ABSTRACT

Title of Thesis: In-line Interrupt Handling and Lockup Free TLBs

Degree Candidate: Aamer Jaleel

Degree and Year: Master of Science, 2002

Thesis directed by: Dr. Bruce L. Jacob

Department of Electrical and Computer Engineering

The effects of the general-purpose precise interrupt mechanisms in use for the past few decades have received very little attention. When modern out-of-order processors handle interrupts precisely, they typically begin by flushing the pipeline to make the CPU available to execute handler instructions. In doing so, the CPU ends up flushing many instructions that have been brought in to the reorder buffer. In particular, these instructions may have reached a very deep stage in the pipeline—representing significant work that is wasted. In addition, an overhead of several cycles and wastage of energy (per

exception detected) can be expected in re-fetching and re-executing the instructions flushed. This thesis concentrates on improving the performance of precisely handling software managed translation lookaside buffer (TLB) interrupts, one of the most frequently occurring interrupts. The thesis presents a novel method of in-lining the interrupt handler within the reorder buffer. Since the first level interrupt-handlers of TLBs are usually small, they could potentially fit in the reorder buffer along with the user-level code already there. In doing so, the instructions that would otherwise be flushed from the pipe need not be re-fetched and re-executed. Additionally, it allows for instructions independent of the exceptional instruction to continue to execute in parallel with the handler code. By in-lining the TLB interrupt handler this provides *lock-up free* TLBs.

This thesis proposes the *prepend* and *append* schemes of in-lining the interrupt handler into the available reorder buffer space. The two schemes are implemented on a processor with a 4-way out-of-order core similar to the Alpha 21264. We compare the overhead and performance impact of handling TLB interrupts by the traditional scheme, the append in-lined scheme, and the prepend in-lined scheme. For small, medium,

and large memory footprints, the overhead is quantified by comparing the number and pipeline state of instructions flushed, the energy savings, and the performance improvements. We find that, lock-up free TLBs reduce the overhead of re-fetching and re-executing the instructions by 30-95%, reduce the energy consumption and execution time by 5-25%, and also reduce the energy wasted by 30-90%.

IN-LINE INTERRUPT HANDLING AND LOCK-UP FREE TLBs

by

Aamer Jaleel

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2002

Advisory Committee:
Professor Bruce L. Jacob, Chairman/Advisor
Professor Donald Yeung
Professor Manoj Franklin

© Copyright by
Aamer Jaleel
2002

ACKNOWLEDGEMENTS

I am extremely grateful to my advisor, Dr. Bruce L. Jacob, for giving me the opportunity to work with him for the past four years. He has provided me with good advice and perspective over these years in the classroom, countless hours in his office, and many informal meetings in the hallways and “other” places.

I would also like to thank my parents for their many sacrifices, love, and unconditional support all along. I would also like to thank my brothers and sisters for their patience and support.

I would also like to thank the members of my research community, especially Brinda, Vinodh, Dave, Paul, Jyoti, Abdel-Hameed, Aneesh, and Deepak for their support, advice, and suggestions over the past two years. Special thanks go to Brinda for always encouraging me and assisting me in solving the different problems I have faced. Another special thanks go to Jyoti for her advice, assistance, and support in all aspects of this thesis.

Finally, thanks to Allah (swt) for blessing me with a number of opportunities, giving me the strength to reach this point, and for providing me with great family and friends.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

THE INTERRUPT PROBLEM

1.1 The Problem

Precise interrupts in modern microprocessors have become both frequent and expensive and are rapidly becoming even more so. One reason for their rising frequency is that the general interrupt mechanism, originally designed to handle the occasional exceptional condition, is now used increasingly often to support normal (or, at least, relatively frequent) processing events such as support for garbage collection, software managed distributed virtual memory, and profiling [34].

Besides their increasing frequency, interrupts are also becoming highly expensive; this is because of their implementation. Modern out-of-order cores typically handle precise interrupts in the same way as a register-file update: i.e., at commit time. When an exception is detected, the fact is noted in the processors reorder buffer (ROB) entry. The exception is not usually handled immediately; rather, the processor waits until the instruction in question is about to commit before handling the exception—doing so ensures that exceptions are handled in program order and not speculatively. Once the processor determines an exception isn't speculative, it proceeds through the following phases:

1. The pipeline and ROB are flushed; exceptional PC is saved, and the PC is set to the appropriate handler.
2. The exception is handled with privileges enabled.
3. Once the interrupt handler has finished execution, the exceptional PC is restored, and the user program continues execution.

Increasing trends in performance improvement conspire against this method of exception handling. First, the cost of exception handling is increasing relative to the performance of microprocessors. The increasing pipeline depths and growing reorder buffer sizes highlight the enormous overhead of handling an exception mainly due to the fact that the pipeline is flushed. With an 80-entry ROB, like the Alpha 21264, this can account for as many as a whole instruction window being flushed at the time of an exception. Additionally, an overhead of several cycles can be expected in re-fetching and re-executing the instructions that were flushed. Even more, there is additional performance loss because no user instructions execute while the interrupt handler is being executed. The interrupt handler was invoked because an instruction that was being retired had an exception, but other instructions independent of the exceptional instruction could have continued execution. Along with these sources of performance loss, there is a significant amount of energy wastage in terms of re-fetching and re-

executing the instructions flushed. Given that we have broken the triple digit Wattage ratings for modern microprocessors, it is imperative that we improve the traditional method of handling interrupts.

1.2 A Novel Solution

If we look at the different sources of performance loss with the traditional scheme of handling interrupts (user code stalls during the execution of handler, many instructions are fetched and executed twice), we see that they are due to the fact that the reorder buffer is flushed at the time the exception is detected. If we can avoid flushing the reorder buffer, but still allow for the interrupt handler to be fetched and executed, we could essentially eliminate these sources of performance loss. This has been pointed out before, but the suggested solutions (refer to the related work section of this thesis) have typically been to save the entire internal state of the pipeline and restore it upon completion of the interrupt handler.

To begin with, why is the reorder buffer and pipeline flushed? Is it to ensure privileges and not allow privileged operating system instructions (interrupt handler) to co-exist with user level instructions already present in the pipeline? A suggested solution to this problem is to add a privileged bit to each entry of the reorder buffer, thus instructions fetched in privileged

mode have their privileged bits set, and those fetched in user mode don't have the privileged bit set [9].

So if a solution to the privilege problem exists, then why do modern microprocessors still flush the pipeline? *Space*, processors flush the pipeline and reorder buffer to make room for the interrupt handler to be fetched, executed, and retired. Since user instructions are held up at commit because of the exceptional instruction, no other instruction would be able to commit. Additionally, if the interrupt handler requires more space than is available in the reorder buffer, the machine would encounter a deadlock situation. However, if their exists enough space for the interrupt handler to fit into the reorder buffer, and with a few modifications to existing microprocessors, one can avoid flushing the pipeline.

Our solution to the interrupt problem uses existing out-of-order hardware to handle interrupts both precisely and inexpensively. If at the point the processor detects an interrupt, it checks to see if there are enough unused reorder buffer slots. If so, the processor will in-line the interrupt handler within the reorder buffer and not flush the pipeline. However, if there aren't enough reorder buffer slots available, the processor will handle the interrupt by the traditional method—flush the pipeline.

Though such a mechanism is generally applicable to all types of software managed interrupts that have relatively short interrupt handlers, for the pur-

pose of this thesis, we will concentrate on one type of interrupt handler—that used by a software managed TLB to invoke the first-level TLB-miss handler. We do this for several reasons:

1. TLB-miss handlers are invoked *very* frequently (once per 100-1000 user instructions)
2. The first-level TLB-miss handlers tend to be short (on the order of ten instructions) [12, 16]
3. These handlers also tend to have deterministic length (i.e., they tend to be straight-line code—no branches)

In-lining the TLB interrupt handler allows for instructions independent of the faulting instruction to continue executing. This essentially provides us with *lock-up free* TLBs.

1.3 Results

We evaluated two separate lock-up free mechanisms (*append* and *prepend* schemes) on a processor model of an out-of-order core with specs similar to the Alpha 21264 (4-way out-of-order, 150 physical registers, up to 80 instructions in flight, etc.). No modifications are required of the instruction-set; this could be implemented on existing systems transparently—i.e., without having to rewrite any of the operating system.

We model only a lock-up-free data-TLB facility, and have left the lock-up-free instruction-TLB facility as part of our future work. Lockup free TLBs reduce the number of instructions flushed by 30-95%, thus significantly reducing the amount of time and energy wasted. When applications generate TLB misses frequently, this reduction in overhead amounts to a substantial performance savings. We find that lock-up-free TLBs enable a system to reach the performance of a traditional fully associative TLB with a lock-up-free TLB of roughly one-fourth the size. We additionally see that the lock-up-free TLB provides performance and energy savings of 5-25%, and reduce the amount of energy wasted by 30-90%.

Chapter 2

INTERRUPT HANDLING AND REORDER BUFFERS

Under ideal conditions, a processor would start executing a program and continue executing it to completion without stopping or encountering any problems. However, in the real world, processors can encounter several different unpredictable situations that hinder the normal flow of execution. For example, during the course of execution, if the processor detects an illegal opcode, a divide by zero, or an overflow occurs, then the processor should know exactly what to do. It could very well be that the processor could completely halt the program and work on another. However, there are some exceptions that should not be suicidal; instead the processor must overcome the exception. An example of such an exception is the TLB miss exception, where the processor performs some behind-the-scenes work (fill the TLB with a virtual to physical address mapping) on behalf of the running program. Additionally, among the several different devices that a processor interacts with, if one of the devices signals the processor, then the processor will have attend to the device in one way or another. The signals and situations that arise and prevent a CPU from continuing to execute a program are called exceptional events, also known as *interrupts*.

In this chapter, we will discuss the different types of interrupts that can occur during the course of execution. We will discuss how these interrupts are handled in general, and how they are handled in the case of modern high performance microprocessors.

2.1 Interrupts

Interrupts can be classified into the internal and external types. Internal interrupts, “sometimes referred to as traps, result from exceptional conditions detected during the fetching and executing of instructions” [20]. These exceptions can be due to software errors such as illegal opcodes, divide by zero, or it could very well be that the instruction itself is an interrupt instruction.

External interrupts however are not caused by specific instructions, instead are caused by sources that are outside the process being executed, sometimes these sources are completely unrelated to the process [20]. Examples of such interrupts are I/O interrupts, timer interrupts, keyboard interrupts, etc.

2.1.1 Handling Internal Exceptional Situations

Since internal exceptions are caused by the program itself, they must be satisfied to allow for the successful execution of the program. When the

exception detecting hardware detects an exception, an interrupt is generated. The operating system then responds by executing specialized instructions to satisfy the exception. These specialized instructions, known as an interrupt handler, are stored in a read only place in memory. When the operating system detects an interrupt, it consults its interrupt vector table (IVT) to determine where in memory the interrupt handler is located. The operating system then saves state, fetches and executes the interrupt handler with privileges enabled.

Software managed interrupts are blocking, i.e. they require the processor to stop fetching program instructions, fetch and execute handler instructions, and then continue fetching and executing the user program. An alternative approach is to have the exception handled by dedicated hardware, and allow the for the parallel execution of the user program [16]. Exceptions that are hardware managed benefit solely in the fact that they are non-blocking, i.e. the user program can continue executing while the exception is being handled.

2.1.2 Issues with Interrupt Handling

Consider a basic processor model where a new instruction is fetched and executed only when the previous instruction has finished execution. With this architectural model, as soon as the processor executes the instruction,

it knows whether or not it has caused an exception. If the instruction causes an exception, the processor stops fetching program instruction, handles the exception by executing the appropriate handler. Once the processor has finished executing the handler, it can then continue fetching and executing program instructions.

Handling an interrupt in the case of a pipelined or superscalar model is not as easy as a basic processor model. This is because there is not just one instruction, but many instructions in the process of execution. Since the processor temporarily stops fetching program instructions and fetches the interrupt handler, it is of extreme importance to save processor state restore it once execution of the handler is finished.

2.1.2.1 Saving the Machine State

Saving the machine state usually involves saving the state of the register file, the address of the next user instruction to fetch (i.e. the PC). Saving the state of the register file is necessary if the interrupt handler uses the same registers as the user program. If the register file isn't saved, the handler could possibly overwrite existing register values, and when the user program is restored, it will have the wrong data to work with. Modern microprocessors overcome this overhead by using a dedicated set of registers for the interrupt handler. Thus any change to the register file while exe-

cuting the interrupt handler is only to the handler registers and not the user registers [13].

