105 East Drive, Melbourne, Florida 32904
http://www.compro.net

Achieving Real-Time Deterministic Processing with Open Systems

White Paper

Document No: 0025W00001-00
Doc. Revision: -
Project No: 0025
Date: 2018-12-01
COMPANY CONFIDENTIAL

This page intentionally left blank.

Revision History

Revision Designator	Date	Description of Change
-	2018-12-01	Initial Release under new cover

Notice

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

Abbreviation	Definition
COTS	Commercial Off the Shelf
CPU	Central Processing Unit
DIS	Distributed Interactive Simulation
FIFO	First-In/First-Out
HIL	Hardware In the Loop
I/O	Input/Output
LAN	Local Area Network
PCI RTOM	Peripheral Component Interconnect Real-Time Option Module
POSIX	Portable Operating System Interface
RCIE	Real-Time Custom Interrupt Environment
RTE	Real-Time Environment
RTOM	Real-Time Option Module
SMP	Symmetric Multi-Processor
TCP/IP	Transmission Control Protocol / Internet Protocol
WAN	Wide Area network

1 INTRODUCTION

UNIX style (UNIX, Linux) operating system environments are robust, versatile and well suited for use in business and scientific applications. However, the *UNIX Universal Time Share Executive Process Scheduling Mechanism* is not ideally suited for applications requiring very fast, predictable process execution such as hardware-in-the-loop and human-in-the-loop simulations. A scheduler suitable for real-time applications must execute processes with a "priority-oriented preemptive mechanism" for repeatable behavior and efficiency. This is necessary for today's high-performance, real-time simulation systems.

Simulator examples include high fidelity Weapons or Flight Systems Trainers. These systems must accurately represent the real device. When a simulator provides good fidelity (i.e. the simulation and real-world are nearly indistinguishable), it "positively" trains a person resulting in specific, desired behaviors. If a simulator lacks fidelity, undesirable behavior modification may result in a "negative" training experience.

Some proprietary operating systems target *embedded* computer systems, requiring a completely different environment for simulation code development. Two examples are LynxOs and VxWorks. With Compro's value-add, UNIX style operating systems now deliver real-time determinism; real-time programmable hardware clocks, multiple external interrupts and a fast interrupt vectoring mechanism. This unified development/execution capability eliminated the need for LynxOS or VxWorks. You can now develop, test, and deploy real-time applications entirely on Open System platforms while enjoying scalability, compute power, and a world-class software suite.

Compro's value-add is the Real-Time Environment (RTE) consisting of PCI Real-Time Option Module(s) (RTOMs) with Real-Time Executive extensions. The Compro RTE and Portable Operating System Interface (POSIX)-compliant operating system combination provides a complete support package. This allows the software engineer to: 1) control when actions will occur, 2) connect actions to time-based triggers, and 3) schedule multiple tasks using a strict, priority-based First-In/First-Out (FIFO) mechanism. These ensure precise and efficient critical real-time task execution.

This white paper explores computer system behavior and how UNIX style operating systems with Compro's RTE addresses the high fidelity simulation community's requirements.

1.1 Real-Time and the Network

In Distributed Interactive Simulation (DIS) environments, real-time events and user interaction often cause message passing across an Intranet (LAN), Internet (WAN) or between processors in a Symmetric Multi-Processor (SMP) environment. These messages represent critical "data exchange" essential for an application's realistic real-world representation. In a distributed non-real-time system (such as an office environment), computer architects view keyboard input queuing, mouse operations, or network packets as more important than critical data exchange.

Non-real-time systems immediately respond to non-critical events at high priority. In this environment, network message processing with FIFO precedence is essential for "equitable responsiveness" to multiple users. This methodology assures that all users obtain reasonable response times, providing the illusion of "near real-time" performance. However, used in simulation system design this methodology is detrimental to process determinism, simulation fidelity and reliable data collection.

Message passing, through Transmission Control Protocol / Internet Protocol (TCP/IP) and similar mechanisms, creates tremendous overhead for processors and networks. Message interrupts impose significant Central Processing Unit (CPU) loads through process scheduling mechanisms, protocol handshaking and data integrity assurance (i.e., packet retransmission). Anytime an Ethernet packet arrives, it must be filtered or locally queued. In response, the process interrupts the CPU for message processing. In addition, TCP/IP delivery mechanisms generate signals (interrupts that require scheduling and servicing), further consuming precious computing cycles. Standard UNIX style operating systems are designed for processing thousands of network messages and simultaneous user inputs from keyboards and mice, with little regard for system resource utilization (such as CPU cycles) critical to simulation code execution.

