Linux and Real-Time: Current Approaches and Future Opportunities

Claudio Scordino, *Student Member, IEEE*
Giuseppe Lipari, *Member, IEEE*

***Abstract*— In the last years, there has been a considerable interest in using the Linux operating system in real-time systems, especially in control systems. The simple and elegant design of Linux guarantees robustness and very good performance, while its Open Source license allows to modify and change the source code according to the user needs.**

However, Linux has been designed to be a general-purpose operating system. Therefore, it presents some issues, like unpredictable latencies, limited support for real-time scheduling, and coarse-grain timing resolution that might be a problem for real-time applications.

For these reasons, several modifications have been proposed to add "real-time" features to the kernel. In this paper, we give a brief description of the many existing approaches to support real-time applications in Linux. Moreover, we take a look at the expected trends, presenting what we believe will be the future of Linux for what concerns real-time support.

I. Introduction

Linux is a full-featured operating system, originally designed to be used in server or desktop environments. Since then, Linux has evolved and grown to be used in almost all computer areas — among others, embedded systems and parallel clusters [1].

In the last years, there has been a considerable interest in using Linux for real-time control systems, from both academic institutions, independent developers and industries. There are several reasons for this raising interest. First of all, Linux is an open source project, meaning that the source code of the operating system is freely available to everybody, and can be customized according to the user needs, provided that the modified version is still licensed under the GNU General Public License (GPL). This license allows anybody to redistribute, and even sell, a product as long as the recipient is able to exercise the same rights (access to the source code included). This way, a user (for example, a company) is not tied to the OS provider anymore, and is free of modifying the OS at will. The GPL open source license helped the growth of a large community of researchers and developers who added new features to the kernel and ported Linux to new architectures. Nowadays, there is a huge amount of programs, libraries and tools available as open source code that can be used to build a customized version of the Linux OS.

Another reason for the usage of Linux in real-time systems is its wide popularity and success. It has the simple and elegant design of the UNIX OSs, which guarantees a very stable, robust and secure system. Moreover, it has excellent performance and a good protocol stack implementation. The portability of code from different UNIX operating systems is ensured by the well-known *"Portable Operating System Interface"* (POSIX) API. This is an IEEE standard defining the basic environment and set of functions offered by the operating system to the application programs. Finally, the huge community of engineers and developers working on Linux makes finding expert kernel programmers very easy.

Thus, when compared to commercial real-time operating systems (RTOSs) in terms of cost of development, Linux has good chances to be the winner. Unfortunately, the standard mainline kernel (as provided by Linus Torvalds) is not adequate to be used as RTOS. Linux has been designed to be a general-purpose operating system (GPOS), and thus not much attention has been dedicated to the problem of reducing the latency of critical operations. Instead, the main design goal of the Linux kernel has been (and still remains) to optimize the average throughput (i.e., the amount of "useful work" done by the system in the unit of time). As we will show in Section III, a Linux program may suffer high latencies in response to critical events.

To overcome these problems, many approaches have been proposed in the last years to modify Linux, in order to make it more "real-time". Kernel developers have worked in parallel toward the goal of reducing the worst-case latency of the standard Linux kernel, and proposed some possible solutions. At the same time, a new approach (called *Resource Reservation*) is slowly making its way to real-time system programming, and many Linux-based implementations of this approach are already available.

In this paper, we discuss the state-of-the-art of the different approaches to a Linux-based RTOS, and we take a look at the future trends. The paper is organized as follows. In Section II, we describe the problems in supporting real-time activities using the standard Linux kernel. Then, we present the possible approaches to create a real-time version of Linux. In particular, in Section III we describe the approach called Interrupt Abstraction, and we present some implementations of this mechanism. In Section IV, instead, we present some techniques to add real-time capabilities to the standard Linux kernel. Finally, in Section V we state our conclusions and

G. Lipari is with Scuola Superiore S.Anna, piazza Martiri della Libertà 33, 56127 Pisa, Italy, lipari@sssup.it

C. Scordino is with Computer Science Department, University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy, scordino@di.unipi.it

[1]Linux currently supports almost every hardware processor, including x86, AMD x86-64, ARM, Compaq Alpha, CRIS, DEC VAX, H8/300, Hitachi SuperH, HP PA-RISC, IBM S/390, Intel IA-64, MIPS, Motorola 68000, PowerPC, SPARC and UltraSPARC.

try to predict the next steps of Linux development for what concerns real-time support.

II. Towards a Real-Time Linux Kernel

A. Problems Using Standard Linux

There are several issues that must be analyzed in supporting real-time activities in a general-purpose operating system like Linux. All these issues are related to non-deterministic behaviours of the system, that make real-time processes experience latencies of unpredictable length during execution [1], [2]. All real-time applications, however, are time-sensitive activities having strict timing constraints (like deadlines) that must be satisfied, otherwise the system does not work properly.

