An Asymmetric Real-Time Scheduling for Linux

Momtchil MOMTCHEV
Momtchil.Momtchev@lifl.fr

Philippe MARQUET
Philippe.Marquet@lifl.fr

Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille
France

Abstract

Real-time intensive signal processing applications have been traditionally deployed on various custom platforms. Meanwhile, the enterprise computing market had spurred the advent of inexpensive and powerful systems based on widely available processors. Today, SMP systems associating a potentially large number of recent processors are deemed to cope with the needs of the most demanding real-time applications. On the operating system side, GNU/Linux is gaining wider acceptance and extending GNU/Linux to tackle real-time application scheduling is a common approach.

We propose ARTiS, an asymmetric real-time scheduling for SMP systems. ARTiS ensures the possible preemption of a processor when the system has to schedule a real-time process. We have modified the GNU/Linux SMP scheduler to implement ARTiS. The evaluation of our approach shows significant improvements.

1. Introduction

In this paper we present an open and industrial standards-based operating system capable of meeting the requirements of real-time intensive signal processing and real-time image processing applications on common, commercially available multiprocessor platforms. These applications have been traditionally deployed on various custom platforms, which usually included specialized DSPs as multi-purpose processors were not meeting the computer power requirements. This platform diversity, both at the hardware and software level has a major impact on product costs and development times. The rapid evolution in general multi-purpose processors, as well as the wider and wider importance of shared-memory multiprocessors in the business world is however narrowing this gap. IA-64, Intel's recently introduced architecture offers the theoretical computational power needed by intensive signal processing applications, and being backed by the IA-32 commercial success, it could become a new industrial standard.

On the software side the emergence of GNU/Linux as a cross-industry backed solution is creating yet another industrial standard. GNU/Linux is already gaining wider acceptance, which is not limited anymore to Internet usage. The large and very active community allows to fully benefit from the open-source model. In this context, we are interested in studying the feasibility of using GNU/Linux as development and deployment platform for intensive signal processing applications. Our goals are to study the feasibility of using GNU/Linux, as well to eventually identify the modifications required to the Linux kernel before it meets the applications' requirements. As intensive signal processing is heavily computationally dependent, we restrained ourselves to multiprocessor platforms.

1.1. The Real-Time Requirements of Signal Processing

The IEEE's definition on a real-time system is "a system whose correctness includes its response time as well as its functional correctness". Further distinction is made between hard and soft real-time systems, with soft real-time being defined as "best effort" real-time, meaning that occasional deadline misses are not fatal for the application. Digital signal processing systems are usually hard real-time systems. Another important characteristic of real-time applications is whether they are periodic or event-based. A periodic real-time application is scheduled by a clock with one or more fixed periods. Such applications, like signal processing, are typically processing an input stream at a fixed rate and have a "regular" and predictable behavior with much weaker real-time requirements than event-based applications, usually in the order of 1 ms. Event-based applications pose a much higher demand on the operating systems, being non-predictable and usually irregular over the time. Numerous commercial real-time platforms address these issues, but this performance usually comes at the cost of a spartan development environment, lack of general-purpose run-time features and an overwhelming API diversity.

1.2. Real-Time and Linux

The current Linux kernel (2.4.12 at the time of this writing) supports a soft real-time scheduling mode for user processes, called `RT_FIFO`. The table 1 presents GNU/Linux (Linux kernel 2.4.9) response times on a idle 4-processor machine with a SCSI hard disk interface.

Table 1. Linux 2.4.9 / idle system

Freq.	Sysload	Deadlines met	Avg lat	Max lat
512	0	100%	7	35
1024	0	100%	7	18
2048	0	100%	7	158
4096	0	100%	7	20
8192	0	99.9+%	7	166

A “real-time” processes is scheduled at the given frequency for period of 30 minutes. Average and maximal latences are, in microseconds, the delays elapsed between the scheduling interrupt delivery and the process execution start time. A deadline is not met when a new scheduling interrupt occurs before the previous one has already been processed.

As we see the average response times are very close to the hardware limits and the worst response times are still acceptable. The table 2 presents the results of the very same test, but this time the machine being heavily-loaded.

