

An Asymmetric Model for Real-Time and Load Balancing on Linux SMP*

Philippe MARQUET
Philippe.Marquet@lifl.fr

Julien SOULA
Julien.Soula@lifl.fr

Éric PIEL
Eric.Piel@lifl.fr

Jean-Luc DEKEYSER
Jean-Luc.Dekeyser@lifl.fr

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

April 2004

Abstract

We propose the ARTiS system, a real-time extension of GNU/Linux dedicated to SMP (Symmetric Multi-Processors) systems. ARTiS exploits the SMP architecture to guarantee the possible preemption of a processor when the system has to schedule a real-time task. The basic idea of ARTiS is to assign a selected set of processors to real-time operations. A migration mechanism of non-preemptible tasks insures a latency level on these real-time processors. Furthermore, specific load-balancing strategies allows ARTiS to benefit of the full power of the SMP systems: The real-time reservation, while guaranteed, is not exclusive and does not imply a waste of resources.

Simulations of the ARTiS performances have been conducted. The level of observed latency comfort the model proposition. A first implementation of ARTiS, while incomplete, also shows significant improvements compared to the standard Linux kernel.

1 Linux, SMP and Real-Time

Nowadays, there is a need for real-time guaranties in general purpose operating systems. "Soft real-time" application democratization is the trend: Everyone wants to play a video while burning a CD. Operating systems such as Linux take this request into account and consider some latency issues in place of the sole fairness of the traditional Unix.

Nevertheless, several application domains require hard real-time support of the operating system: The application contains tasks that expect to communicate with dedicated hardware in a time constrained protocol, for example to insure real-time acquisition.

Those same real-time applications require large amount of computational power: For example in the spectrum radio surveillance applications used to analyze the waveform signatures, the communications and the coverage have an increasing need of power with the apparition of the UMTS (greater bandwidth and more complex algorithms).

*This work is partially supported by the ITEA project 01010, HYADES

The usage of SMP (Symmetric Multi-Processors) to face this computational power need is a well known and effective solution. It has already been experimented in the real-time context [1, 2].

The real-time operating systems market is full of proprietary solutions. Despite the definition of standard POSIX interface for real-time applications [10], each vendor comes with a dedicated API. The lack of a major actor in the real-time community results in a segmented market and we are persuaded that the definition of an Open Source real-time operating system may encounter a success.

Our expectation is then to find an Open Source full operating system for real-time on SMP platforms:

- An Open Source system to gain conformance with a standard,
- A “full” operating system to allow the cohabitation of real-time and general purpose tasks in the system,
- Running on SMP platforms to face the intensive computing aspects of the applications.

Four main categories of operating systems are able to compete for the system we are looking for:

- Dedicated real-time operating systems (such as VxWorks),
- GNU/Linux, and especially the new Linux with its so-called “preemptible” kernel,
- Existing real-time extensions for the Linux kernel,
- Operating systems based on the Asymmetric Multi-Processing approach.

Dedicated real-time operating systems are readily available and extensively tested systems that deliver excellent hard real-time performances. Nevertheless, these systems mostly target embedded applications and the fact, for example, that a system such as VxWorks does not provide a full memory protection [6] makes it poorly suited for large applications (despite the announce of a new memory protection scheme in the upcoming VxWorks 6.0). Furthermore, these systems claim to support SMP architectures but consider them as a “multi mono-processor architecture”. One instance of the operating system is running on each processor, the application tasks must use synchronization or communication primitives based on a message-passing interface. This approach complicates the programming and deprives the SMP architecture of one of its most interesting features, the scalability.

The standard GNU/Linux system is an Open Source operating system, its availability on SMP platforms is now mature [5] and the system has excellent non real-time performances. Nevertheless, the architecture of the Linux kernel is by construction unable to guarantee any latency, neither at the interrupt level, nor at the user level: The Linux kernel is not preemptible and some of the works associated to the latency are deferred to the end of the ongoing system call.

Many attempts to improve the Linux kernel latencies have been proposed. The embedded Linux vendor MontaVista has introduced a rather simple and systematic patch of the Linux kernel [14] to ensure some preemption points in the kernel, and doing so, to reduce the kernel latency. This patch, maintained by Robert Love, has been adopted recently by the mainstream Linux kernel [15], mainly because it also implies a reduction of the latency targeted by multimedia applications.