The "latency" of a OS can be defined in many different ways. In general, latency is the time it takes between the occurrence of an event and the beginning of the action that will respond to the event. In the case of real-time control applications, it is often defined as the time between when the interrupt signal arrives to the processor (signaling that an external event like a sensor reading has occurred) and when the handling routine starts to execute (for example the real-time task that will respond to the event). In the development of critical real-time control systems, it is necessary to account for the worst-case scenario; Hence we are particularly interested in the maximum latency values.

In the 2.4.x versions of Linux, the maximum latency can be very high: for example, it can go up to 230 msec on a native standard kernel running on a Desktop computer [3]. Such a large interval of time is considered inadequate even for soft real-time applications. A control application requiring a sampling rate of 10 Hz (and hence a sampling period of 100 msec) cannot be safely executed in real-time on Linux 2.4.17, as in the worst-case up to two invocations can be delayed or even skipped.

The two main sources of latency in general-purpose operating systems are *task latency* and *timer resolution*. Task latency is experienced by a process when it cannot preempt a lower priority process because this is executing in kernel context (i.e., the kernel is executing on behalf of the process). Typically, monolithic operating systems do not allow more than one stream of execution in kernel context, so that the high priority task cannot execute until the kernel code either returns to user-space or explicitly blocks. This is equivalent to having a *lock* for all the kernel code: Whenever a task invokes a kernel routine, the kernel is *locked*, and no other kernel activity (except very low-level interrupt handling, commonly referred as *top-half* in Linux) can be executed. In case of Linux 2.4.x, many portions of the kernel code require a considerable execution time. If a high priority task is activated in response to an interrupt while the kernel is locked, it must wait for the kernel lock to be released before starting execution.

Another source of latency is related to timing resolution. Every operating system needs to keep track of the flow of time, because a large number of kernel functions (e.g., process scheduling) are time-driven. Operating systems keep track of the time through an electronic timer circuit that issues a hardware interrupt after a pre-programmed amount of time. When the timer issues the interrupt, the kernel knows that the specified interval of time is elapsed. Typically, general-purpose operating systems like Linux set the system timer in order to have periodic interrupts at a certain frequency. The value of the period is called *tick* and it is often a configurable option which depends on the processor speed. For example, in Linux 2.6 the tick value can vary between 1 msec (on fast processors) up to 40 msec (on slow machines). The periodic tick rate directly affects the granularity of all timing activities and it is one of the major causes of latency in operating systems. The kernel, in fact, is not able of measuring (or deferring activities for) intervals of time below a certain threshold. This represents a problem in real-time systems which need an accurate estimation of the current time and the execution of tasks at precise instants.

Finally, another problem in supporting real-time processes in general-purpose operating systems is the limited support for proper real-time scheduling policies. Linux provides the POSIX-compliant SCHED_FIFO and SCHED_RR policies, that are simple fixed priority schedulers. Although fixed priority is an adequate solution for real-time scheduling in embedded systems, it is not suitable for supporting real-time activities in general-purpose operating systems. Notable drawbacks of fixed priority schedulers are the fairness and the security among processes [4]. In fact, if a regular non-privileged user is enabled to access the real-time scheduling facilities, then she can rise her processes to the highest priority, starving the rest of the system. On the other hand, it is very difficult to provide real-time guarantees if only privileged users are allowed to access the scheduling facilities. Moreover, even trusted users may crash the system due to some mistake during development and debugging.

B. Classification of Linux-based RTOSs

For alle the above reasons, the standard Linux kernel is not suitable for supporting real-time control applications. Thus, during the last years, several approaches have been proposed to add real-time features to the kernel. These techniques can be grouped in the following two classes of approaches:

1) *Interrupt Abstraction*, which adds a new abstraction layer beneath the kernel to take full control of interrupts and system timers. This approach creates a hard RTOS that executes Linux as a background task. We describe the approach in Section III;
2) *Kernel Preemption* approaches, that make the behaviour of the system *more deterministic*, by improving kernel preemption, response times and timing resolution. These techniques are described in Section IV.

III. Interrupt Abstraction

The approach based on Interrupt Abstraction consists of creating a layer of virtual hardware between the standard Linux kernel and the computer hardware, as shown in Figure 1. This layer is also called *Real-Time Hardware Abstraction Layer* (RTHAL) [5], although it only virtualizes interrupts. Then, a separate complete *real-time subsystem* that consists of a RTOS and a set of real-time tasks and device drivers, runs together with the Linux OS.

