Concurrent Event Handling through Multithreading

Stephen W. Keckler, *Member*, *IEEE*, Andrew Chang, *Student Member*, *IEEE*, Whay S. Lee, Sandeep Chatterjee, *Member*, *IEEE*, and William J. Dally, *Member*, *IEEE*

Abstract—Exceptions have traditionally been used to handle infrequently occurring and unpredictable events during normal program execution. Current trends in microprocessor and operating systems design continue to increase the cost of event handling. Because of the deep pipelines and wide out-of-order superscalar architectures of contemporary microprocessors, an event may need to nullify a large number of in-flight instructions. Large register files require existing software systems to save and restore a substantial amount of process state before executing an exception handler. At the same time, processors are executing in environments that supply higher event frequencies and demand higher performance. We have developed an alternative architecture, *Concurrent Event Handling*, that incorporates multithreading into event handling architectures. Instead of handling the event in the faulting thread's architectural and pipeline registers, the fault handler is forked into its own thread slot and executes concurrently with the faulting thread. Microbenchmark programs show a factor of 3 speedup for concurrent event handling over a traditional architecture on code that takes frequent exceptions. We also demonstrate substantial speedups on two event-based applications. Concurrent Event Handling is implemented in the MIT Multi-ALU Processor (MAP) chip.

Index Terms—Exceptions, interrupts, multithreading, context switching.

1 Introduction

Modern microprocessors use exception or event hardware to handle infrequently occurring and unpredictable events. Events may be classified as *internal*, caused by the program executing on the processor, or *external*, caused by actions outside of the processor. Internal events may include arithmetic faults, memory protection violations, page faults requiring software to modify the TLB or fetch a new page of data from disk, or even user initiated system calls that trap into the operating system. Traditional examples of external events include watchdog timer interrupts and I/O interrupts caused by the disk drive or a keyboard controller. Regardless of event type, the purpose of the event detection hardware is to prevent a program from being penalized in the absence of events. Without hardware for exception detection, programs would be required to continually check for exceptional conditions.

Two trends in modern computer systems conspire against traditional exception architectures [28]. First, the cost of exception handling is increasing relative to the performance of microprocessors. The increasing width and depth of out-of-order superscalar pipelines accentuate the cost of event handling since the pipeline must be drained in response to an exception or event. In a 6-wide 10-deep pipeline, this results in as many as 60 wasted issue slots. Exception handling is further slowed by software that must frequently modify the pipeline state, vector to trap handlers, and manipulate the machine visible registers [4], [24]. The second trend is the increase in novel uses of event handling that are no longer "exceptional." These include support for garbage collection [5], software managed distributed virtual memory [21], and profiling [32]. Furthermore, for processors in embedded environments, performance can be limited by the ability to respond to a variety of external events, such as sensor inputs or packets in a network router.

The advent of hardware support for multithreading in microprocessors can alleviate both the latency and hardware complexity of event handling. When the hardware detects an event in a traditional single-threaded processor, an expensive sequence of actions must take place even before any of the event handling code is executed. The pipeline must be halted and drained, the stalled thread's registers and protection domain must be saved to memory, and the event handler's state and protection domain must be restored. Once the event handler has completed, this process must be reversed. Instead, with the appropriate mechanisms, a multithreaded processor can avoid most of this overhead by forking the event handler into a separate hardware context. Multithreading also provides the opportunity to execute applications and event handlers simultaneously, further reducing the overhead of events.

This paper introduces a general framework and nomenclature for *Concurrent Event Handling* and presents the resulting architecture, as well as an example implementation in the MIT Multi-ALU processor (MAP) chip. The MAP chip provides hardware support for concurrent execution of

- *S.W. Keckler is with the Department of Computer Sciences, University of Texas at Austin, Taylor Hall, 2.124, Austin, TX 78712. E-mail: skeckler@cs.utexas.edu.*
- *A. Chang and W.J. Dally are with the Computer Systems Laboratory, Stanford University, Gates CS Building, Stanford, CA 94305. E-mail: achang@cva.stanford.edu, billd@csl.stanford.edu.*
- *W.S. Lee is with Sun Microsystems, Inc., 8200 Central Avenue, MS: NWK 02-101, Newark, CA 94560. E-mail: whay.lee@sun.com.*
- *S. Chatterjee is with the Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139. E-mail: sandeep@lcs.mit.edu.*

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 110268.

both internal and external events. It incorporates three on-chip processors, each of which includes six thread slots for interleaving multiple threads of execution. In each processor, three thread slots are available for user level programs, while additional slots are reserved for exception and event handling, including TLB misses, synchronization failures, cache faults, and asynchronously arriving messages from other MAP chips in a multiprocessor system.

Using multithreading for event handling has two advantages. First, it reduces the overhead to invoke and return from an exception since the faulting thread is not removed from the pipeline. Second, event handling can proceed in parallel with the application, enabling faster event execution for both internal and external events. Hardware support for concurrent event handling allows the traditional requirements for sequentially precise interrupts in microprocessors to be relaxed, improving performance without sacrificing correctness.

In our experiments, we show that concurrent event handling outperforms event architectures that swap or stall the faulting thread. If a hardware thread slot is available for event handling, the latency of a null event drops from 100 cycles to 14 cycles. The null event handler's occupancy in a thread slot also drops to less than 10 cycles, reducing the contention between multiple threads for the execution resources. Applications that exhibit a high event rate show substantial performance improvements when using concurrent event handling. Eliminating swapping improves application performance by 19 percent, while running event handlers in parallel with application code provides an additional improvement of 26-43 percent.

The rest of this paper is structured as follows: Section 2 describes modern exception handing in further detail, highlighting the sources of overhead. The necessary architectural support for a concurrent event system and the implementation of concurrent event mechanisms in the MAP are detailed in Section 3. Section 4 explores the benefits of concurrent event handling through both microbenchmarks and two applications. Section 5 examines related work in multithreading and optimizations for fast event handling. Finally, we summarize the architecture and results in Section 6.

2 Events

As described in the introduction, an *event* is a condition that requires the services of the processor to handle. Events may be caused by an executing program (an *internal* event) or by an outside agent (an *external* event). Internal events are triggered by a running program and may result from software errors, such as a protection violation, or from normal execution, such as a page fault. External events are created by sources outside the current program's execution and may include I/O events, such as a disk request, timer generated events, or even a message arrival in a message-passing multiprocessor system.

The response of a processor to an event may be *precise* or *imprecise* and may be *concurrent* or *sequential*. The response to an event is precise if proper handler execution ensures correct program behavior. Precise event handling is guaranteed if the particular instruction that caused the event (the faulting instruction) is identified and any instructions that need the result of the faulting instruction are not issued until the handler has generated this result.

This definition of precise event handling enables us to use multithreading to implement concurrent event handling; the event handler and the faulting program run concurrently in different thread contexts. This is in contrast to sequential event handling in which an event interrupts the faulting program, brings it to a sequentially consistent state, runs the handler to completion, and resumes the faulting program. Concurrent and sequential event handling take different approaches to achieve the same result; ultimately, both guarantee that the correct input operands are delivered to each instruction.

We introduce this nonstandard event nomenclature for two reasons: to avoid confusing the origin of an event (internal or external) with the response of the processor (whether it is interrupted or not) and to avoid confusing the intent of a precise event (correct program behavior) with the method of achieving this intent (sequential or concurrent execution).

The nomenclature used in the literature tends to mix the concepts of origin, intent, and implementation together. In [11], internal and external events are called synchronous and asynchronous events, respectively, terminology that tends to suggest a method of handling the event. Many sources, including [27], call internal events exceptions and external events interrupts. The term interrupt, in particular, suggests a method of handling the event rather than identifying the source of the event. The term exception is neutral on location, as exceptional conditions may be raised by internal or external sources. At the same time, an internal event may occur during normal operation, such as for a system call, and not correspond to what is normally considered an exceptional condition.