Table 2. Linux 2.4.9 / “trashing” system

Freq.	Sysload	Deadlines met	Avg lat	Max lat
512	4	99.79%	10	2920
1024	4	99.91%	5	1380
2048	4	99.68%	4	1448

It’s striking to see that although average response times is barely increased, the worst response time is a few orders of magnitude greater. Two major factors influence worst response time under Linux, the first one being hardware interrupts deactivation for synchronization purposes. The Linux kernel tends to mask hardware interrupts to assure atomicity of critical regions, as this is usually faster then using semaphores. Some hardware drivers, as the IDE driver, are also known to run for large periods of time with interrupts disabled. As on multiprocessor systems interrupts are masked locally, the interrupt latency is less of a factor. The second factor lies in the UNIX architecture itself [2, 9]. Although Linux doesn’t include any original UNIX code, its internal kernel architecture follows quite closely that of UNIX [12, 3]. The kernel is monolithic and system calls are executed within the user process’ context. Asynchronous events, as hardware interrupts, are processed by bottom half handlers outside of the user process’ context. Neither the kernel upper half (the system calls) nor the bottom half are preemptive, meaning that a user process running inside a system call, has to finish its execution before a context switch is possible. Figure 1 illustrates the latency-inducing factors in Linux.

2. Existing Real-Time Add-Ons for Linux

The existing real-time add-ons for the Linux kernel are quite numerous and diverse, but still most of the solutions fall into one of two categories: high performance hard real-time systems, offering response times in order of few tens of microseconds and some degree of Linux compatibility (RTLinux, RTAI) and native Linux solutions, offering usually soft real-time and millisecond response times [8].

All of the currently available hard real-time add-ons depend on an additional real-time micro-kernel, which runs a modified Linux kernel as a user process. Micro-kernel architecture-based operating systems do generally have lower response times [14, 15]. RTLinux is a good example of this approach (Figure 2(a)). It is chronologically the first real-time add-on and it is currently enjoying the largest industrial support. The project goals are creating an operating system, capable of supporting both traditional Linux user processes and hard real-time processes with microsecond-order timing requirements. It is intended to be simple, portable and reliable. The supported multiprocessor architectures in its latest version are IA-32 and PowerPC. The operating system consists of a small hard real-time micro-kernel which offers little more than scheduling services and which runs Linux as its lowest priority process. Hardware interrupts are intercepted by the micro-kernel, which schedules the real-time interrupt handlers. A virtual interrupt controller exports non real-time interrupts to the Linux kernel. Communications between the real-time processes and Linux processes is supported through real-time FIFOs, implemented in a virtual character driver on the Linux side.

As the micro-kernel offers no services, real-time applications are expected to be completely self-contained, including drivers for any hardware devices used. No virtual memory is supported for the real-time processes and they all run in the same address space. The major drawback of this approach is that it presents a dualistic platform to the developer, one in which real-time processes do not benefit from the services of the Linux kernel, and Linux processes