Open system SMP technology supports key components that solve interrupt management problems. These operating systems use a multi-threaded kernel permitting simultaneous multiple processes execution. The kernel protects key data structures and critical code with semaphores and spin-locks, preserving their integrity. Processes executing in a multithreaded kernel can be forced to relinquish the CPU; in other words, processes can be "preempted."

In a preemptive environment, the kernel can transfer CPU control from a lower to a higher priority process. This permits a high priority process, waiting for an external event, to respond immediately when the event occurs — even if the CPU is currently in use. This is a significant benefit in real-time architecture.

2 WHAT IS REAL-TIME?

According to the POSIX 1003.1b standard, real-time in operating systems is defined as:

"The ability of the operating system to provide a required level of service in a bounded response time."

All modern Commercial Off the Shelf (COTS) computer systems have approximately one nanosecond or less CPU clock cycle times while their associated operating system software runs at approximately one millisecond cycle times. This means COTS computers have operating system software one million times slower than the raw capability of their processors (see Table 1). This does not promote real-time performance.

Table 1. Time Reference Table

Description	Part of a Second			Cross Reference
Millisecond (msec)	.001	1/1,000	10^{-3}	1/1000 of a second
Microsecond (μsec)	.000001	1/1,000,000	10^{-6}	1/1000 of a millisecond
Nanosecond (nsec)	.000000001	1/1,000,000,000	10^{-9}	1/1000 of a microsecond

Today's demanding real-time applications must capitalize upon raw processing power beyond standard operating systems. Processes that must complete in microseconds cannot be timed and controlled by software that operates in milliseconds.

2.1 Real-Time with Frame Driven and Event Driven Interrupts

The following two examples are common interrupt methods used in real-time systems.

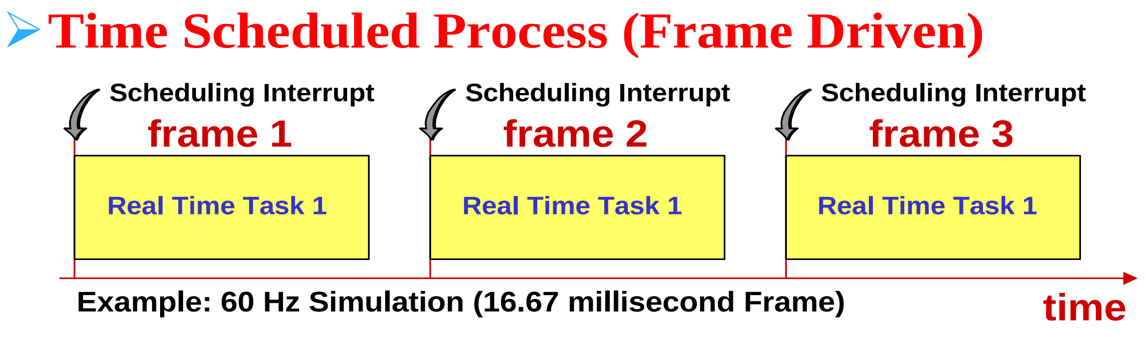

Figure 1. Frame Scheduling Interrupt

A Frame Scheduling interrupt (Figure 1) gates process execution at a predetermined rate. This cyclic interval is a “frame.” Real-world examples of “Frame Driven” scheduling interrupts with various driving frequencies are shown in Table 2.

Table 2. Example Frame Rates

Example Function	Frequency	Frame Time
Missiles, Sensors	4800 Hz	208 μsec
Digital Control Loading	1000 Hz	1,000 μsec
Flight Simulator	60 Hz	16,667 μsec

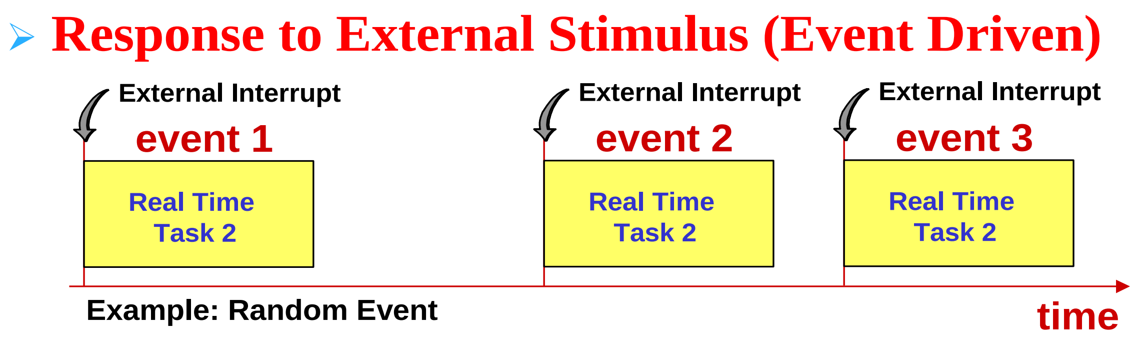

Figure 2. Event Scheduling Interrupt

An example of an “Event Driven Scheduling Interrupt” is random asynchronous missile firing, simulating a missile launch and/or missile hardware stimulation that interacts with a fixed simulation platform (a.k.a. Hardware In the Loop (HIL)). Figure 2 illustrates this example.