Both intent and implementation are combined in the common interpretation of a *precise interrupt*. As defined in [27], this type of event guarantees correct behavior (precise), and achieves this by strictly ordering the execution of faulting program and handler instructions (sequential). Specifically, a sequentially precise interrupt satisfies the following conditions:

- All instructions preceding the faulting instruction have been executed and have modified process state correctly.
- All instructions following the faulting instruction are unexecuted and have not modified the process state.
- If the event is caused by an instruction in the program (the faulting instruction), the program counter points to that instruction. The faulting instruction must either have completed or not have started execution.

Sequential precise interrupts are convenient in single-threaded processors since they limit the software complexity of halting and restarting the faulting program. The program counter captures the entire state of the pipeline. Regardless of the actual instruction execution order, the processor appears to have executed all of the instructions sequentially.

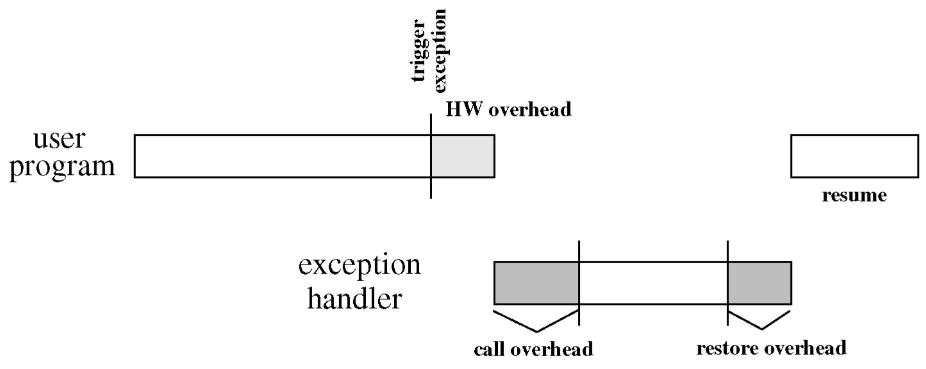

Fig. 1. The user program is interrupted by an exception which incurs both hardware and software overhead before the exception handler can begin executing.

In a single-threaded, unpipelined processor, all events are handled in a sequential fashion, as shown in Fig. 1. When an event is detected, the program halts and an event handler is loaded and run. When the handler completes, the original program is reloaded and restarted at the faulting instruction. In addition to the execution time of the event handler, the diagram also depicts both the hardware and software overheads associated with invoking and returning from an event handler. In a multithreaded processor, or one that is heavily pipelined, it is advantageous to handle events concurrently, without halting the user's program; both the event handler and the user program can run simultaneously. The remainder of this section explores the different overheads associated with event handling and describes how these overheads can be reduced by handling events concurrently.

2.1 Hardware Overhead

Much of the complexity in sequential event handling stems from stopping and restarting a thread that causes an event. In a purely sequential model of instruction execution, an instruction is issued and runs to completion before the next instruction is issued. In this case, event handling is trivial since the instruction after the one that caused the event has not yet been issued. The hardware needs only to squash the faulting instruction and can later resume by reexecuting either it or the subsequent instruction. High-performance processor implementations violate this model by executing multiple instructions simultaneously, through both pipelining and parallel issue. The state of the program may be modified in an order different from the sequential execution of the instructions. In order to allow a process to continue correctly after an interrupt, the process state must be saved in a consistent form.

Smith and Pleszkun describe several methods for implementing sequential precise interrupts in pipelined processors, with the more complex methods logging the process state during execution so that the correct order can be restored at the time of an interrupt [27]. One method uses a history buffer to track the sequential completion order of instructions. Instructions are issued in order and placed in the history buffer along with the original value contained in each instruction's destination register, prior to execution of the instruction. When the instruction that caused the event has reached the head of the history buffer, instruction issue is halted until the pipeline, including the memory system, has drained. The history buffer is then emptied in reverse order, which restores the original values back into the register file. The Motorola MC88110 can issue two instructions per cycle and uses a history buffer to keep track of out-of-order instruction completion [31]. When an instruction triggers an event, it must first wait for all previous instructions to complete, which, in the worst case, includes long latency instructions, such as cache misses and floating-point arithmetic operations, and requires as many as 57 cycles. The replay of the history buffer may take as many as six cycles and the transfer of control to the event handler requires three cycles. Thus, for a relatively simple, 2-way in-order issue superscalar processor, as many as 66 cycles may be required before any of the event code is run.

A second method of implementing sequential precise interrupts holds both the speculative and nonspeculative state in register files. One strategy uses a separate speculative register file, called the Future File, to hold register updates before instructions commit. When an event occurs, the contents of the architectural register file are copied to the future file. The MIPS R10000 implements a similar strategy by using the register renaming logic to save the nonspeculative state until instructions commit [35]. When an event occurs, the prematurely renamed registers are unmapped. To handle the case where a logical register was remapped twice, the logic unmaps up to four instructions per cycle, in reverse order from the active instruction queue.

Regardless of the method of recording speculative state modifications, substantial overhead can result when an interrupt occurs. In recent wide out-of-order issue superscalar processors, the number of uncommitted instructions can be large, leading to a significant penalty to restore the sequential precise state of the faulting instruction. Furthermore, a great deal of useful work can be lost when an event occurs. All of the already issued instructions after the one that causes an event must be reexecuted, even if they are independent of the event.

The hardware required to implement sequential precise interrupts can also be quite complex. In fact, Digital's Alpha 21064 processor implements only a bare-minimum of precise interrupts. If the software requires precise interrupts on an arithmetic operation, it must execute the TRAPB (trap

TABLE 1
Latency for Null System Call and Return

Machine	time (μseconds)	Cycles
Motorola 8800 [4] (20MHz)	11.8	236
MIPS R2000 [4] (16.6MHz)	9	149
MIPS R3000 [4] (25MHz)	4.1	103
Sparc Ultra I (167MHz)	2.9	484
HP PA-8000 (180MHz)	0.9	162
Pentium Pro (200MHz)	1.9	380
Sparc Ultra II (300MHz)	1.6	474

barrier) instruction which serializes instruction issue by preventing further instructions from executing until all previous instructions have been guaranteed not to cause an event. While this dramatically simplifies the hardware, these events are no longer precise unless TRAPB instructions are inserted between every pair of instructions. Consequently, handling these events in software becomes more difficult. A variety of additional design choices for hardware event processing are detailed in [33].

2.2 Software Overhead

Even after the software event handler has been invoked, a substantial number of instructions may need to execute before any useful work can be done. As observed by Anderson et al. [4], trap handling has become even more costly in modern microprocessors. Before any instructions from the specified event handler can execute, the software at the trap entry point must change the addressing environment for the operating system, manipulate any machine state necessary to service the outstanding event or events, save or restore any registers that must be preserved during the operating system call, and vector to the specific event handler. Changing the addressing environment can even entail swapping TLB entries and flushing the cache. If the hardware itself does not implement sequential precise interrupts, the trap handler must examine the machine's pipeline state, store what is necessary, and determine which of the executing instructions has outstanding events. Since the operating system uses the same pipeline and registers as the program that caused the event, all of the architectural state must be saved to memory. Finally, if the hardware does not automatically provide the address of a fault-specific handler, the software must examine the processor state to determine the type of fault that occurred before jumping to the appropriate code.

Table 1 shows the latency for a null system call and return (using a trap into the operating system). The first three rows are results from Anderson et al. [4], while the remaining rows are from more modern systems measured using lmbench [23]. The combination of the hardware and software overheads results in substantial latencies to handle a trap. The number of cycles required to handle a trap has already increased with the deeper pipelines and greater processor state of modern architectures. This trend is likely to continue in the future.

2.3 Concurrent Event Handling

Nearly all of the overhead associated with event handling is due to starting and stopping the event handler and faulting program. A multithreaded processor need not incur this overhead of sequential fault handling. Instead, it can handle the fault concurrently, running the handler thread in parallel with the faulting program thread. As shown in Fig. 2, when an event occurs in this model, the event handler is forked into a waiting thread slot. If the event must be handled before the user thread proceeds, the user thread stalls while the event handler is run. Otherwise, the user thread can continue to execute until it reaches a point where it needs the result from the event handler.