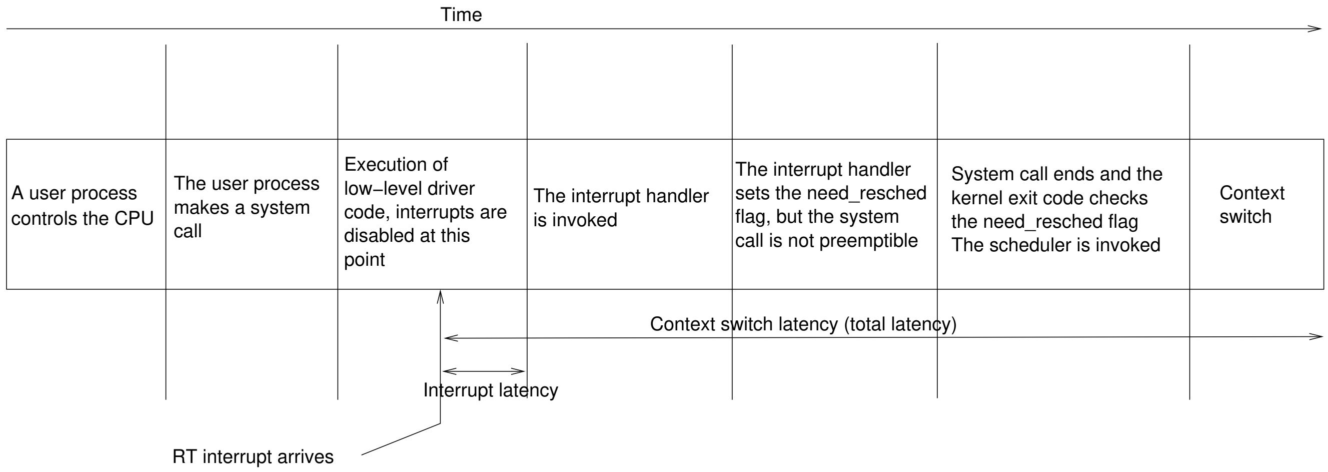

Figure 1. Latency sources in the Linux kernel

do not benefit from real-time enhancements. Such an environment is poorly suited to the development of large parallel applications. The newest version of RTLinux (RTLinux 3.0) supports an alternative way of achieving real-time on multiprocessors systems through processor reservation. A reserved processor is one that does not run a Linux process and is dedicated to real-time processing.

RTAI (Real-Time Application Interface) [6] is an early off-shoot of the RTLinux development. Even if it is been reimplemented from scratch in its latest version, it sill bears some important similarities to the RTLinux architecture. It is based upon a real-time micro-kernel, supporting the proprietary API RTAI and a modified version of Linux, running as a lowest priority process. An intermediate layer, the Hardware Abstraction Layer (HAL) is implemented in an effort to minimize the impact on the Linux kernel (Figure 2(b)). Although RTAI shares RTLinux's three level architecture, it presents an unique API to the developer through a kernel module, called LXRT, which exports real-time services to the Linux processes. Each Linux process using the real-time scheduler has an associated real-time process in the micro-kernel which executes the real-time handlers. The RTAI development is very performance oriented which has a somewhat negative aspect on its portability. A large part of the architecture is based around the Intel 8254 timer chip, present in the interrupt controller on traditional x86 motherboards. The only multiprocessor architecture currently supported is x86 with no near-term porting plans.

A more direct way to real-time, a fully preemptive Linux kernel, has also been proposed. However at the present moment none of these projects (MontaVista, TimeSys) has been able to fully implement it. The question of maintaining such a heavy modification to the Linux kernel, which is constantly evolving, remains open, too. Fixed preemption points is alternative, simpler approach, which comes very close to a fully preemptive kernel. This approach consists of adding static calls to the scheduler function to the longer paths in the upper part of the kernel. Several independent implementations are freely available on the Internet for both of the currently widely used kernel versions (2.2 and 2.4). Even if these implementations are far from commercial quality their performance is constantly increasing, as more and more "long" paths in the upper half of the kernel are identified and remedied. No implementation claims however hard real-time support.

Linux/RK (Resource Kernel) [10], and Linux/RT, developed by TimeSys, are examples of an alternative approach : a resource kernel. A resource kernel manages explicitly shared resources, enforcing strict application priorities. Although not a hard real-time solution on its own, a resource kernel enhances Linux with a strict resource management policy, permitting hard applications reserves of resources (limited for now to CPU time in both systems). Linux/RT 2.0 supports reserving processor time across processors in SMP systems. Both Linux/RT (TimeSys) and Hard Hat Linux (MontaVista) rely on RTAI to address hard real-time applications.

3. The Asymmetric and ARTiS Approaches

A native Linux solution has some important advantages over three-level solutions, especially when it comes to the development of large parallel applications. However all of the native solutions, with one exception, fail to offer hard real-time response guarantees. That one exception is cross-processor reservation in multiprocessor systems, which prevents the non-preemptive Linux kernel from running on one

