Real-Time Systems for Multi-Processor Architectures[*]

Éric Piel   Philippe Marquet   Julien Soula   Jean-Luc Dekeyser

Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille
France
Firstname.Lastname@lifl.fr

Abstract

The ARTiS system is a real-time extension of the GNU/Linux scheduler dedicated to SMP (Symmetric Multi-Processors) systems. It allows to mix High Performance Computing and Real-Time. ARTiS exploits the SMP architecture to guarantee the preemption of a processor when the system has to schedule a real-time task. The implementation is available as a modification of the Linux kernel.

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 permit 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.

1 Introduction

Historically, the notions of High Performance Computing and of Real-Time have often been considered antinomic, the latter one being mostly only associated to embedded devices. Nowadays, the number of applications which can benefit from both properties at the same time is constantly increasing, in particular in the fields of multimedia and of communication. Concurrently, hardware parallelism is not anymore only a solution to bring more performance, but also to reduce energy consumption [2]. To our knowledge, there are still no well defined system that can provide both benefits at the same time. In this article, we will describe a software solution based on multi-processor computer which strives to make those both properties cohabit.

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

1.1 Multi-Processing and Real-time Approaches

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

In their review of current RTOS's, Stankovic and Rajkumar [11] describe a full taxonomy of OS's. The OS's developed from scratch are endanger species mainly because of the complexity to implement all the features now required by developers.

A more powerful approach is to have a re-usable OS from which the developer can compose by selecting components. RTEMS [8, 12] is an example to this, it is an Open-Source dedicated RTOS that supports multi-processor systems. Still, SMP support is limited, as tasks are bound to a CPU during the design phase.

Research kernels are OS's which were designed in order to present one or several new paradigms to handle a given problem. Although it might be a good approach either when the current solutions are very poor or the new paradigm would be much easier to understand or to use, it is not always efficient to force users to entirely re-consider the system organization (for instance by providing a complete new API set or by introducing new concepts).

Another approach is to add real-time extensions to a GPOS. This has the advantage of providing to the users all the facilities of the later one, including better development softwares. The following subsection will detail the different alternatives of this approach by using Linux as the original GPOS.

1.2 Real-time With Linux

The Linux kernel is able to efficiently manage SMP platforms, but it has never been designed as an RTOS. McKenney [6] has described in detail the broad number of solutions that flourished along the last few years.

A well known solution that adds real-time capabilities to the Linux kernel is the so-called *co-kernel approach*. Such a Linux extension consists in a small real-time kernel that provides the real-time services and which runs the standard Linux kernel as a nested OS by considering it as the lowest priority task. RTLinux [14] and RTAI [4] are two famous systems based on this principle. The main drawbacks are the necessity of developing real-time programs dealing with two different OS instances (with different APIs) and the limited support of SMP architectures.

A somewhat opposite solution is to improve the latencies by improving the kernel itself. An option called "kernel preemption", which is already available in the mainstream Linux kernel [7], allows a reduction of the latency targeted by multimedia applications. Currently Ingo Molnar is developing a patch called "preempt-rt" which focuses on hard real-time latencies. The objective is to allow everything 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.

Finally, an other solution relies on the shielded processors or Asymmetric Multi-Processing principle (AMP). On such a system, which is based on a multiprocessor machine, the processors are specialized to real-time or not. Concurrent Computer Corporation RedHawk Linux variant [3] follows this principle. It has the advantage of being designed from the ground with both the support of multi-processor (which can bring HPC) and the respect of real-time properties. 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 approach by also allowing normal tasks to be executed on those processors as long as they are not endangering the real-time properties.

In this article, we start by defining the principles of ARTiS, then follows a description of our ARTiS implementation in the Linux kernel and the deployment of this implementation. Finally, the last section presents experimental validation of the final implementation, focusing on three different aspects of the system, the interrupt latencies, the execution time variation and the load-balancing correctness.

2 ARTiS: Asymmetric Real-Time Scheduler

ARTiS is a real-time Linux extension that targets SMPs. Furthermore, ARTiS promotes a user-space programming model of the real-time tasks: programmers use the usual POSIX and/or Linux API to define their applications. ARTiS real-time 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 300μs. This limit was obtained after a study by the industrial partners concerning their requirements.

