In-Line Interrupt Handling and Lock-Up Free Translation Lookaside Buffers (TLBs)

Aamer Jaleel, *Student Member*, *IEEE*, and Bruce Jacob, *Member*, *IEEE*

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, 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 refetching and reexecuting the instructions flushed. This paper concentrates on improving the performance of precisely handling software managed translation look-aside buffer (TLB) interrupts, one of the most frequently occurring interrupts. The 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 refetched and reexecuted. 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 paper 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 performance model of the Alpha 21264 processor built by Alpha designers at the Palo Alto Design Center (PADC), California. 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 refetching and reexecuting the instructions flushed by 30-95 percent, reduce the execution time by 5-25 percent, and also reduce the energy wasted by 30-90 percent.

Index Terms—Reorder-buffer (ROB), precise interrupts, exception handlers, in-line interrupt, lock-up free, translation lookaside buffers (TLBs), performance modeling.

1 INTRODUCTION

1.1 The Problem

PRECISE interrupts in modern processors are both frequent and expensive and are rapidly becoming even more so [20], [9], [21], [27]. One reason for their rising frequency is that the general purpose 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 integer or floating-point arithmetic overflow, misaligned memory accesses, memory protection violations, page faults or TLB misses in a softwaremanaged TLB [15], [10], [11], and other memory-management related tasks [20]. 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 [8] show that TLB miss handling can account for more than 40 percent of total runtime, and Rosenblum et al. [21] show that TLB miss handling can account for more than 80 percent of the kernel's computation time.

Recent studies have shown that a precise interrupt occurs once every 100-1,000 user instructions on all ranges of code, from SPEC to databases and engineering workloads [21], [2]. 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: for instance, at commit time [20], [23], [27], [19]. When an exception is detected, the fact is noted in the instruction's reorder buffer 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 because doing so ensures that exceptions are handled in program order and that they are not handled speculatively [22]. If the instruction is already at the head of the ROB when the exception is detected, as in late memory traps [20], then the hardware can handle the exception immediately. Exception handling then 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.

In this model, there are two primary sources of application-level performance loss: 1) While the exception is being

• *The authors are with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742. E-mail: {ajaleel, blj}@eng.umd.edu.*

Manuscript received 15 Jan. 2004; revised 13 Dec. 2004; accepted 15 Sept. 2005; published online 22 Mar. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0018-0104.

0018-9340/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

handled, there is no user code in the pipeline 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 refetched and reexecuted, duplicating work that has already been done. The fact that there may be many cycles between the point where the exception is detected and the moment when 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 refetched and reexecuted [20]. 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 toward increasingly large ROB sizes [6]. With an 80-entry ROB, like the Alpha 21264, as much as a whole window of instructions can be flushed at the time of an exception. This can result in a significant amount of energy wasted in the refetching and reexecuting of instructions flushed. Given that we have broken the triple digit Wattage ratings for modern microprocessors, it is thus imperative that we improve the traditional method of handling interrupts.

1.2 A Novel Solution

If we look at the two sources of performance loss (user code stalls during handler; many user instructions are refetched and reexecuted), 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 and Vassiliadis briefly discuss its overhead and portability problems [19]. Such a mechanism would be extremely expensive in modern out-of-order cores, however; Walker and 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 [27]. Zilles et al. discuss a multithreaded approach, where, at the time an exception is detected, the processor spawns a new thread to fetch and execute the interrupt handler [30]. The scheme, however, requires the processor architecture to allow multiple threads executing in parallel.

We are interested instead in using existing out-of-order hardware, on uniprocessor with a single thread of execution, 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. Is it to prevent privileged operating system instructions (interrupt handler) to coexist with user level instructions already present in the pipeline? However, Henry has discussed an elegant method to allow privileged and nonprivileged instructions to coexist in a pipeline [7]; 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 is enough space 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 an ROB [4], [29] (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 *nonspeculative 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 such a mechanism is generally applicable to all types of software managed interrupts with relatively short interrupt handlers, we focus only 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 and account for more than 40 percent of total runtime [1], [8].
2. The first-level TLB-miss handlers tend to be short (on the order of 10 instructions) [10], [20].
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 nonblocking for some time: For example, a TLB-miss in the Pentium-III pipeline does not stall the pipeline—only the exceptional instruction and its dependents stall [2]. 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 nonblocking hardware-managed TLBs without sacrificing flexibility [27].

1.3 Results

We evaluated two separate lock-up free mechanisms (*append* and *prepend* schemes) on a performance model of the Alpha 21264 architecture (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 number of user instructions flushed due to TLB misses by 30-95 percent; 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 savings in performance. 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. In general, we observe performance improvements of 5-25 percent and a reduction in the amount of energy wasted by 30-90 percent.

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], [23]. 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 [24] gives a hardware architecture for resolving interinstruction dependencies that occur through the register file, thereby allowing out-of-order issue to the functional units; Smith and Pleszkun [22] 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 and Vajapeyam [23] 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 mispredicts).

2.2 The Persistence of Software-Managed TLBs

It has been known for quite some time that hardware-managed TLBs outperform software-managed TLBs [10], [25]. Nonetheless, 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 of the increased flexibility inherent in the software-managed design [14], the ability to deal with larger virtual address spaces, and because redesigning system software for a new architecture is nontrivial. 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 and Day discuss an imprecise-interrupt mechanism appropriate for handling interrupts that are transparent to application program semantics [25]. 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 and Dubois recently presented a mechanism for handling memory traps that occur late in the instruction lifetime [20]. 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 and Cragon [27] and Moudgill and Vassiliadis [19] present surveys of the area; both discuss alternatives for implementation of precise interrupts. Walker and Cragon describe a taxonomy of possibilities and Moudgill and Vassiliadis looks at a number of imprecise mechanisms.

Keckler et al. [17] present an alternative architecture for SMT processors, concurrent event handling, which incorporates multithreading into event handling architectures. Instead of handling the event in the faulting thread's architectural and pipeline registers, the exception handler is forked as a new thread that executes concurrently with the faulting thread. Zilles et al. in [30] utilized the concurrent approach to handle TLB miss exceptions on SMT processors. Though the concurrent event handling scheme mirrors our proposed in-line scheme, the processor must be required to support multiple threads of execution. Our approach, however, does not require multiple threads, a single thread of execution is sufficient.

