Implementing Virtual Memory in a Vector Processor with Software Restart Markers

Mark Hampton
MIT CSAIL
32 Vassar Street, Cambridge, MA 02139
mhampton@csail.mit.edu

Krste Asanović
MIT CSAIL
32 Vassar Street, Cambridge, MA 02139
krste@csail.mit.edu

ABSTRACT

Traditional vector architectures often lack virtual memory support because it is difficult to support fast and precise exceptions for these machines. In this paper, we propose a new exception handling model for vector architectures based on software restart markers, which divide the program into idempotent regions of code. Within a region, the processor can commit instruction results to the architectural state in any order. If an exception occurs, the machine jumps immediately to the exception handler and kills ongoing instructions. To restart execution, the operating system has just to begin execution at the start of the region. This approach avoids the area and energy overhead to buffer uncommitted vector unit state that would otherwise be required with a high-performance precise exception mechanism, but still provides a simple exception handling interface for the operating system. Our scheme also removes the requirement of preserving vector register file contents in the event of a context switch. We show that using our approach causes an average performance reduction of less than 3% across a variety of benchmarks compared with a vector machine that does not support virtual memory.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.4.2 [**Operating Systems**]: Storage Management—*virtual memory*

General Terms

Design, Performance

Keywords

Vector processors, Exception handling

1. INTRODUCTION

ICS06, June 28-30, Cairns, Queensland, Australia.

It is well established that vector architectures can provide high performance with relatively low complexity for data parallel codes across a wide range of application domains ranging from supercomputing [27, 9] to embedded media processing [22, 26, 2, 18]. However, extensive vector capabilities have yet to appear in recent commercial products other than newer versions of traditional large-scale vector supercomputers [5, 16]. Although short-vector multimedia extensions such as Intel's MMX/SSE and Motorola/IBM's Altivec provide some of the benefits of vector processing, they cannot provide the same degree of complexity reduction in instruction fetch, decode and register renaming, or the same improvement in sustained memory performance as a traditional long-vector architecture.

One of the factors that has hindered the widespread adoption of vector architectures is the difficulty of implementing virtual memory in these machines. Demand-paged virtual memory is considered a requirement in modern general-purpose computing systems as it allows many interactive processes to be active simultaneously while using only the physical memory required for the current working set. Vector supercomputers, however, have typically not supported virtual memory [27, 30, 16]. Supercomputer jobs are designed to fit into physical memory so that expensive CPUs do not have to wait while pages are swapped in from disk, and batch scheduling is also common where a queue manager ensures the set of running jobs can fit in physical memory.

The main requirement to support virtual memory is that the processor state can be saved to memory when a page fault is detected to allow a different process to use the processor. The original context must then be able to be restored from memory and restarted once the missing page is loaded from disk. Modern processors usually support *precise exceptions*, where only the architectural state of the processor needs to be saved to memory. The operating system (OS) can then restart a job by simplying reloading the architectural state and jumping to the program counter (PC) where execution was interrupted. As an alternative to precise exceptions, the microarchitectural state of the machine can be saved and restored around a page fault (we call this *microarchitectural swapping*). This approach considerably complicates the OS view of the machine, and usually adds significantly to interrupt response time. In addition, microarchitectural swapping can add considerable hardware area, delay, power, and design complexity, as circuit datapaths and control logic may have to be added to allow read and write of otherwise inaccessible state.

Vector machines add two major complications to virtual

memory handling. First, many vector units are designed to run decoupled from the scalar unit, which runs far ahead fetching instructions for subsequent loop iterations [10], and vector instructions can be long running, so a vector instruction might complete hundreds of clock cycles after a scalar instruction that follows it in program order. This makes supporting precise exceptions through conventional register renaming and reorder buffer techniques very expensive due to the length and number of instructions in flight. Second, vector units have far more architectural and microarchitectural state than scalar units. This adds to the cost of renaming if vector registers are renamed, but also lengthens context switch times for both precise exceptions and microarchitectural swapping.

In this paper, we introduce the technique of *software restart markers* to support virtual memory in a vector processor. This is a combined hardware and software scheme that takes advantage of the nature of vector loops to reduce the cost of adding a vector unit to a scalar machine that already supports virtual memory. Software divides the vector instruction stream into *idempotent* regions of code. The hardware can then commit instructions out of order within each region. At the end of each region, the machine must impose a trap barrier to wait for all preceding potentially faulting instructions to clear exception checks before allowing instructions from the following region to begin committing. If an exception is detected, hardware simply discards the entire vector unit state before jumping to an exception handler. However, because the region was idempotent, the OS can simply restart execution at the beginning of the region once the faulting page is loaded. Our approach removes the need to save and restore vector register state around page faults because the vector state will be recreated from the code at the restart point, enabling the operating system to effectively ignore the addition of the vector unit. This retains the same simple restart model of a scalar machine with precise exceptions, and avoids additional interrupt overhead.

2. SOFTWARE RESTART MARKERS

In this section, we present the design of our software restart scheme. We also discuss some of the issues with our scheme and present possible solutions.

2.1 Out-of-Order Instruction Commit

Implementing a precise exception model effectively requires instructions to be committed in program order. This can significantly inhibit parallel execution. To illustrate this, consider the memcpy function, for which the prototype is given below.

```
void* memcpy(void *out, const void *in, size_t n);
```

This function copies the first `n` bytes of `in` to the first `n` bytes of `out`. The input and output arrays do not overlap in memory, otherwise the behavior is undefined. A simple implementation of this function would execute a loop that copies a value from one array to the other in each iteration. As there are no data dependences, all iterations of the loop could theoretically be executed in parallel.