3 WHAT MAKES A COMPUTER A REAL-TIME COMPUTER?

In real-time computing systems, the architectural objective is reducing the number of interrupts and managing asynchronous event response in a predictable fashion. To achieve this objective, synchronous events occur at consistent intervals, and operating system housekeeping tasks (such as moving the cursor) occur using spare processing time. This permits the simulation programmer to: 1) have the computer perform the work for which it was purchased, 2) keep operating system "housekeeping" functions in good working order and 3) manage processes indirectly related to the applications. For these reasons, problems related to "determinism" and the way all software reacts or contributes to "indeterminism" must be controlled. Giving engineers control over interrupt latency, interrupt latency determinism, and machine-level code execution determinism are major factors in controlling an "indeterministic" system.

3.1 Interrupts: Friend or Fiend?

As its name implies, an interrupt stops an executing CPU process and switches execution to the interrupting process. This is called a context switch. When an interrupt occurs, the operating system must obey a set of rules to maintain functional integrity. These rules keep the operating system kernel responsive to all events within its design capacity, resulting in a "friendly" system. If the rules are bent to improve system performance or exceed system capacity (i.e. attempting real-time performance), the computer may misbehave and possibly crash with little explanation.

In a typical distributed architecture, operator inputs compete for CPU cycles. Additionally, network requests, data packet exchange, disk and graphic Input/Output (I/O), all compete for CPU cycles. Servicing these tasks has a "fiendish" impact upon real-time performance. Part of the solution for more manageable applications and better overall throughput in a real-time application is a multiple processor system. However, it takes more that multiple CPUs to achieve acceptable real-time performance. Interrupt latency, as discussed below, is another factor.

3.2 Interrupt Latency

The definition of *interrupt latency* is the time period between a received interrupt and associated user code execution. For example (in Figure 3 below), when a cyclic scheduler fires a scheduling interrupt at the beginning of each frame (t0), there is a delay before the frame code begins execution (t1). This delay is interrupt latency. The shorter the interrupt latency, the less the CPU is consumed with interrupt processing. This means more CPU cycles are available for processing useful application code before the next scheduled interrupt.

Interrupt latency is often confused with an incomplete measurement called *interrupt response*. Interrupt response is only the time required for interrupt receipt and kernel queuing (t0 + (t1-n)). *Interrupt latency* includes interrupt response time, plus queue processing time, plus time until user code execution.

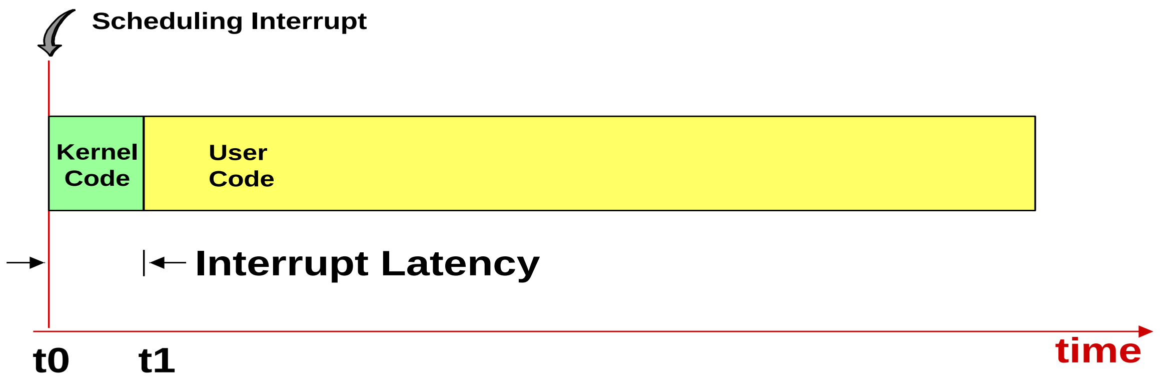

Figure 3. Interrupt Latency

3.3 Iterrupt Latency Determinism

Determinism is consistency. *Interrupt latency determinism* is the consistency of interrupt latency each time the computer responds to an interrupt. Most vendors advertise their best *achievable* interrupt response or interrupt latency time. However, this period can be as short as 10 microseconds or as long as 10 milliseconds in the same one-frame period. (Figure 4.)