Another, and complementary, approach is the so-called “low-latency” patch [20] of Ingo Molnar and Andrew Morton which adds some fixed preemption points into the kernel. If it reduces the kernel latency, the maintaining of this patch against the constant evolution of the kernel is an heavy job and the worst case latency evolution with the kernel “improvements” is still an affair of kernel experts.

Bernard Khun recently proposed a real-time interrupt patch [11] that introduces a notion of interrupt priority, allowing to reduce the worst case latency of the Linux kernel. Still, the extension of this mechanism to SMP systems is not obvious: A high priority interrupt execution may be delayed because it shares a lock with a low priority interrupt execution on an other CPU.

A well known solution that claims to add real-time capabilities to the Linux kernel is the so-called **co-kernel approach**. These Linux extensions consist in a small real-time kernel that

provides the real-time services and that runs the standard Linux kernel as a low priority task when no real-time task is eligible. The interrupts are rerouted to the Linux kernel by the real-time kernel; this virtualization of the Linux kernel interrupts allows the co-kernel to preempt the Linux kernel when needed. RTLinux [9, 21] and RTAI [7] are two famous systems based on this principle.

If the recent versions of RTLinux also target SMP systems [22], RTLinux comes with its “Open RTLinux Patent License” or with a commercial license and uses a FSM Labs patent that may prevent its usage and adoption, despite its current success.

This co-kernel approach suffers from providing a dualistic platform to the developer: Real-time processes do not benefit from the services of the Linux kernel, and Linux processes do not benefit from real-time enhancements. This is a major drawback, even if RTLinux supports communications between the real-time processes and the Linux processes through real-time FIFOs [8], or if RTAI provides a unique API to the developer through a kernel module, called LXRT, which exports real-time services to the Linux processes.

Another approach that exploits the SMP architecture relies on the shielded processors or **asymmetric multiprocessing principle**. On a multiprocessor machine, the processors are specialized to real-time or not: Real-time processors will execute real-time tasks while non-real-time processors will execute non-real-time tasks. Concurrent Computer Corporation RedHawk Linux variant [4, 3] and SGI REACT/pro, a real-time add-on for IRIX [18] follow this principle. However, since only real-time tasks are allowed to run on shielded CPUs, if those tasks are not consuming all the available power then there are some CPU resources which are wasted. In previous works, we had evaluated the effectiveness of this approach [13]. Our proposition of ARTiS enhances this basic concept of asymmetric real-time processing by allowing resource sharing between the real-time and non-real-time tasks.

2 ARTiS: Asymmetric Real-Time Scheduler

Our proposition is a contribution to the definition of a real-time Linux extension that targets SMPs. Furthermore, the programming model we promote is based on a user-space programming of the real-time tasks: The programmer uses the usual POSIX and/or Linux API to define his applications. These tasks are real-time in the sense that they are identified with a high priority and are not perturbed by any non real-time activities. For these tasks, we are targeting a maximum response time below $300\mu s$.

To take advantage of SMP architecture, an operating system needs to take into account the shared memory facility, the migration and load-balancing between processors, and the communication patterns between tasks. The complexity of such an operating system makes it looking more like a general purpose operating system than a proprietary real-time operating system (RTOS). A RTOS on SMP machines must implement all these mechanisms and consider how they interfere with the hard real-time constraints. This may explain why RTOS's are almost mono-processor dedicated. On the other hand, the Linux kernel is able to efficiently manage SMP platforms, but everybody agrees that the Linux kernel has not been designed as a RTOS. Technically, only soft real-time tasks are supported, via the two scheduling policies: FIFO and round-robin.

The ARTiS solution keeps both interests by establishing from the SMP platform an **Asymmetric Real-Time Scheduler** in Linux. We want to keep the full Linux facilities for each process and the SMP Linux properties but we want to improve the real-time behavior too. The core of the ARTiS solution is based on a strong distinction between real-time and non-real-time processors and also on migrating tasks which attempt to disable the preemption on a real-time processor. To provide this system we propose:

- The **partition of the processors in two sets**. A NRT CPU set (Non-Real-Time) and a RT CPU set (Real-Time). Each one has a particular scheduling policy. The purpose is to insure the best interrupt latency for particular processes running in the RT CPU set.

