Implementing Virtual Memory in a Vector Processor with Software Restart Markers

Mark Hampton & Krste Asanovic

Computer Architecture Group

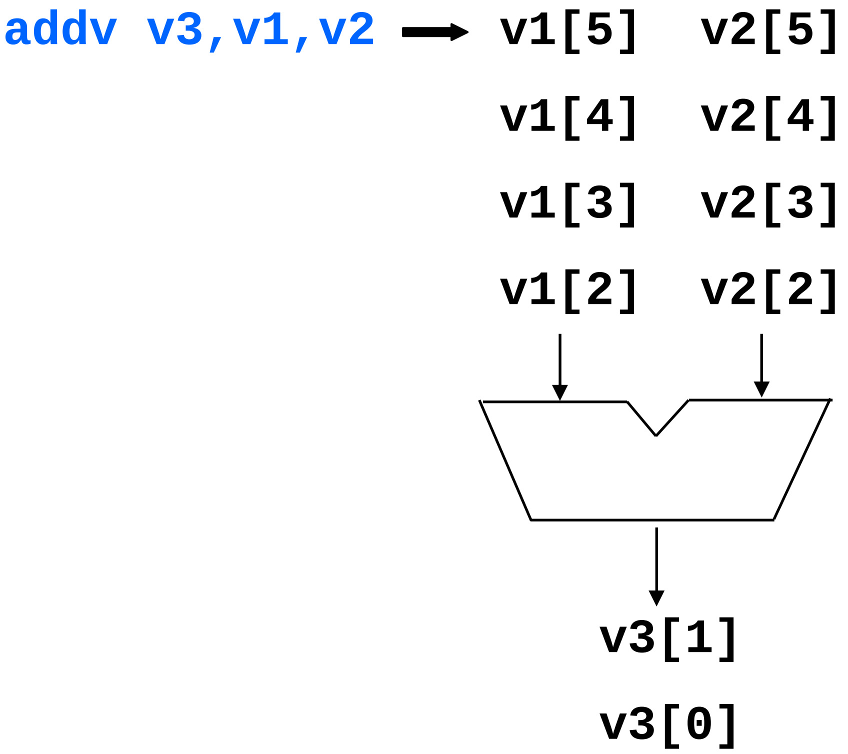
MIT CSAIL

CSAIL

MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Vector processors offer many benefits

One instruction triggers multiple operations



addv v3,v1,v2 →

v1[5] v2[5]

v1[4] v2[4]

v1[3] v2[3]

v1[2] v2[2]

v3[1]

v3[0]

Dependence checking performed by compiler

Reduced overhead in instruction fetch and decode

Regular access patterns

But difficulty supporting virtual memory has been a key reason why traditional vector processors are not more widely used

Demand-paged virtual memory is a requirement in general-purpose processors

A memory instruction uses
a virtual address…

load 0x802b10a4

↓

…which is then translated
into a physical address

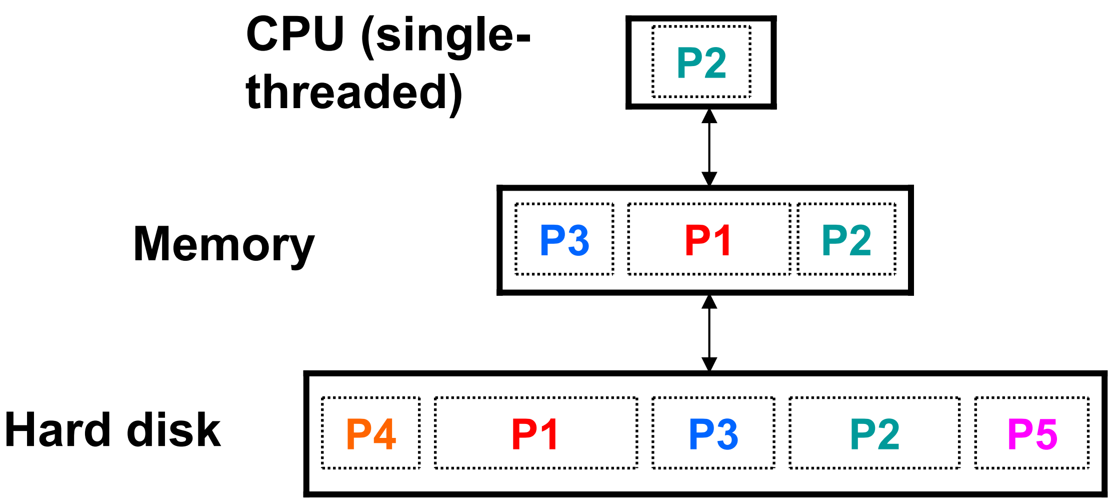
load 0x000c56e0

Requires OS and hardware support

- Protection between processes is supported
- Shared memory is allowed
- Large address spaces are enabled
- Code portability is enhanced
- Multiple processes can be active without having to be fully memory-resident

