In-Line Interrupt Handling for Software-Managed TLBs

Aamer Jaleel and Bruce Jacob

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

ABSTRACT

*The general-purpose precise interrupt mechanism, which has long been used to handle exceptional conditions that occur infrequently, is now being used increasingly often to handle conditions that are neither exceptional nor infrequent. One example is the use of interrupts to perform memory management—e.g., to handle translation lookaside buffer (TLB) misses in today's microprocessors. Because the frequency of TLB misses tends to increase with memory footprint, there is pressure on the precise interrupt mechanism to become more lightweight. When modern out-of-order processors handle interrupts precisely, they typically begin by flushing the pipeline. Doing so makes the CPU available to execute handler instructions, but it wastes potentially hundreds of cycles of execution time. However, if the handler code is small, it could potentially fit in the reorder buffer along with the user-level code already there. This essentially in-lines the interrupt-handler code. One good example of where this would be both possible and useful is in the TLB-miss handler in a software-managed TLB implementation. The benefits of doing so are two-fold: (1) the instructions that would otherwise be flushed from the pipe need not be re-fetched and re-executed; and (2) any instructions that are independent of the exceptional instruction can continue to execute in parallel with the handler code. In effect, doing so provides us with **lockup-free TLBs**. We simulate a lockup-free data-TLB facility on a processor model with a 4-way out-of-order core reminiscent of the Alpha 21264. We find that, by using lockup-free TLBs, one can get the performance of a fully associative TLB with a lockup-free TLB of one-fourth the size.*

1 INTRODUCTION

1.1 The Problem

Precise interrupts in modern processors are both frequent and expensive and are rapidly becoming even more so [17, 11, 18, 25]. 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 TLB misses in a software-managed TLB [16, 13, 12] and other memory-management related tasks [2]. If we look at TLB misses alone, we find that interrupts are common occurrences. For example, 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 increasing frequency, interrupts are also becoming more expensive; this is because of their implementation. Out-of-order cores typically handle precise interrupts much in the same vein as register-file update: i.e., at commit time [17, 21, 25, 15]. When an exception is detected, the fact is noted in the instruction's reorder buffer entry[1]. The exception is not usually handled immediately; rather, the processor waits until the instruction in question is about to commit before handling the exception, because doing so ensures that exceptions are handled in program order and that they are not handled speculatively [20]. If the instruction is already at the head of the ROB when the exception is detected, as in late memory traps [17], then the hardware can handle the exception immediately. Exception handling then proceeds through the following phases:

1. The ROB is flushed; the exceptional PC is saved; the PC is redirected to the appropriate handler
2. The exception is handled, typically with privileges enabled
3. If appropriate, control is returned to the application via a jump to the exceptional PC

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. The fact that there may be many cycles between the point that the exception is detected and the moment that the exception is acted upon is covered by (2): as the time it takes 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, and the current trend is towards increasingly large ROB sizes [8].

1.2 A Novel Solution

If we look at the two sources of performance loss (user code stalls during handler; many instructions are fetched and executed twice), 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.

1. Exceptional events wreak havoc in pipelined processors with out-of-order execution; one must ensure that the state of the processor (register file, caches, main memory) is modified in the sequential instruction order so that one can easily determine what has finished and what has not. Such support typically requires a reorder buffer (ROB) or a ROB-like structure [20, 21]. Following current conventions (e.g. [3, 4, 19, 6]), we will call all such structures "reorder buffers" even if this terminology is not always technically correct.

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].

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 that are already there, the processor core could avoid flushing the ROB and at the same time also 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 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. We call this scheme a *non-speculative in-line interrupt-handling* facility because the hardware knows the length of the handler *a priori*. Speculative in-lining is also possible, as discussed in our future work section.

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. In effect, this gives us *lockup-free TLBs*. 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 the mechanism 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.

