple these so that the refill happens before the access, and thus all the actual accesses should be hits.

Refill/access decoupling prefetches lines into cache

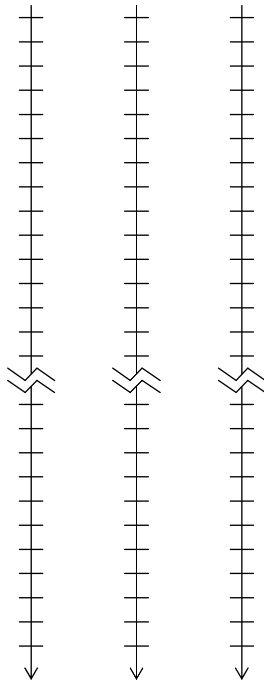
- Acts as inexpensive and non-speculative hardware prefetch
- Only need one prefetch per cacheline
- Prefetch requests are cheaper than the actual accesses

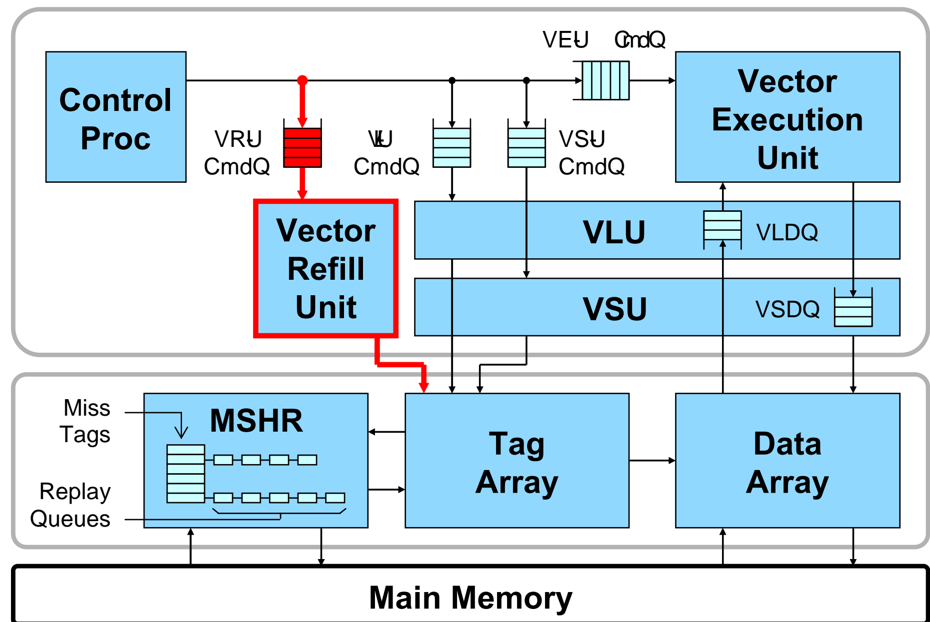


Essentially this acts as an inexpensive and non-speculative hardware prefetch. In addition to the fact that we only generate one prefetch request per cache-line, we will show over the next few slides that prefetch requests require less resources than the actual accesses themselves.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

The vector refill unit brings lines into the cache before the VLU accesses them

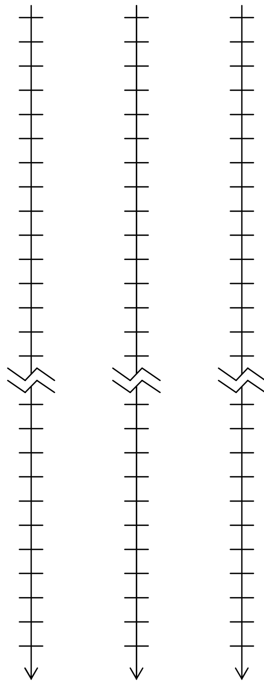


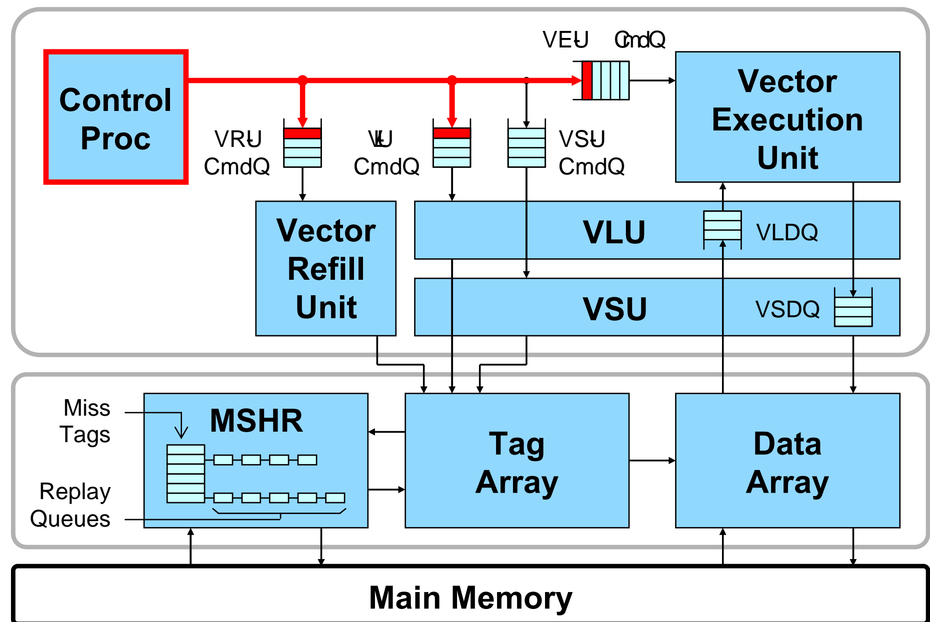


We implement refill/access decoupling by adding a vector refill unit to the previously described decoupled vector machine.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

The vector refill unit brings lines into the cache before the VLU accesses them

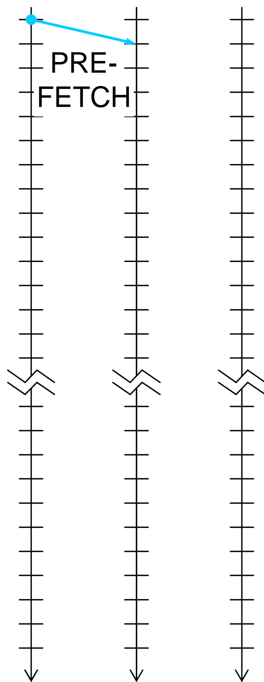


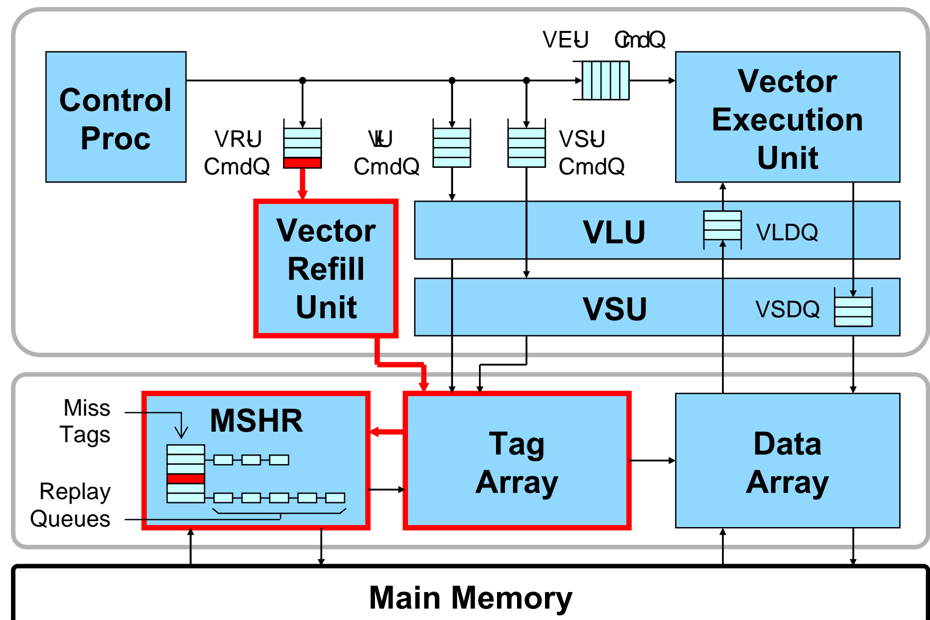


As before the control processor issues a vector load command, but now it also sends the address portion to the vector refill unit.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 3*7th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

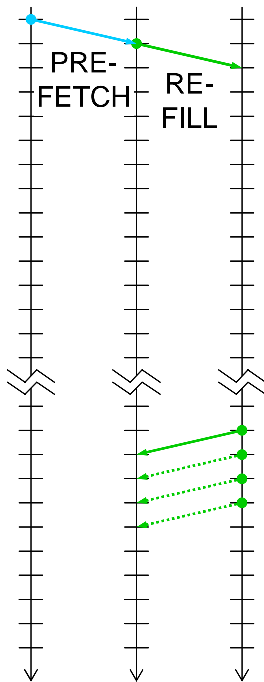
The vector refill unit brings lines into the cache before the VLU accesses them

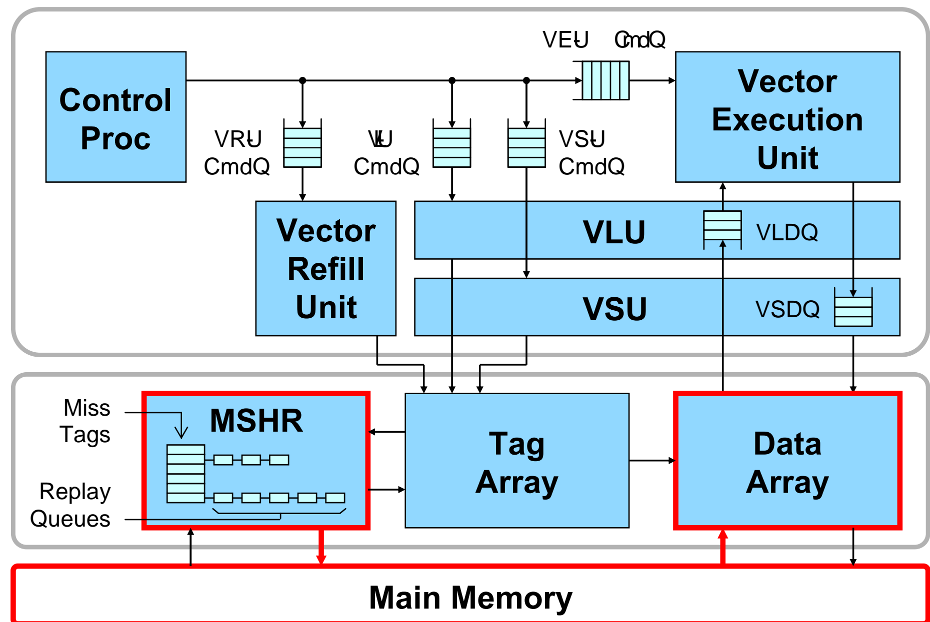




The vector refill unit issues one refill request per cacheline into the memory system. It is important to note that these refill requests are cheaper than normal requests since they only require miss tags – they do not require replay queue entries.