The mechanism is the following. Every interrupt source is marked as real-time or non real-time. Real-Time interrupts are served by the real-time subsystem, whereas non-real-time interrupts are managed by the Linux kernel. To avoid latencies when executing real-time code, every time an interrupt arrives (the arrow marked with **(a)** in Figure 1), the RTHAL checks if it is a real-time interrupt. If so, the interrupt is immediately served by the real-time subsystem. A non-real-time interrupt, instead, is not forwarded to the Linux kernel immediately, but it is stored in a “pending interrupts” vector. The pending interrupts (and all other Linux activities) can be served only when no other real-time activity is running (arrow marked with **(b)** in Figure 1). In practice, the resulting system is a multithreaded RTOS, in which the *standard Linux kernel is the lowest-priority task*. The Linux kernel, and all the normal Linux processes are managed by the abstraction layer as the lowest priority task — the Linux kernel only executes when there are no real-time tasks to run and the real-time kernel is inactive.

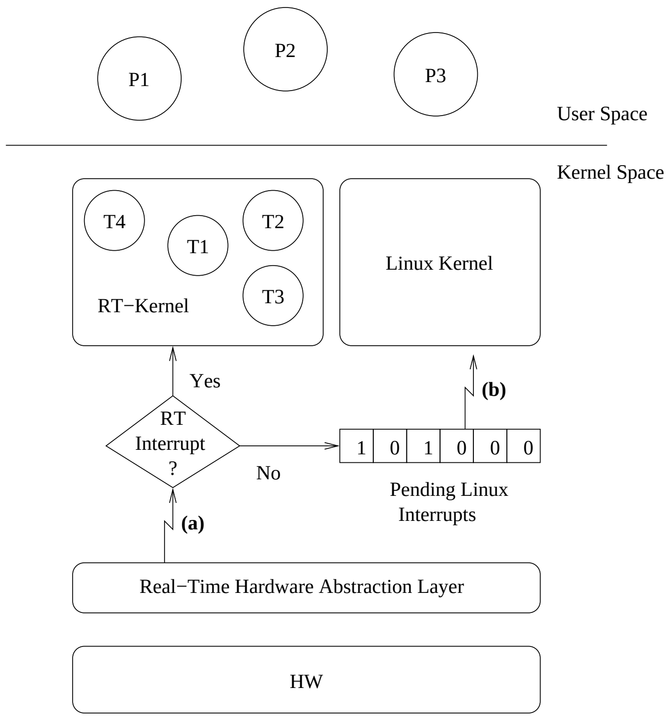

Fig. 1. Interrupt Abstraction.

Three main modifications must be done to the Linux kernel in order to virtualize the hardware and take full control of the machine. The abstraction layer must:

1) take direct control of all the hardware interrupts. The new interrupt handler intercepts all hardware interrupts, and checks whether the interrupt is related to a real-time activity or not, according to the mechanism described in Figure 1;
2) take the control of the hardware timer (8254 and APIC when available) and implement a virtual timer for Linux;
3) remove the basic control of the hardware interrupts from Linux by replacing all the `cli` and `sti` function calls (disable and enable interrupt flag, respectively) from the kernel code so that Linux cannot disable hardware interrupts (but only their virtual counterparts).

The Interrupt Abstraction approach has been successfully implemented in some existing RTOSs, the most famous being RTLinux and RTAI. **RTLinux** is a patch developed at *Finite State Machine Labs* (FSMLabs) to add hard real-time features to the standard Linux kernel [6]. The project started in 1995 and it is released in two different versions: an Open Source (under GPL license) version, and a more featured commercial version. The RTLinux patch implements a small and fast RTOS, compliant with the POSIX 1003.13 “minimal real-time system” profile. This means that it has basic thread management, IPC primitives, semaphores, signals, spinlocks, FIFOs, etc. Some function calls, however, do not follow the POSIX standard. RTLinux is covered by US Patent 5885745 issued on November 30th, 1999. The patent is not valid outside of the USA, but FSMLabs has expressed its intention to enforce the patent. This has generated a massive transition of community developers efforts towards RTAI.

RTAI is the acronym of *“Real-Time Application Interface”* [7], [5]. The project started as a variant of RTLinux in 1997 at Dipartimento di Ingegneria Aerospaziale of Politecnico di Milano (DIAPM), Italy. The project is under LGPL license, and it is supported by a large community of developers based upon the open source model. Although the RTAI project started from the original RTLinux code, the API of the projects evolved in opposite directions. In fact, the main developer (prof. Paolo Mantegazza) has rewritten the code adding new features and creating a more complete and robust system. With respect to the open source version of RTLinux, RTAI has a greater amount of supported architectures and a larger number of mechanisms for the communication between processes.