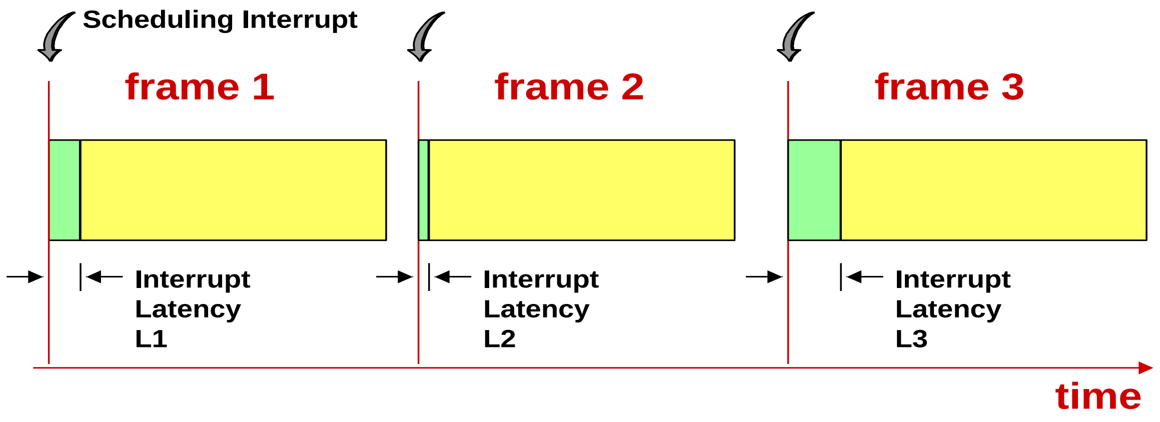

Figure 4. Interrupt Latency Determinism

As shown in Figure 5, typical operating system design exacerbates interrupt latency by consuming processor cycles with aperiodic queue management and assorted system "housekeeping" interrupts. In real-time applications these interrupts create control problems when minimum interrupt latency and solid determinism are essential.

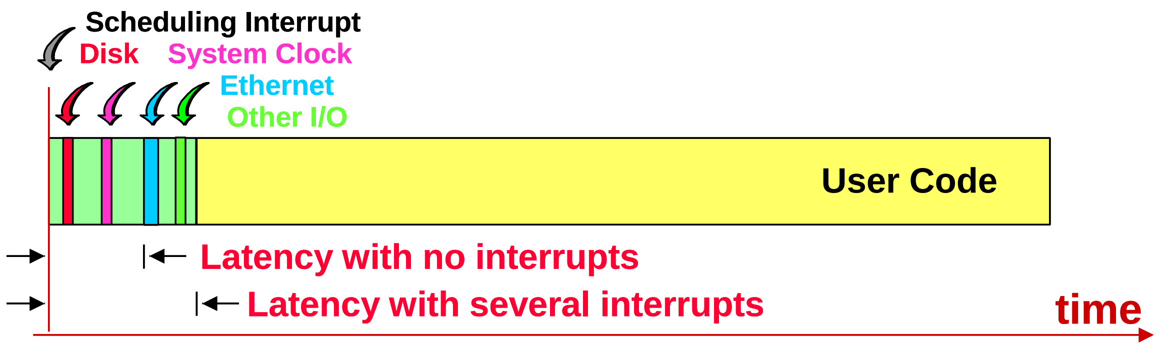

Figure 5. Interrupt Latency with/without System Interrupts

3.4 Interrupt Vector Mechanism

Part of the solution to this control problem is a "tunable pre-emptive" microkernel for quicker real-time response. The microkernel schedules tasks on separate processors using a fast real-time interrupt vectoring mechanism.

