RTAI: Real-Time Application Interface

An introduction to RTAI for deterministic and preemptive real-time behavior for Linux.

by P. Mantegazza, E. Bianchi, L. Dozio, S. Papacharalambous, S. Hughes and D. Beal

Before jumping headlong into our discussion of RTAI, a quick review of what real time means and how standard Linux relates to real time is appropriate. The term ``real time'' can have significantly different meanings, depending on the audience and the nature of the application in question. However, computer science literature generally defines only two types: soft and hard real-time systems.

A ``soft real-time system'' is characterized by the ability to perform a task, which on average, is executed according to the desired schedule. A video display is a good example, where the loss of an occasional frame will not cause any perceived system degradation, providing the average case performance remains acceptable. Although techniques such as interpolation can be used to compensate for missing frames, the system remains a soft real-time system, because the actual data was missed, and the interpolated frame represents derived rather than actual data.

``Hard real-time systems'' embody guaranteed timing, cannot miss timing deadlines and must have bounded latencies. Since deadlines can never be missed, a hard real-time system cannot use average case performance to compensate for worst-case performance. One example of a hard real-time task would be monitoring transducers in a nuclear reactor, which must use a complex digital control system in order to avoid disaster. RTAI provides these necessary hard real-time extensions to Linux, thus enabling it to be targeted at applications where the timing deadlines cannot be missed.

Issues with Real-time Support in the Linux Kernel

RTAI provides hard real-time extensions to Linux, yet standard Linux has support for the POSIX 1003.13 real-time extensions, so how does the ``real-time'' capability of standard Linux fall short?

The POSIX 1003.13 standard defines a ``Multi-User Real-Time System Profile'' which allows for ``real-time'' processes to be locked into memory to prevent the process from being paged to hard disk, and a special scheduler which ensures these processes are always executed in a predictable order.

Linux meets this standard by providing POSIX-compliant memory lock (mlock), special schedule (sched_setsched) system calls and POSIX RT signals. While those features do provide improved deterministic performance, the resultant tasks cannot be defined as hard real-time tasks, since the soft real-time process can be blocked by kernel activity.

The standard Linux real-time POSIX API and the real-time kernel offer dramatically distinct services, especially within a multi-user and multi-tasking application. A simple program in pseudo-code, shown in Listing 1, can be used to demonstrate the system performance and issues surrounding the POSIX approach.

Listing 1

This program reads the absolute time before entering a system call which is used to suspend the process for a predetermined interval. Next, the program reads the absolute time after the system call returns. If the system is heavily loaded with a high level of kernel activity, the return from the system call will be delayed by that activity, thus the magnitude of this delay is defined by the difference between the expected sleep time (i.e., 100 milliseconds) and the actual sleep time.

For a standard Linux system that is idle, the above task performance is extremely stable and the timing deviation is very low, yielding a worst-case deviation from an average of approximately 100 microseconds on a Pentium II 300MHz system.

However, if the program is run on a standard Linux system, using the same POSIX approach but with simultaneous I/O activity, the performance dramatically deteriorates, since the facilities of standard Linux do not allow any way for the test program to preempt the I/O-intensive application. The average execution quickly becomes unacceptable when, for example, one edits a very large file and causes hard disk activity at the same time the test program is running. This inability to preempt causes the test program to suffer typical deviations of 30 milliseconds, and for the worst case, in excess of hundreds of milliseconds. Consequently, standard Linux processes using the POSIX approach cannot be used for reliable hard real-time performance in multi-tasking environments.

Although members of the Linux community have discussed introducing a low-latency scheduler in future kernels to enhance soft real-time performance, standard Linux will still fall well short of the guaranteed response time required by a hard real-time task.

The performance advantage of real-time Linux is easily seen by running this same test program as a real-time task, yielding worst case deviations of less than 12.5 microseconds regardless of I/O activity.

Along with the scheduling benefits of the real-time kernel, the efficiency of the real-time architecture allows Linux to run under aggressive task iteration rates. For example, when running this program at 500-microsecond intervals as a POSIX real-time task, Linux stops completely. Running this program as a real-time task leaves enough resources for Linux to continue to run.

The operating requirements of a general-purpose operating system such as Linux are different from those of a hard real-time system. This difference leads to other issues, summarized below, which limit the potential for standard Linux to act as a real-time operating system.

- The Linux kernel uses coarse-grained synchronization, which allows a kernel task exclusive access to some data for long periods. This delays the execution of any POSIX real-time task that needs access to that same data.
- Linux does not preempt the execution of any task during system calls. If a low-priority process is in the middle of a **fork** system call and a message is received for the video display process, then the message will unfortunately be held in the queue until the call completes, despite its low priority. The solution is to add preemption points in the kernel, having the deleterious effect of slowing down all system calls.
- Linux makes high-priority tasks wait for low-priority tasks to release resources. For example, if any process allocates the last network buffer and a higher-priority process needs a network buffer to send a message, the higher-priority process must wait until some other process releases a network buffer before it can send its message.
- The Linux scheduling algorithm will sometimes give the most unimportant and nicest process a time slice, even in circumstances where a higher-priority process is executable. This is an artifact of a general-purpose operating system that ensures a background maintenance process; e.g., one that cleans up log files and runs even if a higher-priority process were able to use all the available processor time.