However, under the precise exception model, although the loop iterations can be executed in parallel, they have to be committed sequentially. Consider Figure 1(a), which presents an abstract representation of the instructions in

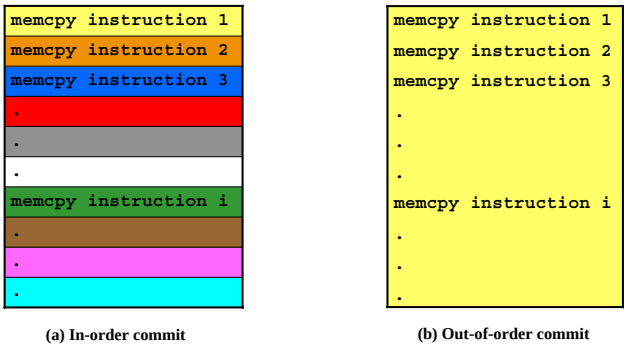

Figure 1: (a) Exception model with in-order commit. There is an implicit trap barrier at the end of each shaded region, which each contain one instruction. (b) Exception model with out-of-order commit. The number of trap barriers has been drastically reduced.

the `memcpy` function. Each instruction has an associated trap barrier, which is indicated by a different shaded region. Thus, the first instruction from iteration 100 cannot commit until all of the instructions from the previous 99 iterations have committed. This means that its result, if any, has to be buffered either in the physical register file or the store queue. Since the processor has finite available resources, if a particular instruction causes a stall—perhaps due to a cache miss—then later instructions might not even be able to issue, thus degrading performance.

For best performance, the `memcpy` instructions should be allowed to commit out-of-order, as shown in Figure 1(b). This depicts a *region* of instructions; regions must be committed in order, but within each region, instructions may be committed in any order that preserves dependences. The problem with out-of-order commit is that upon returning from an exception, the processor has to determine which instructions still need to be executed. This problem can be avoided for the `memcpy` function by taking advantage of the fact that the input array is not overwritten. After handling an exception, if the processor restarts the function from the beginning, the same result will be produced. All that the processor needs is some method to determine where to restart execution after handling the exception.

We generalize this observation to create an alternative to the precise exception model. We allow software to explicitly mark points in the instruction stream where restart is required, dividing the code into *restart regions*, which permit a coarser commit granularity than single instructions. At the beginning of each region, we insert a special trap barrier instruction that waits for all previous instructions to commit. It then updates an OS kernel-visible register, the *restart PC*, with the address of the subsequent instruction. If an exception occurs in the middle of a region, the OS kernel can restart the process by simply jumping to the restart PC. This requires that software ensure the code in each restart region is idempotent—i.e. it can be re-executed multiple times without changing the result. Another trap barrier instruction is placed at the end of the region to indicate that the processor can resume conventional precise exception semantics. However, if two regions are placed back-to-back, then only a single barrier is needed between them.

To illustrate how restart regions are formed, consider the sample vectorized loop shown in Figure 2, which could be part of an implementation of `memcpy`. This example uses the VMIPS instruction set [12]. Each vector register holds 64 8-byte elements, so each loop iteration copies 512 bytes.

```
# void* memcpy(void *out, const void *in, size_t n);
loop: lv v1, a1            # Load the input vector
      sv a0, v1            # Copy the data to the output vector
      addiu a1, 512        # Increment the input base register
      addiu a0, 512        # Increment the output base register
      subu a2, 512         # Decrement the number of bytes remaining
      bnez a2, loop        # Is loop done?
```

Figure 2: Vectorized memcpy loop.

As shown earlier, the `memcpy` function could be contained within a single restart region. However, although the high-level C definition of `memcpy` is idempotent, the VMIPS code that we present is not, as the argument registers are overwritten in each loop iteration. This can be handled by copying the argument register values to other registers at the beginning of the restart region. The registers with the copied values can then be used to execute the loop. Figure 3 shows the `memcpy` loop inside a restart region. Although copying register values can increase the register pressure, the effect will often be insignificant—typically only the number of iterations and a few memory pointers will have to be preserved. As we will see in the results, the performance overhead caused by additional register pressure is usually small compared to the total execution time of the function.

```
begin restart region
      move t0, a0          # Copy the argument registers
      move t1, a1
      move t2, a2
loop: lv v1, t1            # Load the input vector
      sv t0, v1            # Copy the data to the output vector
      addiu t1, 512        # Increment the input base register
      addiu t0, 512        # Increment the output base register
      subu t2, 512         # Decrement the number of bytes remaining
      bnez t2, loop        # Is loop done?
done:
end restart region
```

Figure 3: Vectorized memcpy loop with software restart markers.

Note that in the above loop, no vector registers are live across restart region boundaries. Unless techniques such as interprocedural dependence analysis [25] are used, vector registers are typically only live within a single function (and usually only within a loop iteration). By creating a restart region that encompasses an entire vectorized function, the vector register state will be recreated each time the region is restarted. We exploit this fact to eliminate the saving and restoring of vector registers on a context switch. The vector registers are treated as *temporary state*, which is only valid within a restart region. This both reduces memory required to save the process state and improves context switching time. It also allows the OS to be ignorant of the presence of the vector unit in the user-level architectural state.

Figure 4 summarizes the differences in the exception handling process between the precise exception model and software restart markers. Once an exception is detected, software restart markers simplify hardware by allowing the processor to simply abort pending instructions, rather than enforcing in-order instruction commit. This also avoids the time to undo imprecise state changes in a checkpointing scheme [13]. Although software restart markers require an additional register to be saved and restored—the restart PC—they remove the need to preserve the contents of the vector register file. These advantages are obtained without complicating the exception handler or the method of resuming process execution.

Handling an Exception Precisely

Hardware must…	Software must…
1. Detect exception	
2. Commit earlier pending instructions	
3. Undo imprecise state changes	
	4. Save architectural state (including exception PC and vector register file)
	5. Run exception handler
	6. Restore architectural state
	7. Resume process by jumping to exception PC

Handling an Exception with Software Restart Markers

	Hardware must…	Software must…
	1. Detect exception	
	2. Abort pending instructions	
Before handler		3. Save architectural state (including exception PC and restart PC—but *not* vector register file)
During handler		4. Run exception handler
After handler		5. Restore architectural state
		6. Resume process by jumping to restart PC

Figure 4: Comparison between the exception handling process with precise exceptions and with software restart markers.