- **Two classes of RT processes.** They are all standard RT Linux processes. They just differ in their mapping:
 - Each RT CPU has just one bound RT Linux task, called **RT0** (a real-time task of highest priority). Each of these tasks has the guaranty that its RT CPU will stay entirely available to it. Only these user tasks are allowed to become non-preemptible on their corresponding RT CPU. This property insures a latency as low as possible for all the RT0 tasks. The RT0 tasks are the hard real-time tasks of ARTiS.
 - Each RT CPU can run other RT Linux tasks but **only** in a preemptible state. These tasks are called **RT1+** (real-time tasks of priority 1 and below). They can use CPU resources efficiently if RT0 does not consume all the CPU time. To keep a low latency for RT0, the RT1+ processes are automatically migrated to a NRT CPU by the ARTiS scheduler when they are on the way of becoming non-preemptible (when they call `preempt_disable()` or `local_irq_disable()`). The RT1+ tasks are the soft real-time tasks of ARTiS. They have no firm guaranties, but their requirements are taken into account by a best effort policy. They are also the main support of the intensive processing parts of the targeted applications.
 - The other, non-real-time, tasks are named “Linux tasks” in the ARTiS terminology. They are not related to any real-time requirements. They could coexist with real-time tasks and are eligible as long as the real-time tasks do not require the CPU. As for the RT1+, the Linux tasks will automatically migrate away from a RT CPU if they try to enter in non-preemptible code section on such a CPU.
 - The NRT CPUs mainly run Linux tasks. They also run RT1+ tasks when these are in a non-preemptible state. To insure the load-balancing of the system, all these tasks can migrate to a RT CPU but only in a preemptible state. When a RT1+ task runs on a NRT CPU, it keeps its high priority above the Linux tasks.
- **A particular migration mechanism.** This migration aims at insuring a low latency to the RT0 tasks. All the RT1+ and Linux tasks running on a RT CPU are automatically migrated toward a NRT CPU when they try to disable the preemption. One of the main changes which is required from the original Linux load-balancing mechanism is the removal of inter-CPU locks. To effectively migrate the tasks, a NRT CPU and a RT CPU have to communicate via queues. We implement an asymmetric lock-free FIFO with one reader and one writer to avoid any active wait of the ARTiS scheduler [19].
- **An efficient load-balancing policy.** It will allow to benefit from the full power of the SMP machine. Usually the load-balancing mechanism aims to move the running tasks across the CPUs in order to insure that no CPU is idle while some tasks are waiting to be scheduled on the other ones. Our case is more complicated because of the specificities of the ARTiS tasks. The RT0 tasks will never migrate, by definition. The RT1+ tasks should migrate quicker than Linux tasks to RT CPUs: The RT CPUs offer latency warranties that the NRT CPUs do not. To minimize the latency on RT CPUs and to provide the best performances to the global system, particular asymmetric load-balancing algorithms have been defined [17].
- **Asymmetric communication mechanisms.** On SMP machines, tasks exchange data by read/write mechanisms on the shared memory. To insure the coherence, critical sections are needed. Those critical sections are protected from simultaneous concurrent access by lock/unlock mechanisms. This communication scheme is not suited to our particular case: An exchange of data between a RT0 task and a RT1+ will involve the migration of the RT1+ task before this later takes the lock, to avoid entering in a non-preemptible state on a RT CPU. Therefore, an asymmetric communication pattern should use lock free FIFO in a one-reader/one-writer context.

