REACT™ in IRIX™ 6.4 Technical Report

Silicon Graphics, Inc.

A description of the real-time capabilities of IRIX 6.4 running on Onyx2™ and Origin™ multiprocessor systems.

April 1997

1.0 Introduction

The REACT^{TM} extensions to IRIX^{TM} enable the programmer to configure a multiprocessor system to provide deterministic performance. REACT is the set of real-time features that comes standard with every IRIX installation. These features include POSIX 1003.1b interfaces, processor control capabilities, interrupt control/routing and I/O interfacing to the PCI and VME bus. With 6.4, IRIX is now fully conformant to POSIX 1003.1b. Many of the REACT extensions benefit not only traditional, real-time applications but also other applications such as multimedia and graphics. The REACT features also simplify the implementation of running real-time applications.

The approach used in IRIX with REACT to achieve determinism is to provide the user with full control over the assignment of software activity to processors. One processor (or more, if desired) is designated as the system processor, and all non-deterministic system activity takes place on that processor. For example, system activity typically includes the UNIX scheduler and general-purpose disk and network I/O. The remaining processors are designated as real-time processors, and no system activity takes place on those processors unless explicitly requested by a real-time process.

1.1 Scope Of This Document

This document provides a detailed introduction to real-time computing solutions from Silicon Graphics. It covers the REACT and REACT/pro extensions to the IRIX 6.4 operating system for Onyx2 or Origin200/2000 multiprocessor systems. This report does not describe the real-time features or capabilities of IRIX 6.4 executing on any other hardware platform. Specific hardware design details of the Onyx2 and Origin200/2000 systems are addressed when they are of particular interest to developers of real-time applications. It is assumed that readers of this document already have an understanding of the basic hardware architecture of Onyx2 and Origin systems. If not, please refer to the documents identified in section 1.3.

1.2 Configuration Assumptions

The functionality described in this document is available on any multiprocessor Onyx2, Origin 200 or Origin2000 system running IRIX 6.4. Within this document, these configurations are referred to generically as “the system”. Some real-time features described herein require the addition of REACT/pro or WindView for IRIX software to the base configuration.

1.3 Related Documentation

For more detailed or other related information about the topics discussed in this document, refer to the documents identified here. These documents are available both on the World Wide Web and on-line through SGI’s IRIS InSight^{TM}. Many of these documents plus other current real-time information can be found at: *http://www.sgi.com/real-time/*.

IRIX Admin: System Configuration and Operation (007-2859-002)

IRIX 6.4 Device Driver Programmer’s Guide (007-0911-070)

IRIX Man Pages (available on-line)

Topics in IRIX Programming (007-2478-004)

REACT Real-time Programmer's Guide

WindView for IRIX Programmer's Guide (007-2824-001)

Cellular IRIX™ 6.4 Technical Report

Onyx2™ Technical Report

Origin2000™ Technical Report

IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment 1: Real-time Extension [C Language]

2.0 System Interrupt Response

This section describes the events that occur in response to an external interrupt.

2.1 Total Interrupt Response Time

The REACT extensions included in IRIX 6.4 provide guaranteed deterministic interrupt response on a properly configured Origin or Onyx2 system. Performance is specified in terms of total interrupt response, which is defined as the interval between the occurrence of an external interrupt and the start of execution of a user process that was enabled by that interrupt (full context switch included). The worst-case total interrupt response time for a properly configured system is guaranteed not to exceed 200 µs. This guarantee covers all multiprocessor Onyx2 and Origin systems running IRIX 6.4 or any successive version of IRIX.

Total Interrupt Response time can be divided into two major component intervals (refer to Figure 1):

1. **Interrupt latency.** The time between the occurrence of a hardware interrupt and the instant when the operating system begins responding to that interrupt.
2. **Software interrupt response time.** The system time spent responding to the interrupt, ending when a user process begins executing.

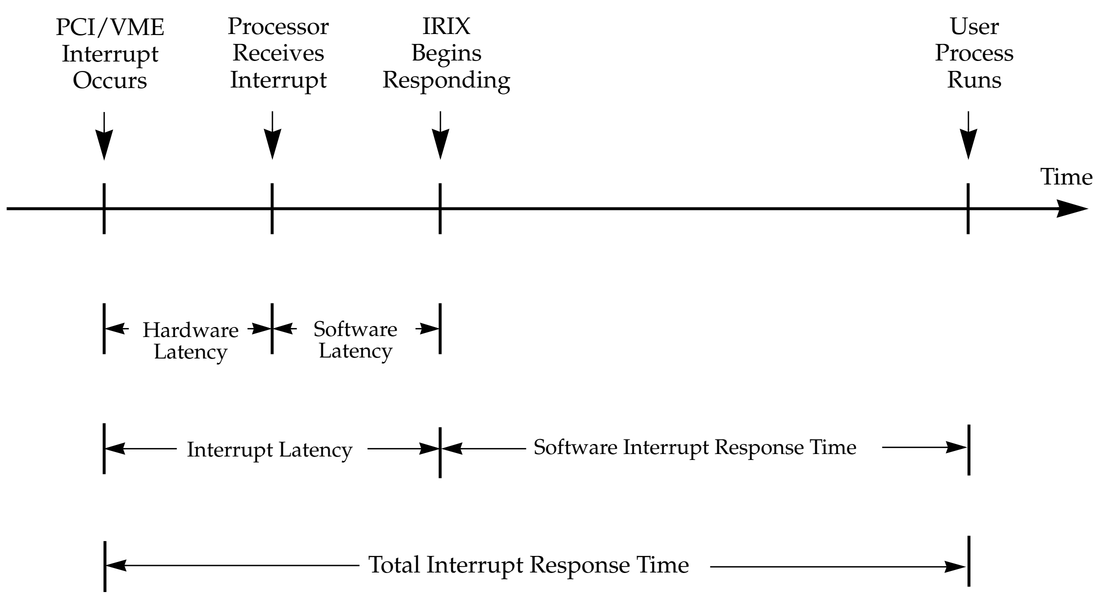

FIGURE 1. Components of Total Interrupt Response Time

Interrupt latency, as defined in the previous section, can be subdivided into two component intervals:

1. **Hardware interrupt latency.** The time required for the interrupt to propagate through the hardware from its source to the processor chip's interrupt pin.
2. **Software interrupt latency.** The interval between the instant when the processor receives the interrupt, and the instant when the operating system begins responding to the hardware interrupt (i.e., when the device specific interrupt service routine begins execution).

