

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

Éric PIEL, Philippe MARQUET, Julien SOULA, and Jean-Luc DEKEYSER

{Eric.Piel,Philippe.Marquet,Julien.Soula,Jean-Luc.Dekeyser}@lifl.fr

Laboratoire d'informatique fondamentale de Lille

Université des sciences et technologies de Lille

France

Abstract. The ARTiS system, a real-time extension of the GNU/Linux scheduler dedicated to SMP (Symmetric Multi-Processors) systems is proposed. ARTiS exploits the SMP architecture to guarantee the 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 from the full power of the SMP systems: the real-time reservation, while guaranteed, is not exclusive and does not imply a waste of resources.

ARTiS have been implemented as a modification of the Linux scheduler. This paper details the evaluation of the performance we conduct on this implementation. The level of observed latency shows significant improvements when compared to the standard Linux scheduler.

1 Real-Time Scheduling on Linux SMP

Nowadays, 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. Many of those same real-time applications require large amount of computational power. A well known and effective solution to face this requirement is the usage of SMP (Symmetric Multi-Processors).

Furthermore, to ensure the durability of application developments, one may prefer to target an operating system that conforms to standards. Despite the definition of a standard POSIX interface for real-time applications [3], each vendor of real-time operating system comes with a dedicated API. From our point of view, this segmented market results from the lack of a major player in the real-time community. To face this situation, we believe in the definition of an Open Source operating system that may federate the real-time community. An real-time extension of the well established GNU/Linux operating system is an attractive proposition. Additionally, it will allow the cohabitation of real-time and general purpose tasks in the system.

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

At a first glance, real-time properties relies on the proposition of a dedicated scheduling policy. Three kinds of modification to the Linux scheduler have been proposed. One consists in running the real-time tasks in a special designed kernel running in parallel, this is what does RTAI [2]. The drawback is that the programming model and configuration methods are different from the usual one: Linux tasks are not real-time tasks and real-time activities can not benefit of the Linux services.

The second way taken is to directly modify the Linux scheduler to minimize the path from a blocking context to the re-scheduling of a real-time task. This is, for instance, the base of the work that Ingo Molnar currently carries on. The patch, called “preempt-rt”, focuses on hard real-time latencies (which is new, as all the patches before only focused on soft real-time constraints). The objective is to allow every part of the kernel to be preempted, including critical sections and interrupt handlers. The drawback is the degradation of performance for some system calls as well as the high technical difficulty to write and verify those modifications.

The third approach relies on the shielded processors or Asymmetric Multi-Processing principle (AMP). On a multi-processor machine, the processors are specialized to real-time or not. Concurrent Computer Corporation RedHawk Linux variant [1] and SGI REACT IRIX variant [8] follow this principle. However, since only RT tasks are allowed to run on shielded CPUs, if those tasks are not consuming all the available power then there is free CPU time which is lost. The ARTiS scheduler extends this second approach by also allowing normal tasks to be executed on those processors as long as they are not endangering the real-time properties.

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 $300\mu\text{s}$. This limit was obtained after a study by the industrial partners concerning their requirements.

To take advantage of an 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 look more like a general purpose operating system (GPOS) than a dedicated real-time operating system (RTOS). An 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. The Linux kernel is able to efficiently manage SMP platforms, but it is agreed that the Linux kernel has not

been designed as an RTOS. Technically, only soft real-time tasks are supported, via the FIFO and round-robin scheduling policies.

The ARTiS solution keeps the interests of both GPOS's and RTOS's by establishing from the SMP platform an **A**symmetric **R**eal-**T**ime **S**cheduler in Linux. We want to keep the full Linux facilities for each process as well as 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.

Partition of the Processors and Processes Processors are partitioned into two sets, an NRT CPU set (Non-Real-Time) and an 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 are defined. These are standard RT Linux processes, they just differ in their mapping:

- Each RT CPU has one or several bound RT Linux tasks, called **RT0** (a real-time task of highest priority). Each of these tasks has the guarantee 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 RT0 tasks. The RT0 tasks are the hard real-time tasks of ARTiS. Execution of more than one RT0 task on one RT CPU is possible but in this case it is up to the developer to verify the feasibility of such a scheduling.
- Each RT CPU can run other RT Linux tasks but **only** in a preemptible state. Depending on their priority, these tasks are called RT1, RT2... or RT99. To generalize, we call them **RT1+**. They can use CPU resources efficiently if RT0 tasks do not consume all the CPU time. To keep a low latency for the RT0 tasks, the RT1+ tasks are automatically migrated to an NRT CPU by the ARTiS scheduler when they are about to become 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 guarantees, 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 can coexist with real-time tasks and are eligible for selection by the scheduler as long as the real-time tasks do not require the CPU. As for the RT1+, the Linux tasks will automatically migrate away from an RT CPU if they try to enter into a non-preemptible code section on such a CPU.
- The NRT CPUs mainly run Linux tasks. They also run RT1+ tasks which are in a non-preemptible state. To insure the load-balancing of the system, all these tasks can migrate to an RT CPU but only in a preemptible state. When an RT1+ task runs on an NRT CPU, it keeps its high priority above the Linux tasks.

