Improving the Precise Interrupt Mechanism of Software-Managed TLB Miss Handlers

Aamer Jaleel and Bruce Jacob

Electrical & Computer Engineering
University of Maryland at College Park
{ajaleel,blj}@eng.umd.edu

Abstract. 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, many of these instructions have reached a very deep stage in the pipeline - representing significant work that is wasted. In addition, an overhead of several cycles can be expected in re-fetching and re-executing these instructions. This paper concentrates on improving the performance of precisely handling software managed translation lookaside buffer (TLB) interrupts, one of the most frequently occurring interrupts. This paper 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. We simulate two different schemes of in-lining the interrupt on a processor with a 4-way out-of-order core similar to the Alpha 21264. We also analyzed the overhead of re-fetching and re-executing instructions when handling an interrupt by the traditional method. We find that our schemes significantly cut back on the number of instructions being re-fetched by 50-90%, and also provides a performance improvement of 5-25%.

1 INTRODUCTION

1.1 The Problem

With the continuing efforts to maximize instruction level parallelism, the out-of-order issue of instructions has drastically increased the utilization of the precise interrupt mechanism to handle exceptional events. Most of these exceptions are transparent to the user application and are only to perform “behind-the-scene” work on behalf of the programmer [27]. Such exceptions are handled via hardware or software means. In this paper, we will be concentrating on those handled via software. Some examples of such exceptions are unaligned memory access, instruction emulation, TLB miss handling.

Among these exceptions, 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 stud-

B. Monien, V.K. Prasanna, S. Vajapeyam (Eds.): HiPC 2001, LNCS 2228, pp. 282−293, 2001.

ies 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].

With the current trends in processor and operating systems design, the cost of handling exceptions precisely is also 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 instructions' 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. While the instructions are being committed, the exception flag of the instruction is checked. If the instruction caused an exception and software support is needed, the hardware handles the interrupt in the following way:

The ROB is flushed; the exceptional PC[1] is saved; the PC is redirected to the appropriate handler

1. Handler code is executed, typically with privileges enabled
2. Once a return from interrupt instruction is executed, the exceptional PC is restored, and the program resumes execution

In this model, 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.

1.2 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 scheme, where a new thread fetches the handler code [27].

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

1. Exceptional PC depends on the exception class. Certain interrupts, such as the TLB interrupts, require the exception causing instruction to re-execute thus 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.

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 also 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 the ROB, we in-line the interrupt handler code without flushing the pipeline. If there are not 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.

Though the mechanism is applicable to all types of interrupts (with relatively short handlers), we focus on only one interrupt in this paper—that used by a software-managed TLB to invoke the first-level TLB-miss handler. We do this for several reasons:

1. As mentioned previously, 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) [16, 12]
3. These handlers also tend to have deterministic length (i.e., they tend to be straight-line code—no branches)

This will give us the flexibility of software-managed TLBs without the performance impact of taking a precise interrupt on every TLB miss. Note that hardware-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. The scheme thus enables software-managed TLBs to reach the same performance as non-blocking hardware-managed TLBs without sacrificing flexibility [11].

1.3 Results

We evaluated two implementations of the in-lined mechanism, *append*: inserting the handler after existing code, and *prepend*: inserting the handler before existing code, 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.

An in depth analysis of the instructions flushed on an interrupt shows that 20-30% of those flushed have finished execution, 50-60% are waiting for execution units, and the remaining are waiting to be decoded and register rename. This shows significant waste in both execution time and energy consumption.

The schemes cut the TLB-miss overhead by 10–40% [28], the number of instructions flushed by 50-90%. When applications generate TLB misses frequently, this reduction in overhead amounts to a performance improvement of 5-25% in execution time in the *prepend* scheme and 5-10% in the *append* scheme. Our scheme only considers the data-TLB misses; we will be considering instruction-TLB misses next, as mentioned in our future work section.

2 BACKGROUND

2.1 Reorder Buffers and Precise Interrupts

Most contemporary pipelines allow instructions to execute out of program order, thereby taking advantage of idle hardware and finishing earlier than they otherwise would have—thus increasing overall performance. To provide precise interrupts in such an environment typically requires a reorder buffer (ROB) or a ROB-like structure in which instructions are brought in at the tail, and retired at the head [20, 21]. The reorder buffer queues up partially-completed instructions so that they may be retired in-order, thus providing the illusion that all instructions are executed in sequential order—this simplifies the process of handling interrupts precisely.