The vector refill unit brings lines into the cache before the VLU accesses them

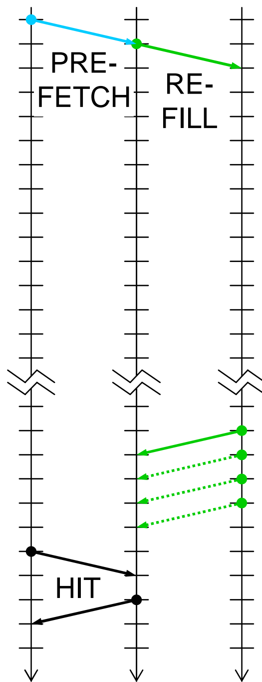


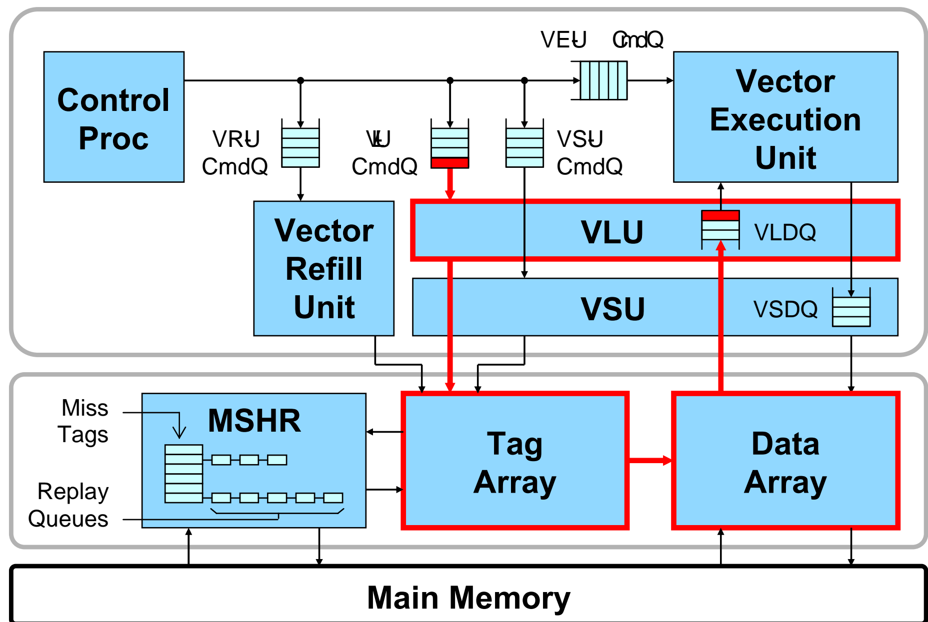


After allocating a miss tag, the cache issues the refill request to main memory and many cycles later main memory returns the data to the cache.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

The vector refill unit brings lines into the cache before the VLU accesses them

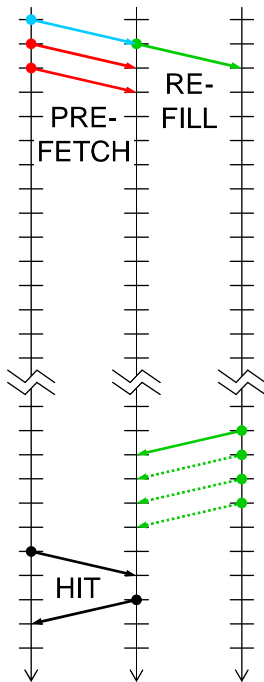


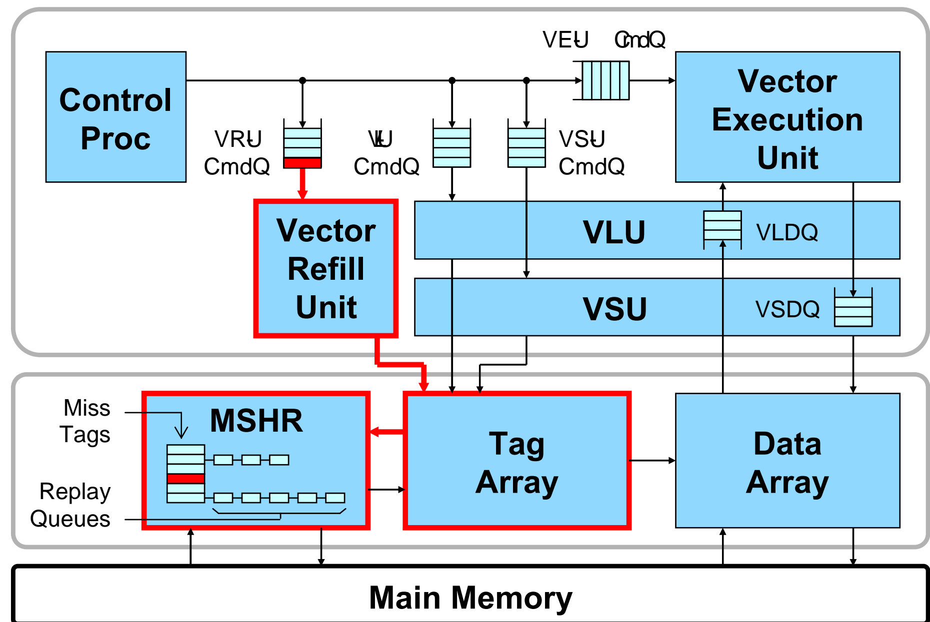


Since the vector load unit is trailing behind the vector refill unit, it should only experience hits in the cache.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

The vector refill unit brings lines into the cache before the VLU accesses them

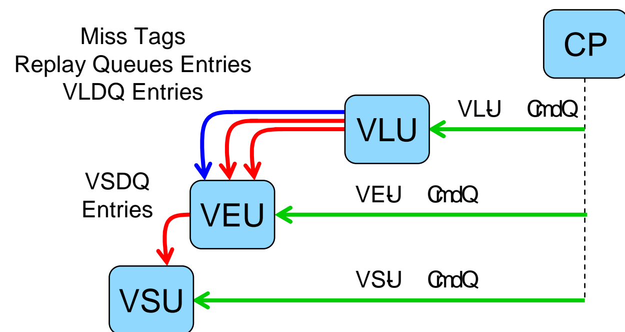




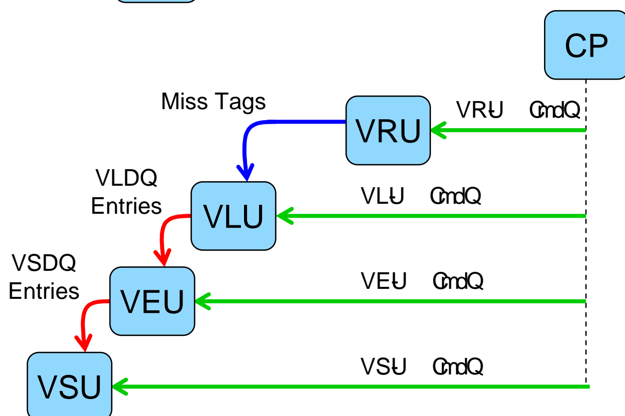
Another important point to make is that refill requests which hit in the miss tags, in other words they would normally be secondary misses, do not require any additional access management state. They can simply be dropped by the cache.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

VRU reduces need for hardware which scale with number of in-flight elements



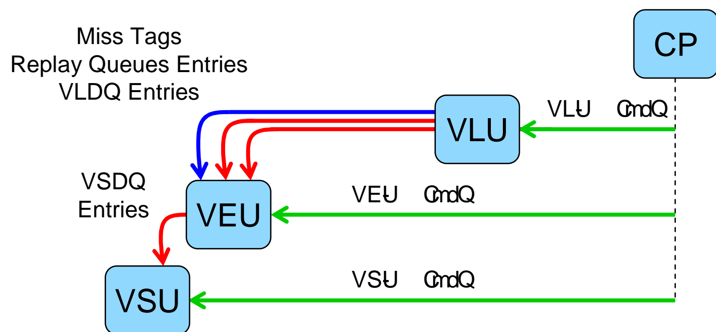
Decoupled Vector Machine



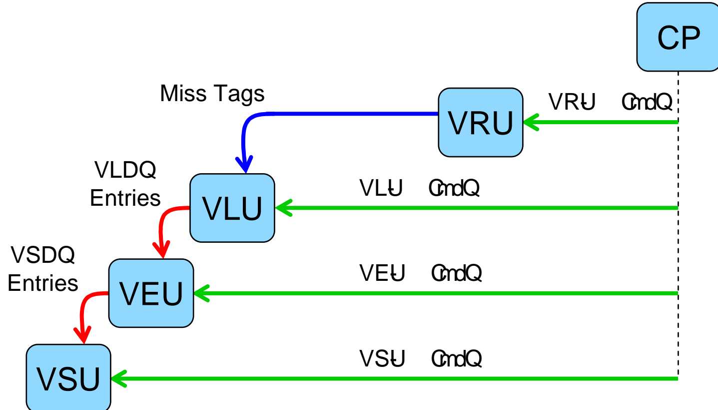
Decoupled Vector Machine with VRU

Now lets revisit the decoupling diagram when the processor experiences a string of misses, but let's include the vector refill unit.