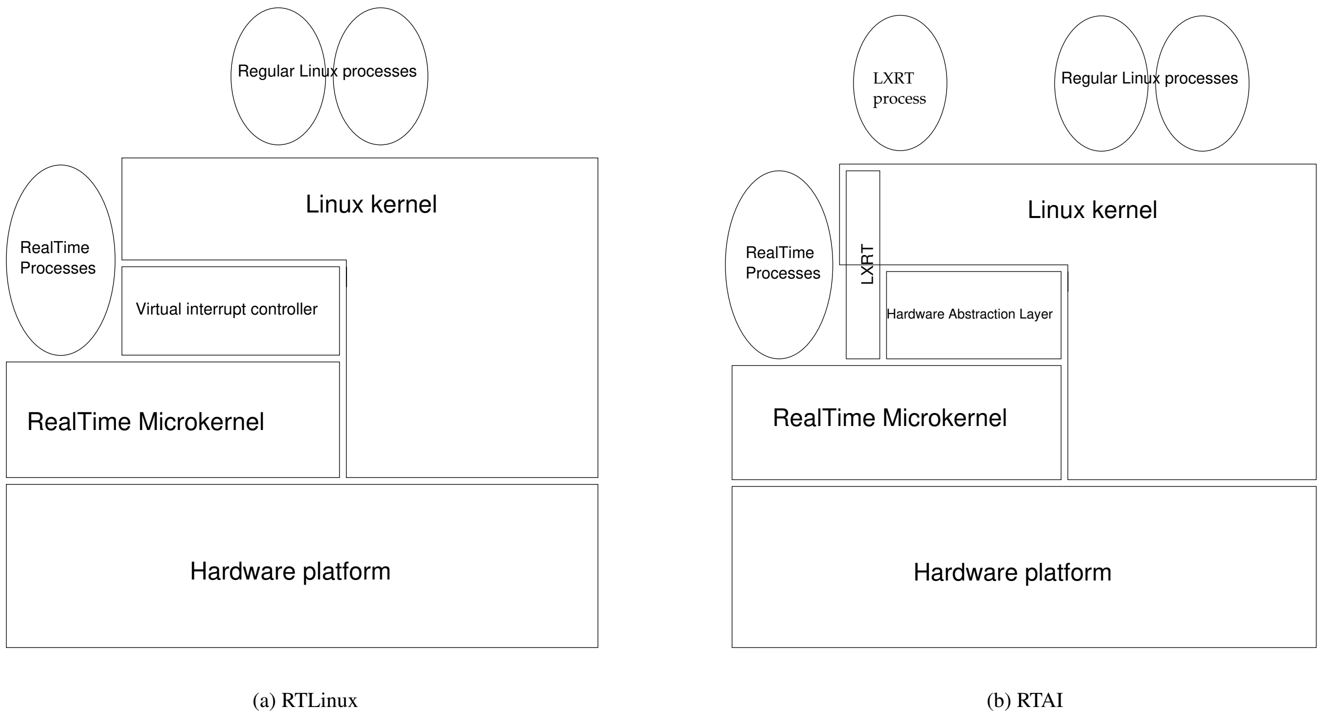

Figure 2. RTLinux and RTAI System Architecture: LXRT exports the RTAI interface to the userspace in RTAI

or more processor.

This approach, called Asymmetric Multi-Processing or AMP (as opposed to Symmetric Multi-Processing or SMP where each processor performs the same function from the operating system point of view) is simple, but has one major drawback : it results an uneven load distribution and imposes a de facto active blocking for the real-time processes, which cannot relinquish temporary the CPU. Classical AMP operating systems (some early VMS versions or MacOS prior to MacOS X) need also special support from the application in order to make use of the additional processors.

To achieve better efficiency we propose ARTiS, a slightly different model, which makes all the processors available to the Linux kernel, but prohibits its simultaneous execution on all of them. CPU-reservations are not bound anymore to a specific processor and could be serviced on any processor running user code. The model is defined by the scheduling rule that the number of processors running either user-code (user-code being preemptive) or the idle processes should be at least equal to the sum of the currently pending processor reservations. This has two major advantages, the first one being that the asymmetry is completely invisible to the developer, and the second one being that user code can still be scheduled on processors with pending guarantees.

4. Modifying the Linux Kernel

SMP support in Linux version 2.4 is already very mature [4] and the coarse-grained locking problems of Linux versions up to 2.2 are solved. Compatibility has its price however and different locking schemes continue to coexist in the kernel, with global locking still being used in some hardware drivers.