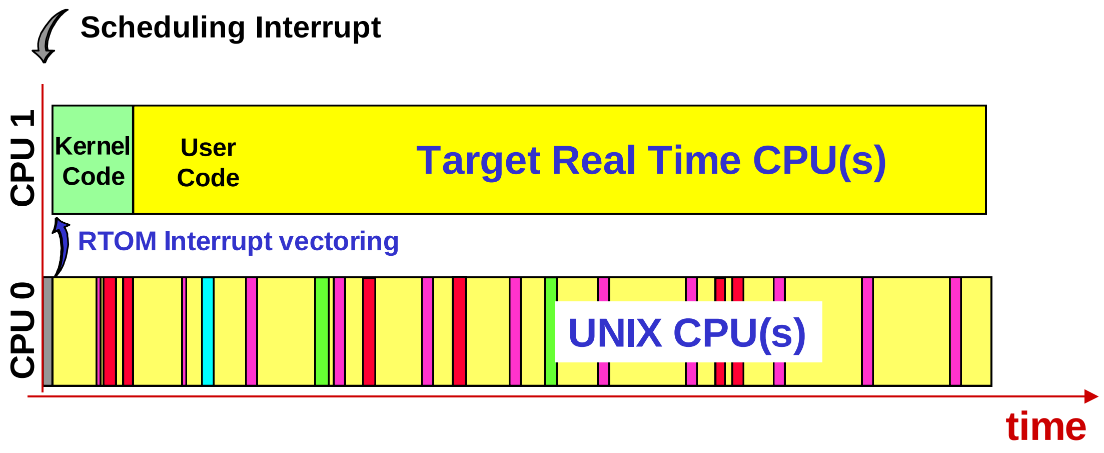

Figure 6. Target Real-Time Processor

A tunable microkernel with a fast vector mechanism gives the applications programmer *processor utilization control*. If an external real-time event needs immediate attention (such as a cyclic interrupt) the event will interrupt the designated processor and execute code per application program design. This is done without waiting for the standard UNIX style time-share scheduler to allocate a servicing time-slice.

Figure 6 illustrates two CPUs; one targeted for real-time applications (CPU 1) and the other for general system UNIX style functions (CPU 0). CPU 0 fields aperiodic interrupts and manages the Real-Time Option Module (RTOM), which triggers periodic real-time tasks in CPU 1.

3.5 Real-Time Custom Interrupt Environment (RCIE)

Also available is the RCIE (Figure 7) that can deliver extremely low latency interrupts with maximum determinism. This customization facilitates user code insertion at the kernel level (interrupt service routine) and can provide shared memory interaction with a user-level application. "Ready-to-run" examples permit rapid RCIE implementation.

When used judiciously, this powerful capability can provide significant performance enhancement. Care must be taken while RCIE code is executing, because this is a system interrupt level and nothing else happens in the system until RCIE code completes. More than a small amount of code, repeated many times, can cause excessive overhead.

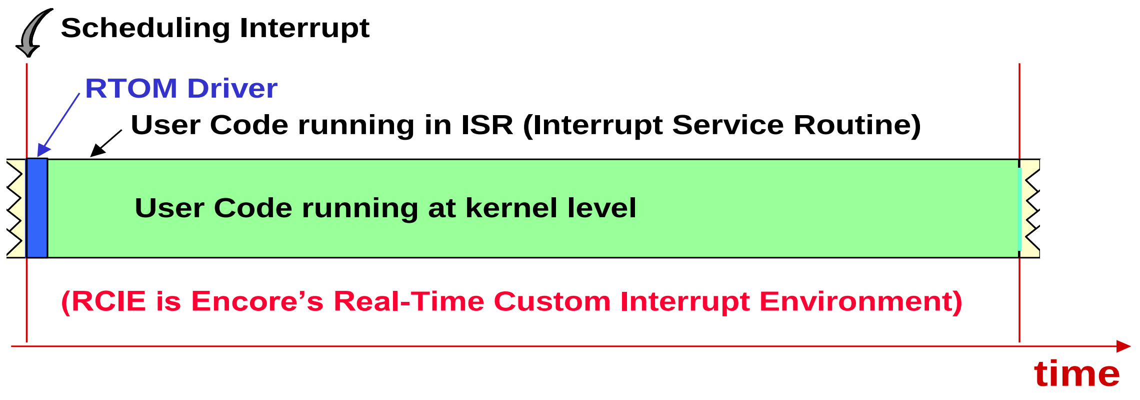
Scheduling Interrupt
RTOM Driver
User Code running in ISR (Interrupt Service Routine)
User Code running at kernel level
(RCIE is Encore's Real-Time Custom Interrupt Environment)
time

Figure 7. Real-Time Custom Interrupt Environment

3.6 Execution Determinism

Execution determinism is defined as the *variance* in user code execution time each time a specific program runs (Figure 8).