2.1.1 Hardware Interrupt Latency

For Origin and Onyx2 systems, the typical time required for the interrupt to propagate from the VME bus to the appropriate CPU chip is less than 2µs (interrupt latency from the PCI bus is slightly less). The theoretical worst-case propagation delay is under 5 µs. Note that the theoretical worst case requires a very large configuration with propagation over several routers. For single module configurations, 3µs is a more appropriate worst-case hardware interrupt propagation delay. All of these delays assume that no other PCI/VME master has the bus when the interrupt is generated.

Upon receiving a PCI/VME interrupt, the Origin/Onyx2 PCI/VME interface will read status registers to identify the interrupting device(s) and clear the interrupt. This information is used to select which device-specific interrupt service routine (ISR) will execute.

In order to minimize the time spent handling interrupts on the VME bus, it is preferable to connect only a single device at each VME interrupt level.

2.1.2 Software Interrupt Latency

Software interrupt latency is the interval between the time when the hardware interrupt arrives at the processor, and the time when the OS begins responding to the interrupt (i.e., the interrupt service routine for that specific interrupt executes). It is important to understand a couple of new features in IRIX 6.4 before identifying the components of software interrupt latency.

2.1.2.1 Interrupt Threads

Under traditional IRIX (prior to 6.4) each CPU in the system had an interrupt stack of one page size from which interrupts service routines were executed. When an interrupt was received, the context of the currently executing process (if there was one) was saved and the interrupt service routine executed on the CPU's interrupt stack.

Starting with IRIX 6.4, most interrupts are now serviced through interrupt threads. Interrupt threads are lightweight kernel execution entities that are scheduled at priorities shared with user processes. The main benefit of interrupt threads for real-time developers is that they permit greater control over when interrupt service routines will execute thereby allowing for more deterministic real-time response. Previously, with interrupt service routines executing on the interrupt stack, all interrupts executed at priorities higher than all user and kernel processes. This could result in non-real-time interrupts having priority over critical real-time processes, most notably on single processor systems. Conversely, interrupt threads share priority space with all other processes. Therefore, real-time developers can now execute their highest-priority real-time processes at priorities higher than all non-real-time interrupt service routines.

2.1.2.2 Fully Preemptable Kernel

With the implementation of interrupt threads, IRIX 6.4 is fully preemptable. A preemptable kernel minimizes priority inversion by allowing a high priority process (or thread) to immediately preempt a lower priority process (or thread) when it is executing kernel code. Previously, IRIX was preemptable at only a finite number of points within the kernel. With 6.4, however, IRIX is now preemptable except at the areas where a kernel spinlock is being held.

2.1.3 Software Interrupt Latency Components

Software interrupt latency under IRIX 6.4 with interrupt threads is made up of the following components: the time needed for IRIX to detect the interrupt, the time to perform a mode switch to the CPU interrupt dispatcher, the time to execute the CPU interrupt dispatcher, the time to dispatch the device-specific interrupt thread and the time it takes to perform a thread context switch to the interrupt thread (refer to Figure 2).

Anytime interrupts are not masked in software, then the time needed for IRIX to detect the interrupt is less than one instruction cycle. Under IRIX 6.4, the interrupt dispatcher executes very quickly - within a few microseconds. If the interrupt is the highest-priority executable thread, then the device-specific interrupt thread is dispatched immediately (preempting any lower priority process or interrupt thread).

Since IRIX 6.4 is fully preemptable and interrupts are not blocked in device-specific interrupt threads via raising of the Interrupt Priority Level (IPL), software interrupt latency is kept to a minimum.

2.1.4 Potential Sources Of Software Interrupt Latency

The two situations in which interrupts are potentially held off are:

1. When a higher priority interrupt handler is executing.
2. When critical regions of kernel code are executing.

The following sections examine these two situations in detail.

2.1.4.1 Higher Priority Interrupt Handlers Executing

While the CPU interrupt dispatcher is executing, it masks interrupts from being serviced for a very short period of time. All other interrupt service routines executing on processors configured for real-time will execute as interrupt threads and can therefore be interrupted by higher priority threads at any time. This enables high-priority, real-time interrupts to preempt lower priority interrupt threads and execute with very short latencies. Additionally, since interrupt threads run only on the processor to which the interrupt is directed, latency resulting from non-real-time interrupts and unrelated CPU interrupt dispatcher cycles can easily be prevented. If non-real-time interrupts are directed away from real-time processors as described in Section 3, they will never introduce latency.

2.1.4.2 Critical Regions of Kernel Code Executing

IRIX 6.4 is a fully symmetric, fully preemptable, multiprocessor operating system that allows multiple processors to execute simultaneously within the single memory-resident image of the kernel. As previously mentioned, IRIX 6.2 was preemptable at only a finite number of points within the kernel. With 6.4, however, IRIX is now preemptable except at the areas where a kernel spinlock is being held. Only during very short periods of time, where tasks such as thread scheduling and TLB management are occuring, will preemption not occur immediately in IRIX 6.4.

To maximize determinism, real-time processes should avoid making system calls during time-critical regions. Instead, activities such as forking processes, allocating memory, etc. should be done prior to real-time execution as part of an initialization routine. Typically, the only types of kernel activity that should be initiated during a time-critical region are those involved with process synchronization, communication or scheduling.

2.2 Software Interrupt Response Time

The software interrupt response time is the interval between the time when the device-specific interrupt service routine begins responding to an interrupt and the time when a user level process begins executing. Figure 2 shows the sequence of operations that are included in the software interrupt response time. These are described below:

- **Interrupt service routine (ISR).** This is the time spent processing the interrupt in a device-specific interrupt thread. ISRs always run on the processor where the hardware interrupt signal has been directed. Users must provide an ISR for any device they add to the system that generates interrupts. Typically, an ISR will unblock a process by incrementing a semaphore or sending a signal to alert a process that the interrupt has occurred. Since ISRs are application dependent, the time actually spent in the ISR can vary depending upon what activity is performed. ISRs typically run at kernel level, but with the addition of REACT/pro User Level Interrupts (ULIs), user-level code can be written to service an interrupt (refer to Section 4.5 for further discussion of ULIs).
- **Dispatch cycle.** During the dispatch cycle, the scheduler daemon determines which user process should run next. A dispatch cycle will be followed either by a context switch or a mode switch, depending on the outcome of the dispatch cycle. The scheduler either resumes execution of the current process, or initiates a context switch. If the processor was in the kernel idle loop at the time the interrupt occurred and the next process to run was the one running just before the processor entered the idle loop, a context switch will not be required.
- **Context switch.** During a context switch, the kernel saves the context of one user process and restores the context of another user process.