ARTiS supports three different levels of real-time processing: RT0, RT1+ and Linux. RT0 tasks are implemented in order to minimize the jitter due to non-preemptible execution on the same CPU, but these tasks are still user-space Linux tasks. RT1+ are soft real-time tasks but they are able to take advantage of the SMP architecture, in particular for intensive computing. They are also able to trigger asymmetric communications that avoid inappropriate migrations

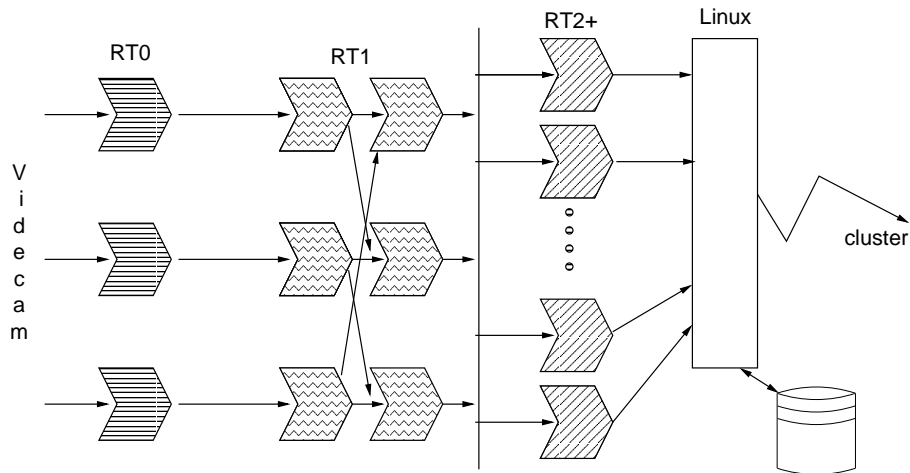


Figure 1: Example of a task architecture in an ARTiS application.

due to lock/unlock calls. A migration mechanism insures a good load-balance between all the processors. Eventually, Linux tasks can run without intrusion on the RT CPUs. Then they can use the full resources of the SMP machines.

3 Application Deployment over ARTiS

A real-time application on a SMP machine exhibits its full meaning when real-time constraints are combined with intensive computing. ARTiS is dedicated to this kind of applications. Real-time constraints are satisfied by RT0 tasks. Communications between RT0 and RT1+ provide intensive computing with data flow. Linux tasks insure additional processing. The load-balancing insures a dynamic mapping of the different tasks on the CPUs.

The implementation of an application on ARTiS requires to identify a specific level of real-time for each task of the algorithm and to map these tasks on the CPUs of the multiprocessor. To illustrate this, we use a real-time manufacturing quality management: For example, defect identification on a continuous production line running on a four CPUs SMP, three CPUs for the RT set and one for the NRT set. Several tasks may be identified, their communications and mapping are illustrated on the figure 1:

- Some videocam and/or sensors receive data periodically. Up to three RT0 tasks can manage the data acquisition with a latency compatible with real-time. Each of these tasks is assigned to a RT CPU.
- Directly connected to those tasks, intensive data processing with regular data structures has to be done for image processing. A static number of RT1 tasks are dedicated to this data-parallel processing (à la OpenMP). They should communicate with RT0 and with other RT1 tasks without inappropriate migration. They are also mostly bound to a RT CPU, but will migrate to the NRT CPU if they encounter a non-preemptible code. They make the most of the SMP facilities.
- Then defect identifications have to be achieved with irregular data structures concerning subsets of default parts. A dynamic number of RT2 are created. They have to communicate with RT1 and together. Because the number of tasks is dynamic, the load-balancing policy of ARTiS is very necessary: These tasks will run on the CPU dynamically chosen by the system to uniformly load the different CPUs.
- Finally, some defect fitting can be done with a local database or an external database (accessed via MPI) to produce statistics... This is no more RT processing and Linux tasks can take care of that. They are mainly mapped on the NRT CPUs but they can also use RT CPUs

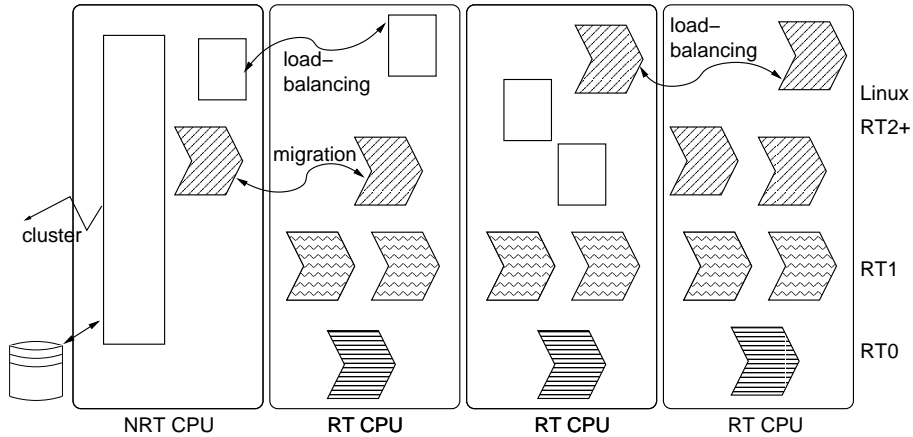


Figure 2: Mapping of the tasks on the RT and NRT processors.

