

ARTiS Functional Specification

Éric PIEL Philippe MARQUET Julien SOULA
Christophe OSUNA Jean-Luc DEKEYSER
`Firstname.Lastname@lifl.fr`

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

May 2005

Abstract

The ARTiS system is a real-time extension of the GNU/Linux scheduler dedicated to SMP (Symmetric Multi-Processors) systems. ARTiS exploits the SMP architecture to guarantee the preemption of a processor when the system has to schedule a real-time task. The implementation, which was developed within the HYADES project, 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 allow 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 various aspect a real-time system based on ARTiS. The theoretical principles of ARTiS and its implementation are presented for the developer's interest while some latency measurements and precise description of the deployment of the system are focused toward the needs of the system administrator and of the user.

Contents

Introduction	5
1 ARTiS: Asymmetric Real-Time Scheduler	7
1.1 Partition of the Processors and Processes	7
1.2 Migration Mechanism	8
1.3 Load-Balancing Policy	9
1.4 Interprocess Communication Mechanisms	9
2 Real-Time Application Deployment	11
2.1 Installation	11
2.2 Machine Partitioning	12
2.3 RT Process Identification	13
3 Implementation	15
3.1 ARTiS Migration	15
3.1.1 Migration Triggering	15
3.1.2 Task Migration Pathway	16
3.1.3 Lock Free FIFO	17
3.2 ARTiS Load-Balancing	17
3.2.1 ARTiS Specific Constraints	18
3.2.2 Load-Balancing Implementation	19
4 Performance Evaluation	23
4.1 Measurement Method	23
4.2 Interrupt Latency Types	24
4.3 Measurement Conditions	25
4.4 Observed Latencies	26
Bibliography	27

Introduction

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

One of the approaches to obtain hard real-time performances on a GPOS 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 [1, 2] and SGI REACT IRIX variant [8] follow this principle. However, since only RT tasks are allowed to run on shielded CPUs, if those tasks are not consuming all the available power then there is free CPU time which is lost. The ARTiS scheduler extends this 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 document, we will start by having a look at the basic principles of ARTiS. The second chapter is dedicated to the deployment of a real-time application on a system based on ARTiS. The implementation of the kernel patch is described in depth in the third chapter. The fourth and last chapter presents some latency measurements of the typical HYADES real-time system.

Chapter 1

ARTiS: Asymmetric Real-Time Scheduler

ARTiS is a real-time Linux extension that targets SMPs. Furthermore, the programming model ARTiS promotes on a user-space programming of the real-time tasks: the programmer uses the usual POSIX and/or Linux API to define his 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. As identified in the requirements for the HYADES targeted applications, a maximum response time below $300\mu s$ is targeted for these tasks.

The ARTiS solution keeps interests of both GPOS's and RTOS's by establishing from the SMP platform an **Asymmetric Real-Time Scheduler** 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 1.1.

1.1 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 generalise, 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 when these 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, but 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.

1.2 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 described in [9], ARTiS implements a lock-free FIFO with one reader and one writer to avoid any active wait of the scheduler.

1.3 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 [7].