Demand paging allows multiple interactive processes to run simultaneously

The hard disk enables the illusion of a single large memory system



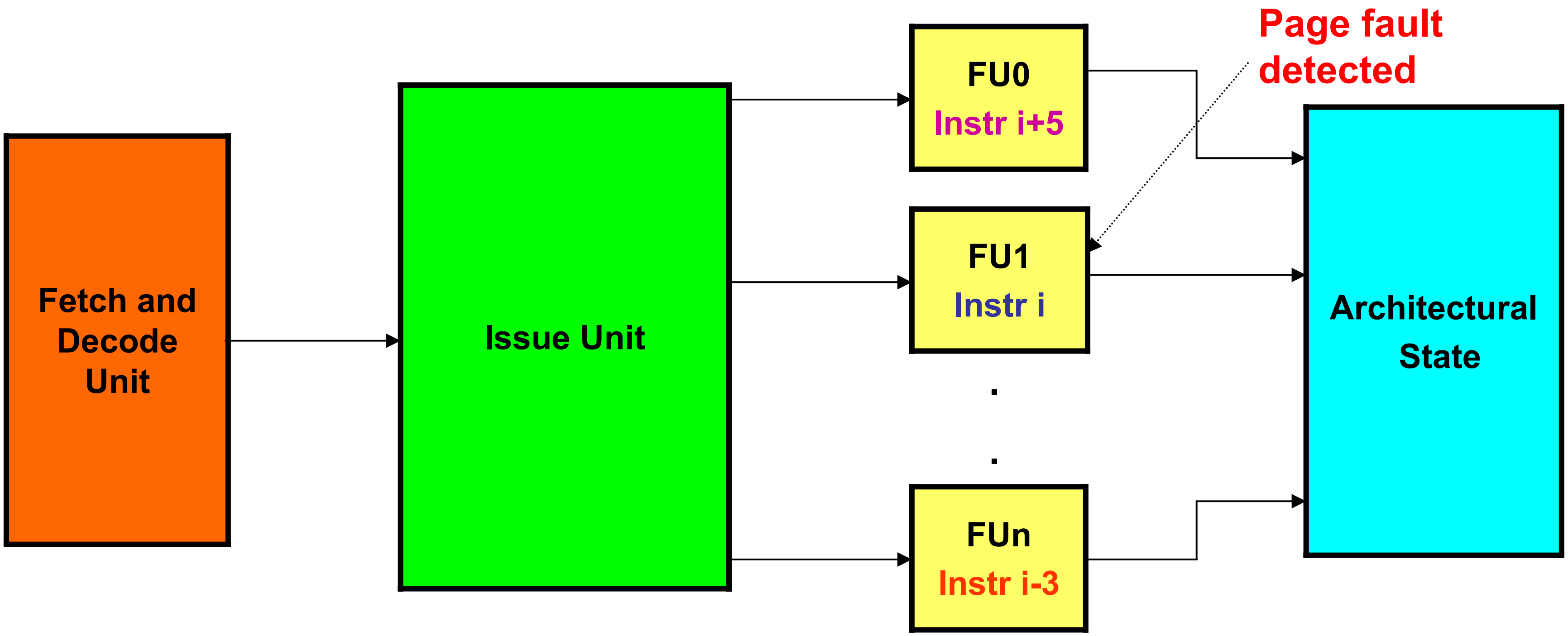
CPU executes one process at a time

Processes share physical memory…

…and use larger hard disk as “virtual” memory

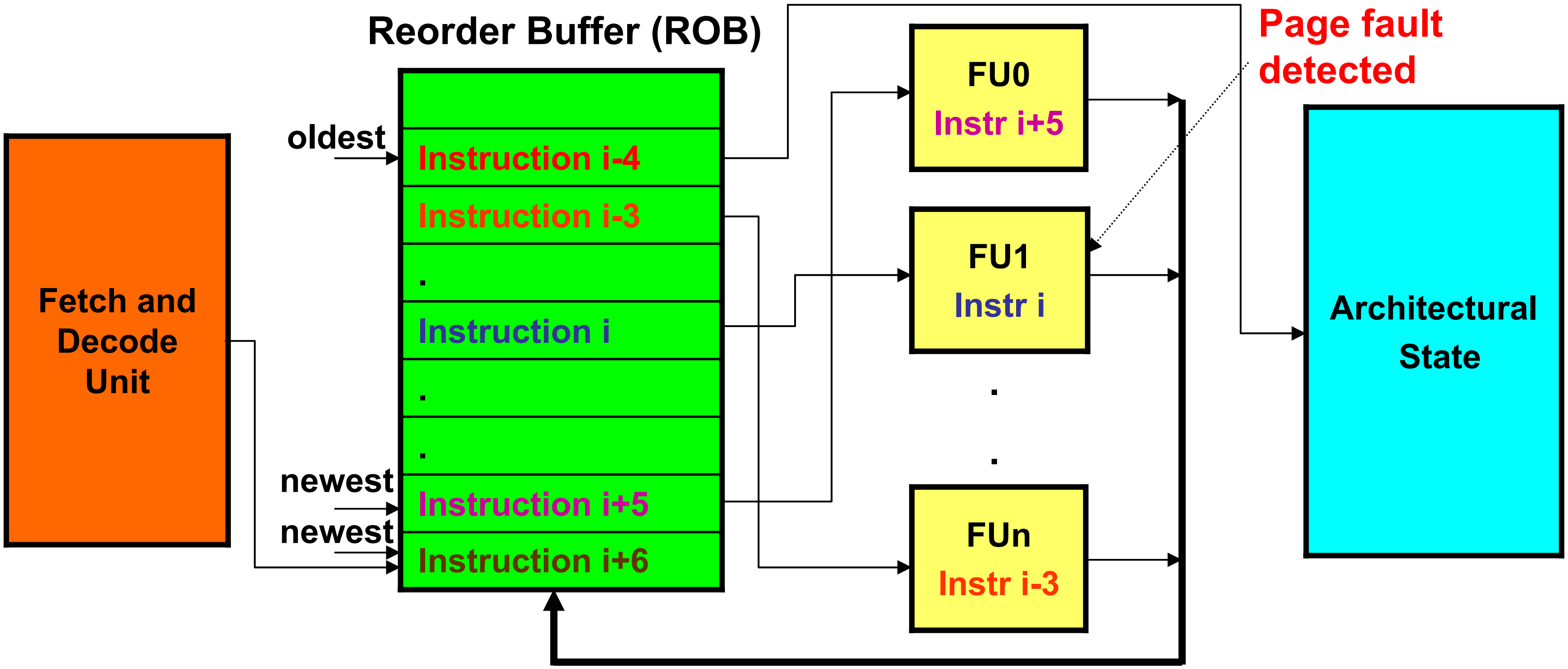
- If needed page is not in physical memory, trigger a page fault
- Page fault is very long-latency operation, and don’t want CPU to be idle, so perform context switch to bring in another process
- Context switch requires ability to save and restore CPU state needed to restart process

Parallel functional units complicate the saving and restoring of state



- Could save all pipeline state, but this adds significant complexity
- Precise exceptions only require architectural state to be saved by enforcing restrictions on commit