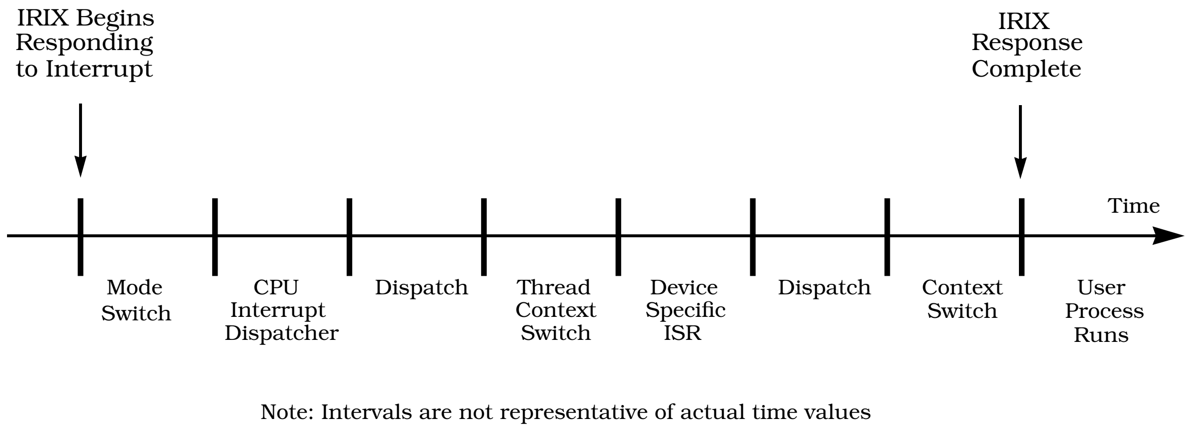

FIGURE 2. Components of Software Interrupt Response Time

3.0 Configuring For Real-Time Operation

Configuration checklist:

1. Direct interrupts not related to real-time processes away from the real-time processors. Direct real-time interrupts to the real-time processors.
2. Restrict real-time processors. This excludes the real-time processors from running any processes not explicitly assigned to them.
3. Isolate the real-time processors from virtual memory management processing (isolate is a superset of restrict - if isolate is done restrict is not necessary).
4. Assign real-time processes to real-time processors.
5. Allocate and lock all physical memory used by real-time processes.
6. Exempt real-time processors from system clock interrupts and UNIX timesharing scheduler activity.

Each of these steps is supported by the REACT extensions that are included in Irix. Once these steps are complete, the total response time to an interrupt directed to a real-time processor will be less than 200 µs, from the time the hardware interrupt occurs until a user process begins executing.

This timing assumes one of two possible configurations. The recommended configuration is that the interrupt signals start-of-frame to the REACT/pro frame scheduler (see “REACT/pro Frame Scheduler (FRS)” on page 16). The frame scheduler will the initiate execution of the highest priority process enqueued in that frame’s run queue.

An alternate scenario for achieving the guaranteed worst-case timing requires a PCI or VME device with a kernel-level device driver that handles interrupts from the device. The interrupt service routine will unblock a specific user process that is sleeping awaiting the interrupt. The guaranteed timing assumes the worst-case scenario in which a full context switch is required.

When the frame scheduler is invoked, steps 2-6 listed above are invoked automatically, and the only additional action required of the user is to redirect interrupts.

3.1 Redirecting Interrupts

Each hardware interrupt must have an associated Interrupt Service Routine (ISR), either as part of a kernel device driver or a REACT/pro User Level Interrupt. When an interrupt occurs, the ISR executes on the processor to which the interrupt has been directed at boot time.

Unless otherwise directed, at boot time IRIX will distribute the interrupts from the configured I/O devices across all available processors in order to distribute the processing load evenly. Real-time behavior can be disrupted if a non-real-time interrupt is assigned to a real-time processor.

NOTE: Interrupts must be explicitly redirected by the developer to ensure proper real-time behavior. Otherwise, the system may assign non-real-time interrupts to real-time processors without notifying the user.

The user has full control over the assignment of any XIO-based interrupts (e.g., PCI and VME interrupts) to processors. This is controlled through the DEVICE_ADMIN INTR_TARGET directive that can be specified in the irix.sm file. A device driver doesn't need to do anything special in order to have its interrupts routed to a specified target CPU - this can be done completely through the irix.sm file. For more information, consult the IRIX 6.4 Device Driver Programming Guide and the man page *system(4)*.

3.2 Allocating Processors

In a hard real-time application, the user should bind each real-time process to a processor and restrict these processors from running any other processes. This can be done using shell commands, or more typically, using system calls. Once it is done, the system processor and any other non-real-time processors can run other applications without impacting real-time operation.

REACT provides a pair of calls for processor allocation. One command removes a processor from the pool of available processors. It will then run only processes that have been assigned to it using the other command. Once assigned to a processor, the process will always execute there, including kernel services executing on behalf of the process.

Note: Allocating processes to processors does not affect where interrupt service routines execute.

3.3 Locking Processes Into Memory

In a virtual memory system, any memory reference potentially can cause a page fault. The time required to bring the data being referenced from disk into physical memory will destroy real-time determinism. IRIX with REACT enables you to lock processes into memory.

REACT supports several mechanisms for locking processes into memory, including POSIX 1003.1b. In addition to providing a portable interface, this mechanism allows a process to lock both current memory and subsequently allocated memory with a single call.

3.4 Processor Isolation

3.4.1 Overview

Processor isolation enables the user to prevent the processor from receiving an interprocessor interrupt that would otherwise suspend execution of user code on a real-time processor. The kernel generates these inter-processor interrupts in order to perform certain housekeeping functions that are not visible to user processes.

In order to maintain the integrity of its shared memory, symmetric multiprocessing programming environment, IRIX must carry on two system activities that are not visible to user processes. These are instruction cache flushes, and Translation Look-aside Buffer (TLB) flushes. Without processor restriction, at irregular intervals IRIX will generate an inter-processor interrupt to cause all processors to flush their TLB or instruction cache.

For isolated processors, IRIX will instead set a status bit to indicate that a flush is pending. When an isolated processor enters kernel mode, the status bits are tested and any pending flushes are carried out. (A processor enters kernel mode whenever the running process makes a system call, or when an interrupt occurs that is directed to that processor.) Processor isolation ensures that the user's real-time process will never be preempted by an unsolicited interrupt from the kernel.