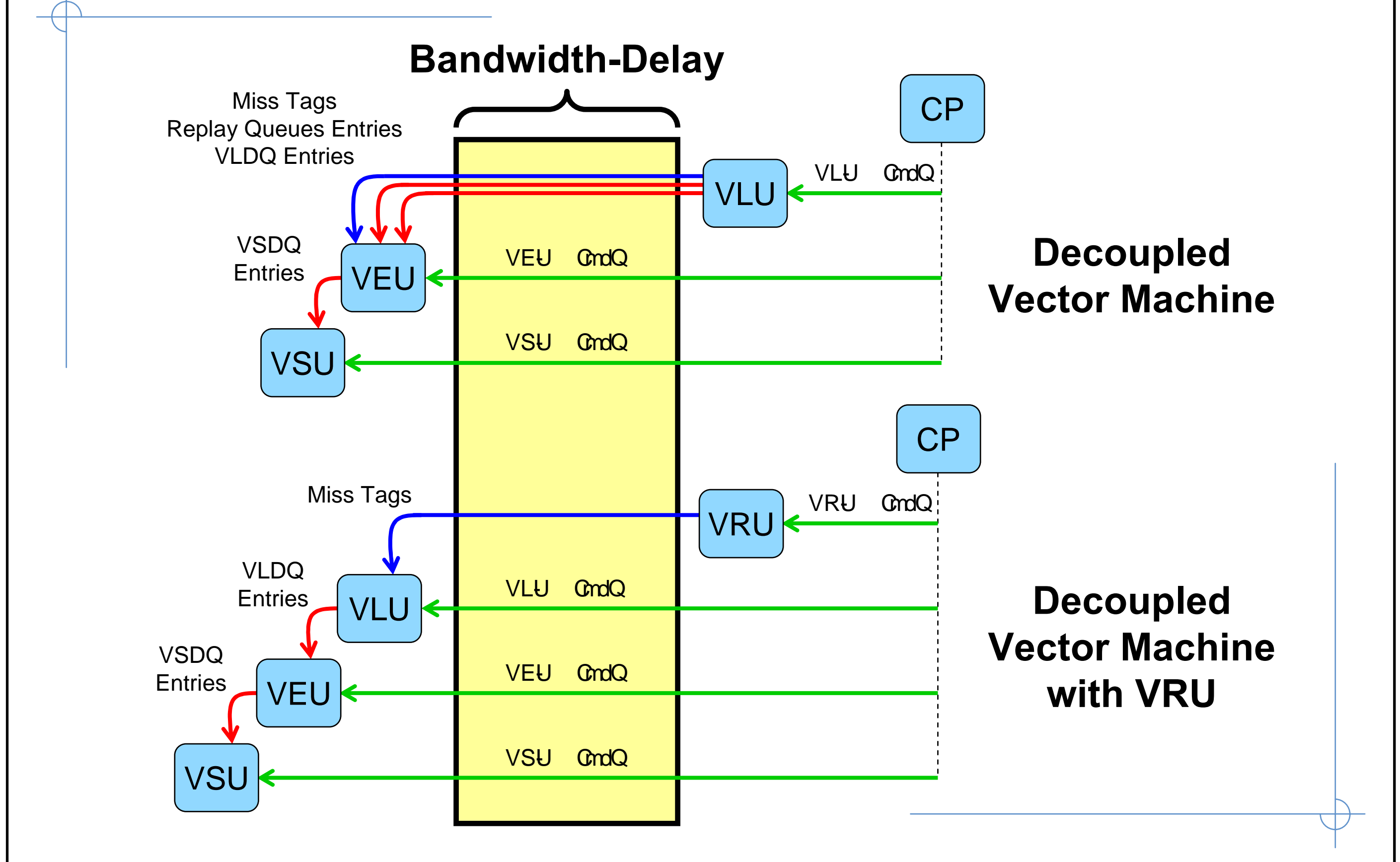
VRU reduces need for hardware which scale with number of in-flight elements



Decoupled Vector Machine



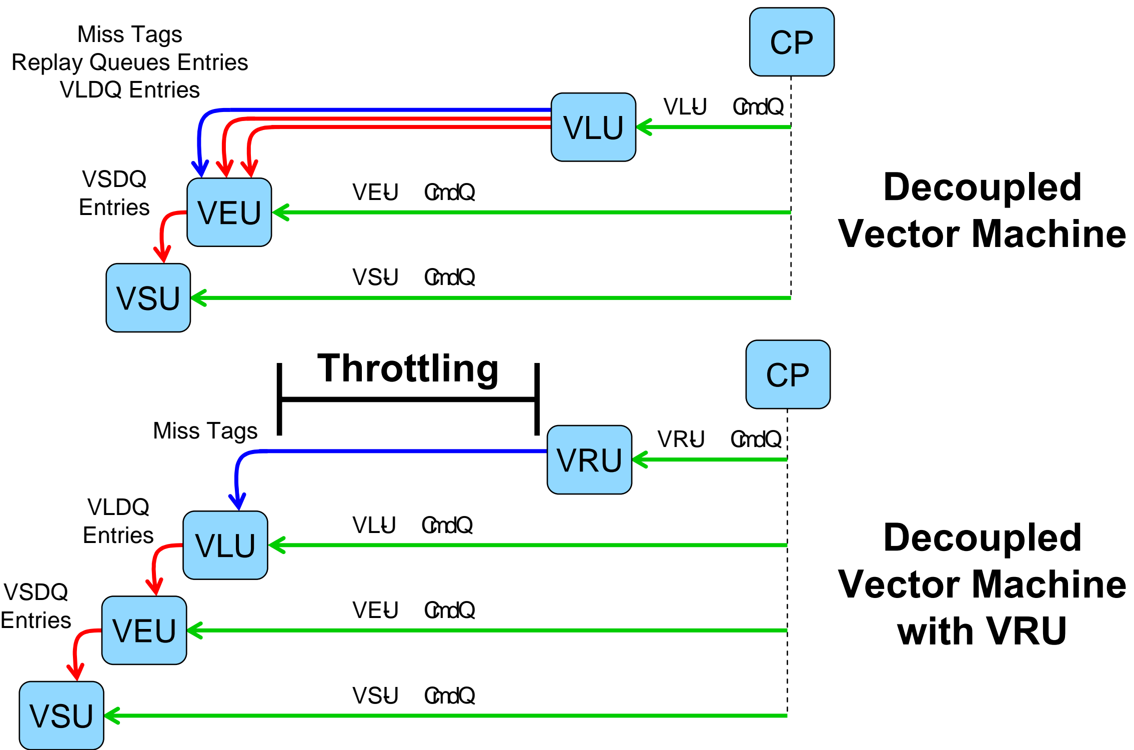
Decoupled Vector Machine with VRU



At the top is the decoupled vector machine without the VRU and at the bottom is the vector machine with the VRU. Notice that now its the VRU running ahead issuing refill requests as opposed the the VLU running ahead. The key point is that adding the VRU decreases the amount of resources needed to saturate large bandwidth-delay product memory systems. Without the VRU, expensive queues such as the VLDQ and the replay queues must scale with the number of in-flight elements, but with the VRU the only queues which must scale are the miss tags and the command queues. Both of theses resources are relatively efficient. There is one miss tag for each in-flight cache line. The command queues are very compact, since each command queue entry contains a vector instruction and thus can encode over a hundred element accesses.

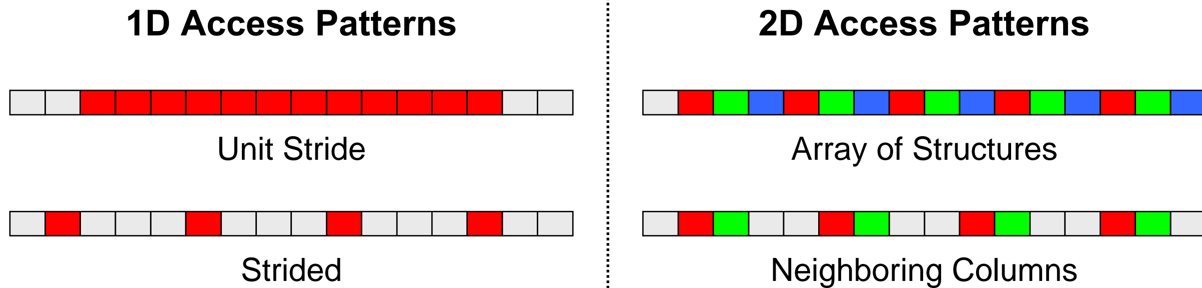
Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

VRU reduces need for hardware which scale with number of in-flight elements



Effective refill/access decoupling requires throttling between the VRU and the VLU to help maintain an appropriate prefetch distance. Although throttling is an important topic, I am not going to talk about it anymore in this presentation but there is more information in the paper.

Vector Segment Accesses

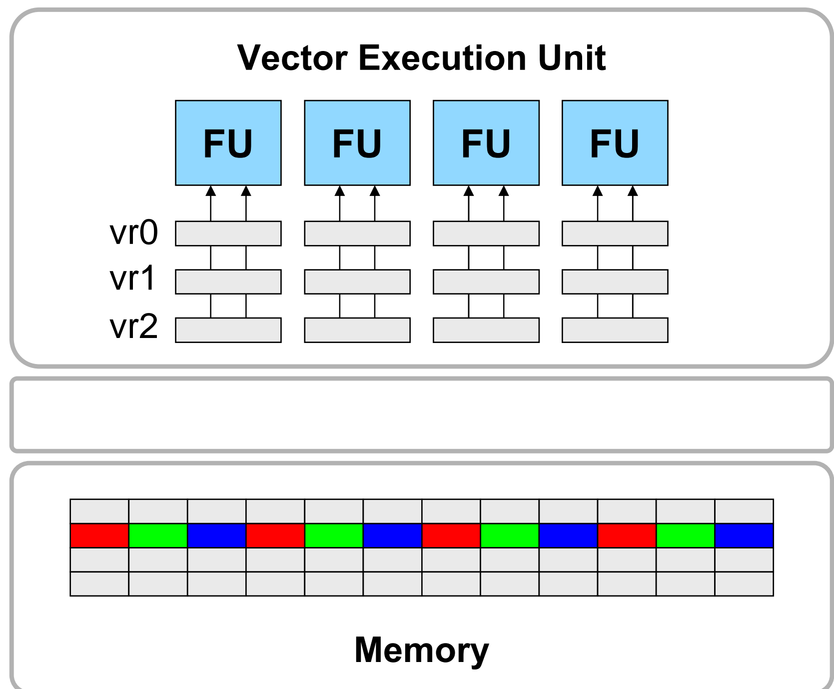


Vector segment memory accesses explicitly capture two-dimensional access patterns and thus make the memory system more efficient

Refill/access decoupling is one technique which makes it easier to turn access parallelism into performance. We are now going to look at another technique which is also going to make it easier to turn memory access parallelism into performance, but it will do so by exploiting the structure found in certain types of access patterns. Traditional vector machines exploit 1D access patterns such as unit-stride and strided, but many applications include 2D access patterns as well. For example, on the right we are accessing an array of RGB pixels or the first two columns in an four column matrix stored in row-major order. We propose vector segment memory accesses which explicitly capture two-dimensional access patterns and thus make the memory system more efficient.

