ARTiS, an Asymmetric Real-Time Scheduler for Linux on Multi-Processor Architectures

Éric Piel, Philippe Marquet, Julien Soula, Christophe Osuna, Jean-Luc Dekeyser

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

ARTiS, an Asymmetric Real-Time Scheduler for Linux on Multi-Processor Architectures

Éric PIEL — Philippe MARQUET — Julien SOULA — Christophe OSUNA — Jean-Luc DEKEYSER

N° 5781

November 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5781--FR+ENG

ARTiS, an Asymmetric Real-Time Scheduler for Linux on Multi-Processor Architectures

Éric PIEL , Philippe MARQUET , Julien SOULA , Christophe OSUNA , Jean-Luc DEKEYSER

Thème COM — Systèmes communicants
Projet DaRT

Rapport de recherche n° 5781 — November 2005 —32 pages

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, especially focusing (but not restricted to) IA-64 architecture.

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.

This document describes the theoretical approach of ARTiS as well as the details of the Linux implementation. Several kind of measurements are also presented in order to validate the results.

Key-words: real-time, multi-processor architecture, Linux, scheduling, load-balancing, task migration

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

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,

ARTiS, un ordonnanceur temps-réel asymétrique pour Linux sur architectures multi-processeurs

Résumé : Le système ARTiS est une extension temps-réel de GNU/Linux dédiée aux architectures SMP (multi-processeurs symétriques). Il permet de mixer calcul à haute performance et temps-réel. ARTiS exploite la caractéristique SMP de l'architecture pour garantir la possible préemption d'un processeur quand le système doit ordonnancer une tâche temps-réel. L'implémentation est disponible sous la forme d'une modification du noyau Linux, visant en particulier (sans être une restriction) l'architecture IA-64.

Le principe d'ARTiS est d'identifier un ensemble de processeurs dédiés aux opérations temps-réel. Un mécanisme de migration automatique des activités non préemptibles assure une garantie de latence sur ces processeurs temps-réel. De plus, une stratégie spécifique d'équilibrage de charge permet à ARTiS d'exploiter la pleine puissance d'une machine SMP : les réservations temps-réel, bien que garanties, ne sont pas exclusives et n'entraînent pas de sous-utilisations des ressources.

Nous présentons ici l'approche théorique d'ARTiS ainsi que les détails de l'implémentation dans Linux. Différents types de mesures sont égalements présentés afin de valider les résultats.

Mots-clés : temps-réel, architecture multi-processeurs, Linux, ordonnancement, équilibrage de charge, migration de tâche

1 Introduction

Computer systems need to provide continuously more computational power to follow the new applications demand. Hardware parallelism is a usual solution to bring more performance. While it was historically restrained to super-computers, parallelism is becoming used in every kind of hardware due to requirements to reduce energy consumption and temperature of the processors [2].

Concurrently, as the computer is more frequently used in systems which require interactivity with the external world (as opposed to computational purpose), the need for real-time properties increases. 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).

Historically, the notions of High Performance Computing and of Real-Time have often been considered antinomic, the latter one being mostly only associated to to only embedded devices. Nowadays, this assumption cannot be hold true and there are applications which can benefit from both properties at the same time. 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.

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. This may explain why RTOS's are almost mono-processor dedicated.

In their review of current RTOS's, Stankovic and Rajkumar [20] describe a full taxonomy of OS's. The OS's developed from scratch are an endanger specie mainly because of the complexity to implement all the features now required by developers. The management of an SMP machine is part of this difficulties. In our situation, the necessary engineering work to provide both a full feature RTOS system and manage an SMP architecture would be too costly either in time or money.

Another approach is to have a re-usable OS from which the developer can select among a set of components those that will be useful for the targeted hardware. RTEMS [16, 21] 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 organisation (for instance by providing a complete new API set or by introducing new concepts).

The last approach we will describe is to add real-time extension 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 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. Technically, only soft real-time tasks are supported, via the FIFO and round-robin scheduling policies. McKenney [12] has described in detail the broad number of solutions that flourished along the last few years. In addition to the dedicated RTOS's which emulate the Linux ABI, approaches vary from the Linux kernel nested and separated from a small RTOS to real-time support directly within the kernel.

A well known solution that adds 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 which runs the standard Linux kernel as a nested OS by considering it the lowest priority task. 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 [7, 23] and RTAI [5] are two famous systems based on this principle. I-Pipe [10] is a new project which allows such co-kernel system to be easily built-up. The main drawbacks are the necessity of developing real-time programs dealing with two different OS instances (with different API) and the limited support of SMP architectures.

A somewhat opposite solution is to attempt to improve the Linux kernel latencies by improving the kernel itself. The embedded Linux vendor MontaVista has introduced a rather simple and systematic patch of the Linux kernel [13] to ensure some preemption points in the kernel, and doing so, to reduce the latencies. This patch, called "kernel preemption" and maintained by Robert Love, was adopted by the mainstream Linux kernel [14], mainly because it also implies a reduction of the latency targeted by multimedia applications. Ingo Molnar continues to work in this direction by developing a patch called "preempt-rt" which focuses on hard real-time latencies. 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 last solution that we will present relies on the shielded processors or Asymmetric Multi-Processing principle (AMP). On such a system, which is based on a multi-processor machine, the processors are specialized to real-time or not. Concurrent Computer Corporation RedHawk Linux variant [3, 4] and SGI REACT IRIX variant [19] follow 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 as well as its formal behaviour. Then follows a description of our ARTiS implementation in the Linux kernel and the deployment of this implementation. Finally, the forth and 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, the programming model ARTiS promotes a user-space programming 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 from 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 RT processes are defined. These are standard RT Linux processes, they just differ in their mapping:

- Each RT CPU has one or several RT Linux tasks bound to it, 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.

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. One of the main changes which is required from the original Linux load-balancing mechanism is the removal of 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. Based on the work by Valois [22], ARTiS implements a lock-free FIFO with one reader and one writer to avoid any active wait of the scheduler.

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 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 specificities of the ARTiS 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 [18].

2.4 Interprocess Communication Mechanisms

ARTiS includes asymmetric communication mechanisms. On SMP machines, tasks exchange data by read/write mechanisms on the shared memory. To insure 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 an RT0 task and a RT1+ task will involve the migration of the RT1+ task before this later takes on the lock, to avoid entering in a non-preemptible state on an RT CPU. Therefore, an asymmetric communication pattern should use lock free FIFO in a one-reader/one-writer context.