2.1.2.2 When Should an Interrupt Be Handled?

Since modern superscalar processors improve performance and throughput by executing instructions out of order, exception handling becomes non-trivial. If the exception is handled immediately (i.e. *imprecisely*), the processor will not only have to remember the PC and the state of the register file, but will also have to remember the entire state of the pipeline. This is because the incoming interrupt handler will step through and overwrite the different pipeline registers. Saving the pipeline state becomes highly expensive especially with the growing trends in the number of pipeline stages. Additionally, the overhead can be an utter waste if the exception was handled for an instruction that was down a speculative path, meaning an instruction past a branch that hasn't been resolved yet. If the branch happens to mispredict, then the interrupt should never have been handled.

Thus, to avoid the overheads of handling an interrupt imprecisely, an interrupt should be handled only if its absolutely sure that the exception wasn't down a speculative path. One is absolutely sure that an instruction is not speculative if the instruction in question is being committed to the reg-

ister file. Thus, to avoid speculative exception handling, exceptions should be handled at instruction commit time, i.e. *precisely.*

2.2 Precise Interrupts

An interrupt is precise if the state saved is consistent with the basic processor model, i.e. the sequential model [15, 20]. To be more specific, the interrupt is not handled speculatively but in program order and follows the following conditions:

- All instructions preceding the exception causing instruction have finished execution and have committed state.
- All instructions after the exception causing instruction have NOT committed state. They could have finished execution.
- The exception causing instruction may or may not have finished execution dependant on the exception class.

If the stated conditions are met, then the interrupt is said to be precise. However, if the interrupt doesn't satisfy these conditions, then the interrupt is imprecise.

2.2.1 Implementing Precise Interrupts

To ensure that an interrupt is not handled speculatively, before retiring an instruction, the processor checks to see if the instruction in question has caused an exception or not. If the processor sees that the instruction caused an exception, it:

- Saves the state, and the exceptional PC. The exceptional PC depends on the exception class. Certain interrupts, such as TLB interrupts, require the exception causing instruction to re-execute and thus cause the hardware to set exceptional PC to be the PC of the exception causing instruction. Other interrupts, such as I/O interrupts, set the exception PC to be the PC of the next instruction after the exception causing instruction.
- Set the PC to the address of the interrupt handler, enable privileges, and fetch and execute the handler.
- Once it has finished executing the handler, the processor will restore the exceptional PC and continue executing program instructions.

To allow for precise interrupts, processors can force in order execution of instructions. This would be ideal if the frequency of interrupts is extremely high. However, if exceptions don't occur very often, there is a significant performance loss with in order execution.

If out of order execution is allowed, instructions leave the pipeline in a different order than there were brought in. To maintain precise interrupts, the processor must somehow keep track of the original order of instructions before sending them to functional units. Special hardware data structures and techniques have been mentioned [20], the most commonly used is the *reorder buffer.*

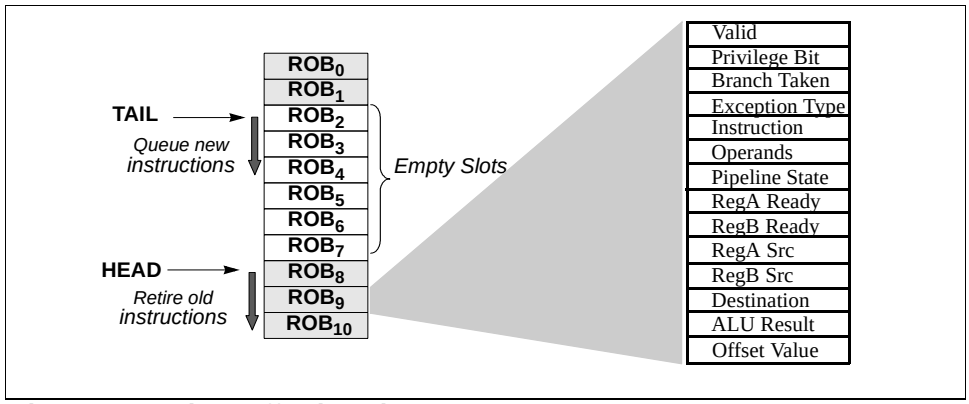

Fig. 2.1. Reorder Buffer (ROB). A hardware data structure, essentially a circular queue, used to maintain the program order of instructions. New instructions are fetched in at the tail, and are retired from the head. Instructions are issued from the ROB to functional units. Each ROB entry holds the entire state of an instruction as shown. Since the ROB maintains the actual program order of instructions, it serves as good purpose for precise interrupts

2.2.2 Reorder Buffer (ROB)

A reorder buffer (ROB) is a circular buffer, basically a queue, of entries with a head and tail pointer (see Figure 2.1). The head and tail pointer both point to entries within the reorder buffer. When the processor fetches new instructions, they are queued at the tail of the reorder buffer. During the retire phase, the processor checks if instructions at the head are ready to retire, if so, it commits their state and moves the head pointer down. Retiring from the head of the reorder buffer allows for in order commit. The entries between the tail and head are empty slots.

Each reorder buffer entry keeps track of the entire state of an instruction: validity, pipeline state, destination register, result, operands, operand sources, exception flags, etc. Instructions are fetched into the reorder

buffer, the appropriate reorder buffer entries are set during the decode phase. When the dependencies of instructions are resolved, they are sent to execution units from the reorder buffer, the results are then written back into the reorder buffer entry. If an instruction causes an exception, the fact is noted in the reorder buffer entry. During the retire stage, the processor checks to see if there are instructions at the head waiting to retire. If the instruction at the head is ready to retire and the exception flag is not set, the processor commits the state of the reorder buffer entry to the global state. However, if the processor detects that the instruction caused an exception, the processor saves state, flushes the reorder buffer and executes the interrupt handler. After the handler has finished execution, the processor then resumes fetching user instructions. The reorder buffer thus allows in order committing of user instructions and at the same time allows for precise interrupt handling.

Chapter 3

INTERRUPTS IN MODERN MICROPROCESSORS

The use of interrupts in modern microprocessors has been increasing drastically. This is because they are being used to support normal or relatively frequent activities such as garbage collection, profiling, and virtual memory. This chapter discusses how the increase in the use of interrupts impacts performance of modern microprocessors and suggests a novel solution to help alleviate the performance loss due to frequent interrupts.

3.1 Traditional Scheme of Interrupt Handling

With the current trends in processor and operating systems design, the cost of handling exceptions precisely is becoming extremely expensive; this is because of their implementation. Most high performance processors typically handle precise interrupts at commit time [15, 17, 21, 25]. When an exception is detected, a flag in the instruction’s reorder buffer entry is set indicating the exceptional status. Delaying the handling of the exception ensures that the instruction didn't execute along a speculative path. During the retire phase, before committing an instructions state, the exception flag of the instruction is checked. If the instruction caused an excep-

tion and software support is needed, the hardware handles the interrupt in the following way:

- The ROB is flushed; the exceptional PC is saved; the PC is redirected to the appropriate handler.
- Handler code is executed, typically with privileges enabled.
- Once a *return from interrupt* instruction is executed, the exceptional PC is restored, and the program resumes execution.

3.1.1 Problems with the Traditional Scheme of Handling Interrupts

With this model of handling interrupts, there are two primary sources of application-level performance loss: (1) while the exception is being handled, there is no user code in the pipe, and thus no user code executes—the application stalls for the duration of the handler; (2) after the handler returns control to the application, all of the flushed instructions are re-fetched and re-executed, duplicating work that has already been done. Since most contemporary processors have deep pipelines and wide issue widths, there may be many cycles between the point that the exception is detected and the moment that the exception is acted upon. Thus, as the time to detect an exception increases, so does the number of instructions that will be re-fetched and re-executed [17]. Clearly, the overhead of taking an

interrupt in a modern processor core scales with the size of the reorder buffer, pipeline depth, issue-width, and each of these is on a growing trend.

3.2 In-line Interrupt Handling - A Novel Solution

If we look at the two sources of performance loss (user code stalls during handler; many user instructions are re-fetched and re-executed), we see that they are both due to the fact that the ROB is flushed at the time the PC is redirected to the interrupt handler. If we could avoid flushing the pipeline, we could eliminate both sources of performance loss. This has been pointed out before, but the suggested solutions have typically been to save the internal state of the entire pipeline and restore it upon completion of the handler. For example, this is done in the Cyber 200 for virtual-memory interrupts, and Moudgill & Vassiliadis briefly discuss its overhead and portability problems [15]. Such a mechanism would be extremely expensive in modern out-of-order cores, however; Walker & Cragon briefly discuss an extended shadow registers implementation that holds the state of every register, both architected and internal, including pipeline registers, etc. and note that no ILP machine currently attempts this [25]. Zilles, et al. discuss a multi-threaded approach, where at the time an exception is detected, the processor spawns a new thread to fetch and execute the inter-

rupt handler [33]. The scheme works effectively, but requires the processor architecture to allow multiple threads executing in parallel.

We are interested instead in using existing out-of-order hardware to handle interrupts both precisely and inexpensively. Looking at existing implementations, we begin by questioning why the pipeline is flushed at all—at first glance, it might be to ensure proper execution with regard to privileges. However, Henry has discussed an elegant method to allow privileged and non-privileged instructions to co-exist in a pipeline [9]; with a single bit per ROB entry indicating the privilege level of the instruction, user instructions could execute in parallel with the handler instructions.

If privilege level is not a problem, what requires the pipe flush? Only space: user instructions in the ROB cannot commit, as they are held up by the exceptional instruction at the head. Therefore, if the handler requires more ROB entries than are free, the machine would deadlock were the processor core to simply redirect the PC without flushing the pipe. However, in those cases where the entire handler could fit in the ROB in addition to the user instructions already there, the processor core could avoid flushing the ROB and at the same time avoid such deadlock problems.

Our solution to the interrupt problem, then, is simple: if at the time of redirecting the PC to the interrupt handler there are enough unused slots in

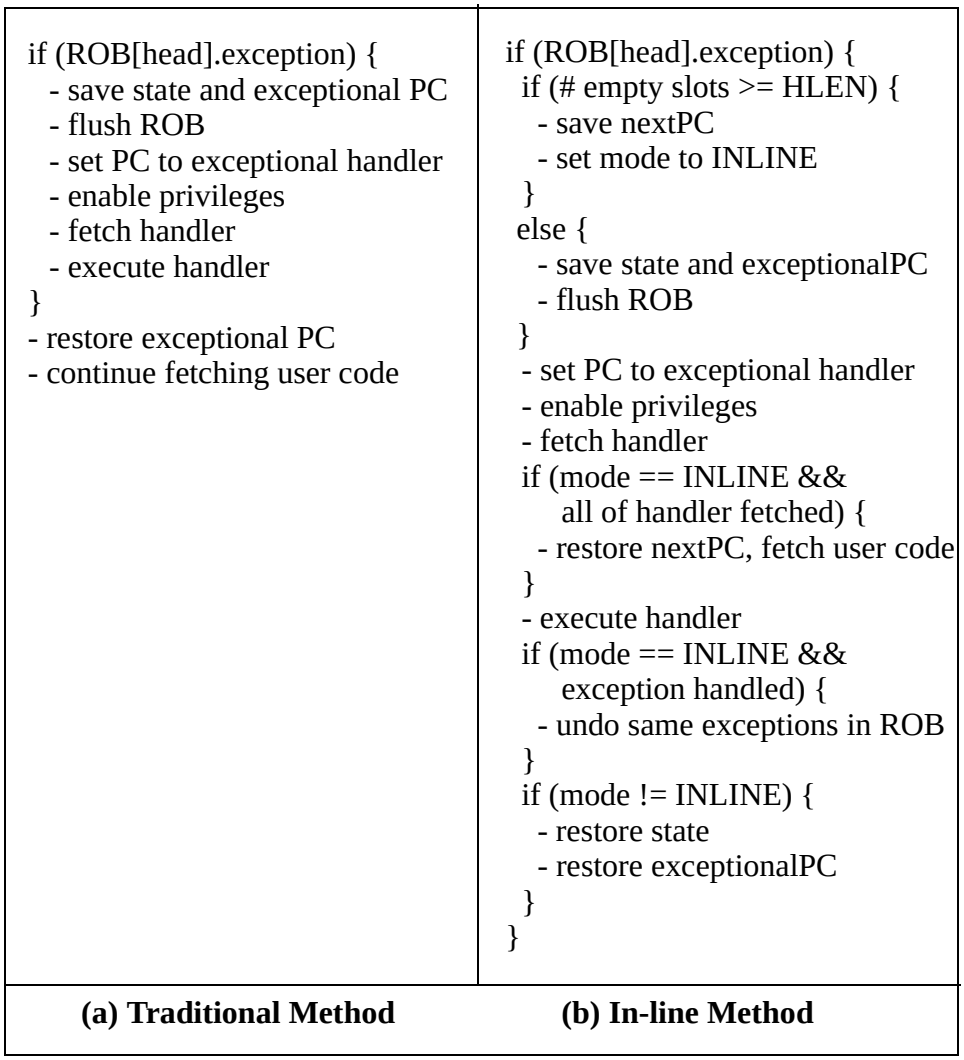

Fig. 3.1. Interrupt Handling (Traditional vs. In-line). The traditional method of handling an interrupt, and the proposed scheme of in-lining an interrupt handler.

the ROB, we in-line the interrupt handler code without flushing the pipeline. However, if there aren't sufficient empty ROB slots, we handle the interrupt as normal. If the architecture uses reservation stations in addition to a ROB [7, 26] (an implementation choice that reduces the number of

result-bus drops), we also have to ensure enough reservation stations for the handler, otherwise handle interrupts as normal.