when idle.

The figure 2 shows a possible mapping of those tasks on the two sets of processors: The RT and NRT.

4 Performance Evaluation

Before implementing the ARTiS kernel itself, we have conducted some experiments in order to evaluate the potential benefits of the approach in term of interrupt latency. We distinguished two types of latency, one associated to the kernel and the other one associated to the user tasks.

Measurement method The experiment consisted in measuring the elapsed time between the hardware generation of an interrupt and the execution of the code concerning this interrupt. The experimentation protocol was written with the wish to stay as close as possible to the common mechanisms employed by real-time tasks. The measurement task sets up the hardware so it generates the interrupt at a precisely known time, then it gets unscheduled and wait for the interrupt information to occur. Once the information is sent, the task is woken up and the current time is saved and the next measurement starts. This scheme is typical from the real-time applications, waiting for an hardware event to happen, processing data according to the new parameters, sending new information and returning to waiting mode. For one interrupt there are five associated times, corresponding to different locations in the executed code (Figure 3):

- t'_0 , the interrupt programming,
- t_0 , the interrupt emission, it is chosen at the time the interrupt is launched,
- t_1 , the entrance in the interrupt handler specific to this interrupt,
- t_2 , the end of the interrupt handler,
- t_3 , the entrance in the user-space RT task.

We conducted the experiments on a 4-way Itanium II 1.3Ghz machine. It ran instrumented Linux kernel versions 2.6.0, 2.6.1, and later, 2.6.4. Globally, we could notice improvements along the versions, in particular when kernel preemption was activated. Hence, we are only presenting the result from the latest tested version. The timer on which are based all the measurements is the `itc` (a processor register counting the cycles) and the interrupt was generated with a cycle accurate precision by the PMU (a debugging unit available in each processor [16]).

Even with a high load of the computer, bad cases leading to long latencies are very unusual. Thus, a large number of measures are necessary. In our case, each test is composed of 300 mil-

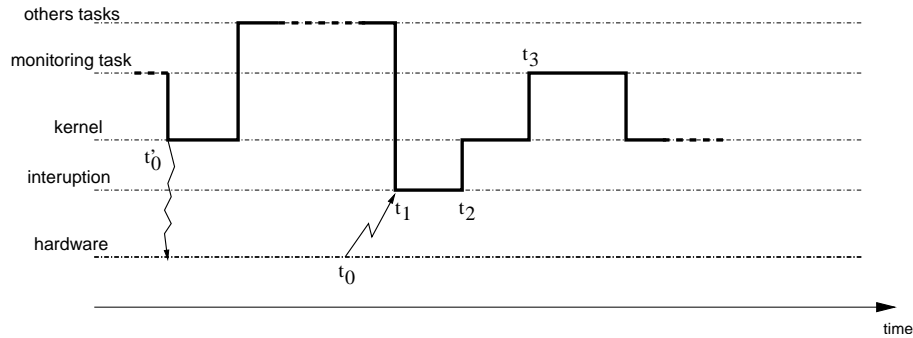


Figure 3: Chronogram of the tasks involved in the measurement code.

lions of measures, making each test about 8 hours long. With such a length, the results are reproducible.

