On the Integration of Real-time Asynchronous Event Handling Mechanisms with Existing Operating System Services

Gerald Fry and Richard West
{gfry, richwest}@cs.bu.edu
Computer Science Department, Boston University

Abstract

This paper presents an asynchronous event handling mechanism for real-time applications that leverages existing system services in COTS systems. In our implementation, event dispatching is initiated within bottom-half *interrupt handling routines, in order to support predictable, safe, and efficient event handling functionality at user-level. We compare our asynchronous I/O mechanism with existing user-level approaches, such as the GNU C library implementation of the AIO API and the Linux signal abstraction. Using a user-level sandboxing scheme for asynchronous event handling, results show that network receive events can be dispatched in less than* 15 *microseconds using commodity hardware components. We show that the proposed architecture significantly outperforms the above mentioned user-level asynchronous I/O mechanisms and provides more flexibility than available hard real-time extensions.*

Keywords: real-time, COTS, asynchronous I/O, event dispatching

1. Introduction

Recent research in real-time systems seeks to extend COTS systems with support for predictable and efficient resource allocation for real-time and best-effort tasks. Traditionally, operating systems supporting execution of tasks with hard deadlines are developed in a fashion that trades overall system throughput for a desirable degree of predictability. Although such systems work well in specific scenarios in which the set of real-time tasks and their execution characteristics are known a priori, the concurrent execution of best-effort tasks along with real-time activities is not well supported by these systems.

Typical general purpose operating systems, such as Linux, do not support predictable resource allocation for real-time tasks. Real-time tasks that are posted in response to I/O events on file or socket descriptors, for example, are delayed until the original requesting process is dispatched by a best-effort scheduler, resulting in a non-deterministic worst-case execution time of the real-time event handler. In addition, when responding to I/O requests in the form of system calls, the kernel may disable local interrupts to perform necessary actions on behalf of the calling applications. In this scenario, interrupt processing for real-time tasks may be delayed due to execution of kernel code on behalf of other applications. However, in order to provide deadline-based execution guarantees, it is necessary to dispatch event handlers associated with real-time tasks asynchronously, without interference from the (non-real-time) system scheduler or other tasks with lower or best-effort priority.

Solutions are available for integrating real-time task execution with general purpose COTS systems, including RTLinux, RTAI, and Montavista [7, 9, 3]. Standards such as the POSIX real-time signal API describe mechanisms for integrating real-time IPC support into compliant operating systems [5]. Many such solutions approach the problem of real-time integration into existing general purpose systems by deferring interrupt handling for best-effort tasks until all outstanding events are processed by real-time applications. IRQs associated with real-time tasks are vectored directly to interrupt handling routines that respond within the time necessary to dispatch the interrupt service routine, and real-time code is executed in interrupt context.

Although this method results in low event dispatching latency, executing real-time event handling code within an interrupt context comes with several disadvantages, including **(1)** the inability to isolate tasks using traditional process protection mechanisms, **(2)** lack of support for shared user-level libraries and system calls in event-handling code, and **(3)** poor integration of real-time task scheduling with best-effort resource allocation policies, possibly resulting in degraded performance for co-existing best-effort processes.

Most modern operating systems divide interrupt processing into top-half and bottom-half interrupt handling routines. Device drivers typically implement an interrupt service routine that is dispatched directly upon receipt of an associated IRQ. This procedure performs only the operations necessary to communicate with the controller on the corresponding device that initiated the interrupt, as well as

acknowledging the interrupt and posting a pending event to indicate subsequent "deferrable" processing. The rest of the processing associated with the interrupt can then take place at a later time in a *bottom-half* handler. In Linux, for example, "soft irqs" are implemented to perform deferrable interrupt handling functions. † Such functions may be executed with interrupts enabled and are invoked when a hardware interrupt handler exits, or when a system call is about to return to user-space.