3 In-Line Interrupt Handling

We present two separate schemes of in-lining the interrupt handler within the reorder buffer. Both our schemes use the properties of an ROB: 1) Queue new instructions at the tail and 2) retire old instructions from the head [22]. If there is enough room between the head and tail pointer of the ROB, we in-line the interrupt by either inserting the interrupt handler code before 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 illustrate our schemes, 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,

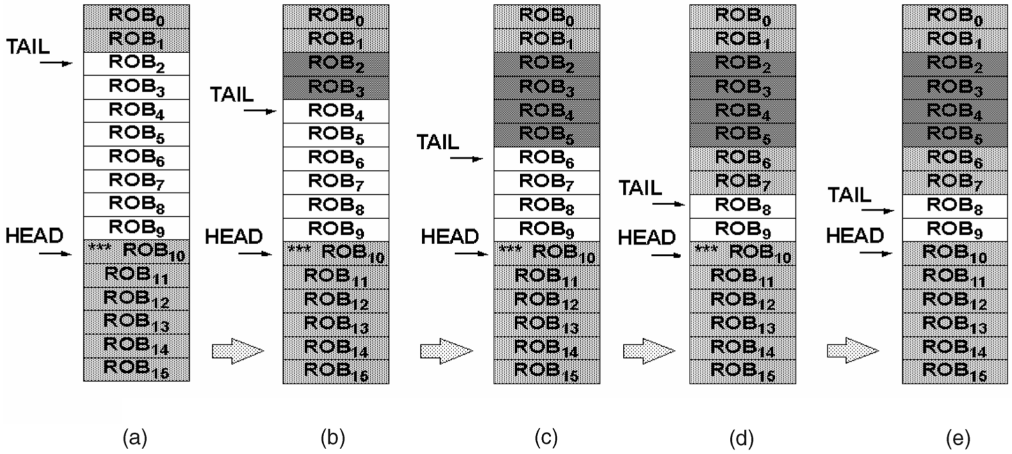


Fig. 1. Append In-line scheme: In-line the interrupt handler by fetching the instructions at the tail of the reorder buffer. The figure shows the in-lining of a 4-instruction handler, assuming that the hardware fetches and enqueues two instructions at a time. The hardware stops fetching user-level instructions (light gray) and starts fetching handler instructions (dark gray) 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.

we assume all instruction state is held in the ROB entry, as opposed to being spread out across ROB and reservation-station entries.

We now provide a detailed description and implementation of the two in-lining schemes.

3.1 Append In-Line Scheme

Fig. 1 illustrates the append scheme of in-lining the interrupt handler. In the first state (Fig. 1a), 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 can fit into the available space—in this case, there are eight empty slots in the ROB. Since we are assuming our handler is four instructions long, the handler will fit in the available ROB 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 Fig. 1b. In Fig. 1c, the last of the handler instructions have been enqueued, the hardware then resumes fetching of user code, as shown in Fig. 1d. 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, Fig. 1e.

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 executes rather than wait for the instruction to commit. 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 one drawback: The interrupt handler occupies 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 the handler's job is done. This will 1) provide more available space for user instructions to be fetched and executed and 2) in the event of future exceptions, precious ROB space is not occupied by already executed handler instructions. To avoid these drawbacks, we now describe the *prepend* scheme of in-lining interrupts.

3.2 Prepend In-Line Scheme

Fig. 2 illustrates the prepend scheme of in-lining the interrupt handler. In the first state, Fig. 2a, 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 moves the head and tail pointer to four instructions before the current head, shown in Fig. 2b. The old tail pointer needs to be saved and the tail pointer changed because new instructions are always queued at the tail of a reorder buffer and instructions are committed from the head. Once the head and tail pointers are changed, 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 pipeline. They are enqueued at the tail of the reorder buffer as usual, shown in Fig. 2c. The hardware finishes fetching the handler code (Fig. 2d) 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 finishes execution, the flag of the excepted instruction can be removed and the exceptional instruction may retry the operation (Fig. 2e). 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),

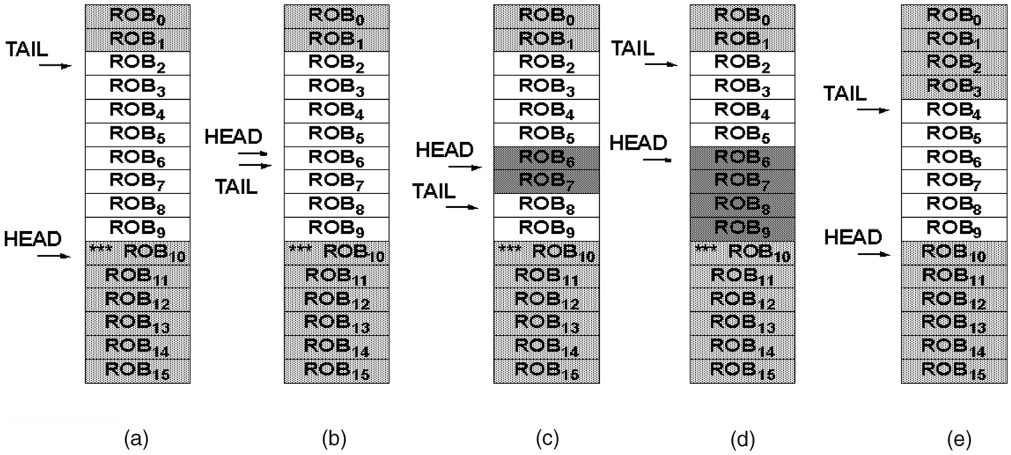


Fig. 2. Prepend In-line scheme: In-line the interrupt handler by resetting the head and tail pointers. The figure shows the in-lining of a 4-instruction handler, assuming that the hardware fetches and enqueues two instructions at a time. Once the exceptional instruction, identified by asterisks, reaches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

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, with the prepend scheme the handler instructions are brought physically "ahead" of all user instructions. Since the handler instructions are at the head of the reorder buffer and can retire and update state, no modifications to the pipeline were required. For example, the *TLB write* instruction is no longer required to update the TLB when it has executed, rather the TLB is updated when committing the handler instructions. Since handler instructions are retired from the head of the ROB, 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 in-lining of interrupts.

3.3 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 nospeculative 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 pipeline and ROB) 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 [7]. Note that this mechanism is already implemented in existing processors (e.g., Alpha) and each structure checks whether the operator is allowed for a specific operation before it is actually executed.
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 an 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 reaccess the TLB and 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 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 an MUX.
6. *The processor needs to make sure it is not already stalled*. If, at the time the TLB miss is discovered, the processor will need to make sure it is not stalled in one of the critical paths of the pipeline, e.g., register renaming. A deadlock situation might occur if there are not 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: 1) Handle the interrupt via the traditional method, or 2) flush all instructions in the fetch, decode, and map 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 provide such registers, the hardware will need to ensure adequate physical register availability before vectoring to the handler code. For our simulations, we implemented scheme 1).
7. *Branch mispredictions in user code should not flush handler instructions*. If, while in append INLINE mode, a user-level branch instruction is found to have been mispredicted there are one of two possibilities. If the interrupt handler has finished execution, the branch mispredict can be handled in the traditional manner (i.e., pipeline flush and fetch user instructions from the new target PC). However, if the interrupt handler has not yet finished execution, the resulting pipeline flush should not flush the handler instructions. 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. Note that this applies only to the append scheme and not the prepend scheme, of handling interrupts. In the prepend scheme, the handler instructions within the reorder buffer are unaffected by the branch mispredict and will not be flushed (even though they came in logically after the branch).

In addition, the in-lined schemes interaction with the register-renaming mechanism is nontrivial. 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 [23], [24] 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.

A possible 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 [29], 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, a possible 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 schemes 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.

3.4 Append Scheme versus Prepend Scheme

Though the append scheme and prepend scheme are very similar, the implementation differences between the two schemes may show different performance behaviors. The schemes differ in terms of the location where the interrupt handler is in-lined. With the append scheme, the interrupt handler is embedded in 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, replay traps, etc.) need to be tolerated by the append scheme as the handler instructions are embedded within user code. Even more, the mere fact that the handler instructions can update state (e.g., TLB state) a few cycles earlier in the append scheme than the prepend scheme can also allow for minor differences in terms of performance. We now analyze the performance behavior of both these schemes of interrupt in-lining.