Global locking [13] could prove particularly dangerous for real-time performance, as a real-time process could be blocked waiting on a global lock held on another processor by a non-RT process executing in kernel mode.

Our tests show that global locking in the Linux kernel does not seem to influence real-time responsiveness in the millisecond range, but some hardware drivers, like the (E)IDE driver or some framebuffer drivers, could have a dramatic impact, adding latencies of 500 ms or even more. The same drivers are known to generally scale less and even to hinder system performance in SMP configurations.

Explicit using of multiple processors by the kernel code, by the Adaptec SCSI driver for example, could turn to be another source of culprit to real-time performance.

In this first test implementation processors are statically tagged as system or real-time processors via the `/proc` system. All processors run the same kernel.

The kernel entry routines were modified to distinguish between RT and non-RT tasks, so that non-RT tasks are not allowed to enter kernel mode on real-time processors.

4.1. Linux Bottom-half Handling

Bottom-half handling [1] is the name used to describe the processing of all the tasks queued by the hardware drivers, usually because they require a user context. These tasks are executed in kernel mode without having an UNIX process id associated with them. They are not scheduled by the process scheduler and have follow own rules of execution. To make the matter worse, they come in several flavors, each one having been added in a new major kernel release and then kept for compatibility. Some of them are queued for execution by a specific processor, while other are system-wide.

Bottom-half handlers are kernel-mode code so they should not be processed on real-time processors. Tasks queued without a specific processor are serviced only on the user processors. As processor-specific tasks are usually queued for execution on the same processor, we should not have bottom-half handlers marked for execution on real-time processors, except on a behalf of a real-time task.

The most recent kernel versions (2.4.5 and newer, 2.4.12 used) include a new way of scheduling bottom-half handlers that involves the scheduler, but the immediate performance benefit is still not clear.

4.2. Asymmetric Interrupt Handling

Interrupt handlers are not preemptive, so the asymmetric scheduling has to be extended to interrupts, too. Thus, we will distinguish between real-time interrupts, defined as interrupts being waited upon by real-time processes, and ordinary system interrupts. Although both of these handlers are not preemptive, on most system architectures interrupt handlers have shorter execution times than delivery times of inter-processor interrupts, so real-time interrupts are to be serviced on the user processors.

The only valid real-time interrupt in the current implementation is the RTC scheduling interrupt described in the next section.

4.3. Asymmetric Scheduling, System calls and the RTC interrupt

The Linux SMP scheduler is a quite simple one. Its implementation can be found in the function `schedule()` in `kernel/sched.c`. It is independently executed on each processor by a local interrupt with a fixed rate. On IA-32 platforms this interrupt is generated by the local interrupt controllers for each processor and should not be confused with the timer interrupt and kernel jiffies. The timer interrupt is a global interrupt, serviced by a random processor and its sole purpose is time-keeping in Linux/SMP. At each invocation the scheduler will select the next highest priority process and if the processor is currently running preemptive code (as defined by non-kernel code) it will trigger a context switch. Otherwise a flag is set (`need_resched`) and the context switch is delayed until the current process leaves the kernel. The scheduler can also be invoked explicitly (for example when a process is explicitly blocking on I/O).

As we wanted to preserve maximum compatibility with the unmodified kernel, we decided to not modify the standard timer interrupt handling, but rather to schedule real-time processes using another timing source – the RTC (Real-Time Clock) device. The RTC device driver calls the scheduler function explicitly. As the RTC period is not fixed, the standard local timer interrupt is still delivered even on real-time processors.

In an effort to minimize expensive inter-processor communication the RTC interrupt is delivered always on a real-time processor.

In order to honor the processors' reservations, the scheduler was modified to distinguish between processes currently executing in kernel and in user mode. This done by examining the return address in the process' stack. Kernel-mode processes are not eligible for scheduling when this will result failure to meet the asymmetric rule.

As changing execution mode under Linux does not involve the scheduler, dynamic allocation of real-time processors will require some additional book-keeping modifications to the system call interface for tracking the number of system calls currently running.

4.4. Real-Time API

We decided to not modify the existing soft real-time interface in the Linux kernel – that presented by the RTC (Real-Time Clock) character device, as it suits our needs. Periodic RT tasks have to set their scheduling policy to `RT_FIFO` via the POSIX call `sched_setscheduler` and then to start the real-time clock via an `ioctl` call.