An in-depth comparison of the latency between the standard Linux kernel 2.4 and RTAI on a platform with an Axis ETRAX processor has been done in [8]. The main results of the experiments are summarized in Tables I.The values are measured without system load and when a load is applied, respectively. The upper part of Table I shows the values of the interrupt latency (i.e., the time between the interrupt arrival and the execution of the interrupt handler). Notice that, on average, the RTHAL imposes a slight increase in latency, due to the additional overhead of intercepting every interrupt with the RTHAL. However, the maximum latency values using RTAI are much smaller than using a standard kernel (especially for a loaded system), meaning that the determinism and the responsiveness of the system have been actually improved.

The bottom part of Table I shows the values of the task latency (i.e., the time between the interrupt arrival and when the task starts processing). Task latency is essentially composed by two components: the interrupt and the scheduling latencies. In this case, the behaviour of the two systems differs even in those situations where the interrupt latency values were almost close. This difference is due to how the Linux scheduler works: while on RTAI a real-time process is scheduled maintaining interrupts disabled, on Linux the interrupts are re-enabled after the interrupt handler finishes, leading to much more non-

			Linux 2.4	RTAI
Interrupt latency	Idle system	Avg.	4.3	5.8
		Max.	32.5	25.9
	Loaded system	Avg.	14.3	17.9
		Max.	162.9	64.9
Task latency	Idle system	Avg.	49.7	33.2
		Max.	332.3	68.0
	Loaded system	Avg.	3147.5	63.0
		Max.	84585.0	142.0

TABLE I
INTERRUPT AND TASK LATENCY IN THE STANDARD LINUX 2.4 AND RTAI. ALL NUMBERS ARE IN MICROSECONDS.

determinism.

FSMLabs, the owner of RTLinux, did not clear up the uncertainty around the legal repercussion of its patent on RTAI. For this reason, the RTAI community has developed the *Adaptive Domain Environment for Operating Systems* (Adeos) nanokernel as alternative for RTAI's core, to get rid of the old kernel patch and exploit a more structured and flexible way to add a real-time environment to Linux [5]. The purpose of the Adeos nanokernel is not limited to be the new RTAI's core, but it is to provide a flexible environment for sharing hardware resources among multiple operating systems (or among multiple instances of the same OS).

A. Advantages

It is important to highlight the advantages of using the Interrupt Abstraction approach. First of all, the latency reduction is really effective: measurements show a maximum latency below the microsecond [9] on a Intel Pentium M processor at 1.60 GHz. This allows the implementation of very fast control loops for applications like vibrational control. Also, thanks to the interrupt virtualization, it is possible to use a full-featured OS like Linux for the non-real-time activities. As a matter of fact, even the most critical control application includes non real-time activities, like logging and monitoring, man-machine interface, remote access through Internet, and so on. Using a system like Linux can reduce considerably the effort in developing this part of the system, and the programmer can concentrate on the most critical part. Finally, a further advantage is the possibility of developing and then executing the code on the same hardware platform, simplifying considerably the complexity of the development environment.

B. Limitations of RTLinux and RTAI

Both RTLinux and RTAI in their basic versions suffer from some software engineering and programming problems. As shown in Figure 1, the real-time subsystem (RTOS and tasks) executes in the same memory space and with the same privileges as the Linux kernel code. This means that there is no protection of memory between the real-time tasks and the Linux kernel. The real-time tasks are typically executed as modules dynamically loaded into the kernel. Therefore, a real-time task with errors (like wrong memory references, or unbounded execution time) may crash the entire system. Such situation is frequent during debugging and development, and it is a very common experience for programmers of such systems to reboot the computer several times before identifying the error. Both the commercial version of RTLinux and the most recent versions of RTAI partially solved this problem. In particular, RTAI supports the LXRT interface that lets developers try out real-time tasks in user space, where memory protection is enabled, at the cost of some more latency. Once the task has been properly debugged, it can be executed on RTAI without changing the task code. The LXRT mechanism has evolved in the Xenomai system that we descibe in the next section.

Another problem is the communication with the non-real-time Linux activities. In particular, the real-time subsystem *cannot use the Linux device drivers*. For example, both RTLinux and RTAI have their own network protocol stacks for communicating through Ethernet and with the serial driver, because the real-time tasks cannot use the Linux protocol stack. Therefore, in the same system, there is duplication of code for both the real-time and the non-real-time parts. Moreover, the effort of developing device drivers is always a consistent part of the development.