The ARTiS solution keeps the interests of both GPOS's and RTOS's by establishing on the SMP platform an **A**symmetric **R**eal-**Ti**me **S**cheduler in Linux. ARTiS keeps the full Linux facilities for each process as well as the SMP Linux properties but also improves the real-time behavior. 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. An example of typical architecture of a system based on ARTiS is presented in figure 1.

2.1 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 processes are defined. The processes with no particular real-time constraints are called, in our implementation, *Linux tasks*. The processes with real-time constraints are called *RT tasks*. Precisely, depending on their priority, they are called RT0, RT1... or RT99, from the highest priority to the lowest one. Due to technical reasons which we will expose just after, this second type of processes is further divided in

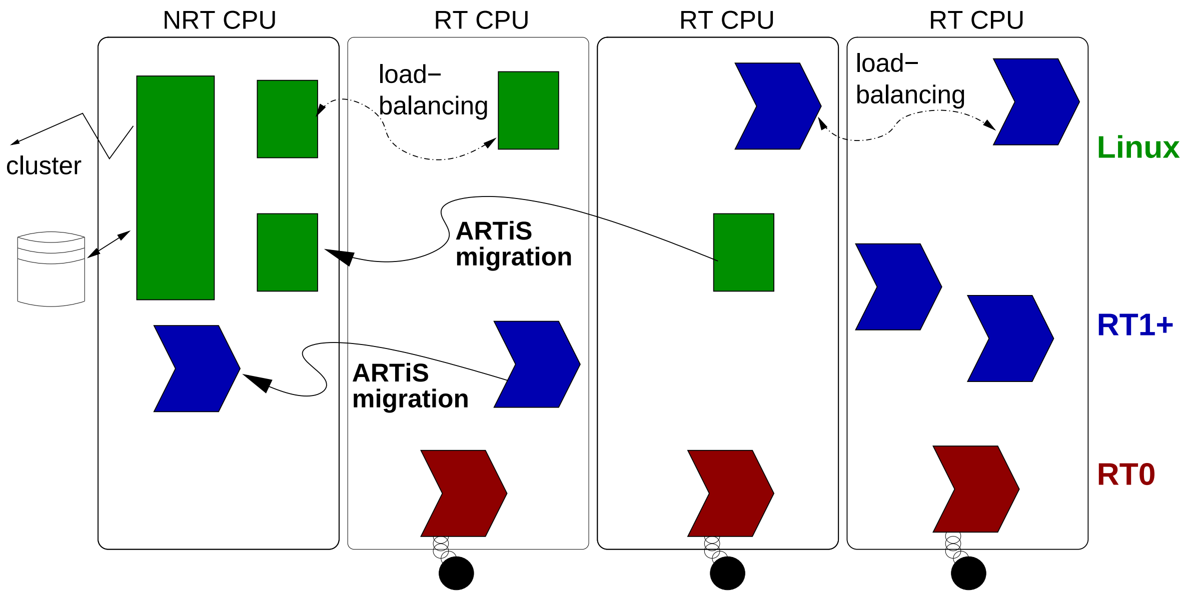

Figure 1. Example of a typical usage of a system based on ARTiS. The application is separated along different levels of real-time priorities. Tasks are moved by the ARTiS mechanisms of migration and load-balancing.

two sets. The *RT0 tasks* are distinguished from all the lower priority tasks, generalized as *RT1+ tasks*.

All those tasks are user-space tasks, they just differ in their mapping:

- Each RT CPU has one or several RT0 tasks bound to it. Each of these tasks has the guarantee that its RT CPU will stay entirely available to it. Only these 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.

- RT1+ tasks can run on any CPU. However, on a RT CPU they are **only** allowed in a preemptible state. 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. 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 Linux tasks, similarly to RT1+ tasks, can run on any CPU but, **only** in a preemptible state on the RT CPUs. 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.

RT0 tasks are implemented in order to minimize the jitter due to non-preemptible execution on the same CPU. RT1+ tasks are soft real-time tasks but they are able to take advantage of the SMP architecture, particularly for intensive computing. 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.

2.2 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. The mechanism is decomposed into two parts, one which detects the entrance of a task into a non-preemptible section of code (that is, a state into which the kernel is not be able to guarantee the scheduling of another task within a bounded time). The second part consists in moving the task from the RT CPU to an NRT CPU. More details are available in [10].