Precise exceptions preserve the illusion of sequential execution



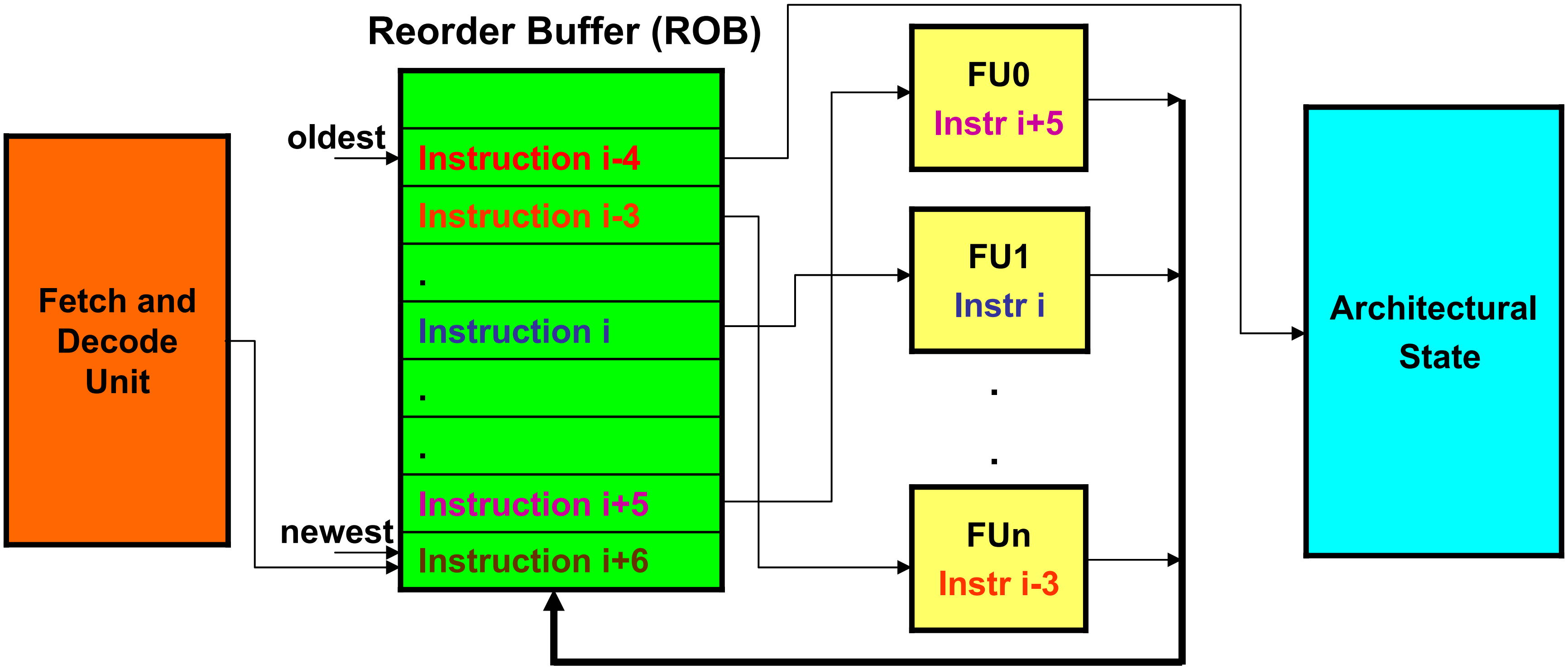
Fetch and decode in order

Execute and writeback results out of order (detect exceptions)

Commit results in order (handle exceptions)

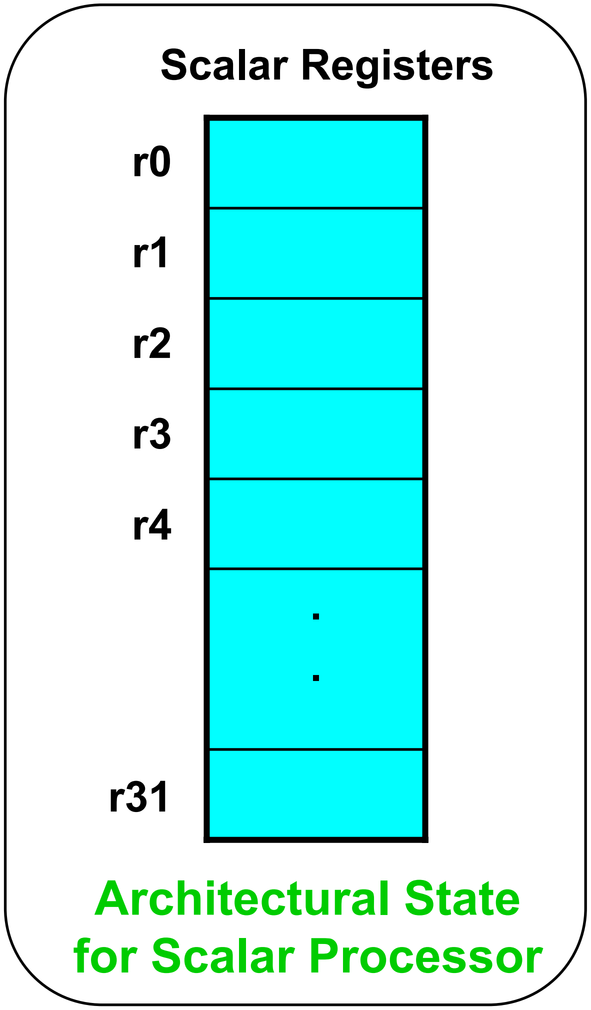
Key advantage is that restarting after exception is simple

Most precise exception designs support a relatively small number of in-flight operations



- Each in-flight operation needs a temporary buffer to hold result before commit
- Problem with vector processors is that a single instruction can produce hundreds of results!

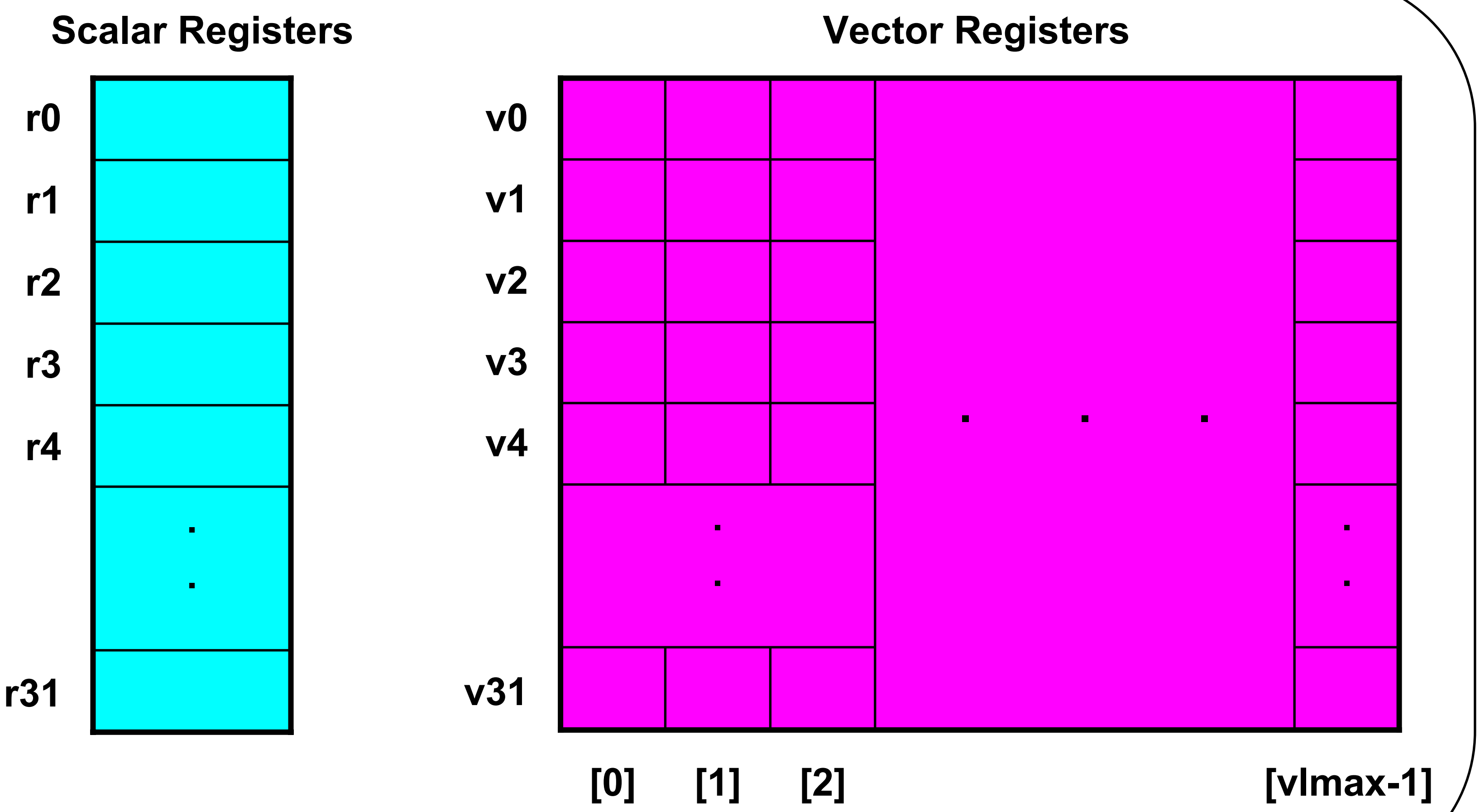
Vector processors also have a large amount of architectural state to preserve