3.3 In-lining Translation Look Aside Buffer (TLB) Interrupts

Though the in-lined mechanism of handling interrupts is applicable to all types of transparent interrupts (with relatively short handlers), we focus on only one interrupt in this paper—that used by a software-managed translation look aside buffer (TLB) to invoke the first-level TLB-miss handler.

3.3.1 What are TLB Interrupts?

A translation lookaside buffer (TLB) is a hardware data structure that aids in the quick translation of a virtual address to a physical address. Whenever a running application generates an address, it is a virtual address. To determine the corresponding location in memory, the operating system needs to generate the physical address. The physical address is computed by a walk of the page table. Rather than walking the page table on each address generated, to speed up address translation the TLB holds within it recent address translations. If the translation is not present within the TLB, the exception can be handled in one of two ways: either by software or by hardware.

A software managed TLB generates an interrupt. The processor, stops fetching program code, vectors to the TLB miss handler, executes the handler, and then returns back to fetching program code. The TLB miss handler has special operating system level instructions that walk the page table and fill the appropriate translation within the TLB.

A hardware managed TLB on the other hand does not generate an interrupt. Rather than having software walk the page table and fill the TLB, dedicated hardware takes care of doing the page table walk and refilling the TLB. Since no software support is needed, a hardware managed TLB is significantly faster than a software managed TLB. This is because of a few reasons:

- The dedicated hardware to fill the TLB does not miss in the instruction cache, thus it is faster to handle the exception.
- The overhead of flushing the pipeline is avoided.
- In the case of a software managed TLB, no user code executes while processing the TLB miss. With a hardware managed TLB, user code executes while the hardware manages the TLB miss.

3.3.2 Software Managed TLB vs. Hardware Managed TLB

If hardware managed TLBs are significantly faster than software managed TLBs [11, 16], then why not implement all TLB management in hardware? Most modern high-performance architectures use software-managed

TLBs (e.g. MIPS, Alpha, SPARC, PA-RISC), not hardware-managed TLBs (e.g. IA-32, PowerPC), largely because they save hardware and there is increased flexibility with the software-managed design [12]. With a hardware managed TLB, the page table is fixed, and any future changes requires non-trivial porting of the operating system code.

3.3.3 Why In-line TLB Interrupts?

Anderson, et al. [1] show TLB miss handlers to be among the most commonly executed OS primitives; Huck and Hays [10] show that TLB miss handling can account for more than 40% of total run time; and Rosenblum, et al. [18] show that TLB miss handling can account for more than 80% of the kernel's computation time. Recent studies show that TLB-related precise interrupts occur once every 100-1000 user instructions on all ranges of code, from SPEC to databases and engineering workloads [5, 18].

Besides their frequent nature, an additional reason for in-lining of TLB interrupt handlers is because the handler lengths tend to be short (on the order of ten instructions) [16, 12] and the handlers also tend to have deterministic length (i.e., they tend to be straight-line code without any branches)

In-line interrupt handling for software-managed TLBs can ultimately achieve the same performance as hardware-managed TLBs. Note that hard-

ware-managed TLBs have been non-blocking for some time: e.g., a TLB-miss in the Pentium-III pipeline does not stall the pipeline-only the exceptional instruction and its dependents stall [24]. Our proposed scheme emulates the same behavior when there is sufficient space in the ROB, hence providing *lock-up free* TLBs. The scheme thus enables software-managed TLBs to reach the same performance as non-blocking hardware-managed TLBs without sacrificing flexibility [11].

Chapter 4

IN-LINE INTERRUPT HANDLING

4.1 In-line Interrupt Handling

We present two methods of in-lining the interrupt handler within the reorder buffer. Both of our schemes queue new instructions at the tail, and retire old instructions from the head [20]. If there is enough room between the head and the tail for the interrupt handler to fit, we essentially in-line the interrupt by either inserting the handler before the existing user-instructions or after the existing user-instructions. Inserting the handler instructions after the user-instructions, the *append* scheme, is similar to the way that a branch instruction is handled: the PC is redirected when a branch is predicted taken, similarly in this scheme, the PC is redirected when an exception is encountered. Inserting the handler instructions before the user-instructions, the *prepend* scheme, uses the properties of the head and tail pointers and inserts the handler instructions before the user-instructions. The two schemes differ in their implementations, the first scheme being easier to build into existing hardware. To represent our schemes in the following diagrams, we are assuming a 16-entry reorder buffer, a four-instruction interrupt handler, and the ability to fetch, enqueue, and retire two instructions at a time. To simplify the discussion, we assume all instruction state is held in the ROB entry, as opposed to being spread out across ROB

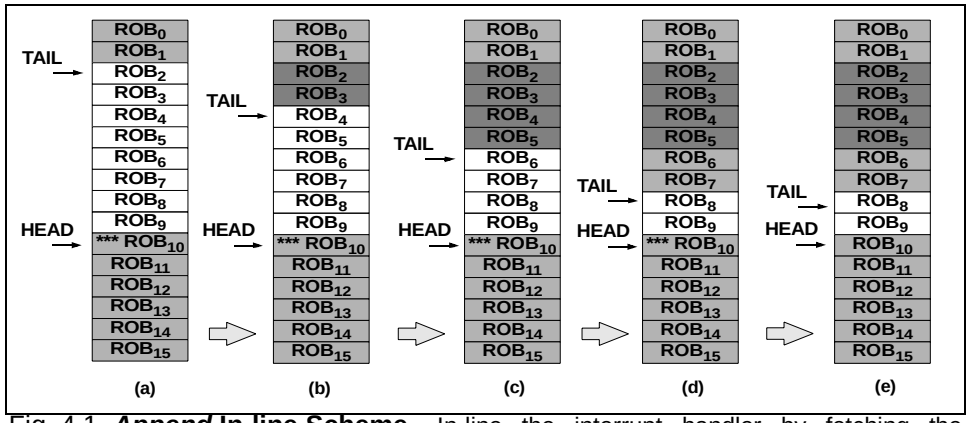

Fig. 4.1. ***Append* In-line Scheme.** In-line the interrupt handler by fetching the instructions at the tail of the reorder buffer. The figure shows the in-lining of a 4-instruction handler, assuming that the hardware fetches and enqueues two instructions at a time. The hardware stops fetching user-level instructions (light grey) and starts fetching handler instructions (dark grey) once the exceptional instruction, identified by asterisks, reaches the head of the queue. When the processor finishes fetching the handler instructions, it resumes fetching user instructions When the handler instruction handles the exception, the processor can reset the flag of the excepted instruction and it can retry the operation.

and reservation-station entries. A detailed description of the two in-lining schemes follow.

4.1.1 Append In-Line Mode

Figure 4.1 illustrates the append scheme of in-lining the interrupt handler. In the first state [state (a)], the exceptional instruction has reached the head of the reorder buffer and is the next instruction to commit. Because it has caused an exception at some point during its execution, it is flagged as exceptional (indicated by asterisks). The hardware responds by checking to see if the handler would fit into the available space-in this case, there are eight empty slots in the ROB. Assuming the handler is four instructions long, it would fit in the available space. The hardware turns off user-instruction fetch, sets the processor mode to INLINE, and begins fetching

the first two handler instructions. These have been enqueued into the ROB at the tail pointer as usual, shown in state (b). In state (c) the last of the handler instructions have been enqueued, the hardware then resumes fetching of user code as shown in state (d). Eventually when the last handler instruction has finished execution and has handled the exception, the processor can reset the flag of the excepted instruction and retry the operation, shown in state (e).

Note that though the handler instructions have been fetched and enqueued after the exceptional instruction at the head of the ROB, in order to avoid a deadlock situation (instructions are held up at commit due to the exceptional instruction at the head of the reorder buffer), the handler must be allowed to update the state of the exceptional instruction—for example in the case of a TLB miss, the *TLB write* instruction should be able to update the TLB without having to commit. This can be achieved by allowing the TLB to be updated when the *TLB write* instruction reaches the commit stage (when they are pulled in from the memory pipeline stage into the commit pipeline stage), and not when it is committing. Though this may seem to imply out-of-order instruction commit, this does not represent an inconsistency, as the state modified by such handler instructions is typically transparent to the application—for example, the TLB contents are merely a hint for better address translation performance.

The append scheme of in-lining however has certain drawbacks. First of all, since the handler is brought at the tail of the reorder buffer, the handler

instructions have the lowest priority in terms of scheduling. If no instruction before the handler instructions can be issued, then the handler will be able to execute, otherwise the handler will have to wait until all previous independent instructions have finished execution. Thus, the more the number of independent instructions before the handler instructions, the longer the wait. Besides this drawback, with the append scheme, the interrupt handler occupies precious space within the reorder buffer. Since the interrupt handler is only required to perform behind the scenes tasks for the program, it would seem befitting for the interrupt handler to "disappear" after it has done it's job. The reason being, (1) more user instructions can be fetched and executed or (2) any exception detected later may not have enough space to in-line, and possibly could've used the in-lined mechanism had the handler instructions not occupied the reorder buffer slots. To avoid these possible drawbacks, we propose the second scheme of interrupt in-lining: the *prepend* scheme.

4.1.2 Prepend In-Line Mode

Figure 4.2 illustrates the prepend scheme of in-lining the interrupt handler. In the first state, [state (a)], the exceptional instruction has reached the head of the reorder buffer. The hardware checks to see if it has enough space, and if it does, it saves the tail pointer into a temporary register and

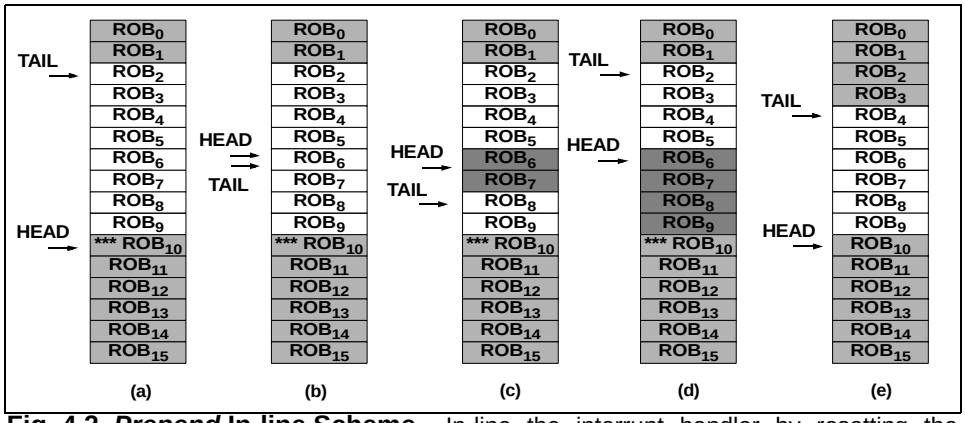

Fig. 4.2. *Prepend* In-line Scheme. In-line the interrupt handler by resetting the head and tail pointers. The figure shows the in-lining of a 4-instruction handler, assuming that the hardware fetches and enqueues two instructions at a time. Once the exceptional instruction, identified by asterisks, reaches the head of the queue, the hardware stops fetching user-level instructions (light grey), saves the current tail pointer, resets the head and tail pointers and starts fetching handler instructions (dark grey). When the entire handler is fetched, the old tail pointer is restored and the normal fetching of user instruction resumes.

moves the head and tail pointer to four instructions before the current head, shown in (b). At this point the processor is put in INLINE mode, the PC is redirected to the first instruction of the handler, and the first two instructions are fetched into the pipe. They are enqueued into the tail of the reorder buffer as usual, shown in (c). The hardware finishes fetching the handler code [state (d)], and restores the tail pointer to its original position, and continues fetching user instructions from where it originally stopped. Eventually, when the last handler handles the exception, the flag of the excepted instruction can be removed and the exceptional instruction may retry the operation [state (e)]. This implementation effectively does out-of-order committing of instructions (handler instructions that are fetched after user instructions, retire before user instructions, however instructions are

still committed in ROB order), but again, since the state modified by such instructions is transparent to the application, there is no harm in doing so.

Unlike the append scheme, in the prepend scheme the handler instructions are brought physically “ahead” of all user instructions, thus giving them the highest priority in terms of scheduling. Since the handler instructions are at the head of the reorder buffer and can retire and update state, like the append scheme, it isn’t required for the handler instructions to update state early, i.e. when they reach the commit stage (when they are pulled in from the memory pipeline stage into the commit pipeline stage), they can update state while they are committing. Since handler instructions commit their state, this allows for the prepend scheme to restore the occupied space within the reorder buffer, thus making room for more user instructions to execute or allow room for subsequent interrupt in-lining.