Migration Triggering Entrance detection was done by inserting a check to the only two possible ways that a task disable the preemption, in the functions `preempt_disable()` and `local_irq_disable()`. The migration triggering is not systematic, several checks are also done to allow authorized cases to continue. For instance, it is allowed to disable the preemption if the task is RT0 or the idle task, or if it is requested by an interrupt handler. Moreover, one can locally disable the migration in order to protect a part of the kernel code, for instance in the `schedule()` function.

Task Migration Pathway Locks are an easy and light mechanism to use when several threads might try to access to the same data at the same time. Unfortunately, this mechanism has no way to support priority nor preemption. Therefore inter-CPU locks are unsafe because an NRT processor may block an RT processor that shares the lock. Consequently, the original task migration code in Linux was not usable due to the inter-CPU locks it uses. In ARTiS, the migration is based on a specific intermediate queue, called RT-FIFO. It is described in detail later. In our implementation, an RT-FIFO connects every processor to every other processor.

As it is not possible to migrate a task within its own context, the migration pathway begins by changing the context to the next scheduled task. Then the task is "pushed" into an RT-FIFO to an NRT CPU. By the use of an inter-processor interrupt, the target CPU will be notified, it will then read the FIFO and insert the task into its own run-queue.

Lock Free FIFO The RT-FIFO data structure introduced in ARTiS is characterized by the fact that its accesses must be lock free. The algorithm proposed by Valois [13] insures that neither the pushing nor the pulling on an RT-FIFO is blocked. It is a lock free and wait free algorithm (wait free because we restrict the use of the FIFO to only one reader and one writer) based on a linked chain: one edge is pulled while another is pushed.

The main characteristic of the Valois algorithm is that the list is never empty: there is always at least a dummy node. The usage is to allocate dummy nodes dynamically. In a real-time context, such a dynamic allocation is not affordable (due to inter-CPU locks). Our solution consists in allocating a new node each time a task structure is allocated. When a task is pulled, its node stays as a dummy and the old dummy node is re-associated to the task structure.

2.3 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 at moving the running tasks across CPUs in order to insure that no CPU is idle while tasks are waiting to be scheduled. When trying to impose fairness between the tasks, this is usually equivalent to maintaining the same load on every processor. Our case is more complicated because of the asymmetry introduced by ARTiS and the specificities of the RT tasks. The RT0 tasks will never migrate, by definition. The RT1+ tasks should migrate back to RT CPUs quicker than Linux tasks: 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 for the global system, particular asymmetric load-balancing algorithms have been defined [9].

The current Linux implementation of load-balancing is simple, compact, modifiable and proven to work well with most of the usual workloads. Therefore, we have decided to base the ARTiS load-balancer on this implementation.

Run-queue length weighting The pairing policy of Linux selects the processor that will receive the tasks by choosing the most loaded one. The load is estimated using the number of tasks ready to be run. This estimation works well as long as there are only Linux tasks being executed, because they share their CPU time. When there is a high number of real-time tasks —which is probable in a system based on ARTiS— this last assumption is not valid anymore. Because real-time tasks have an absolute priority over the other tasks, the CPU time is not shared, and a small group of tasks might take most of the CPU. Therefore, an algorithm adequately measures the load of the RT tasks was introduced. A CPU load is computed by weighting the number of tasks in its run-queue by the RT load; the more CPU time the RT tasks take the higher will be the load. This improves the fairness between Linux tasks.

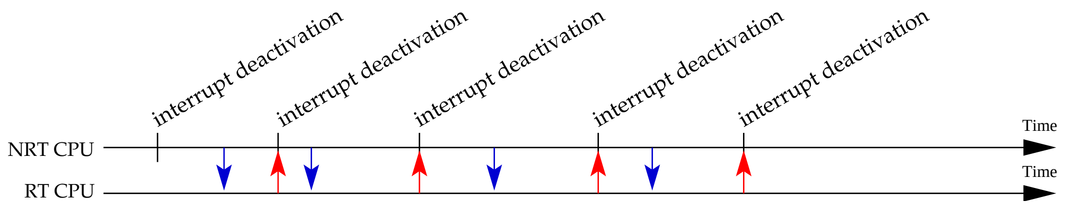

Figure 2. The so-called "ping-pong" problem. A task running on a NRT CPU will be migrated by the load-balancer to a, less loaded, RT CPU. Due to frequent interrupt deactivation, it soon goes back to a NRT CPU.