3 Implementation

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. The platform dependant part is very small, in the order of few code lines. Therefore a port to another architecture already supported by Linux should be fairly easy. This implementation works on SMP hardware and on multi-threaded processors – allowing computers with only one,

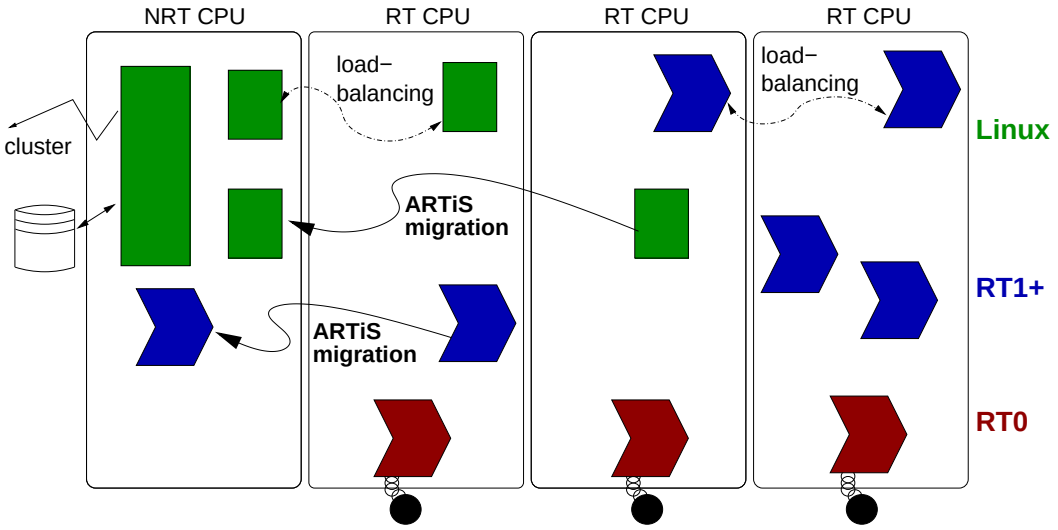


Figure 1: *Example of a typical usage of a system based on ARTiS. The application is separated along different levels of real-time priorities. Tasks are moved by the ARTiS mechanisms of migration and load-balancing.*

multi-threaded, processor to benefit from the ARTiS approach to obtain real-time guaranties.

The main modification on the Linux kernel concerns the automatic migration of a non RT0 task that is about to enter into a non preemptible section of code on an RT processor: this is a requirement from the ARTiS model. Furthermore, to benefit of the whole system, tasks must be able to move from one processor to another depending on the processor load. The usual algorithm included in Linux for this purpose has also been enhanced to deal with the real-time aspects of ARTiS.

3.1 ARTiS Migration

ARTiS migration refers to the mechanism that automatically migrates a task from an RT processor to an NRT processor because the task is about to enter a non preemptible section of code. As such, the mechanism requires that firstly the point of entry into such a section of code be identified, and secondly that the task be moved from an RT processor to an NRT processor. This latter relies on a specific implementation which guarantees that an RT processor will not wait for a lock shared with an NRT processor.

3.1.1 Migration Triggering

The ARTiS automatic migration mechanism is not systematic. Many conditions must be satisfied before allowing the migration: automatic migration only affects RT processors, neither RT0 tasks nor the idle task are concerned, and interrupt handlers are not considered.

Migration must be triggered as soon as a task enters into a state where it will not be able to stop, so that an RT0 task can be scheduled. Two paths of this kind have been identified:

- The preemption is disabled (IRQs are still handled but no re-scheduling is possible), *i.e.* a call to `preempt_disable()`.
- The interruption is disabled (IRQs are no longer received), *i.e.* a call to `local_irq_disable()`.

A task that enters into one of these two functions must migrate to an NRT processor. These two functions have been patched to include a call to the function `artis_try_to_migrate()`. This function checks the migration conditions and, if the migration is possible, effectively triggers the migration by calling `artis_request_for_migration()`.

Moreover, one can locally disable the migration in order to protect a part of the code, for instance, the `schedule()` function. To achieve this, ARTiS provides the two functions `artis_migration_disable()` and `artis_migration_enable()`. They (un)set the so-called "ARTiS flag" that is used as a complement to validate an automatic migration.

3.1.2 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 have 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, in ARTiS the RT processor must not take the lock on the local run-queue and the lock on the destination run-queue at the same time.

ARTiS takes advantage of the fact that the scheduler already takes a lock on its run-queue in order to perform migration during the scheduler execution. Therefore, the actions of dequeuing and queuing are executed by different CPUs, the link between them being achieved by an intermediate queue specific to ARTiS, called RT-FIFO. On ARTiS, an RT-FIFO connects every processor to every other processor (although the migration mechanism only uses paths from RT CPUs to NRT CPUs).

A task triggers its own migration using a call to `artis_request_for_migration()` but a task can not queue itself in an other run-queue because, in this case, it would be runnable on two CPUs at the same time. Consequently, it needs a helper task in the same way that changing its own processor affinity requires the `kmigration` kernel thread. In ARTiS, the duty of helper task is devolved to the next scheduled task.

In total, the migration process involves the interaction of three tasks: the migrating task, the next task on the same CPU and the next scheduled task on the other CPU. Each of these tasks performs a part of the migration:

- **The request part** is carried out by the task itself by executing the function `artis_request_for_migration()`. When the task has decided to migrate, it first sets a special flag to identify the migration step. It also sets its processor affinity to that of the only local processor in order to insure that it will not be moved unwillingly. It then re-enables the preemption and calls the scheduler. ARTiS guarantees that the task will release the CPU and that, the next time it is scheduled, it will be on the requested CPU. Then the flag and affinity are reset and the task resumes its normal course.
- **The completion part** is achieved by the "next task". When a "previous task" has set the ARTiS migration flag, it is dequeued in the scheduler, and, following the context switch, the new current task will execute the completion function `artis_complete_migration()`. It uses the special Linux callback `finish_task_switch()` which is always called after a task has finished being scheduled. The completion function chooses an NRT processor as a destination, enqueues the designated task in RT-FIFO and forces a re-schedule on the destination CPU via an inter-processor interruption.
- **The fetch part** is achieved on the destination processor. At every scheduling tick, the function `artis_fetch_migration()` is used to verify the RT-FIFOs for the NRT processors (potential migration designation). All the tasks present in those special FIFOs are pulled out and enqueued into the local run-queue.

3.1.3 Lock Free FIFO

The RT-FIFO data structure introduced in ARTiS is characterized by the fact that accesses to it must be lock free: RT processors should never share any lock with any NRT processor.

The algorithm proposed by Valois [22] 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:

- on initialization, a dummy node is introduced into the structure,
- the last pulled node stays on the head list as a dummy node.