3.4.2 Activities That Override Processor Isolation

All IRIX kernel services are available to a process running on an isolated processor, but certain system calls will introduce additional latency because they generate inter-processor interrupts that are not blocked by processor isolation.

The following system calls will override processor isolation if they are executed by a process running on an isolated processor, or by an *sproc* of such a process running on any processor. To avoid introducing non-determinism into a real-time application, these system calls should be used only during an initialization routine which executes prior to beginning real-time operation.

- *cachectl(2)* system call to mark pages cacheable or uncacheable (note that *cachectl(2)* system calls have no effect on Challenge/Onyx systems)
- *fork(2)* system call to create a new process
- *sproc*(2) system call to create a new share group process
- *munmap(2)* system call to release pages of memory
- *sbrk(2)* system call that releases memory or grows memory past a 4 MB boundary
- *mprotect(2)* system call to set protection on a portion of memory that is shared (MAP_LOCAL is immune)
- *prctl(2)* system call to acquire information on the current process

The following system calls will generate inter-processor interrupts to an isolated processor if they are called from any processor and passed the *pid* of a process running on the isolated processor.

- *prctl(2)* system call to acquire information on a process running on an isolated processor
- a write using *proc(4)* to the address space of a process running on an isolated processor

3.4.3 Minimizing Memory Management Overhead

To minimize memory management overhead, a real-time process should not free any memory pages during real-time execution. Accordingly, real-time processes and device drivers should be written so that memory resources are allocated once and reused, rather than repeatedly allocated and freed.

3.5 Controlling Process Scheduling

Real-time developers have a choice of three approaches to scheduling their application processes under IRIX.

- POSIX 1003.1b IRIX scheduler - the general-purpose scheduler included in IRIX

- REACT/pro Frame Scheduler (FRS) - sold separately as part of REACT/pro, the FRS replaces the POSIX 1003.1b IRIX scheduler on processors where it is enabled. Available only for Origin, Onyx2, Challenge, and Onyx multiprocessor systems, the FRS can run on up to n-1 processors in the system. (The FRS cannot run on the system processor.)
- User scheduling - Simple scheduling algorithms can be easily implemented by the user.

3.5.1 POSIX 1003.1b Scheduler

The scheduler in IRIX 6.4 and subsequent releases conforms to the API specifications defined by POSIX 1003.1b. IRIX 6.4 also supports the *schedctl* semantics used in previous versions of IRIX, but the POSIX semantics are recommended for portability.

POSIX 1003.1b allows users to specify one of two scheduling policies: FIFO or Round-Robin. Silicon Graphics has extended the POSIX scheduler to include a timesharing policy that is the default on all processors.

3.5.1.1 First-In-First-Out (FIFO)

The FIFO policy implements preemptive, fixed-priority based scheduling. The highest priority process that is ready-to-run will execute either until it blocks or until a higher priority process becomes ready-to-run. When choosing between two processes with equal priority, the scheduler selects the one that first became ready-to-run. (Thus the name “FIFO”.)

No timesharing is imposed under the FIFO policy. A processor running under the FIFO policy will continue to take a scheduler clock interrupt every 10 mS, but the interrupt will not trigger any scheduler activity.

3.5.1.2 Round-Robin

The round-robin policy is similar to FIFO, but with the addition of an execution time quantum. The highest priority, ready-to-run process will execute until:

- it blocks
- it is preempted by a higher priority process
- its time quantum expires and a process of equal priority is ready-to-run

The duration of a time slice is user selectable in increments of 10 mS. This relatively course resolution makes the round-robin policy inappropriate for most hard real-time applications.

3.5.1.3 Setting Process Priority

Beginning with version 6.4, IRIX has adopted a new model for process priority. The previous model divided the range of priorities into three bands, one of which was real-time. Assigning a process a priority in the real-time band affected the aging policy for that process, and in some cases other characteristics such as timer resolution.

Under IRIX 6.4 there are two priority bands: timesharing and real-time. A process specifies its priority band by selecting a scheduler policy. Selecting either FIFO or Round-

Robin scheduling will place a process in the real-time priority band. By default, a process will be assigned the timesharing policy and priority band.

Within the real-time band IRIX 6.4 offers a set of 256 priority levels in a unified priority scheme that includes kernel threads as well as real-time user processes. Most kernel activity, including most interrupt service routines, in IRIX 6.4 consists of threads executing at a fixed priority in the real-time band. Accordingly, it is possible to assign to a user process a priority higher than most kernel activity. Developers who choose to do this should proceed with caution.

The lower 128 priority levels are available for real-time user processes that do not require priorities above those of the kernel threads.

3.5.1.4 Avoiding Priority Inversion

When executing competing fixed-priority processes there is always the danger of priority inversion. Priority inversion occurs when a high priority thread is blocked waiting for a lower priority thread to release a resource. IRIX is equipped with a basic priority inheritance protocol which handles priority inversion. This protocol is best described in:

“Sha, Lui et al., Priority Inheritance Protocols: An Approach to Real-Time Synchronization, IEEE Transactions on Computers, Vol. 39, No. 9, Sept. 1990”

For example, if a real-time process is waiting for a lock that is held by a lower priority process, the priority of the process that holds the lock will be temporarily raised to that of the higher priority process. This avoids a form of priority inversion that would otherwise occur if the low priority process had to wait for a resource held by a process whose priority was above its own, but below that of the real-time process.

3.5.2 REACT/pro Frame Scheduler (FRS)

REACT/pro, an add-on software product from Silicon Graphics, includes a periodic scheduler that is tailored to the requirements of many hard real-time applications. It is a general purpose facility that can be configured to meet a wide range of scheduling requirements. The FRS offers the lowest scheduling latency possible on Origin, Onyx2, Challenge, and Onyx multiprocessor systems.

The FRS is a kernel module that controls the scheduling on each processor where it is enabled.

3.5.2.1 Interrupt Sources

The FRS is driven by an interrupt source whose period is equal to the shortest frame-time required. The user chooses an interrupt source from among the following:

- High-resolution timers
- External interrupt
- Any kernel-level device driver (e.g.VME or PCI)
- Vertical retrace interrupt (graphics systems only)
- User process (useful for debug)

The most commonly used interrupt source for the FRS is the internal high-resolution real-time clock. Driving the FRS with the internal timer allows the user to set up frames at one or more related frequencies, and schedule processes in priority order within each frame.Other interrupt sources allow the FRS to easily be synchronized with the external world, either through the external interrupt connector on the system's back panel, or via an interrupt generated by a VME or PCI board.