Using multiple strided accesses for 2D access patterns is inefficient

```
la    r1, A
li    r2, 3
vlbst vr0, r1, r2
addu  r1,  r1, 1
vlbst vr1, r1, r2
addu  r1,  r1, 1
vlbst vr2, r1, r2
```

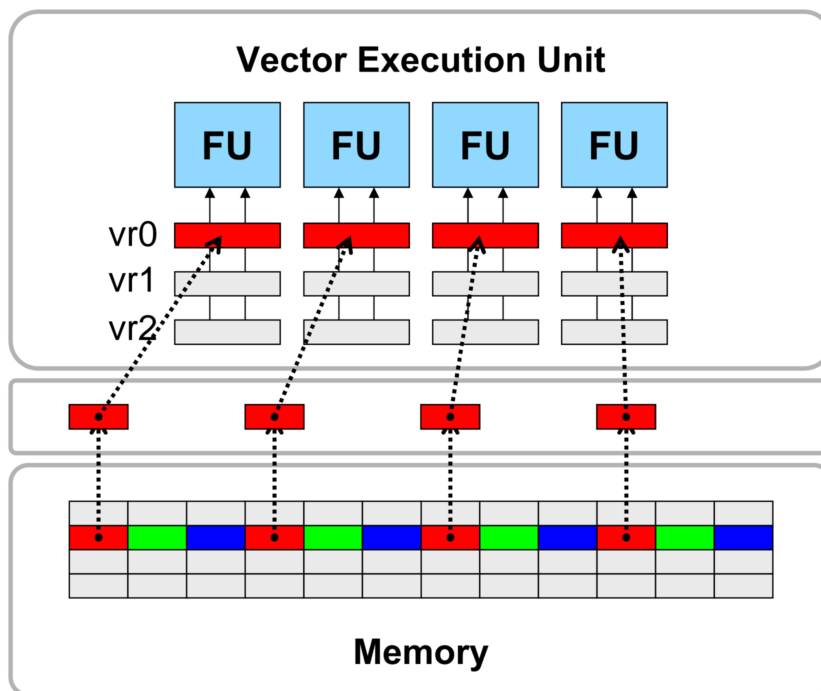


Let's look at an example to see how vector segment accesses work. Let's assume that the application wants to load an array of RGB pixels into the vector registers, and it wants each functional unit to process a single pixel. A traditional vector machine would use three strided accesses to load the data.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

Using multiple strided accesses for 2D access patterns is inefficient

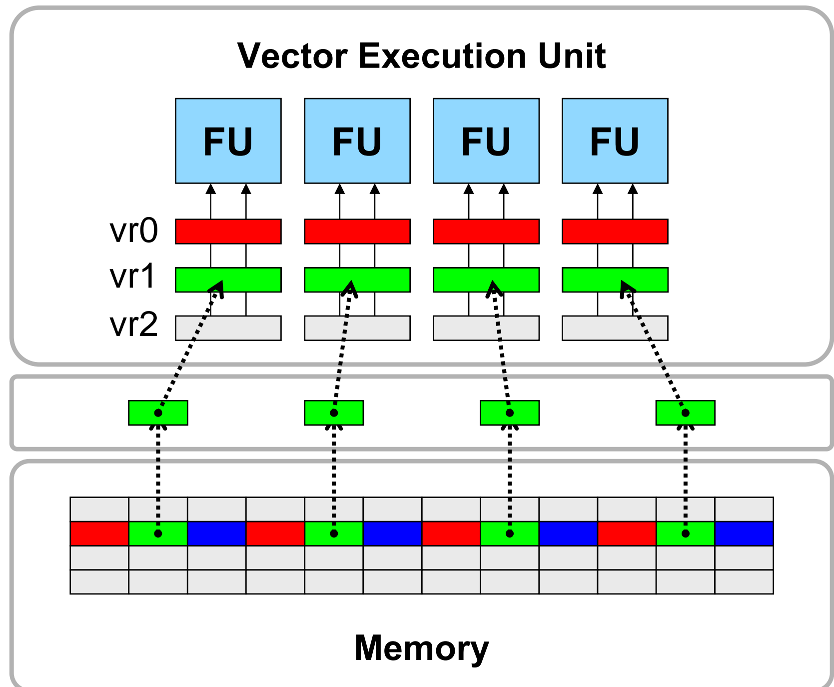
```
la     r1, A
li     r2, 3
vlbst  vr0, r1, r2
addu   r1, r1, 1
vlbst  vr1, r1, r2
addu   r1, r1, 1
vlbst  vr2, r1, r2
```



The first strided access would pull out the red data and write it into vector register zero …

Using multiple strided accesses for 2D access patterns is inefficient

```
la    r1, A
li    r2, 3
vlbst vr0, r1, r2
addu  r1,  r1, 1
vlbst vr1, r1, r2
addu  r1,  r1, 1
vlbst vr2, r1, r2
```

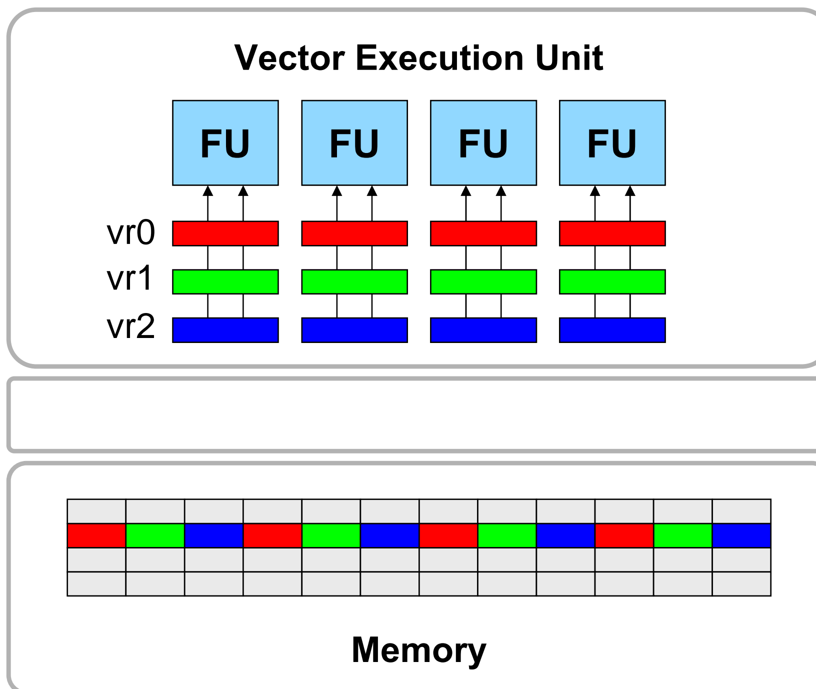


… while the second strided access would pull out the green data. I have drawn this figure with a four read port memory – often memories use banks to enable multiple read ports, but a key disadvantage of strided accesses is that it is common for these access to have bank conflicts which decrease performance.

Using multiple strided accesses for 2D access patterns is inefficient

```
la    r1, A
li    r2, 3
vlbst vr0, r1, r2
addu  r1,  r1, 1
vlbst vr1, r1, r2
addu  r1,  r1, 1
vlbst vr2, r1, r2
```

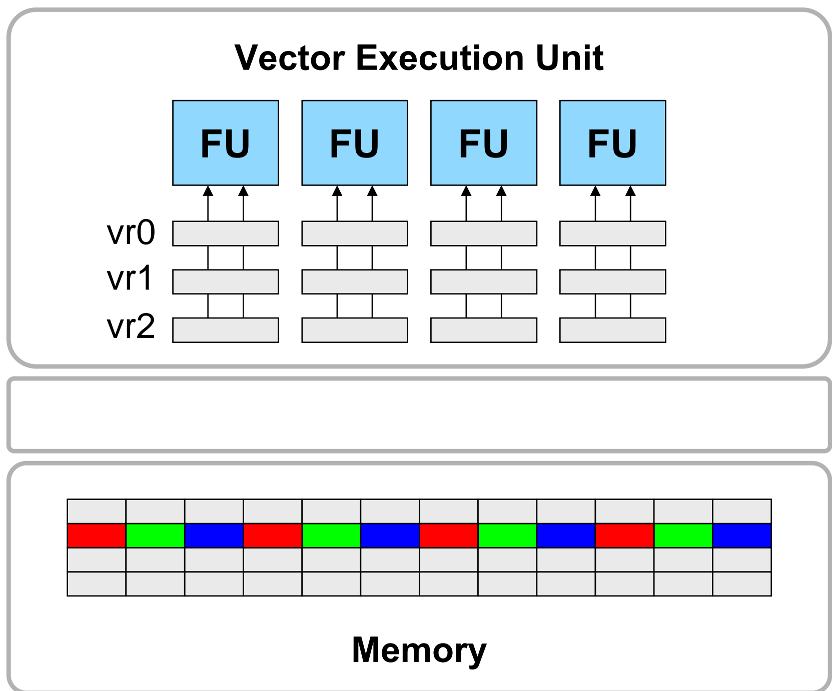
Multiple strided access do not capture the spatial locality inherent in the 2D access pattern



A final strided access pulls the blue data into the vector register file. The problem with using multiple strided accesses is that they do not capture the spatial locality inherent in the 2D access pattern.