This work presents an asynchronous I/O event handling mechanism that enables user-level code to be executed in response to events as they are generated within the context of *bottom-half* interrupt service routines. The mechanisms provided in this paper enable the predictable execution of event handling code at user-level, with acceptable dispatch latency when compared with approaches that rely on scheduling and dispatching process address spaces. An API for registering real-time event handling code, that is executed at the user-level processor protection level upon dispatch of the event, is defined, implemented, and analyzed for effectiveness. Experimental analysis includes comparison of the performance of network communication leveraging asynchronous I/O event-handling approaches including SIGIO, aio_read/aio_write, and the techniques presented in this paper.

The next section describes our real-time event model and reviews our prior work on "user-level sandboxing" techniques. The implementation of an asynchronous real-time event handling mechanism for processing network packets is detailed in Section 3, followed by experimental analysis comparing the effectiveness of this approach with other methods, given in Section 4. Section 5 summarizes related work on event-handling in real-time systems, and conclusions as well as directions for future work are discussed in Section 6.

2 Event Model

A crucial performance requirement for real-time task execution involves *event dispatch latency*, or the period of time between the occurrence of an event (as an IRQ), and the beginning of the execution of the event handler code. The dispatch latency, δ_e, of an event e is described by the formula: $\delta_e = ID + IE + BHD$, where ID refers to the time taken for the hardware and underlying subsystem to vector an interrupt request to a top-half interrupt service routine, IE denotes the execution time of the the associated ISR, and BHD is given by the time between the exit of the ISR and the beginning of execution of an associated bottom-half handler, including any processing that must be done to set up a context for executing the real-time task instance.

Deterministic event dispatch latency is important for predictable real-time service, thus it is desirable to minimize the dispatch latency, δ_e, for each event e. However, in systems supporting highly dynamic process sets, each of the components of the dispatch latency, ID, IE, and BHD, depend upon the non-deterministic state of the system at the time of the event. Specifically, the latency of dispatching an IRQ to an interrupt service routine depends upon the priorities of all interrupts waiting to be serviced at a given time, since a lower priority request will be deferred until all higher priority IRQs are serviced. Moreover, if an ISR is currently executing and an interrupt request of higher priority arrives at the processor, the original executing routine may be preempted to serve the higher priority request. When an event handler is deferred for execution from within a bottom-half context, the delay associated with dispatching the bottom-half handler is dependent upon other pending bottom-half functions and interrupt requests.

This paper investigates two methods of dispatching real-time event handlers. An *event hook*, or an operation that dispatches an event handler, may be called from within *hard interrupt* context, or alternatively, from within *soft interrupt* context, as described below. *Hard interrupt context:* The event hook is placed within a top-half interrupt service routine and begins execution of the real-time task instance just before the ISR exits, but after all necessary device operations are completed. In this scenario, the dispatch latency of the corresponding event depends only upon the interrupt dispatch and execution terms, ID, and IE, in the formula for δ_e, that is, $BHD = 0$. *Soft interrupt context:* An event handler is dispatched by an event hook placed within a bottom-half handler for the corresponding event. In this case, the total event dispatch latency is dependent upon the delay incurred between the end of the ISR and the beginning of the bottom-half handler execution.

Hard real-time applications require complete predictability for executing task sets according to their associated deadlines. Supporting such tasks requires a priori knowledge of the interaction of events that may occur in the system, thus resulting in a deterministic event dispatching latency, δ_e for all events e. The actual dispatch latency incurred determines the time granularity at which real-time parameters, such as deadlines and start times, can be specified. Execution of real-time task instances directly from within top-half ISRs has the advantage of low dispatch latency, but this method suffers disadvantages in extensibility and system integration that become alleviated when event handlers are executed during bottom-half (deferrable) processing. The extensibility and functionality issues that arise

†We use the term *bottom-half* to refer to a deferrable function that executes as a consequence of a prior interrupt. Linux has deprecated the term *bottom-half* but nonetheless deferrable functions exist in the form of *softirqs* and *tasklets*. We will use the term *bottom half* to refer to a deferrable function throughout the paper unless we wish to explicitly qualify a Linux softirq or tasklet.

when deciding the context for event handler execution include the following:

System integration: Much of the processing related to device I/O event handling is executed within the bottom-half interrupt handlers implemented in device driver modules. In Linux, for example, a bottom-half routine is responsible for such tasks as analyzing the header of an incoming network packet, determining the destination socket for which the packet data is buffered, and signaling to wake up processes waiting on network requests when data is available. In this case, the network driver code is complex, but has been optimized over time for good performance. Real-time application developers may benefit significantly from having access to the services of the existing system, such as an operational and efficient network stack. Although executing real-time event handlers in hard interrupt context reduces overall dispatch latency, the programmer cannot assume, in this case, that the existing device driver architecture and interfaces will be available for predictable service. Instead, an event that is handled in hard irq context may only be bound to a specific interrupt request line that cannot be shared with other non-real-time tasks or services. In contrast, deferring event handling to a bottom-half context allows for a more sophisticated mapping between events and their associated handler tasks.

Programming convenience: It is assumed that a programmer implementing real-time event handling tasks is comfortable with the services and APIs provided by the system and would like to leverage these services in writing real-time code. The development process is more efficient if the programmer can use existing I/O abstractions, such as sockets and file descriptors, instead of being required to import or write specialized device controlling code to program I/O activities. This sort of API integration is easily obtained if tasks' event handlers are activated from a bottom-half ISR, since an event can be defined as a condition on a system abstraction (i.e., a socket) rather than be limited to association with a specific interrupt request line.

Event scheduling: Real-time tasks executing in top-half ISR context may arbitrarily delay execution of bottom-half procedures, and the same applies to event handler execution from soft interrupt context. However, the IRQ priority and execution relationship are (at least partially) controlled by the hardware IRQ scheduling mechanisms, whereas soft interrupt execution is fully controllable in software, allowing synchronization of bottom-half routines based upon a wide variety of scheduling policies and priority models.

A series of timing measurements were collected in order to indicate possible relationships between the various components of event dispatching latency. Modifications were made to a Linux 2.6.15 kernel running on a 2.4 GHz Pentium 4 processor. The first experiment records the value of the time stamp counter just after a timer IRQ is raised and subtracts this value from the next time stamp counter read, just after the actual top-half handler function begins execution. The computed difference measures the number of cycles required to dispatch the IRQ request to an appropriate ISR for processing, and is thus an estimate of the interrupt dispatch latency term, ID, in the formula for δ_e. The value of IE, or the interrupt execution latency, is determined by subtracting the previous counter value from the value recorded after the ISR exits. The next time stamp counter read is performed just after the timer bottom-half handler function begins execution, allowing for the calculation of the bottom-half delay term, BHD.

	Mean	Median	Stdev
ID	6311.16	6060	559.94
IE	10958.15	10740	779.26
BHD	308.04	252	1233.26
δ_e	17577.35	17048	1628.05

Figure 1. Statistical measurements of values for ID, IE, BHD, and δ_e (in processor cycles) for timer events.

The data in Figure 1 indicate the mean, median, and standard deviation over one thousand trials of the experiment described above (one trial per timer interrupt). Each statistical value is given for each of the terms ID, IE, and BHD as estimated in the above discussion. Results are also shown for the total event dispatch latency, δ_e, incurred. The average event dispatch latency in the experiment is calculated based on the processor clock speed and the cycles recorded in Figure 1, and the resulting time to dispatch a timer event handler from soft interrupt context is approximately 7.3 microseconds.

The latencies shown in Figure 1 infer that the time between the exit of the top-half ISR and the beginning of execution of the timer bottom-half is typically much smaller than the combined time needed to dispatch and execute the top-half. Since the timer bottom-half runs at higher priority than other soft irq functions, and since the timer soft irq is scheduled to run just after the ISR returns, most values for BHD in this case are only a fraction of the total event dispatching latency, δ_e. Thus, the experiment described above is a measure of the best case delays for dispatching events from within soft interrupt context. However, it should be noted that the variance of bottom-half dispatching delay is still higher than that of the hard interrupt dispatch latency and execution time, since the bottom-half handler may be delayed when top-half handler instances are executing.