ARTiS then 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. Note that these tasks are still user-space Linux tasks. RT1+ tasks are soft real-time tasks but they are able to take advantage of the SMP architecture, particularly for intensive computing. Eventually, Linux tasks can run without intrusion on the RT CPUs. Then they can use the full resources of the SMP machines. This architecture is adapted to large applications made of several components requiring different levels of real-time guarantees and of CPU power.

Migration Mechanism A particular migration mechanism has been defined. It aims at insuring the low latency of the RT0 tasks. All the RT1+ and Linux tasks running on an RT CPU are automatically migrated toward an NRT CPU when they try to disable the preemption. One of the main requirement is a mechanism without any inter-CPU locks. Such locks are extremely dangerous for the real-time properties if an RT CPU have to wait after an NRT CPU. To effectively migrate the tasks, an NRT CPU and an RT CPU have to communicate via queues. We have implemented a lock-free FIFO with one reader and one writer to avoid any active wait of the ARTiS scheduler based on the algorithm proposed by Valois [9].

Load-Balancing Policy An efficient load-balancing policy allows the full power of the SMP machine to be exploited. Usually a load-balancing mechanism aims to move the running tasks across CPUs in order to insure that no CPU is idle while tasks are waiting to be scheduled. Our case is more complicated because of the introduction of asymmetry and the heavy use of real-time tasks. To minimize the latency on RT CPUs and to provide the best performances for the global system, particular asymmetric load-balancing algorithms have been defined [7].

3 ARTiS Current Implementation

A basic ARTiS API has been defined. It allows the deployment of 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 by configuring the CPUs, identifying the real-time tasks and their processor affinity via a basic `/proc` interface and some system calls (`sched_setscheduler()`...).

The current implementation[5] first consists of a migration mechanism that ensures only preemptible code is executed on an RT CPU. This migration relies on a task FIFO implemented with lock-free access [9]: one writer on the RT CPU and one reader on the NRT CPU.

The load-balancer implementation was carried out by improving or specializing several parts of the original Linux one. The main modification was to change from a “pull” policy to a “push” policy: it is the over-loaded CPUs which send tasks to the under-loaded ones. Although it slightly decreases performances because idle CPUs might spend more time idle, this permitted the removal of inter-CPU locks. The load estimation of a processor has also been enhanced in

order to correctly estimate a load of a real-time task (which will never share CPU power with other tasks). This enhancement permit better equity among Linux tasks. Additionally, the designation criteria has been modified to favor the presence of RT1+ (resp. Linux) tasks on RT CPUs (resp. NRT CPUs) because that is where latencies are the smallest. Similarly, tasks which are unlikely to block the interrupts soon (according to statistics about their previous behavior) will be preferred for going to an RT CPU.

A specific tool was designed to test the load-balancer correctness. It allows to run a specific set of tasks characterized by properties (CPU usage, scheduler priority, processor affinity...) to be launched in a very deterministic way. Some statistics about the run are provided but the interpretation of the results is not straightforward. Additional studies need to be done on the load-balancer implementation.

4 Performance Evaluation

While implementing the ARTiS kernel, some experiments were conducted in order to evaluate the potential benefits of the approach in terms of interrupt latency. We distinguished two types of latency, one associated with the kernel and the other one associated with user tasks.

Measurement Method The experiment consisted of 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, the current time is saved and the next measurement starts. For one interrupt there are four associated times, corresponding to different locations in the executed code (figure 1):

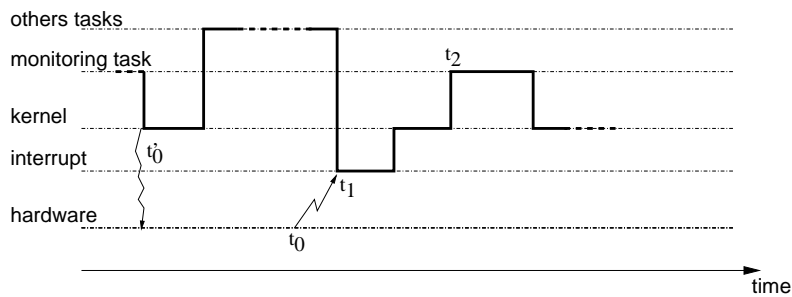


Fig. 1. Chronogram of the tasks involved in the measurement code.

- 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 entrance in the user-space RT task.

We conducted the experiments on a 4-way Itanium II 1.3GHz machine. It ran on a instrumented Linux kernel version 2.6.11. The interrupt was generated with a cycle accurate precision by the PMU (a debugging unit available in each processor [6]).

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 was run for 8 hours long, this is equivalent to approximately 300 million measures. Given such duration, the results are reproducible.

Interrupt Latency Types From the three measurement locations, two values of interest can be calculated:

- The **kernel latency**, $t_1 - t_0$, is the elapsed time between the interrupt generation and the entrance into the interrupt handler function. This is the latency that a driver would have if it was written as a kernel module.
- The **user latency**, $t_2 - t_0$, 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 of a real-time application entirely written in user-space. In order to have the lowest latency, the application was notified via 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, user latency is the most important latency to study and analyze.