The scheme cuts the TLB-miss overhead by 10–40%; the handler still must be executed, and the PTE load often causes a cache miss. When applications generate TLB misses frequently, this reduction in overhead amounts to a substantial performance savings. We model a lockup-free data-TLB facility; instruction TLBs do not benefit from the mechanism because, in most architectures, by the time an instruction-TLB miss is handled, the ROB is already empty. We find that lockup-free TLBs enable a system to reach the performance of a traditional fully associative TLB with a lockup-free TLB of roughly one-fourth the size.

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 [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

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

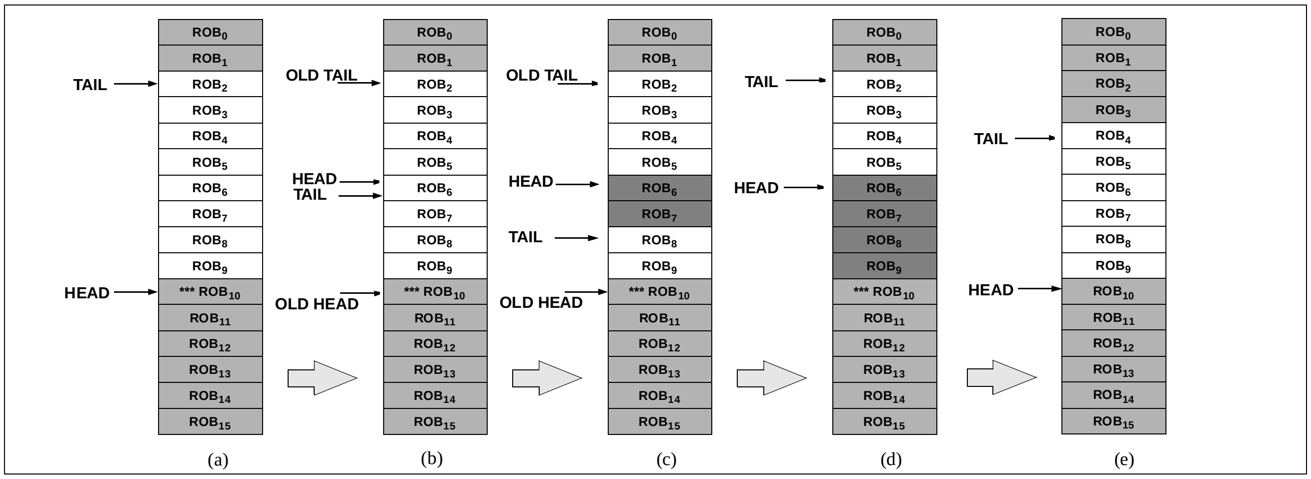

Figure 1: Non-Speculative In-lining of handler code. The figure illustrates the in-lining of a 4-instruction handler, assuming that the hardware fetches and enqueues two instructions at a time. The hardware stops fetching user-level instructions (light grey) and starts fetching handler instructions (dark 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**.**

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.

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 NON-SPECULATIVE IN-LINING OF HANDLERS

Figure 1 illustrates the non-speculative in-line interrupt-handling mechanism. To represent our scheme, 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.

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 six 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, saves the head and tail piont-ers into temporary registers and sets the head and tail pointers to four entries before the current head, as shown in state (b). The processor now begins fetching the first two handler instructions. These have been enqueued into the ROB at the tail pointer as usual, shown in state (c). In state (d) the last of the handler instructions have been enqueued, the old tail pointer is restored, the hardware then resumes fetching of user code as shown in state (e). Eventually when the the last handler instruction has finished execution and has reached the retire stage, the processor can reset the flag on the excepted instruction and retry the operation.

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

There are a few implementation issues concerning non-speculative in-lining of interrupt handlers. They include the following:

1. *The hardware knows the handler length*. The hardware must determine whether to handle an interrupt as usual (flush the reorder buffer) or to in-line the handler code. Therefore the hardware must have some idea how long the handler code is, or at least must have an upper limit on how long the code could be—for example, the hardware can assume that a handler is 16 instructions long, and a handler that is shorter than 16 instructions can fail to be in-lined occasionally, even though there was enough room for it in the ROB.