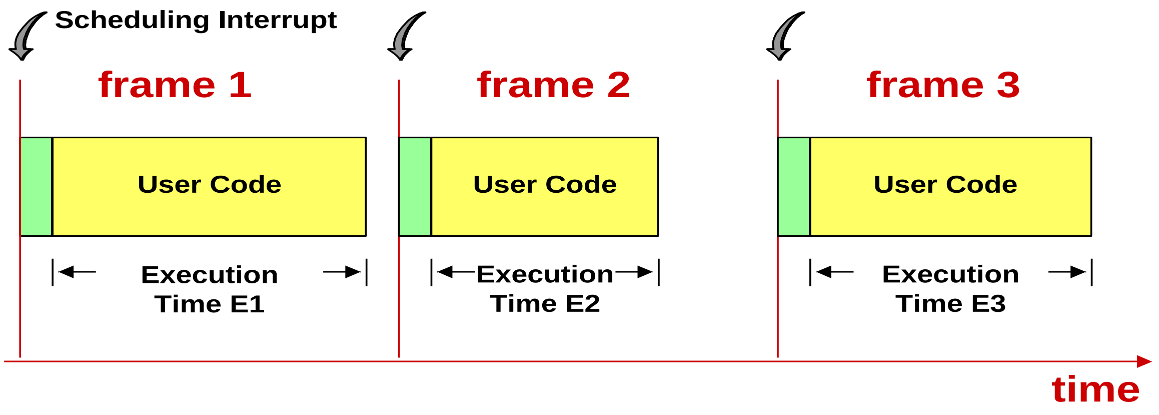

Figure 8. Code Execution Determinism

Variance occurs because the operating system performs different housekeeping tasks when diverse interrupts occur (like processing TCP/IP message traffic). With respect to application code execution, the operating system is designed for asynchronous interrupt response. Although the system clock is synchronous, peripheral I/O, network traffic and other system elements generate asynchronous interrupts resulting in code execution time variance (Figure 9).

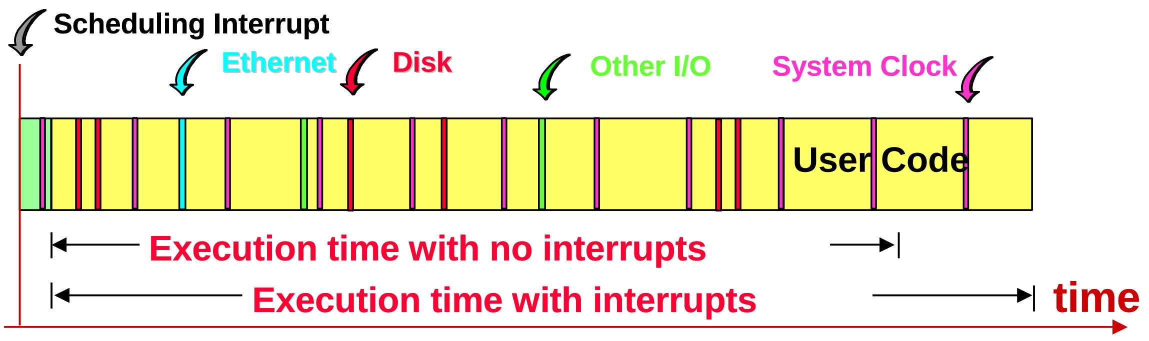

Figure 9. Interrupts During Code Execution

4 SYSTEM PERFORMANCE

4.1 Example of Non-Real-Time System

When a simulation system includes an HIL device, inadequate spare frame time can profoundly affect performance. For example, in actual Mil-Std 1553 implementations, the real-world device expects certain stimuli and response times. If the simulated system does not precisely mimic the real-world device, the simulation is ineffective. This non-deterministic behavior is sometimes called *frame overrun* or *frame jitter*. In visual simulation systems this is observed as a jerky or flashing image. Figure 10 illustrates this problem.

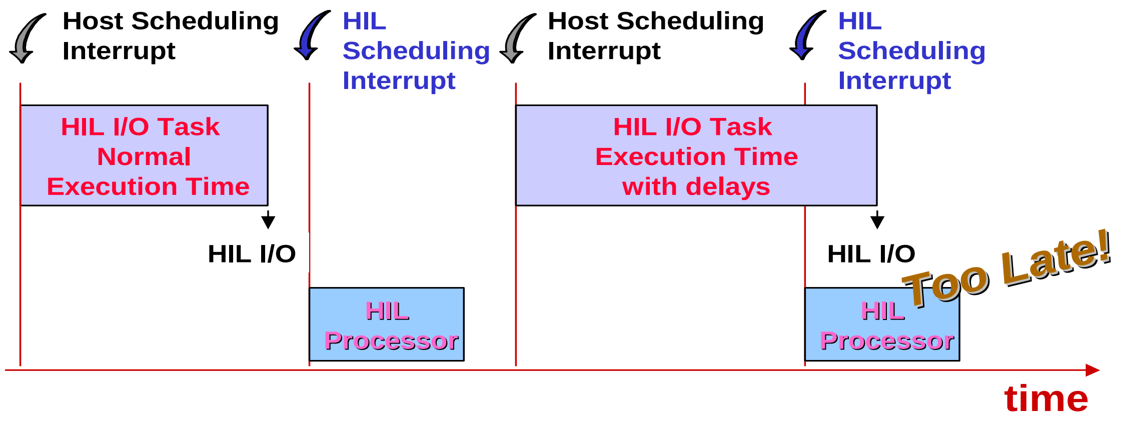
Host Scheduling Interrupt
HIL Scheduling Interrupt
Host Scheduling Interrupt
HIL Scheduling Interrupt
HIL I/O Task Normal Execution Time
HIL I/O Task Execution Time with delays
HIL I/O
HIL I/O
Too Late!
HIL Processor
HIL Processor
time

Figure 10. Frame Overrun

4.2 Improving Efficiency

Why are latency and determinism important to overall system *efficiency*? To ensure that all required work executes in each allotted frame (i.e. no "blown" frames) the system must be "sized" to accommodate a worst-case task execution time. When execution determinism is poor, (i.e., user code executes with excessive time variance) accommodating system designs result in wasted CPU cycles (Figure 11). Good execution determinism minimizes wasted CPU cycles by accurately matching system capability with predictable performance requirements, thereby improving efficiency.