4.1.3 Append Scheme Vs. Prepend Scheme

At first glance, the append scheme and prepend scheme seem to be very similar and that they should have almost identical behavior. However, the implementation differences between these two schemes may show behavior that is unexpected. The schemes differ in terms of the location where the interrupt handler is in-lined. With the append scheme, the interrupt handler is embedded into the user code, whereas with the prepend scheme, the interrupt handler is brought physically before all user instructions. The fact

that the append scheme occupies reorder buffer space and prepend scheme restores the reorder buffer space can allow for the two schemes to behave differently. Additionally, the problems associated with speculative execution (e.g. branch mispredicts, load store ordering, etc.) need to be tolerated by the append scheme as the handler instructions are embedded within user code, this is discussed further in the next section. Even more, the mere fact that the handler instructions have different priorities in terms of scheduling and that handler instructions can “commit” state a few cycles earlier in the append scheme than the prepend scheme can also allow for noticeable differences in terms of performance.

4.2 Issues With Interrupt In-lining

The two schemes presented differ slightly in the additional hardware needed to incorporate them into existing high performance processors. Both the schemes require additional hardware to determine if there are enough reorder buffer entries available to fit the handler code. Since the prepend scheme exploits the properties of the head and tail pointers, an additional register is required to save the old value of the tail pointer. Besides the additional hardware, there are a few implementation issues concerning the in-lining of interrupt handlers. They include the following:

1. *The hardware knows the handler length.* To determine if the handler will fit in the reorder buffer, the hardware must know the length of the handler. If there aren't enough slots in the reorder buffer, the interrupt must be handled by the traditional method. If speculative in-lining is used, as mentioned in our future works section, this attribute is not required, but the detection and recovery from a deadlock must be incorporated.

2. *There should be a privilege bit per ROB entry.* Since both user and kernel instructions coexist within the reorder buffer when in-lining; to prevent security holes, a privilege bit must be attached to each instruction, rather than having a single mode bit that applies to all instructions in the pipe [9].

3. *Hardware needs to save nextPC and not exceptionalPC.* If the hardware determines that it can use the in-line scheme, it should save *nextPC*, i.e. the PC of the instruction that it was about to fetch had it not seen the exception. The logic here amounts to a MUX that chooses between exceptionalPC and nextPC.

4. *Hardware needs to signal the exceptional instruction when the handler is finished.* When the handler has finished execution, i.e. the exception has been satisfied, the hardware must convey this information to the

exceptional instruction, and perhaps any other instruction that has faulted for the same type of exception. For example, a TLB-miss handler must perform the following functions in addition to refilling the TLB: (1) undo any TLBMISS exceptions found in the pipeline; and (2) return those instructions affected to a previous state so that they re-access the TLB & cache. This does not need a new instruction, nor does it require existing code to be rewritten. The signal can be the update of TLB state. The reason for resetting all instructions that have missed the TLB is that several might be attempting to access the same page—this would happen, for example, if an application walking a large array walks into a new page of data that is not currently mapped in the TLB: every load/store would cause a DTLB miss. Once the handler finishes, all these would hit the TLB upon retry. Note that there is no harm in resetting instructions that cause TLB misses due to access to different pages, because these will simply cause another TLB-miss exception when they access the TLBs on the second try.

5. *After loading the handler, the "return from interrupt" instruction must be killed, and fetching resumes at nextPC, which is unrelated to exceptionalPC.* When returning from a interrupt handler, the processor must NOP the "return from interrupt" instruction, and resume fetching

at some completely unrelated location in the instruction stream at some distance from the exceptional instruction.

6. *The processor needs to make sure it isn't already stalled.* If at the time the TLB miss is discovered, the processor will need to make sure it isn't stalled in one of the critical paths of the pipeline, e.g. register renaming. A deadlock situation might occur if there aren't enough free physical registers available to map existing instructions prior to and including those in the register renaming phase. To prevent this, the processor can do one of two things: (a) handle the interrupt via the traditional method, or (b) flush all instructions in the fetch, decode, and map stage and set nextPC to the earliest instruction in the map pipeline stage. As mentioned, since most architectures reserve a handful of registers for handlers to avoid the need to save and restore user state, the handler will not stall at the mapping stage. In architectures that do not provide such registers, the hardware will need to ensure adequate physical register availability before vectoring to the handler code. For our simulations, we only simulated scheme (a).

7. *Branch mispredictions in user code should not flush handler instructions.* If, while in INLINE mode, a user-level branch instruction is found to have been mispredicted, the resulting pipeline flush should

not effect the handler instructions already in the pipeline. This means that the hardware should overwrite nextPC (described above) with the correct branch target, it should invalidate the appropriate instructions in the ROB, and it should be able to handle holes in the ROB contents. The append scheme will have to account for this, while the prepend scheme doesn't have to worry about this as all the handler instructions are physically before the interrupted instruction.

In addition, the in-lined scheme's interaction with the register-renaming mechanism is non-trivial. There are several different alternative implementations of register renaming, and each interacts with this mechanism differently. For example, a Tomasulo or RUU-style register-renaming mechanism [22, 21] tags a register's contents as "invalid" when an instruction targeting that register is enqueued, and the ID of that instruction (its reservation station number or its ROB-entry number) is stored in the register. When an instruction commits that matches the stored ID, the committed result is then stored in the register, and the register contents are then tagged as "valid".

If an in-lined interrupt handler is going to share the same register space as normal instructions, this must be modified. In the prepend scheme, because the handler instructions are enqueued after existing user instructions but commit before those user instructions, it is possible for the handler instructions to leave the register file in an incorrect state in which a register that is

targeted by an outstanding user instruction is marked “valid”, which will cause the user instruction’s result to never update the register file. The append scheme will face a similar problem in the case of branch mispredicts: handler instructions will have an incorrect state of registers.

The easy solution is to reserve registers for kernel use that are never touched by user code; for example, the MIPS register usage convention partitions the register file in exactly this fashion (the k0 and k1 registers in the MIPS TLB-miss handler code listed above are never touched by user code). The more complex solution, which allows user instructions to share the register space with kernel instructions, is for an in-lined handler to remember the previous register state and restore it once the last handler instruction commits. Note that, if this is the case, then user instructions cannot be decoded while the handler is executing, otherwise they might obtain operand values from the handler instructions instead of other user instructions.

Another register renaming scheme is that of the MIPS R10000 [26], in which a mapping table the size of the architectural register file points to a physical register file of arbitrary size. Just as in the Tomasulo mechanism, it is possible for in-lined handler instructions in a MIPS R10000 register-renaming implementation to free up physical registers that are still in use. When an instruction in the R10000 that targets register X is enqueued, a physical register from the free pool is assigned to that instruction, and the mapping table is updated to reflect the change for register X. The previous

mapping for register X is retained by the instruction, so that at instruction commit time, that physical register can be placed on the free list. This works because the instruction in question is a clear indicator of the register lifetime for register X, because the instruction targets register X, indicating that the previous contents are dead. Therefore, when this instruction commits, the previous contents of register X held in the previously mapped physical register can be safely freed. Because in-lined handler instructions are decoded after and commit before user instructions already in the pipe, the physical register that a handler instruction frees might belong to a user-instruction that is in-flight.

Just as in the Tomasulo case, the simple solution to the potential problem is to provide a set of registers that are used only by the handler instructions—but they must be physical registers, not necessarily architectural registers, because otherwise the free pool may become empty, stalling the handler instructions and therefore putting the processor into a deadlock situation. Alternatively, the hardware could verify the availability of both sufficient ROB entries *and* sufficient physical registers before committing itself to in-lining the handler code. Moreover, a committing interrupt-handler instruction, if in-lined, cannot be allowed to free up physical registers that belong to user-level instructions. Like the Tomasulo scheme, this can be avoided if the user and kernel instructions do not use the same architectural registers, and it can be solved by the handler saving and restoring register-file state.

The hardware requirements otherwise are minimal: one can staple the scheme onto an existing ROB implementation with a handful of registers, a CPU mode bit, a privilege bit per ROB entry, and some combinational logic. Instructions are still enqueued at the tail of the ROB and retired from the head, and register renaming still works as before. Precedence and dependence are addressed by design (the in-lining of the handler code). The most complex issue—that of the scheme's interaction with the register renaming mechanism—is that of ensuring correct implementation: In-line interrupt handling does not preclude a correct implementation coexistent with register renaming, they simply require a bit of diligent design in the implementation.

Chapter 5

PERFORMANCE OF LOCK-UP FREE TLBs

5.1 Experimental Methodology

5.1.1 Simulator

Our simulator models an out-of-order processor core similar to the Alpha 21264. It has 64K 2-way L1 instruction and data caches, fully associative 16/32/64/128 entry separate instruction and data TLBs with an 8KB page size. It can issue up to four instructions per cycle and can hold 80 instructions in flight at any time. It has a 72-entry register file (32 each for integer and floating point instructions, and 8 for privileged handlers), 4 integer functional units, and 2 floating point units. The model also provides 154 renaming-registers, 41 reserved for integer instructions and 41 for floating point instructions. The model also has a 21 instruction TLB miss handler. The model doesn't have any renaming registers reserved for privileged handler instructions as they are a class of integer instructions. Therefore, the hardware must know the handler's register needs as well as the handler’s length in instructions.

We chose this for two reasons: (1) the design mirrors that of the 21264; and (2) the performance results would be more conservative than otherwise.

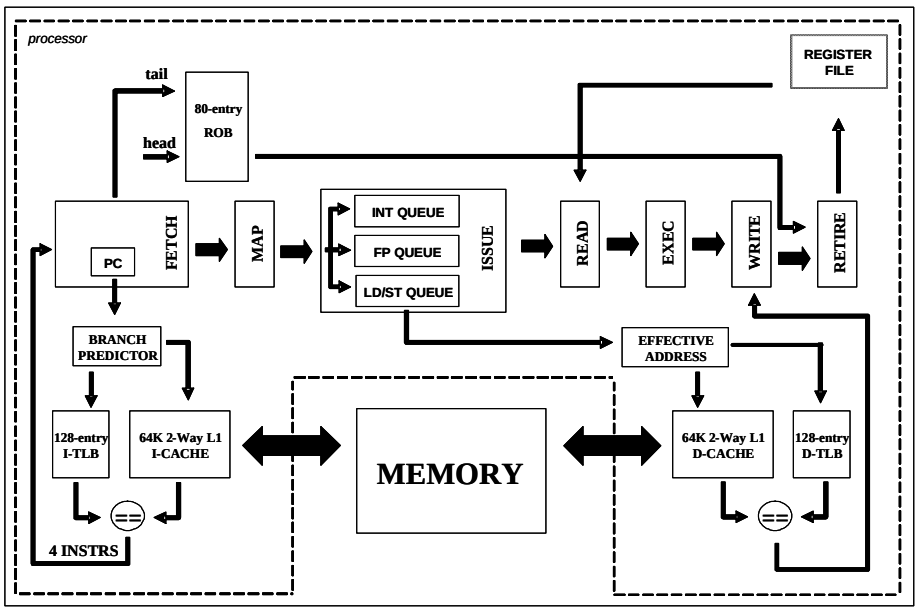

Fig. 5.1. Alpha 21264 Simulator Model. We use a cycle accurate simulator model of the ALPHA 21264. The pipeline model consists of a 4-way out-of-order processor core with an 80-entry reorder buffer, and separate integer, floating point, and load store queues. The memory model consists of 2-way 64K L1 instruction and data caches, separate fully associative 128-entry instruction and data TLBs, and an 8 KB page size.

Like the Alpha 21264 and MIPS R10000 [7, 26], our model uses a reorder buffer as well as reservation stations attached to the different functional units—in particular, the floating-point and integer instructions are sent to different execution queues. Therefore, both ROB space and execution-queue space must be sufficient for the handler to be in-lined. The page table and TLB-miss handler are modeled after the MIPS architecture [14, 12] for simplicity.