Consider, for example, a load instruction that causes a TLB miss that must be handled in software. The program that issued the load need not stall until it actually requires the data. When the TLB miss occurs, the memory system hardware sends the address that caused the miss, along with other control information, including the ultimate destination of the load result, to an event handler waiting in an idle thread slot. The event handler then executes concurrently with the user thread. After the handler replaces the TLB entry, it can reexecute the load in its own thread slot and deliver the data to the original destination register. A register scoreboard can stall the user thread when it reaches an instruction that needs the result from the TLB handler.

These events do not satisfy the definition of sequential precise interrupts, in part since the faulting program is not interrupted and may continue to issue and retire instructions. However, they do maintain precise event semantics.

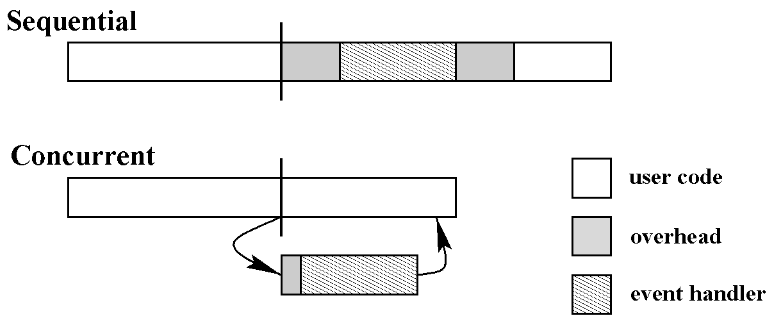

Fig. 2. The sequential event handler is run in the same thread slot as the user code. Concurrent event handlers run simultaneously in a different thread slot, reducing the hardware and software overheads associated with switching between threads.

When the event handler is invoked, it knows exactly which instruction caused the event and has enough information to complete the execution of the instruction. Correct program behavior is guaranteed as the program that caused the event is prohibited from executing any instructions which use a result that subsequently will be produced by the event handler.

External events can be handled in the same fashion, but are simpler since they need not interact with the executing program. By combining multithreading with event handling, we eliminate the hardware and software overhead associated with swapping between user and event threads. These same mechanisms can provide additional benefit by exploiting parallelism between the user code and the operating system.

3 Architectural Support for Concurrent Event Handling

Modern microprocessor architectures are highly tuned for execution of sequential programs and the features required to implement concurrent event handling do not come without cost. While many implementations are possible, the necessary mechanisms for concurrent event handling include:

1. Spare execution resources: The architecture must support concurrent execution from multiple threads of control. Architectures that include multithreading, multiple processors on a single chip, or a combination of both are all viable alternatives.
2. Hardware thread invocation: The chip must have the ability to invoke an event handler without causing any other running threads to stall. Any software overhead between event detection and event handler invocation would defeat the purpose and efficiency of concurrent handling.
3. Synchronization: A thread that causes a fault must be able to proceed until it reaches a point where it needs the result that is to be generated by the event handler. This synchronization should have little, if any, overhead, not block the faulting thread indiscriminately, and not require the application to be modified (e.g., spin loops are not acceptable).
4. Protection: Multiple threads from different protection domains, such as a user's application and the operating system, must be able to coexist and execute simultaneously without risk of any protection violations. Furthermore, the event handler program must be able to communicate results back to the thread that caused the event without requiring an expensive interprocess communication.

Omitting any of these features will result in a system that will not execute concurrent events efficiently. The remainder of this section describes these mechanisms in more detail and how they are implemented in the Multi-ALU Processor chip. At the end of this section is a discussion of some important design considerations for concurrent events that are applicable to other processors.

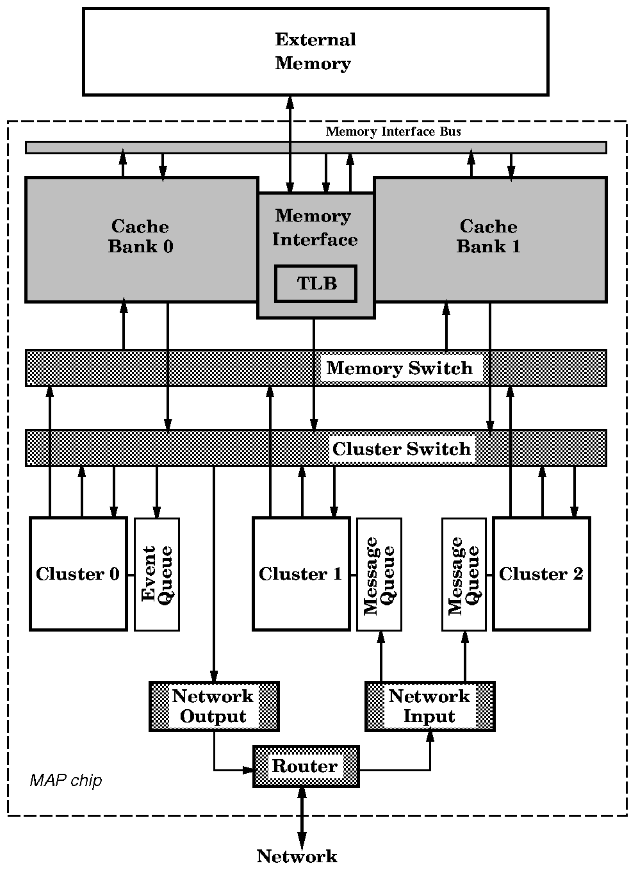

Fig. 3. Block diagram of the MAP chip, containing three clusters (processors), each with a dedicated event or message queue.

3.1 The Multi-ALU Processor

The Multi-ALU Processor (MAP) is a sample implementation of our architecture that provides hardware support for concurrent event handling The MAP includes three processor clusters that execute independently but can communicate and synchronize directly through processor registers [10], [15], [13]. In addition, each cluster is multithreaded to enable multiple programs to share the execution resources in an interleaved fashion. Hardware support allows internal and external events to execute concurrently with each other and with the user-level programs. This section describes the architecture and implementation of the MAP chip including both the mechanisms for multithreading and concurrent event handling.

Fig. 3 shows a block diagram of the MAP chip's features for concurrent event handling. Each of the three MAP clusters is a 64-bit, three-issue, pipelined processor consisting of two integer ALUs, a floating-point ALU, associated register files, and a 4 KB instruction cache. The memory system includes two on-chip cache banks and an external memory interface which interfaces to the off chip synchronous DRAM. These components are connected through two crossbar switches (the memory and cluster switches), which enables a cluster to communicate with the cache banks and with other clusters. An on-chip multiprocessor network interface and two-dimensional router allow a message to be transmitted from a cluster's register file into the network. The event queue and the network queues hold events from the memory system and the network until they can be processed by the specified clusters.

3.2 Multithreading Support on the MAP

Multithreading on the MAP chip is implemented by interleaving instructions from different threads over the execution resources of each cluster on a cycle-by-cycle basis. Unlike block multithreading [1], [34], which switches threads on long latency operations, such as cache misses, and round-robin thread scheduling [2], [26], which forces a thread switch every cycle, the MAP chip makes its thread selections based upon the availability of an instruction's register operands. If an instruction's input data is not available, the instruction will not execute. Scheduling threads based on data availability enables the processor to tolerate long memory latencies, as well as short instruction latencies, for a particular thread, filling in the otherwise idle execution units with instructions from other threads. This is ideal for concurrent event handling as the thread that caused the event, as well as other executing threads, can continue to execute by issuing into the event handler's unused instruction slots. Single thread efficiency is maintained because, even if only one thread is active, it may issue an instruction every cycle as long as the operands for its instructions are ready. The two main features of the MAP that provide tremendous flexibility in executing multiple threads are the thread scheduler and the memory protection mechanisms that allow threads from different protection domains to execute simultaneously.