Scalar Registers

Architectural State for Scalar Processor

Vector processors also have a large amount of architectural state to preserve



Scalar Registers

r0
r1
r2
r3
r4
. .
r31

Vector Registers

v0
v1
v2
v3
v4
. .
v31

[0] [1] [2] . . . [vlmax-1]

Architectural State for Vector Processor

This hurts performance and complicates OS interface

Our work addresses the problems with virtual memory in vector processors

- **Problem: All of the vector instruction results have to be buffered for in-order commit**

  Solution: We don’t buffer results; instead we use idempotent regions to allow out-of-order commit

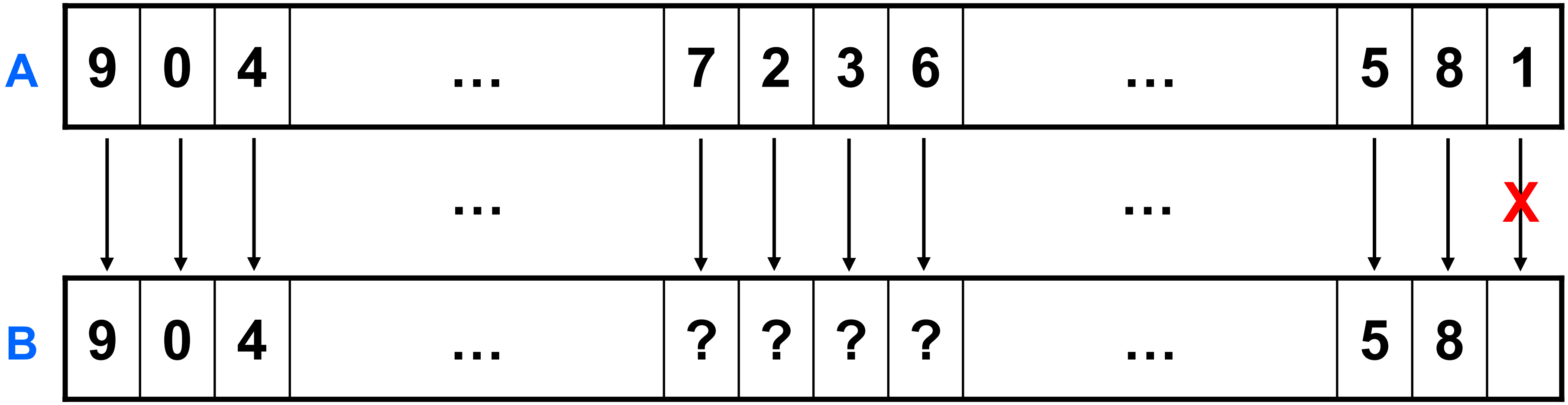
- **Problem: The vector register file significantly increases the amount of state to save**

  Solution: We don’t save vector registers; instead we recreate that state after an exception

The problem with parallel execution is knowing where to restart after an exception

Copying one array to another can be done in parallel:

But suppose something goes wrong

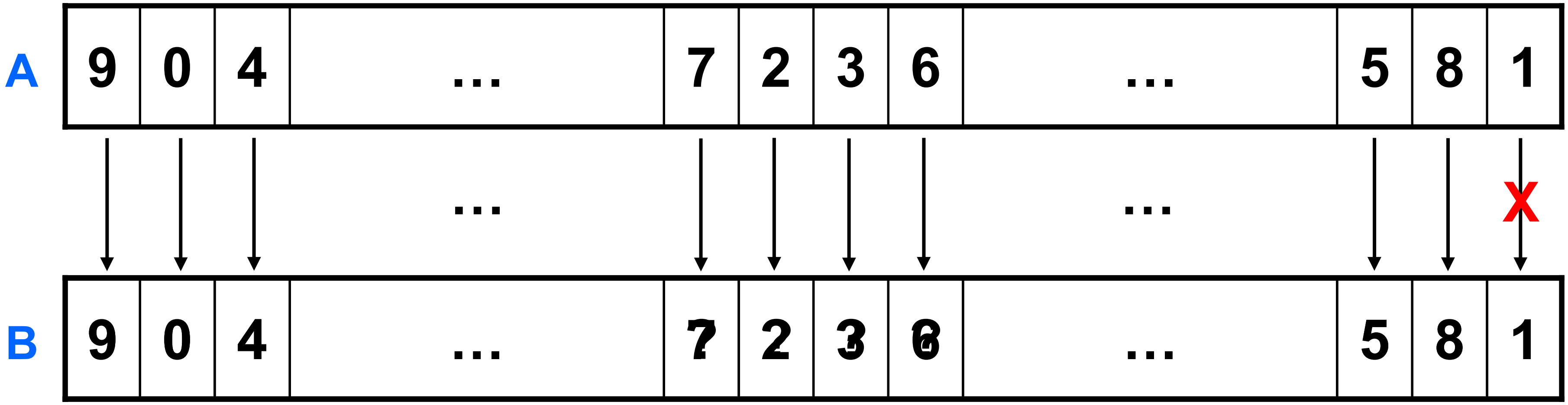


A	9	0	4	…	7	2	3	6	…	5	8	1
B	9	0	4	…	?	?	?	?	…	5	8	

Can’t simply restart from the faulting operation because all of the previous operations may not have completed