The second timing experiment measures dispatching and execution latencies related to network processing. The table in Figure 2 summarizes 1000 measurements of the combined top-half interrupt and execution latencies resulting

	Mean	Median	Stdev
$ID + IE$	9489.14	9108	1229.95
BHD	17769.23	13320	24705.58
δ_e	27258.37	22692	24883.65

Figure 2. Statistical measurements of values for $ID+IE$, BHD, and δ_e (in processor cycles) for network events.

from processing small incoming packets on an Ethernet device, as well as the associated delays in dispatching the network receive bottom-half. The endpoint measurements for the bottom-half associated delays are taken at the beginning of the execution of a callback function, which is automatically called by the network subsystem upon arrival of data at the socket. The callback function is executed in soft interrupt context and is called from within a bottom-half handler associated with network receive events.

As seen in the results, the values for BHD are somewhat greater than $ID + IE$, which is partially due to the fact that the network soft irq function may be delayed by timer interrupts and associated timer soft irq instances. An extra delay is also incurred by computation in the network stack, including maintenance of the socket data structures and protocol processing. Furthermore, the variance of the network soft irq dispatching delays is high in comparison to the hard irq case, for similar priority-related reasons. However, the total delay between the start of IRQ dispatching and the beginning of the socket callback function execution is approximately 11.4 microseconds on average, which may be acceptable for many real-time applications that are dependent upon statistical delay guarantees.

The above results provide evidence that delaying real-time event handling to soft interrupt context may be feasible from a performance perspective, while providing the ability to leverage the infrastructure of existing system services (i.e., the network stack). In this example, an event is defined with respect to a particular socket data structure, which requires information that is not available until some bottom-half processing is completed by the underlying network subsystem. Consequently, the same mapping between events associated with individual sockets and corresponding real-time code is not possible when dispatching directly from hard interrupt context unless an independent and specialized network subsystem is written specifically for use by real-time event handlers.

User-level Sandboxing: Although executing events in soft interrupt context may result in desirable integration with existing system services, as well as an acceptable dispatch latency, direct execution of real-time extension code at the highest processor protection level circumvents the usual protection semantics of application address spaces. In order to protect memory associated with real-time tasks, a user-level sandbox abstraction is leveraged for setting up a user-space context in which to *safely* execute event handling extensions [21].

The user-level sandboxing mechanism allows application or system extension code to be loaded into a 4MB page frame. All data associated with such extensions, including a user-level stack for each module, is maintained within an additional super-page in memory. The sandbox virtual pages are pinned in memory and the associated virtual addresses are mapped into the address spaces of each process.

Page table entries mapping the sandbox regions are initially set to be accessible only when the processor is executing in kernel mode. Upon arrival of an event that is associated with a registered sandbox thread, the permissions associated with the sandbox page mappings are modified by the kernel to allow access at user-level. Control is subsequently transferred from within the kernel to code within the appropriate sandbox page via an *upcall* to user space. The registered event handling extension is executed in the context of the process that is running at the time the event is raised as an IRQ, but a separate stack is maintained within the sandbox data region for use by the event handling thread. After completion of the thread, the access permissions for the sandbox pages are reset to supervisor mode and the corresponding page translations are flushed from the TLB. The kernel then resumes execution just after the point at which the upcall is initiated.

Hardware protection mechanisms may be used to protect memory associated with sandbox threads from code running in the context of process-private address spaces, but in this scheme sandbox code may arbitrarily access memory reserved for a process that just happens to be running during dispatch of the event handling extension. To alleviate the resulting protection problem, the sandbox mechanism may require event handling threads to be implemented using a type-safe language, thus leveraging language-level protection domains for ensuring that traditional process address spaces are protected from the activities of real-time event handlers.

In addition to the protection features mentioned above, user-level sandbox event processing has the advantage of allowing use of shared library routines in programming extension threads. For instance, a small version of the standard C library may be loaded into the sandbox region and shared among various event handlers requiring access to functions such as system call wrapper routines. The resulting functionality greatly simplifies the development of real-time event handlers that need access to typical system APIs and widely used application-level library functions.