2.2 Obstacles to Restart Region Generation

There are two primary impediments to the use of software restart markers: overwritten inputs and livelock.

2.2.1 Overwritten Inputs

A restart region has to be idempotent so that it can be executed more than once while maintaining correctness. A sufficient, but not necessary, condition for idempotency is that the set of all external sources (registers and memory) read by the region is disjoint from the set of destinations written by the region (note that it is acceptable to overwrite a value produced within the region). This is not a necessary condition as shown by the following instruction:

```
andi r1, r1, 3
```

Although the `andi` instruction overwrites its input data, it does so with an idempotent operation (masking out all but the bottom two bits). However, typically the input values to a restart region will need to be preserved.

As shown in Figure 3, input registers to a region can simply be copied to spare registers. Input memory values that are overwritten are more difficult to handle, as the amount of data is often much larger. Although a vectorizing compiler may apply transformations to remove dependences that inhibit restart region formation, certain types of code will still cause problems. Consider the vectorized function in Figure 5, which takes an input array `in` of size `n` and multiplies each element by 2, overwriting the input values. One approach is to to copy `in` to a temporary array, and then create a new loop—contained within a single restart region—that loads values from the temporary array. However, this

could introduce significant performance and memory overhead, depending on the number of elements to be copied.

```
# void* multiply_2(void *in, size_t n);
loop: lv v1, a0            # Load the input array
      mulvs.d v1, v1, f0   # Multiply array by 2
      sv v1, a0            # Store result
      addiu a0, 512        # Increment the base register
      subu a1, 512         # Decrement the number of bytes remaining
      bnez a1, loop        # Is loop done?
```

Figure 5: Vectorized multiply_2 loop.

Alternatively, we can divide each loop iteration into two restart regions, as shown in Figure 6. This sacrifices performance in machines that implement vector chaining, as the store will not be able to chain to the multiply. Additionally, performance will be degraded in systems that otherwise decouple the scalar unit and vector unit, as the scalar operations at the end of the loop have to wait for the vector store to complete, which in turn delays the vector load in the subsequent iteration.

```
loop: begin restart region
      lv v1, a0            # Load the input array
      mulvs.d v1, v1, f0   # Multiply array by 2
      begin restart region
      sv v1, a0            # Store result
      end restart region
      addiu a0, 512        # Increment the base register
      subu a1, 512         # Decrement the number of bytes remaining
      bnez a1, loop        # Is loop done?
```

Figure 6: Vectorized multiply_2 loop with software restart markers.

A separate issue with the code in Figure 6 is that for correctness the vector register state must now be saved and restored around a context switch, removing one of the advantages of the basic software restart scheme. Software can set a status bit on the begin region instruction to inform the kernel whether the vector register state is live on a fault. This status bit must be saved along with the process state to indicate if the vector register state should be reloaded when swapping the process back in.

2.2.2 Livelock

The system must ensure forward progress in the face of finite resources. For example, for a demand-paged virtual memory system the number of memory pages touched in a region must be less than the number of available physical pages to allow execution to proceed through the entire region without a page fault. In practice, this particular restriction is not usually significant.

A more significant restriction arises if TLB refills are handled in software, where the number of different pages accessed within a restart region must be less than the number of available TLB entries so that the region can run from start to finish without incurring a TLB miss.

Despite the performance overhead of software-managed TLBs, they are still popular because they give the OS the flexibility of choosing a page table structure [29]. Additionally, a vector unit may be added to an existing processor that handles TLB refills in software [9]. We therefore investigate alternative approaches that reduce the overhead of avoiding livelock with a software-managed TLB.

Since unit-stride vector memory accesses are so common [12], it is overly conservative to assume that each operation within a vector memory instruction touches a different page. For example, in the `memcpy` function, copying a single page of data would typically require several vector memory instructions, since the accesses are sequential. Where the compiler can determine strided memory accesses are used and the stride is small, the compiler can often create restart regions without having to account for the possibility of livelock. In order to handle datasets that are too large to be completely mapped by the TLB, the standard technique of *strip mining* can be employed so that a new restart region is started before there is any danger of overflowing the TLB. For code with indexed memory accesses or strided memory accesses with large strides, restart region formation may be significantly restricted if software TLB refill is used, possibly leading to performance degradation.

Hardware page table walkers are commonly used to refill TLBs for processors that execute large numbers of concurrent operations [14]. These avoid any TLB livelock issues, as TLB misses do not trigger exceptions. Note that handling TLB refills in hardware does not eliminate the need for software restart markers, as the process still has to be able to restart after a page fault or a memory protection violation.

2.3 Performance Optimizations

Thus far we have not addressed the potential performance overhead if multiple exceptions occur within a restart region, causing the same instructions to be executed more than once. This can be particularly harmful in a system with a software-managed TLB if a restart region contains accesses to several different data pages. For example, if maximally sized restart regions are used with the `memcpy` routine, then there will be a significant overhead from performing the same memory transfers multiple times. After copying the first page of data, a TLB miss will occur when the second page of data is accessed, causing the program to begin from the head of the restart region and copy the first page again. The subsequent TLB miss on the third page of data will cause the first and second pages to be copied again, and this pattern will continue, with the overhead becoming worse as more of the restart region is processed. Additionally, if a context switch occurs, then when the original process resumes, all of the TLB mappings will have been flushed. We discuss two possible solutions to this problem.

2.3.1 Prefetching for Software-Managed TLBs

One method to reduce performance overhead of repeated restarts with a software-managed TLB is to prefetch the mappings that will be needed for each region. As long as the data pages that will be accessed within each region can be statically determined, a value can be loaded from each page, causing the corresponding mapping to be placed in the TLB. Prefetching can also be used with data-dependent memory accesses if the entire dataset can be mapped by the TLB. The compiler can simply load a byte from each page in the dataset, and place all of the memory accesses within a single restart region.