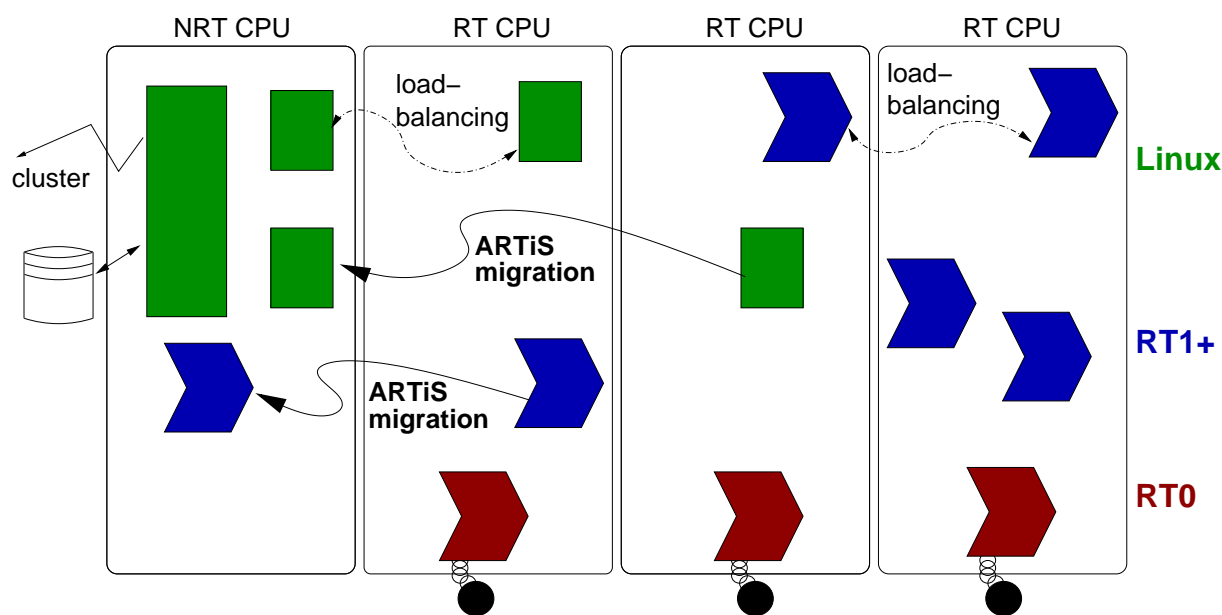


Figure 1.1: Example of a typical usage of a system based on ARTiS. The application is separated along different levels of real-time priorities. Tasks are moved by the ARTiS mechanisms of migration and load-balancing.

1.4 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.

Chapter 2

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 in the processors of the system.

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. ARTiS applications are defined by a configuration of CPUs, an identification of real-time tasks and their processor affinity.

2.1 Installation

ARTiS is provided as a set of Linux kernel patch. They apply against the vanilla Linux Kernel, available at <http://www.kernel.org>. Some versions might need to be applied against the Bull version of the Linux kernel. Use the command:

```
% patch -p1 < ../the-name-of-the.patch
```

ARTiS was designed to run on IA-64 and x86 SMP systems. You can compile it into the kernel by selecting:

```
[X] File systems -> Pseudo filesystems -> /proc file system support
[X] Processor types and features -> Preemptible Kernel
[X] Processor types and features -> Compile the kernel
                                with ARTiS support
```

Optionally you can select:

```
[X] Processor types and features -> Compile the kernel
                                with ARTiS debugging support
[X] Processor types and features -> Compile the kernel
                                with ARTiS accounting support
```

You also need to disable the energy saving functions that could put the CPUs into a state with a long transition latency. For example, on IA-64 the `pal_halt` optimisation must be disabled, this can be done by appending `nohalt` to the kernel command line (in `elilo.conf`).

2.2 Machine Partitioning

The CPUs are partitioned into two sets, the RT and NRT CPU, via a basic `/proc` interface :

- A value greater than or equal to 1 in `/proc/artis/activate` dynamically activates the ARTiS functioning.
- The `/proc/artis/maskrt` file contains the mask (in hexadecimal) of the RT CPUs. It can only be modified while ARTiS is inactive. There must be at least one CPU identified as an NRT CPU.
- Additionally, you can obtain various statistics about ARTiS by reading `/proc/artis/cpustat`.

The active state of ARTiS as well as the `maskrt` properties can also be set on the command line (in `elilo.conf`) via respectively `artis_active` and `artis_maskrt=`. As ARTiS configuration of the machine is dynamic. It is advised to disable ARTiS during the Linux boot phase.

A user may prefer to take advantage of the CPUSSETS patch provided by Bull that also allows such partitioning of a multiprocessor [3].

To maintain coherence with this machine partitioning, a redirection of the interrupts has to be programmed. All IRQs must be delivered exclusively on the NRT CPU, excepting those IRQs used by the RT0 tasks, which must be delivered on the CPU hosting the task. The `/proc/irq/*/smp_affinity` files are used for this purpose. Alternatively, you can use the `change_all_irqs.sh` script available on the ARTiS webpage [5] in the `affinity-utils` package. Please note that on IA-64 with a kernel 2.6.7 we have noticed that some IRQ may not be able to change of mask (this bug is fixed in kernel 2.6.11). You must not select as RT CPU a CPU where the IRQ are.

Also note that as of May 2005, there seems to be a latency problem on SMPs with Linux. This bug appears both on x86 and IA-64, even when the computer is purely idle and without ARTiS. The problem: there is always one (and only one) CPU where the latencies are much higher than anywhere else (10x or 100x higher). This CPU seems to be the same that receive all the IRQs at boot, so you can detect it by having a look at `/proc/interrupts`. Set this CPU as an NRT CPU, that's all.