For video and real-time graphics applications on Onyx and Onyx2, the frame scheduler can be driven by the vertical retrace interrupt generated by the graphics subsystem's frame buffer. This arrangement generates frames at the same rate as the video frame rate, with the FRS's start-of-frame occuring at the beginning of the vertical blanking interval. Vertical retrace should be chosen as the interrupt source in applications that combine IRIS Performer and the FRS on the same multiprocessor system.

While the FRS is typically used for periodic scheduling, it can also be driven aperiodically. An interface is provided to allow a user process such as a debugger to trigger a frame, thus allowing single-step operation.

3.5.2.2 Operation

When the FRS is enabled on a processor, it removes that processor from the pool of processors available to the POSIX 1003.1b scheduler, and configures it for real-time operation. The user specifies the number of minor frames per major frame. This relationship is shown in Figure 3.

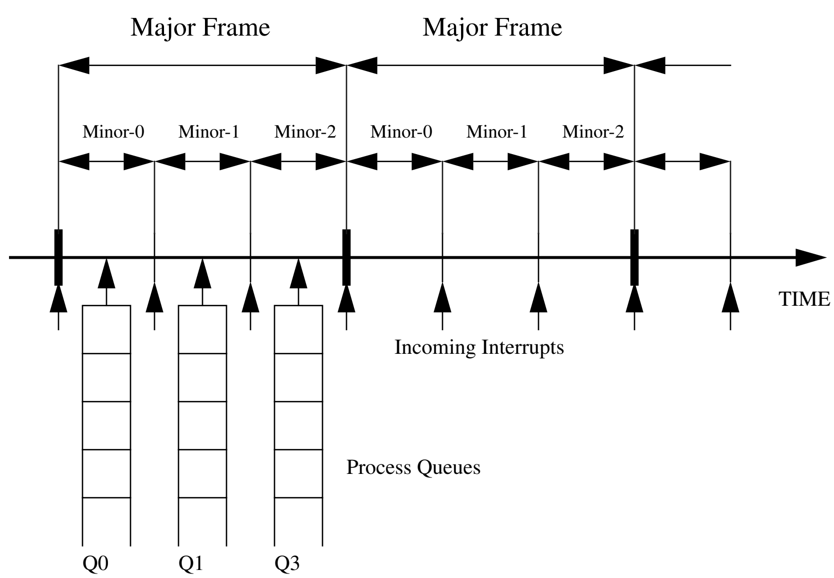

FIGURE 3. REACT/pro Frame Scheduler

Each minor frame has an associated queue of processes to be executed within that frame. The processes are executed in the order they are enqueued. Processes can be added or deleted from a queue dynamically after the FRS has been started.

The minor / major frame construct enables multiple, related frame rates to be created. The incoming interrupts determine the minor frame's period. By setting the number of minor frames per major frame, the user can create a major frame with a period equal to any number of the minor frames.

For example, a set of related frame rates can be created. Specify a timer interrupt to start minor frames at the highest desired frame rate, and specify the number of minor frames per major frame equal to the ratio of the lowest and highest desired rates. Enqueue processes to be run at the highest rate in every minor frame, and enqueue processes to be run at the lowest rate in only one minor frame per major frame. Enqueue processes to be run at the middle rate in every other minor frame, and so forth.

Each enqueued process is assigned a *discipline*, which specifies the process's preemption behavior. Nominally, a process that has not been found ready to run throughout the duration of a minor frame will generate an underrun error, and a process that has not yielded the CPU by the end of its minor frame will generate an overrun error. Other behaviors can be invoked through the assignment of disciplines to processes.

Processes controlled by the frame scheduler can block waiting for an event, such as completion of an I/O operation or availability of a system resource. The frame scheduler will start execution of the next enqueued process. As soon as the higher priority process becomes ready-to-run, the frame scheduler will resume its execution.

Frame scheduling can be enabled on all but one processor in a multiprocessor system; i.e. all but the system processor. Each frame scheduler handles only one processor, so a separate FRS must be initialized on each processor independently. One of the FRSs is designated the master, and the others are slaves. The slaves register with the master, then wait for the signal to start. In this way the FRS ensures all processors begin operation with the same frame.

3.5.3 User Scheduling

3.5.3.1 User-level Scheduling

Applications being ported to Silicon Graphics from other platforms often have an existing scheduler or executive process. While porting an existing user-level scheduler will not provide optimum performance, it is often useful.

In these cases, it is desirable to suspend most or all IRIX scheduling activity and give control of the processor to the user's executive process. The simplest and most portable method is to select the FIFO policy of the POSIX 1003.1b scheduler, and give the executive process a high process priority.

The executive process can establish fixed-rate execution by programming a POSIX timer (or *itimer*) to fire at the desired rate. Because it has the highest priority, the executive process will always run immediately after the timer fires. The executive process unblocks the processes it wants to run, then sleeps awaiting the next timer interrupt. The unblocked processes will then run in priority order.

3.5.3.2 Kernel-level Scheduling

Timers notify the requesting process of timer expiration using a signal. The time required to deliver a a signal is typically in the range of 100 to 350 microseconds. When a timer is used to wake up the executive process at start-of-frame, the time required to deliver the signal is part of the start-of-frame overhead.

The overhead of delivering a signal can be avoided by using an external interrupt source and a kernel-level device driver. In this scenario, a PCI or VME board generates the interrupt that triggers the start of a new frame. The kernel-level interrupt service routine can set semaphores to wake up the processes it wants to run. This saves the overhead of delivering a signal to wake up a sleeping process.

In this scenario, the interrupt service routine can either wake up the executive process, or it can directly wake up the process(es) to run during the frame. Having the interrupt service routine wake up an executive process maximizes flexibility and minimizes the amount of kernel-level code, but it also adds an additional context switch.

The free-running hardware timer provides the highest resolution and accuracy, and is recommended for use in real-time applications.

4.0 POSIX 1003.1b Features

This section discusses the POSIX 1003.1b features of IRIX with REACT. IRIX 6.4 is fully conformant to POSIX 1003.1b.

4.1 Timers and Clocks

Origin and Onyx systems include hardware support which enable the clocks and timers in REACT to provide high precision with minimal system overhead. Each CPU includes its own hardware clock circuit. All clock circuits are initialized simultaneously and driven by the same frequency source. This ensures complete synchronization of timers and clocks on all CPUs.

Clocks return a value for the current time. They are useful for timestamping events and for measuring the wall clock time that elapsed between execution of two points in user code.

Timers send a signal to the process that requested them after the specified period has elapsed.