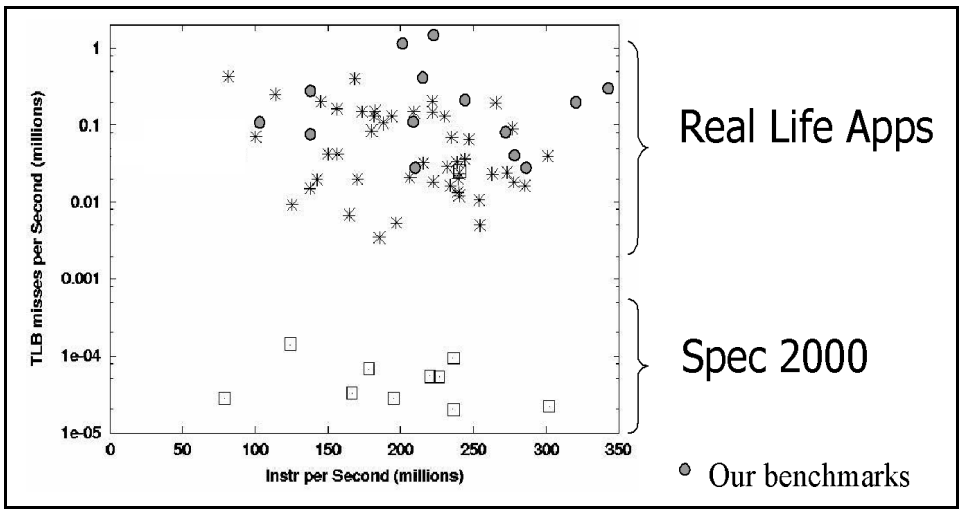

Fig. 5.2. TLB behavior of SPEC 2000 suite (source McCalpin). McCalpin, in this graph, compares the TLB behavior of SPEC 2000 to a set of "apps" that, according to him, truly represent high-performance computing. The TLB behavior of our benchmarks, shown in circles, emulate the behavior of the real life applications.

5.1.2 Benchmarks

While the SPEC 2000 suite might seem a good source for benchmarks, as it is thought to exhibit a good memory behavior, the suite demonstrates TLB miss rates that are three orders of magnitude lower than those of realistic high-performance applications. In his WWC-2000 Keynote address [2], John McCalpin presents, among other things, the graph shown in Figure 5.2, which compares the behavior of SPEC 2000 to the following set of applications that he claims are most representative of real-world high performance programs:

- Linear Finite Element Analysis (3 data sets, 2 applications)
- Nonlinear Implicit Finite Element Analysis (8 data sets, 3 applications)

- Nonlinear Explicit Finite Element Analysis (3 data sets, 3 applications)
- Finite Element Modal (Eigenvalue) Analysis (6 data sets, 3 applications)
- Computational Fluid Dynamics (13 data sets, 6 applications)
- Computational Chemistry (7 data sets, 2 applications)
- Weather/Climate Modelling (3 data sets, 2 applications)
- Linear Programming (2 data sets, 2 applications)
- Petroleum Reservoir Modelling (3 data sets, 2 applications)

The SPEC results are run with a larger page size than the apps, which would reduce their TLB miss rate, but even accounting for that, SPEC TLB miss rates are off those of McCalpin's suite by a factor of at least 10, largely because SPEC applications tend to access memory in sequential fashion [28].

McCalpin's observations are important because we will see that our work suggests that the more often the TLB requires management, the more benefits one sees from handling the interrupt by the in-line method. Therefore, we use a handful of benchmarks that display typically non-sequential access to memory and emulate the TLB behavior of McCalpin's benchmark suite (see Figure 5.2). The benchmark applications include quicksort, red-black, Jacobi, and matrix-multiply. Each of the benchmarks are run using different memory footprints: small, medium and large. Since all of

our benchmarks are array based, a small memory footprint is where the array element size is 500 bytes, a medium memory footprint is where the array element is three kilo-bytes, and a large memory footprint is where the array element is five to six kilobytes. Changing the memory footprints of these applications may change the nature of these applications, but the focus here is to attempt to emulate the TLB behavior of McCalpins' benchmark suite.

5.2 Performance of Lock-Up Free TLBs

We first take a look at how often our applications benefit from the in-line scheme. Our studies show that there are certain limitations to interrupt in-lining, most importantly available reorder buffer space, and additionally, the availability of free registers to map instructions within the pipeline. Of the times where the handler could not be in-lined, the primary reason was because the pipeline was already stalled due to an insufficient number of renaming registers. The processor can overcome this hurdle and achieve an additional performance boost by allowing for separate renaming registers for handler instructions (like the MIPS architecture does) [14] and at the same time also allowing for partial flushing of pipeline stages, namely the stages upto and including the mapping stage. Normally, when a pipeline is

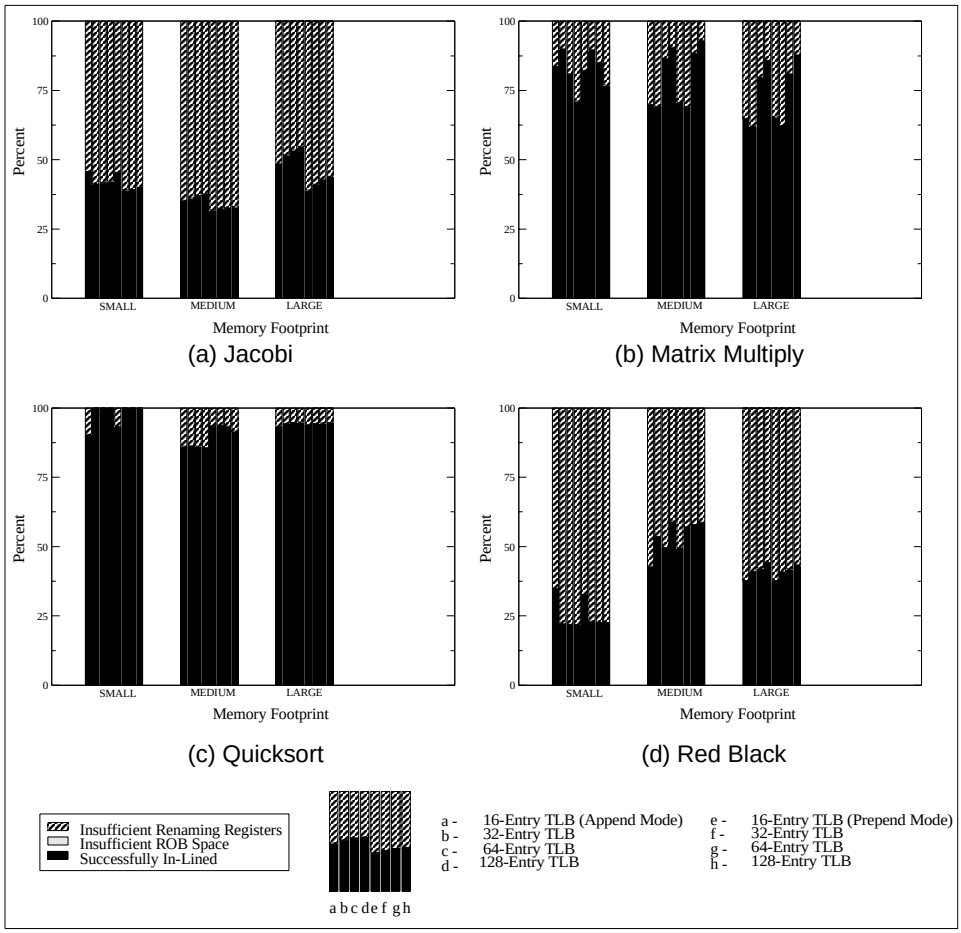

Fig. 5.3. Limitations of Interrupt In-lining. This figure shows the number of times interrupt in-lining was used, and the reasons why interrupt in-lining could not be used. The figure shows that space is not an issue, instead the pipeline being stalled due to lack of renaming registers is the primary reason for not in-lining.

flushed, it is done stage by stage, so hardware can use existing logic to flush only certain stages of a pipeline.

Figure 5.3 shows the limitations of the lock-up free scheme for the different benchmarks. We see that matrix multiply and quicksort benefit from in-lining for about 60-90% and 80-90% of the times respectively. However jacobi and red black benefit from in-lining for only 25-50% of the time.

The reason why red black and jacobi (including quicksort and matrix multiply) are not able to in-line the interrupt handler is because the pipeline is already stalled due to insufficient renaming registers. Red black and jacobi suffer tremendously because both these benchmarks in their innermost loops perform a significant amount of arithmetic computation which exhausts all available renaming registers.

The primary source of performance loss with the traditional method of interrupt handling is that there exists a significant overhead due to the large number of instructions flushed. To quantify this hypothesis, Figure 5.4 shows the average number of user instructions flushed when a D-TLB miss is detected. The x-axis represents the different memory footprint sizes, the y-axis represents the average number of instructions flushed per D-TLB miss. In each graph, the first four bars represent 16/32/64/128 entry TLBs managed by the append in-lined scheme, the next four are for those TLBs managed by the prepend in-lined scheme, and the last four are for those TLBs managed by the traditional scheme of handling interrupts (i.e. flush the ROB and pipeline). The figure shows that on average, with the traditional scheme of interrupt handling (last four bars), at the time a D-TLB miss is detected, the reorder buffer is 50-55% full. This observation is promising in that relatively large interrupt handlers can be in-lined provided there are enough resources. The benefit of in-lining can be also seen

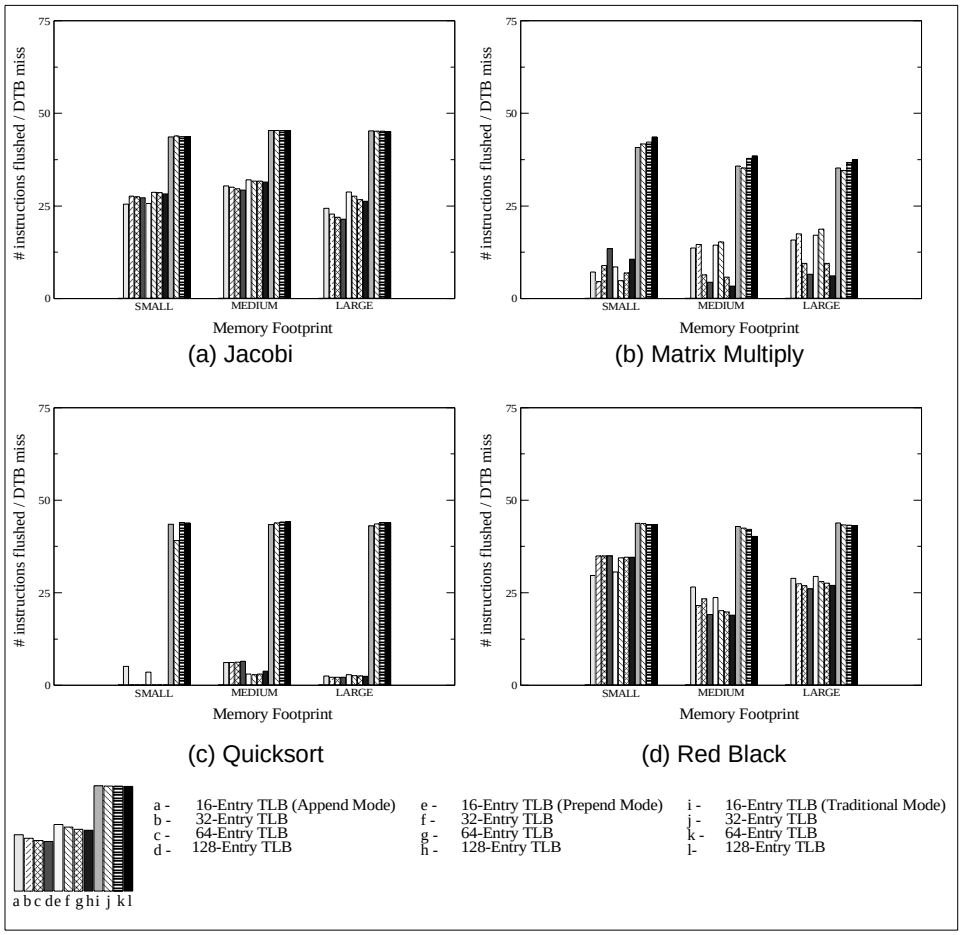

Fig. 5.4. Average Number of Instructions Flushed per DTB miss. The traditional method of handling an interrupt shows that with our 88 entry ROB, at the time the DTLB miss is detected, the ROB is 40-45% full (40-45 user instructions flushed). This is promising in that one doesn't have to restrict themselves to small handlers. Additionally, we see that in-lining significantly cuts the number of instructions flushed by 30-95%.

readily from the graphs in Figure 5.4. Both the in-lined schemes significantly reduce the number of instructions that are flushed. The figure shows that both the append and prepend scheme reduce the number of instructions flushed by 30-95%.