4 Experimental Methodology

4.1 Simulator

We use the performance model of the Alpha 21264 (EV67) that was developed by the Alpha designers at the Palo Alto Design Center (PADC) in Palo Alto, California. We use 64KB 2-way L1 instruction and data caches, fully associative 16/32/64/128-entry separate instruction, and data TLBs with an 8KB page size. The model issues four integer instructions and two floating-point instructions per cycle and holds a maximum of 80 instructions in-flight. The simulator also models a 72-entry register file (32 each for integer and floating-point instructions and eight for privileged handler instructions), four integer functional units, and two floating-point units. The model also provides 154 renaming-registers, 41 reserved for integer instructions and 41 for floating-point instructions.

The performance model also provides a 21 instruction TLB miss handler. The model does not reserve any renaming registers for privileged handler instructions as they are a subset of integer instructions. Therefore, to use the in-lined schemes, the hardware must know the handler's register needs as well as the handler's length. Additionally, the 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 also be sufficient for the handler to be in-lined. The page table and TLB-miss handler are modeled after the MIPS architecture [16], [10] 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 [18], McCalpin presented, among other things, the graph shown in Fig. 3, 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 (three data sets, two applications),
- Nonlinear Implicit Finite Element Analysis (eight data sets, three applications),
- Nonlinear Explicit Finite Element Analysis (three data sets, three applications),
- Finite Element Modal (Eigenvalue) Analysis (six data sets, three applications),
- Computational Fluid Dynamics (13 data sets, six applications),
- Computational Chemistry (seven data sets, two applications),
- Weather/Climate Modeling (three data sets, two applications),
- Linear Programming (two data sets, two applications),
- Petroleum Reservoir Modeling (three data sets, two 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 a sequential fashion [18].

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 Fig. 3). The benchmark applications include QUICKSORT, MATMULT, REDBLACK, and JACOBI. QUICKSORT sorts a 1,024-array using the quick-sort sorting algorithm, MATMULT multiplies two 100×100 matrices, REDBLACK performs a 3D red-black

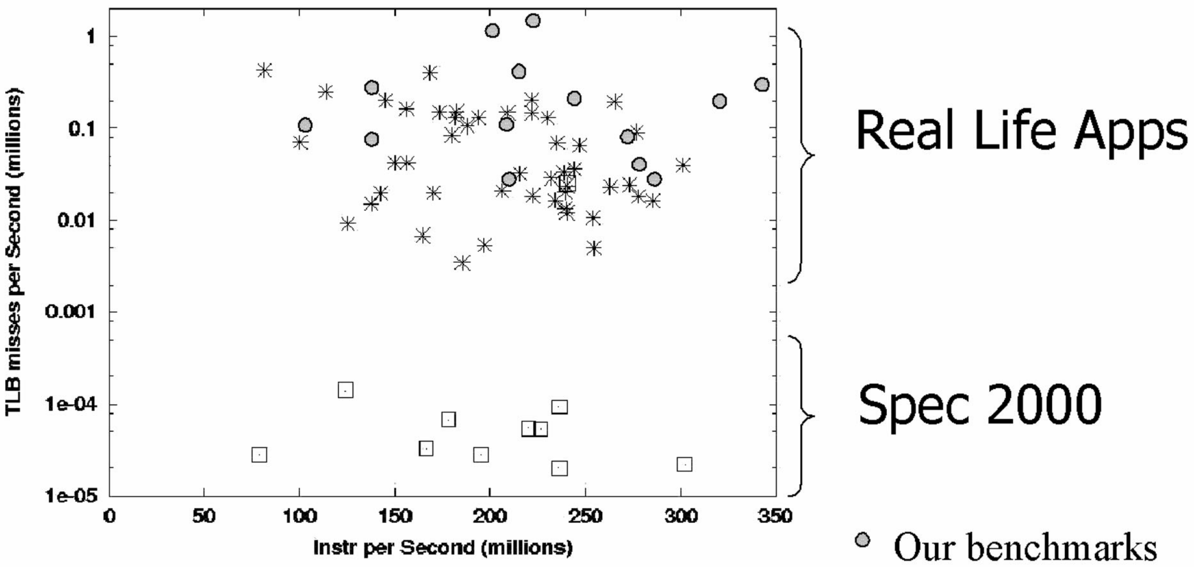


Fig. 3. TLB behavior of the SPEC 2000 suite (source McCalpin). McCalpin, in this graph, compares the TLB behavior of SPEC 2000 to a set of "apps" that, according to him, truly represent high-performance computing. The TLB behavior of our benchmarks, shown in circles, emulates the behavior of the real-life applications.

successive-over-relaxation using two 100 × 100 matrices, and JACOBI performs a 3D Jacobi relaxation using two 100 × 100 matrices. Both JACOBI and REDBLACK are frequently found in multigrid PDE solvers, such as MGRID and the SPEC/NAS benchmark suite. The benchmarks were compiled on an Alpha 21264 processor using the cc compiler with optimization -O3. Each of the benchmarks is run using different memory footprints: small, medium, and large. Since the benchmarks are array-based, a small memory footprint is where the array element size is 512 bytes (on average, 10,000 TLB misses with a 16-entry TLB), a medium memory footprint is where the array element is three kilobytes (on average, 60,000 TLB misses with a 16-entry TLB), and a large memory footprint is where the array element is five kilobytes (on average 80,000 TLB misses with a 16-entry TLB). 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 McCalpin's benchmark suite.

5 Performance of Lock-Up Free TLBs

5.1 Limitations of In-Lining TLB Interrupts

We first investigate how often our applications benefit from the in-line scheme of handling TLB interrupts. Fig. 4 illustrates the limitations of the lock-up free TLB scheme for the different benchmarks. The x-axis represents the different memory footprint sizes, and the y-axis represents the percent of time where the processor was: successful in-lining TLB interrupts (black), unsuccessful in-lining TLB interrupts due to insufficient ROB space (light gray), and unsuccessful in-lining TLB interrupts due to insufficient renaming registers (stripes). In the graphs, the first four bars represent 16/32/64/128 entry TLBs managed by the append in-lined scheme and the last four bars are for those TLBs managed by the prepend in-lined scheme.

From the figure, our studies show that, besides the need for available reorder buffer space, the processor also needs to ensure the availability of free renaming registers to map instructions that are newly fetched into the pipeline. This is to be expected as the simulated processor reserves separate renaming registers for integer and floating-point instructions (41 each) and, with an 80-entry reorder buffer, it is possible to have more than 41 instructions of just integer or just floating-point data-type to be in-flight. From the graphs, we observe that in-lining of the interrupt handler is successful for 25-90 percent of all TLB interrupts. We observe that the lack of reorder buffer space is a nonissue when in-lining TLB interrupts. The primary reason that a TLB interrupt is unable to be in-lined is because the processor is already stalled due to insufficient renaming registers available to map existing user instructions. Hence, to realize the full potential of interrupt in-lining, we observe two possible modifications to existing architectures as part of our future work: 1) reserve separate rename registers for privileged instructions and 2) be able to partially flush the pipeline (i.e., just the fetch and map/decode stage) without affecting the rest of pipeline.