Prefetching only helps with TLB misses, and does not avoid having to repeat work for other types of exceptions, such as timer interrupts. Also, the compiler must employ

a suitable heuristic to determine when the overhead introduced by adding prefetching instructions outweighs the potential benefit.

2.3.2 Counted Loop Optimization

When dealing with counted loops, which make up the bulk of vectorizable code, another way of avoiding wasteful repeated restarts is to only restart execution from the *earliest uncompleted loop iteration* after handling an exception, instead of restarting at the beginning of a region. To accomplish this, we can adapt a previous scheme from sentinel scheduling [23], which creates idempotent blocks of code to support compiler-controlled speculative execution in a VLIW architecture. This work was extended in [3] to use *recovery blocks* to improve performance. Similarly, we can use recovery blocks with software restart markers. When an exception is handled, the processor will then branch to a block of fixup code which will adjust the counters and pointer values before resuming execution of the loop. We present a modified version of the `memcpy` loop in Figure 7 as an example.

```
      move t0, a0          # Copy the argument registers
      move t1, a1
      move t2, a2
      move lc, 0           # Initialize loop iteration counter
begin restart region—put fixup code address into restart PC
loop: lv v1, t1            # Load the input vector
      sv t0, v1            # Copy the data to the output vector
      addiu t1, 512        # Increment the input base register
      addiu t0, 512        # Increment the output base register
      subu t2, 512         # Decrement the number of bytes remaining
      excepbar_loop        # Non-blocking instruction that increments
                           # a loop iteration counter when all previous
                           # instructions have cleared exception checks
      bnez t2, loop        # Is loop done?
done:
end restart region
fixup_code:
      move t3, lc          # Get value of loop iteration counter
      sll t3, 9            # Each iteration copies 512 bytes
      addu t0, a0, t3      # Set output base register
      addu t1, a1, t3      # Set input base register
      subu t2, a2, t3      # Set loop counter
      j loop               # Restart loop
```

Figure 7: Vectorized memcpy loop with loop iteration counter.

The loop in Figure 7 uses a special barrier instruction—`excepbar_loop`—that will update a loop iteration counter when it is known that no previous instructions will cause an exception. This allows execution to resume from the earliest uncompleted iteration after handling an exception, which helps to reduce the number of re-executed instructions. Note that this instruction does not update the restart PC.

3. COMPILER IMPLEMENTATION

In this section, we describe the compiler analysis necessary to insert software restart markers.

3.1 Infrastructure

Our compiler infrastructure is based on the Trimaran system [1]. The advantage of this infrastructure is that it uses region-based compilation [11], which is a better fit for our scheme than a more traditional function-based approach. We have adapted SUIF [31] to serve as our compiler frontend, by porting the tool flow used in [21]. The dependence analysis in SUIF is used to determine when instructions are vectorizable. This information is passed to the Trimaran backend through the use of instruction annotations. Currently, we only have a backend for scalar code generation and are still in the process of implementing Trimaran passes to generate code for our target vector architecture.

We added support to SUIF for the C `restrict` keyword, which informs the compiler that a variable is not aliased. Without this information, the compiler would have to make restrictive assumptions about whether an array could be overwritten, which would inhibit both vectorization and restart region formation. The vectorizer is only enabled for a C function if all of the pointer arguments are qualified as restricted. Note that aliasing is an orthogonal problem to that of creating software restart markers. Our compiler analysis could also be used within a Fortran compiler, for which aliasing would not be a issue.

Although we are still implementing our vectorizing compiler, it is important to note that the generation of restart regions will not be negatively affected by the use of the vectorizer. In fact, the use of vectorizing transformations should eliminate some dependences that would otherwise impede restart region generation.

3.2 Restart Region Analysis

We added an analysis to the compiler to create a restart region. This encompasses an entire idempotent function by inserting a trap barrier after the function prologue (so that the values saved to the stack are not corrupted) and another one at the end of the function. Additionally, the argument registers are preserved. However, due to the potential performance overhead of using large restart regions, we also implemented the more robust counted loop optimization described in Section 2.3.2.

Figure 8 shows the algorithm used in the compiler for the counted loop optimization. First, all of the loop variables that are live upon entering the loop and are modified by the loop body are found. Since the fixup code must be able to update their values after an exception by using just the value of the loop iteration counter, all of the variables need to be induction variables in order to use the optimization. If this is the case, then the compiler inserts code to preserve the original values of the variables and also to initialize the loop iteration counter before starting the restart region. It also inserts a barrier instruction at the end of the loop so that the counter will be updated at the completion of each iteration. In the fixup code, the value of the loop iteration counter is used to create the correct values for the induction variables. Note that this is frequently a simple procedure, as shown in Figure 7.

The software restart marker analysis is executed before the passes that handle register allocation and optimizations such as common subexpression elimination. This is done so that the restart region code will be optimized. However, creating restart regions before the optimization passes causes a slight complication. Because the fixup code is outside of the program control flow, the instructions that copy the induction variables to backup registers are treated as dead code. We handle this by explicitly marking the backup registers as live throughout the function.

Although this technique cannot be used for every type of

Determine if counted loop optimization can be used:

- Find the set S of all variables $v_0, v_i, \ldots$ in loop L that are live upon loop entry and modified within the loop body.
- For each variable v_i in S, determine if it is an induction variable.
- If all elements of S are induction variables, then optimization can be used.

Code to be inserted before loop:

1. For each variable v_i, copy its value to another variable `restart_`v_i.
2. Initialize loop iteration counter to 0 and place address of fixup code in restart PC.

Code to be inserted at end of loop:

1. Add `excepbar_loop` instruction to end of loop body.

Fixup code to be inserted outside of program control flow:

1. Obtain value of loop iteration counter c.
2. For each variable v_i, apply its induction operation c times to the value in `restart_`v_i to get the current value of v_i.
3. Jump to beginning of loop L.