- Linux reorders requests from multiple processes to use the hardware more efficiently. For example, hard-disk block reads from a lower-priority process may be given precedence over read requests from a higher-priority process in order to minimize disk-head movement or improve the chances of error recovery.
- Linux will batch operations to use the hardware more efficiently. For example, instead of freeing one page at a time when memory gets tight, Linux will run through the list of pages, clearing out as many as possible, which will delay the execution of all processes. This is clearly desirable for a general-purpose operating system, but is undesirable for real-time processes.

Real-time and general-purpose operating systems have contradictory design requirements. A desirable effect in one system is often detrimental in the other.

Unfortunately, any attempt to satisfy both requirements in the same kernel often results in a system that does neither very well. That is not to say general-purpose functionality and real-time determinism cannot be achieved simultaneously. In fact, operating systems that combine these requirements exist now, and they are indeed deterministic, preemptive and contain bounded latencies (thus meeting the requirement for a hard real-time system). However, the worst-case behavior for those latencies can be unacceptably slow.

An Introduction to RTAI

In order to make Linux usable for hard real-time applications, members of the Department of Aerospace Engineering, Politecnico di Milano (DIAPM) envisioned a real-time hardware abstraction layer (RTHAL) onto which a real-time applications interface (RTAI) could be mounted. Unfortunately, further investigation revealed that the Linux kernel available in late 1996, 2.0.25, was not yet mature enough for the concept.

Around the same time, a group headed by Victor Yodaiken at the New Mexico Institute of Mining and Technology (NMT) in Soccorro, NM introduced its real-time Linux (RTLinux), which provided the DIAPM team with the opportunity to learn further about the Linux kernel, the hardware and the modifications necessary to provide preemptive and deterministic scheduling. The turning point came in 1998 when the Linux 2.2.*x* kernel featured key improvements, including much-needed architectural changes to the Linux/hardware interface. These changes, combined with the experience gained by the DIAPM team while working with their own evolution of the NMT-RTLinux kernel, and the concepts of 1996, resulted in RTAI.

RTAI provides guaranteed, hard real-time scheduling, yet retains all of the features and services of standard Linux. Additionally, RTAI provides support for UP and SMP--with the ability to assign both tasks and IRQs to specific CPUs, x486 and Pentiums, simultaneous one-shot and periodic schedulers, both inter-Linux and intra-Linux shared memory, POSIX compatibility, FPU support, inter-task synchronization, semaphores, mutexes, message queues, RPCs, mailboxes, the ability to use RTAI system calls from within standard user space and more.

RTAI Architecture

The underlying architecture for RTLinux and RTAI is quite similar. For each implementation, the Linux operating system is run as the lowest priority task of a small real-time operating system. Thus, Linux undergoes no changes to its operation from the standpoint of the user or the Linux kernel, except that it is permitted to execute only when there are no real-time tasks executing. Functionally, both architectures provide the capability of running special real-time tasks and interrupt handlers that execute whenever needed, regardless of what other tasks Linux may be performing. Both implementations extend the standard Linux programming environment to real-time problems by allowing real-time tasks and interrupt handlers to communicate with ordinary Linux processes through a device interface or shared memory.

The primary architectural difference between the two implementations is in how these real-time features are added to Linux. Both RTLinux and RTAI take advantage of Linux's loadable kernel modules when implementing real-time services. However, one of the key differences between the two is how these changes, to add the real-time extensions, are applied to the standard Linux kernel.

RTLinux applies most changes directly to the kernel source files, resulting in modifications and additions to numerous Linux kernel source files. Hence, it increases the intrusion on the Linux kernel source files, which can then result in increased code maintenance. It also makes tracking kernel updates/changes and finding bugs far more difficult.

RTAI limits the changes to the standard Linux kernel by adding a hardware abstraction layer (HAL) comprised of a structure of pointers to the interrupt vectors, and the interrupt enable/disable functions. The HAL is implemented by modifying fewer than 20 lines of existing code, and by adding about 50 lines of new code. This approach minimizes the intrusion on the standard Linux kernel and localizes the interrupt handling and emulation code, which is a far more elegant approach. Another advantage of the HAL technique is that it is possible to revert Linux to standard operation by changing the pointers in the RTHAL structure back to the original ones. This has proven quite useful when real-time operation is inactive or when trying to isolate obscure bugs.