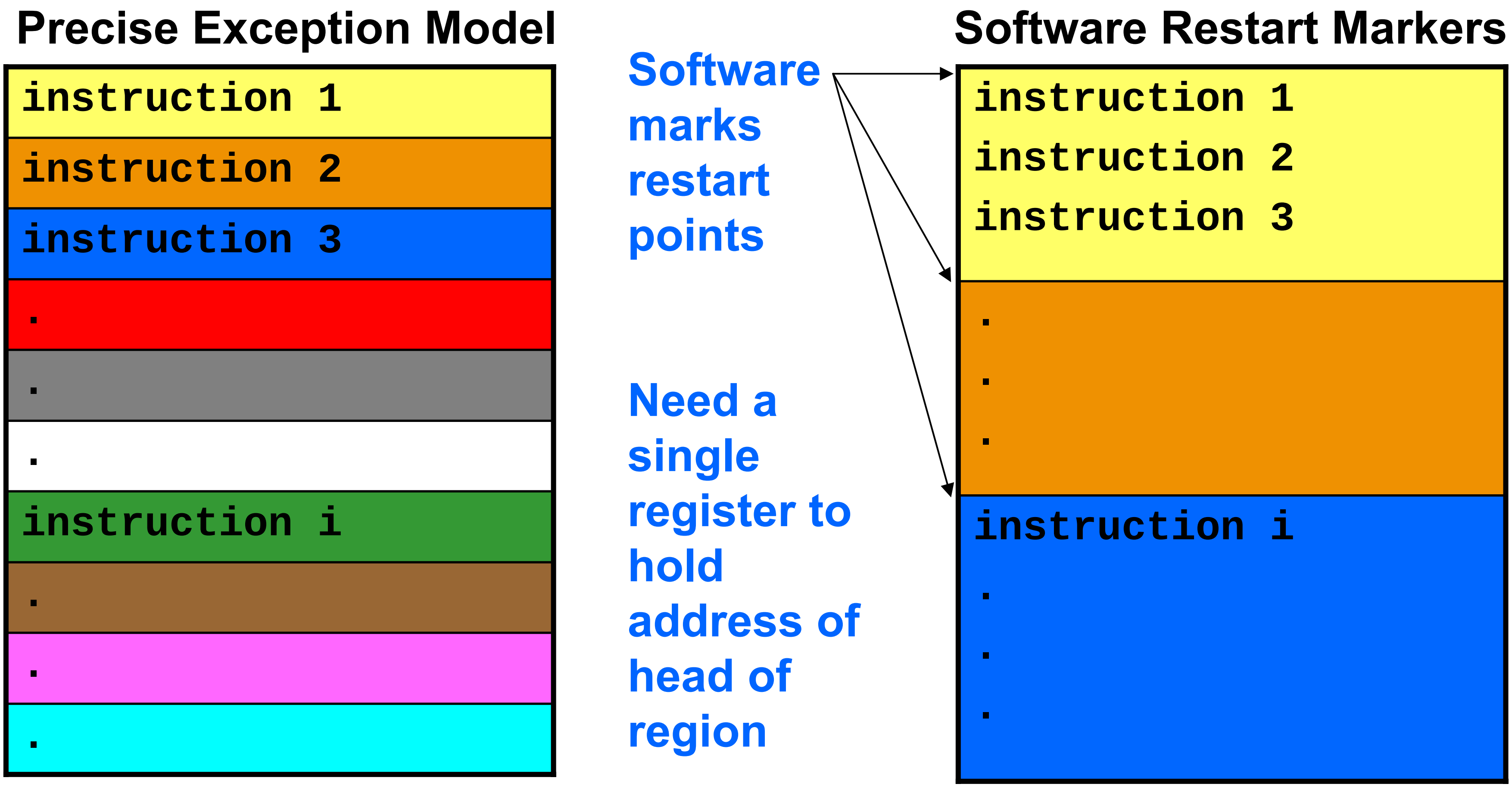
What if we didn’t worry about which instructions were uncompleted?

- In this example, A and B do not overlap in memory → original input data still exists
- Could copy everything again and still get same result



Only works if processor knows it’s safe to re-execute code, i.e. code must be idempotent

Software restart markers delimit regions of idempotent code



Precise Exception Model

instruction 1
instruction 2
instruction 3
.
.
.
instruction i
.
.
.

Software marks restart points

Need a single register to hold address of head of region

Software Restart Markers

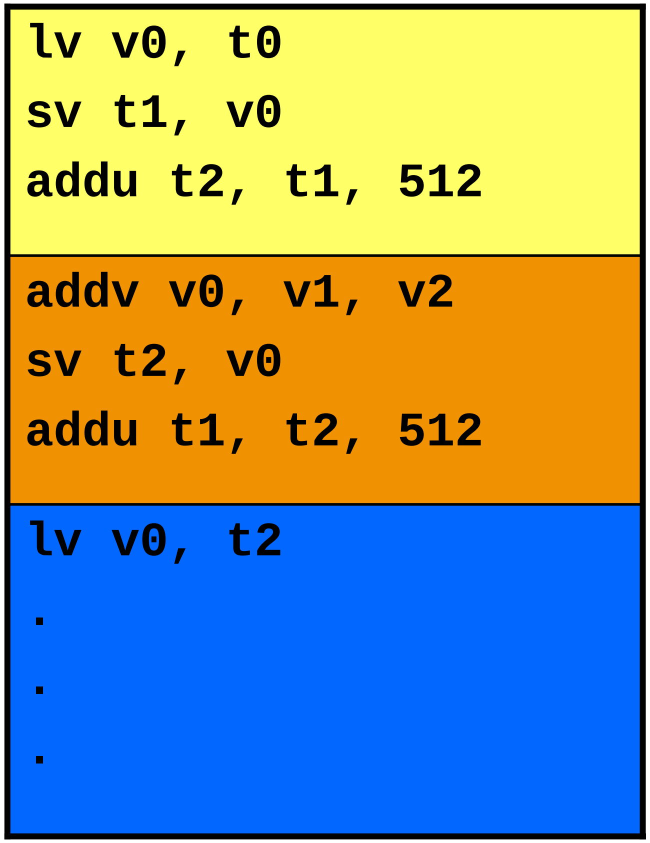
instruction 1 instruction 2 instruction 3
. . .
instruction i . . .