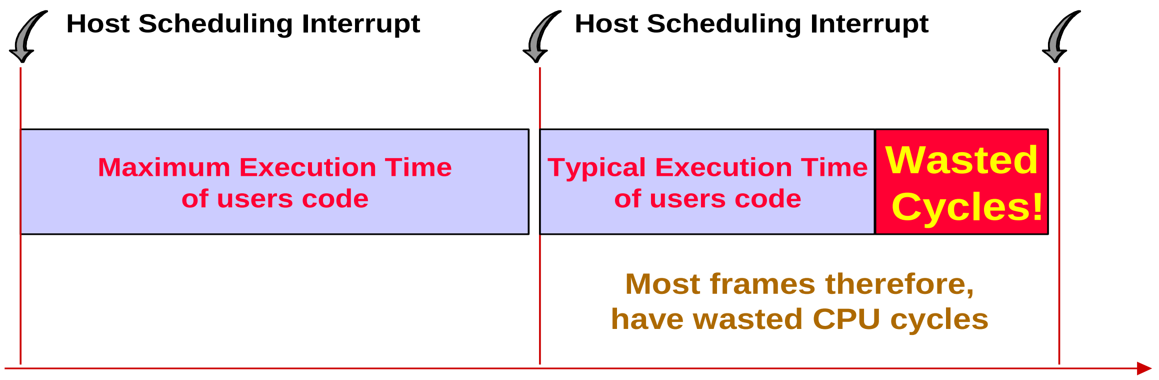
Host Scheduling Interrupt
Host Scheduling Interrupt
Maximum Execution Time of users code
Typical Execution Time of users code
Wasted Cycles!
Most frames therefore, have wasted CPU cycles

Figure 11. System Efficiency Improvement

5 REAL-TIME ENVIRONMENT (RTE) SOLUTION

5.1 High Performance Interrupts

Using the RTOM and associated Real-Time Extensions (comprising the RTE) results in extraordinary interrupt latency and determinism performance. Without RTE, average native operating system interrupt latency is approximately 60~100µsec. With RTE, latency is reduced to 10µsec or less – a factor of 10x improvement. Without RTE, average native operating system determinism is ranges wildly from 200~1000µsec. With RTE, determinism is dramatically improved to less than 8µsec – a factor of 250x improvement.

above conservatively illustrates this performance. Note that Standard UNIX style figures indicate average and typical times, with worst-case up to two hundred times slower. In contrast, Compro's RTE performance figures are tightly grouped around the indicated values.

5.2 Real-Time Configuration

This section integrates previous discussions concerning latency and determinism with the following configuration considerations:

- Objective
  - Rmove all unnecessary interrupts from the real-time execution path
- Method
  - Use a separate processor to run standard UNIX, performing system and application I/O
  - Dedicate other processors to real-time tasks by locking the tasks to specific processors and associated memory
  - Turn off all interrupts to the targeted real-time processor set
  - Turn on RTOM interrupts only for necessary real-time task communication to the target processor
  - Use non-interrupting real-time I/O co-processor to pass data to the real-time target processor. This may be used for internal transfers to shared memory regions, and Reflective Memory™ transfers from other computers.
  - Vector only task-specific real-time interrupts to the target processor set

The result is a Real-Time open system running a COTS UNIX style operating system with Compro's Real-Time Executive and Extensions. Figure 13 functionally illustrates this RTE.