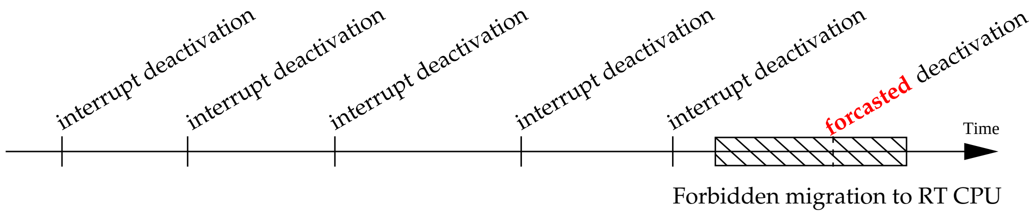

Figure 3. Period of forbidden migration (hatched rectangle). The period is deducted from the study of the previous behavior of the given task.

Inter-CPU locks withdrawal One of the direct constraints of ARTiS is the avoidance of all the locks that could be taken at the same time by RT and NRT processors. The original load-balancer does not need locks when reading the load of other CPUs but, when moving tasks from a highly loaded CPU to the current CPU, it uses inter-CPU locks on the two run-queues involved. Using the RT-FIFO (as described previously) allows to solve this problem but implies several changes in the load-balancer. The original version uses a "pull" policy (under-loaded CPUs initiate the load-balancing and pull the tasks from another CPU) but the FIFO model is much more easily implemented within a "push" policy: a processor can just select a task, put it into the FIFO and later on, another processor will asynchronously take it.

Next migration attempt estimation A special mechanism was introduced in order to provide the return of the RT1+ tasks from an NRT CPU to an RT CPU in an effective way. Typically, an RT1+ application might call several consecutive functions that disable preemption. The calls will have to be made on an NRT processor. If the load-balancer migrates it back to an RT CPU as soon as a call was finished it would lead to a ping-pong effect between the two types of processors, as represented in figure 2. Not only the execution would be slowed down for this task but the load-balance would not be achieved. Therefore, we propose that the task selection favors tasks which are more likely to stay a long time on the RT processor. By simple observations of the calls made by the application it is possible to obtain the frequency of the calls as well as the time of the last one. Hence, it is possible to estimate the next time a migration attempt will be made. As represented in figure 3, the load-balancer will not migrate the tasks for which the risk of a second migration is high.

Task/processor association The mechanisms which decide which task should be moved and which CPU is the target have been modified so they respect the asymmetry of ARTiS. Concerning the symmetric load-balancing (NRT to NRT and RT to RT), the original behavior was fine. For the load-balancing from RT to NRT, we modified the functions so that NRT tasks are moved in priority over RT1+ tasks because the latter one will have better response time on the RT CPUs. Obviously, the load-balancing from NRT to RT has to behave in the opposite way. Additionally, it will check more frequently for RT1+ tasks to move, so that their time on RT CPUs can be maximized.

2.4 System and Application Deployment

The ARTiS model is currently implemented as a modification of the 2.6 Linux kernel. The implementation has been successfully tested on IA-64 and x86 architectures. It works on SMP hardware and on multi-threaded processors — allowing computers with only one, multi-threaded, processor to benefit from the ARTiS approach to obtain real-time guarantees.

As the API of ARTiS entirely relies on the Linux API (which is very close the POSIX one), in general nearly no modifications of the applications is required. The RT ARTiS tasks are identified as Linux tasks scheduled with the FIFO scheduling policy (`SCHED_FIFO`). An RT0 task must be bound to one and only one RT CPU. The non POSIX `sched_setaffinity()` primitive is used for this. In case the user does not want, or cannot, recompile an application to fit the specific requirements for ARTiS, it is possible to set the priority of a task to RT0 using a helper program.

ARTiS is provided as a set of Linux kernel patches. They apply against the vanilla Linux kernel. A compilation of this kernel and a reboot of the machine are enough to have a working ARTiS system. Once the system is running, a setup is necessary to specify the CPU partitioning, the association between the tasks of the real-time application and the processors, as well as the affinity of interrupts towards processors.