Vector segment accesses perform the 2D access pattern more efficiently

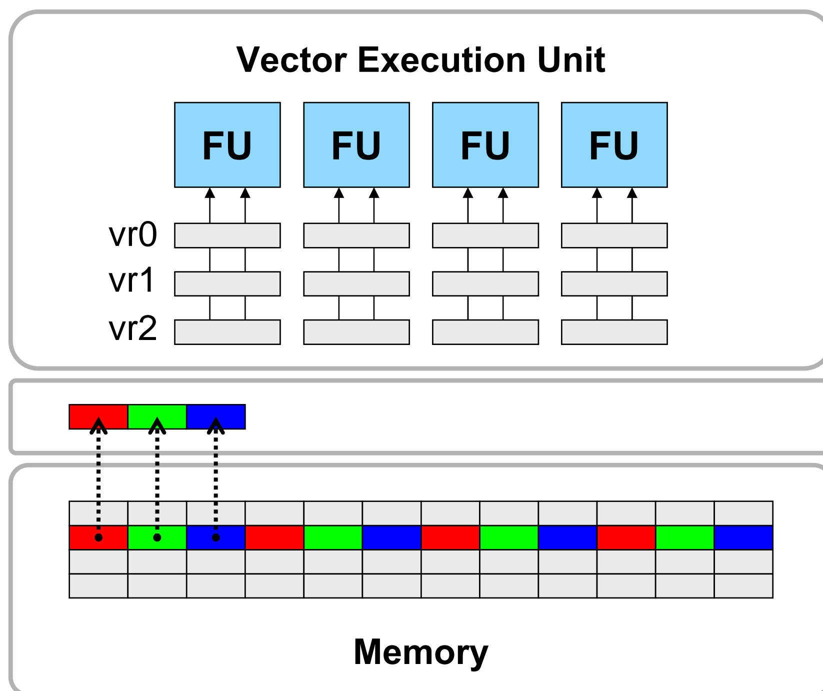
```
la     r1, A
vlbseg 3, vr0, r1
```



Now let's see how vector segment accesses better capture this spatial locality.

Vector segment accesses perform the 2D access pattern more efficiently

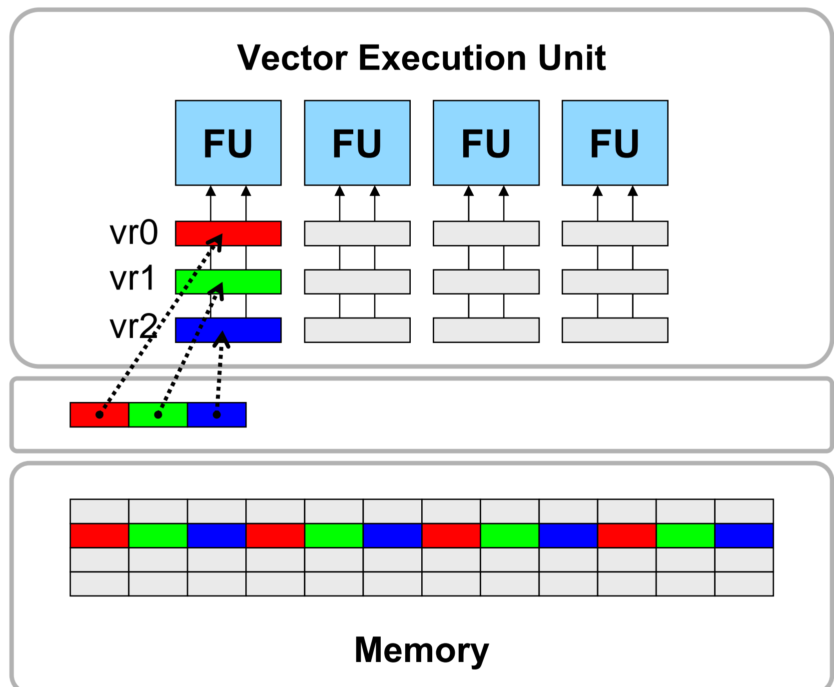
```
la     r1, A
vlbseg 3, vr0, r1
```



Conceptually, a segment access works by reading the first segment in memory.

Vector segment accesses perform the 2D access pattern more efficiently

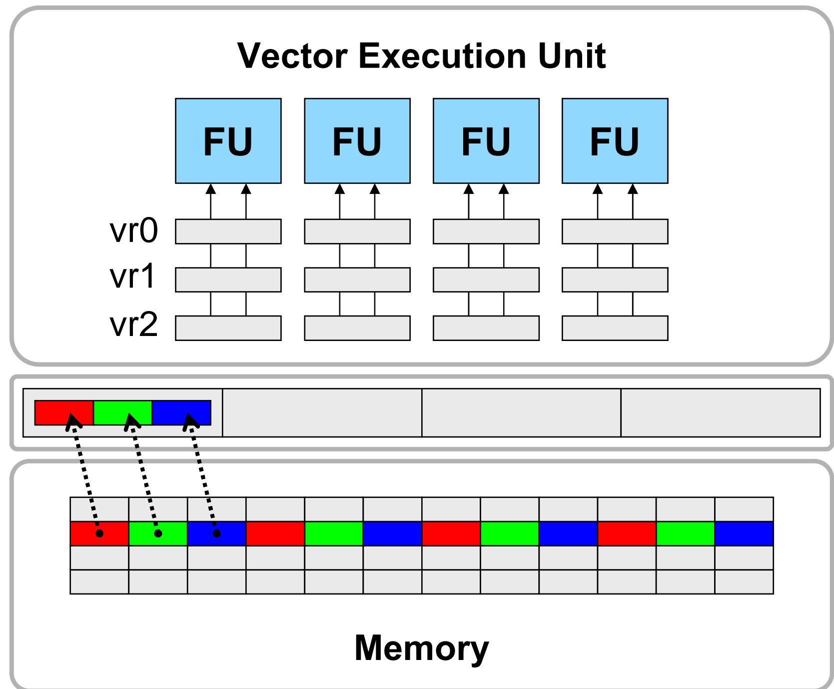
```
la     r1, A
vlbseg 3, vr0, r1
```



And then writing this data into the first element of each vector register. The segment instruction includes three fields which specify the number of elements in a segment, the base vector register, and the base address. Unfortunately, this diagram implies that the number of vector register write ports is equal to the segment length. Obviously this is unreasonable, so an efficient implementation of segment accesses will include segment buffers.

Vector segment accesses perform the 2D access pattern more efficiently

```
la      r1, A
vlbseg 3, vr0, r1
```

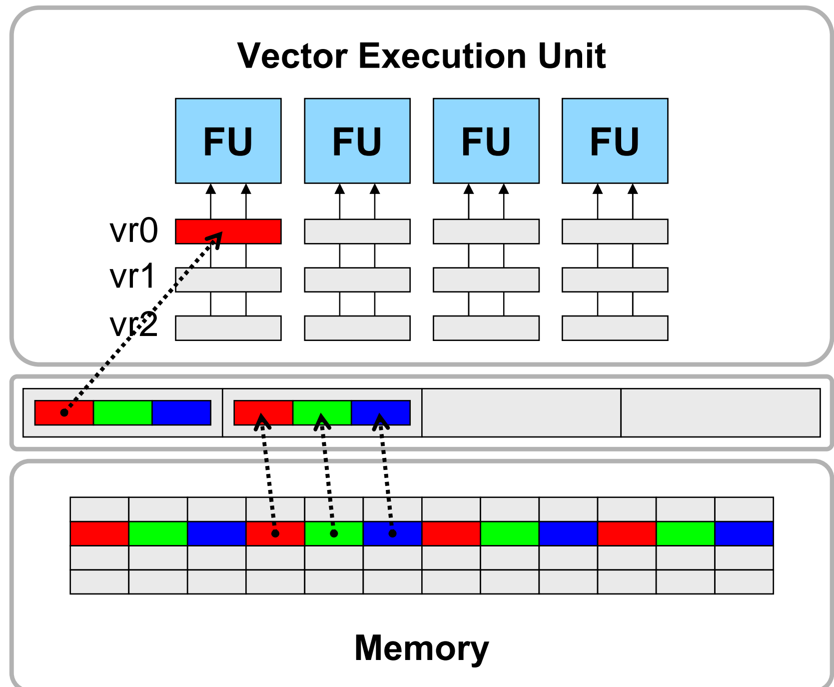


So now on the first cycle we read a segment out and store it in a segment buffer.

Vector segment accesses perform the 2D access pattern more efficiently

```
la      r1, A
vlbseg 3, vr0, r1
```

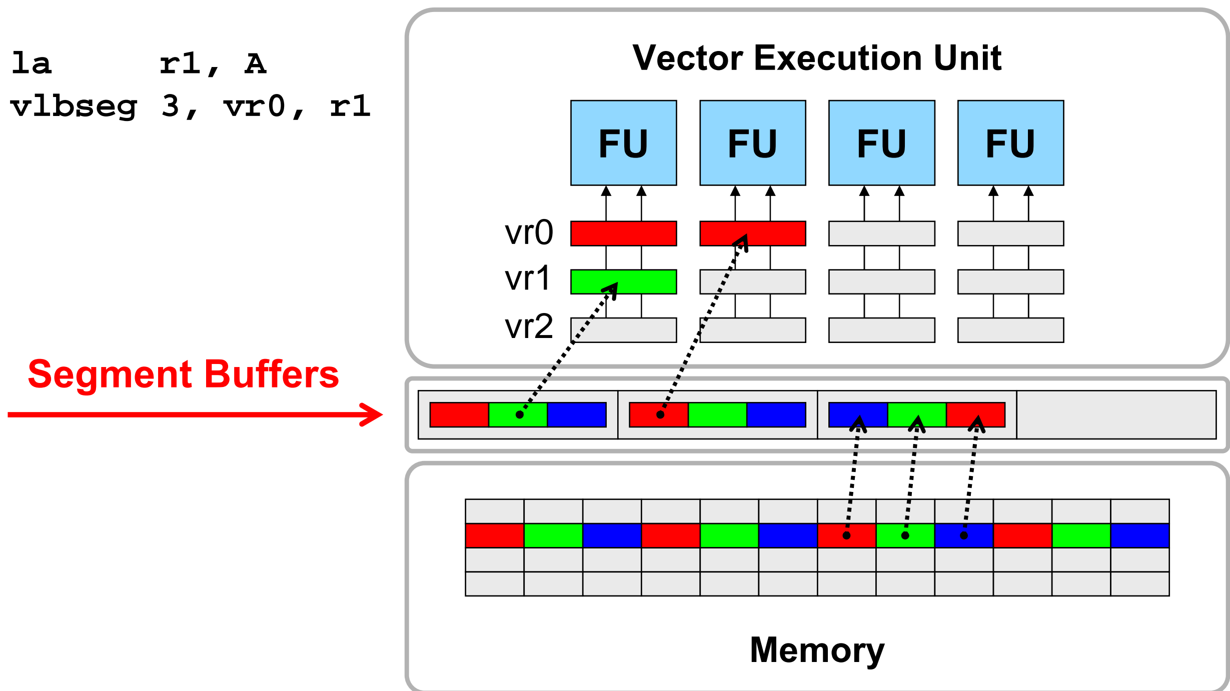
Segment Buffers



On the second cycle we read out the second segment into the second buffer, but at the same time we move the red element of the first segment into the first vector register.