3 Implementation

This section describes the implementation of an asynchronous real-time I/O event handling mechanism designed to efficiently dispatch application code. Although the focus of this paper is predictable execution of real-time event handling functions, the proposed mechanism is also applicable to best-effort applications that need to service events in an asynchronous manner. The following subsections provide background information on currently available asynchronous I/O mechanisms and detail the implementation of an alternative method for safe, low-latency event handler registration and dispatching.

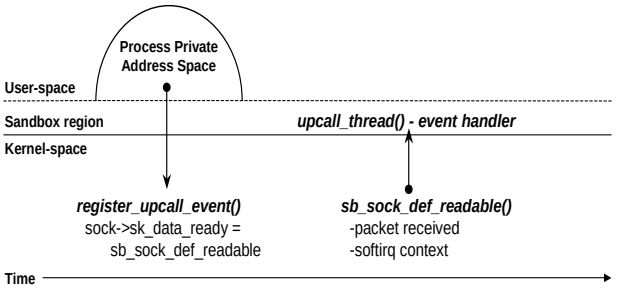

Figure 3. Event handling control flow.

3.1 Background

There are several existing methods that allow an application thread to overlap I/O operations with computation. One such method leverages an implementation of the POSIX AIO interface for efficient asynchronous I/O access. As the standard relates only to the API semantics, the implementations among various systems may differ in performance. For example, as of version 2.6, the Linux kernel supports a subset of the AIO interface by using pinned pages and wait queues associated with completion ports. In systems that do not include direct kernel support for AIO, a user level library may implement the API by spawning new threads or registering signal handlers, each of which responds to I/O events on a separate file descriptor. The AIO interface includes the following functions:

aio_read()/aio_write() - submit an I/O request to read/write to/from a file descriptor.

aio_return() - obtain the return status of a pending I/O operation.

aio_suspend() - block until an I/O operation has completed. Multiple I/O requests may be passed to this function and the process is placed on a wait queue until at least one has completed.

A thread that unblocks after a call to aio_suspend() may need to wait until it is next scheduled before an application-level response to the completion event can be initiated. Due to the nature of scheduling in Linux, multiple context switches (and consequent TLB flushes) may occur between the occurrence of an I/O event and the execution of the event handling code. Alternatively, a Linux thread may asynchronously call aio_return() at strategic points within its execution to test the status of pending operations, requiring the process to trap to the kernel each time a request on I/O status is invoked.

An alternative to the AIO mechanism described above involves the registration of a signal handler that is to be invoked upon the occurrence of an event on a file descriptor (i.e., new connection, data ready, etc.). After initialization of the file descriptor, the process issues system calls to inform the kernel that requests pertaining to the file descriptor should not block the process, but rather wake up the process and invoke the signal handler whenever the I/O operation is at least partially completed.

In order to support predictable event handling service for hard real-time applications, best-effort associated interrupts may be virtualized so that events destined for real-time tasks can be processed in hard interrupt context. This is the method used by systems such as RTLinux and RTAI. Although this mechanism provides very predictable and low-latency event handling functionality, a real-time task is executed at the most privileged processor protection mode, thus making it possible for buggy real-time code to undermine the integrity of the system. Furthermore, I/O processing in real-time for a given IRQ cannot co-exist with best-effort services that must use the same interrupt request line.

3.2 Asynchronous I/O Implementation

The asynchronous I/O mechanism proposed in this work attempts to provide event handling services that are predictable and well-integrated with existing system services available in Linux. The API includes a function, *register_upcall_event(int fd, char *upcall_module)*, which first loads a compiled upcall module specified by the filename parameter *upcall_module* into a sandbox memory area. The function then places an event hook in the kernel to invoke the module upon the occurrence of events associated with an open file descriptor, *fd*. When an event occurs on the file descriptor (i.e., data ready), control is transferred directly from softirq context to user-space for event processing. The sandbox code may be statically linked with a lightweight libc implementation, such as dietlibc, to allow the user-space event handling function to execute system calls and user-space library routines [1].