Many have surmised that the HAL could cause unacceptable delays and latencies through the real-time tasking path. In fact, the HAL's impact on the kernel's performance is negligible, reflecting highly on the maturity and design of the Linux kernel and on those who contributed to its development.

Real-Time Application Interface

The HAL supports five core loadable modules which provide the desired on-demand, real-time capability. These modules include **rtai,** which provides the basic rtai framework; **rtai_sched,** which provides periodic or one-shot scheduling; **rtai_mups,** which provides simultaneous one-shot and periodic schedulers or two periodic schedulers, each having different base clocks; **rtai_shm,** which allows memory sharing inter-Linux, between real-time tasks and Linux processes, and also intra-Linux as a replacement for the UNIX V IPC; and **rtai_fifos,** which is an adaptation of the NMT RTLinux FIFOs (file in, file out).

Like all kernel modules, these can be loaded and unloaded (using the standard Linux **insmod** and **rmmod** commands) as their respective capabilities are required or released. For example, if you want to install only interrupt handlers, you have to load only the rtai module. If you also want to communicate with standard Linux processes using FIFOs, then you would load the rtai and rtai_fifos modules. This modular and non-intrusive architecture allows FIFOs to run on any queue and use immediate wake-ups as necessary.

The Real-Time Task

The real-time task is implemented similarly to RTAI; i.e., it is written and compiled as a kernel module which is loaded into the kernel after the required RTAI core module(s) has been loaded. This architecture yields a simple and easily maintained system that allows dynamic insertion of the desired real-time capabilities and tasks. The example below shows all that is required for a task to be scheduled in real time:

```
insmod /home/rtai/rtai insmod /home/rtai/modules/rtai_fifo insmod /home/rtai/modules/rtai_sched insmod /path/rt_process
```

To stop your application and remove the RTAI:

```
rmmod rt_process rmmod rtai_sched rmmod rtai_fifo rmmod rtai
```

The facilities **ldmod** and **remod** may be used to load and unload the core modules.

Task Scheduler

RTAI's task scheduler allows hard real-time, fully preemptive scheduling based on a fixed-priority scheme. All schedules can be managed by timing functions and real-time events such as semaphore acquisition, clocks and timing functions, asynchronous event handlers, and include inter-task synchronization.

One-Shot and Periodic Scheduling

RTAI supports both one-shot and periodic timers on Pentium and 486-class CPUs. Although both periodic and one-shot timers are supported, they may not be instantiated simultaneously; i.e., one-shot and periodic tasks may not be loaded into the kernel as modules at the same time.

Using these timers (instantiated by rtai_sched), periodic rates in excess of 90KHz are supported, depending on the CPU, bus speed and chip set performance. On Pentium processors, one-shot task rates in excess of 30KHz are supported (Pentium II, 233 MHz), and on 486 machines, the one-shot implementation provides rates of about 10KHz, all while retaining enough spare CPU time to service the standard Linux kernel.

The limitation of rtai_sched to support simultaneously one-shot and periodic timers is mitigated by the MultiUniProcessor (MUP) real-time scheduler (rtai_mups), which provides the capability to use, simultaneously, both a periodic and a one-shot timer or two periodic timers with different periods, at a performance equivalent to that noted above under rtai_sched. Note that, since the MUP uses the APIC (advanced programmable interrupt controller) timer, it cannot operate under SMP and requires each MUP-scheduled task to be locked to a specific CPU (thus the MultiUniProcessor designation); however, the MUP retains all coordination and IPC services, so that no other capabilities are lost.

Floating-Point Operations

Floating-point operations within real-time tasks/ISRs (interrupt service routines) are possible, provided these tasks are marked, upon loading, as tasks which require the FPU. This method provides real-time task access to the FPU while still allowing FPU access to standard Linux tasks.

Interrupt Handling

RTAI provides efficient and immediate access to the hardware by allowing, if one chooses, interaction directly with the low-level PC hardware, without first passing through the interrupt management layers of the standard Linux kernel.

The ability to individually assign specific IRQs to specific CPUs, as described in further detail below, allows immediate, responsive and guaranteed interface times to the hardware.

Inter-Process Communication

The term inter-process communication (IPC) describes different ways of message passing between active processes or tasks, and also describes numerous forms of synchronization for the data transfer.

Linux provides standard system V IPC in the form of shared memory, FIFOs, semaphores, mutexes, conditional variables and pipes which can be used by standard user processes to transfer and share data. Although these Linux IPC mechanisms are not available to do real-time tasks, RTAI provides an additional set of IPC mechanisms which include shared memory, message queues, real-time FIFOs, mutexes, semaphores and conditional variables. These are used to transfer and share data between tasks and processes in both the real-time and Linux user-space domains.

RTAI's remote procedure call (RPC) mechanism is similar in operation to QNX messages available to real-time tasks, and transfers either an unsigned integer or a pointer to the destination task(s).