The algorithm uses nodes containing the linkage and a reference to the value (the task structure, `task_struct`, in our case). These nodes are allocated and freed dynamically. In a real-time context, such a dynamic allocation is not affordable. The node can no longer be embedded in the task structure. This is because the node part of a pulled task would stay as dummy node in the data structure and consequently it would prevent the task being pushed again.

Our solution consists of an allocation of a node when the task structure is allocated. The node and the task structure are associated. When a task is pulled, its node stays as a dummy and the old dummy node is re-associated with the task structure.

3.2 ARTiS Load-Balancing

A load-balancing mechanism aims at optimising processor exploitation by the simple means of moving tasks from one processor to another. The aim can also be stated as being the minimization of the total running time for a given set of tasks. This is usually equivalent to maintaining the same load on every processor.

The characteristics of a load-balancer are explained in detail by Fonlupt in [8] and can be enumerated as follows:

- information update policy: how to renew statistics on the entire system. it can be either of type "pull" –under-loaded CPUs initiate the load-balancing and pull the tasks from another CPU– or "push" –over-loaded CPUs initiate the load-balancing in order to push some of their tasks– or a mix of both,
- trigger policy: how to decide it is time to redistribute the tasks,
- selection policy: a method for selection of imbalanced nodes,
- local designation policy: a method for selection of tasks that will be moved,
- pairing policy: a method for selection of the destination node for a given task.

3.2.1 ARTiS Specific Constraints

The Linux load-balancer works well, especially in real-life conditions. However with the addition of the ARTiS constraints, its behaviour is far from being optimal. In particular, the introduced asymmetry between processors requires a load-balancer that can handle the specific affinities between processors and tasks.

The three main new constraints are the removal of inter-CPU locks (required to guarantee real-time properties), the minimisation of time spent by RT1+ tasks on NRT processors, and the minimisation of the number of automatic migrations. An in-depth study of all the different load-balancing scenarios which highlights the constraints is available in [18].

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 load-balancer for ARTiS on this implementation.

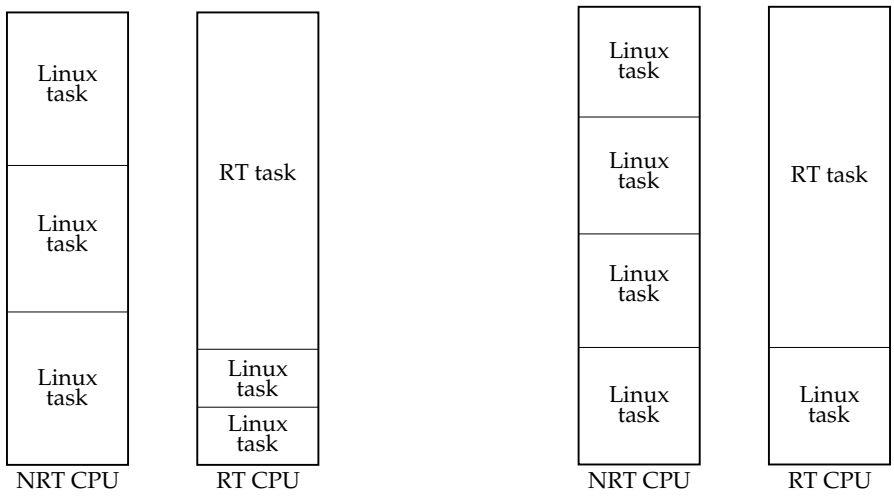
3.2.2 Run-queue length weighting

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 (the length of the run-queue). This estimation works well as long as there are only Linux tasks

being executed, this is because they share the CPU time and consequently the longer the run-queue is, the less time there is for every task.

This last assumption is false when there is a high number of real-time tasks on the computer. Because real-time tasks have an absolute priority over the other tasks, the CPU time is not shared. Therefore, the run-queue length is no longer representative of the available power. We propose improving equity between Linux tasks by adding the CPU time consumed by RT tasks as a parameter of the load estimation.

For example, on a bi-processor computer, if a real-time task consumes $3/4$ of a processor time and there are 5 Linux tasks also being executed then the current Linux implementation will put 3 tasks on each processor. This implies that some Linux tasks will have $1/3$ of the CPU time while others (with the same priority) will only have access to $1/8$ of the time, as shown on figure 2(a). By taking into account the real consumption of the RT task, equity is recovered and every Linux task is given $1/4$ of the CPU time, as shown on figure 2(b). This type of scenario is highly probable on an ARTiS system because the real-time tasks are asymmetrically distributed.



(a) *Original Mechanism* (b) *Run-queue length weighting*

Figure 2: *Improvement of the fairness between Linux tasks done using weighting of the run-queue length.*

The solution we propose is to measure the load of each processor using the formula $L \times \frac{1}{1-RT}$, where L is the run-queue length **without** the real-time tasks and RT is the ratio of time that was consumed by real-time tasks. One could note that with this algorithm, if no Linux tasks are running, the load will always be null, even when real-time tasks are using a lot of the processor power. It may lead to the load to be balanced only after some tasks were first move to the RT CPU and then back to an NRT CPU. However, it has the

benefit of avoiding the RT CPU so spend some time idle, which would be extremely bad performance-wise.

Consequently, the implementation requires the addition of statistics regarding the number of RT tasks being executed on each processor, and also the measurement of the RT ratio. At a given instant the RT ratio is either 1 (there is a real-time task) or 0 (the processor is idle or executing a Linux task). To obtain the intended value it is necessary to smooth the value over time. ARTiS uses a similar mechanism to the `CALC_LOAD()` one which weights the values so that more recent values have more importance. The last 500 samples are taken into account, which corresponds to about 0.5 second on the architecture that we used.

3.2.3 Inter-CPU locks withdrawal

One of the direct constraint of ARTiS is avoidance of all the locks that could be taken at the same time by RT and NRT processors. 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 have no way to support priority nor preemption and therefore an RT CPU could be waiting after an NRT CPU (on which latencies are not guaranteed). The original implementation 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 in section 3.1.3) allows to solve this problem but implies several changes in the load-balancer. The original version uses a "pull" trigger policy but the FIFO model is much more easily implemented within a "push" policy: a processor can just select a task, put it inside the FIFO and later on, another processor will asynchronously take it. A "pull" policy would be possible but it would be more complex and less time effective.

In order to inverse the trigger policy the main thing that is changed is the function `find_busiest_queue()` which should no longer look for the longest run-queue but for the smallest one. This new function is called `find_idlest_queue()`. All the sub-functions had to be changed similarly. Another implication of the change is that processors will not execute the load-balancer when they become idle.

3.2.4 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 endanger real-time properties. The calls will have to be made on an NRT processor. If the load-balancer migrated 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 3. Not only would execution be slowed down for this task but the load-balance would not be achieved.