Figure 8: Compiler steps to generate code with counted loop optimization.

vectorizable loop—all variables restored by fixup code must be induction variables—it is applicable to a wide variety of programs. We have tested our compiler algorithm on several different benchmarks, including code that contains multiple loops, and the compiler generated correct code for all of the examples. For programs with nested loops, we only generate fixup code for the innermost loop, although the algorithm could easily be extended to handle outer loops as well.

4. EVALUATION

In this section, we evaluate the performance effects of using software restart markers when compared to a baseline system that does not support virtual memory.

4.1 Methodology

We evaluate our techniques using the Scale vector-thread processor [20]. Scale flexibly supports both vector processing and multithreaded execution, but in this work we only use its vector capabilities. The default Scale configuration has a single-issue MIPS control processor and a decoupled vector unit. The vector unit has four lanes with clustered functional units, and can support vector lengths of up to 128 elements. Maximum vector length varies from loop to loop, and is dependent on the number of vector registers used by the loop body.

From the perspective of our model, the implementation details of Scale are unimportant, as the vector unit is simply treated as a black box containing temporary state that is discarded at an exception but which can be easily recreated when execution resumes. However, to obtain a realistic performance evaluation, it is important that we use a high-performance vector implementation. Otherwise, any overhead from using software restart markers would be largely hidden due to a slow vector unit. It has been previously shown that Scale provides competitive performance in the embedded domain [20]. Table 1 illustrates the speedup on Scale for our workload, which consists of a variety of programs from the EEMBC 1.1 benchmark suite. Differences between the speedups in Table 1 and the data in [20] are due to recent changes in both the baseline architecture and the vectorized code. These speedup results were obtained using the Scale simulation environment, which includes a cycle-level, execution-driven microarchitectural simulator with detailed models of the vector unit and the DRAM memory system. When evaluating the performance impact of software restart markers below, we replaced the DRAM model with a single-cycle "magic" memory to increase baseline performance and hence emphasize the relative impact of overhead instructions.

We model a standard MIPS R3000-style unified TLB [15] for the control processor. This contains 64 entries, 8 of which are "wired entries" reserved for root-level page table entries and kernel mappings. A separate 128-entry TLB is used for vector memory accesses. We model a system with a software-refilled TLB to provide a worst-case scenario. We stress that many vector implementations would choose a hardware page table walker to give much better performance and remove restrictions on restart region size, but the software-managed TLB configuration allows us to generate a significant number of exceptions to test our software restart mechanism.

Each page has a fixed size of 4KB. We model the 2MB virtually-addressed linear user page table as well as the 2KB physically-addressed root-level page table. We do not model page faults, as our primary interest is in the overhead to allow a machine to take page faults rather than in the speed impact of page faults themselves.

Because we do not yet have a complete vectorizing compiler, all vectorized code was written in assembly for the actual evaluation, and the restart regions were manually created using the techniques described in Section 2. The high quality hand-vectorized code also ensures a conservative estimate of the relative overhead of the software restart scheme. A poorer quality code generator would experience relatively less impact from the software restart scheme. To compile programs, we used a modified version of GNU `gcc` version `egcs-1.0.3a` with the `newlib` standard C library. We used GNU `gas` as our assembler. All code was compiled with the `-O` flag and statically linked to produce an ELF executable. To verify the correctness of our restart region generation, we added an option to the simulator to randomly replay execution from the beginning of a region at various points in the program. For our results, each benchmark was run for the recommended number of iterations, and benchmarks were weighted equally to compute an average.

To delimit restart regions, we created new opcodes to serve as trap barriers. Each of these instructions can only commit once all previous exception checks have cleared. A `restart_on` instruction indicates the beginning of a new restart region. This instruction uses a PC-relative offset to determine the address to be placed in the restart PC—either the subsequent instruction or the beginning of the appropriate fixup code. A `restart_off` instruction indicates that the processor should revert to conventional precise exception semantics. An `excepbar_loop` instruction performs the function described in Section 2.3.2. Since we only use restart regions for vectorized code, the performance overhead of executing these scalar opcodes is very small.

For all vectorized functions, we align the program code to page boundaries. Since all of the code sizes are small and each function fits within a single page, this avoids the possibility of taking a TLB miss in the middle of a restart region because sequential instructions are located on different pages. Of course, if a context switch occurs in the middle of a region and the TLB is flushed, then when the original

Benchmark	EEMBC Category	Data Set	Description	Speedup of Vectorized Code Over Scalar Code
rgbcmy	consumer	-	RGB to CMYK color conversion	15.6
rgbhpg	consumer	-	High pass grey-scale filter	41.6
rgbyiq	consumer	-	RGB to YIQ color conversion	41.3
dither	office	banded	Floyd-Steinberg grayscale dithering	5.3
rotate	office	medium	Binary image 90 degree rotation	14.4
autcor	telecomm	data3	Autocorrelation program	31.6
conven	telecomm	data3	Convolutional encoder	914.1
fft	telecomm	data_3	Fast Fourier transform	15.0
viterbi	telecomm	data_4	Viterbi decoder	8.9

Table 1: Benchmark descriptions and speedups of vectorized code for the default Scale configuration with four lanes.

process resumes at the head of the region, the mapping for the program code will need to be re-fetched.

As discussed in Section 2.3.1, one way to reduce the performance overhead of our scheme in a system with a software-managed TLB is to prefetch the necessary mappings. Since this adds overhead to a program, we only use this approach for benchmarks that are likely to access multiple pages of data. These are the image-processing programs, listed as the first five benchmarks in Table 1.

4.2 Results

The results presented in this section represent a very pessimistic evaluation of the overhead of software restart markers. As mentioned previously, our simulation environment was designed to minimize program runtime—by using a high-performance Scale implementation with magic memory running hand-optimized assembly code—thus overstating the performance impact of our approach. We also begin by considering the worst-case scenario of using a software-managed TLB.