An event handling function is placed within a separate source file. The implementation ensures that this function is passed the file descriptor of interest and the number of

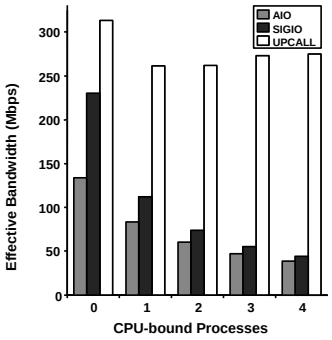

Figure 4. Effective bandwidth achieved in UDP stream transmission.

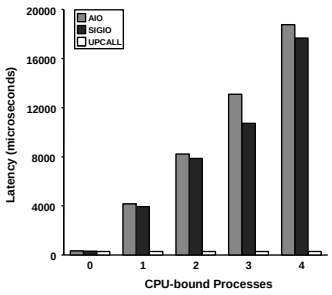

Figure 5. UDP message transmission round-trip latencies.

bytes available for reading[†] from a socket descriptor. The function is compiled to an object file and loaded into the sandbox code page for later invocation, and memory is allocated within the sandbox data page for storing the contents of a stack associated with the sandbox routine. Figure 3 illustrates the control flow associated with dispatching an event handler in response to a network packet arrival.

4 Experimental Analysis

A series of experiments were conducted to compare the effectiveness of initiating I/O completion operations from within soft interrupt context with other approaches. All experiments were conducted using Intel Pentium 4 2.4GHz platforms running Linux 2.6.15, patched for user-level sandboxing support (unless otherwise stated).

The first experiment measures the bandwidth of transmitting and receiving a UDP stream between two hosts, via Gigabit Ethernet. In each trial, a 4MB file is transmitted as a series of 512B messages from the *client* host to the *server* host. Each message is then returned from the server to the client. In the implementation, the client host uses blocking I/O, while the server host uses one of the following methods of asynchronous I/O:

SIGIO: The server application registers a signal handler for response to the SIGIO signal that is delivered upon arrival of each message. The actual invocation of the signal handler may occur after a call to the system scheduler. A return message is transmitted within the SIGIO signal handling routine.

GLIBC AIO: An implementation of the AIO POSIX standard API is leveraged from the GLIBC library to perform asynchronous I/O operations [2]. The server application submits a read request for each message with the option SIGEV_SIGNAL. A signal handler is registered to be invoked upon signal delivery.

Softirq Upcall: The server application registers a sandbox code module to be executed from within the network receive softirq handler. A callback function is registered with a UDP socket created by the initial server process, such that each arrival of a message on the network triggers the invocation of an upcall from within soft interrupt context. Within the soft interrupt context, a message is transmitted back to the client. Although the softirq may be preempted by interrupts, there is no possibility of scheduling between the arrival of the message and the return transmission.

The experiment measures the effective bandwidth achieved when transmitting and receiving the stream as described above. All measurements are recorded at the client side, and the raw data collected represents the time including the first send and the last receipt of a message (including system call latencies). 100 trials are run and median values are reported, for each of the asynchronous I/O mechanisms stated above. The trials are repeated after starting 0 to 4 concurrent CPU-bound processes. The results are illustrated in Figure 4, in which the horizontal axis specifies the number of CPU-bound processes running in the system and the vertical axis represents the effective round-trip bandwidth achieved during the stream transmission.

A second experiment compares the round-trip latencies incurred in transmitting a 512B message between the client and server when using each of the I/O mechanisms mentioned above. For each asynchronous I/O method, and for background CPU-bound processes numbering from 0 to 4, the measurements are recorded from 1000 trials of the message round-trip transmission and mean values are calculated. The results are reported in Figure 5, in which the horizontal axis depicts the number of competing processes, and the vertical access specifies the round-trip latency measured in microseconds. Additionally, the standard deviation of each set of 1000 trials is calculated, and the resulting data is shown in Figure 6.