The RTC itself is implemented as character device delivering one word per tick.

4.5. Performance

In order to evaluate the performance potential of an asymmetric kernel we tried scheduling a periodic real-time task at increasing frequencies and varying the system load, while keeping an eye for missed deadlines.

All tests were performed on a 4-processor Xeon II at 450MHz.

We are currently using linux-2.4.9 with the lowlat patch and our custom modifications.

Time is measured via the time stamp counter present on Pentium-class CPUs and reflects the delay elapsed between the moment the interrupt is delivered and until the user real-time process starts executing. As we are not interested in measuring the underlying hardware performance, we ignore hardware-induced interrupt delivery time. This doesn't void the results since an excessive interrupt delivery time could come still result lost interrupts, it simply means that the above times reflect the software-induced part of the latency, as this is what we are interested in. All times are in microseconds.

In order to stress the system at different levels we ran the tests with a constantly increasing number of actively paging processes, resulting in memory "trashing" – "trashing" describes a situation where the total current working set of all active processes does not fit in the memory. "Trashing" is known to dramatically bring down the responsiveness of the system.

Test runs lasted 30 minutes for each case.

System-load refers to the number of "trashing" processes running.

As one could expect from its architecture, the Linux kernel has fantastic average response times, that tend to drop severals order of magnitude under heavy load.

Average latency is the arithmetic average of the time elapsed after a hardware interrupt is registered and before the user-land handler is invoked. This does not include the time needed for the hardware signal to be delivered via the PCI bus to the CPU.

In-kernel interrupts is the number of times real-time interrupts interrupted kernel-mode code, meaning that an immediate context switch was not possible. Average in-kernel latency reflects the average latency in these cases. As this is clearly violates the asymmetric model, where do they come from?

There are two ways for a non-real-time task to enter kernel mode on a real-time processor - via the various bottom-half handling mechanisms and while the task is being scheduled. As of kernel version 2.4.9 most bottom half handlers that were using explicit per-processor scheduling or obsolete interfaces seem to be fixed, although some still persist. We are currently investigating how to take advantage of the new kernel mechanisms. However it is the scheduler, being the longest non-preemptive code path on real-time processors, that is the current performance limiting factor. A rather good proof could be obtained in comparing the 2.4.9-ARTiS results with 4 and with 8 "trashing" processes, knowing that the scheduler uses a O(N)-complexity algorithm, with N being the number of processes on the run-queue. As a result the average latency, caused by in-kernel interrupts seems to be affected only by the number of actually running processes.

A very interesting fact is that the average interrupt delivery time gets better on higher frequencies. This is not an error and we believe it is due to memory caches.

5. Conclusion and future works

We proposed ARTiS, a new scheduling policy of real-time processes on SMP systems that allows non real-time processes to be scheduled on all the processors of the system. ARTiS has been implemented as an extension of the current SMP Linux kernel. We evaluated the ARTiS performances on a pool of periodic real-time tasks with various frequencies. ARTiS compares favorably with the standard Linux kernel, specially under heavy load of the system.

Many technical improvements will be integrated into ARTiS in the short term:

- In order to avoid a ping-pong scenario, where 2 processes are actively using system calls, but only one processor is allowed to execute them (this scenario would cause 2 context switches per system call) a slight optimization is possible: the Linux kernel keeps track of the scheduling quantum usage of each process, in order to identify highly interactive processes as opposed to "heavier" background processes. Processes that tend to use their entire quantum are penalized in their dynamic priority. These processes are the ideal candidates for sharing a CPU with a real-time task.

- The scheduler function being non-preemptive, it is currently the major limiting factor in hard real-time response latency, with twice its maximum running time being the upper latency bound. We are currently investigating the possibility of adding fixed preemption points, which would permit aborting a context switch in progress [16].

- Scheduling multiple RT tasks with different periods is currently not supported, due to hardware limitations. Adding support for RT tasks with periods that are multiples of one another should prove easy, but reprogramming of the standard for IA-32 RTC clock at each tick (the so called 'one-shot' mode) is currently prohibitive.