3.2.1 Thread scheduling

In each cluster, the MAP chip dedicates a pipeline stage (the synchronization or SZ stage) to thread scheduling. One 3-wide instruction waits in the SZ stage's reservation station until the instruction is ready to issue [29]. The SZ stage maintains valid bits from the register scoreboard for the input operands of all instructions waiting in the reservation stations. The scoreboard tracks which registers are valid and which are waiting for data to arrive. Once all of the operands are available, instructions from different threads compete for use of the execution units based on policies of priority and nonstarvation. The thread slots are divided into two categories, system and user, and each thread has two priority levels. A simple three state priority scheme is used to determine which threads can compete for the issue slots. High priority system threads are the most important and can monopolize the execution resources when necessary. Low priority system threads and high priority user threads are at the same level and have equal access to the execution units. At the bottom of the hierarchy are low priority user threads. This priority scheme enables critical threads, including those dedicated for event handling, to use a higher fraction of the execution resources than user-level threads. The synchronization stage also implements a preempt state (that includes idle cycle counters and limit registers) to guarantee that a ready instruction can only be stalled for up to 255 cycles, depending on the parameter stored in the limit register. This strategy ensures that, even if one thread is creating a tremendous number of events, other threads will be able to continue making forward progress. Further details on the design and implementation of the MAP's pipeline can be found in [13].

3.2.2 Protection

Since the event handler may execute code or access data that is privileged, concurrent event handling must allow multiple threads from different protection domains to execute simultaneously. Furthermore, an event handler must be able to communicate with the threads that invoked it so that it can read the state of the faulting thread and pass results back when the event handler completes. While low overhead protection has been incorporated into virtual memory systems to reduce the overhead of context switching [16], [19], these systems do not allow data to be easily passed between protection domains. The MAP chip separates protection from virtual memory by using Guarded Pointers, which enable threads from different protection domains to reside within the same virtual address space and be loaded simultaneously on the processor [7]. Guarded pointers implement a light-weight capability system that organizes the global address space into segments which must be accessed with unforgeable tagged pointers. The pointer tag is represented by a 65th bit attached to each 64 bit word in datapaths, registers, and memory. Segmentation checks are performed in the cluster during address calculation. When a memory operation executes, the permission of the pointer is examined to determine if the operation is legal. If the check determines that a memory operation is illegal, a cluster's memory unit triggers an internal event. Data is controlled on a segment-by-segment basis, which can prevent protected data from being read and read-only segments from being written. In addition to enabling threads from different protection domains to execute simultaneously, guarded pointers allow fast memory communication between threads through shared objects and simple kernel level access to user data structures. The MAP also allows a system level thread to write directly into the registers of a user level thread, which is necessary when reenabling a user thread when the event handler terminates.

3.3 MAP Event Handling Mechanisms

The MAP chip uses multithreading to handle both internal and external events. Internal events are divided into those that are handled sequentially and those that are handled concurrently. Sequential events occur within the pipeline's execution units and can be detected at the beginning of the execute stage. Examples include protection violations and speculative exception detection. Concurrent events are detected outside of the pipeline and include TLB misses, synchronization failure, and cache status failures. Table 2 shows the allocation of thread slots to each event type. The sequential events are handled in thread slot 3 of each cluster, while the TLB and general events are handled in different thread slots on cluster 0. The general event handler executes synchronization and memory coherence events using all of the clusters in slot 4 if necessary. External events are executed in dedicated message handler threads on clusters 1 and 2. The key hardware components for handling any of these types of events quickly include the

TABLE 2
The Six Threads Are Partitioned into Three User Slots, One Sequential Event Slot and Two Concurrent Event Slots

	Cluster 0	Cluster 1	Cluster 2
slot 0	user	user	user
slot 1	user	user	user
slot 2	user	user	user
slot 3	sequential event	sequential event	sequential event
slot 4	general events	system	system
slot 5	TLB events	priority 0 messages	priority 1 messages

generation of event records, synchronization to indicate to the handler when an event has occurred, and memory mapped access to thread registers so that the handler can return data to a waiting user thread.

3.3.1 Sequential Internal Events

The MAP chip minimizes the downtime of threads due to events by reducing the overhead for each event and providing mechanisms to eliminate as many events as possible. On each cluster, slot 3 is reserved for handling local events that can be detected within the cluster during the first half of the execution unit clock cycle. These sequential internal events include executing a privileged instruction while in user mode, storing to an illegal pointer, and sending to an illegal address. When the execution unit detects an event, it stalls the pipeline and writes the information associated with the event into the registers of thread slot 3. Since the event is executed in its own thread slot, the thread that caused the event merely waits until the event is complete. No user registers need to be saved and restored and the pipeline does not need to be restarted. As a result, a null event handler can start and complete in less than 14 cycles.

The event record includes the address of the instruction that faulted and the reason for the event. When an event handler is running, all user threads are prohibited from executing instructions so that a second event cannot occur while this event handler is busy. When the event handler is complete, it reenables the user threads. A sequential internal event caused by the event handler or by a system level event handler is a system software error and results in an unrecoverable catastrophic event.

In order to enable speculative execution of instructions, the MAP provides a mechanism for deferring events. Deferred events can result from the creation of an illegal pointer or loading from an illegal address. When a deferrable event is detected, a special pointer called an ERRVAL (error value) is written into the result register. The ERRVAL is a tagged guarded pointer that encodes the address of the instruction that created it and the reason for its creation. ERRVALs can propagate through subsequent arithmetic instructions, allowing a stream of speculative instructions to occur without the risk of an unwanted event. Instructions that have no result (such as a store) and comparison operations that write into a single-bit condition code register cannot propagate an ERRVAL and must invoke a sequential internal event handler.

3.3.2 Concurrent Internal Events

Events that are caused by an executing program but are detected outside the MAP cluster where the thread is running are termed concurrent internal events and are handled asynchronously by generating an *event record* and placing it in a hardware event queue. Local TLB misses, block status faults, and memory synchronization faults are events that are handled without blocking execution of any user level program. These events are precise since the faulting operation and its operands are specifically identified in the event record.

For example, when the TLB thread is idle, it stalls waiting for a particular register to be written by the event detection hardware. When a TLB miss occurs, the external memory interface hardware formats an event record, shown in Fig. 4, containing information about the cause of the fault: the address that faulted, any data that was associated with a faulting store, a memory mapped pointer to the destination register in case of a load, and a memory mapped pointer to the physical location in the TLB in which to put the new entry. The event record is written directly into the integer register file of the TLB thread in slot 5 on cluster 0. When the event record has been received in the register file, the specified registers are marked full, which wakes up the TLB thread to handle the event. The software TLB miss handler reads the record, places the requested page table entry in the TLB, and restarts the memory reference. The thread that issued the reference does not block until it needs the data from the reference that caused the miss.

For general events, such as memory synchronization failures or cache status faults, the event record is written into a hardware *event queue*. A general event record is nearly identical to a TLB record, except that it does not need the pointer to the TLB entry. A software handler reads event records from the queue and processes them sequentially. Integer registers are mapped to the queue so that a read

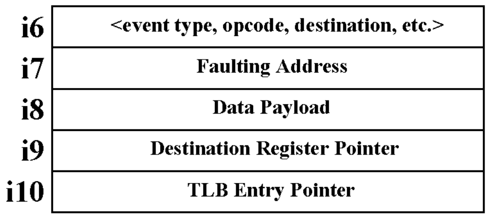
i6
<event type, opcode, destination, etc.>
i7
Faulting Address
i8
Data Payload
i9
Destination Register Pointer
i10
TLB Entry Pointer

Fig. 4. Format of a TLB miss event entry written directly into the integer registers of the TLB miss handler on cluster 0.

from register `i15` will dequeue the word at the head of the queue and a read from register `i14` will pop the current event record from the queue and return the first word from the next record. If there are no records in the queue, the event handler will stall trying to read the queue head.

Handling events concurrently obviates the need to cancel all of the issued operations following the faulting operation, a significant penalty in a wide machine with deep pipelines. Dedicating thread slots to this purpose accelerates event handling by eliminating saving and restoring of thread state and allowing concurrent (interleaved) execution of user threads and event handlers. However, concurrent event handling does require sufficient queue space to handle the case where every outstanding instruction generates an event. If insufficient space exists in the event queue, user threads must be stalled to prevent additional events from overflowing the queue. As many as 13 memory instructions can be outstanding in the MAP chip's memory system pipeline. As each instruction could require a four word event record, the minimal size of the event queue is 52 words. To provide the elasticity necessary to prevent premature stalling of threads on a saturated event system, the MAP implements a larger 128 word event queue.