Figure 9 shows the reduction in performance when executing vectorized code with restart regions. This overhead includes three components: the cost of handling TLB misses; the overhead introduced by the extra instructions used to create the restart regions; and the cost of re-executing instructions if an exception occurs in the middle of a restart region. Simulations were conducted using four different machine configurations with one, two, four, and eight lanes to show the impact of scaling baseline performance. However, we discuss the results for the default four-lane configuration unless otherwise noted. Results are presented for three different schemes: the original software restart marker design discussed in Section 2, with an average performance reduction of 15.5%; the same design with prefetching of TLB entries enabled, with an average performance reduction of 0.8%; and the counted loop optimization described in Section 2.3.2, with an average performance reduction of 2.7%. We did not implement the counted loop optimization for the `viterbi` benchmark because in each loop iteration it switches the roles of the input and output buffers, making it difficult to write the fixup code for this situation.

The `rgbcmy` and `rgbyiq` benchmarks have default datasets that require a large number of TLB entries in order to be fully mapped. Thus, there was a significant performance overhead when using the original software restart marker design, as the multiple exceptions within each region caused the re-execution of a great deal of work. Using prefetching in conjunction with the original restart marker design produced the best results. The counted loop optimization did not perform as well as expected, largely due to the fact that there is a significant latency in our system between the TLB check for a memory operation and the time that the operation is retired. As a result, if multiple iterations of a loop are active at the same time, and a TLB miss occurs for the latest iteration, the exception is immediately handled, and all of the earlier work is discarded.

It should be noted that the prefetching scheme produces code that is tied to a particular hardware implementation—if the actual TLB size is less than that used by the compiler in its analysis, then the prefetch code will livelock. Thus, the counted loop optimization is more suitable for producing code that is compatible with a variety of implementations. Additionally, prefetching does not account for the possibility of other types of exceptions occurring within a restart region. We placed each prefetching loop at the beginning of the function, outside of any restart regions. As a result, if a context switch occurs within a restart region and the TLB entries are flushed, when the process resumes, the performance will be similar to that of the original design. It is possible to place a prefetching loop inside of a restart region, but this can introduce additional overhead if the entire region is contained within a loop and is executed multiple times, or if an exception occurs that does not cause a context switch, as the prefetching loop will be executed unnecessarily upon each restart.

As expected, increasing the number of lanes improves performance and thus increases the overhead of software restart markers in most cases. Two notable exceptions occur with the original software restart marker design for the `rgbcmy` and `rgbyiq` benchmarks. In these programs, the overhead of having to re-execute instructions dominates program runtime. However, because the vectorized functions have little scalar overhead, increasing the number of lanes also reduces the time required to re-execute instructions after a TLB miss. Thus, the overhead due to software restart markers remains relatively constant.

Once the first iteration of a benchmark kernel has completed execution, there will be no further TLB misses in future iterations unless the dataset is too large to be fully mapped by the TLB. This is similar to warming up a cache that is large enough to hold the working set of a program. To approximate the performance effect of a TLB with no valid entries, we ran each benchmark for one iteration on the default four-lane configuration and flushed the TLB halfway

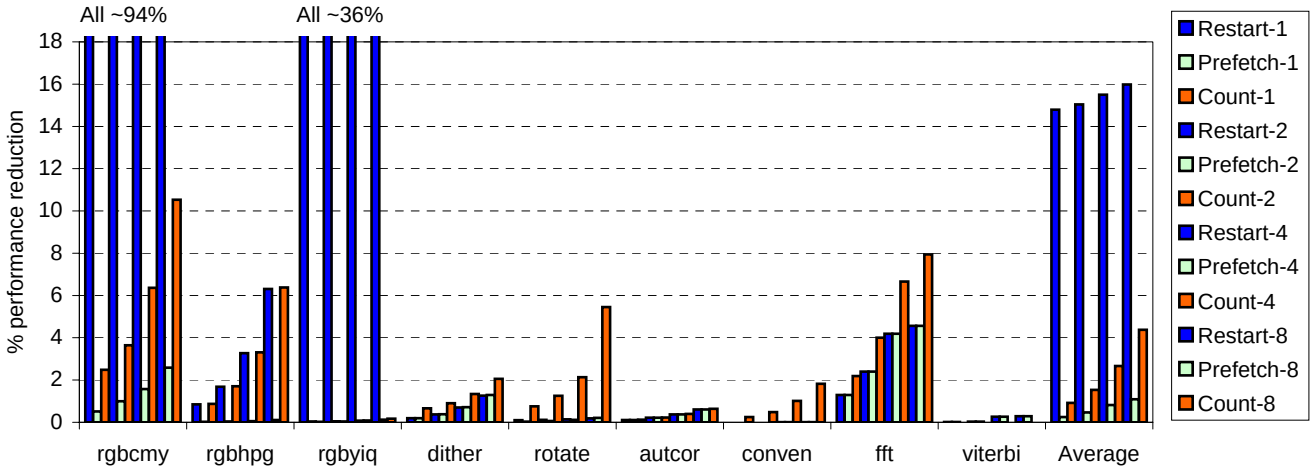

Figure 9: Total performance reduction in vectorized code due to using software restart markers to implement virtual memory in a system with software-managed TLB. Three schemes are shown—the original restart marker design, restart markers with prefetching enabled (which affects the first five benchmarks), and restart markers using the counted loop optimization. The counted loop optimization is not implemented for the viterbi benchmark. Results are presented for machine configurations with 1, 2, 4, and 8 lanes. The values that exceed the graph boundaries are approximately constant across different configurations.

through the restart region that executed for the longest time. This could happen in practice if a context switch occurred at that point. Figure 10 shows the results of flushing the TLB. In general, the original software restart marker design and the prefetching scheme perform poorly across most benchmarks. The counted loop optimization has an overhead of greater than 20% for programs with small datasets—`autcor`, `conven`, and `fft`—as these kernels finish execution quickly and are therefore sensitive to TLB misses. However, for the image-processing programs, which have larger datasets, the loop optimization is more robust. The one exception is `rgbhpg`, which processes an array a column at a time. As a result, there can be accesses to several different pages in a given loop iteration, or spread across a few iterations, leading to a great deal of re-executed work.