IRIX with REACT supports several clock and timer programming interfaces. For maximum performance and portability, Silicon Graphics recommends use of the POSIX 1003.1b clocks and timers interfaces. Only POSIX.1b clocks and timer will be described in this document. For a description of the System V and BSD interfaces supported by IRIX, refer to the *REACT in IRIX Version 5.3 Technical Report*, or the *REACT/pro Real-Time Programmer's Guide*.

4.1.1 Interval Timers

Timers enable a user process to cause itself or another user process to receive an asynchronous interrupt after a specified time interval has elapsed. A single timer call can generate a string of periodic interrupts. When the timer expires, the process is notified using a Posix 1003.1b signal. IRIX will wake up (context-switch in) the process if it has been sleeping while waiting for that signal (and it is the highest priority). This behavior enables timers to be used to implement a simple executive that runs processes periodically.

Compared with the REACT/pro frame scheduler, using timers for scheduling incurs the additional overhead of a signal, which is typically in the range of 100 to 350 microseconds,. Also, timers handle can handle only simple timing and priority scenarios. However, scheduling based on a timer works well for many applications.

4.1.2 Clocks

When called, a clock returns the current time of day to the calling process. POSIX clocks return the time to the caller as one 64-bit word formatted as two 32-bit values. The lower 32 bits indicate nanoseconds, and the upper 32 bits indicate seconds.

Clocks are typically used to measure the elapsed time between events and to synchronize execution across systems.

By reading the time before and after an operation and taking the difference, the application can calculate elapsed time. IRIX with REACT supports four types of clock facilities: POSIX 1003.1b clocks, UNIX System V timers, BSD4.2 timers, and direct access to hardware timers from user code.

The free-running hardware timer provides the highest resolution and accuracy, and is recommended for use in real-time applications.

4.1.3 Clock Sources

When initializing a POSIX timer, the user must specify a clock source from between the two sources supported in IRIX. CLOCK_SGI_CYCLE should always be used when possible in real-time applications, because it provides the highest accuracy. However, this clock source is an SGI-specific extension to POSIX. CLOCK_REALTIME is a source available in all compliant POSIX.1b implementations, but its resolution is limited to 1 millisecond in IRIX 6.4.

As implemented in IRIX 6.4, the user chooses from between two clock sources when initializing a POSIX 1003.1b clock or timer:

CLOCK_REALTIME is available as a source for either clocks or timers. When read as a clock, it returns the system's notion of the elapsed time since January 1, 1970, expressed in seconds and nanoseconds. Select this clock to maximize portability of code.

CLOCK_SGI_CYCLE - This clock source will always be the highest resolution clock source available on any SGI platform. The resolution will vary across platforms, but the time will always be returned in the seconds/nanoseconds format specified by Posix. (Unit conversion is performed by the user-level library routine.) Use *clock_getres(2)* to determine the underlying resolution of the clock. This clock source is reset to zero each time IRIX is booted.

CLOCK_SGI_FAST - On every SGI platform, provides user access to the highest resolution clock source with interrupt capability. It is usable only for POSIX timers.

The accuracy of the crystal oscillator that drives all clock sources is 100 parts per million. This translates to a maximum timing error of 100 microseconds per second. While a timer may drift this much relative to an external time source, the skew among timers for different processors in the same system will be less than one clock tick, since all timers are clocked by the same hardware clock signal and are initialized to zero simultaneously.

Earlier versions of IRIX employed a construct called *fasthz* which imposed restrictions on the intervals that could be timed using *itimers*. Beginning with IRIX 6.2, the *fasthz* construct has been eliminated.The interface still exists, but setting *fasthz* has no effect on POSIX timers or real-time *itimers*. Instead, all timers automatically provide the full resolution of the hardware.

4.2 Shared Memory and Memory-Mapped Files

Silicon Graphic's Origin and Onyx2 systems are designed with distributed shared-memory (DSM). DSM partitions main memory among the processors but accessible to and shared by all processors. To every processor, main memory appears as a single addressable space. The effect of a single, common memory is that processes running on different CPUs can easily share pages of memory, and can update identical memory locations concurrently.

Using POSIX 1003.1b shared memory (for example), a single segment of memory can be mapped into the virtual address spaces of two or more processes. Two processes can share/transfer data at memory speeds, one putting the data into a mapped segment and the other process taking the data out. They can then coordinate their access to the data, if required, using POSIX 1003.1b semaphores. But shared memory is more than just providing common access to data, it is providing the fastest possible communication between processes.

In a program that starts multiple, lightweight processes with *sproc(2)*, all processes share the same address space and its contents. In these programs, the entire address space is shared automatically. Normally, distinct processes (created by the *fork(2)* system call) have distinct address spaces, with no writable contents in common. Shared memory facilities within an OS allow one to define a segment of memory that can be part of the address space of more than one process. The basic IRIX system operation for shared memory is the POSIX 1003.1b conformant *mmap(2)* function, with which a process makes the contents of a shared memory object (i.e., a file) part of its address space. The POSIX shared memory facility is a simple, formal interface to the use of *mmap(2)* to share segments of memory.

For backward compatibility, REACT also supports UNIX System V Release 4 (SVR4) shared memory and SGI's own arena shared memory.

4.3 Semaphores

The REACT implementation of POSIX 1003.1b semaphores provides a high-speed, user-level synchronization mechanism. POSIX defines two types of semaphores: named and unnamed. An unnamed semaphore is a semaphore object that exists in memory only. It can be identified only by its memory address, so it can be shared only by processes or threads that share that memory location (typically through the POSIX shared memory facility). Since POSIX unnamed semaphores have been implemented in user space, acquisition of an uncontested semaphore occurs immediately without any kernel overhead.

A named semaphore is named in the IRIX filesystem, so it can be opened by any process (subject to access permissions), even when the process does not share address space with the creator of the semaphore. Named semaphores are persistent, meaning that they are preserved (along with their state) until they are no longer referenced by any active process or they have been explicitly deleted from the system file.

Both POSIX named and unnamed semaphores support priority queuing such that processes blocked on semaphores are unblocked in priority order. Since unnamed semaphores are a more lieghtweight mechanism than named semaphores, they will generally out perform them.

For backward compatibility purposes, IRIX also contains an implementation of UNIX System V Release 4 (SVR4) and SGI's own IRIX arena semaphores.