As shown in the results, the upcall method achieves significantly higher bandwidth for the UDP stream transmission in comparison to the SIGIO and AIO mechanisms. Moreover, the bandwidth in the case of upcall from softirq context depends much less on the number of CPU-bound

[†]Without loss of generality, we focus on asynchronous read events in this paper, but our approach is applicable to asynchronous writes as well.

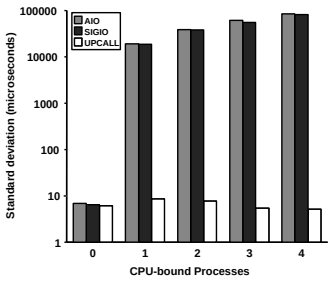

Figure 6. UDP message transmission jitter.

processes running in the system than in the case of AIO or SIGIO. However, there is a slight drop in effective bandwidth during transmission when at least one CPU-bound process is running, due to increased scheduling latency when the system scheduler is invoked as a result of timer interrupts, but the addition of more background processes has relatively little effect beyond one task. In the case of SIGIO and AIO, there is a significant decrease in effective bandwidth with the presence of each new background process, since the associated signal handlers may not be executed until the time quanta for other processes have expired. Processes that use the SIGIO or AIO techniques rely on the scheduler for invocation of I/O event handling procedures, and thus an increase in CPU contention causes an increase in event handling latency. It should be noted that whether or not the processes involved in the experiments described above are classified as real-time by the Linux scheduler has little effect on the overall results. Since the softirq upcall approach effectively bypasses the task scheduler in order to dispatch events from softirq context, even in the case where CPU-bound processes are set to high priority, the dispatch latencies for I/O events remain unchanged. Although such experimental results are not given here, prior work shows that the softirq upcall mechanism achieves lower latency than the signal-based approaches when the associated event-handling tasks are given real-time priorities under the Linux scheduler [21].

Comparison of the softirq upcall mechanism with techniques such as RTAI/RTNet [12] real-time extensions is currently underway, but the difficulty in producing such results is related to the lack of support for various network interfaces in RTNet. For this reason, the ability to use extensions such as RTAI and RTNet for real-time communication is limited to very particular hardware configurations. However, any network interface card that is supported by Linux is usable in the upcall-based asynchronous I/O mechanism, since in this case existing drivers are used to perform top-half operations. Alternative asynchronous I/O techniques include implementations of the select() and poll() interfaces [6]. However, we omit comparison with these approaches as they are known to incur relatively high latency.

5 Related Work

A number of real time operating systems have been designed with the goal of providing hard real-time guarantees to applications. Among others, VxWorks and the QNX Neutrino kernel provide a variety of features supporting the implementation of real-time tasks, such as real-time scheduling policies, synchronization mechanisms, and IPC [4, 11, 20]. For instance, the Neutrino kernel supports a sporadic task model in which a task may be assigned two priority levels. At strategic points in the execution of the application, the task may be promoted to its highest priority level, in order to obtain resources necessary to complete a portion of processing before a specified deadline. Such systems are dedicated to servicing real-time applications, but are not designed to handle a mixture of concurrent real-time and non-real-time tasks.

There are several extensions to COTS systems, such as RTLinux, RTAI, and Montavista, that attempt to enable support for real-time tasks [7, 9, 19, 8]. These systems perform well and allow for best-effort tasks to coexist with real time applications. Disadvantages to the approaches taken in the case of RTAI and RTLinux include the inability for tasks to share interrupt request vectors and poor integration with existing driver code. Although I/O events can be handled in real time using special driver modules (i.e., RT-Net [12] for processing network events), implementations of such extensions currently do not support many common devices. For example, the RTNet driver suite does not support Intel-based 1Gb Ethernet cards. Additionally, some researchers have focused on the structure of systems and the use of micro-kernels in real-time application domains, showing how the communication costs between separate address spaces is often less than the overheads associated with interrupt processing [16].