Although the performance overhead from flushing the TLB is high in many instances, it should be noted that this will typically occur infrequently. Also, these results reflect the use of a software-managed TLB; in a machine with a hardware TLB refill, the performance degradation would be much lower. However, even in the worst-case scenario of a software-managed TLB, some of the performance may be reclaimed due to the fact that the vector register file contents do not have to be saved and restored in the event of a context switch. Additionally, with the counted loop optimization, even in cases with significant overhead, the vectorized code is still significantly faster than the scalar code.

Figure 11 shows the performance overhead caused by the extra instructions used to create restart regions, without virtual memory enabled. This is roughly equivalent to the performance impact for a design with a hardware TLB refill. The original design of software restart markers has less than a 1% overhead from the use of restart regions. We also present the performance overhead for the counted loop optimization, as this has the advantage over the original design of reducing the amount of repeated work in the event of a context switch. We find that the counted loop optimization incurs nearly three times the overhead of the original design in this case, for two main reasons: First, in benchmarks with multiple loops, each within its own restart region, the control processor has to synchronize with the vector unit after finishing each loop so that the proper fixup code address can be placed into the restart PC. Second, each instance of the `excepbar_loop` instruction occupies an issue slot in the vector unit which could be used by another operation.

5. RELATED WORK

The IBM System/370 vector facility [6] only allowed one vector instruction to be in execution at a time. Although this avoids the problems caused by multiple in-flight vector instructions, it severely limits potential performance. The IBM design also maintains in-use and dirty bits for the vector registers to avoid unnecessary register saves and restores in the event of a context switch. However, this still requires a significant amount of state to be saved and restored when the vector unit is active.

The DEC Vector VAX [7] attempts to reduce context switch overhead by only saving the vector unit state if the new process contains a vector instruction. The DEC design uses microarchitectural swapping to preserve the internal state of the vector unit in the event of an exception, which can hurt performance and energy consumption, as well as complicate the OS interface.

Register renaming is widely used in superscalar processors to not only eliminate false dependences, but also to facilitate in-order instruction commit. This technique has been adapted to vector processors [8, 9, 19] to provide virtual memory support but requires a substantial amount of buffering in the form of additional physical vector registers. Register renaming can provide other benefits if is used to support out-of-order execution with a limited number of architectural vector registers [8], but this performance advan-

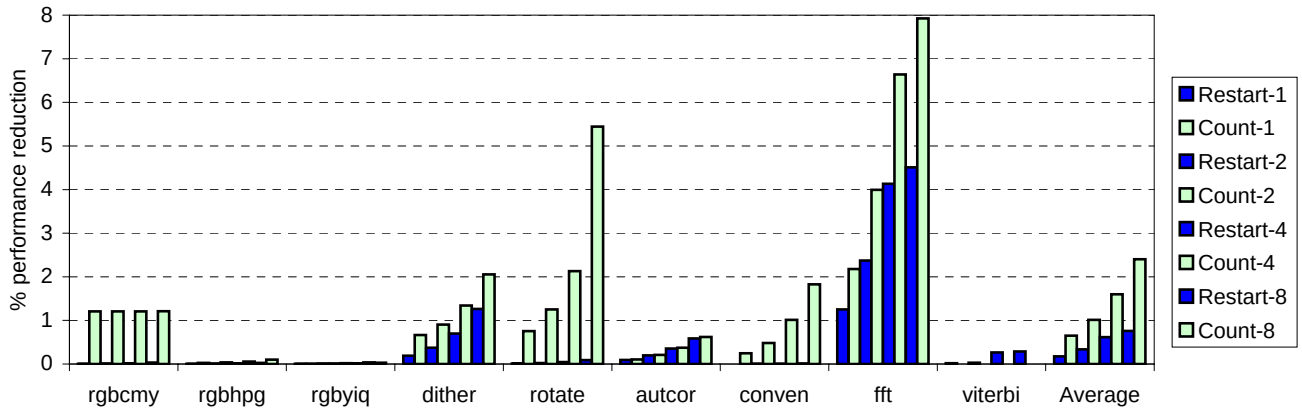

Figure 11: Total performance overhead in vectorized code from using software restart markers in a system with a hardware-managed TLB.

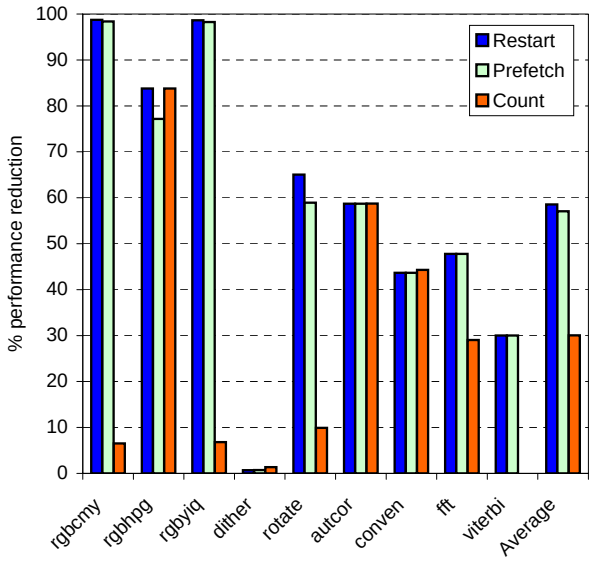

Figure 10: Total performance reduction when using software restart markers to implement virtual memory and the TLB is flushed halfway through a restart region. Results are presented for a configuration with 4 lanes.

tage will be reduced for newer designs with more architectural vector registers.