There have been several influential papers on precise interrupts and out-of-order execution. In particular, Tomasulo [22] gives a hardware architecture for resolving inter-instruction dependencies that occur through the register file, thereby allowing out-of-order issue to the functional units; Smith & Pleszkun [20] describe several mechanisms for handling precise interrupts in pipelines with in-order issue but out-of-order completion, the reorder buffer being one of these mechanisms; Sohi & Vajapeyam [21] combine the previous two concepts into the register update unit (RUU), a mechanism that supports both out-of-order instruction issue and precise interrupts (as well as handling branch misspeculations).

2.2 The Persistence of Software-Managed TLBs

It has been known for quite some time that hardware-managed TLBs outperform software-managed TLBs [11, 16]. Nonetheless, most modern high-performance architectures use software-managed TLBs (eg. MIPS, Alpha, SPARC, PA-RISC), not hardware-managed TLBs (eg. IA-32, PowerPC), largely because of the increased flexibility inherent in the software-managed design [12], and because redesigning system software for a new architecture is non-trivial. Simply redesigning an existing architecture to use a completely different TLB is not a realistic option. A better option is to determine how to make the existing design more efficient.

2.3 Related Work

In our earlier work, we presented the *prepend* method of handling interrupts [28]. When the processor detects a TLB miss, it checks to see if enough space exists within the reorder buffer and enough resources exist, if so, it sets the processor to INLINE mode, resets the head and tail pointer, and starts fetching handler code into the empty space of the reorder buffer. Section 3 of this paper presents this scheme again for ease.

Torng & Day discuss an imprecise-interrupt mechanism appropriate for handling interrupts that are transparent to application program semantics [23]. The system considers the contents of the instruction window (i.e., the reorder buffer) part of the machine state, and so this information is saved when 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 scheme could be used for handling TLB-miss interrupts, it is more likely to be used for higher-overhead interrupts. Frequent events, like TLB misses, typically invoke low-overhead interrupts that use registers reserved for the OS, so as to avoid the need to save or restore any state whatsoever. Saving and restoring the entire ROB would likely change TLB-refill from a several-dozen-cycle operation to a several-hundred-cycle operation.

Qiu & Dubois recently presented a mechanism for handling memory traps that occur late in the instruction lifetime [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.

Zilles, Emer, & Sohi recently presented a multithreaded mechanism to handle precise interrupts [27]. They propose the creation of a second thread that fetches the exceptional handler. After the second thread is done finishing the handler, the main thread continues fetching user level instructions. Their mechanism is similar to ours, with the added necessity that the processor be able to create and handle multiple threads.

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

3 IN-LINE INTERRUPT HANDLING

We present two methods of in-lining the interrupt handler within the reorder buffer. Both of our schemes exploit the property of a reorder buffer: instructions are brought in at the tail, and retired from the head [20]. If there is enough room between the head and the tail for the interrupt handler to fit, we essentially inline 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 a TLB miss is encountered. Inserting the handler instructions before the user-instructions, the *prepend* scheme [28], 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 and reservation-station entries. A detailed description of the two in-lining schemes follows:

1. ***Append in-line mode***: Figure 1 illustrates the *append* scheme of inlining 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 updated the TLB, the processor can reset the flag on 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, the handler is nonetheless allowed to affect the state of that exceptional instruction (which logically precedes the handler, according to its relative placement within the ROB). Though this may seem to imply out-of-order instruction commit, it is current practice in the design of modern high-performance processors. For example, the Alpha's TLB-write instructions modify the TLB state once they have finished execution and not at instruction-commit time. In many cases, this does not represent an inconsistency, as the state modified by such handler instructions is typically trans-

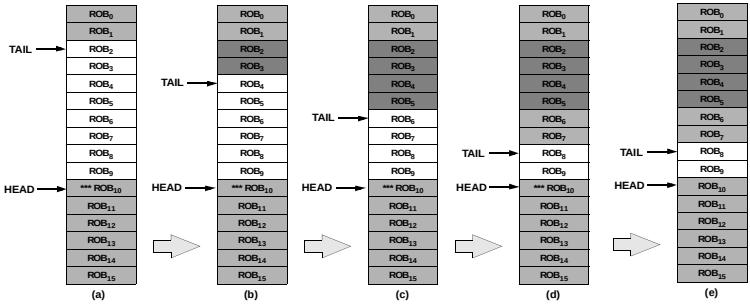

Fig. 1. **Append Scheme:** This figure illustrates the in-lining of a 4-instruction handler, assuming that the hardware fetches and enqueues two instructions at a time. The hardware stops fetching user-level instructions (light grey) and starts fetching handler instructions (dark grey) once the exceptional instruction, identified by asterisks, reaches the head of the queue. When the processor finishes fetching the handler instructions, it can resume fetching the user instructions When the handler instruction updates the TLB, the processor can reset the flag of the excepted instruction and it can reaccess the TLB.

parent to the application—for example, the TLB contents are merely a hint for better address translation performance.