C. The Xenomai approach

A spin-off of the RTAI project [2], Xenomai [10] brings the concept of virtualization one step further. Like RTAI, it uses the Adeos nanokernel to provide the interrupt virtualization, but it allows a real-time task to execute in user space. Xenomai uses extensively the concept of domain provided by Adeos. In particular, Xenomai defines a *primary domain*, which is controlled by the RTOS (called *RT-Nucleus*), and the secondary domain, which is controlled by the Linux scheduler. A real-time task can execute in user space or in kernel space. Normally, it starts in the *primary domain*, where it remains as long as it invokes only the RTOS API. When the real-time task invokes a function belonging to the Linux standard API or libraries, it is automatically *migrated* to the secondary domain, under the control of the Linux scheduler. However, it keeps its real-time priority, being scheduled with the SCHED_FIFO or SCHED_RR Linux policies. While the real-time task is in the secondary mode, it can experience some delay and latency, due to the fact that it is scheduled by Linux. However, at any time after the function call has been completed, it can go back to the primary mode by explicitly calling a function. In this way, at the cost of some limited unpredictability, the real-time programmer can use the full power of Linux also for real-time applications. In fact, real-time tasks can run in their own memory space and are protected from the other tasks. This isolation facilitates debugging and fault confinement, reducing considerably the development time, and adding robustness to software faults.

Regarding the latency, the tasks in primary domain experience latencies comparable with the execution on RTAI. In secondary domain, instead, the maximum latency is higher, but

[2]Xenomai is the evolution of the Fusion project (in its turn a generalization of the LXRT interface), which was an effort to execute real-time RTAI task in user space.

it is still acceptable. As stated by Philippe Gerum, Xenomai leader, improvements on the standard Linux latency can help Xenomai too. For this reason, Xenomai developers put a constant effort in ensuring the simplicity and minimal invasivity of their approach with respect to the Linux code, thus that it is possible to use Xenomai among with separate solutions (like the PREEMPT_RT presented in Section IV-C) proposed by other developers.

IV. Making the Kernel More Predictable

An alternative to interrupt and hardware abstractions consists on making the Linux kernel more deterministic, by improving some parts that do not allow a predictable behaviour. As we have seen in Section II, the main sources of unpredictable behaviour in Linux are the kernel latency, the timing resolution and the process scheduling [1], [2]. We now present all the solutions that have been proposed to address these issues.

A. Reducing Kernel Latency

Two different approaches were proposed to reduce kernel latency in the 2.4 version of the Linux kernel. These two approaches were the *Low Latency Patch* and the *Preemptible Kernel Patch*, respectively. The former patch was introduced by Ingo Molnar and then maintained by Andrew Morton [11]. Rather than attempting a brute-force approach (i.e., preemption) in a kernel that is not designed for it, this patch focuses on introducing explicit preemption points in blocks of code that may execute for long intervals of time. The idea is to find places that iterate over large data structures and figure out how to safely introduce a call to the scheduler. Sometimes this implies releasing a spinlock, scheduling and then reacquiring the spinlock, which is also known as *"lock breaking"*.

A different strategy has been proposed by Robert Love with MontaVista's Preemptible Kernel Patch. This patch makes the kernel preemptible, just like user-space: if a high priority task becomes runnable, the patch allows a context switch even if another process is running in kernel context. Hence, it becomes possible to preempt a process at any point, as long as the kernel is in a consistent state (i.e., no lock is held). Kernel preemption is subject only to Symmetric Multi-Processing (SMP) locking constraints (i.e., spinlocks are used as markers for regions of preemptibility). With the advent of Linux 2.6, Robert Love's patch has been accepted in the mainline kernel, thus that the Linux kernel has become a fully preemptive kernel [12], unlike most existing operating systems (UNIX variants included).

A comparison of the two techniques has been performed by Clark Williams [3] and is summarized in Table II. The hardware used for the experiments is a 700 MHz AMD Duron system with 360MB RAM and a 20GB Western Digital IDE drive attached to a VIA Technologies VT82C686 IDE controller. The experiments show that the maximum latency on a native 2.4.17 standard kernel can be as high as 232.7 msec, which is not a negligible value even on Desktop machines. The Preemptible Kernel Patch can reduce this value, but it is the Low Latency Patch that really makes the difference in the latency behaviour of the kernel, allowing a maximum latency of 1.3 msec. Obviously, the two techniques can also be combined together. In this case, the result is quite unexpected: the maximum latency measured is 1.2 msec, which is a small improvement with respect to the gain obtained using only the Low Latency Patch.

	Linux 2.4.17	Preempt. Kernel	Low Latency	Both Patches
Avg.	88 μsec	53.8 μsec	54.2 μsec	52 μsec
Max.	232.7 msec	45.3 msec	1.3 msec	1.2 msec

TABLE II

Average and maximum latency values using a standard Linux 2.4.17, the Preemptible Kernel and the Low Latency Patches.