An interesting observation from the graphs is that the number of instructions flushed is independent of the TLB size. It would seem that increasing

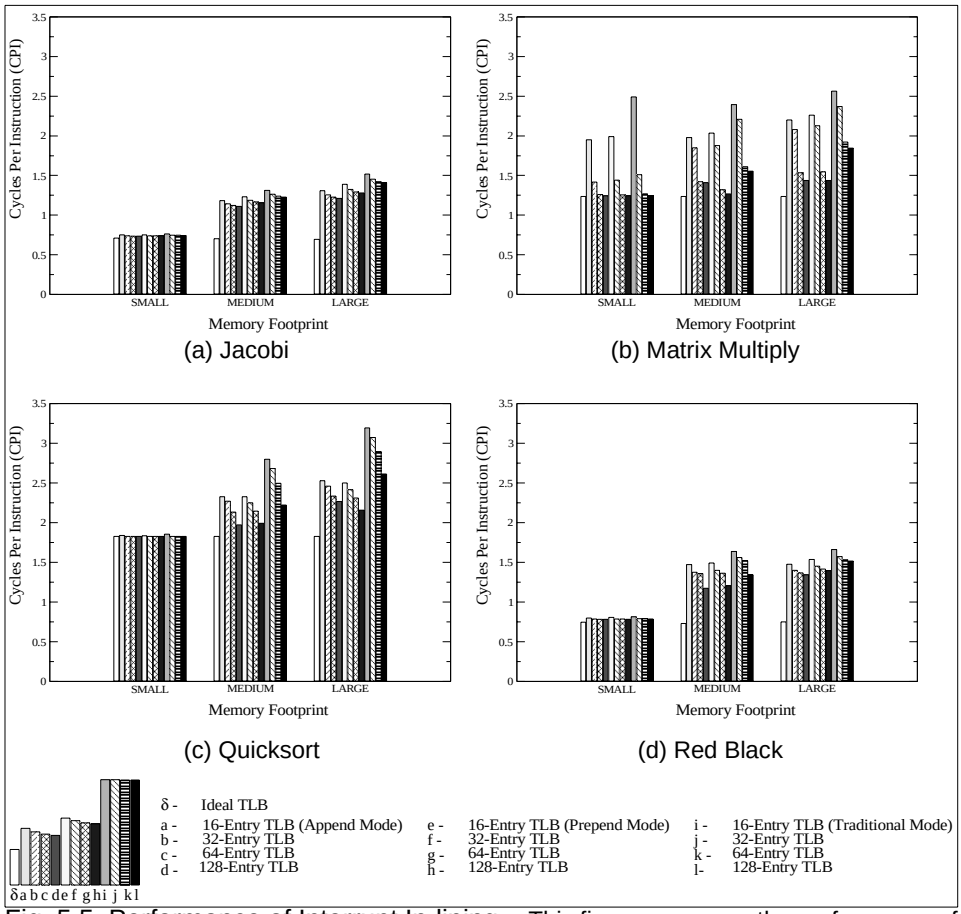

Fig. 5.5. Performance of Interrupt In-lining. This figure compares the performance of the benchmarks for an ideal TLB, append scheme, prepend scheme, and traditional TLBs. Both the schemes of in-lining improve performance in terms of CPI by 5-25%.

the TLB size should reduce the number of instructions flushed, however; this isn't the case: the number of instructions flushed can either increase or decrease. This is because number of instructions flushed on a TLB miss is dependent on the contents of the reorder buffer which is independent of the TLB size.

The lock-up-free scheme of handling interrupts allows for user instructions to execute in parallel with the interrupt handler and also significantly

reduces the overhead caused due to the flushing of user instructions. Figure 5.5 compares the performance of an ideal TLB, the lock-up-free schemes, and the traditional method of interrupt handling.

Figure 5.5 also shows that virtual memory adds a significant amount of overhead. We see that for medium and large memory footprints, we see a 25-50% performance degradation from the optimal case—a perfect TLB. This is to be expected when TLB misses are frequent, as is the case with realistic applications [28], and it is clearly important to optimize TLB management so as to reduce this overhead.

The figures show the performance benefit of using lock-up-free TLBs: for the same-size TLB, execution time is reduced by 5-25%. Another way of looking at this is that one can achieve the same performance level with a smaller TLB, if that TLB is lock-up-free. As the figures show, one can usually reduce the TLB by a factor of four by using a lock-up-free TLB, and still achieve the same performance as a traditional software-managed TLB.

Thus, we see that lock-up-free TLBs can reduce power requirements considerably: A recent study shows that a significant portion of a microprocessor's power budget can be spent in the TLB [27], even if that microprocessor is specifically a low-power design. Therefore, any reduction in the TLB's size is welcome if it comes with no performance degradation.

With respect to the different benchmarks, we see that both matrix multiply and quicksort have significant performance improvements, while red black and jacobi don't have an equivalent performance improvement. This can be explained in the fact that both red black and jacobi weren't able to benefit as much from in-lining as did matrix multiply and jacobi (see Figure 5.3).

Both the schemes provide performance improvements both in terms of the number of instructions flushed and execution time. However, performance results from Figure 5.5 show an unexpected behavior—the append scheme performs better than the prepend scheme for the different applications by 2-5%. We had expected that the append scheme would perform worse than the prepend scheme because it retains space within the reorder buffer and also the handler instructions had a lower priority in terms of scheduling. From, Figure 5.3, we see clearly that space was not an issue for interrupt in-lining, thus the small differences possibly rely on the timing of execution of the interrupt handler.

The reasons why the append scheme performs better than the prepend scheme is due to some improvements built into the append scheme. To allow for the append scheme to work, we needed the exception to be "handled" transparently without having to commit the handler instructions. Rather than wait for the *TLB write* instruction to commit and update the

TLB, we allowed for the TLB to be updated when the *TLB write* instruction "hits" the commit state, i.e. when it moves from the memory stage of the pipeline to the commit stage. Doing so, prevented the processor from entering a dead lock state, as instructions are held up at the head of the reorder buffer by the exceptional instruction. An additional enhancement incorporated was with respect to branch mispredicts. An issue with regards to the append scheme, we mentioned earlier, was in the case of branch mispredicts, the append scheme should not flush the embedded handler instructions, thus allow for "reorder buffer holes". However, if the exception has already been handled (for e.g. the TLB has been updated), then the handler instructions can be flushed along with the other user instructions, thus avoiding "reorder buffer holes" and at the same time restoring space within the reorder buffer. Thus, the timing of the TLB update and the enhancement with respect to branch mispredicts both allow for the append scheme to have behaviors that were unexpected.

The append and prepend scheme also differ with respect to overhead in terms of the number of user instructions flushed. Results from Figure 5.4 show that for Jacobi, Matrix Multiply, Quicksort (large memory footprint) and Red Black (large memory footprint) the append scheme flushes fewer instructions than the prepend scheme. For Jacobi, this can be explained in the fact that the append scheme benefitted from in-lining more than the

prepend scheme (see Figure 5.3). However, for the cases where the append and prepend scheme benefitted from in-lining equally, or where the prepend scheme benefitted from in-lining more than the append scheme, the explanation lies solely in the implementation of the two schemes. With the append scheme, the handler occupies space within the reorder buffer, thus reducing the number of user instructions within the reorder buffer. However, in the case of the prepend scheme, the handler disappears after having finished it's job, thus making room for more user instructions to be within the reorder buffer. For the cases where space was an issue, more user instructions would be flushed on average in the case of the prepend scheme, than in the case of append scheme.

The favoring of one scheme over the other depends on which scheme is easier to integrate into existing hardware. We showed earlier that the append scheme was easier to build into existing hardware and also suggested the additional minimal logic for the prepend scheme. Both schemes require the hardware to maintain separate renaming registers for the handler and also require partial pipeline flushing for an additional performance boost.

We also wanted to see if a correlation exists between an application's working-set size (as measured by its TLB miss rate) and the benefit the application sees from lock-up free TLBs. In addition to running the bench-

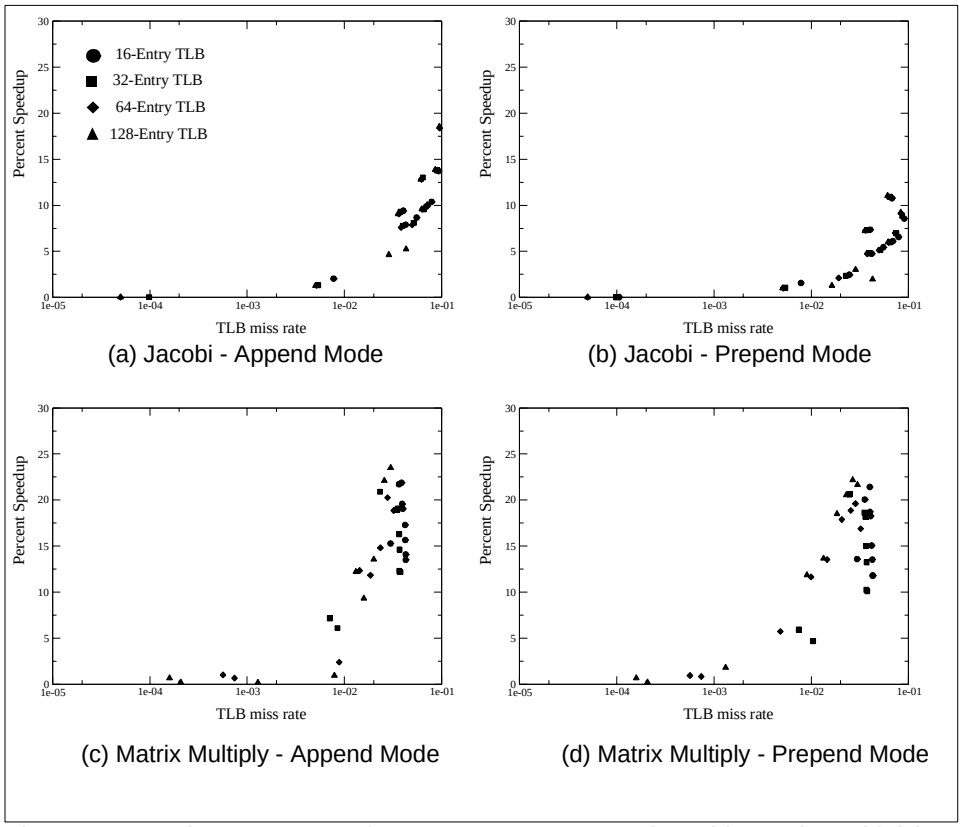

Fig. 5.6. TLB miss rate vs. Performance Improvement (Jacobi, Matrix Multiply). This figure shows that performance is independent of the TLB size, instead is dependant on the TLB miss rate. The figure shows that the more often the TLB requires management, the more the benefit the application sees from interrupt in-lining.

marks “out of the box,” we also modified the code to obtain different working-set sizes, for example by increasing the array sizes and data structure sizes. The results are shown in Figures 5.6 and 5.7, which present a scatter plot of TLB miss rate to application speedup. The figure first of all shows that performance is independent of TLB sizes, instead dependent on the TLB miss rate. We see a clear correlation between the TLB miss rate and application speedup: the more often that the TLB requires management, the more benefit one sees from lock-up free TLBs. This is a very encouraging

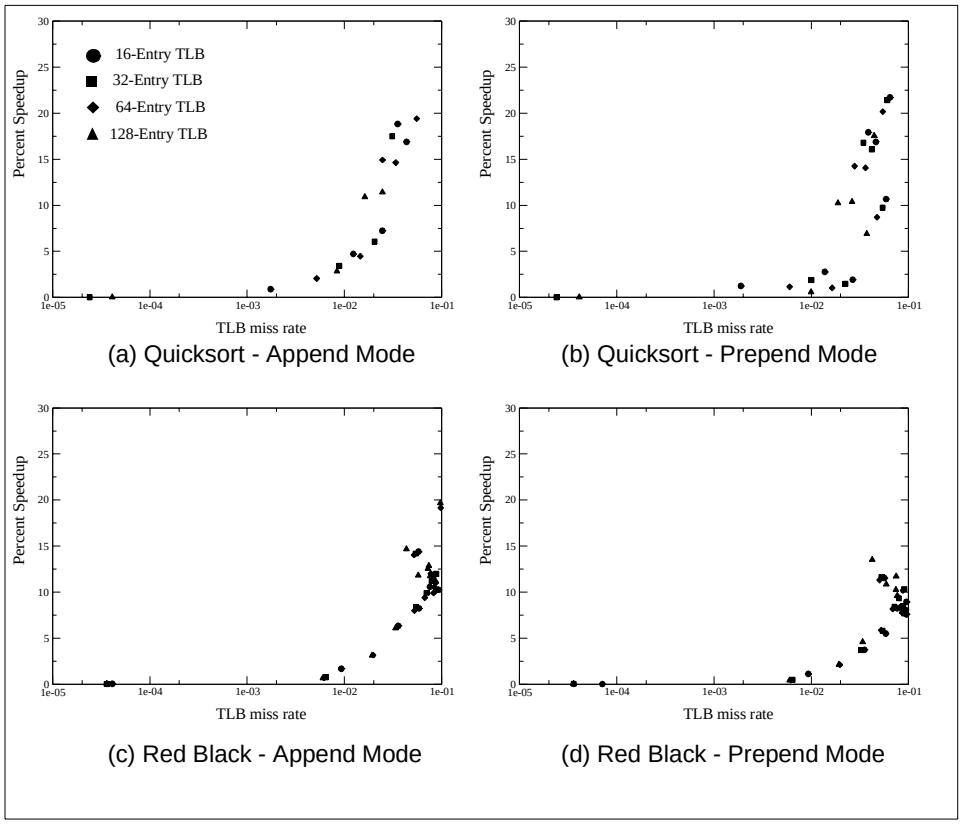

Fig. 5.7. TLB miss rate vs. Performance Improvement (Quicksort, Red Black). This figure shows that performance is independent of the TLB size, instead is dependant on the TLB miss rate. The figure shows that the more often the TLB requires management, the more the benefit the application sees from interrupt in-lining.

scenario: the applications that are likely to benefit from in-line interrupt handling are those that need it the most.