Several asynchronous I/O methods have been proposed and implemented in the research community. The select() and poll() system calls allow application programmers to efficiently multiplex network events to handlers [6]. Signals also provide a generic mechanism for event handling in most modern systems and are used as a means of implementing a variety of asynchronous I/O interfaces, such as in the GLibc AIO implementation. Additionally, the kevent()/kqueue() system calls in FreeBSD allow a user application to register a set of *kevents*, each of which includes a filter condition and an identifier, that are placed into a *kqueue* by the kernel when the condition becomes true [14]. A process may issue system calls to traverse and manipulate the queue of pending events.

The scheduling of real-time tasks concurrently with best-effort tasks is studied in several projects, including

the Rialto system and Borrowed Virtual-Time Scheduling (BVT) [15, 10, 13]. Hierarchical scheduling research also contributes to the need for predictable and efficient service for multiple co-existing execution contexts [18]. Also, systems such as the Linux Resource Kernel focus on proper accounting of resource consumption in order to accurately schedule tasks [17]. Such efforts are complimentary to the contributions of this paper, since effective scheduling policies are necessary for deciding when to dispatch event handlers. The methods we propose for asynchronous I/O provide the framework upon which well-designed real-time scheduling algorithms may be implemented and tested.

6 Conclusion

This work focuses on extending existing general purpose systems with support for real-time asynchronous I/O. We propose a method of dispatching events from within *soft interrupt* context, in order to provide reliable service for soft real-time applications. In our analysis, it is shown that events associated with real-time device I/O, specifically network communication, can be handled with a total dispatch latency in the 10's of microseconds range when using typical COTS components. We present and analyze an asynchronous I/O mechanism for handling network socket events that exhibits message transfer latencies that are independent of the number of background processes competing for system resources. Results indicate that our method significantly out-performs the implementations of the GLibc AIO and Linux signal APIs.

Insights from this work are being considered in the design of new system abstractions for safe, predictable and efficient service execution. Specifically, future work involves the study of composable real-time service modules, with the ability to define scheduling and dispatching hooks along both synchronous and asynchronous control flow paths. Hardware and software techniques will also be investigated, for use in the isolation of various application-specific and system service extensions.

References

[1] Diet Libc: http://www.fefe.de/dietlibc.

[2] GNU C library: http://www.gnu.org/software/libc.

[3] Montavista Software - Powering the embedded revolution: http://www.mvista.com.

[4] VxWorks Programmers Guide. Technical report, Wind River Systems, Alameda, CA, 1995.

[5] The Open Group Base Specifications Issue 6: http://www.opengroup.org/onlinepubs/009695399, 2004.

[6] G. Banga, J. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *USENIX Annual Technical Conference*, Monterey California, June 1999.

[7] M. Barabanov. A Linux-based Real-Time Operating System. Master's thesis, New Mexico Institute of Mining and Technology, June 1997.

[8] L. E. L. del Foyo, P. MejiaAlvarez, and D. de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, USA, April 2006.

[9] L. Dozio and P. Mantegazza. Real-Time Distributed Control Systems Using RTAI. In *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 2003.

[10] K. Duda and D. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Symposium on Operating Systems Principles*, pages 261–276, 1999.

[11] D. Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.

[12] Jan Kiszka, Bernardo Wagner, Yuchen Zhang, and Jan Broenink. RTnet - A Flexible Hard Real-Time Networking Framework. In *10th IEEE International Conference on Emerging Technologies and Factory Automation*, September 2005.

[13] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, December 1993.

[14] Jonathan Lemon. Kqueue: A Generic and Scalable Event Notification Facility. In *FREENIX Track (USENIX-01)*, pages 141–154, Berkeley, California, June 2001.

[15] M. Jones, J. Barrera, A. Forin, P. Leach, D. Rosu, and M. Rosu. An Overview of the Rialto Real-Time Architecture. In *Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[16] F. Mehnert, M. Hohmuth, and H. Hartig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, Austin, Texas, December 2002.

[17] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, June 1998.

[18] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *24th IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December 2003.

[19] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, June 1998.

[20] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, pages 73–82, 1990.

[21] R. West and G. Parmer. Application-specific service technologies for commodity operating systems in real-time environments. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Jose, California, April 2006.