B. Improving Timing Resolution

The fact that periodic timer interrupts are not suitable for real-time kernels is well known in the literature [1]. For this reason, most of the existing real-time kernels provide a *"High Resolution Timers"* (HRT) API, that issues the interrupts aperiodically — i.e., the system timer is programmed to generate the interrupt after an interval of time that is not constant, but that depends on the next event scheduled by the operating system. Often, these implementations exploit also processor-specific hardware (like the APIC on modern x86 processors) to obtain a better timing resolution (typically, in the order of microseconds, or even fraction of microseconds).

There are two different projects to provide HRT in the Linux kernel. The first project, called *High-Resolution POSIX Timers* [13], started in 2001 as a separate patch and never became part of the standard kernel.

Very recently, a newer API developed by Thomas Gleixner has been accepted into the 2.6.16 version of the mainline kernel [14]. Rather than using a "timer wheel" data structure, this implementation uses a time-sorted linked list, with the next timer to expire being at the head of the list. A separate red/black tree is also used to enable the insertion and removal of timer events without scanning through the list. A new type (called `ktime_t`) is used to store a time value in nanoseconds and it is meant to be used as an opaque structure. Interestingly, its definition changes depending on the underlying architecture. On 64-bit systems, it is just a 64-bit integer value in nanoseconds. On 32-bit machines, instead, it is a two-field data structure: one 32-bit value holds the number of seconds and the other holds nanoseconds. The order of the two fields depends on whether the host architecture is big-endian or not — they are always arranged so that the two values can, when needed, be treated as a single 64-bit value. Doing things this way complicates the header files, but provides efficient time value manipulation on all architectures.

C. The PREEMPT_RT patch

The latest modification, still at the level of proposed patch, is the PREEMPT_RT patch by Ingo Molnar [15]. This work brings the kernel preemption to an unprecedent level of sophistication by introducing the Priority Inheritance Protocol

Kernel	sys load	Aver	Max	Min	StdDev
Vanilla-2.6.12	None	5.8	51.9	5.6	0.3
	Ping	5.8	49.1	5.6	0.8
	lm. + ping	6.1	53.3	5.6	1.1
	lmbench	6.1	77.9	5.6	0.8
	lm. + hd	6.5	128.4	5.6	3.4
	DoHell	6.8	555.6	5.6	7.2
RT-V0.7.51-02	None	5.7	48.9	5.6	0.2
	Ping	7.0	62.0	5.6	1.5
	lm. + ping	7.9	56.2	5.6	1.9
	lmbench	7.3	56.1	5.6	1.4
	lm. + hd	7.3	70.5	5.6	1.8
	DoHell	7.4	54.6	5.6	1.4
Ipipe-0.7	None	7.2	47.6	5.7	1.9
	Ping	7.3	48.9	5.7	0.4
	lm.+ ping	7.6	50.5	5.7	0.8
	lmbench	7.5	50.5	5.7	0.9
	lm. + hd	7.5	50.5	5.7	1.1
	DoHell	7.6	50.5	5.7	0.7

TABLE III
LATENCY COMPARISON BETWEEN STANDARD LINUX, LINUX WITH THE PREEMPT_RT PATCH, AND ADEOS. ALL NUMBERS ARE IN MICROSECONDS.

in the kernel locks. The Priority Inheritance (PI) protocol, first proposed by Sha et al. [16], solves the problem of unbounded *priority inversion*. A priority inversion is when a high priority task must wait for a low priority task to complete a critical section of code and release the lock. If the low priority task is preempted by a medium priority task while holding the lock, the high priority task will have to wait for a long time. The priority inheritance protocol dictates that in this case, the low priority task *inherits* the priority of the high priority task while holding the lock, preventing the preemption by medium priority tasks.

In the general case (i.e., nested spinlocks, readers/writers locks) the priority inheritance mechanism is a complex algorithm to implement. Nevertheless, it can help reduce the latency of Linux activities even further, reaching the level of the *Interrupt Abstraction* methods.

In Table III we report the results of a comparison between a standard Linux (denoted as Vanilla-2.6.12), the same Linux with the RT patch applied, and the Adeos microkernel, used by both RTAI and Xenomai (denoted with Ipipe-0.7)[3].

On each kernel configuration, a number of standard tests have been run to stress the system and measure the worst-case latency. The interrupt latency (i.e., the time it takes from the raise of the interrupt signal to the execution of the first instruction of the interrupt handler) has been measured in all cases. As you can see from the table, the maximum latencies are quite high in the Vanilla kernel (in the order of half a millisecond), while the maximum latencies in the PREEMPT_RT kernel and with the Adeos microkernel are comparable. However, other tests seems to show a slight advantage to the Adeos approach. It is important to point out that these numbers are referred to *interrupt latency*, while task latency can be much higher and depends also upon timer resolution and scheduling latency.