3.3.3 *External Events*

External (asynchronous) events are executed using the dedicated message handling threads without requiring any user threads to be interrupted. Like the general concurrent event handling mechanism described above, arriving messages are placed in an incoming message queue. The handler thread reads the message contents, processes messages sequentially, and automatically stalls when the queue is empty. The message format is different from the general event format. By convention, the first word is an instruction pointer to which the handler jumps for fast event vectoring. The subsequent words are the payload of the message and can serve as arguments to the procedure specified in the instruction pointer. In addition to servicing messages in a multiprocessor environment [20], the message handlers can also process other types of events, including external timers or data delivery from a disk.

3.4 Design Considerations

Architectures for concurrent event handling must pay specific attention to managing the resources that can be accessed concurrently from different threads of control. For example, what happens when a second event occurs while a previous one is already in progress? This situation can arise if multiple threads can cause events independently or even if the thread that created an event is allowed to create another before the first one has been handled. Preventing all other threads from executing when any single thread has caused an event is far too restrictive. The Multi-ALU Processor partitions the events into different categories and events of a particular category are queued together and handled one at time. Another architecture might use any available coprocessor or thread slot to handle multiple events, but an event must not be lost if all of the thread slots are already busy.

Another complication can arise if event handlers are allowed to cause events, for example a page fault taken by a memory synchronization event handler. The design of the event system must prevent circular dependences between event handlers in order to avoid deadlock. The MAP chip achieves this through the event type partitioning described above and by creating a hierarchy of event handlers: Memory synchronization and status fault events can cause a TLB miss event, but the TLB miss handler cannot cause any faults. Resorting to a synchronous exception and running the event handler in the thread slot that caused the event can also be used to help break circular resource dependences.

A third consideration is the order in which events should be processed. Depending on the programming model, it may be necessary to execute all events from a particular thread in order or to execute all events of a particular class in temporal order. The event queue in the MAP enforces one ordering, but it may not be appropriate in other applications. In our experience, defining the event ordering and concurrency protocols was nearly as difficult as implementing them (including all of the special cases) in hardware.

4 Evaluation of Mechanisms

This section examines the performance benefits of using a dedicated thread to reduce overhead and execute events concurrently with the faulting thread. The evaluation includes microbenchmarks to examine specific aspects of event handling, as well as two applications that use frequent events as a part of their execution. Since no current microprocessor provides fast event handling, finding applications that use events as an integral part of their execution is challenging. We selected and implemented two application kernels that take events frequently: 1) Software Shared Memory in which events are automatically generated to fetch remote data on memory operations and 2) Dynamic Profiling which nonintrusively samples the program counter of a running program based on an externally triggered event.

The microbenchmarks were written in MAP assembly code and run on MSIM, the functional simulator of the MAP. MSIM executes 400-1,000 MAP cycles per second, depending on how many clusters are active, and has been validated to be cycle-accurate using the register transfer level (RTL) model of the chip. The applications are compiled using MMCC, the MAP C compiler, which is a derivative of the Multiflow C compiler [22].

4.1 Baseline Mechanisms

As described in Section 3, the MAP's event handling mechanisms can eliminate nearly all of the overhead associated with switching between the user and event handling threads. Since the event handler runs in its own context, the pipeline need not be drained and no context switching or address environment modifications are required. Fig. 5 shows the null event handler that is used to examine the speed of the MAP's event handling. The event handler stalls on integer register `i14` waiting for an event to be triggered. When the event record is written into the event queue, the handler removes all four words, returns a meaningless value to the thread that caused the event, and

```
            EVENT_WAIT:
                    instr ialu mov i14, i6;   -- get next event header
                    instr ialu mov i15, i7;   -- word #2
                    instr ialu br EVENT_WAIT;-- branch back + 3 delay slots
                    instr ialu mov i15, i8;   -- word #3
                    instr ialu mov i15, i9;   -- word #4
                    instr memu st i2, i9;     -- return data to user
```

Fig. 5. A null event handler waits until a new event arrives, pulls the entire event record from the event queue, and returns a word to the faulting thread.

loops back to wait for the next event to occur. The fourth word in the event, loaded into `i9`, contains the memory mapped address of the destination register from the instruction that caused the fault. The store to `i9` writes the value of `i2` into the original destination register. If the faulting thread reaches an instruction that needs the result of the faulting instruction, it stalls until the event handler delivers the data.

Fig. 6 shows the time required to resolve an event both by forking to another thread and by context switching in the existing thread slot. The test program causes an event by performing a synchronizing `load` that fails in the cache. The faulting thread immediately uses the result of the faulting `load` in the subsequent instruction. As shown in the *Fork* timeline, three cycles elapse from the time the `load` issues until the first word of the event record reaches the event thread. Eight cycles are spent in the event handler (Fig. 5) and three more cycles are required to deliver the dummy word back to the faulting thread. If the event is handled in the original slot, the state of the faulting thread must be saved to memory. The *Swap* timeline in Fig. 6 shows the additional overhead required to drain the pipeline and save/restore 38 architectural registers to memory (15 integer, 15 floating-point, and 8 condition code registers). Combining the 43 cycle swap overhead with the previous 14 cycles results in a total event latency of 100 cycles, a factor of 7 more than *Fork*. While this cycle count is perhaps pessimistic in the amount of user thread state that needs to be saved, it does not take into account backing up pipelines or changing addressing environments. Note that even the swap overhead shown in Fig. 6 is substantially lower than the null event latency from the modern microprocessors shown in Section 2, which ranges from 162-474 cycles. Of course, the latency benefits of forking the event handler will be less significant if the event handler runs for a long time before returning to the user program.

In addition to lowering the overall overhead of events, the MAP's event mechanisms can reduce the impact of events by concurrently executing the event handler and the thread that caused the fault. Events that have *slack* between the instruction that causes an event and the instruction that needs the result can benefit from concurrent event handling. For example, if a prefetching instruction that causes a software handled TLB miss is pushed up in the instruction stream, it may have enough time to complete before the resulting data is loaded downstream. The microbenchmark that illustrates this executes a tight loop that causes a synchronous event and immediately uses the result. The number of instructions between the event source and sink (i.e., the slack) is varied from 0 to 30 cycles. The results for four different event handling mechanisms are shown in Fig. 7. The **Swap** line corresponds to the case where the faulting thread is swapped out in favor of the event handler. **Stall** assumes that a shadow register set is available so that the thread swap overhead is not incurred, but the faulting thread is still halted. **MT** is the base multithreaded case where the fault handler and the user thread share the execution resources of a single MAP processor cluster. **Co_Proc** executes the event handler and the faulting thread concurrently on different clusters. All of the results are normalized to the thread swapping case with no slack cycles.

As shown in the figure, eliminating the thread swap is the most important consideration, as it reduces the overhead by more than a factor of two. Executing the event handler concurrently with the faulting thread (**MT**) is only

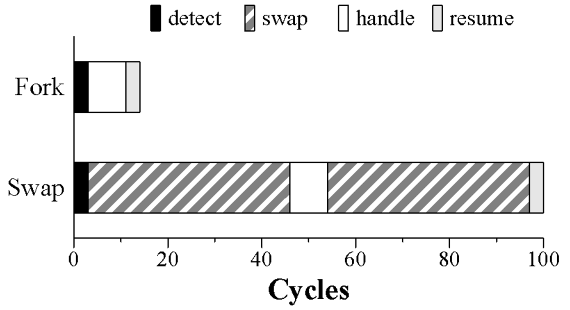

Fig. 6. Event handling latency using multithreading (Fork) and thread swapping (Swap).