Therefore, we propose the modification of the task selection method so that it can favour tasks which are more likely to stay a long time on the RT processor. By simple observation

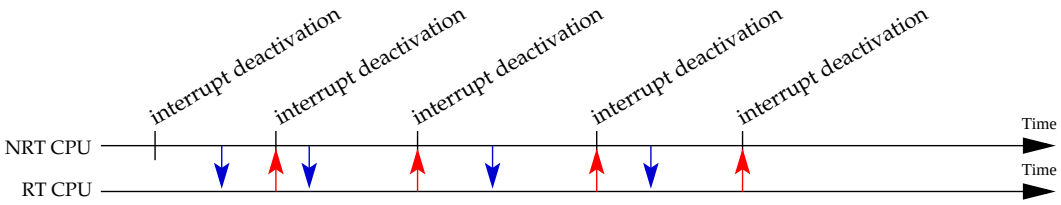


Figure 3: *The so-called "ping-pong" problem. A task running on a NRT CPU will be migrated by the load-balancer to a, less loaded, RT CPU. Due to frequent interrupt deactivation, it soon goes back to a NRT CPU.*

of the calls endangering real-time (that is to say, a migration attempt) made by an 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. The load-balancer can avoid migrating the tasks for which the risk of a second migration is high. This mechanism is represented in figure 4.

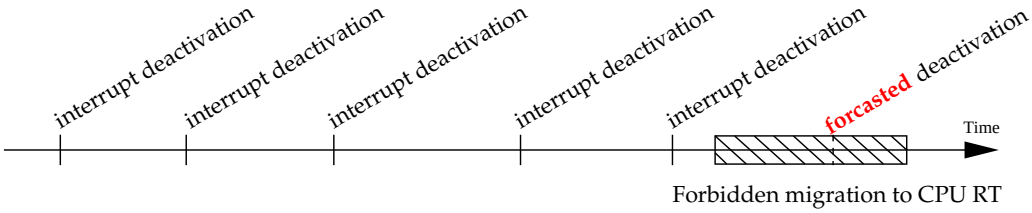


Figure 4: *Period of forbidden migration (hatched rectangle). The period is deducted from the study of the previous behavior of the given task.*

Typically, at a given time t, there are two possibilities:

- The next estimated migration is after t, if the migration is likely to happen soon then no migration should be carried out. On the contrary, if the migration is not likely to happen for some time (specified as a system wide constant), the task can be migrated back to an RT CPU.
- The next estimated migration is before t, if it should have happened recently then it is still likely to happen soon and no migration should be carried out. If the migration was forecasted considerably beforehand, the task can be migrated back to an RT CPU. This test is relative to the measured period of the task.

A detailed mathematical representation of this conditions is available in [17].

Of course the implementation of this predicting mechanism consists in slightly modifying the load-balancer code (the function `load_balance()`) but it also consists of getting

the statistics about the migration attempts. The statistics are saved inside the task structure as two numbers, one for the time weighted average period between two attempts and one for the timestamp of the last attempt. Each time the function `artis_try_to_migrate()` is called, and would trigger a migration if the current task was on an RT CPU, the statistics are updated.

3.2.5 Task/processor association

The local designation policy (the mechanism which selects which task should be moved) and the pairing policy (the mechanism which decides the new location for a task) were modified so they respect the asymmetry of ARTiS. Based on the original function `load_balance()` and all its sub-functions, `load_balance_push()` was derived. Depending on the types of the origin and destination CPUs, this function is called to move only RT tasks or all the tasks.

Concerning the symmetric load-balancings (NRT to NRT and RT to RT), very little was necessary, the policies are identical to the original ones.

For the load-balancing from RT to NRT, the function `move_tasks_push()` was modified so NRT tasks are moved before RT1+ tasks because the latter will have better response time on the RT CPUs. Obviously, the load-balancing from NRT to RT has to behave in the opposite way by favoring the move of real-time tasks (which is the normal policy). The function `move_tasks_push()` takes the parameter `only_rt` which specifies if only RT1+ tasks should be moved or all tasks should be considered.

One very important aspect of modifying the `rebalance_tick()` function is the ability to have different triggering frequencies according to the CPUs involved. In particular, the migration of RT1+ tasks from NRT to RT processors is triggered with a high frequency. The exact frequency was experimentally tuned, it is called 4 times more often than the original version so that the time the tasks spend on NRT CPUs can be minimized. It should also be noted the removal of the trigger which occurs when the processors happen to become idle, because it is not beneficial for the "push" trigger policy.

4 Real-Time Application Deployment

Despite the fact that real-time applications do not need to be recompiled to benefit from the ARTiS enhancement, the user must specify the system partitioning and the way the application will be deployed on the processors of the system. ARTiS applications are defined by a system configuration, mainly CPU orientation and interrupts affinity, and by the task configuration, mainly their priority and their processor affinity.

A basic ARTiS API has been specified to do so. It allows the deployment of applications on the current implementation of the ARTiS model, available as a modification of the 2.6 Linux kernel.

4.1 Example of Application Deployment over ARTiS

A real-time application on an SMP machine reveals its full significance when real-time constraints are combined with intensive computing. ARTiS is dedicated to this kind of application. 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.

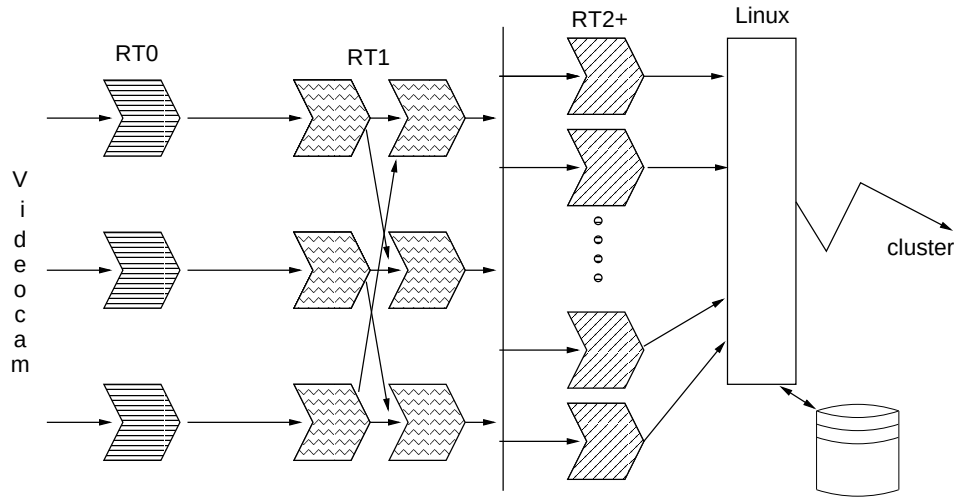


Figure 5: Example of a task architecture in an ARTiS application.

