A Case for User-Level Interrupts

Mike Parker
School of Computing, University of Utah
map@cs.utah.edu

1 The Problem

Interrupt overhead for high-speed I/O devices can have a significant negative impact on system performance. Often, the sole purpose of an interrupt from an I/O device is to notify a particular user process about the completion of a request or the arrival of a message from a peer process. General-purpose machines do not provide a mechanism for I/O to directly deliver the notification. The kernel is responsible for acknowledging the interrupt and delivering a notification to the appropriate user process. The act of taking the interrupt, determining the proper destination process, and delivering the notification can be costly in both the number of cycles spent and in the cache overhead incurred.

Gigabit ethernet without interrupt coalescing can produce on the order of 5-6 million packets per second for maximum sized packets. Minimum sized packets may produce 10 to 20 times as many interrupts per second. This implies that for a high traffic gigabit ethernet, if the CPU does no other work than processing interrupts, it has somewhere between 10 to 200ns, or 20 to 400 cycles at 2 GHz to handle each interrupt. For clusters using a high-speed network interface, packets may arrive at a phenomenal pace. The overhead of handling each packet severely limits scalability. User-level I/O devices (primarily network interfaces) have reduced other aspects of the I/O bottleneck, leaving event notifications in the form of interrupts as the critical remaining component.

Most of the time dealing with interrupts is spent servicing cache misses[3]. This overhead worsens as the memory gap widens. Coalesced interrupts and efficient interrupt handlers can significantly reduce this overhead. However, the time spent in servicing operating system induced cache misses remains a dominant part of the interrupt overhead. Providing mechanisms to deliver interrupts directly to the user process without kernel involvement in the common case can significantly reduce the overhead of I/O event notification. Frequently, the I/O subsystem can determine the destination process of an interrupt using small and simple lookup tables. The basic work of dispatching interrupts to user-level processes is, in this common case, pushed off to the I/O hardware.

2 User-mode Interrupts

From the machine's point of view, current kernel-mode interrupts appear as unanticipated traps into the kernel. Often, the true destination of the interrupt is a user-level process. Under Unix this interrupt eventually manifests itself as a kernel-induced "surprise" branch into a user-level signal handler. I/O hardware that supports user-level access could easily be extended to compute the destination process for an interrupt being generated. This hardware could exploit knowledge of the destination process to drastically reduce interrupt overhead if the processor were extended to allow interrupts to be delivered directly to the user process.

On a conventional single-threaded processor, this user-level interrupt could be implemented as an "asynchronous" branch. If an I/O device posts an interrupt for a running process, the program counter is redirected to an alternate location. This alternate program address could either be in a special register in the processor, or could be provided by the I/O device via the interrupt mechanism. Program execution would thus be diverted to a user defined interrupt handler, and the user process would be responsible for handling the interrupt.

For more recent multi-threaded processors, the asynchronous branch mechanism would work, but other options are also viable. For example, a blocked thread in one of the contexts could wait for an interrupt using a wait-for-interrupt instruction. This thread would be made runnable again as a side effect of the interrupt. That thread would then handle the interrupt. In another model, a new thread could be created on the fly in either an empty context, or by evicting a lower priority context. The new thread would have minimal initial state set by the interrupt transaction, and would be responsible for handling the interrupt.

Under both of these processor models, the I/O subsystem would provide a destination tag, such as a process ID, to the processor with the interrupt request. A bit would be set in a privileged register

if an interrupt was not successfully delivered. This might occur when the running process does not match the interrupted process, when a spare thread context is not available to execute the interrupt handler, or under other similar circumstances. The operating system could then reap that information during the next context switch. If the I/O subsystem runs into a critical situation that demands immediate attention, it can still interrupt the kernel directly. Figure 1 illustrates how user-level interrupts may be more efficient than traditional kernel interrupts. In a conventional system, several hundred cache misses are induced by the kernel interrupt handler[3]. Once the interrupt is delivered to the user process via a signal, the user process executes its local handler and then returns back through the OS. Direct user-level interrupts would not only eliminate the kernel cache miss overhead, but would also eliminate the additional cache misses the user process incurs to reload the working set displaced by the kernel's interrupt dispatch and signal return code.

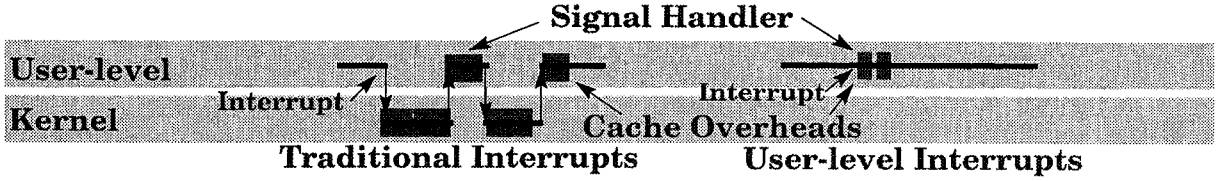

Figure 1: Traditional interrupts vs. User-level Interrupts

3 Synopsis of Current Work

User-level interrupts could benefit many types of intensive I/O devices, most notably system area network interfaces. High-speed system area networks are a popular method of connecting processors in a cluster organization. System networks are also starting to get attention for more general I/O interconnects, as is evidenced by InfiniBand[5] and Motorola's RapidIO[6].

The user-level interrupts described in this paper are part of a larger work to investigate a full system approach to I/O and multiprocessing by providing an efficient user-level system level network interface that is tightly coupled with the CPU. In this larger work, there is also a focus on using a simultaneous multi-threading (SMT) processor[1] to hide and tolerate remaining I/O overheads [3]. Early results indicate that the cost of a message notification can be reduced by more than an order of magnitude without giving up Unix-style security. This significant reduction in overhead makes it possible for fine-grained message programs to scale efficiently on message-passing systems.

LRSIM[4], a simulator based on RSIM[2], is being extended to accurately model an SMT processor and a system area network interface. LRSIM adds to RSIM accurate I-cache, memory and I/O architecture models. The simulator includes a fairly complete NetBSD based kernel that will be extended to handle the SMT processor. For evaluation, a 2-5 GHz 4-8 thread SMT that can issue 8-16 instructions per cycle will be modeled. L1 instruction and data caches will be 32KB to 128 KB each, and the L2 Cache will be 4-16MB. The system network bandwidth modeled will range from 4Gb/s to 32Gb/s. Disk controllers, LAN interfaces, and other I/O devices will hang off the system network.

References

[1] Susan J. Eggers, et al. Simultaneous Multithreading: A Platform for Next-Generation Processors. IEEE Micro, 17(5):12-19, October 1997.

[2] V. S. Pai et al. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In Proceedings of the 3rd Workshop on Computer Architecture Education, 1997.

[3] Mike Parker, Al Davis, Wilson Hsieh, Message-Passing for the 21st Century: Integrating User-Level Networks with SMT, In Proceedings of the 5th Workshop on Multithreaded Execution, Architecture, and Compilation, 2001.

[4] Lambert Schaelicke. Architectural Support for User-Level I/O. Ph.D. Dissertation, University of Utah, 2001.

[5] InfiniBand Trade Association. http://www.infinibandta.org.

[6] Motorola Semiconductor Product Sector. RapidIO: An Embedded System Component Network Architecture. February 22, 2000.