[3]The results are provided by Paolo Mantegazza and are taken from `https://mail.rtai.org/pipermail/rtai/2005-October/013265.html`. We are not aware of the original source of these numbers.

D. Resource Reservations

As we have seen, the scheduling policies offered by Linux are not suitable for supporting the execution of real-time applications. A real-time general-purpose OS should support scheduling policies providing *temporal protection* among the running processes. This means that the timely execution of a process should not be affected by the behaviour of the other processes executing on the system. This way, if a process misbehaves, and tries to use all the resources of the system, it cannot starve the other processes. The same problem is present in the Interrupt Abstraction methods: if a real-time task enters an infinite loop of code, the other low priority activities in the system cannot execute anymore. It is important then to provide *temporal protection* among different tasks, similarly to the way the Linux kernel provides memory protection.

The Resource Reservation mechanism [17] is an effective way for providing such temporal protection in GPOSs. The basic idea behind the resource reservation technique is to *reserve* a fraction of the time to real-time applications. This way, real-time priorities can be safely used even by non-privileged users. The mechanism works as follows. Each real-time process is assigned a “reservation” (Q_i, T_i), meaning that the process is reserved the processor for a time of length Q_i every period T_i. During its execution, the task is executed at an appropriate real-time priority. However, if the task tries to execute for a longer time, then it is suspended and resumed later. In this way, each task is constrained to not use more than its reserved share — i.e., a maximum of Q_i every P_i units of time.

A real-time scheduler based on Resource Reservation has been developed for Linux 2.4.18 within the OCERA (*“Open Components for Embedded Real-time Applications”*) European project, and it is available as Open Source code [4], [18], [19], [20]. To minimize the modifications to the standard kernel code, the real-time scheduler has been developed as a loadable kernel module [21]. A small patch (called *“Generic Scheduler Patch”*) applied to the Linux kernel exports the necessary symbols and the relevant events to the real-time scheduler. Based on the information provided by the patch, the real-time scheduler modifies the task priority, raising the selected task to the maximum priority, and then calls the Linux scheduler. In practice, the standard Linux scheduler acts as a dispatcher for the external real-time scheduler. The interface to the scheduler has been exported through the standard `sched_setscheduler()` system call, adding a new scheduling policy, and extending the structure `sched_param`. The scheduler implements the CBS [22], [4], the GRUB [23], [24] and the GRUB-PA [19], [20] scheduling algorithms. The last algorithm allows to reduce energy consumption of embedded systems with Intel PXA250 [25] processors.

The real-time scheduler needs to know all the relevant events regarding the processes in the system (i.e., process creation, termination, blocking and unblocking). For this reason, the patch exports some *hooks* that are used to intercept

Hook	Idle	10 tasks	20 tasks	30 tasks
creation	119	117	107	105
termination	48	44	39	35
unblock	316	387	421	483
block	138	6431	8101	9164
budget exhaustion	202	252	276	312

TABLE IV

OVERHEAD INTRODUCED BY THE HOOKS. ALL NUMBERS ARE IN NANOSECONDS.

the interesting scheduling events. The execution of the hooks introduces an overhead which is *at most* 10 μsec (see Table IV) on a AMD Athlon XP at 1.6 GHz running Linux 2.4.27 with High Resolution Timers and Linux Trace Toolkit patches. Although not comparable with the values obtainable using Interrupt Abstraction, this overhead is acceptable for most soft real-time applications.

A Resource Reservation scheduling policy for Linux has been developed also by Davide Libenzi with the SCHED_SOFTRR project [26]. Using this policy, a task can run with real-time priority, but it is subject to a constraint on the maximum processor time it can consume. Thus, non-privileged users can have deterministic latencies when running time-sensitive applications, while system stability and fairness are enforced by the bound.

Another scheduling policy, called SCHED_ISO (that stands for "Isochronous Scheduling") has been implemented by Con Kolivas [27]. Also this policy does not require superuser privileges and is starvation-free. Tasks running under the SCHED_ISO policy actually execute as SCHED_RR unless the processor usage exceeds a specified limit (i.e., 70%). The value of this limit can be configured through the `proc` filesystem.

V. CONCLUSIONS

Linux has become very polular for supporting real-time applications for many reasons, among the other the availability of a huge amount of programs distributed with open source license, the robustness and flexibility of the kernel, its standard interface. Many projects have been proposed to make Linux more real-time, both by using the Interrupt Abstraction approach, and by directly modifying the internals (preemption patches, and resource reservations).

The choice of which Linux flavor to use for executing a real-time application depends entirely on the requirements of the application. For hard real-time applications with very small constants of time (below the milliseconds), it is still necessary to use RTLinux,RTAI or Xenomai, since they can provide very low latencies. Also, RTAI and Xenomai provide nice integration with control design tools, like Scilab/Scicos, Matlab/Simulink, etc.