The implementation of an application on ARTiS requires the identification of a specific level of real-time for each task of the algorithm and a mapping of these tasks on the CPUs of the multi-processor machine. To illustrate this, we use real-time manufacturing quality management: for example, defect identification on a continuous production line running on a four CPUs SMP system, three CPUs for the RT set and one for the NRT set. Several tasks may be identified, their communications and mapping are illustrated in figure 5:

- A 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 carried out 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 identification has to be achieved using irregular data structures: each defect processing is specific. A dynamic number of RT2 are created. They have to communicate with RT1 and with each other. Because the number of tasks is dynamic, the load-balancing policy of ARTiS is indispensable.
- Finally, some defect fitting can be done with a local database or an external database (accessed via MPI) to produce statistics... This stage does not require RT processing, and thus can be carried out by Linux tasks. They are mainly mapped on the NRT CPUs but they can also use RT CPUs when idle.

Figure 6 shows a possible mapping of those tasks on the two sets of processors: RT and NRT.

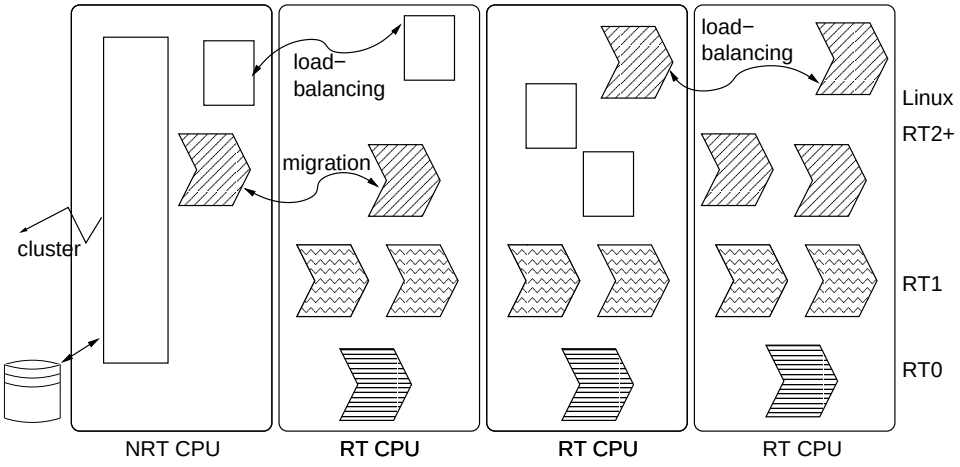


Figure 6: Mapping of the tasks on the RT and NRT processors.

4.2 Installation and System Setup

ARTiS is provided as a set of Linux kernel patch. They apply against the vanilla Linux Kernel, available at `http://www.kernel.org`. A compilation of this kernel and a reboot of the machine are enough to have a working ARTiS system. It is sometimes necessary to also deactivate all power saving options which often tend to increase interrupt latency. No other package is required nor any re-compilation of the applications.

Once the system is running, a setup is necessary to let the kernel know about the required machine partitioning. The administrator can specify which CPU is RT and which is NRT via a basic `/proc` interface. To maintain coherence with this machine partitioning, a redirection of the interrupts has to be programmed. All IRQs must be delivered exclusively to the NRT CPUs, excepted those IRQs used by the RT0 tasks, which must be delivered on

the CPU hosting the task. This can easily be automated by a script. The `/proc` interface allow also to activate and deactivate ARTiS dynamically and to retreive some statistics describing the task migrations.

4.3 RT Process Identification

The RT0 ARTiS tasks are identified as Linux tasks scheduled with the FIFO scheduling policy (`SCHED_FIFO`) and having the highest priority. The POSIX functions `sched_setscheduler()`, `sched_setparam()` and `sched_get_priority_max()` are used for this purpose. An RT0 task must be bound to an RT CPU. The non POSIX `sched_setaffinity()` primitive is used for this. The set of CPUs on which an RT0 is allowed to run on must be limited to a single CPU, and this CPU must be an RT CPU.

Due to the nature of ARTiS, the only constraint compared to a usual POSIX RT application is the requirement on the order of the calls to `sched_setscheduler()` and `sched_setaffinity()`. The priority set up must always be **before** the affinity set up. Otherwise it will fail because changing the priority requires to take a lock, forcing the task to run on an NRT CPU, which in turn force ARTiS to modify the CPU affinity which has just been set. In case the user does not want, or cannot, recompile her application to fit those specific requirements for ARTiS, it is possible to change the task to RT0 from another task. For instance, `schedtool` [9] is a program able to perform this task.

```
unsigned int rt_cpu;
struct sched_param schedp;

/* lock the address space of the process */
if (mlockall(MCL_CURRENT|MCL_FUTURE) != 0)
        perror(...);

/* set the scheduling policy */
memset(&schedp, 0, sizeof(struct sched_param));
schedp.sched_priority = sched_get_priority_max(SCHED_FIFO);

if (sched_setscheduler(0, SCHED_FIFO, &schedp) != 0)
        perror(...);

/* bound the process to the rt_cpu CPU */
if (sched_setaffinity(0, sizeof(unsigned long), 0x1UL << rt_cpu ) == -1)
        perror(...);
```

Figure 7: *RT0 identification*

Figure 7 presents an outline of the code a task may include in order to be identified as an RT0 task. ARTiS also comes with a basic interface library (available on the web page in the `libartis` package) that provides functions to register and unregister an RT0 task:

```
int artis_enter_rt0 (pid_t pid, int rt_cpu);
int artis_leave_rt0 (pid_t pid);
```

The RT1+ tasks are all the tasks associated with either the FIFO or round-robin scheduling policy (`SCHED_FIFO` or `SCHED_RR`). As with the standard POSIX definition, the priorities of these tasks define their relative priority. The ARTiS library provides the following two functions to identify these tasks:

```
int artis_enter_rt1plus(pid_t pid, int policy, int priority);
int artis_leave_rt1plus(pid_t pid);
```

The so-called Linux tasks, *i.e.* the non real-time tasks, are all tasks scheduled with the usual Linux `SCHED_OTHER` policy.

The CPU affinities of non RT0 tasks must include an NRT CPU, otherwise ARTiS will automatically assign one when entering a non-preemptible code section. In addition, the CPU affinities of RT1+ tasks should also include at least one RT CPU.

5 Experimental Validation

While implementing the ARTiS kernel, some experiments were conducted in order to evaluate the potential benefits and drawbacks of the approach. We are presenting three measurements: interrupt latency, execution time jitter and load-balancing effectiveness.