Vector segment accesses perform the 2D access pattern more efficiently

```
la     r1, A
vlbseg 3, vr0, r1
```



We continue to overlap reading out segments into the buffers with moving the data into the vector registers. Notice that we only need one wide read port into the memory - this wide read port is the same one needed to efficiently handle unit stride accesses. We also only need one vector register file write port per lane.

Vector segment accesses perform the 2D access pattern more efficiently

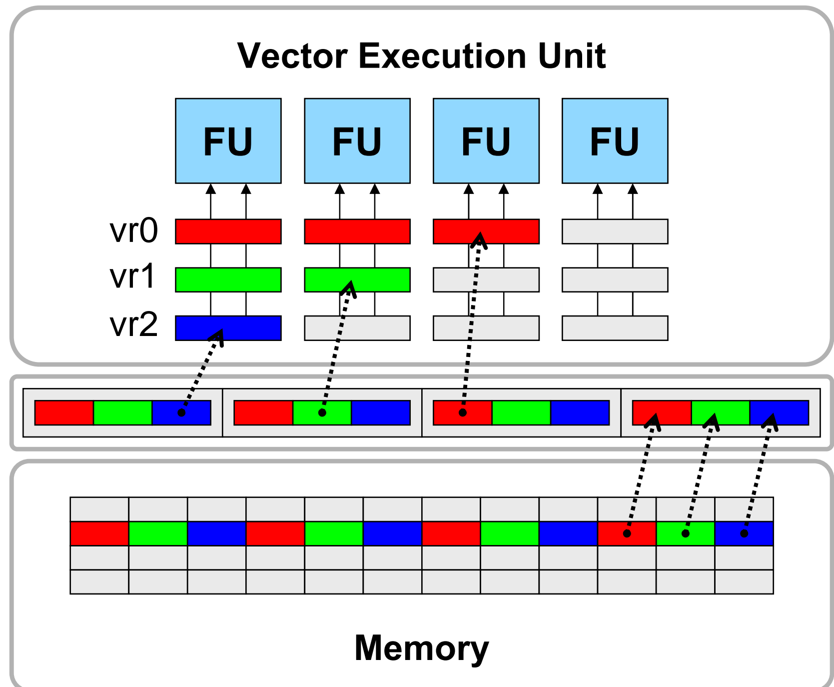
```
la      r1, A
vlbseg 3, vr0, r1
```

Efficient encoding

- More compact command queues
- VRU process commands faster

Captures locality

- Reduces bank conflicts
- Moves data in unit-stride bursts



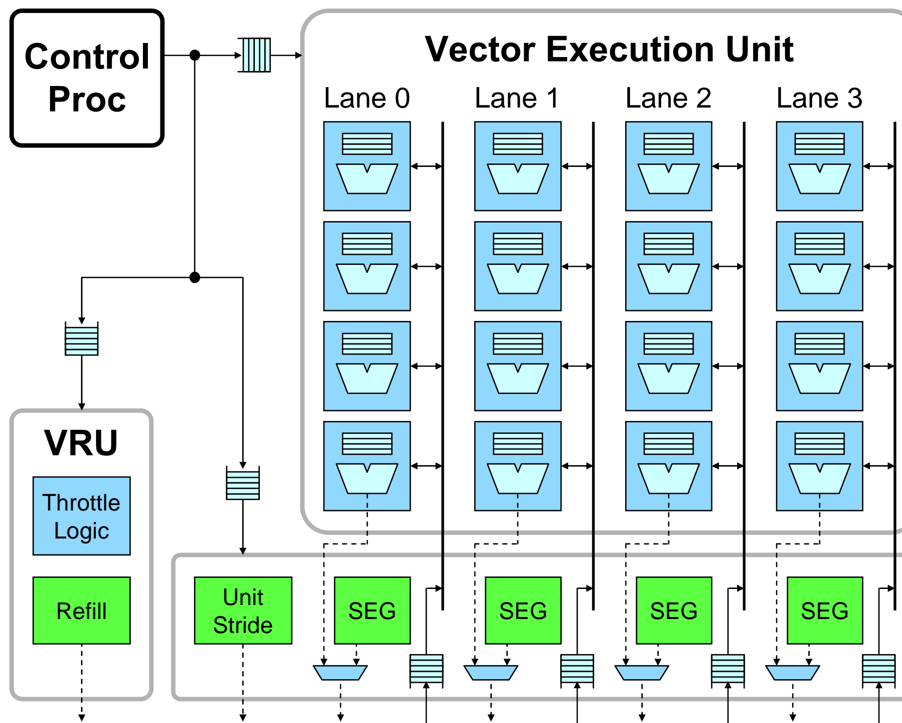
Vector segments have two fundamental advantages over multiple strided access. First they offer a more efficient encoding which results in more compact command queues and allows the VRU to process commands faster. And secondly, vector segment accesses better capture the spatial locality inherent in the 2D access pattern which reduces bank conflicts and moves data in and out of the cache in unit-stride bursts.

Cache Refill/Access Decoupling for Vector Machines

- **Intuition**
  - Motivation
  - Background
  - Cache Refill/Access Decoupling
  - Vector Segment Memory Accesses
- **Evaluation**
  - The SCALE Vector-Thread Processor
  - Selected Results

In the first part of the talk, I have provided some intuition behind our proposed techniques. It is important to note that cache refill/access decoupling is a microarchitectural technique and thus could be applicable to any vector machine, while vector segment memory accesses require an ISA change. In the final part of the talk, I will briefly evaluate an implementation of these ideas within the SCALE vector-thread processor.

SCALE Vector Processor

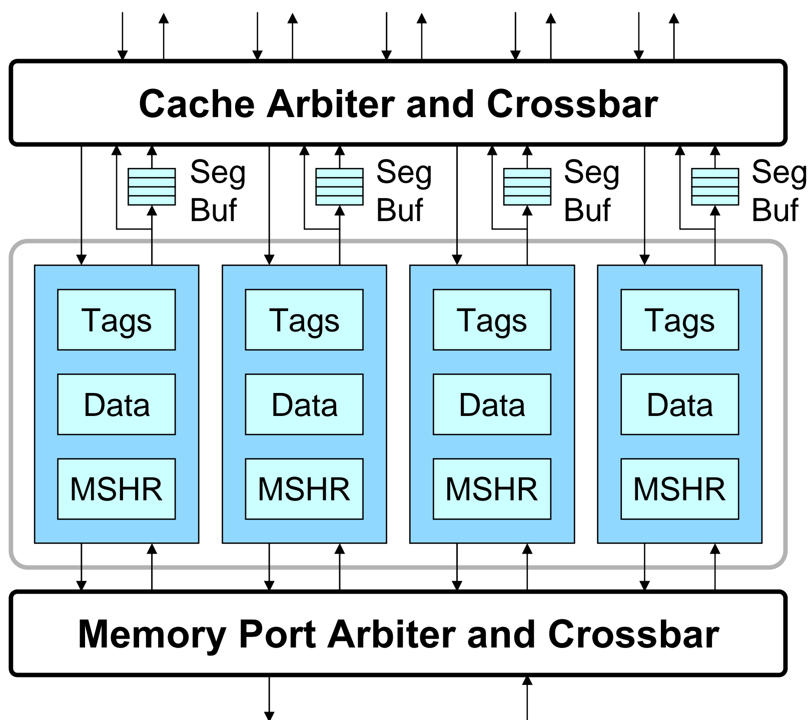


Key Features

- 4 lanes, 4 clusters
- Cluster for indexed accesses
- 4 segment address generators
- 4 VLDQs
- VRU includes throttle logic, refill address generator

On this slide, I just want to highlight some of the key differences between the SCALE vector-thread processor and the abstract decoupled vector machine we have been talking about so far. First, we do not use any of the advanced threading features of the vector-thread processor – in this work we are strictly using the SCALE processor as a more traditional decoupled vector machine. SCALE has four lanes and four clusters per lane. One cluster is able to do indexed accesses and thus has its own load data queue. There are five address generators in the vector load unit: one for unit stride and four for segment and strided accesses. It is important to note that traditional strided accesses are simply treated as segment accesses with a segment length of one both in the ISA and in the implementation. There are four VLDQs – one per lane. The VRU requires relatively little hardware – it requires its own address generator to generate refill requests and some throttling logic. Throttling is an important part of this work and it is discussed further in the paper.

SCALE Cache



Key Features

- Unified I/D cache
- Two cycle hit latency
- Four 8 KB banks
- 32 way associative
- 32B cache lines
- 16B/cycle per bank
- Four 16B segment buffers per bank

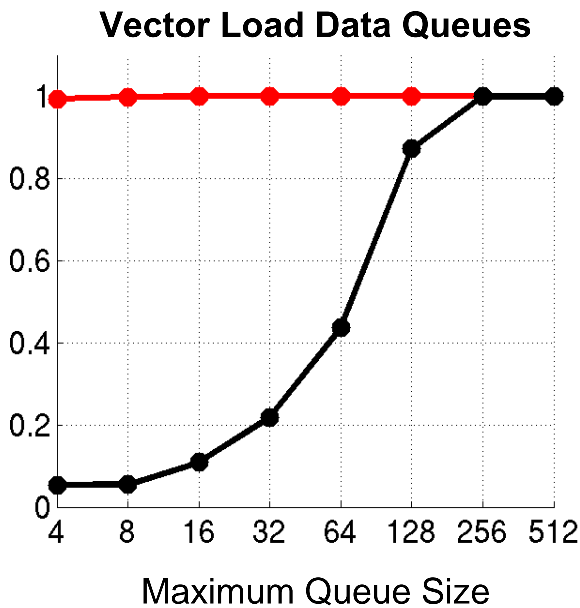
SCALE uses a unified 32 KB cache. The tag and data arrays as well as the MSHRs are divided into four independent banks with a bank data bandwidth of 16 bytes per cycle. SCALE includes four 16 byte load segment buffers per bank.