3 Experimental Validation

The ARTiS implementation was validated by several tests: interrupt latency, execution time jitter and load-balancing effectiveness. Due to size constraints, it is not possible to describe in details those measurements but the interested reader can refer to our research report [10]. All the measures were performed on the same hardware, a 4-way Itanium II machine. The Linux kernel was either version 2.6.11 or 2.6.12 (depending on the test).

3.1 Latency Measurement

In order to evaluate the interrupt reaction latency, we did measurements of the elapsed time between the hardware generation of an interrupt (at a precisely known time) and the execution of the code concerning this interrupt. Two kinds of latencies were measured:

- The **kernel latency** is the elapsed time until the interrupt handler function is entered. This is the latency that a driver would have if it was written as a kernel module.
- The **user latency** is the elapsed time until 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.

For comparison to ARTiS, the standard Linux kernel with and without the “kernel preemption” was evaluated too. Each test was run for 8 hours long, this is equivalent to approximately 300 millions measures. All along the test the system was highly loaded by five types of program corresponding to five loading methods: computing load, input/output load, network load, kernel locks load, cache miss load.

The table 1 summarizes the measurements. From the 8 hours of measurement, the highest measured latency is reported. The kernel latencies were mostly not influenced by the configurations (about $60\mu s$), which was expectable as the modifications did not modify the interrupt handler management. On the other hand, it can be noticed that while the “kernel preemption” option does improve the user latencies (passing from the 49ms to $1155\mu s$), only the ARTiS configuration did avoid maximum latencies over our original real-time constraint of $300\mu s$ (with a maximum of $104\mu s$).

3.2 Execution Time Variation

A second evaluation consisted in studying the stability of execution duration. From a different point of view, this assesses the ability of the system to leave the CPU to a task which is currently running. In a real-time context, this corresponds to a similar need that the interrupt latency, bounding the response time.

The experiments involved measuring the execution time of a routine doing one million integer divisions, taking approximately 10ms. This routine duration was selected for being of the same order than the longest

Configurations	**Kernel**	**User**
standard Linux	$63\mu s$	49ms
Linux with preemption	$60\mu s$	$1155\mu s$
ARTiS	$43\mu s$	$104\mu s$

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

computations needed by real-time tasks that can benefit from ARTiS. Measurements were repeated one million times. For comparison with ARTiS, the standard Linux kernel was evaluated too. The task was scheduled with the highest available priority (maximum priority, `SCHED_FIFO`, equivalent to an RT0 in ARTiS). The systems were loaded with the same load as in the previous experiment.

The measurements were also executed without load. We call T_{min} the shortest time that the routine was measured among this run. It is taken as a reference for the comparison of the other times, and it is very likely the minimum time reachable by the routine on the CPU. T_{min} was 9,269μs. With the standard Linux, the maximum execution time was 20.6μs more than T_{min}, while with ARTiS we measured up to 27.1μs more. That is respectively 0.22% and 0.29% more time spent to execute the routine in the worst case.

Those fairly small variations are explained by the fact that, at this priority, the scheduler never stops the task for another one. The only slowdowns can be caused by the interrupt handlers. ARTiS brings mostly no overhead in this domain. The reason is that ARTiS modifies how fast the kernel can handle interrupts but it does not change the scheduler behavior with respect to the priorities. The overhead is probably originated by the automatic migration mechanism. Added to the measured maximum interrupt latency of 104μs, the 27.1μs variation keeps ARTiS compatible with the maximum latency targeted around 300μs. Consequently, the system can be considered as a hard real-time system, insuring real-time applications very low interrupt response time.

3.3 Load-balancing Observation

The last evaluation that we present concerns the load-balancing. Although performance benchmark tools could permit the evaluation a load-balancer, they have several limitations, mainly they do not permit broad testing of the different workloads. In addition, the code complexity of performance tests leads to non-reproducible results. A dedicated tool, called lbμ and available on the ARTiS web page [5], was designed to answer these limits.

A Load-balancer Tester lbμ focuses on running a set of tasks with as much reproducibility as possible. A set of task is called a scenario. The tasks of a scenario are fake, they only *simulate* the behavior of real tasks, and have very reproducible behavior. The same scenario can be replayed and compared using different load-balancers. A scenario is written by defining the properties of each task that will be executed. The figure 4 shows the definition of such a scenario. All the tasks are started at the same time. One scenario is not enough to evaluate a load-balancer from every angle, for this, a set of scenarios assessing all the various aspects of the policies is necessary.