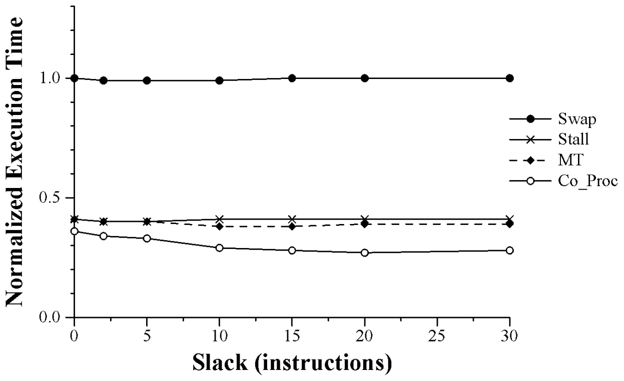

Fig. 7. Effect of context switching and concurrency in event handling.

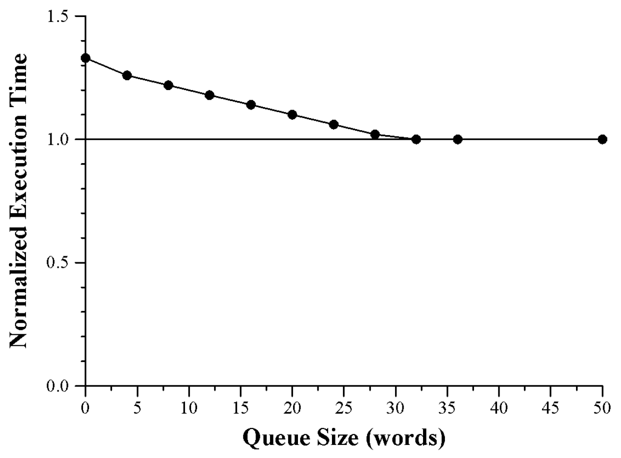

Fig. 8. Effect of elasticity in event handling due to queuing in the event system.

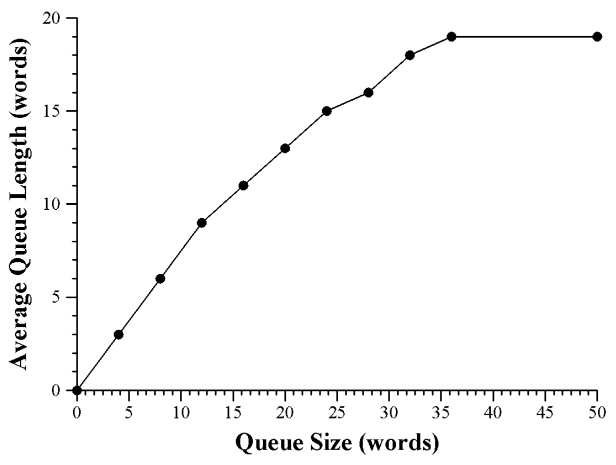

Fig. 9. Average event queue length.

slightly faster than stalling the user thread (**Stall**). Since the null event handler runs for only a few cycles and does not stall during execution, the benefit of interleaving instructions from one thread into the stalled issue slots of another is somewhat mitigated. However, if the event handler and faulting thread can run without conflicting for issue slots, the benchmark runs approximately 20 percent faster. No further improvement is seen for **MT** or **Co_Proc** when the slack exceeds the number of cycles required to execute the event handler.

A critical microarchitecture parameter that determines the performance of a concurrent event system is the length of the event queue. Since any thread that can cause events must be halted if the event queue is full, a long event queue allows applications to execute even if many events are pending. When running an application with multiple threads, continuing to execute application code during event handling is necessary for good performance. For example, a software shared memory application will be able to overlap computation with repeated invocations to an event handler that fetches cache lines from a remote processor. The following experiment measures the sensitivity of an application's performance to queue length. A simple microbenchmark executes a loop that causes 20 events, followed by equivalent computation time that causes no events. This sequence is repeated twice. Fig. 8 shows the execution time of this program when the queue size is varied from 0 to 36 words (nine event records), normalized to the execution time using an infinite queue. If the queue becomes full, the thread that causes the events is stalled so that the queue does not overflow. The figure shows that the elasticity of the queue enables event processing to be spread out into those periods of the program in which events are infrequent. When queuing is removed from concurrent event handling, the faulting thread must be stalled more frequently. Fig. 9 shows the average length of the event queue when a new event occurs. The fraction of its size that is full decreases until the queue length reaches 36 words, at which point the queue is, on average, half full. In this experiment, an event queue of 36 words is large enough to capture all of the outstanding events without requiring any stalling. Since this experiment generates a new event every eight instructions, a queue size of 36 words can be considered an upper bound for single-threaded applications or uniprocessor jobs. However, more queue entries may be needed if multiple threads are generating events simultaneously.

4.2 Software Shared Memory

Fine-grained software shared memory can be implemented on the M-Machine using a combination of the event handling mechanisms and tag bits in the cache. Two block status bits are associated with each cache line and implement four states in a cache coherence protocol. If a read/write to an invalid cache line or a write to a read-only cache line is detected in the memory system, an event record is written into the event queue. The event handling thread sends a message (using the MAP's `send` instruction) to the home processor of the address. At the home processor, the request is handled by the priority 0 message thread and a result is sent back to the originator. The response message is handled as an external event by the priority 1 message thread and the data is returned to the original faulting thread. The full details of using the MAP's mechanisms to implement software shared memory can be found in [6].

The sample shared-memory benchmark is a solution to a 3D Poisson partial differential equation and is based on the multigrid kernel from the NAS parallel benchmarks and SPEC95. The input data set consists of a 3D matrix of 512 elements and each of the four MAP chips computes a subset of the three-dimensional data space. The different computation phases are separated by barriers. Three different event architectures are modeled. **MAP** uses the MAP's concurrent event handling. **Stall** stalls the main program thread on the faulting processor during any event execution, emulating a dedicated event register file. **Swap** stalls the main program thread and incurs a thread swap overhead of 86 cycles for each event.

Table 3 shows the event latencies and frequencies averaged across all of the processors for each event architecture. The time to handle each of the event types is fairly short (115-237 cycles) and each of the event types occurs frequently. When the individual event frequencies

TABLE 3
Event Latencies and Frequencies

Event Type	Execution Time (Cycles)	Frequency (Cycles) MAP	Stall	Swap
Cache Status	237	1872	1973	2605
Request Message	175	693	768	988
Response Message	115	656	729	941
Total		285	314	407

are combined, the overall frequency ranges from 285-407 cycles. The architectures that incur more event overhead (**Stall** and **Swap**) have lower event frequencies, since their running time is extended by slow event handling.

With high event frequencies, long event overheads can contribute substantially to the overall runtime of the program. Table 4 shows the execution time for the multigrid computation using the three different event architectures. Stalling the program thread on every event results in a 40 percent penalty in execution time, while including the additional overhead of swapping the state out increases the execution time by 70 percent. The low overhead and concurrent execution of internal memory events and external message events provided by the multithreading of the MAP enables faster execution of programs with high event rates.

4.3 Dynamic Profiling

Hardware support for on-line profiling of programs running on out-of-order processors has recently been examined by Dean et al. [9]. Their system periodically tags an instruction, collects statistics about its execution behavior (time in different pipeline stages, etc.) in hardware counters, and causes an event to log the information. A similar study used random sampling to reduce the event overhead and showed only a 1-5 percent performance penalty when the periods between sample points is at least 60,000 cycles [3]. Concurrent event handling offers a method of improving both the performance and the sampling rate by tabulating the statistics simultaneously with the running program.

The Dynamic Profiling benchmark described in this section captures the essentials of on-line performance monitoring through concurrent events. One MAP processor runs the application to be profiled and periodically receives external events. For simplicity, the external event just reads and logs the program counter of the executing application. The program counter is read through a memory mapped access to thread registers and does not require the sampled program to be halted. For simulation purposes, the external events are generated by messages sent by a program running on a separate MAP processor. Four event architectures are examined: **Swap** stalls the application and incurs an 86 cycle thread swap penalty, **Stall** stalls the application and uses a spare register set for the event handler, **MT** interleaves execution of the event handler and the application, and **Co_Proc** runs the application and event handler concurrently on different MAP clusters. The application used for profiling is a 10×10 matrix multiplication.