- Instructions from a single region can be committed out-of-order—no buffering required
- An exception causes execution to resume from head of region
- If regions are large enough, CPU can still exploit ample parallelism

Software restart markers also create a new classification of state

- “Temporary” state only exists within a single restart region, e.g. `v0`
- After exception, temporary state will be recreated and thus does not have to be saved
- Software restart markers allow vector registers to be mapped to temporary state

Software Restart Markers

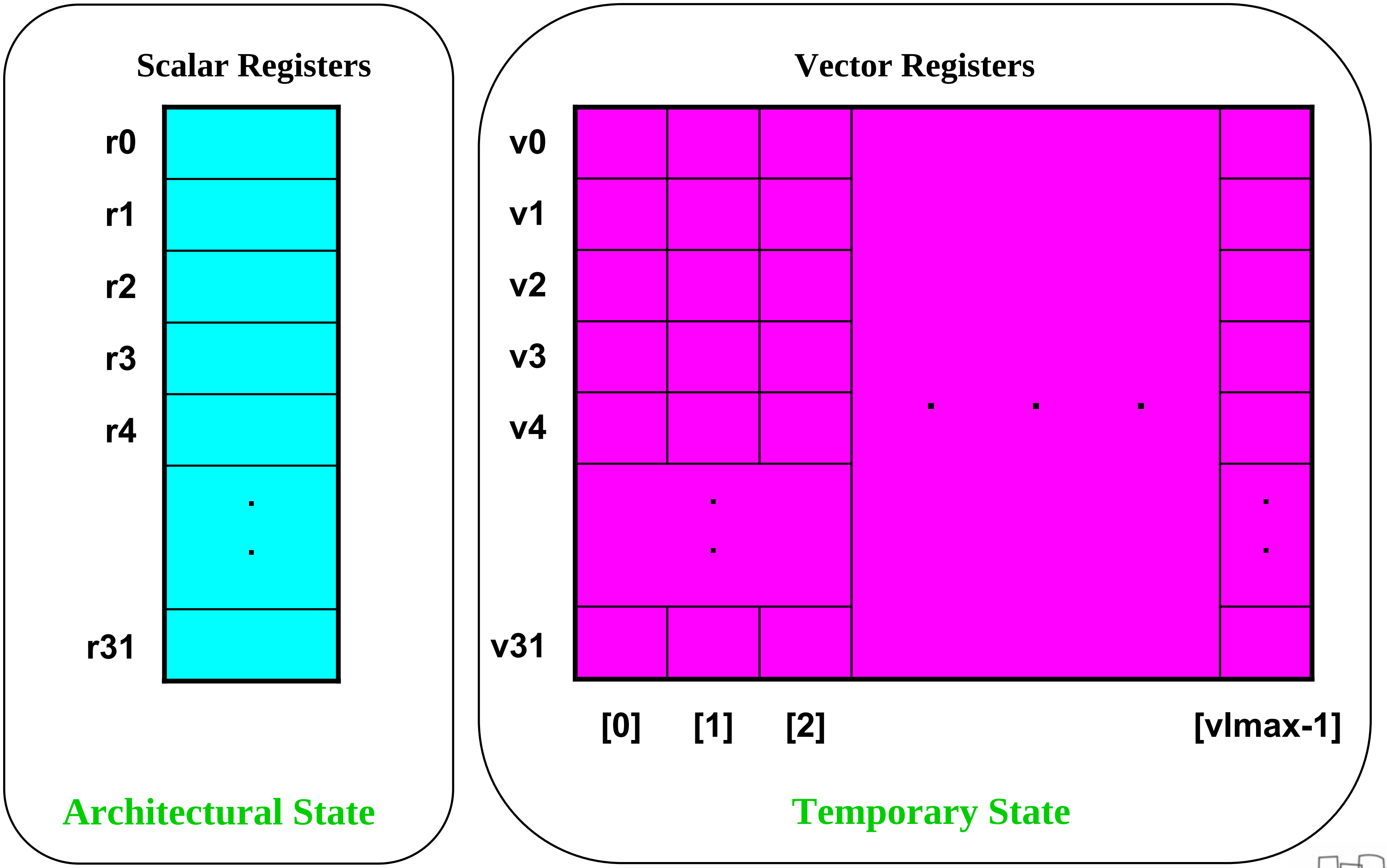


```
lv v0, t0
sv t1, v0
addu t2, t1, 512

addv v0, v1, v2
sv t2, v0
addu t1, t2, 512

lv v0, t2
.
.
.
```

Vector registers don't need to be preserved across exceptions



Creating restart regions can be done by making sure input values are preserved

Vectorized memcpy() loop

```
# void* memcpy(void *out, const void *in, size_t n);
loop: lv v0, a1      # Load from input
      sv a0, v0      # Store to output
      addiu a1, 512  # Increment pointers
      addiu a0, 512
      subu a2, 512   # Decrement counter
      bnez a2, loop  # Is loop done?
```

- **Want to place entire loop within single restart region, but argument registers are overwritten in each iteration**
- **Solution: Make copies of the argument registers**

Creating restart regions can be done by making sure input values are preserved

```
# void* memcpy(void *out, const void *in, size_t n);
begin restart region
      move t0, a0    # Copy argument registers
      move t1, a1
      move t2, a2
loop: lv v0, t1      # Load from input
      sv t0, v0      # Store to output
      addiu t1, 512  # Increment pointers
      addiu t0, 512
      subu t2, 512   # Decrement counter
      bnez t2, loop  # Is loop done?
done:
end restart region
```

This works for all functions with separate input and output arrays

But what if an input array is overwritten?

Vectorized loop for multiply_2() function

```
# void* multiply_2(void *in, size_t n);
loop: lv v0, a0               # Load from input
      mulvs.d v0, v0, f0  # Multiply vector by 2
      sv a0, v0               # Store result
      addiu a0, 512           # Increment pointer
      subu a1, 512            # Decrement counter
      bnez a1, loop           # Is loop done?
```

Can’t simply copy array to backup register

But what if an input array is overwritten?

```
# void* multiply_2(void *in, size_t n);
loop: lv v0, a0           # Load from input
      mulvs.d v0, v0, f0  # Multiply vector by 2
      sv a0, v0           # Store result
      addiu a0, 512       # Increment pointer
      subu a1, 512        # Decrement counter
      bnez a1, loop       # Is loop done?
```

Option #1: Copy input values to temporary buffer

But what if an input array is overwritten?

```
# void* multiply_2(void *in, size_t n);
# Allocate temporary buffer of size n pointed to by t2
      memcpy(t2, a0, a1)  # Copy input values to temp buffer
      begin restart region
      move t0, a0         # Get original inputs
      move t1, a1
      memcpy(a0, t2, a1)
loop: lv v0, t0           # Load from input
      mulvs.d v0, v0, f0  # Multiply vector by 2
      sv t0, v0           # Store result
      addiu t0, 512       # Increment pointer
      subu t1, 512        # Decrement counter
      bnez t1, loop       # Is loop done?
      end restart region
```