On the other hand, thanks to the constant attention to reducing the latency of the standard Linux kernel, soft real-time applications, or even hard real-time applications with large constants of time, can be scheduled directly by the Linux scheduler, maybe with the help of a resource reservation scheduler like GRUB or SCHED_SOFTRR or SCHED_ISO.

In the future, we believe that the two approaches will merge into a single product, able to provide different levels of services and latencies to different applications. In this sense, Xenomai is pavig the way to such integration.

REFERENCES

[1] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of linux," in *Proceedings of the 8^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, San Jose, California, Sept. 2002, pp. 133 – 142.

[2] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "Supporting time-sensitive applications on a commodity os," in *Proceedings of the 5^{th} symposium on Operating systems design and implementation(OSDI'02)*, vol. 36, Boston, MA, Dec. 2002, pp. 165 – 180.

[3] C. Williams, *Linux Scheduler Latency*, Red Hat Inc., http://www.linuxdevices.com/articles/AT8906594941.html, Mar. 2002.

[4] L. Abeni and G. Lipari, "Implementing resource reservations in linux," in 4^{th} *Real-Time Linux Workshop*, Boston, MA, Dec. 2002.

[5] L. Dozio and P. Mantegazza, "Real-time distributed control systems using rtai," in *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, Hakodate, Hokkaido, Japan, May 2003.

[6] V. Yodaiken, "The RTLinux Manifesto," in *Proceedings of the 5^{th} Linux Expo*, Raleigh, North Carolina, Mar. 1999.

[7] *RTAI - Beginner's Guide*, http://www.aero.polimi.it/~rtai/documentation/articles/guide.html.

[8] M. P. Andersson and J. H. Lindskov, "Real-time linux in an embedded environment," in *Master of Science Thesis*, Lund Institute of Technology, Lund University, Sweden, Jan. 2003.

[9] P. Issaris, "RTAI Testsuite LiveCD," http://issaris.org/~takis/projects/rtai/livecd/.

[10] P. Gerum, *The XENOMAI Project, Implementing a RTOS emulation framework on GNU/Linux*, Nov. 2002.

[11] A. Morton, *Low Latency Patches*, http://www.zipworld.com.au/~akpm/linux/schedlat.html.

[12] R. Love, *Linux Kernel Development, 2^{nd} edition*. Novell Press, Feb. 2005.

[13] G. Anzinger, *High Resolution POSIX timers*, http://high-res-timers.sf.net/.

[14] L. W. News, *Kernel development*, http://lwn.net/Articles/167315/.

[15] I. Molnar, *Real-Time Preempt*, http://people.redhat.com/mingo/realtime-preempt/.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE transaction on computers*, vol. 39, no. 9, September 1990.

[17] C. W. Mercer, R. Rajkumar, and H. Tokuda, "Applying hard real-time technology to multimedia systems," in *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.

[18] I. Ripoll, P. Pisa, L. Abeni, P. Gai, A. Lanusse, S. Saez, and B. Privat, "WP1 - RTOS State of the Art Analysis: Deliverable D1.1 - RTOS Analysis," OCERA, Tech. Rep., 2002. [Online]. Available: http://www.ocera.org

[19] C. Scordino and G. Lipari, "Energy saving scheduling for embedded real-time linux applications," in 5^{th} *Real-Time Linux Workshop*, Valencia, Spain, 2003.

[20] ——, "Using resource reservation techniques for power-aware scheduling," in *Proceedings of the 4^{th} ACM International Conference on Embedded Software (EMSOFT)*, Pisa, Italy, Sept. 2004, pp. 16–25.

[21] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3^{rd} edition*. O'REILLY, Feb. 2005.

[22] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19^{th} IEEE Real-Time Systems Symposium*. Madrid, Spain: IEEE, Dec. 1998.

[23] G. Lipari and S. K. Baruah, "Greedy reclamation of unused bandwidth in constant bandwidth servers," in *IEEE Proceedings of the* 12^{th} *Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, June 2000.

[24] ——, "A hierarchical extension to the constant bandwidth server framework," in *IEEE Proceedings of the Real-Time Technology and Applications Symposium*. Taipei, Taiwan: IEEE Computer Society Press, May 2001.

[25] *Intel PXA250 and PXA210 Application Processors Developer's Manual*, Intel Corporation, Feb. 2002.

[26] D. Libenzi, "SCHED_SOFTRR linux scheduler policy," http://xmailserver.org/linux-patches/softrr.html.

[27] C. Kolivas, "Isochronous class for unprivileged soft RT scheduling," http://ck.kolivas.org/patches/SCHED_ISO.