Methodology and Kernels

- Simulation methodology
  - Microarchitectural C++ simulator of SCALE vector processor and non-blocking multi-banked cache
  - Main memory is modeled with a simple pipelined magic memory
  - Benchmarks were compiled for the control processor with gcc and key kernels were coded by hand in assembly
- 14 kernels with varying access patterns
  - `vvaddw` Add two word element vectors and store result
  - `hpg` 2D high pass filter on image with 8 bit pixels [EEMBC]
  - `rgbyiq` RGB to YIQ color conversion with segments [EEMBC]

To evaluate our ideas we used a micro-architectural C++ simulator of both the SCALE processor and the cache. For all of these results, main memory was modeled as a simple pipelined magic memory. Benchmarks were compiled for the control processor using g++ and then key kernels were hand coded in assembly for the vector execution unit. There our 14 kernels with varying access patterns in the paper, but in this talk I am only going to look at three simple kernels to illustrate some of the basic concepts. VVADDW adds two vectors together and stores the result. HPG is an EEMBC benchmark which does a 2D high-pass filter on an image of 8 bit pixels. RGBYIQ is also an EEMBC benchmark which does a color conversion from RGB to YIQ using segments.

Normalized performance for vvaddw with varying queue sizes



Configuration

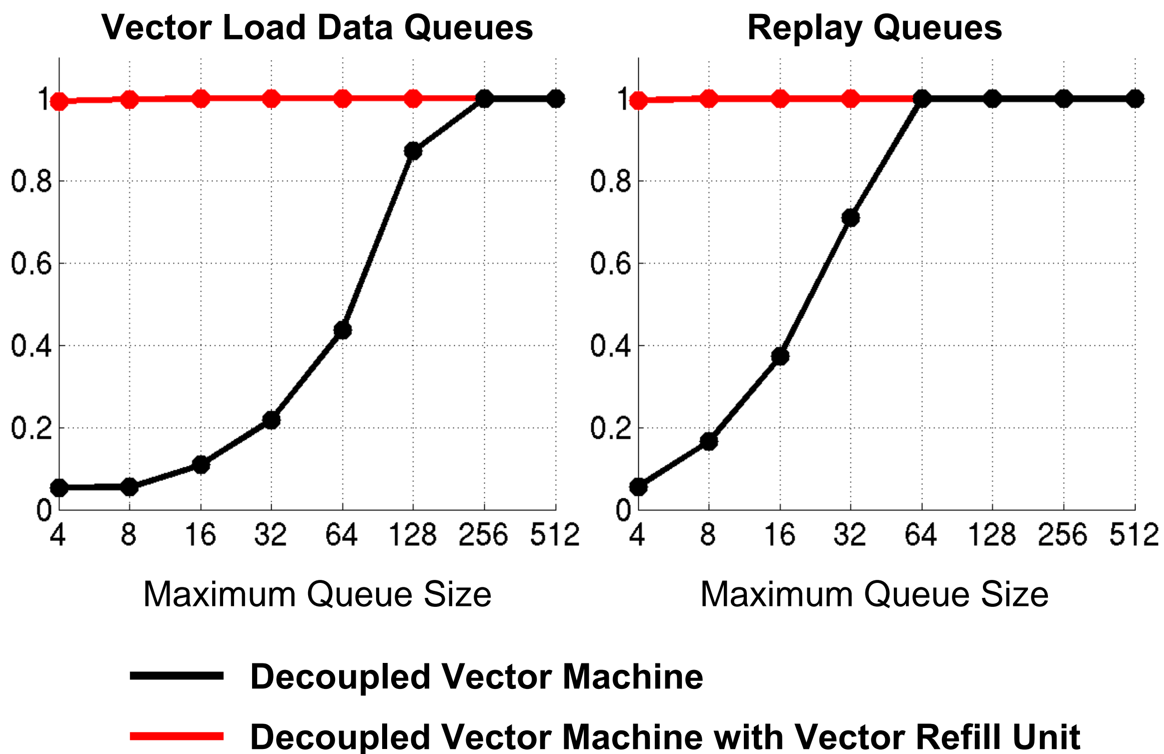
- Limit study with very large queue sizes except for queue under consideration
- 8B/cycle bandwidth and 100 cycle latency main memory
- Normalized performance with and without the vector refill unit

Decoupled Vector Machine

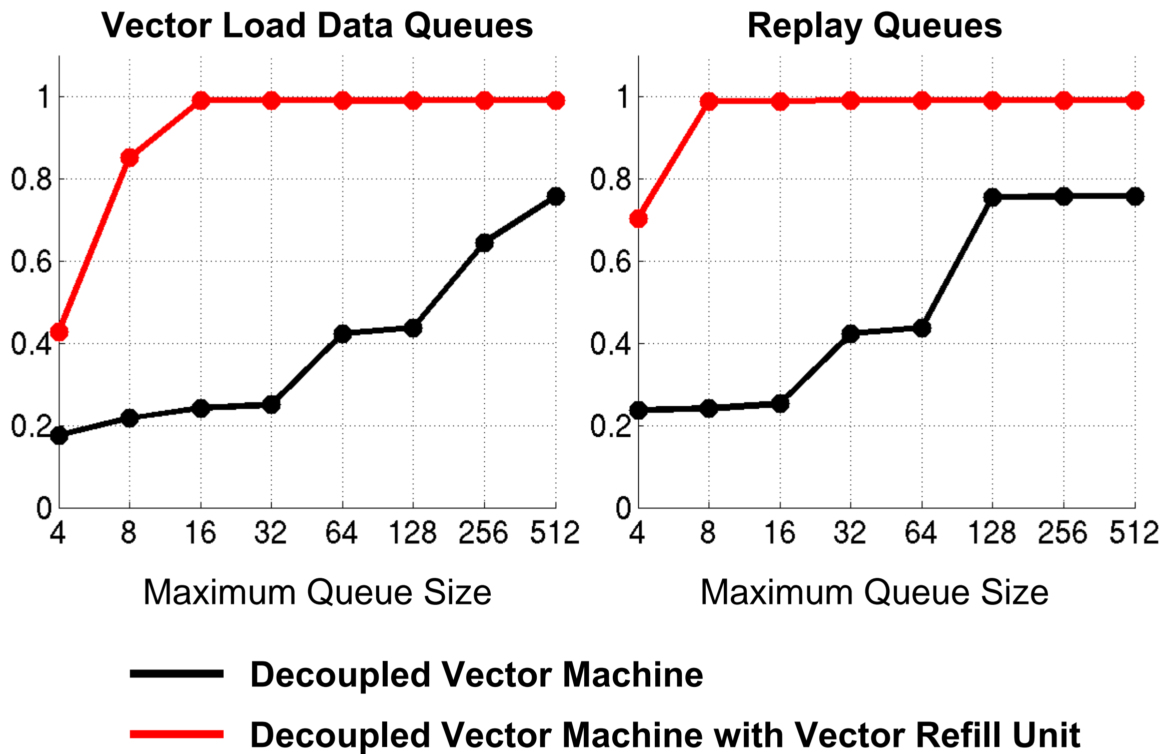
Decoupled Vector Machine with Vector Refill Unit

We began with several limit studies – in these experiments all queue and buffer sizes are very large except for a specific queue under consideration. For example, in this figure we are examining the impact of the VLDQ size on performance. All of these experiments use a 8 B/cycle main memory with a 100 cycle latency. The performance is normalized to that application's peak performance. We examine two configurations: the black line is the baseline decoupled vector machine while the red line is the decoupled vector machine with the vector refill unit. So let's first look at the black line - as we increase the size of the VLDQ, performance also increases. This is to be expected since a larger VLDQ means that the processor can get more accesses in-flight and can thus better turn memory access parallelism into performance. Notice that when we add the vector refill unit, we are able to achieve peak performance with drastically fewer resources. For example, without the VRU we need 256 VLDQ entries to achieve peak performance but with the VRU we only need four.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 3*7th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

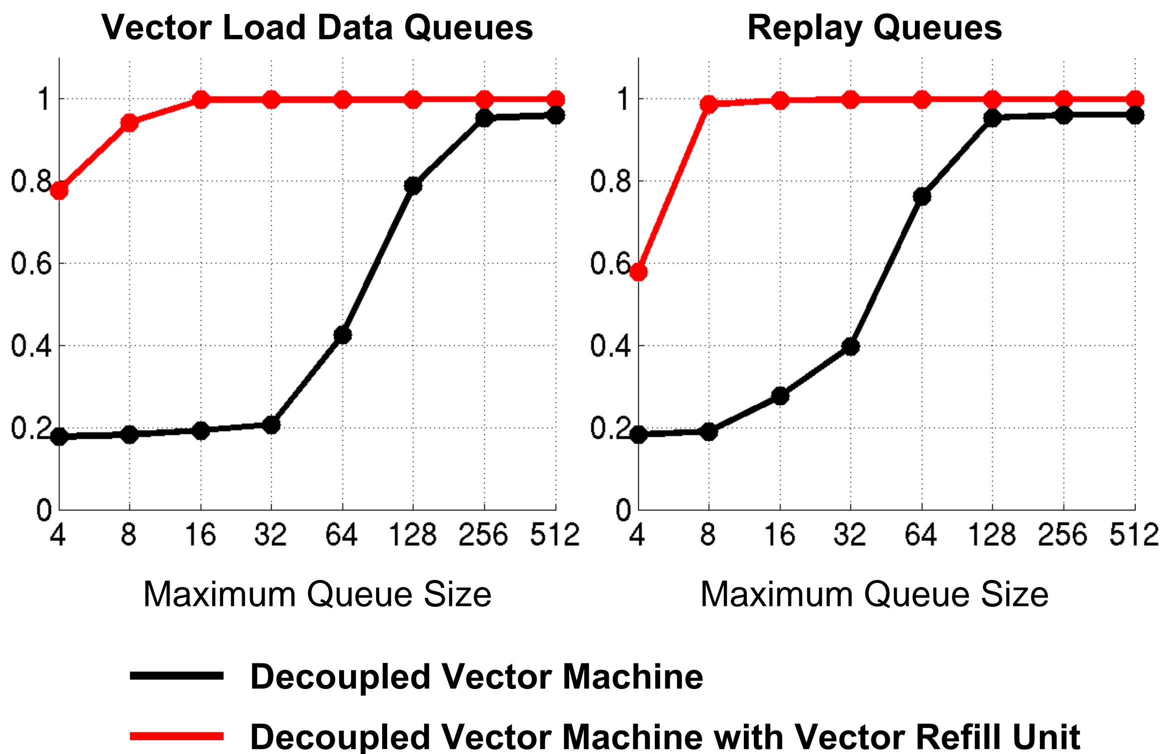


In this plot we see a similar trend with the number of replay queue entries.