On current hardware, the performance optimisations introduce non-deterministic execution time, as McGuire and Zhou have shown in [11]. In particular, mechanisms of cache and branch prediction might produce long latencies which can easily be overlooked by experimental measurements. In the presented results we strove to keep the effects in mind. For instance, specific loads were designed to trigger cache line loads.

5.1 Latency Measurement

The first evaluation presented is the interrupt reaction latency. This is the time needed by the system to execute the routine corresponding to a particular hardware event. In a real-time context, the important point is to minimize the longest latencies in order to be able to answer in time to the hardware. We distinguished two types of latency, one associated with the kernel and the other one associated with user tasks.

Although a theoretical analysis of the worst case execution time (WCET) is better because it provides a maximum latency estimation always superior (but, unfortunately, often very superior) to the real maximum one, we did not practice such analysis due to its complexity. Difficulties of such computation have been well defined by the study of the WCET of RTEMS (reduced to one processor) in [6]. In addition to the problems the authors faced, it would have been necessary to describe the IA-64 architecture, simplify and annotate the Linux source code, and take into account the execution parallelism both at the hardware and software levels.

5.1.1 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. 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. Each interrupt is associated to four different times, corresponding to different locations in the executed code (figure 8):

- 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.

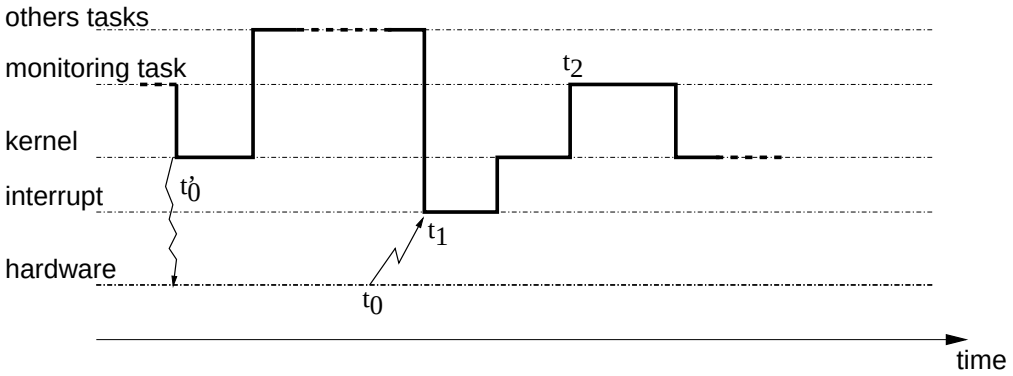


Figure 8: *Chronogram of the tasks involved in the measurement code.*

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 `itc` (a processor register counting the cycles) is the timer on which all the measurements are based and the interrupt was generated with a cycle accurate precision by the PMU (a debugging unit available in each processor [15]).

Even with a high loading 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.

5.1.2 Interrupt Latency Types

From the three measurement locations, two values of interest 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 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 real-time application entirely written in user-space. The application was waiting for the interrupt via a blocking system call (a `read()`), on our system this was the notification method which exhibited the lowest latency.

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.

5.1.3 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 the cache is cold for the interrupt handler, scheduler and the RT0 task when they are executed.

5.1.4 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.

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

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

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 4µs.

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 238µs instead of 731µs. 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 300µs), this cannot be considered as a hard real-time system.

In the ARTiS configuration, both kind of latencies are very significantly lowered, with a maximum of 104µs for user latencies. This is below the specified maximum latency of 300 µs. Consequently, the system can be considered as a hard real-time system, insuring real-time applications very low interrupt response.

5.2 Execution Time Variation

The second presented evaluation is the study of execution duration stability. It consists in running a small computational code several times and measure how long each run took. In a real-time context it is necessary to be able to compute a response to the hardware in less than a fixed amount of time. This corresponds to a similar need that the interrupt latency (as presented in the previous section).

5.2.1 Measurement Method

The experiments consisted of measuring the execution time of a routine doing one million integer divisions, taking approximately 10ms. This routine duration was selected to be of the same order than the longest computations needed by real-time tasks that can benefit from ARTiS. It worth noticing that it corresponds to approximately ten scheduling slices. Measurements were repeated one million times. Each measurement consisted in saving the time before and after the call to the routine with a precision of the cycle.

5.2.2 Measurement Conditions

All the measurements were done on the same computer than for the latency measurements, on a Linux kernel version 2.6.12. The tests were run 8 times, in different conditions. Three different aspects varied:

- **The priority**. The task can either be a task with normal priority (niceness of 0, `SCHED_OTHER`) or a high priority real-time task (maximum priority, `SCHED_FIFO`, equivalent to an RT0 in ARTiS).
- **The load**. The machine can be either idle (without any load) or highly loaded (same load than for the previous measurement).
- **The ARTiS activation**. The kernel was compiled either with or without ARTiS.

5.2.3 Observed Execution Time

The table 2 summarizes the measurements for the different tested configurations. We call T_{min} the shortest time that the routine was measured among all the configurations. It should be very close or equal to the minimum possible execution time of the routine on the hardware. For those measurements, T_{min} was 9,269µs. Every 3-hours test is summarized by two numbers. The first one is the difference of time between T_{min} and the maximum execution time recorded. The second number, in parenthesis, represents the percentage of additional time taken by the longest execution compared to T_{min}.

The results show that the load, as could have been expected, is highly affecting the execution duration, in particular when the task is scheduled with normal priority. Running the task with RT0 priority does avoid mostly all the execution time jitter (worst case is

kernel	load	Normal priority	RT0 priority
standard Linux	idle	0.5926ms (6.39%)	8.30µs (0.08%)
standard Linux	loaded	516.1ms (5567.49%)	20.61µs (0.22%)
ARTiS	idle	5.675ms (61.22%)	27.07µs (0.29%)
ARTiS	loaded	659.7ms (7116.43%)	21.61µs (0.23%)

Table 2: *Execution time difference between T_{min} and configuration maximums.*

only 0.22% slower than the fastest case). That is because, at this priority, the scheduler never stops the task for another one. The only slowdowns can be caused by the interrupt handlers.

Concerning ARTiS, the results demonstrate it mostly does not influence the execution time jitter. For RT0 tasks, an increase of up to 0.07 points was noticed (on an idle system). The reason is that ARTiS modifies how fast the kernel can handle interrupts but it does not change the scheduler behaviour with respect to the priorities. The overhead due is probably originated by the automatic migration mechanism. Even with this overhead, the execution time jitter for RT0 tasks is very low, 27.07µs at maximum. Added to the measured maximum interrupt latency of 104µs, it keeps ARTiS compatible with the maximum latency targeted around 300µs.