Measurement Conditions The measurements were conducted under four configurations. Those configurations were selected for their relevance toward latency. First of all, the standard (vanilla) kernel was measured without and with load. Then, a similar kernel but with the preemption activated was measured. When activated, this new feature of the 2.6 Linux kernel allows tasks to be rescheduled even if kernel code is being executed. Finally, the current ARTiS implementation was measured. Only the first kernel is also presented when idle because the results with the other kernels are extremely similar.

In the experiments, the system load consisted of busying the processors by user computation and triggering a number of different interruptions in order to maximize the activation of the inter-locking and the preemption mechanisms. Five types of program corresponding to five 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 executing ICMP echo/reply.
- **Locking load:** The `ioctl` program calls the `ioctl()` function that embeds a *big kernel lock*.
- **Cache miss load:** The `cachemiss` program generates a high rate of cache misses on each processors. This adds latencies because a cachemiss implies a full cache line is read from the memory, blocking the CPU for a long time.

Observed Latencies The table 1 summarizes the measurements for the different tested configurations. Two values are associated to each latency type (kernel and user). “Maximum” corresponds to the highest latency noticed along the 8 hours. The other column displays the maximum latency of the 99.999% best measures. For this experiment, this is equivalent to not counting the 3000 worse case latencies.

Table 1. Kernel/User latencies of the different configurations.

Configurations		Kernel		User	
		99.999%	Maximum	99.999%	Maximum
standard Linux	idle	1 μ s	6 μ s	5 μ s	78 μ s
standard Linux	loaded	6 μ s	63 μ s	731 μ s	49ms
Linux with preemption	loaded	4 μ s	60 μ s	258 μ s	1155 μ s
ARTiS	loaded	8 μ s	43 μ s	18 μ s	104 μ s

The study of the idle configuration gives some comparison points when measured against the results of the loaded systems. While the kernel latencies are nearly unaffected by the load, 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. We should also mention that for all the measurement configurations the average user latency was under 4 μ s.

The kernel preemption does not change the latencies at the kernel level. This was expected as the modifications focus only on scheduling faster user tasks, nothing is changed to react faster on the kernel side. However, with regard to user-space latencies, a significant improvement can be noticed in the number of observed high latencies: 99.999% of the latencies are under 238 μ s instead of 731 μ s. This improvement is even better concerning the maximum latency, which is about forty times smaller. This enhancement permits soft real-time with better results than the standard kernel, still, in our case (latencies always under 300 μ s), this cannot be considered as a hard real-time system.

The ARTiS implementation reduces again the user latencies, with a maximum of 104 μ s. The results obtained from the measurements of the current ARTiS should not change as the additional features will only focus on performance enhancements, not on latencies. Consequently, the system can be considered as a hard real-time system, insuring real-time applications very low interrupt response.

5 Conclusion

We have proposed ARTiS, a scheduler based on a partition of the multiprocessor CPUs into 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 about to become non-preemptible.

An implementation of ARTiS was evaluated on a 4-way IA-64 and a maximum user latency as low as $104\mu\text{s}$ can be guaranteed (against latencies in the $1100\mu\text{s}$ range for the standard 2.6 Linux kernel). The implementation is now usable. The ARTiS patches for the Linux 2.6 kernel are available for Intel i386 and IA-64 architectures from the ARTiS web page [4].

A limitation of the current ARTiS scheduler is the consideration of multiple RT0 tasks on a given processor. Even if ARTiS allows multiple RT0 tasks on one RT processor, it is up to the programmer to guarantee the schedulability. We plan to add the definition of usual real-time scheduling policies such as EDF (earliest deadline first) or RM (rate monotonic). This extension requires the definition of a task model, the extension of the basic ARTiS API and the implementation of the new scheduling policies. The ARTiS API would be extended to associate properties such as periodicity and capacity to each RT0 task. A hierarchical scheduler organization would be introduced: the current highest priority task being replaced by a scheduler that would manage the RT0 tasks.

References

1. Steve Brosky and Steve Rotolo. Shielded processors: Guaranteeing sub-millisecond response in standard Linux. In *Workshop on Parallel and Distributed Real-Time Systems, WPDRTS'03*, Nice, France, April 2003.
2. 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.
3. Bill Gallmeister. *POSIX.4, Programming for the Real World*. O'Reilly & Associates, 1994.
4. Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille. ARTiS home page. <http://www.lifl.fr/west/artis/>.
5. Philippe Marquet, Éric Piel, Julien Soula, and Jean-Luc Dekeyser. Implementation of ARTiS, an asymmetric real-time extension of SMP Linux. In *Sixth Realtime Linux Workshop*, Singapore, November 2004.
6. David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel : Design and Implementation*. Prentice-Hall, 2002.
7. Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. Load-balancing for a real-time system based on asymmetric multi-processing. In *16th Euromicro Conference on Real-Time Systems, WIP session*, Catania, Italy, June 2004.
8. Silicon Graphics, Inc. REACT: Real-time in IRIX. Technical report, Silicon Graphics, Inc., Mountain View, CA, 1997.
9. 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.