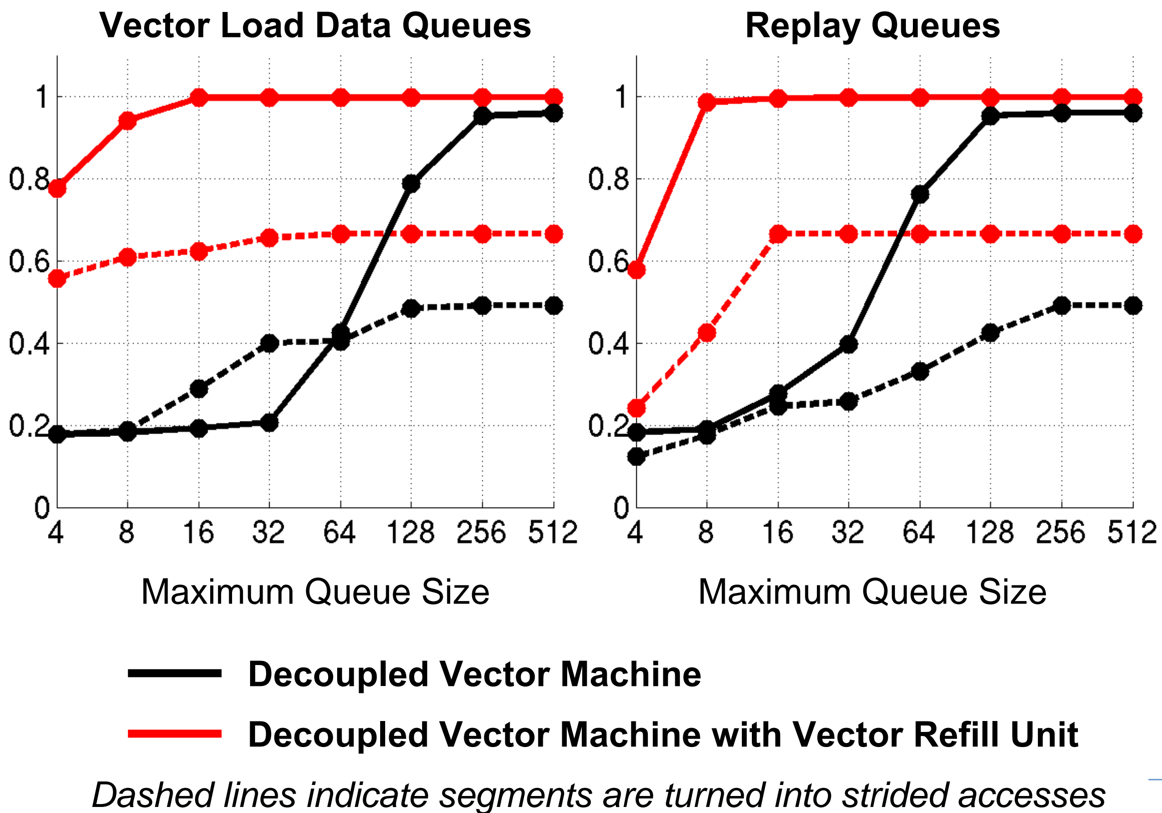


The difference is even more clear with HPG since HPG has a moderate bit of reuse – it loads each input element three times. This means that without the VRU, the decoupled vector machine must queue up 96 secondary misses before it can get to the next primary miss and thus issue the next refill request. So even with 512 VLDQ entries, the baseline decoupled vector machine is unable to achieve peak performance. With refill/access decoupling we only need 16 VLDQ entries and 8 replay queue entries. Notice that we cannot completely eliminate the VLDQ and replay queues – we need a few entries for a bit of pipelining and decoupling between the various units.

Cache Refill/Access Decoupling for Vector Machines, Christopher Batten, Ronny Krashinsky, Steven Gerding, and Krste Asanovic, 37*th International Symposium on Microarchitecture,* Portland, Oregon, December 2004

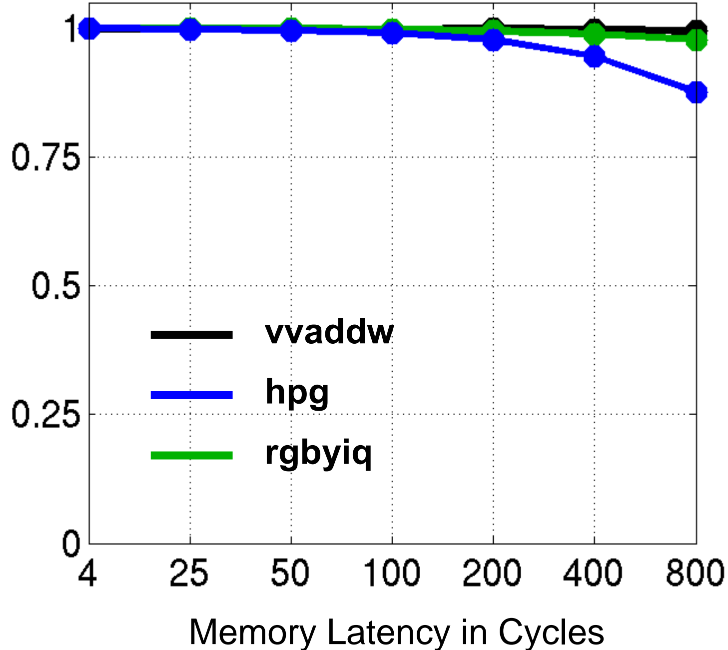


We see a similar trend for RGBYIQ with segments.



Here we also show the performance without segments. To emulate a traditional vector machine we turn segment accesses into multiple strided accesses, and you can see adding segments almost always increases the performance.

Performance with refill/access decoupling scales well with longer memory latencies



Configuration

- Includes the VRU
- Reasonable queues and buffering
- 8B/cycle mem bandwidth
- VLDQ and replay queues are a constant size
- Command queues and miss tags are scaled linearly with latency

Finally, I would like to look at how the performance of these kernels scale with longer memory latencies. For this experiment we use reasonable queue and buffering sizes and again the main memory bandwidth is 8 bytes per cycle. The VLDQ and replay queues are kept at a constant size. As we discussed earlier in the talk, the only resources which must scale when we add the VRU are the command queues and the miss tags. Overall this is a pretty boring plot, but that actually is a good thing. It means that these kernels are able to achieve close to peak performance even with an 800 cycle main memory latency. From the previous limit studies, it should be clear that a decoupled vector machine without the refill/access decoupling would scale drastically worse if given the same amount of resources.

Paper includes additional results and analysis

- 14 kernels with varying access patterns
- Performance versus number of miss tags
- Performance versus memory latency and bandwidth
- Comparison with an approximation of a scalar machine
- Various VRU and VLU throttling schemes

There are many more results and additional analysis in the paper. The paper includes fourteen kernels and examines ...

... the performance versus the number miss tags

... the performance versus memory latency and bandwidth

... a comparison with an approximation of a heavily decoupled scalar machine

... and various VRU and VLU throttling schemes

Related Work

- **Refill/Access Decoupling**
  - Software prefetching
  - Second-level vector register files [NEC SX, Imagine]
  - Speculative hardware prefetching [Jouppi90, Palacharla94]
  - Run-ahead processing [Baer91, Dundas97, Mutlu03]
- **Vector Segment Memory Accesses**
  - Streaming loads/stores [Khailany01, Ciricescu03]

I want to just very quickly touch on some related work - first with respect to refill/access decoupling. **Software prefetching** requires an intimate knowledge of the memory system at compile time and thus is not performance portable across different architectures. Refill/access decoupling is a microarchitectural technique and is transparent to the application. Many vector machines include **second-level vector register files** and essentially prefetch into this extra buffering. Although second-level vector register files avoid the tag overhead of caches they are significantly less flexible and waste storage for subword elements. ...

Related Work

- **Refill/Access Decoupling**
  - Software prefetching
  - Second-level vector register files [NEC SX, Imagine]
  - Speculative hardware prefetching [Jouppi90, Palacharla94]
  - Run-ahead processing [Baer91, Dundas97, Mutlu03]
- **Vector Segment Memory Accesses**
  - Streaming loads/stores [Khailany01, Ciricescu03]

... **Speculative hardware prefetching** uses stream buffers between the cache and main memory and on a miss prefetches extra cache lines into these buffers. These techniques usually do poorly on short vectors and can waste bandwidth on mispredictions. **Run-ahead processing** runs ahead after a cache miss and attempts to find additional cache misses which it can overlap with the first miss. Decoupling is a similar yet more efficient way of achieving the same effect. Refill/access decoupling should perform just as well as all of these other techniques but it does so in a simple and elegant way by exploiting the specific characteristics of vector machines. Vector segment memory accesses are similar in spirit to the streaming loads and stores found in Imagine and RSVP but are implemented in SCALE in a very different way.

Conclusions

- Saturating **large bandwidth-delay memory systems** requires many in-flight accesses and thus a great deal of access management state and reserved element data storage
- **Refill/access decoupling** and **vector segment accesses** are simple techniques which reduce these costs and improve performance

I would like to conclude with two take away points. First, saturating large bandwidth delay memory systems requires many in-flight elements and thus a great deal of access management state and reserved element buffering. Second, refill/access decoupling and vector segment accesses are simple techniques which reduce these costs and improve performance.