4.4 Memory Locking

IRIX with REACT fully supports POSIX 1003.1b memory locking through the *mlock(3c)* and *mlockall(3C)* system calls. A single POSIX call to *mlockall(3C)* with the MCL_CURRENT parameter locks into physical memory the entire address space of the calling process. Locking a process' address space in memory not only keeps it from being paged but also helps to minimize the possibility of TLB misses during execution. Additionally, by using the MCL_FUTURE option to this same call, a process can also specify that future memory allocated be automatically locked. This includes the locking of memory that gets allocated via *malloc(3C)*, *sbrk(2)*, or through growth of the stack.

For backward compatibility, REACT also supports SGI's own memory locking call - *mpin(2)* and the SVR4 *plock(2)* call but these have no functional or performance advantages over the POSIX functions.

4.5 Asynchronous Disk I/O

Typically when a user process makes a system call to perform disk I/O, the calling process blocks until the I/O operation is complete. To meet the needs of real-time applications, IRIX with REACT supports POSIX 1003.1b asynchronous I/O. When an asynchronous I/O system call is made, the kernel initiates the I/O request on behalf of the user process and returns control to the user process. The user process can either wait for the I/O operation to complete, or it can continue executing until an I/O completion notification is sent. This sequence of events is shown in Figure 4.

.

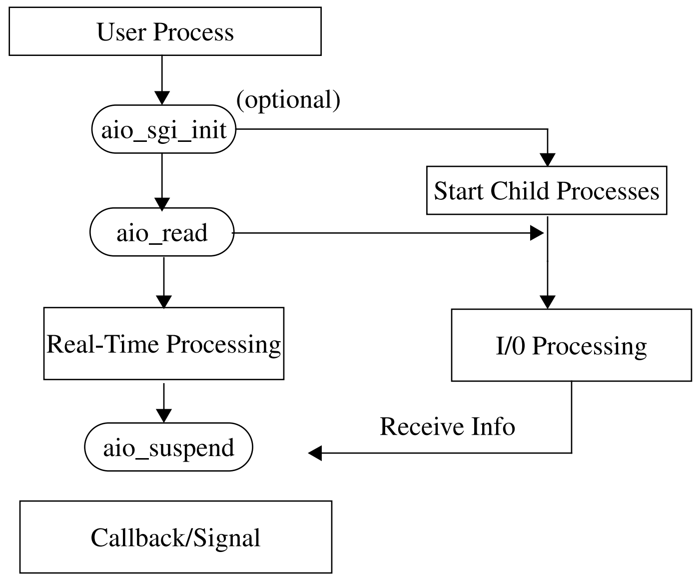

FIGURE 4. Asynchronous I/O Operation

The asynchronous I/O interface is implemented using child processes that perform the actual I/O operations, and a control block in memory (*aiocb*) containing user and system defined status and control information for the transaction, such as the file pointer and the number of bytes.

The child processes can be created by the *aio_sgi_init* system call, or by the call that initiates the first asynchronous I/O transaction. For real-time applications, it is preferable to create the child processes ahead of time, which allows the calling process to assign execution of the child processes to another processor.

The *lio_listio(3)* feature allows multiple I/O requests to be made in a single function call. The user process can simultaneously enqueue a number of *aio* requests to a device and optionally receive a queued signal or execute a callback function when the request completes.

The user process can choose to synchronously wait a specified amount of time for *aio* completion, using the *aio_suspend(3)* call. This gives a program the capability of queuing a number of *aio* requests, and then waiting until at least one of them has completed, or the program is interrupted by a signal, or the timeout specified in the call expires. When *aio_suspend* is used with *aio_error(3)* and *aio_return(3)*, the user process incurs the least amount of overhead using asynchronous I/O. Upon return from *aio_suspend*, the functions *aio_error* and *aio_return* can be applied to the individual *aiocb* for completion status.

4.6 Synchronized I/O

Synchronous I/O operations ensure that data written by a real-time process has reached the physical disk before the process resumes execution. IRIX with REACT provides the POSIX 1003.1b synchronized I/O interface for this purpose. An application specifies that synchronized I/O is to be performed on a file by simply specifying the *O_SYNC* flag on *fcntl(2)* or *open(2)* functions. When a disk file is opened specifying *O_SYNC*, each call to *write(2)* blocks until the data has been written to disk. This provides a way of ensuring that all output is complete as it is created. If *O_SYNC* access is combined with asynchronous I/O, one can let the asynchronous process suffer the delay.

IRIX with REACT also provides an *O_DIRECT* option to the *fcntl(2)* and *open(2)* functions. *O_DIRECT* provides all the advantages of *O_SYNC* but with the additional advantage of performing all reads and writes directly to or from the user program buffer (without copying to a buffer in the kernel first). As with *O_SYNC*, *O_DIRECT* can be used with asynchronous I/O.

4.7 Signals

A signal is an asynchronous notification of an event that is sent to a process when the event associated with that signal occurs. Examples of such events include hardware exceptions, timer expirations, terminal activity, as well as calls to *kill(2)*, *sigqueue(3)*, *sigsend(2)*, or *raise(3c)*. IRIX with REACT supports the signal functions in BSD4.3 and System V, as well as the POSIX P1003.1b real-time signals extension. Because only the POSIX convention provides reliable and deterministic signal notification, the remaining discussion in this section will be confined to those signals.

IRIX supports signal numbers between 0 and 64. The POSIX 1003.1b standard reserves all signals between 33 (SIGRTMIN) and 64 (SIGRTMAX) for real-time applications. The signals between 1 and 32 are of equal priority, but have a higher priority than real-time signals. The real-time signals are prioritized such that the lower the signal number, the higher the signal's priority. With the POSIX 1003.1b signal interface if multiple unblocked signals are pending, the highest priority signal will be delivered first. Conversely, a lower priority signal cannot preempt a higher priority signal handler.

4.7.1 Handling Signals

A process can request that a signal be caught and handled asynchronously, by specifying the address of a function to be called when the signal is received (*sigaction(2)*). The signal-handling function is entered asynchronously, without regard for what the process was doing at the time the signal was delivered. A process can also synchronously wait for the occurrence of a signal either with a specified timeout (*sigtimedwait(3)*) or without a timeout (*sigwait(3) or sigwaitinfo(3)*). Additionally, it can unblock a signal and then wait for that signal in a single atomic operation (*sigsuspend(3)*).

Signal latency can be long (as real-time programs measure time) and signal latency can have high variability. In real-time applications where low signal latency is critical, synchronous signal reception can be used to minimize this latency and variability. Latency is reduced by eliminating the need for an additional context switch into a signal handler.

Variability is minimized since signals are received when the process is in a known state without the uncertainties of asynchronous delivery.