Option #1: Copy input array to temporary buffer

But what if an input array is overwritten?

```
# void* multiply_2(void *in, size_t n);
# Allocate temporary buffer of size n pointed to by t2
      memcpy(t2, a0, a1)  # Copy input values to temp buffer
      begin restart region
      move t0, a0         # Get original inputs
      move t1, a1
      memcpy(a0, t2, a1)
loop: lv v0, t0           # Load from input
      mulvs.d v0, v0, f0  # Multiply vector by 2
      sv t0, v0           # Store result
      addiu t0, 512       # Increment pointer
      subu t1, 512        # Decrement counter
      bnez t1, loop       # Is loop done?
      end restart region
```

Option #1: Copy input array to temporary buffer

Disadvantages: Space and performance overhead

Strip mining

Usually still faster than scalar code

But what if an input array is overwritten?

```
# void* multiply_2(void *in, size_t n);
loop: lv v0, a0           # Load from input
      mulvs.d v0, v0, f0  # Multiply vector by 2
      sv a0, v0           # Store result
      addiu a0, 512       # Increment pointer
      subu a1, 512        # Decrement counter
      bnez a1, loop       # Is loop done?
```

Option #2: Use scalar version when vector overhead is too large

But what if an input array is overwritten?

```
# void* multiply_2(void *in, size_t n);
      sltiu t0, a1, 16        # Is n less than threshold?
      bnez t0, scalar_version # If so, use scalar version
# Vector version of function with restart markers here
      .
      .
      j done
scalar_version:
# Scalar code without restart markers here
      .
      .
done: # return from function
```

Option #2: Use scalar version when vector overhead is too large

But what if an input array is overwritten?

```
# void* multiply_2(void *in, size_t n);
      sltiu t0, a1, 64        # Is n less than threshold?
      bnez t0, scalar_version # If so, use scalar version
# Vector version of function with restart markers here
      .
      .
      j done
scalar_version:
# Scalar code without restart markers here
      .
      .
done: # return from function
```

Option #2: Use scalar version when vector overhead is too large

Goal of our approach is to implement virtual memory cheaply while being able to handle the majority of vectorized code

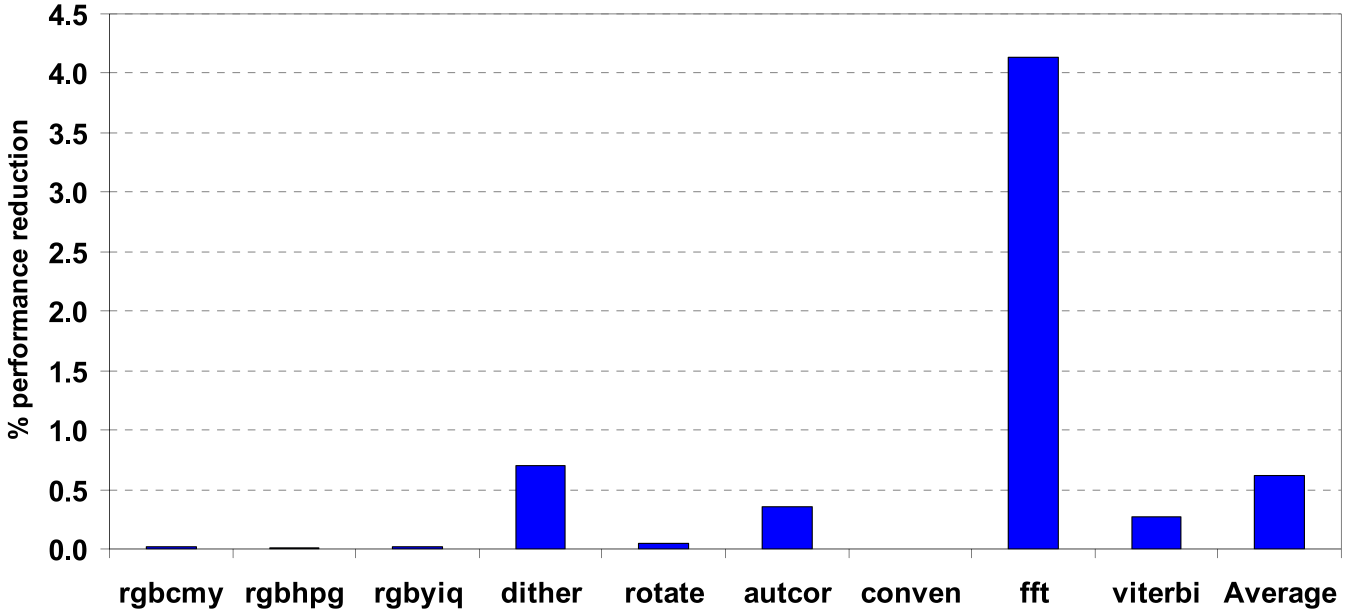
The compiler implementation takes advantage of existing techniques

- **We can create restart regions for scalar code with Trimaran, which uses region-based compilation [Hank95]**
- **Vectorizing compilers employ transformations to remove dependences, facilitating creation of restart regions**
- **We are currently working on a complete vectorizer**
    - SUIF frontend provides dependence analysis
    - Trimaran backend is used to generate vector assembly code with software restart markers
    - gcc creates final executables
    - This is a work in progress, so all evaluation is done using hand-vectorized assembly code

We evaluate the performance overhead of creating idempotent regions in actual code

- **Scale vector-thread processor [Krashinsky04] is target system**
  - Provides high performance for embedded programs
  - Only vector capabilities are used in this work
  - Microarchitectural simulator used for vector unit
  - Single-cycle magic memory emphasizes overhead of restart markers
- **A variety of EEMBC benchmarks serve as workload**
  - gcc used to compile code
  - Results shown for default 4-lane Scale configuration

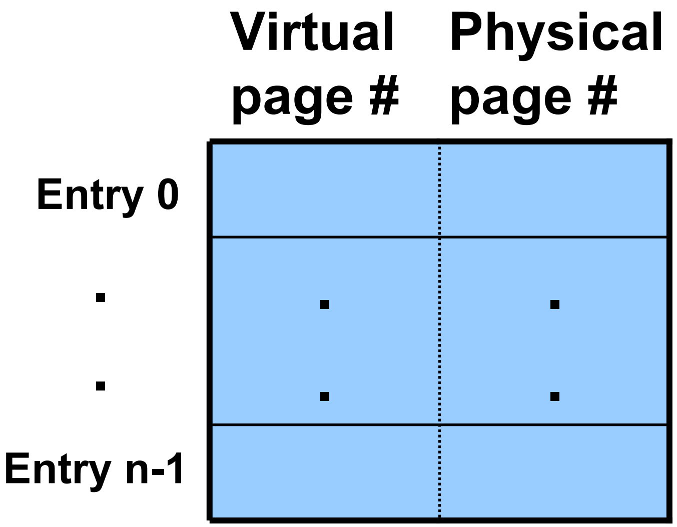
The performance overhead due to creating restart regions is small



- **For most benchmarks, performance reduction is negligible**
- **fft is an example of a fast-running benchmark with small restart regions**
- **An input array is preserved in viterbi to make the function idempotent**

But what about the overhead of re-executing instructions after a page fault?