The CODE vector microarchitecture [19] is a decoupled machine which supports virtual memory by using a clustered vector register file with register renaming. The performance of CODE is evaluated across a workload containing many of the same benchmarks that we use. There is an average performance reduction of 5% when supporting precise exceptions on the default CODE configuration with 8 vector registers per cluster. This overhead is strictly due to the register pressure from implementing register renaming—the cost of handling TLB misses is not considered, although with a hardware refill scheme, the results should be roughly unchanged. Our scheme incurs a much smaller overhead, although a completely accurate comparison is not possible without normalizing the baseline hardware configurations and using the same compilation infrastructure. However, our approach also requires no additional registers or store buffering, making it more scalable to future processor generations that will have larger numbers of simultaneous in-flight operations. It can also be adapted to a variety of vector microarchitectures, and does not require a particular hardware implementation. The disadvantage of our model is that it is less general, as it can only be used with regions of code that are idempotent. However, a wide variety of vectorizable functions fall into this category.

The Alpha floating-point architecture has imprecise floating-point traps which require the user insert trap barrier instructions to delimit safe regions to allow code to resume after the trap [28], but this scheme does not consider other classes of fault or allow irrevocable memory updates.

As mentioned in Section 2, there are some similarities with our idempotent region analysis and the sentinel scheduling analysis used to restart execution after a misspeculation in a software-speculated VLIW architecture [23]. However, we are concerned with providing a simple interface for an OS to restart a process after a virtual memory exception without incurring significant hardware overhead.

The Transmeta Crusoe processor has software-controlled exception barriers at the borders of blocks of x86 code that have been translated into the native VLIW format [17]. The Transmeta scheme has a future file for registers and a speculative store buffer that allow state updates to be revoked if an exception is encountered in a translated block. A commit operation copies state updates into the architectural state. By contrast, our scheme requires no state buffers because it allows irrevocable state changes in the middle of a restart region. Also, we allow for temporary architectural state that is simply discarded at a commit point.

Moudgill and Vassiliadis [24] classify different types of interrupts and discuss when precise interrupts are not required. They also consider the notion of *sparse restart*, in which an interrupt-causing instruction is in the middle of a region that contains freely intermingled completed and uncompleted instructions. They argue that a hardware mech-

anism is necessary in this case to selectively execute only the uncompleted instructions after the interrupt handler is finished. In contrast, we use idempotent blocks of code to avoid having to implement costly hardware mechanisms.

Our approach to handling exceptions is similar to using hardware checkpoints in out-of-order processors [13]. In our scheme software restart markers act as checkpoints that are statically determined by the compiler. This is potentially less flexible than a hardware checkpointing scheme, since restart regions must be idempotent. However, it has the advantage of simplifying the hardware exception mechanisms since the only checkpointed state is the restart PC. Also, while hardware checkpoints require all of the process state to be copied—even if it is not being used—our scheme uses compile-time analysis to only copy what is necessary.

The idea of prefetching TLB entries is presented in [4]. This scheme is designed to decrease the kernel TLB miss penalty by prefetching entries on the inter-process communication path. Our work focuses on reducing restart region overhead by prefetching user TLB entries.

6. CONCLUSION

Support for virtual memory in vector processors has been limited. Where it has been provided, it has often come at the cost of other features such as vector chaining, or with considerable additional hardware complexity. Software restart markers leverage compile-time analysis to coarsen the granularity at which exceptions can be reported, and treat vector registers as temporary state only visible inside a restart region, significantly reducing the hardware cost and performance overhead of supporting virtual memory for vector processors. Our evaluations showed that the average performance reduction due to our scheme when compared to a system without virtual memory was less than 1% when a TLB with hardware refill was used, and was still less than 3% even when a software-managed TLB was used.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This work was partly funded by NSF CAREER award CCR-0093354, DARPA PAC/C award F30602-00-2-0562, and the Cambridge-MIT Institute.

8. REFERENCES

[1] Trimaran homepage. http://www.trimaran.org.

[2] K. Asanović. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, May 1998.

[3] D. I. August et al. Sentinel scheduling with recovery blocks. Technical Report CRHC-95-05, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, January 1995.

[4] K. Bala et al. Software prefetching and caching for translation lookaside buffers. In *OSDI-1*, November 1994.

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

[6] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25(1), 1986.

[7] DEC. *Exception reporting mechanism for a vector processor*. U.S. Patent 5,043,867, August 1991.

[8] R. Espasa et al. Out-of-order vector architectures. In *MICRO-30*, December 1997.

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

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

[11] R. E. Hank et al. Region-based compilation: an introduction and motivation. In *MICRO-28*, December 1995.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, chapter Appendix G. Morgan Kaufman Publishers, third edition, 2003.

[13] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *ISCA-14*, June 1987.

[14] B. L. Jacob and T. N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *ASPLOS-8*, October 1998.

[15] G. Kane. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1989.

[16] K. Kitagawa et al. A hardware overview of SX-6 and SX-7 supercomputer. *NEC Research & Development Journal*, 44(1), January 2003.

[17] A. Klaiber. The technology behind Crusoe processors. White paper, Transmeta Corporation, January 2000.

[18] C. Kozyrakis. *Scalable vector media-processors for embedded systems*. PhD thesis, University of California at Berkeley, May 2002.

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

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

[21] S. Larsen et al. Exploiting vector parallelism in software pipelined loops. In *MICRO-38*, November 2005.

[22] C. G. Lee and M. G. Stoodley. Simple vector microprocessors for multimedia applications. In *MICRO-31*, 1998.

[23] S. A. Mahlke et al. Sentinel scheduling for VLIW and superscalar processors. In *ASPLOS-5*, October 1992.

[24] M. Moudgill and S. Vassiliadis. Precise interrupts. *IEEE Micro*, 16(1), February 1996.

[25] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *CACM*, 29(12), 1986.

[26] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a vector unit to a superscalar processor. In *ICS-13*, June 1999.

[27] R. M. Russell. The Cray-1 computer system. *CACM*, 21(1), 1978.

[28] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press, October 1992.

[29] R. Uhlig et al. Design tradeoffs for software-managed TLBs. *ACM Transactions on Computer Systems*, 12(3), August 1994.

[30] T. Utsumi et al. Architecture of the VPP500 parallel supercomputer. In *ICS-8*, November 1994.

[31] R. P. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), December 1994.