5.2 Interrupt Overhead—Instructions Flushed

One of the primary overheads associated with the traditional method of handling interrupts is the wastage of time and energy in redoing work that has already been done before, i.e., refetch and reexecute those instructions that were flushed. Fig. 5 illustrates the average number of user instructions flushed when a D-TLB miss is detected. The x-axis represents the different memory footprint sizes and 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, with the traditional scheme of handling interrupts (last four bars), at the time a D-TLB miss

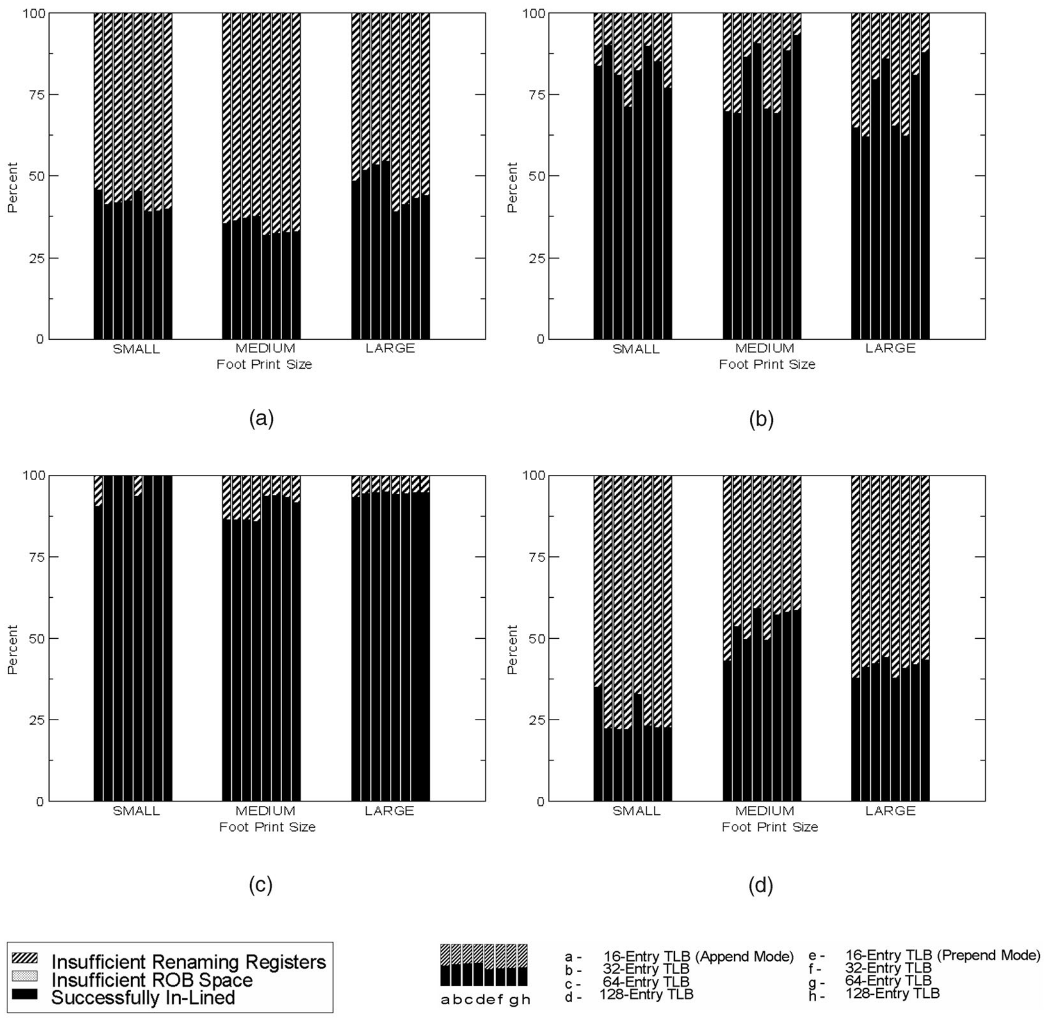


Fig. 4. Limitations of interrupt in-lining. This figure shows the number of times interrupt in-lining was used and the reasons why interrupt in-lining could not be used. The figure shows that space is not an issue; instead, the pipeline being stalled due to the lack of renaming registers is the primary reason for not in-lining.

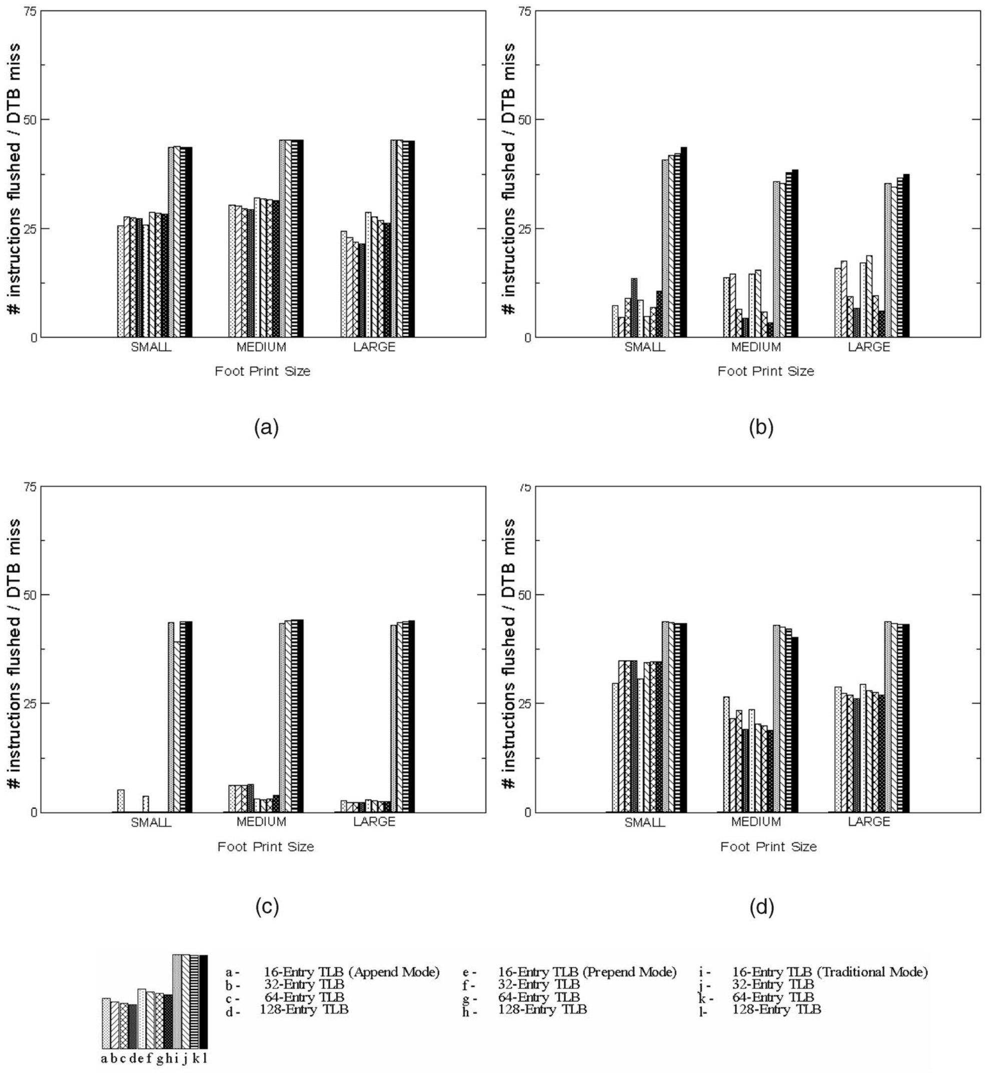
is detected, on average, an 80-entry ROB is about 50-55 percent full. This observation is promising in that relatively large interrupt handlers can be in-lined provided there are enough resources. From the figure, we also observe that the in-lined schemes of handling TLB interrupts reduce the number of instructions flushed per TLB interrupt by 30-95 percent. This implies that a processor can possibly save energy that is needlessly spent in refetching and reexecuting those user instructions that were flushed.