Interrupt latency types From the 4 measurement locations, two interesting values can be calculated. Their interest comes from the ability to associate them to common programming methods and also from the significant differences along the tested configurations. Those two kinds of latencies can be described as follow:

- The **kernel latency**, $t_1 - t_0$, is the elapsed time between the interrupt generation and the entrance inside the interrupt handler function (`pfm_interrupt_handler()` in our case). This is the latency that a driver would have if it was written as a kernel module following the usual design method.
- The **user latency**, $t_3 - t_0$, is the elapsed time between the interrupt generation and the execution of the associated code in the user-space real-time task. This is the latency a real-time application would have if it was written entirely in the user-space. In POSIX systems, there are two ways for the application to be notified of the interrupt: Either by handling a signal or by returning from a blocking system call. The signal handling gave a minimum user latency of $4\mu s$ instead of $2\mu s$ but the maximum values were very similar. The blocking call method is more efficient so all the exposed results are only based on a blocking system call (a `read()`).

The real-time tasks designed to run in user-space are programmed using the usual and standard POSIX interface. This is one of the main advantage that ARTiS provides. Therefore, within the ARTiS context, the user latency is the most important one to study and analyze.

One could notice that the time t_2 was not cited. Actually, $t_2 - t_1$ represents the jitter of the interrupt handler execution time. However, it is fairly stable in regard to the configurations and is always very small compared to the other two intervals (approximately $1\mu s$).

The “ideal” ARTiS configuration In order to estimate the performance of the ARTiS system before its full completion, we had to simulate it. So, based on a vanilla Linux kernel, we set up a configuration which is asymmetric in the same way that ARTiS is. It consists in a manual binding of each task on the CPU on which ARTiS would execute it:

- The measurement task, equivalent to a RT0 process, is run on a specific CPU (a RT CPU),
- The tasks doing CPU load exclusively in user-space, equivalent to the RT1+ or NRT tasks never disabling preemption, are run on each CPU,
- All the other tasks, equivalent to the NRT tasks disabling preemption, are run on another specific CPU (a NRT CPU).

Additionally, the interrupts which are not used by the measurement task are bound to a NRT CPU. The repartition was pre-selected and non-modifiable during the tests.

This environment is purely static and is equivalent to a snapshot of the ARTiS system with those specific tasks running. Excepted a slight modification of the kernel to prevent kernel threads to run on RT CPU, all the configuration has been done from user-space. From the measurement point of view, this configuration can be considered as an “ideal” ARTiS because there is no latency overhead due to the migrations or the load-balancing.

Measurement conditions The measures have been conducted under different conditions. We have identified three unrelated parameters which affect the interrupt latencies:

- **The load.** The machine can be either idle (without any load) or highly loaded (all the programs described below are executed concurrently).
- **The kernel preemption.** When activated, this new feature of the 2.6 Linux kernel allows to re-schedule tasks even if kernel code is being executed. This configuration of the Linux kernel correspond to the so-called “preemptible Linux kernel”.
- **The CPU partitioning** can be either symmetric, the standard configuration, or asymmetric, as defined in the “ideal” ARTiS configuration.

The eight configurations that can be obtained from the combination of those three parameters were measured during short runs (10 minutes). From the results, four combinations were selected for their particular interest. They can be built by incrementally switching the parameters: A vanilla kernel without and with load, a loaded kernel with preemption activated and a loaded “ideal” ARTiS configuration. We restrained the set because idle configurations would not show any particular long latency and an asymmetric configuration without kernel preemption would not make sense.

In the experiments, the system load consisted of busying the processors by user computation and triggering a lot of different interruptions in order to maximally activate the inter-locking and the preemption mechanisms. To achieve this goal, four types of program corresponding to four loading methods were used:

- **Computing load:** A task that executes an endless loop without any system call is pinned on each processor, simulating a computational task.
- **Input/output load:** The `iodisk` program reads and writes continuously on the disk.
- **Network load:** The `ionet` program floods the network interface by doing ICMP echo/reply.
- **Locking load:** The `ioctl` program calls the `ioctl()` function that embeds a *big kernel lock*.

Observed latencies The tables 1 and 2 summarize the measures for the different tested configurations. Three values are associated to each kind of latency type (kernel and user). “Maximum” is the highest latency noticed along the 8 hours. The two other columns display the maximum latency of the 99.999% (respectively 99.999999%) best measures. For this experiment, this is equivalent to not counting respectively the 3000 (resp. 3) worse case latencies.