2. ***Prepend in-line mode*** [28]: Figure 2 illustrates the *prepend* scheme of inlining 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 head and tail pointer into temporary registers and 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 instruction fills the TLB, the flag of the excepted instruction can be removed and the exceptional instruction may re-access the TLB [state (e)]. This implementation effectively does out-of-order committing of handler instructions, but again, since the state modified by such instructions is transparent to the application, there is no harm in doing so.

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, additional registers are required to save the old values of the head and tail pointers. As we shall see later, incorporating these additional registers will allow for the *prepend* scheme to outperform the *append* scheme by 20-30%. 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 [28]. 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 withing the reorder buffer when inlining; to prevent security holes, a privilege

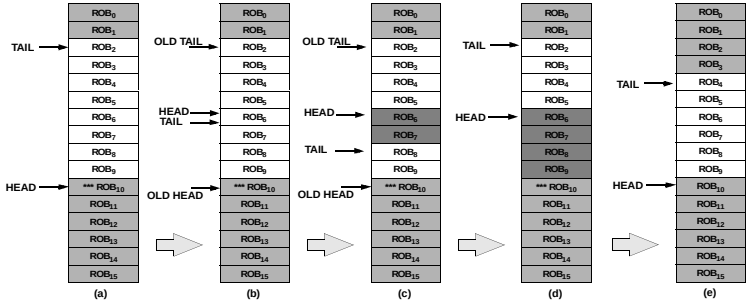

Fig. 2. **Prepend scheme:** This figure illustrates the in-lining of a 4-instruction handler, assuming that the hardware fetches and enqueues two instructions at a time. The hardware stops fetching user-level instructions (light grey), saves the current head and tail pointers, resets the head and tail pointers and starts fetching handler instructions (dark grey) once the exceptional instruction, identified by asterisks, reaches the head of the queue. When the entire handler is fetched, the old tail pointer is restored and the normal fetching of user instruction resumes.

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 signal the exceptional instruction when the handler is finished.* When the handler has finished updating the TLB, it should undo any TLBMISS exceptions found in the pipeline, and restore those instructions affected to a previous state so they can re-access the TLB & cache. The signal can be the update of the TLB state while in INLINE mode [28].
4. *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 [28].
5. *In-lined handler instructions shouldn't affect the state of user registers.* Since handler instructions are brought in after the excepted instruction but commit before the excepted instruction, we have to make sure that when they commit, they don't wrongly update and release user registers. To fix this, when mapping the first handler instruction, the handler instruction should receive a copy of the current committed register file state rather than the register state of the previous instruction. Additionally, when a user instruction is being mapped after the handler is completely fetched, it should copy the register state from a previous user instruction, whose location can be stored in a temporary register. The logic here amounts to a MUX [28].
6. *The hardware might need to know the handler's register requirements.* 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, eg. 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 the traditional method, or (b) flush all instructions in the fetch and decode stage and set nextPC to the earliest instruction in the decode/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

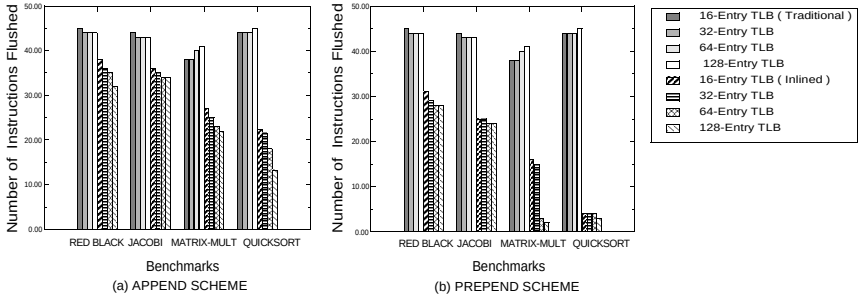

Fig. 3. This figure compares the average number of instructions flushed when handling a TLB miss by the traditional method to those of the two inlined schemes. The append scheme isn't able to reduce the number of instructions being flushed as it takes up space within the reorder buffer, while the prepend scheme is able to reduce the number of instructions being flushed significantly.

before vectoring to the handler code [28]. 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.