5.3 Load-balancing Observation

The last evaluation that we present concerns the load-balancing. To our knowledge, there is no program available to specifically estimate a load-balancing. Of course, a better load-balancing should give better performance. Therefore simply a performance evaluation, for which there already exist a plethora of measurement tools, would provide a basic assessment. Comparison can be done on the (wall clock) time it takes to finish the execution of all the tests. However, this approach has several limitations. In particular, it might be slowed down by a different component than the scheduler or load-balancer, it is also limited in the coverage of the task set organisation. Additionally, the code complexity of performance tests leads to non-reproducible results. Finally, when implementing new load-balancing (or scheduling) policies, fine tuning the algorithm often requires to observe the behaviour with very precise configuration of the task, which can be extremely hard to reproduce just with performance tools. During the implementation of ARTiS, it was also necessary to confirm that a modification of the load-balancer was really effecting the behaviour as it was designed to. A dedicated tool was written to answer these limits and needs.

5.3.1 A Load-balancer Tester

Scenarios lbµ is a small tool that focuses on running a set of tasks with as much reproducibility as possible. A set of task is called a scenario. It is written with one specific

behaviour of the load-balancer to test in mind. The tasks of a scenario are fake, they only *simulate* the behaviour of real tasks. This allow them to have very constant properties and behaviour. Therefore, the same scenario can be replayed as many time as wished using different load-balancers. A scenario is written by defining the properties of each task that will be executed. Those properties are the scheduling policy and priority, the CPU affinity, the computational power (number of loops executed), the frequency and duration of wait, and the frequency of a system call blocking the preemption (this is meaningful only when ARTiS is used). The figure 9 shows the definition of such a scenario. All the tasks are started in the same time, at the beginning of the measurements. One scenario is not enough to evaluate a load-balancer in its globality, for this, a set of scenarios assessing all the various aspects of the policies is necessary.

```
# 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 9: *Extract of a lbμ scenario definition*

Observable properties As the result of a run, the user will get information about the behaviour and the mapping of the tasks. The available information is mostly restrained only by the information given by the Linux kernel. During the experiments, the information that was collected by lbμ was for each task:

- execution time of the task (wall clock time),
- percentage of time spent on each processor,
- number of times the task was context switched (differentiating when the task did it willingly and when it was preempted).

The amount of time spent by each CPU to be idle is also collected as soon as the first task finish. Additionally, to allow a faster analysis, statistics grouping the results by tasks with identical properties are also provided.

5.3.2 Measurement Conditions

The experiments were conducted on the same computer than in the two previous one. Several scenarios were written, each one orientated toward a different and specific configuration of task which the new load-balancer should handle better or not worse (depending on the tested policy). Three configurations of the system were tested, one is running the normal Linux kernel (version 2.6.12), another one uses ARTiS but with the original load-balancer, the last one uses the full implementation of ARTiS. The second configuration, which exists only for comparison purpose, will sometimes migrate tasks from an NRT CPU to an RT CPU, leading to inter-CPU locks. Therefore, although it is based on ARTiS the real-time constraints can not be guaranteed on this configuration.

5.3.3 Experiments Observation

The experiments will be presented one by one, each of them corresponding to one policy modified during the implementation of ARTiS.

Run-queue length weighting In order to check that the new implementation improve the estimation the load generated by the real-time tasks, a scenario with 13 Linux tasks and 3 RT0 tasks. Each of the RT0 task is bound to a different RT CPU and consumes about 90% of the processor power. The Linux tasks do only computational processing. ARTiS was configured to have three RT CPUs and one NRT CPU. In this scenario, the theoretical best load-balancing, which would give equity and performance, would consist in placing 10 Linux tasks on the NRT CPU and one Linux task on each RT CPU (then all would have 10% of the CPU time available). We are not presenting the time of the RT tasks because the scenario was tuned so that they were running even after all the Linux tasks finished; in addition, it is identical on every configuration (617 seconds).

	Standard Linux	**ARTiS normal LB**	**ARTiS enhanced LB**
CPU idle time (µs)	0 0 0 0	0 0 0 0	0 0.02 0.06 0.09
CPU repartition of Linux tasks (%)	70 11 8.5 10.5	68.5 11 10.5 10	68 11 11 10
Linux tasks completion time (s)	190 / 315 / 447	188 / 312 / 438	377 / 381 / 485

Table 3: *Execution results of 13 Linux tasks and 3 RT0 tasks with CPU repartition as **NRT RT RT RT**. Completion time is express as minimum / average / maximum.*

The table 3 summarizes the results. On the standard Linux and on ARTiS with a normal load-balancing the inequity between the tasks can be noticed by the completion time of Linux tasks. Some tasks were completed more than twice faster than other tasks. With the run-queue length weighting mechanism all the task were given mostly the same amount of computational power, therefore they all finish in the same time, which is close to the

time it took for the longest task in the previous configurations. This effect implies that, counter-intuitively, a *high* average of completion time is significative of good equity. Only the maximum completion time is an indicator for computational power.

Equity was confirmed by observation during the experiments. The first two configurations were executing 4 tasks on each processor, while the configuration with the modified load-balancing had 10 Linux tasks on the NRT processor and 2 (the bounded RT0 task and a Linux task).

With equity, the last task took a bit more time to complete than in the first two configurations (485s instead of about 445s). This is because on the RT CPUs with only one Linux task, when the Linux task completes there is a small time gap before another task arrives. This does not happen when three Linux tasks are running on the RT CPU, there is always a task to consume the available CPU time. This is a small loss in CPU power which can be considered the "cost" of equity.

Next migration attempt estimation This load-balancer enhancement should favor tasks often endangering real-time to stay on the NRT-CPUs, avoiding a ping-pong problem between NRT and RT CPUs. The scenario to validate it is based on having two sets of 8 Linux tasks. One set takes a lock very often (approximately every 200µs), while the other one take a lock 1000 times less often. The former set is called OM (Ofen Migrating) while the latter one is called IM (Infrequently Migraring). The CPUs are partitioned as two NRT CPUs and two RT CPUs. An theoretical ideal load-balancer would schedule the OM set on the two NRT processors while the IM set would be scheduled on the RT processors (migrating to an NRT processor only very briefly when taking a lock).

	Standard Linux	**ARTiS normal LB**	**ARTiS enhanced LB**
CPU idle time (µs)	0 0 0 0	0 0 150 195	0 0 130 142
CPU repartition of IM tasks (%)	25 37.5 37.5 0	10 53 18 19	15 50 20 15
CPU repartition of OM tasks (%)	25 12.5 12.5 50	63 21 15 1	68 31 1 0
IM tasks completion time (s)	191 / 191 / 192	191 / 239 / 277	144 / 247 / 316
OM tasks completion time (s)	191 / 191 / 192	191 / 221 / 278	220 / 245 / 264