Fig. 10 shows the execution time for these four event implementations normalized to the execution time of the unsampled application. The x-axis displays the time between samples and is on a logarithmic scale. The handler that samples the events executes in 30 cycles and the time between samples is varied from 30 to 6,600 cycles. Executing the handler completely concurrently (**Co_Proc**) has no impact on the running time of the application, but the other three event models are far more intrusive. In fact, at the highest rates (lowest time between events), the application cannot make forward progress. As the time between events increases, the sampling becomes less intrusive. Using a metric of 3 percent as the tolerable sampling overhead, **MT** can sample every 500 cycles, **Stall** can sample every 1,000 cycles, and **Swap** can sample every 3,600 cycles. As in the previous experiments, these results show that the most important aspect of concurrent event handling is to reduce the event overhead. Furthermore, to

TABLE 4
Effect of Events on Application Execution Time

Event Model	Execution Time (Cycles)
MAP	2.8M
Stall	4.0M
Swap	4.9M

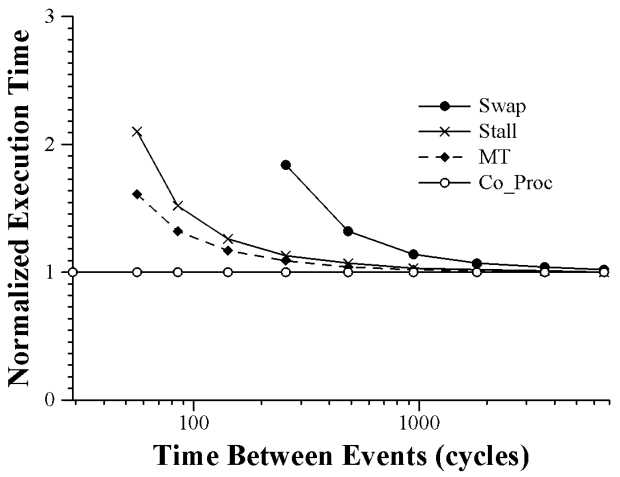

Fig. 10. Impact on the execution time of an application during dynamic profiling.

achieve high sampling rates with minimal impact on the runtime of the application requires the event and the application to execute in parallel.

4.4 Summary

The experimental results in this section show the benefits of concurrent event handling in microbenchmarks and in applications that have high event frequencies. The first component of concurrent event handling is the elimination of the high software overhead associated with stopping the application and replacing it with the event handler. In the software shared memory example, this improved the application's performance by 19 percent. With a rate of one sample every 500 cycles in the Dynamic Profiler, performance also improves by 19 percent. Executing events concurrently with application is also critical, with shared memory improving an additional 30 percent and the profiler improving 4 percent. Future computer systems that intend to support event rates on par with those shown in the experiments (30-500 cycles between events) will need to rely on concurrent event mechanisms to reduce event overhead.

5 Related Work

In addition to the block [1] and round-robin multithreading [2], [26], two other multithreaded architectures are of note. Like the MAP, Interleaving schedules multiple threads over a single processor without requiring a thread switch on every cycle [18]. Since pipeline registers are not replicated, events that require a thread to stall, such as a cache miss, must squash instructions that have already been issued into the pipeline. Simultaneous Multithreading extends an out-of-order superscalar architecture to execute instructions from multiple threads at the same time [30]. SMT uses a traditional exception model and does not execute interrupts at the same time as user threads. However, the extensions described in this paper can be incorporated into a dynamically scheduled multithreaded architecture in a straightforward manner.

Other architectures and machines have provided hardware support for asynchronous event handling, typically in the framework of a message-passing multiprocessor. The J-Machine's Message Driven Processor (MDP) implements one hardware context for background tasks and two contexts for message handling [8]. When a message arrives, the task running in the background context suspends (stalls) and the incoming message invokes a thread to handle the contents of the message. The *T architecture [25] and the FLASH multiprocessor [17] use coprocessors to process incoming messages and to communicate with the master computation processors.

Thekkath and Levy observed that many exceptions need to be handled in user code and that standard event architectures impose a large overhead jumping into and out of the kernel [28]. Consequently, they proposed a hardware mechanism for vectoring events directly to user level code without entering the kernel. In the absence of hardware support, they also examined methods which limit the kernel overhead by detecting a user level event and vectoring directly to it. As a result, they were able to reduce the latency of a null user-handled event from 1,225 to 125 cycles on a 25 MHz R3000 workstation running Ultrix. An even more lightweight form of user-level events is Informing Memory Operations [12]. This event architecture can trigger a fast user specified and user handled trap on a cache miss. The executing program determines which memory operations are enabled for traps. When a cache miss occurs, the trap is handled in the faulting program's register set. The hardware trap handling can be folded into existing branch or event hardware in an out-of-order superscalar microprocessor. While this eliminates full context switching (values in registers can be saved by the trap handler as necessary), it may still incur the hardware execution overhead required by precise interrupts.

6 Conclusion

In this paper, we introduce *concurrent event handling*, an architecture that increases event performance through multithreading. When an event occurs, the handler is forked into the dedicated thread slot and can execute simultaneously with other programs that are already running. Concurrent event handling has three advantages over traditional event handling. First, dedicating a thread slot in a multithreaded processor to an event handler eliminates the need to stop the faulting thread. The faulting thread state remains in the pipeline registers and none of the nonfaulting instructions need to be nullified. Second, replicating register files for multiple threads eliminates the software overhead that normally is required to remove and restart a faulting thread. Finally, greater thread level parallelism can be achieved by running the event handler concurrently with the faulting threads and other event handlers. As microprocessors are able to issue more instructions per cycle, the number of wasted issue slots (due to a lack of sufficient instruction level parallelism) is increasing. Interleaving execution of event handlers and an application will help improve program throughput and reduce overall execution time.

Many modern microprocessors implement sequentially precise interrupts to enforce a strict order of instructions in a faulting thread when it is halted and replaced by the event handler [27]. However, in multithreaded processors, this conservative approach can unnecessarily hinder performance. Instead, concurrent event handling with precise events maintains consistency while improving performance, as the original thread continues to execute until it reaches a true dependency on the result from the faulted instruction. While the faulting thread proceeds, all necessary information required to resolve the fault is captured and provided to the event handler.

Using a combination of microbenchmarks and applications with high event rates, we demonstrate the advantages of concurrent event handling over more traditional event architectures. Without the software overhead of replacing the faulting thread with the event thread, the latency of a null internal event drops from 100 cycles to 14 cycles. Our sample applications, software shared memory and dynamic profiling, have events that occur on average every 30-500 cycles. In this environment, the reduction in event overhead improves application performance by 19 percent.

The additional advantage of running event handlers in parallel with application code provides a total execution time reduction of 26-43 percent.

Concurrent event handling has been incorporated into the MIT Multi-ALU Processor, a 5 million transistor custom VLSI chip [14]. The MAP chip implements multithreading by replicating pipeline resources across the on-chip processor clusters to improve latency tolerance, function unit utilization, and overall performance. Incorporating concurrent event handling requires only a small amount of additional real estate for the message and event queues, less than 6 percent of the total chip area. While the MAP chip issues instructions in-order, concurrent event handling and the precise event model are applicable as well in out-of-order processors and can simplify the hardware required to maintain consistent state during synchronous and asynchronous events.

ACKNOWLEDGMENTS

The research described in this paper was supported by the U.S. Defense Advanced Research Projects Agency under ARPA order 8272 and monitored by the U.S. Air Force Electronic Systems Division under contract F19628-92-C-0045. Thanks also to Nick Carter for providing the infrastructure to run the software-based shared memory benchmark application.

REFERENCES

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* pp. 104-114, 1990.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," *Proc. Int'l Conf. Supercomputing,* pp. 1-6, June 1990.

[3] J.M. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T.A. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W.E. Weihl, "Continuous Profiling: Where Have All the Cycles Gone?" *Proc. 16th Symp. Operating Systems Principles,* pp. 1-14, Oct. 1997.

[4] 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-IV),* pp. 108-120, Apr. 1991.

[5] A. Appell, J. Ellis, and K. Li, "Real-Time Concurrent Collection on Stock Multiprocessors," *Proc. 1988 SIGPLAN Conf. Programming Languages Design and Implementation,* pp. 11-20, June 1988.