4.8 Message Queues

POSIX 1003.1b message queues provide a simple interface to pass data among processes. In most cases, the recommended real-time Inter-Process Communication (IPC) mechanism on Silicon Graphics systems is shared memory because of its ease of use and lack of kernel overhead. There are cases, however, where software applications prefer to hold data locally to minimize potential data corruption - sharing data by using a well defined sender/receiver protocol. In these cases, an IPC mechanism such as POSIX message passing can be used.

As implemented in REACT, message queues support priority queuing and priority inheritance to ensure correct operation when processes with different priority are sharing a queue. Message queues have been implemented in user space using shared memory. The send and receive operations work entirely at user-level when there is no process blocking/unblocking involved. This permits message queue operations to be faster and more deterministic.

For backward compatibility purposes, IRIX also contains an implementation of message queues compatible with UNIX System V Release 4 (SVR4).

5.0 Other Real-Time Programming Features

This section describes additional features of IRIX with REACT that are useful to real-time developers.

5.1 External Interrupts

Standard on each Origin and Onyx2 system are two external interrupt ports (one input and one output), which are designed to be connected to external equipment. The interface to these lines is provided by special device files found in */dev/external_int* (see *ei(7)*). This interface allows separate machines to send and receive interrupts over a dedicated wire for inter-machine synchronization. Physically, the ports are female 3-conductor 1/8 inch audio jacks identical to those found on portable stereo headphones.

In IRIX the external interrupt device driver maintains per-process state information, allowing any number of processes to open this device and use it without interfering with each other. On Origin and Onyx2 systems, external interrupts can be routed to specific processors at kernel build time through the INTR_TARGET directive found in the irix.sm file.

A user process can enable or disable interrupts (interrupts are automatically disabled when the device is closed by the last process), assert or de-assert an interrupt, generate an outgoing interrupt pulse (one shot), or program a repetitive square wave interrupt pulse (outgoing square ware and repetitive pulse are available only on Origin and Onyx2 hardware).

Incoming interrupts can be handled in a number of ways. A process can instruct the driver to send a signal when each interrupt arrives. The interrupt queue permits the signal handler to know exactly how many interrupts have arrived, even if a signal was discarded. Or a user process may request to block in an *ioctl()* until an interrupt is received. Finally, in situations where the overhead of a system call is unacceptable (for example, when interrupts occur frequently), a process can busy wait for an interrupt to arrive, using the *eicbusywait* library function. The interrupt queue maintained for this function insures that an interrupt arriving before the library call is made will still be available to the calling process. External interrupts can also be handled by user-level interrupts, described further in Section 4.5. Finally, external interrupts can also be used as the input interrupt for the frame scheduler.

5.2 User-Level Interrupts

The user level interrupt (ULI) facility allows a hardware interrupt to be handled by a user process. The ULI facility is intended to both simplify and streamline the response to external events. On Origin and Onyx2 platforms ULIs can be written to respond to interrupts initiated either from the VMEbus, the PCIbus or from the external interrupt ports. ULIs permit users to effectively provide the Interrupt Service Routine (ISR) for an interrupt responding to events without taking a context switch. The ULI facility is both easy to use and very high performance - typical interrupt response times are under 20 µsecs.

With ULI, one sets up a handler function within a user program. The handler is called whenever the device causes an interrupt. The function is entered asynchronously from the IRIX kernel's interrupt-handling code. The kernel transfers from the kernel address space into the user-process address space, and makes the call in user (not privileged kernel) execution mode. Despite this more complicated linkage, one can think of the ULI handler as a subroutine of the kernel's interrupt handler.

Like the kernel's interrupt handler, the ULI handler can be entered at almost any time, regardless of what code is being executed by the CPU - a process of the parent program or a process of another program, executing in user space or in a system function. In fact, the ULI handler can be entered from one CPU while the parent program executes concurrently in another CPU. The parent process and the ULI function can execute in true concurrency, accessing the same global variables.

Because the ULI handler is called in a special context of the kernel's interrupt handler, it is severely restricted in the system facilities it can use. The list of features the ULI handler may not use includes the following:

- Any use of floating-point calculations. The kernel does not take time to save floating-point registers during an interrupt trap. The floating-point coprocessor is turned off and an attempt to use it in the ULI handler causes a SIGILL (illegal instruction) exception.
- Any use of IRIX system functions. Because most of the IRIX kernel runs with interrupts enabled, the ULI handler could be entered while a system function was already in progress. System functions do not support reentrant calls from the same process. In addition, many system functions can sleep, which an interrupt handler may not do.
- Any storage reference that causes a page fault. The kernel cannot suspend the ULI handler for page I/O. Reference to an unmapped page causes a SIGSEGV (memory fault) exception. This limitation can be overcome by locking down pages before entering the ULI handler (through a call to the POSIX routine *mlockall* for instance).
- Any calls to C library functions that might violate the preceding restrictions.

Not only do ULIs allow for the fastest possible response to external events, but they are also very simple to use. A program initializes ULIs by implementing the following simple steps:

1. Open the device special file for the device
2. For a VME/PCI device, map the device addresses into process memory
3. Lock the program address space in memory
4. Initialize any data structures used by the interrupt handler
5. Register the interrupt handler (single library call)

Any time after the handler has been registered, an interrupt can occur, causing entry to the ULI handler.

5.3 User-PCI Interface

The USRPCI interface provides a mechanism to access PCI bus address space from user programs. This provides a convenient mechanism for writing user level PCI device drivers if DMA capabilities on the PCI card are not required. By using the USRPCI interface to PCI devices, a user does not need to know any addressing information about the device to be interfaced - simply the slot # and type of memory to be accessed. Reads and writes can then be performed to the card through Programmed I/O (PIO) without any kernel interaction. Mapping of PCI address space to user level device drivers is supported via the *mmap(2)* system call. In this mechanism, user level drivers do an *open(2)* of the appropriate device path and invoke *mmap(2)* on the open file descriptor to map the required device space to the process address space.

One consideration to be made when accessing PCI devices is that access times will vary slightly based on the location of the PCI bus relative to the CPU where the PIO process is being executed. To minimze latency, PIO should be done from one of the CPUs that reside on the same node as the PCI bus itself.

On Origin and Onyx2 systems, PIO writes will nominally complete within 500 nanoseconds and PIO reads within 2200 nanoseconds (this assumes that there is no other conflicting traffic on the PCI bus). If access to the PCI bus includes passage through a router, one should expect an addtional latency of about 200-300 nanoseconds per router.