From Fig. 5, we also observe that the append and prepend schemes have a varying behavior in terms of the number of user instructions flushed. We observe that, in some cases, the append scheme flushes fewer instructions than the prepend scheme. This is because, for one benchmark (JACOBI), the append scheme benefited from in-lining more than the prepend scheme (see Fig. 4). However, for the cases where the append and prepend scheme benefited from in-lining equally or where the prepend scheme benefited from in-lining more than the append scheme—it is due to the implementation of the in-line 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 executed, thus making room for more user instructions to reside within the reorder buffer.

An interesting observation from the graphs is that the number of user instructions flushed per TLB-miss is independent of the TLB size. It would seem that increasing the TLB size should reduce the average number of instructions flushed per TLB interrupt; however, this is not the case: The number of instructions flushed can either increase or decrease. This is because the number of instructions flushed, on a TLB miss, is dependent on the contents of the ROB and this is independent of the TLB size.

5.3 Performance of Lock-Up Free TLBs

The benefits of lock-up-free TLBS are two-fold: 1) User instructions execute in parallel while the interrupt handler fills the missing information in the TLB and 2) the overhead of refetching and reexecuting user instructions flushed is eliminated. We now compare the performance of an ideal TLB, the lock-up-free schemes, and the traditional method of handling interrupts (Fig. 6). The x-axis represents the different memory footprint sizes and the y-axis represents



(a)

(b)

(c)

(d)

Fig. 5. Average number of instructions flushed per D-TLB 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 percent full (40-45 user instructions flushed). This is promising in that one does not have to restrict oneself to small handlers. Additionally, we see that in-lining significantly cuts the number of instructions flushed by 30-95 percent.

the CPI. In each graph, the first bar represents an ideal TLB, the next 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, finally, the last four bars are for those TLBs managed by the traditional scheme of handling interrupts (i.e., flush the ROB and pipeline). From Fig. 6, we observe that virtual memory adds significant overhead in the system. For medium and large memory footprints we observe a 25-50 percent performance degradation from the optimal case-a perfect TLB. This is to be expected when TLB misses are frequent and would be the case with realistic applications [18]. Hence, it is clearly important to optimize TLB management so as to reduce the overhead.

The figures show the performance benefit of using lock-up-free TLBs: For the same-size TLB, execution time is reduced by 5-30 percent. 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. This observation shows 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 [14], even if that microprocessor is specifically a low-power design. Therefore, any reduction in the TLB size is welcome if it comes with no performance degradation.

Both the in-line schemes aid in reducing the overheads in terms of the number of instructions flushed and the total execution time. However, performance results from Fig. 6 show an unexpected behavior—the append scheme performs better than the prepend scheme by 2-5 percent. Based on the implementation of the append scheme, it was expected that the prepend scheme would outperform the append scheme because: 1) The append scheme occupies valuable space within the reorder buffer and 2) handler instructions have lower priority in terms of scheduling to functional units. Since we observed from Fig. 4 that space is clearly not an issue for interrupt in-lining, the small differences in performance can thus be correlated to the

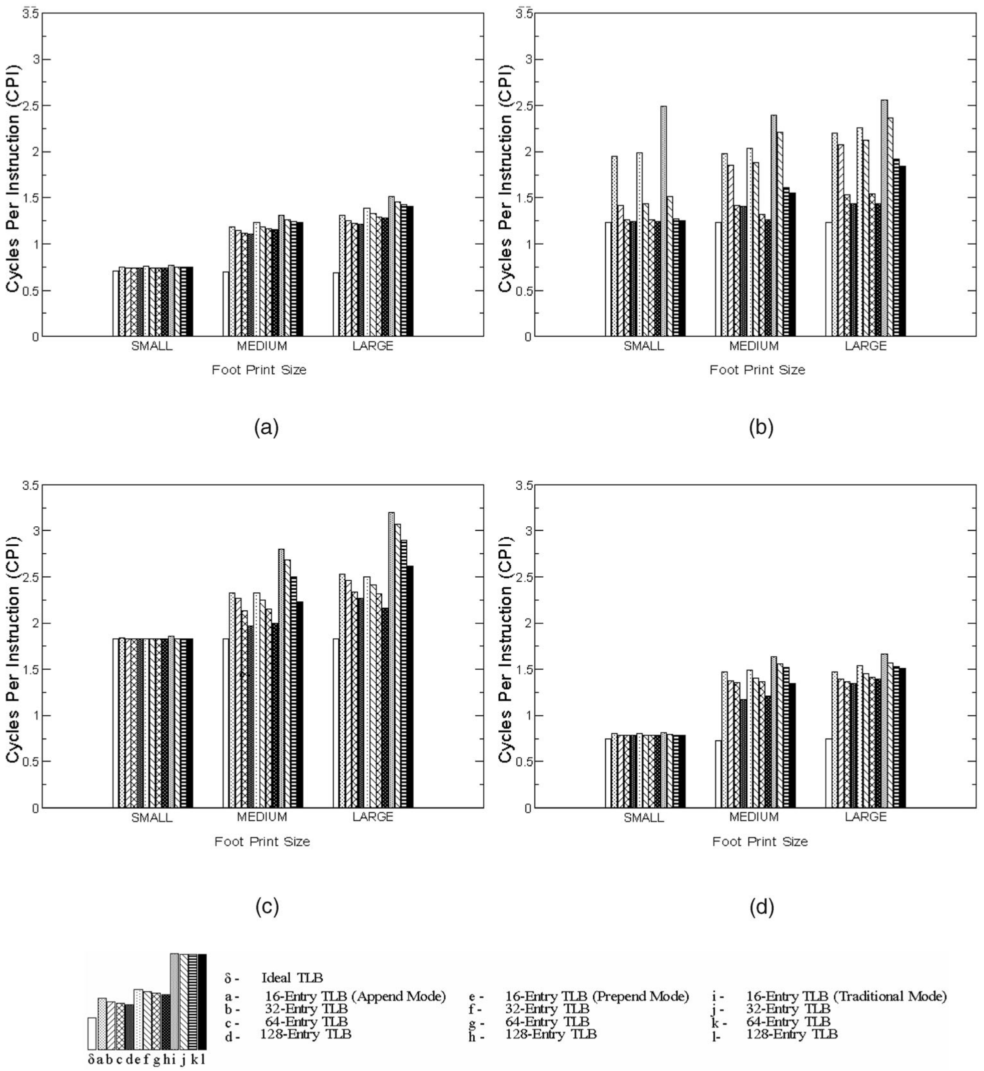


Fig. 6. Performance of interrupt in-lining. This figure compares the performance of the benchmarks for an ideal TLB, append scheme, prepend scheme, and traditional TLBs. Both the schemes of in-lining improve performance in terms of CPI by 5-25 percent. (a) Jacobi, (b) Matmult, (c) Quicksort, and (d) Redblack.

commit logic modification needed to implement the append scheme—we required the exception to be "handled" transparently without having to commit the handler instructions. Thus, 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 actually executed. This was required to prevent the processor from entering a dead lock situation as instructions are held up at the head of the reorder buffer by the exceptional instruction. With this modification, instructions that missed in the TLB are awakened and executed a few cycles earlier than the prepend scheme. Thus, due to this modification, we observe that the append scheme outperforms the prepend scheme by 1-5 percent.