- **Restarting after a page fault is not a significant concern**
  - Disk access latency is so high that it will dominate re-execution overhead
  - Page faults are relatively infrequent
- **However, to test our approach sufficiently, we examine TLB misses**



Virtual page #

Physical page #

Entry 0

.

.

Entry n-1

- **TLB holds virtual-to-physical address translations**
- **If translation is missing, need to walk the page table to perform TLB refill**
- **TLB refill can be handled either in hardware or software**

The method of refilling the TLB can have a significant effect on the system

- **Software-refilled TLBs cause an exception when a TLB miss occurs**
  - Typical designs flush the pipeline when handling miss
  - If miss handler code isn't in cache, performance is further hurt
  - For vector processors, the TLB normally has to be as large as the maximum vector length to avoid livelock
  - Advantage of this scheme is that it gives OS flexibility to choose page table structure
- **Hardware-refilled TLBs (found in most processors) use finite state machine to walk page table**
  - Disadvantage is that page table structure is fixed
  - Doesn't cause an exception, so performance hit is small (previous overhead results are an approximation of using system with hardware-refilled TLB)
  - No livelock issues

Although hardware refill is good for vector processors, we use software refill to provide a worst-case scenario

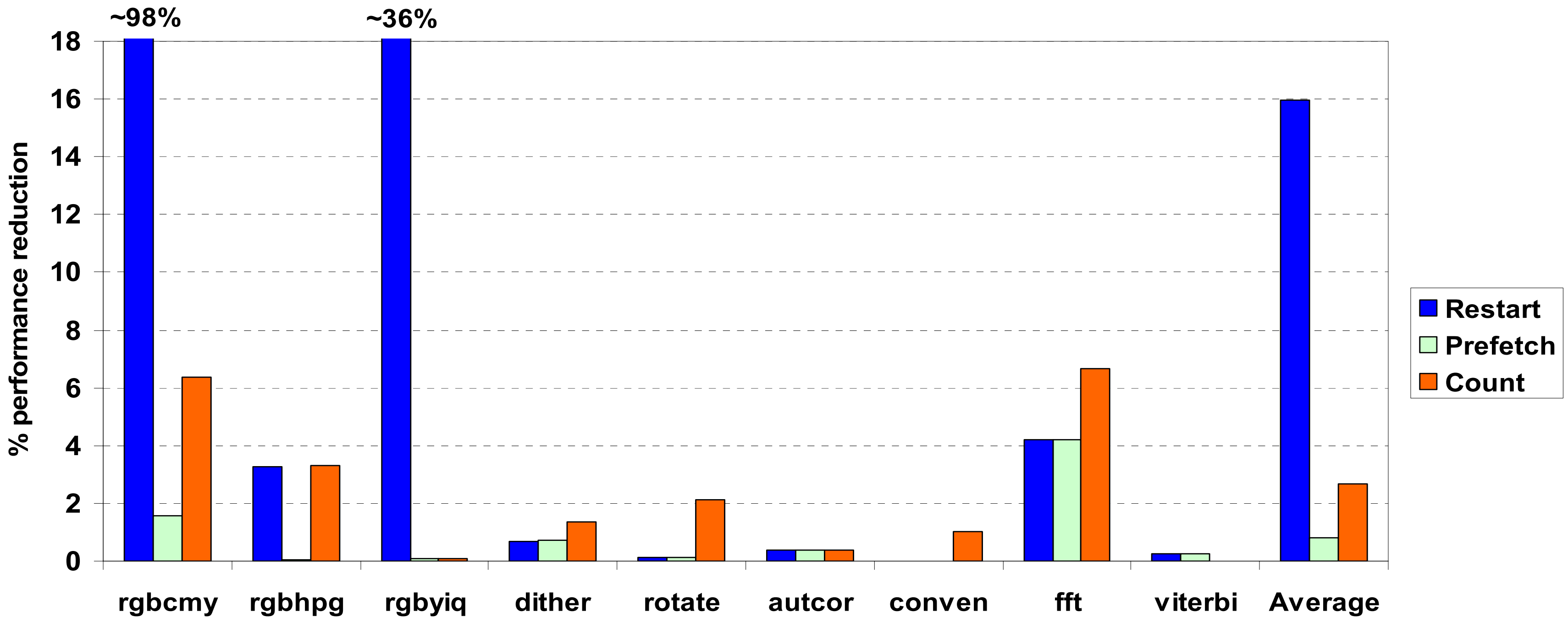
Performance optimizations can reduce the re-execution cost with a software-refilled TLB

- **Prefetching loads a byte from each page in the dataset before beginning the region**
  - Gets the TLB misses out of the way early
  - Disadvantage is extra compiler effort required
- **Counted loop optimization restarts after an exception from earliest uncompleted loop iteration**
  - Limits amount of repeated work
  - Compiler algorithm in paper

We evaluate the performance overhead of our worst-case scenario

- **Same simulation and compilation infrastructure is used**
- **Virtual memory configuration uses standard MIPS setup with software refill**
  - Default 64-entry MIPS TLB for control processor
  - 128-entry TLB for vector unit
  - Fixed 4 KB page size—smallest possible for MIPS
  - All page tables modeled, but no page faults
- **Two additional overhead components are introduced**
  - Cost of handling TLB miss (usually negligible)
  - Cost of re-executing instructions after a TLB miss

The performance overhead of using software-refilled TLB is small with optimizations



- Original design does not perform well with large datasets
- Prefetching incurs smallest degradation
- Counted loop optimization has small overhead, but still leads to some re-executed work

Related Work

- IBM System/370 [Buchholz86] only allowed one in-flight vector instruction at a time, hurting performance
- DEC Vector VAX [DEC91] saved internal pipeline state, causing performance and energy problems
- CODE [Kozyrakis03] uses register renaming to support virtual memory, while our scheme can be used in processors with no renaming capabilities
- Sentinel scheduling [Mahlke92, August95] uses idempotent code and recovery blocks, but for the purpose of recovering from misspeculations in a VLIW architecture
- Checkpoint repair [Hwu87] is more flexible than our software "checkpointing" scheme, but incurs more hardware overhead

Concluding Remarks

- **Traditional vector architectures have not found widespread acceptance, in large part because of the difficulty in supporting virtual memory**
- **Software restart markers enable virtual memory to be implemented cheaply**
  - They allow instructions to be committed out-of-order
  - They reduce amount of state to save in event of context switch
- **Our approach reduces hardware overhead while incurring only a small performance degradation**
  - Average overhead with hardware-refilled TLB less than 1%
  - Average overhead with software-refilled TLB less than 3%