Overall, the hardware design is relatively simple, requiring beyond this a status bit that identifies when the processor is handling interrupts in this manner. Otherwise, the design of the processor is unmodified.

4 THE PERFORMANCE OF IN-LINING INTERRUPTS

4.1 Simulation Model

We model an out-of-order processor similar to the Alpha 21264. It has 64K/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 82 free renaming-registers, 32 reserved for integer instructions and 32 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 handlers as they are a class of integer instructions. Therefore, the hardware must know the handler's register needs as well as 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.

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, and

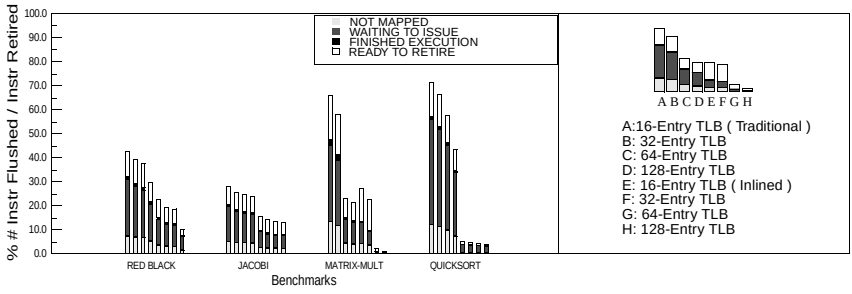

Fig. 4. This figure shows the stages in which the instructions were before they were flushed. The graphs show that 15-30% of the instructions have already finished execution, 50-60% are waiting to be dispatched to execution units, and the remaining are awaiting decoding and register renaming. In-lining using the prepend scheme cuts back significantly on the number of instructions being flushed.

instruction-issue to the execution queues stalls for user-level instructions during the handler execution. The page table and TLB-miss handler are modeled after the MIPS architecture [14, 12] for simplicity.

4.2 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, a comparison between SPEC 2000 and a set of more real-world high-performance programs. The reason why SPEC applications don't portray "real world applications" is because they tend to access memory in sequential fashion [2].

McCalpin's observations are important because our previous work suggests that the more often the TLB requires management, the more benefits one sees from handling the interrupt by the in-line method [28]. Therefore, we use a handful of benchmarks that display typically non-sequential access to memory and have correspondingly higher TLB miss rates than SPEC 2000. The benchmark applications include quicksort, red-black, Jacobi, and matrix-multiply.

4.3 Results

We first take a look at the number of instructions that are flushed on average when a TLB miss is detected. Figure 3 shows that when handling TLB misses traditionally, the reorder buffer, at the time that a TLB miss is detected, is only 50% full. The figure shows that with our schemes of in-lining the interrupt, *prepend* scheme performs the best in terms of reducing the number of instructions being flushed. Our studies show that roughly 60% of the time TLB interrupts in our benchmarks were able to benefit from in-lining using the *append* scheme, while 80-90% of the time interrupts were able to benefits using the *prepend* scheme. This can be explained in the fact that the *append* scheme retains the miss handler code within the reorder buffer after having finished refilling the TLB, thus occupying reorder buffer space, while in the *prepend* scheme the handler instructions exit the reorder buffer, thus restoring the amount of free space within the reorder buffer. We find that in both schemes, 10-20% of the time the handler cannot be in-lined due to insufficient physical registers available to map the user instructions already present in the pipeline. As men-

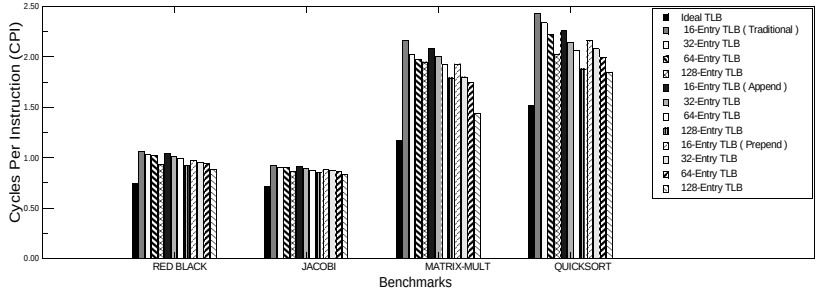

Fig. 5. The figure shows the execution time (cycles-per-user-instructions) of a perfect TLB, 16/32/64/128-entry traditionally handled TLBs, append in-lined TLBs, and prepend in-lined TLBs.

tioned earlier, the in-lined interrupt method can still be made to work if we allow for partial flushing of pipeline stages. Most modern high performance processors, like the Alpha 21264, currently allow for such techniques.