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 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 Fig. 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 the TLB requires management, the more benefit one sees from lock-up free TLBs. 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.

5.4 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. Fig. 8 presents important results. Most noticeably, the absolute number of instructions flushed is very large: The y-axis in the figures indicates 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 create a great

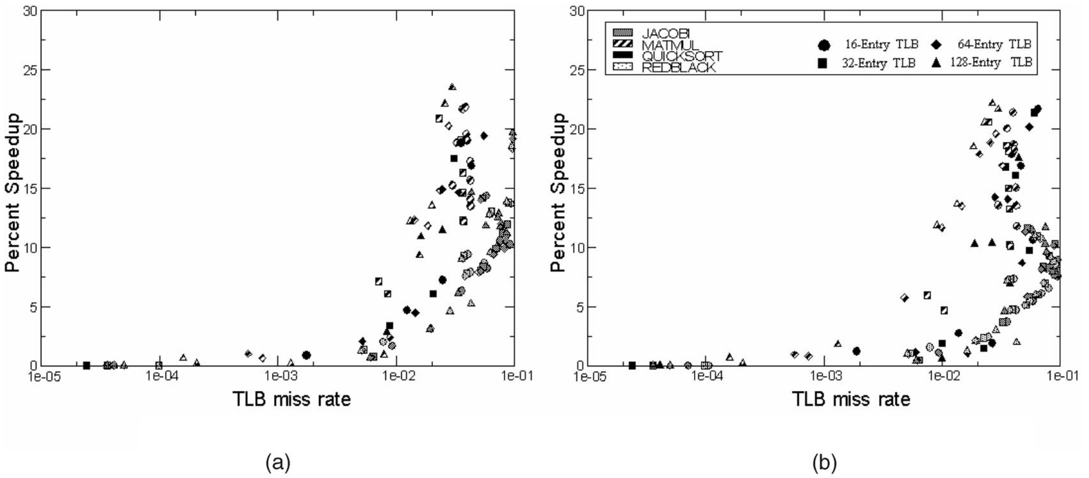


Fig. 7. TLB miss rate versus performance improvement (append, prepend). This figure shows that performance improvement is dependent not on TLB size but TLB miss rate. The figure shows that the more often the TLB requires management, the more the benefit the application sees from interrupt in-lining. (a) Append in-line scheme. (b) Prepend in-line scheme.

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 percent or more.

Fig. 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. We observe that more than 50 percent of the instructions flushed have been decoded, renamed (mapped), and enqueued into the reorder buffer and issue queues by the time they are flushed (labeled to be issued). About 10-15 percent 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, designers of 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 budget [3], [28]. The following are the power breakdowns (with respect to the entire chip) for the 21264, with the IBOX detailed:

- IBOX: Integer and Floating-Point Map Logic, Issue Queues, and data path: 32 percent, Issue Unit: 50 percent, Map Unit: 23 percent, Fetch Unit: 23 percent, Retire Unit: 4 percent,
- MBOX: Memory Controller: 20 percent,
- EBOX: Integer Units (L, R) 16.5 percent,
- DBOX: Dcache 13 percent,
- CBOX: BIU Data & Control Busses: 12 percent,
- JBOX: Icache 6.5 percent.

Using these breakdowns, we computed the power breakdowns (with respect to the entire chip) for the 21264 by pipeline stage:

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

With the power breakdowns by pipeline stage, we now quantify the energy-consumption benefits by using lock-up-free TLB schemes. Fig. 9 shows trends in energy savings that are very similar to the performance benefits (they differ by about 5-10 percent). An immediately obvious feature of the data in the Fig. 9 is the huge amount of energy spent on partially executed instructions that are flushed before they can retire. This is a significant result by itself. Today, 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. 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 percent overall and also reduce the energy wasted in refetching and reexecuting by 30-90 percent, 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 increase. Thus, an increase in the 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 refetched and