Although the study of an idle configuration does not bring very much informations by itself, it gives some comparison points when confronted to the results of the loaded systems. The kernel latencies are nearly unaffected by the load. However, the user latencies are several orders bigger. This is the typical problem with Linux simply because it was not designed with real-time constraints in mind.

The kernel preemption does not change the latencies at the kernel level. This was expected as the modifications focus only in scheduling faster user tasks, nothing is changed to react faster in the kernel side. However, considering user-space latencies, a significant improvement can be noticed in the number of observed high latencies: 99.999% of the latencies are under $457\mu\text{s}$ instead of 2.829ms. Unfortunately, the maximum value of these user-space latencies are very similar, in the order of 40ms. This enhancement permits soft real-time with better results than the standard kernel but in no way it allows hard real-time for which even one latency over the threshold is unacceptable.

Table 1: Kernel latencies of the different configurations.

Configurations		Kernel		
		99.999%	99.999999%	Maximum
standard Linux	idle	78 μ s	94 μ s	94 μ s
standard Linux	loaded	77 μ s	98 μ s	101 μ s
Linux with kernel preemption	loaded	76 μ s	98 μ s	101 μ s
“ideal” ARTiS	loaded	3 μ s	8 μ s	9 μ s

Table 2: User latencies of the different configurations.

Configurations		User		
		99.999%	99.999999%	Maximum
standard Linux	idle	82 μ s	174 μ s	220 μ s
standard Linux	loaded	2.829ms	41ms	42ms
Linux with kernel preemption	loaded	457 μ s	29ms	47ms
“ideal” ARTiS	loaded	8 μ s	27 μ s	28 μ s

Eventually, in the ARTiS simulated environment both kind of latencies are very significantly lowered. It can be surprising that the kernel latencies are even better than with the idle configuration. This is due to the fact all the interrupts are redirected to another processor and much less use of code is disabling the interrupts. Concerning the user latencies, the improvement is even better (dropping from a maximum around 40ms to about 30 μ s). With the limit we have fixed to 300 μ s, the system can be considered as a hard real-time one, insuring real-time applications very low interrupt response.

The standard Linux kernel, with more than 3000 response times longer than 2ms can not be considered for real-time. The gain from the kernel preemption may permit soft real-time. However, only the simulation of ARTiS has small variance between latencies and can insure latencies always under 300 μ s. It is the only configuration on which a hard real-time system can be based.

5 ARTiS Current Implementation

A basic ARTiS API has been defined. It allows to deploy applications on the current implementation of the ARTiS model, defined as a modification of the 2.6 Linux kernel.

A user defines its ARTiS application with a configuration of the CPUs, an identification of the real-time tasks and their processor affinity via a basic `/proc` interface and some functions (`sched_setscheduler()`...).

The current implementation mainly consists in the migration mechanism that ensures only preemptible code is executed on a RT CPU. This migration relies on a task FIFO implemented with lock-free access [19]: One writer on the RT CPU and one reader on the NRT CPU.

Despite that this first implementation is not yet complete, especially in regards with the load-balancing algorithms and the interprocess communication mechanisms, the first performance elements show good results.

The point is that the ARTiS implementation only adds limited non-preemptible code and that this code is independant from the other CPUs. For instance, it does not share a lock with another CPU, thus not introducing any unbounded latency. We have measured the maximum latency penalty of the ARTiS implementation over the “ideal” ARTiS configuration in the order of 10 μ s. Consequently, this qualifies the ARTiS system as a hard real-time operating system.

Although all the ARTiS mechanisms are not yet implemented, the system is already usable. It effectively guarantees the possible preemption on a RT CPU, excepted when this CPU runs the

Linux load-balancing. The main evolution of this implementation concerns the load-balancing but we are also investigating the kernel threads compartment and their influence on the latencies. Obviously some of them may migrate or be bound to a NRT CPU (such are `kjournald` that deals with journalized file systems). Other may have to be replaced, as it is the case with the per-CPU load-balancing thread.

6 Conclusion

We have identified applications that may benefit from an Open Source real-time systems that is able to exploit the full power of multiprocessors and that let real-time tasks cohabit with other and general purpose tasks. We have proposed ARTiS, a system based on a partition of the multiprocessor CPUs, between RT processors where tasks are protected from jitter on the expected latencies and NRT processors where all the code that may lead to a jitter is executed. This partition does not exclude a load-balancing of the tasks on the whole machine, it only implies that some tasks are automatically migrated when they are on the way of being non-preemptible.