The RTAI mailbox implementation provides the ability to send any message from user space to real-time tasks, real-time tasks to real-time tasks, and user tasks to user tasks (using LXRT) via any definable mailbox buffer size. Multiple senders and receivers are allowed, where each is serviced according to its priority.

proc Interface

From version 0.9, RTAI includes a proc interface which gives useful information on the current status of RTAI, including schedulers loaded; real-time tasks activity, priority and period; and FIFOs in use and their associated buffer sizes. The development of even more features is currently underway.

SMP Support

RTAI provides true support for symmetric multi-processing (SMP) architectures through its task and IRQ management.

By default, all tasks are assigned to run on any CPU (of a SMP platform). Each task, however, can be individually assigned to any CPU subset, or even to a single CPU. Additionally, it is possible to assign any real-time interrupt service to any specific CPU. Because the ability to force an interrupt to a specific CPU is not related to the SMP scheduler, RTAI retains the flexibility to perform these two operations independently.

These capabilities provide a method of statically optimizing the real-time application, if manual task distribution handles the task more efficiently than the automatic SMP load-distribution services of Linux.

Linux-RT (LXRT)

Since real-time Linux tasks are implemented as loadable modules, they are, for all practical purposes, an integral part of the kernel. As such, these

tasks are not bounded by the memory protection services of Linux, and they have the ability to overwrite system-critical areas of memory, bringing the system to an untimely halt. This limitation has been a large frustration to those of us who have erred during real-time task development.

RTAI's LXRT solves this problem by allowing the development of real-time tasks, using all of RTAI's hard real-time system calls from within the memory-protected space of Linux and under a ``firm'' real-time service. When the developer is satisfied with a task's performance within LXRT, the task is simply recompiled as a module and inserted into the kernel (along with the associated modules which provide RTAI's real-time features) to transition from firm to hard real-time.

LXRT's firm real-time service, similar to that offered by the Kansas University Real-Time (KURT) patch, provides soft real-time combined with fine-grained task scheduling. Performance under LXRT is quite good, yielding latencies not much greater than for a standard Linux system call leading to a task switch. Although this is very valuable as a development tool, we should not lose sight of the fact that RTAI's firm real-time implementation can prove especially useful for those tasks which don't require hard real-time, but yet are not quite satisfied with the scheduling performance of standard Linux.

POSIX Compatibility API

RTAI implements a compliant subset of POSIX 1003.1.c through the use of a single loadable module. These calls support creation, deletion, attribute control and environment control for threads, mutexes and condition variables. The resultant POSIX support is similar to standard Linux threads, except that parent-child functions (which are not appropriate for real-time tasks, since all threads are considered to be part of a single process) and signal handling (which is currently in development) are not supported.

Typical Performance

RTAI is now competitive from both a cost and performance perspective with the commercial RTOS currently available.

Since the performance of any RTOS system is determined by the performance of the RTOS itself, the performance of the hardware on which it is running, and the test procedure used to acquire the data, absolute performance figures are very difficult to quantify, often making comparisons difficult between fundamentally similar RTOSes. However, the data below was measured on, and is representative of, typical Pentium II 233MHz and 486 platforms.

For these performance characterizations, an early version of the RTHAL module was demonstrated running a timer at 125KHz (Pentium II, 233MHz) while simultaneously servicing Linux, which was working under a heavy load. During this demonstration, the average and maximum jitters about the periodic timer were 0<\#181>s and 13<\#181>s, respectively. This performance, combined with additional tests, can be summarized in this way:

- Maximum periodic task iteration rate: 125KHz
- Typical sampling task rate: 10KHz (Pentium 100)
- Jitter at maximum task iteration rate: 0-13<\#181>s UP, 0-30<\#181>s SMP
- One-shot interrupt integration rate: 30KHz (Pentium-class CPU), 10KHz (486-class CPU)
- Context switching time: approximately 4<\#181>s

Future Developments

RTAI continues to grow and mature through the combined contributions of the DIAPM development team and those across the Internet who are encouraged by RTAI's flexible architecture, high performance and feature set. While this real-time Linux extension is very capable, stable and mature today, work is far from finished. The future holds many things, including:

- VxWorks compatibility libraries
- pSOS compatibility libraries
- Enhanced development tools
- Enhanced debug tools
- Real-time Ethernet functionality

Resources

The information provided here was produced with respect to the current performance and feature sets of RTAI and RealTime Linux. However, the RTAI and Zentropix home pages always contain the most up-to-date information.

P. Mantegazza, E. Bianchi and L. Dozioi work for Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano and can be reached at luca.bianchi@infementia.it.

S. Papacharalambous, S. Hughes and D. Beal work for Zentropix Computing, LLC and can be reached at daveb@zentropix.com. (Note: Zentropix was acquired by Lineo on February 1, 2000.)