- In order to alleviate the cost of the scheduling algorithm when many non real-time processes are ready, we may separate the processes in different lists; we can even consider a list for each priority level [11].

Table 3. Linux 2.4.9-reference

Frequency	Sysload	Deadlines met	Avg lat	Max lat	In-kernel IRQs	Avg In-kernel Lat
512	0	100%	7	35	2	26
1024	0	100%	7	18	1	18
2048	0	100%	7	158	3	64
4096	0	100%	7	20	3	11
8192	0	99.9+%	7	166	4	28
512	4	99.79%	10	2920	832	81
1024	4	99.91%	5	1380	660	75
2048	4	99.68%	4	1448	1299	50

Table 4. Linux 2.4.9-ARTiS+lowlat

Frequency	Sysload	Deadlines met	Avg lat	Max lat	In-kernel IRQs Lat	Avg In-kernel Lat
512	0	100%	3	20	0	N/A
1024	0	100%	3	46	0	N/A
2048	0	100%	3	76	0	N/A
4096	0	100%	4	48	12	5
512	4	100%	12	96	0	N/A
1024	4	100%	11	156	0	N/A
2048	4	100%	10	203	0	N/A
4096	4	100%	9	152	23	26
8192	4	99.99+%	10	242	18	23
512	8	100%	14	113	0	N/A
1024	8	100%	14	127	0	N/A
2048	8	100%	13	114	0	N/A
4096	8	100%	13	234	1	37
8192	8	99.99+%	13	242	18	23

- The local timer interrupt is currently delivered to real-time processors, even if they use another source of scheduling (the Real Time Clock interrupt). It is used for bookkeeping purposes and the minimal overhead does not justify the double bookkeeping.

ARTiS is a new way of adding real-time support to a general purpose operating system. This approach has advantages over dedicated real-time operating systems: real-time and non real-time processes coexists in the system, they use the same interface to the system, and they may naturally co-operate. Nevertheless, comparison between real-time operating systems is also a matter of predictable performances. Despite the objectives of the numerous real-time operating systems vary, our future works include exploring benchmark suite for parallel real-time operating systems evaluation.

We believe in the advance of shared-memory programming in the context of parallel computing. Proposition such as OpenMP [7, 5] that gives a simple API for developing parallel applications may be extended to the real-time field. Our future long term work is mainly focusing on the definition of a platform for parallel real-time applications based on SMP systems.

References

[1] T. Aivazian. *Linux Kernel 2.4 Internals, version 1.2*. Linux Documentation Project, Aug. 2001.

[2] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[3] I. T. Bowman. Conceptual architecture of the Linux kernel. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, Jan. 1998.

[4] R. Bryant, B. Hartner, Q. He, and G. Venkitachalam. SMP scalability comparisons of Linux kernels 2.2.14 and 2.3.99. In *4th Annual Linux Showcase and Conference*, Atlanta, GA, Oct. 2000.

[5] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, Oct. 2000.

[6] P. Cloutier, P. Montegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. DIAPM-RTAI position paper. In *Second Real Time Linux Workshop*, Orlando, FL, Nov. 2000.

[7] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan.-Mar. 1998.

[8] J. Keidel. Telemetry using soft-realtime. In *Workshop on Real Time Operating Systems and Applications and second Real Time Linux Workshop*, Lake Buena Vista, FL, Nov. 2000.

[9] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD UNIX*. Addison-Wesley, 1996.

[10] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Real Time Linux Workshop*, Vienna, Austria, Dec. 1999.

[11] Real-time scheduler project. `http://sf.net/projects/rtsched/`, 2001.

[12] D. A. Rusling. *The Linux Kernel*. Linux Documentation Project, 1999.

[13] P. R. Russell et al. Unreliable guide to locking. Linux Kernel Documentation, 2000.

[14] A. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 2nd edition, 1997.

[15] A. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2nd edition, 2001.

[16] Y.-C. Wang and K.-J. Lin. Some discussion on the low latency patch for Linux. In *Workshop on Real Time Operating Systems and Applications and second Real Time Linux Workshop*, Lake Buena Vista, FL, Nov. 2000.