The ARTiS model has been evaluated on a 4-way IA-64 and a maximum user latency as low as $30\mu s$ can be guaranteed (against latencies in the 40ms range for the standard 2.6 Linux kernel, an improvement factor of 1000).

ARTiS is also currently implemented as a modification of the Linux kernel. Despite not being yet completed, the implementation is already usable. The ARTiS patches against the Linux 2.6 kernel are available for Intel i386 and IA-64 architectures from the ARTiS web page [12].

References

- [1] Gregory E. Allen and Brian L. Evans. Real-time sonar beamforming on workstations using process networks and POSIX threads. *IEEE Transactions on Signal Processing*, pages 921–926, March 2000.
- [2] Gregory Eugene Allen. Real-time sonar beamforming on a symmetric multiprocessing UNIX workstation using process networks and POSIX pthreads. M.Sc. Thesis, The University of Texas at Austin, Austin, TX, August 1998.
- [3] Stephen Brosky. Symmetric multiprocessing and real-time in PowerMAX OS. White paper, Concurrent Computer Corporation, Fort Lauderdale, FL, 2002.
- [4] Steve Brosky and Steve Rotolo. Shielded processors: Guaranteeing sub-millisecond response in standard Linux. In *Workshop on Parallel and Distributed Real-Time Systems, WP-DRTS'03*, Nice, France, April 2003.
- [5] Ray Bryant, Bill Hartner, Qi He, and Ganesh Venkitachalam. SMP scalability comparisons of Linux kernels 2.2.14 and 2.3.99. In *4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.
- [6] Paul Chen. Implementing basic memory protection in VxWorks: A best practices guide. White paper, Wind River, Alameda, CA, 2003.
- [7] Pierre Cloutier, Paolo Montegazza, Steve Papacharalambous, Ian Soanes, Stuart Hughes, and Karim Yaghmour. DIAPM-RTAI position paper. In *Second Real Time Linux Workshop*, Orlando, FL, November 2000.
- [8] Cort Dougan and Matt Sherer. RTLinux POSIX API for IO on real-time FIFOs and shared memory. White paper, Finite State Machine Labs, Inc., 2003.
- [9] Finite State Machine Labs, Inc. RealTime Linux (RTLinux). <http://www.fsmlabs.com/>.

- [10] Bill Gallmeister. *POSIX.4, Programming for the Real World*. O'Reilly & Associates, 1994.
- [11] Bernhard Kuhn. Interrupt prioritisation in the Linux kernel, January 2004. http://home.t-online.de/home/Bernhard_Kuhn/rtirq/20040304/rtirq.html.
- [12] Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille. ARTiS home page. <http://www.lifl.fr/west/artis/>.
- [13] Momtchil Momtchev and Philippe Marquet. An asymmetric real-time scheduling for Linux. In *Tenth International Workshop on Parallel and Distributed Real-Time Systems*, Fort Lauderdale, FL, April 2002.
- [14] Kevin Morgan. Linux for real-time systems: Strategies and solutions. White paper, MontaVista Software, Inc., 2001.
- [15] Kevin Morgan. Preemptible Linux: A reality check. White paper, MontaVista Software, Inc., 2001.
- [16] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel : Design and Implementation*. Prentice-Hall, 2002.
- [17] Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. Load-balancing for a real-time system based on asymmetric multi-processing. Research Report 2004-06, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, April 2004.
- [18] Silicon Graphics, Inc. REACT: Real-time in IRIX. Technical report, Silicon Graphics, Inc., Mountain View, CA, 1997.
- [19] John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, October 1994.
- [20] Clark Williams. Linux scheduler latency. *LinuxDevices.com*, March 2002. <http://www.linuxdevices.com/articles/AT8906594941.html>.
- [21] Victor Yodaiken. The RTLinux manifesto. In *Proc. of the 5th Linux Expo*, Raleigh, NC, March 1999.
- [22] Victor Yodaiken. RTLinux beyond version 3. In *Third Real-Time Linux Workshop*, Milano, Italy, November 2001.