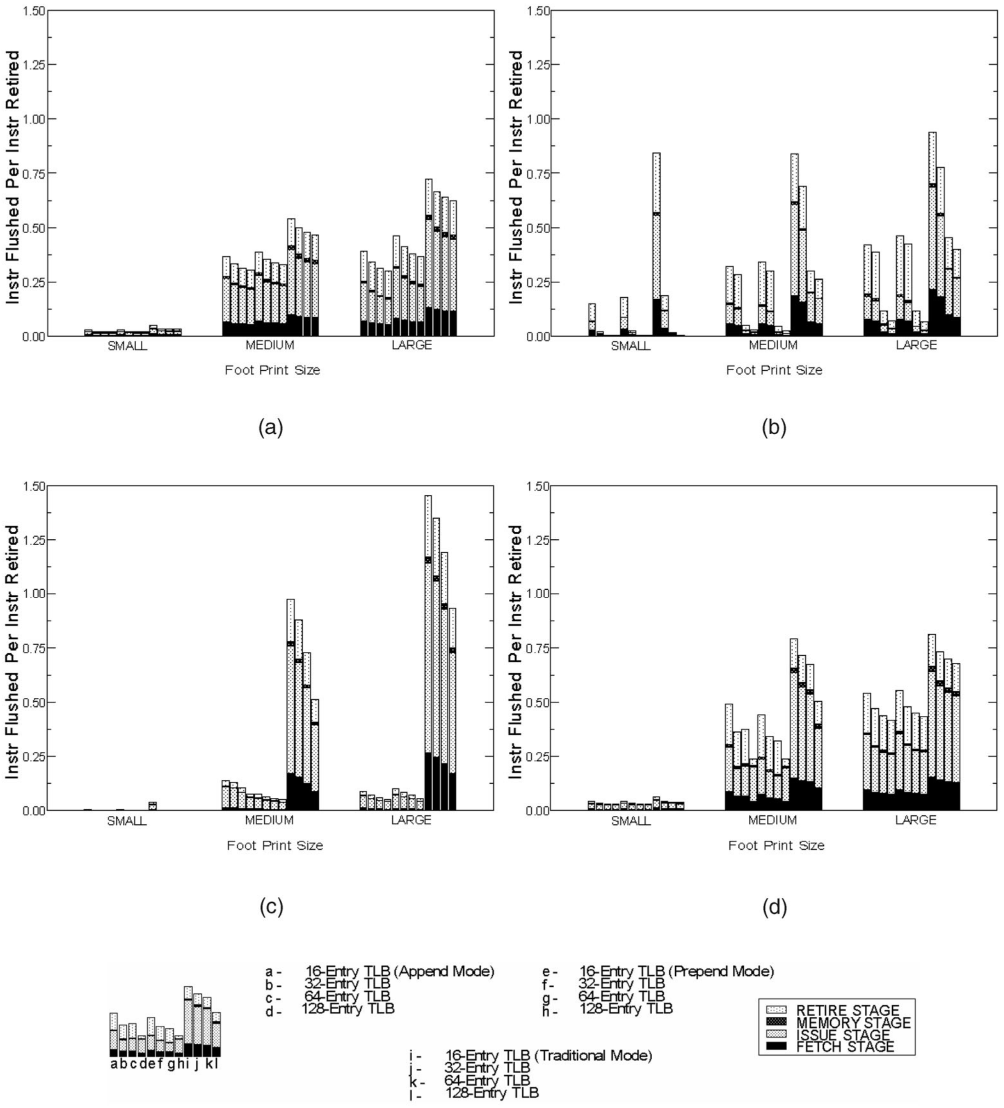


Fig. 8. Location of instructions flushed due to a TLB miss. This figure shows the stages in which the instructions were before they were flushed. The y-axis shows the number of instructions flushed for each instruction retired. In-lining significantly reduces the number of instructions flushed. Additionally, the graph shows that the majority of the instructions flushed are relatively early in the pipeline with 10-15 percent in the final stages. (a) Jacobi, (b) Matmult, (c) Quicksort, and (d) Redblack.

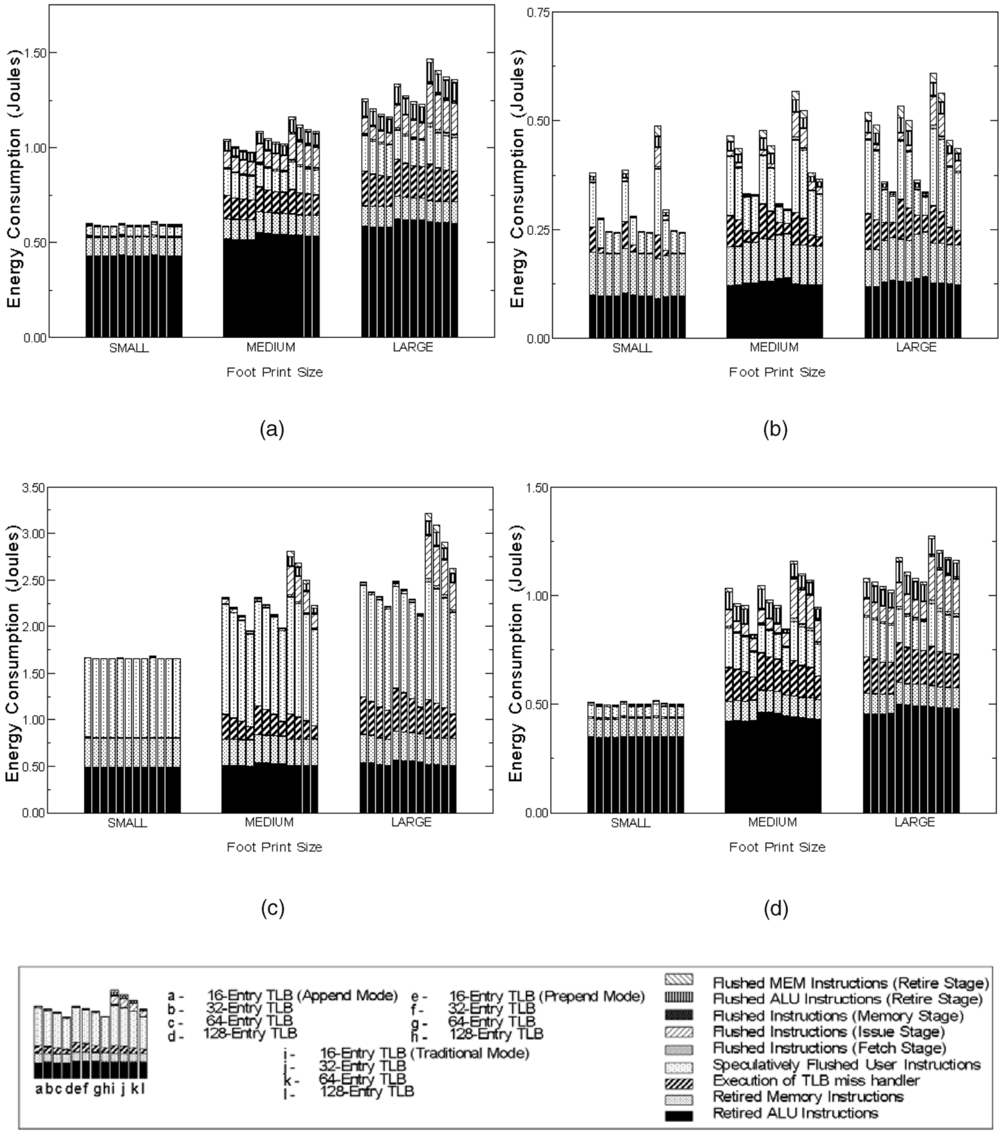
reexecuted. The average joules-per-instruction for these redundant operations is slightly lower than the average joules-per-instruction for the entire run because many are only partially executed and, therefore, contribute less than an average instruction to the total. Therefore, eliminating these instructions 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 exceptions that require a pipeline flush. The graphs show that about 5-25 percent of an application's energy budget is spent in speculative execution, which is a tremendous waste in itself.

6 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 design, the overhead of refetching and reexecuting 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 current microprocessors. This is putting pressure on the precise 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 if there is enough space for the handler instructions to be fetched. 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).



(a)

(b)

(c)

(d)

Fig. 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 refetching and reexecuting instructions by 30-90 percent. (a) Jacobi, (b) Matmult, (c) Quicksort, and (d) Redblack.

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 the handler instructions in processor fetch 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 processor fetch order without incurring any side affects.

With interrupt in-lining, at the time the processor detects an exception, the processor first checks if there is enough space within the reorder buffer for the interrupt to be in-lined. If there is not enough space, the interrupt is handled 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, a deadlock situation can occur if the mode were changed to INLINE. For our studies, we observed that (besides no reorder buffer space) interrupt in-lining is not possible if the pipeline stalls (at the decode stage) due to insufficient renaming registers available to map user instructions fetched.

Our studies also show that lock-up free TLBs reduces the overhead due to the flushing of instructions by 30-95 percent. This is significant in that a processor no longer wastes time or energy refetching and reexecuting instructions. A reduction in the number of instructions flushed allows for lock-up free TLBs to provide a performance improvement of 5-25 percent. 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. Furthermore, we observed that applications that often require TLB management receive the most benefit from in-lining. 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 not only aids in performance improvement, but also reduces the energy consumption. By avoiding the refetch and reexecute of instructions, a processor can spend time doing useful work by executing those instructions independent of the exception causing instruction. Our studies show that modern 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 percent. 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 refetching and reexecuting 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 percent, reduces execution time by 5-25 percent, and reduces the energy wasted by 30-90 percent.

7 Future Work

For the purposes of this paper, we proposed nonspeculative 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 does not 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, four to eight 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.