[6] N.P. Carter, "Processor Mechanisms for Software Shared Memory," PhD thesis, Massachusetts Inst. of Technology, Dept. of Electrical Eng. and Computer Science, 1999.

[7] N.P. Carter, S.W. Keckler, and W.J. Dally, "Hardware Support for Fast Capability-Based Addressing," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS VI),* pp. 319-327, Oct. 1994.

[8] W.J. Dally, J.S. Fiske, J.S. Keen, R.A. Lethin, M.D. Noakes, P.R. Nuth, R.E. Davison, and G.A. Fyler, "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," *IEEE Micro,* vol. 12, no. 2, pp. 23-39, Apr. 1992.

[9] J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos, "Profileme: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 292-302, Dec. 1997.

[10] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee, "The M-Machine Multicomputer," *Proc. 28th Int'l Symp. Microarchitecture,* pp. 146-156, Dec. 1995.

[11] J.L. Hennessy and D.A. Patterson, *Computer Architecture a Quantitative Approach.* San Mateo, Calif.: Morgan Kaufmann, 1990.

[12] M. Horowitz, M. Martonosi, T.C. Mowry, and M.D. Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors," *Proc. 23th Int'l Symp. Computer Architecture,* pp. 260-270, May 1996.

[13] S.W. Keckler, "Fast Thread Communication and Synchronization Mechanisms for a Scalable Single Chip Multiprocessor," PhD thesis, Massachusetts Inst. of Technology, Dept. of Electrical Eng. and Computer Science, 1998.

[14] S.W. Keckler, W.J. Dally, A. Chang, N.P. Carter, and W.S. Lee, "The MIT Multi-ALU Processor," *Proc. Hot Chips IX,* pp. 1-8, Aug. 1997.

[15] S.W. Keckler, W.J. Dally, D. Maskit, N.P. Carter, A. Chang, and W.S. Lee, "Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor," *Proc. 25th Int'l Symp. Computer Architecture,* pp. 306-317, June 1998.

[16] E.J. Koldinger, J.S. Chase, and S.J. Eggers, "Architectural Support for a Single Address Space Operating System," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV),* pp. 175-186, Oct. 1992.

[17] J. Kuskin et al., "The Stanford FLASH Multiprocessor," *Proc. 21st Int'l Symp. Computer Architecture,* pp. 302-313, Apr. 1994.

[18] J. Laudon, A. Gupta, and M. Horowitz, "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations," *Proc. Sixth Int'l Conf.n Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI),* pp. 308-318, Oct. 1994.

[19] R. Lee, "Precision Architecture," *Computer,* vol. 21, no. 1, pp. 78-91, Jan. 1989.

[20] W.S. Lee, W.J. Dally, S.W. Keckler, N.P. Carter, and A. Chang, "Efficient, Protected Message Interface in the MIT M-Machine," *Computer,* vol. 31, no. 11, pp. 69-75, Nov. 1998.

[21] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems,* vol. 7, no. 4, pp. 321-359, Nov. 1989.

[22] P.G. Lowney, S.G. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg, "The Multiflow Trace Scheduling Compiler," *The J. Supercomputing,* vol. 7, nos. 1-2, pp. 51-142, May 1993.

[23] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," *Proc. USENIX 1996 Ann. Technical Conf.,* pp. 279-294, Jan. 1996.

[24] J.K. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proc. Summer 1990 USENIX Conf.,* pp. 247-256, June 1990.

[25] G.M. Papadopoulos, G.A. Boughton, R. Grainer, and M.J. Beckerle, "*T: Integrated Building Blocks for Parallel Computing," *Proc. Supercomputing 1993,* pp. 624-635, 1993.

[26] B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *SPIE Vol. 298 Real-Time Signal Processing IV,* pp. 241-248. Aurora, Colo.: Denelcor, Inc., 1981.

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

[28] C.A. Thekkath and H.M. Levy, "Hardware and Software Support for Efficient Exception Handling," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI),* pp. 110-119, Oct. 1994.

[29] R. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J.,* vol. 11, pp. 25-33, Jan. 1967.

[30] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 392-403, May 1995.

[31] N. Ullah and M. Holle, "The mc88110 Implementation of Precise Exceptions in a Superscalar Architecture," *Computer,* vol. 21, no. 1, pp. 15-25, Jan. 1989.

[32] R. Wahbe, "Efficient Data Breakpoints," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV),* pp. 200-212, Oct. 1992.

[33] W. Walker and H.G. Cragon, "Interrupt Processing in Concurrent Processors," *Computer,* vol. 28, no. 6, pp. 36-46, June 1995.

[34] W.-D. Weber and A. Gupta, "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results," *Proc. 16th Ann. Int'al Symp. Computer Architecture,* pp. 273-280, 1989.

[35] K.C. Yeager, "The mips r10000 Superscalar Microprocessor," *IEEE Micro,* vol. 16, no. 2, pp. 28-40, Apr. 1996.

Stephen W. Keckler received the BS degree in electrical engineering from Stanford University in 1990, and both the MS and PhD degrees in computer science from the Massachusetts Institute of Technology in 1992 and 1998, respectively. From 1989 to 1990, he worked at Intel designing circuits for the 386SL microprocessor. He is currently an assistant professor of computer science at the University of Texas at Austin. His research interests include computer architecture, parallel and embedded processors, VLSI design, and the relationship between VLSI technology and computer architectures. He is a member of Phi Beta Kappa, Tau Beta Pi, Sigma Xi, the IEEE, and the ACM.

Andrew Chang received the SB degree in 1988 and the SM degree in 1998, both in electrical engineering, from the Massachusetts Institute of Technology. As an engineering group leader at Loral Fairchild Imaging Sensors from 1989 to 1990 and the principal electrical engineer at AOI Systems from 1990 to 1992, he designed CCD imaging systems for automated optical inspection. He is currently a PhD student in the Concurrent VLSI Architecture Group at Stanford University. His research interests include CAD, computer architecture, and VLSI circuit design. He is a student member of both the IEEE and the ACM.

Whay S. Lee received the MS degree in 1994 and the PhD degree in 1999, both in computer science from the Massachusetts Institute of Technology. He is working on architectures for enterprise storage solutions as a member of the Network Storage Division at Sun Microsystems. His research interests include computer architecture, multicomputer networks, and fault-tolerant computing.

Sandeep Chatterjee received the BS degree in electrical engineering and computer science from the University of California at Berkeley in 1995 and the MS degree in electrical engineering and computer science from the Massachusetts Institute of Technology in 1997. He is currently a doctoral candidate at MIT's Laboratory for Computer Science. His research interests include computer architecture, low-power design, and intelligent systems. He is a member of Eta Kappa Nu, Tau Beta Pi, the IEEE, and the ACM.

William J. Dally received the BS degree in electrical engineering from Virginia Polytechnic Institute, the MS degree in electrical engineering from Stanford University, and the PhD degree in computer science from the California Institute of Technology. Dr. Dally and his group have developed system architecture, network architecture, signaling, routing, and synchronization technology that can be found in most large parallel computers today. While at Bell Telephone Laboratories, he contributed to the design of the BELLMAC32 microprocessor and designed the MARS hardware accelerator. He was a research assistant and then a research fellow at Caltech, where he designed the MOSSIM Simulation Engine and the Torus Routing Chip which pioneered wormhole routing and virtual-channel flow control. While a professor of electrical engineering and computer science at the Massachusetts Institute of Technology, he and his group built the J-Machine and the M-Machine, experimental parallel computer systems that pioneered the separation of mechanisms from programming models and demonstrated very low overhead mechanisms for synchronization and communication. He has worked with Cray Research and Intel to incorporate many of these innovations in commercial parallel computers and with Avici Systems to incorporate this technology into Internet routers. Dr. Dally is currently a professor of electrical engineering and computer science at Stanford University, where he leads projects on high-speed signaling, multiprocessor architecture, and graphics architecture. He has published more than 80 papers in these areas and is an author of the textbook, *Digital Systems Engineering*.