IM tasks involuntary ctxt switchs	575	400	310
OM tasks involuntary ctxt switchs	535	7200	845

Table 4: *Execution results of 8 tasks often taking a lock and 8 tasks infrequently taking a lock with CPU repartition as* ***NRT NRT RT RT****. Completion time is express as minimum / average / maximum.*

Table 4 summarizes the results. On a standard kernel, the two types of tasks are considered similar (because taking a lock does not influence the scheduling), consequently the tasks are scheduled similarly. All the tasks completed in the same amount of time. The CPU repartition difference is simply due to some randomness when the load-balancer selected the tasks, in total there were 4 tasks running on each processor.

With ARTiS, on both configurations, tasks cannot run all the time on the RT CPUs. Those CPUs were sometime idle, which implies computational power loss. This is due to the automatic migration mechanism necessary to guarantee real-time properties. While with the normal load-balancing the OM tasks spent about 16% of their time on the RT CPUs, with the enhanced load-balancing they were running only 1% of the time. This confirms the estimation of the next migration attempt does work correctly. On both configurations the IM tasks spent about 35% of their time on the RT processors. On the last configuration, IM tasks could have spent more time on RT CPUs, this inefficiency is the result of the reduced performance of the "push" policy, necessary to avoid inter-CPU locks, compared to the "pull" policy.

An effect of the new mechanism is seen with the number of context switches of the tasks taking often a lock. A CPU migration leads to one context switch. With the original mechanism, there are approximately 7200 context switches per task, this is the "ping-pong" effect. The enhanced load-balancer avoids this effect and the tasks endure nearly 10 times less context switches.

Task/processor association The last verification concerns the modification of the load-balancer which should favor Linux tasks on NRT CPUs and RT tasks on RT CPUs (because the interrupt latency is better on those processors). The scenario contains two sets of 8 tasks. Both sets are tasks only doing computational processing. The only difference is that the first set are RT tasks with priority 50 while the second are Linux tasks. The CPUs are partitioned as two NRT CPUs and two RT CPUs. All the RT tasks should run on the RT CPUs while the Linux tasks use the NRT CPUs.

The results are summarised in table 5. On standard Linux the tasks are equally balanced between the processors. ARTiS with the original load-balancing code produces mostly the same results. With the modified load-balancer, the repartition of the tasks is much less balanced, as expected. The Linux tasks spend more time on the two NRT CPUs and RT50 tasks are more often executed on a RT CPU. This observation confirms the correctness of the modification, which allows RT tasks to benefit more often from the low latencies on the RT CPUs.

To conclude, the load-balancing modifications were successfully validated. Even with the presence of the "push" policy necessary to avoid inter-processor locks, the balance was as good or better than on the original ARTiS kernel. lbμ has proved to be useful tool. Although it is lacking some possibilities, like being able to see the evolution of number of tasks per CPU, it is at least very convenient to generate reproducible loads.

	Standard Linux				**ARTiS normal LB**				**ARTiS enhanced LB**			
CPU idle time (µs)	0	0	0	0	0	0	0	0	0	0	0	0.2
CPU repartition of Linux tasks (%)	25	25	25	25	34	25	21	21	40	33	4	23
CPU repartition of RT tasks (%)	25	25	25	25	16	25	29	30	12	18	43	27
Linux tasks completion time (s)	94 / 132 / 142				113 / 155 / 187				109 / 145 / 167			
RT tasks completion time (s)	50 / 83 / 103				48 / 81 / 113				47 / 136 / 167			
Linux tasks involuntary ctxt switchs	541				496				500			
RT tasks involuntary ctxt switchs	230				230				350			

Table 5: Execution results of 8 Linux tasks and 8 RT50 tasks with CPU repartition as **NRT NRT RT RT**. Completion time is express as minimum / average / maximum.

6 Conclusion

6.1 Summary

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 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. 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. In addition to the mechanism of automatic migration and the adaptation of the load-balancing, an interface was developed in order to allow the user to dynamically modify the settings of ARTiS. 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 standard kernel. The new load-balancing policies has been proven to be correct with respect to the the-

ory, permitting to keep a high computational power throughput even with the introduced asymmetry in the system.

6.2 Future Work

Although lbμ, the load-balancing observation tool, has provided enough information to validate the load-balancing modifications, several enhancements could be useful. In particular, it would be interesting to be able to follow the load of each CPU along the time. It might also be possible to find a smaller set of metrics which can describe the effectiveness of a given load-balancing. Additionally, the tool could be adapted to also support the observation of real loads, helping the developers and system administrators for their tasks.

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 a given RT processor, it is up to the programmer to manage the share of the processor resources between these tasks. We plan to add the definition of usual real-time scheduling policies such as EDF (earliest deadline first) of RM (rate monotonic) at this level. This extension requires the definition of a task model, the extension of the basic ARTiS API, the implementation of the new scheduling policies. The new RT0 tasks would be periodic tasks running an endless loop. 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] 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] Ben 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, September 2005.

[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, WPDRTS'03*, Nice, France, April 2003.

[5] 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.

[6] Antoine Colin and Isabelle Puaut. Worst-case execution time analysis of the RTEMS Real-Time operating system. In *13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, Delft, The Netherlands, June 2001.

[7] Finite State Machine Labs, Inc. RealTime Linux (RTLinux). `http://www.fsmlabs.com/`.

[8] Cyril Fonlupt. *Distribution Dynamique de Données sur Machines SIMD*. Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, December 1994. (In French).

[9] Freek. Schedtool home page. `http://freequaos.host.sk/schedtool/`.

[10] Philippe Gerum. I-pipe, 2005. `http://www.adeos.org/`.

[11] Nicholas Mc Guire and Qingguo Zhou. Benchmarking - cache issues. In *Seventh Real-Time Linux Workshop, RTLWS'05*, Lille, France, November 2005.

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

[13] Kevin Morgan. Linux for real-time systems: Strategies and solutions. White paper, MontaVista Software, Inc., 2001.

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

[15] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel : Design and Implementation*. Prentice-Hall, 2002.

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

[17] Éric Piel. *Équilibrage de charge pour systèmes temps-réel asymétriques sur multi-processeurs*. Master thesis, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, June 2004. (In French).

[18] 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.

[19] Sillicon Graphics, Inc. REACT: Real-time in IRIX. Technical report, Sillicon Graphics, Inc., Mountain View, CA, 1997.

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

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

[22] 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.

[23] Victor Yodaiken. The RTLinux manifesto. In *Proc. of the 5th Linux Expo*, Raleigh, NC, March 1999.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399