We further investigate the properties of all the instructions that were flushed due to TLB misses. Figure 4 shows the stages at which the instructions were flushed when the TLB miss handler was executed. The first four bars show the statistics for the traditional method of handling interrupts using 16/32/64/128 entry TLBs, while the last four bars show the same with the *prepend* scheme. The x-axis shows the different benchmarks, and the y-axis represents the ratio of the total number of instructions flushed (because of TLB misses alone) to the number of user instructions retired. The graph first of all shows that in the traditional scheme, the number of instructions (including speculative) flushed are about 20-70% of the instructions that are retired, therefore, representing significant work wasted. Of the instructions being flushed: 20-30% of the instructions being flushed have already finished execution; 50-60% are waiting to be dispatched to execution units, and the others are awaiting instruction decoding and register renaming. This overhead causes more time and energy to be consumed in re-fetching and re-executing the instructions. The *prepend* in-lined scheme reduces the number of instructions being flushed by 50-90% and the *append* in-lined scheme by 25-50%. Thus the in-lined methods significantly reduce the tremendous overhead in handling TLB interrupts using traditional means.

Additionally, we compare the performance of perfect TLBs, traditional software-managed TLBs, and in-lined TLBs. Figure 5. shows the *append* scheme reducing the execution time by 5-10% and the *prepend* scheme reducing the execution time by 5–25% for the same-size TLB. The significant performance difference again is due to the interrupt handler retaining space within the reorder buffer in the *append* scheme. Both the schemes provide performance improvements both in terms of the number of instructions flushed and execution time. 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, but also suggested the additional minimal logic for the *prepend* scheme.

5 CONCLUSIONS & FUTURE WORK

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 a software-managed TLB miss, this means that only those instructions stall that are dependent on the instruction that misses the TLB. All other instructions continue executing, and are only held up at commit (by the instruction that missed the TLB).

We present the *append* and *prepend* schemes that allow the in-lining of the interrupt handler. The *append* scheme inserts the handler instructions after the user-instructions, thus retiring them in program order. The *prepend* scheme utilizes the properties of the reorder buffer and inserts the handler instructions before the user-instructions, thus retiring them out of program order without any side affects. We concluded that though both schemes are very similar and provide a degree of performance improvement, because the *prepend* scheme restores reorder buffer space, it provides better performance and also significantly reduces the amount of instructions being flushed.

Our simulations show that in-lined interrupt handling cuts the overhead by 10–40% [28], with a performance improvement of 5-25% for the *prepend* scheme and a 5-10% improvement for the *append* scheme. Additionally, our simulations revealed the overhead of handling TLB misses traditionally: 20-30% of the instructions being flushed will be re-executed again, and 80% of the instructions being flushed will need to be re-decoded. The overhead in terms of performance and power consumption is significantly high. The in-lined schemes reduce the number of instructions being flushed by 25-50% in the *append* scheme and 50-90% in the *prepend* scheme.

We are currently exploring the performance of in-lined handling of interrupts from the power consumption perspective. Flushing the pipeline, re-fetching and re-executing the instructions will definitely consume more power than not doing so. We are also looking into instruction-TLB interrupts, and the non-speculative in-lining of handler code. It is possible to begin fetching the handler into the ROB without first checking to see if there is enough room or resources. This requires a check for deadlock, and the system responds by handling a traditional interrupt when deadlock is detected—flush the pipe and resume at the handler. This allows support for variable-length TLB-miss handlers, such as the Alpha's.

References

[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] 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.

[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] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. “Design tradeoffs for software-managed TLBs.” In *Proc. 20th Annual International Symposium on Computer Architecture (ISCA’93)*, May 1993.

[17] 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.

[18] 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.

[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] C.B. Zilles, J.S. Emer, and G.S. Sohi, “Concurrent Event-Handling Through Multithreading”, *IEEE Transactions on Computers*, 48:9, September, 1999, pp 903-916.

[28] Jaleel, Aamer and Jacob, Bruce. “In-line Interrupt Handling for Software Managed TLBs”. *Proc. 2001 IEEE International Conference on Computer Design (ICCD 2001)*, Austin TX, September 2001.