2. *There should be a privilege bit per ROB entry.* As mentioned earlier, handler in-lining allows the coexistence of user and kernel instructions in the pipeline, each operating at a different privilege level. The most elegant way to allow this without creating security holes is to attach a privilege bit 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*. 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

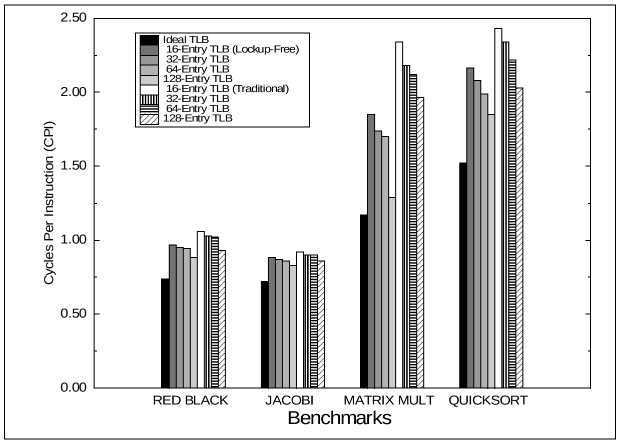

Figure 2: Performance of benchmarks. The figure shows the execution time (cycles-per-user-instruction) of a perfect TLB, 16/32/64/128-entry lockup-free TLBs, and 16/32/64/128-entry software-managed TLBs.

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 initializing a large array walks into a new page of data that is not currently mapped in the TLB: every 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.

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 an interrupt handler, the "return from interrupt" instruction is usually executed which jumps to the exceptional PC, and disables privileges. However, the processor must NOP this return from interrupt instruction, and resume fetching at some completely unrelated location in the instruction stream at some distance from the exceptional instruction. Therefore, we require additional logic to ignore the exceptional PC and instead store the PC of the next-to-fetch instruction at the time of in-lining the handler code. The logic amounts to a MUX.

5. *The hardware might need to know the handler's register requirements*. If at the time the TLB miss is discovered, there are user instructions waiting to be decoded and mapped to physical registers, a deadlock situtation might occur if there aren't enough free physical registers available to map the user instructions. To prevent this, the processor can do one of two things: (a) handle the interrupt by the traditional method, or (b) flush all instructions in the fetch and decode stage and set ***nextPC*** (described above) 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 rovide such registers, the hardware will need to ensure adequate physical register availablity before vectoring to the handler code. For our simulations, we only simulated scheme (a).

6. *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 modify the state of those registers mapped to user instructions. In the conventional method of register renaming, an instruction receives the register mapping of the previous instruction in the pipe. If this scheme is used to map the handler instructions, then when they commit, they will 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.

7. *Branch mispredictions in user code should be handled appropriately.* If, while in INLINE mode, a user-level branch instruction is found to have been mispredicted, the hardware should overwrite ***nextPC*** (described above) with the correct branch target. The handler instructions within the reorder buffer are unaffected by the misprediction and won't be flushed (even though they came in logically after the branch).

The hardware design is 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 LOCKUP-FREE TLBs

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 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 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 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.

We model only the data-TLB as lockup-free. Most architectures (including the 21264) handle I-TLB misses by pushing NOPs into the ROB, and once the first NOP is at the head of the ROB, the TLB miss is handled. This would receive no benefit from in-lining the handler, because the ROB is already empty at this point.

4.2 Results

Our simulations show that the reorder buffer is often only 50% full, and, when TLB misses occur, there is usually enough room to in-line the interrupt handler. For example, in the range of TLB

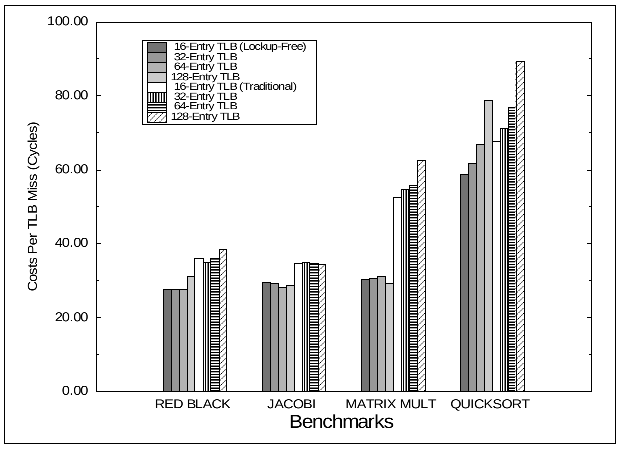

Figure 3: Cost per TLB miss. This figure illustrates how many cycles the application spent in TLB misses, on a per-miss basis.