5.3 Energy Savings With Lock-Up Free TLBs

To determine the amount of energy wasted, we first characterize the properties of the instructions flushed as a result of TLB-miss alone. Figure 5.8. presents important results. Most noticeably, the absolute number of

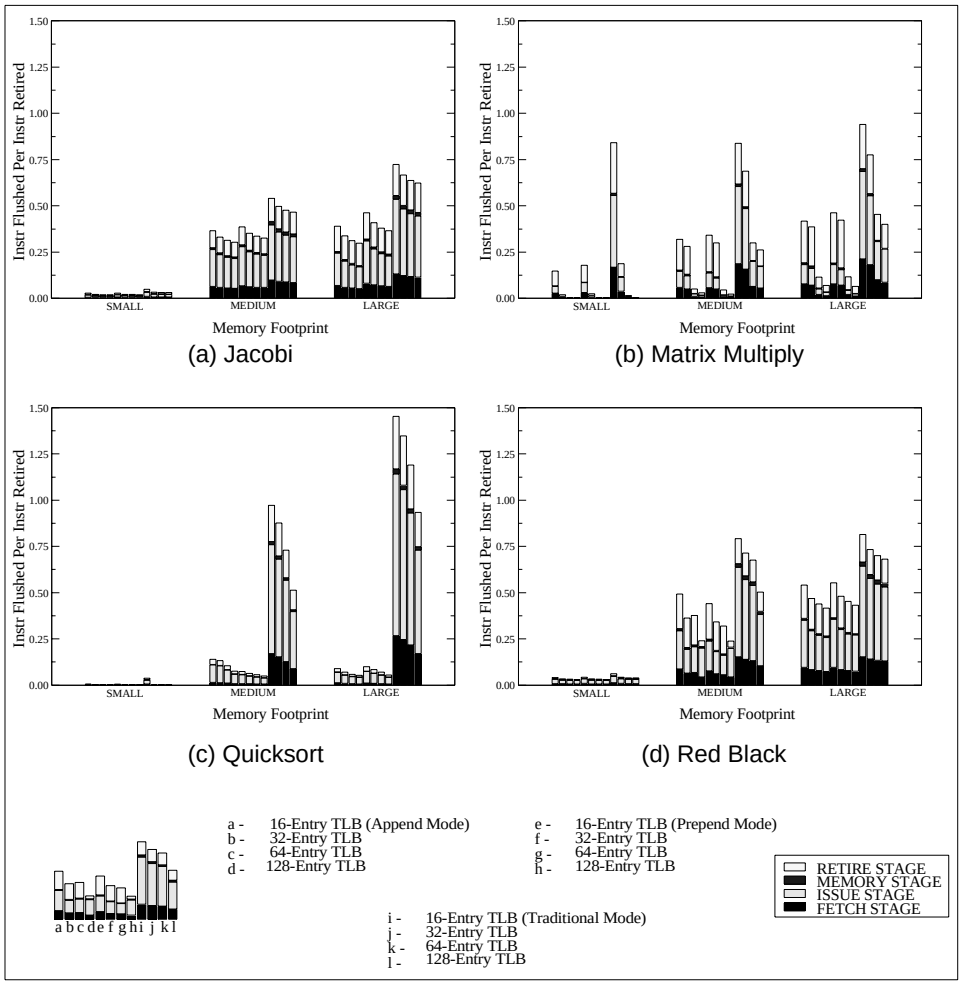

Fig. 5.8. Location of Instructions Flushed due to a TLB miss. This figure shows the stages in which the instructions were before they were flushed. The y-axis shows the number of instructions flushed for each instruction retired. In-lining significantly reduces the number of instructions flushed. Additionally, the graph shows that the majority of the instructions flushed are relatively early in the pipeline, and about 10-15% in the final stages.

instructions flushed is very large: the y-axis in the figures indicate the number of instructions flushed due to a TLB miss for each instruction that is retired. The graphs show that applications can end up flushing an enormous portion of the instructions that are fetched speculatively into the pipeline. The lock-up-free schemes thus creates a great opportunity to save time and

energy by reducing this waste. This can be seen by the effectiveness of the lock-up free schemes in reducing the number of instructions flushed—the schemes reduce instructions flushed by 30% or more.

Figure 5.8. also shows where in the pipeline the instructions are flushed. We see that the bulk of the instructions are flushed relatively early in the instruction life-cycle, i.e. before they have been executed. However, more than 50% of the instructions flushed have been decoded, renamed (mapped), and enqueued into the reorder buffer and execution queues by the time they are flushed (labeled *to be. issued*). About 10-15% of the instructions flushed have already finished execution and are waiting to be retired or are waiting to access the data cache. The figure thus shows the significant amount of work wasted and how in-line interrupt handling effectively reduces the additional overhead in terms of execution.

Even though a majority of the instructions have been flushed relatively early during their life-cycle, researchers on the Alpha 21264 processor have shown that simply fetching and mapping instructions is relatively expensive, together accounting for a fifth of the CPUs total power [30, 31]. The following are the power breakdowns for the 21264, with the IBOX detailed:

- IBOX: Integer/FP Mapper & Queue, Datapath: 32%
  Issue Unit: 50%

Map Unit: 23%

Fetch Unit: 23%

Retire Unit: 4%

- MBOX: Memory Controller: 20%
- EBOX: Integer Units (L, R) 16.5%
- DBOX: Dcache 13%
- CBOX: Biu Data & Control Busses: 12%
- JBOX: Icache 6.5%

Using these breakdowns, we computed the power breakdowns for the 21264 by pipeline stage:

- Instruction in Fetch Stage: 13.7%
- Instruction in Issue Stage: 49.2%
- Instruction in Memory Stage: 65.7%
- Instruction (ALU) in Retire Stage: 65.7%
- Instruction (MEM) in Retire Stage: 98.7%
- Retired ALU instruction: 67.0%
- Retired MEM instruction: 100%

With the power breakdowns by pipeline stage, we now quantify the energy-consumption benefits by using a lock-up-free TLB schemes. Figure 5.9 shows trends in energy savings that are very similar to the performance benefits (they differ by about 5-10%). An immediately obvious feature of the data in the Figure 5.9 is the huge amount of energy spent on partially

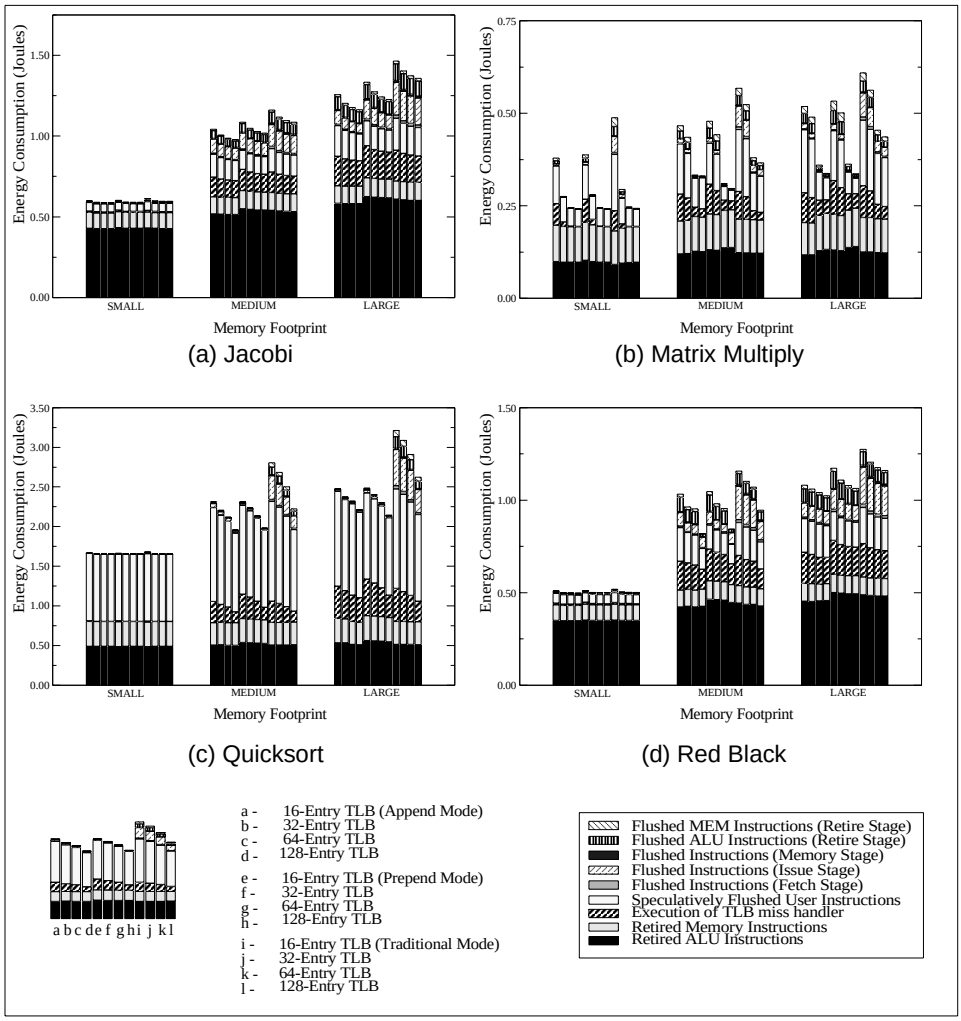

Fig. 5.9. Energy Distribution of Application. This figure shows the energy consumption of instructions that were retired, handler instructions, speculative instructions, and those that were flushed due to a TLB miss. In-lining reduces the energy wasted in re-fetching and re-executing instructions by 30-90%.

executed instructions that are flushed before they can retire. This is a significant result by itself. Today's high-performance CPUs can waste as much as quarter of their energy budget on instructions that are ultimately flushed due to TLB misses alone. Given that we have broken the triple-digit Wattage rating (Pentium 4 with 0.18 micron technology has rating of

100.6 Watts), it seems like the in-lined scheme of handling interrupts is an obvious candidate for power reductions. We see that the in-lined schemes reduce the total energy consumption by 5-25% overall, and also reduce the energy wasted in re-fetching and re-executing by 30-90%, which is very significant.

The breakdowns indicate what types of instructions contribute to the energy total. One feature to notice is that the energy used by the TLB miss handler reduces with growing TLB sizes. This is no mistake; as TLB sizes decrease, the frequency of invoking TLB-miss handlers increases, and therefore the number of handler instructions retired by the CPU also increases. Thus an increase in number of instructions executed results in an increase in energy used. This is an effect not often talked about—that different runs of the same application on different hardware might execute different numbers of instructions—but these results demonstrate that it is a significant effect.

As mentioned earlier, we found that the reduction in performance is slightly higher than the reduction in energy consumption. Execution time is reduced because the lock-up free scheme eliminates a significant number of redundant instructions that would otherwise be re-fetched and re-executed. The average joules-per-instruction for these redundant operations is slightly lower than the average joules-per-instruction for the entire run

because many are only partially executed and therefore contribute less than an average instruction to the total. Therefore, eliminating these instructions reduces power slightly less than if one eliminated fully-executed instructions.

An additional overhead in terms of energy wastage is due to those instructions that were flushed speculatively, i.e. not due to a TLB miss, but due to branch mispredictions, load store ordering, and other such exceptions. The graphs show that about 5-25% of an applications energy budget is spent in speculative execution, which is a tremendous waste in itself. Quicksort however spends as much (in fact more) energy in speculative execution as in retiring the required user instructions. This can be explained in the fact that quicksort is severely branch dominated, thus in the case of branch mispredicts the processor will have to frequently flush the ROB. This additionally explains why quicksort has been able to benefit from in-line interrupt handling the most. The reorder buffer is relatively empty, thus the pipeline is not stalled as often due to a lack of renaming registers.

In conclusion, we see that lock-up free TLBs significantly reduce the number of instructions flushed by 30-95%, reduce the energy wasted by 30-90% and improve performance and reduce the energy consumed by 5-25%.

Chapter 6

RELATED WORK

The overheads dealing with the precise interrupt mechanism have been pointed out before, but the suggested solutions have usually involved saving the entire state of a the machine or leaning towards an imprecise mechanism.