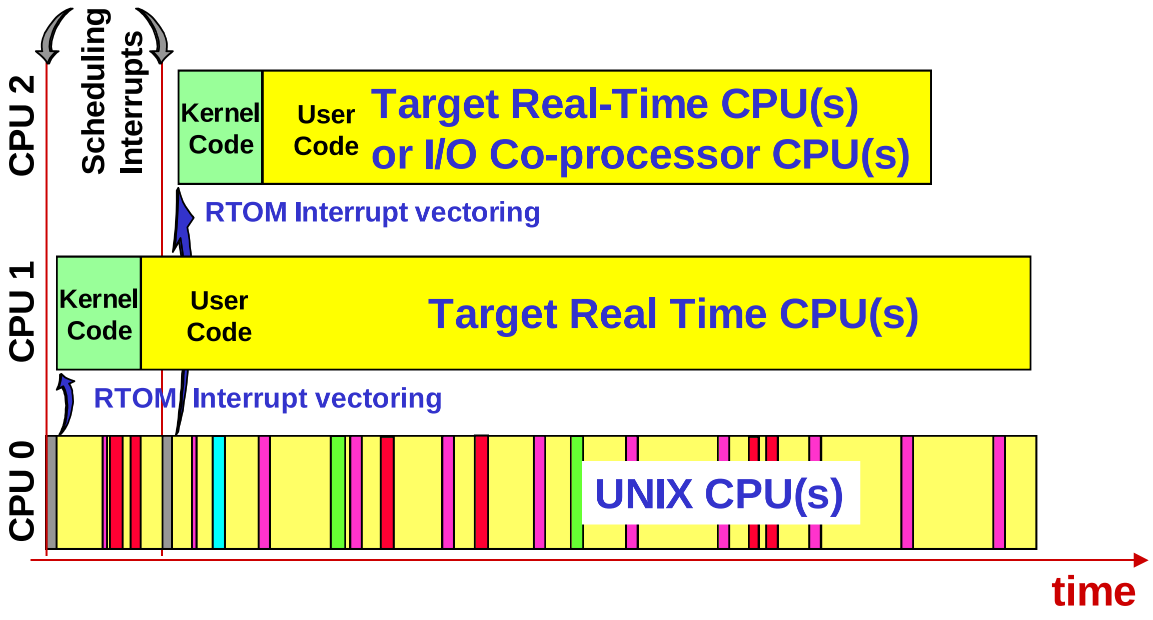

Figure 12. Real-Time Environment (RTE)

Figure 13 graphically demonstrates how a cyclic based application should operate. At t0, the scheduling interrupt occurs at a programmed, consistent interval. The PCI Real-Time Option Module (PCI RTOM) provides this scheduling interrupt, with programmable frequencies from 1Hz to 10,000 Hz. At t1, the executive scheduler completes the context switch and the targeted task begins executing. The time between t0 and t1 is the Interrupt Latency. The time between t1 and t2 is when the deterministic real-time process on the CPU occurs. When the work completes at t2, the scheduling mechanism enters a wait state, awaiting the next frame scheduling interrupt to occur at t0. The time between t2 and the next t0 is spare frame time.

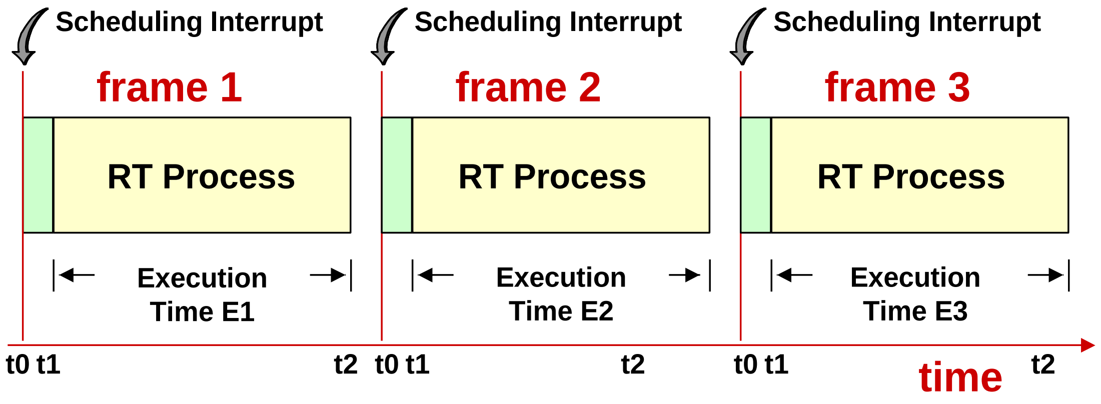

Figure 13. Execution Determinism

Controlling interrupt latency determinism and real-time process execution determinism delivers consistent real-time performance each frame.

6 SUMMARY

Open system UNIX style operating systems can provide a truly deterministic real-time environment suitable for the most demanding simulation applications with the addition of the PCI-RTOM and Real-Time Executive extensions. With this enhancement, measured interrupt latency is 10 microseconds with four microseconds of determinism, resulting in an ideal platform for hosting high fidelity applications. The PCI-RTOM, Real-Time Executive Scheduler, and operating system POSIX compliance enables features providing real-time support. These allow the programmer to: 1) precisely control when actions will occur, 2) connect actions to time-based triggers, and 3) schedule multiple tasks using a strict, priority-based FIFO mechanism. These ensure critical real-time tasks that are executed precisely and efficiently.