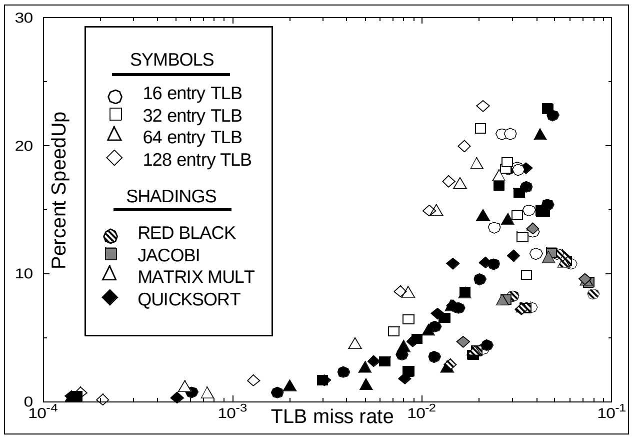

Figure 4: Speedup as a function of working-set size. Working-set size is represented by TLB miss rate: the number of data-TLB misses per access.

sizes modeled, roughly 90% of the interrupts in quicksort can benefit from in-lining, and 80% of the interrupts in matrix multiplication benefit. We find that, out of the instances when the handler cannot be in-lined, it is usually due to insufficient physical registers available to map the instructions in the pipeline.

Figure 2 compares the performance of perfect TLBs, traditional software-managed TLBs, and lockup-free TLBs. It shows the lockup-free scheme reducing execution time by 5–25% for the same-size TLB; alternatively, the lockup-free scheme achieves the same performance as a traditional software-managed TLB with a TLB one-half to one-eighth the size.

Figure 3 shows the cost per TLB miss for each benchmark. By in-lining the interrupt handlers, one can reduce the cost of a TLB miss by 10–40%. This difference represents the cost of flushing the ROB and then re-fetching and re-executing those instructions flushed. As expected, this overhead of flushing the pipeline is not a constant value, but instead it depends on the contents of the ROB at the time of handling the exception. This is shown in the fact that there is such a wide difference in the amount of reduction seen in the per-miss costs.

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 using in-line interrupt handling. In addition to running the benchmarks "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 Figure 4, which presents a scatter plot of TLB miss rate to application speedup.

The figure shows a clear correlation between the two: the more often that the TLB requires management, the more benefit one sees from handling the interrupt in-line. 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.

5 CONCLUSIONS & FUTURE WORK

The general-purpose interrupt mechanism, which has long been used to handle exceptional conditions that occur infrequently, is being used increasingly often to handle conditions that are neither exceptional nor infrequent. 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, which enables such mechanisms as lockup-free TLBs. In such an implementation, the reorder buffer is not flushed on an interrupt unless there are so many instructions in the buffer that the handler instructions would not fit. 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).

Our simulations show that lockup-free TLBs cut TLB-miss handling overhead by 10–40%, and a system with 32-entry lockup-free TLBs has the same performance as a system with regular 128-entry TLBs. This allows software-managed TLBs to reach the same performance as hardware-managed TLBs, or to reduce the TLB size (and thus energy consumption) at the same performance level.

We are currently exploring two additional avenues: first is the counterpart to 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. Second is a speculative initiation of the handler before the exceptional instruction reaches the head of the ROB, which can offer higher performance in cases when the exceptional instruction is not discarded as a result of a mispredicted branch or preceding exception. This could also be used to implement lockup-free instruction TLBs.

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] 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] 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.