As the result of a run, the user will get information about the behavior and the mapping of the tasks. During the experiments, the collected information was for each task: the execution time of the task (wall clock time), the percentage of time spent on each processor and the number of times the task was context switched (meaning scheduled and un-scheduled).

```
# a normal task
{
        cpu_mask        = 0xffff
        loop            = 10000000
}
# a RT task
{
        cpu_mask        = 0x2
        sched           = FIFO
        priority        = 99
        loop            = 110000000
        sloop           = 4000
        sleep           = 1000000
}
```

Figure 4. Extract of an lbμ scenario definition

The execution of specific scenarios permitted to validate the new or enhanced load-balancing mechanisms introduced in ARTiS and as described in section 2.3. For instance, in order to check that the new implementation improved the estimation the load generated by the real-time tasks, we used a scenario with 13 Linux tasks and 3 RT0 tasks. Each of the RT0 tasks consumed about 90% of the processor power. While with the original load-balancer, the Linux task took between 188s and 438s to complete, the enhanced one lead to smaller variations, between 377s and 485s. This shows the improved fairness brought by the modifications.

The *Next migration attempt estimation* and the *Task/processor association* mechanisms were also validated this way. Even with the presence of the "push" policy necessary to guarantee the real-time constraints, the balance was as good or better than on a ARTiS kernel without modified load-balancer.

4 Conclusion

In this document, we have proposed a system model which can provide real-time properties and high performance computing at the same time. The approach is based on a partitioning of the multi-processor computer between RT processors, where tasks are protected from jitter on the interrupt response time, 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. Additionally, we have proposed specific load-balancing policies which take into account the asymmetry in order to maintain the maximum usage of all the available computing power.

An implementation of ARTiS is available, based on Linux 2.6 and written for IA-64 and x86 architectures. The API closely follows the POSIX API and it is not even necessary to recompile Linux applications to benefit from the real-time properties. The system set up is done by specifying tasks priority and partitions for CPUs and interrupts. The validation of the current implementation of ARTiS was done by observing three main aspects of the system. A huge improvement of the interrupt latencies over the standard kernel was shown, reducing to 104μs the re-scheduling of a real-time task. The execution time variation of a real-time priority task is extremely low, as on a standard kernel. The new load-balancing policies has been proven to be correct with respect to the theory.

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. It would be interesting 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] G. E. Allen and B. L. Evans. Real-time sonar beamforming on workstations using process networks and POSIX threads. *IEEE Transactions on Signal Processing*, pages 921–926, Mar. 2000.

[2] B. Bennet. From dual-core to many-core, is the industry ready? In *PPAM 2005, Sixth international conference on parallel processing and applied mathematics*, Poznan, Poland, Sept. 2005.

[3] S. Brosky and S. Rotolo. Shielded processors: Guaranteeing sub-millisecond response in standard Linux. In *Workshop on Parallel and Distributed Real-Time Systems, WPDRTS'03*, Nice, France, Apr. 2003.

[4] 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.

[5] Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille. ARTiS home page. `http://www.lifl.fr/west/artis/`.

[6] P. E. McKenney. Attempted summary of "RT patch acceptance" thread. Linux Kernel Mailing List, July 2005. `http://lkml.org/lkml/2005/7/11/118`.

[7] K. Morgan. Preemptible Linux: A reality check. White paper, MontaVista Software, Inc., 2001.

[8] OAR Corporation. RTEMS home page. `http://www.rtems.com/`.

[9] E. Piel, P. Marquet, J. Soula, and J.-L. 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.

[10] E. Piel, P. Marquet, J. Soula, C. Osuna, and J.-L. Dekeyser. ARTiS, an asymmetric real-time scheduler for Linux on multi-processor architectures. Research Report RR-5781, INRIA, France, Dec. 2005.

[11] J. A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3):237–253, Nov. 2004.

[12] T. Straumann. Open source real-time operating systems overview. In *8th International Conference on Accelerator and Large Experimental Physics Control Systems*, San Jose, California, USA, Nov. 2001.

[13] J. D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, Oct. 1994.

[14] V. Yodaiken. The RTLinux manifesto. In *Proc. of the 5th Linux Expo*, Raleigh, NC, Mar. 1999.