Furthermore, in this study, we observed that the lack of renaming registers, and not ROB space, is the primary reason for not in-lining interrupts. As mentioned earlier, a possible solution 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. Such a mechanism if integrated will allow a processor to realize the full potential of interrupt in-lining.

Acknowledgments

The work of Aamer Jaleel was supported in part by US National Science Foundation (NSF) Career Award CCR-9983618, NSF grant EIA-9806645, and NSF grant EIA-0000439. The work of Bruce Jacob was supported in part by NSF Career Award CCR-9983618, NSF grant EIA-9806645, NSF grant EIA-0000439, US Department of Defense MURI award AFOSR-F496200110374, the Laboratory of Physical Sciences in College Park, Maryland, the US National Institute of Standards and Technology, and Cray Inc.

References

[1] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska, "The Interaction of Architecture and Operating System Design," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '91),* pp. 108-120, Apr. 1991.

[2] Z. Cvetanovic and R.E. Kessler, "Performance Analysis of the Alpha 21264-Based Compaq ES40 System," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA '00),* pp. 192-202, June 2000.

[3] M.K. Gowan, L.L. Biro, and D.B. Jackson, "Power Considerations in the Design of the Alpha 21264 Microprocessor," *Proc. 35th Design Automation Conf.,* pp. 726-731, June 1998.

[4] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report, vol. 9, no. 2,* Feb. 1995.

[5] L. Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report, vol. 10, no. 14,* Oct. 1996.

[6] D. Henry, B. Kuszmaul, G. Loh, and R. Sami, "Circuits for Wide-Window Superscalar Processors," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA '00),* pp. 236-247, June 2000.

[7] D.S. Henry, "Adding Fast Interrupts to Superscalar Processors," Technical Report Memo-366, MIT Computation Structures Group, Dec. 1994.

[8] J. Huck and J. Hays, "Architectural Support for Translation Table Management in Large Address Space Machines," *Proc. 20th Ann. Int'l Symp. Computer Architecture (ISCA '93),* pp. 39-50, May 1993.

[9] B. Jacob and T.N. Mudge, "A Look at Several Memory-Management Units, TLB-Refill Mechanisms, and Page Table Organizations," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '98),* pp. 295-306, Oct. 1998.

[10] B. Jacob and T.N. Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE Micro,* vol. 18, no. 4, pp. 60-75, July/Aug. 1998.

[11] B. Jacob and T.N. Mudge, "Virtual Memory: Issues of Implementation," *Computer,* vol. 31, no. 6, pp. 33-43, June 1998.

[12] A. Jaleel and B. Jacob, "In-Line Interrupt Handling for Software-Managed TLBs," *Proc. 2001 IEEE Int'l Conf. Computer Design (ICCD 2001),* Sept. 2001.

[13] A. Jaleel and B. Jacob, "Improving the Precise Interrupt Mechanism for Software Managed TLB Interrupts," *Proc. 2001 IEEE Int'l Conf. High Performance Computing (HiPC 2001),* Dec. 2001.

[14] T. Juan, T. Lang, and J.J. Navarro, "Reducing TLB Power Requirements," *Proc. 1997 IEEE Int'l Symp. Low Power Electronics and Design (ISLPED '97),* pp. 196-201, Aug. 1997.

[15] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown, "Design Tradeoffs for Software-Managed TLBs," *Proc. 20th Ann. Int'l Symp. Computer Architecture (ISCA '93),* May 1993.

[16] G. Kane and J. Heinrich, *MIPS RISC Architecture.* Englewood Cliffs, N.J.: Prentice-Hall, 1992.

[17] S.W. Keckler, A. Chang, W.S. Lee, S. Chatterjee, and W.J. Dally, "Concurrent Event Handling through Multithreading," *IEEE Trans. Computers,* vol. 48, no. 9, pp 903-916, Sept. 1999.

[18] J. McCalpin, "An Industry Perspective on Performance Characterization: Applications vs. Benchmarks," *Proc. Third Ann. IEEE Workshop Workload Characterization,* keynote address, Sept. 2000.

[19] M. Moudgill and S. Vassiliadis, "Precise Interrupts," *IEEE Micro,* vol. 16, no. 1, pp. 58-67, Feb. 1996.

[20] X. Qiu and M. Dubois, "Tolerating Late Memory Traps in ILP Processors," *Proc. 26th Ann. Int'l Symp. Computer Architecture (ISCA '99),* pp. 76-87, May 1999.

[21] M. Rosenblum, E. Bugnion, S.A. Herrod, E. Witchel, and A. Gupta, "The Impact of Architectural Trends on Operating System Performance," *Proc. 15th ACM Symp. Operating Systems Principles (SOSP '95)*, Dec. 1995.

[22] J.E. Smith and A.R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Ann. Int'l Symp. Computer Architecture (ISCA '85)*, pp. 36-44, June 1985.

[23] G.S. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors," *Proc. 14th Ann. Int'l Symp. Computer Architecture (ISCA '87)*, June 1987.

[24] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Research and Development*, vol. 11, no. 1, pp. 25-33, 1967.

[25] H.C. Torng and M. Day, "Interrupt Handling for Out-of-Order Execution Processors," *IEEE Trans. Computers,* vol. 42, no. 1, pp. 122-127, Jan. 1993.

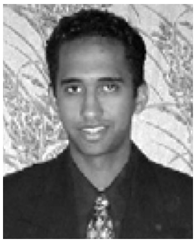
[26] M. Upton, personal comm., 1997.

[27] W. Walker and H.G. Cragon, "Interrupt Processing in Concurrent Processors," *Computer,* vol. 28, no. 6, June 1995.

[28] K. Wilcox and S. Manne, "Alpha Processors: A History of Power Issues and a Look to the Future," Compaq Computer Corp., 2001.

[29] K.C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro,* vol. 16, no. 2, pp. 28-40, Apr. 1996.

[30] C.B. Zilles, J.S. Emer, and G.S. Sohi, "The Use of Multithreading for Exception Handling," *Proc. 32nd Int'l Symp. Microarchitecture,* pp 219-229, Nov. 1999.



Aamer Jaleel received the BS degree in computer engineering from the University of Maryland, College Park, in 2000, and the MS degree in electrical engineering from the University of Maryland, College Park, in 2002. He is currently pursuing the PhD degree in the Department of Electrical and Computer Engineering at the University of Maryland, College Park. His research interests include memory-system design, computer architecture/micro-architecture, and workload characterization. He is a student member of the IEEE, IEEE Computer Society, and the ACM.

Bruce Jacob received the AB degree cum laude in mathematics from Harvard University in 1988 and the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1995 and 1997, respectively. At the University of Michigan, he was part of a design team building high-performance, high-clock-rate microprocessors. He has also worked as a software engineer for two successful startup companies: Boston Technology and Priority Call Management. At Boston Technology, he worked as a distributed systems developer and, at Priority Call Management, he was the initial system architect and chief engineer. He is currently on the faculty of the University of Maryland, College Park, where he is an associate professor of electrical and computer engineering. His present research covers memory-system design, DRAM architectures, virtual memory systems, and microarchitectural support for real-time embedded systems. He is a recipient of a US National Science Foundation Career award for his work on DRAM architectures and systems. He is a member of the IEEE, the IEEE Computer Society, and the ACM.