Torng & Day discuss an imprecise-interrupt mechanism that would be appropriate for handling interrupts that are transparent to application code, for example TLB-miss interrupts [23]. The system considers the contents of the instruction window (e.g. the reorder buffer) part of the machine state, and so this information is saved upon handling an interrupt. Upon exiting the handler, the instruction window contents are restored, and the pipeline picks up from where it left off. Though the mechanism is very different the behavior of this scheme is very similar to in-line interrupt handling in that instructions are not flushed from the pipe. The only difference is that, in the in-line interrupt handler mechanism, they never leave the reorder buffer, and user instructions continue to execute in parallel with handler instructions.

Qiu & Dubois recently presented a mechanism for handling memory traps that occur late in the instruction execution cycle [17]. They propose a

tagged store buffer and prefetch mechanism to hide some of the latency that occurs when memory traps are caused by events and structures distant from the CPU (for example, when the TLB access is performed near to the memory system, rather than early in the instruction-execution pipeline). Their mechanism is orthogonal to ours and could be used to increase the performance of our scheme, for example in multiprocessor systems.

Walker & Cragon [25] and Moudgill & Vassiliadis [15] present surveys of the area, and both discuss a number of alternatives for implementation of precise interrupts. Walker describes a taxonomy of possibilities, and Moudgill looks at a number of imprecise mechanisms.

Keckler et al. [34] present an alternative architecture for SMT processors, *Concurrent Event Handling*, that incorporates multi-threading into event handling architectures. Instead of handling the event in the faulting thread's architectural and pipeline registers, the exception handler is forked into it's own thread and executes concurrently with the faulting thread. Zilles et al. in [33] utilized this scheme to handle TLB miss exceptions in SMT processors. The only difference between this scheme and the in-line scheme is that the base architecture is different: SMT vs. non-SMT processor architecture.

Chapter 7

CONCLUSIONS

7.1 Conclusions

The general purpose precise interrupt mechanisms in use for the past few decades have received very little attention. With the current trends in processor and operating systems design, the overhead of re-fetching and re-executing instructions is severe for applications that incur frequent interrupts. One example is the increased use of the interrupt mechanism to perform memory management—to handle TLB misses in today's microprocessors. This is putting pressure on the interrupt mechanism to become more lightweight.

We propose the use of in-line interrupt handling, where the reorder buffer is not flushed on an interrupt unless there isn't enough space for the handler instructions. This allows the user application to continue executing while an interrupt is being serviced. For our studies, we in-lined the TLB interrupt handler to provide us with lock-up free TLBs. For a software-managed TLB miss, this means that only those instructions stall that are dependent on the instruction that misses the TLB. All other user instructions continue executing in parallel with the handler instructions, and are only held up at commit (by the instruction that missed the TLB).

We present the *append* and *prepend* schemes of in-lining of the interrupt handler. The *append* scheme temporarily stops fetching user code and inserts the handler instructions after the user-instructions, thus retiring them in program order. The *prepend* scheme however utilizes the head and tail properties of the reorder buffer and inserts the handler instructions before the user-instructions, thus retiring them out of fetch order without any side affects.

With interrupt in-lining, at the time the processor detects an exception, it first checks if there is enough space within the reorder buffer for the interrupt to be in-lined. If their isn't enough space, then the processor would handle the interrupt by the traditional scheme, i.e. flushing the pipeline. Our studies additionally show another limitation of interrupt in-lining: pipeline stalls. If the pipeline is already stalled when the exception is detected, then the processor would lock-up if the mode were changed to INLINE. For our studies, we noted that the primary cause for the pipeline being stalled is due to insufficient renaming registers available to map the existing user instructions in the pipeline.

Our studies show that a lack of renaming registers, and not ROB space, is the primary reason for not being able to in-line TLB interrupts. A possible solution to this problem is to allow for partial flushing of the fetch, decode,

and map stages of the pipeline, and additionally reserve a set of renaming registers for handler instructions.

Our studies also show that lock-up free TLBs reduces the overhead due to the flushing of instructions by 30-95%. This is significant in that the processor no longer has to waste time or energy in re-fetching and re-executing instructions. Rather, it can spend time fetching and executing instructions independent of the exception causing instruction.

Since the number of instructions flushed is reduced dramatically, this allows for lock-up free TLBs to provide a performance improvement of 5-25%. Additionally, we see that one can achieve the same performance level with a smaller TLB, if that TLB is lock-up-free. Our results show that one can usually reduce the TLB size by a factor of four by using a lock-up-free TLB, and still achieve the same performance as a traditional software-managed TLB.

We also saw that applications that often require TLB management, receive the most benefit from in-lining and also gained a higher performance improvement (about 25-30%). This is a very encouraging scenario: the applications that are likely to benefit from in-line interrupt handling are those that need it the most.

The use of lock-up free TLBs doesn’t only help in performance improvement, but also reduces the energy consumption by a similar amount. By not

having to re-fetch and re-execute instructions, the processor spends time doing useful work by executing those instructions independent of the exception causing instruction. An investigation of the instructions flushed revealed that about 10-15% of the instructions flushed had already finished execution, and the bulk of the instructions were waiting to be issued to functional units. One can easily overlook this fact, but Alpha researchers show that as much as one fifth of the 21264's energy consumption is spent in the fetching, mapping, and queuing of instructions. This is a significant result by itself. Our studies show that today's high-performance CPUs can waste as much as a quarter of their energy budget on instructions that are ultimately flushed due to TLB misses alone. In-line interrupt handling reduces this waste by 30-90%. Given that we have broken the triple-digit Wattage rating (Pentium 4 - 100.6 Watts), it seems like the in-lined approach of handling interrupts is an obvious candidate for power reductions.

In conclusion, in-line interrupt handling reduces the two sources of performance loss caused by the traditional method of handling interrupts. In-line interrupt handling avoids the re-fetching and re-executing of instructions, and allows for user instructions and handler instructions to execute in parallel. In-line interrupt handling can be used for all types of *transparent* interrupts, i.e. interrupts that perform behind the scenes work on behalf of

the running program. One such example is the TLB interrupt. In-lining the TLB interrupt provides for *lock-up* free TLBs and reduces the number of instructions flushed by 30-95%, reduces execution time and energy consumption by 5-25% and reduces the energy wasted by 30-90%.

7.2 Future Work

For the purpose of this thesis, we propose non-speculative interrupt in-lining, i.e. the hardware knows the length of the interrupt handler (or knows of an upper limit) before hand. It is possible however to do speculative interrupt in-lining, where the hardware in-lines the interrupt handler without checking to see if there is enough reorder buffer space. With such a scheme, the processor will need to be able to detect a deadlock. If the processor detects a deadlock, it will flush the pipeline and reorder buffer and handle the interrupt by the traditional scheme.

Additionally, since interrupt in-lining avoids flushing the pipeline, it is also possible to handle interrupts speculatively, i.e. one doesn't need to wait till commit time to decide on whether or not the interrupt be handled. With the growing lengths in pipelines, and the fact that modern microprocessors wait to retire instructions in large chunks (for example 4-8 instructions at a time) the time to handle an interrupt increases. By handling

interrupts speculatively, we can allow for exceptional instructions and their dependencies to finish executing early, thus improving performance.

BIBLIOGRAPHY

[1] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. "The interaction of architecture and operating system design." In *Proc. Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, April 1991, pp. 108–120.

[2] A. W. Appel and K. Li. "Virtual memory primitives for user programs." In *Proc. Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, April 1991, pp. 96–107.

[3] B. Case. "AMD unveils first superscalar 29K core." *Microprocessor Report*, vol. 8, no. 14, October 1994.

[4] B. Case. "x86 has plenty of performance headroom." *Microprocessor Report*, vol. 8, no. 11, August 1994.

[5] Z. Cvetanovic and R. E. Kessler. "Performance analysis of the Alpha 21264-based Compaq ES40 system." In *Proc. 27th Annual International Symposium on Computer Architecture (ISCA'00)*, Vancouver BC, June 2000, pp. 192–202.

[6] L. Gwennap. “Intel’s P6 uses decoupled superscalar design.” *Microprocessor Report*, vol. 9, no. 2, February 1995.

[7] L. Gwennap. “Digital 21264 sets new standard.” *Microprocessor Report*, vol. 10, no. 14, October 1996.

[8] D. Henry, B. Kuszmaul, G. Loh, and R. Sami. “Circuits for wide-window superscalar processors.” In *Proc. 27th Annual International Symposium on Computer Architecture (ISCA’00)*, Vancouver BC, June 2000, pp. 236–247.

[9] D. S. Henry. “Adding fast interrupts to superscalar processors.” Tech. Rep. Memo-366, MIT Computation Structures Group, December 1994.

[10] J. Huck and J. Hays. “Architectural support for translation table management in large address space machines.” In *Proc. 20th Annual International Symposium on Computer Architecture (ISCA’93)*, May 1993, pp. 39–50.

[11] B. L. Jacob and T. N. Mudge. “A look at several memory-management units, TLB-refill mechanisms, and page table organizations.” In *Proc. Eighth Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’98)*, San Jose CA, October 1998, pp. 295–306.

[12] B. L. Jacob and T. N. Mudge. “Virtual memory in contemporary microprocessors.” *IEEE Micro*, vol. 18, no. 4, pp. 60–75, July/August 1998.

[13] B. L. Jacob and T. N. Mudge. “Virtual memory: Issues of implementation.” *IEEE Computer*, vol. 31, no. 6, pp. 33–43, June 1998.

[14] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs NJ, 1992.

[15] M. Moudgill and S. Vassiliadis. “Precise interrupts.” *IEEE Micro*, vol. 16, no. 1, pp. 58–67, February 1996.

[16] X. Qiu and M. Dubois. “Tolerating late memory traps in ILP processors.” In *Proc. 26th Annual International Symposium on Computer Architecture (ISCA’99)*, Atlanta GA, May 1999, pp. 76–87.

[17] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. “The impact of architectural trends on operating system performance.” In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP’95)*, December 1995.

[18] B. Shriver and B. Smith. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. IEEE Computer Society Press, Los Alamitos CA, 1998.

[19] M. Slater. “AMD’s K5 designed to outrun Pentium.” *Microprocessor Report*, vol. 8, no. 14, October 1994.

[20] J. E. Smith and A. R. Pleszkun. “Implementation of precise interrupts in pipelined processors.” In *Proc. 12th Annual International Symposium on Computer Architecture (ISCA’85)*, Boston MA, June 1985, pp. 36–44.

[21] G. S. Sohi and S. Vajapeyam. “Instruction issue logic for high-performance, interruptable pipelined processors.” In *Proc. 14th Annual International Symposium on Computer Architecture (ISCA’87)*, June 1987.

[22] R. M. Tomasulo. “An efficient algorithm for exploiting multiple arithmetic units.” *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[23] H. C. Torng and M. Day. “Interrupt handling for out-of-order execution processors.” *IEEE Transactions on Computers*, vol. 42, no. 1, pp. 122–127, January 1993.

[24] M. Upton. *Personal communication*. 1997.

[25] W. Walker and H. G. Cragon. “Interrupt processing in concurrent processors.” *IEEE Computer*, vol. 28, no. 6, June 1995.

[26] K. C. Yeager. “The MIPS R10000 superscalar microprocessor.” *IEEE Micro*, vol. 16, no. 2, pp. 28–40, April 1996.

[27] T. Juan, T. Lang, and J.J. Navarro. “Reducing TLB power requirements.” In *Proc. 1997 IEEE International Symposium on Low Power Electronics and Design (ISLPED’97)*, Monterey CA, August 1997, pp. 196-201.

[28] J. McCalpin. *An Industry Perspective on Performance Characterization: Applications vs Benchmarks*. Keynote address at Third Annual IEEE Workshop on Workload Characterization, Austin TX, September 16, 2000.

[29] A. Jaleel and B. Jacob. “In-Line Interrupt Handling for Software-Managed TLBs.” *Proc. 2001 IEEE InternationalþConference on ComputerþDesign (ICCD 2001)*, Austin TX, September 2001.

[30] K. Wilcox and S. Manne. *Alpha Processors: A History of Power Issues and A Look to the Future*. Compaq Computer Corporation, 2001.

[31] M. K. Gowan, L. L. Biro, D. B. Jackson. “Power considerations in the design of the Alpha 21264 microprocessor.” In *35th Design Automation Conference*.

[32] A. Jaleel and B. Jacob. “Improving the Precise Interrupt Mechanism for Software Managed TLB Interrupts” *Proc. 2001 IEEE. Internation*

Conference on High Performance Computing (HIPC 2001), Hyderabad, India, December 2001.

[33] C.B. Zilles, J.S. Emer, and G.S. Sohi, "The Use of Multithreading for Exception Handling", In *Proc. 32nd International Symposium on Microarchitecture*, pp 219-229, Nov. 1999.

[34] Stephen W. Keckler, Andrew Chang, Whay S. Lee, Sandeep Chatterjee, and William J. Dally, "Concurrent Event Handling through Multithreading", *IEEE Transactions on Computers*, 48:9, September, 1999, pp 903-916.