2.3 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. Obviously, 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 order of `sched_setscheduler()` and `sched_setaffinity()`. The priority set up must always be **before** the affinity set up (otherwise, the affinity cannot be guaranteed). In case you don't want (or cannot) re-compile your application to fit those specific requirements for ARTiS, you can still change it to RT0 from outside (`schedtool` is probably what you need, available at <http://freequaos.host.sk/schedtool/>).

```
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 2.1: *RT0 identification*

Figure 2.1 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 Linux 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_rtlplus(pid_t pid, int policy, int priority);  
int artis_leave_rtlplus(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 they will no longer be eligible for execution when entering a non-preemptible code section. In addition, the CPU affinities of RT1+ tasks should also include at least one RT CPU.

Chapter 3

Implementation

The ARTiS model is currently implemented as a modification of the 2.6 Linux kernel. The main modification 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

Inter-CPU locks are unsafe because an NRT processor may block an RT processor that shares the lock, consequently ARTiS is not able to directly use the original Linux run-queues. 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.

Therefore, 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 access to these FIFO must be lock free: RT processors should never share any lock with any NRT processor.

The algorithm proposed by Valois [9] 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 in [4] and can be enumerated as follows:

- information update policy: how to renew statistics on the entire system,
- 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.

The trigger policy 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.

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.

3.2.1 ARTiS Specific Constraints

In addition to the normal load-balancing in Linux which will only be activated inside the RT CPU set and inside the NRT CPU set, we have specified three other constraints that the load-balancer will take care of. An in-depth study of all the different load-balancing scenarios which highlights the constraints is available in [7].

Load-balancing without inter-CPU locks One of the main changes which is required from the original load-balancing mechanism is the removal of inter-CPU locks. For the same reasons as those described in section 3.1, the locks will have to be avoided in order to insure RT properties on RT CPUs.

Return of the RT1+ tasks The ARTiS migration may move RT1+ tasks from an RT CPU to an NRT CPU. However best latencies are available on the RT CPUs, so RT1+ tasks should move back to any RT CPU as soon as possible once preemption is re-enabled. Therefore, one task of the load-balancer will consist of migrating RT1+ tasks from NRT CPUs to RT CPUs.

Reduction of the RT CPUs load It might occur that an NRT CPU has less load than the RT CPUs. In this case, to get the best performance from the computer, some tasks should be migrated from an RT CPU to the NRT CPU. However, as the latencies are better on real-time processors, non real-time tasks should be given migration priority. In practice, most of the tasks trigger preemption disabling code with enough frequency so that such load-balancing is usually not needed. Still, it is necessary to handle this case in order to guarantee the best use of all the CPUs in every configuration (for instance with tasks which are only concerned with computational work).

3.2.2 Load-Balancing Implementation

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.

Load-balancing evaluation tool A specific tool was designed to test the load-balancer correctness. It allows to run a specific set of tasks characterised by properties (CPU usage, scheduler priority, processor affinity...) to be launched in a very deterministic way. Some statistics about the run are provided but the interpretation of the results is not straightforward. This tool, called $lb\mu$ is available on the ARTiS webpage [5].

Run-queue length weighting The pairing policy of Linux selects the processor that will receive the tasks by locating the one which is the most loaded. The load is estimated using the number of task 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 application.

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 3.1(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 3.1(b). This type of scenario is highly probable on an ARTiS system because the real-time tasks are asymmetrically distributed.

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

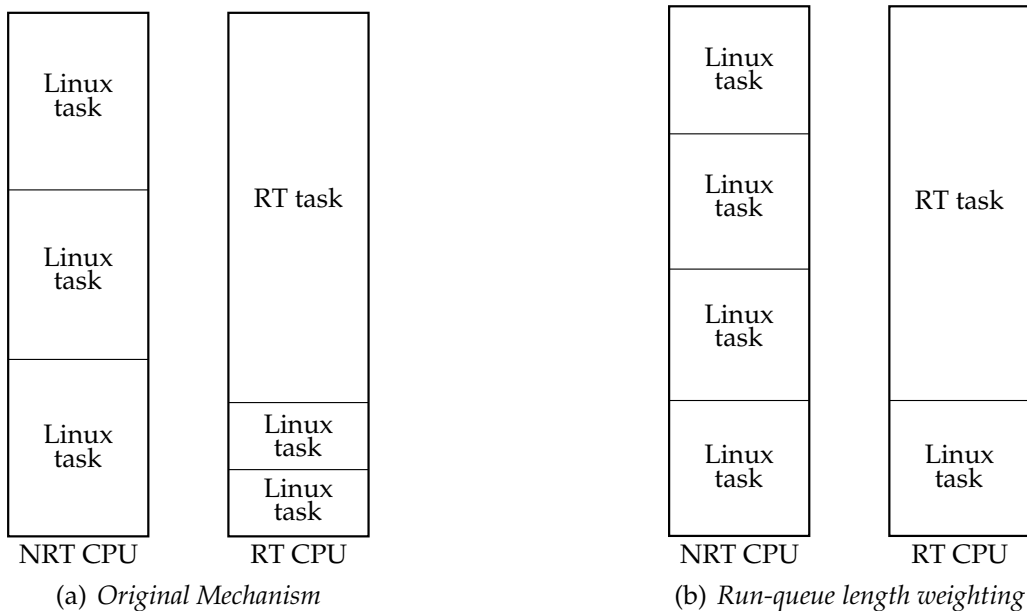


Figure 3.1: *Improvement of the fairness between Linux tasks done using weighting of the run-queue length.*

ratio of time that was consumed by real-time tasks. 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.

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. Using the RT-FIFO allows this problem to be resolved 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 any search for a busier processor at the moment they enter into the idle state.

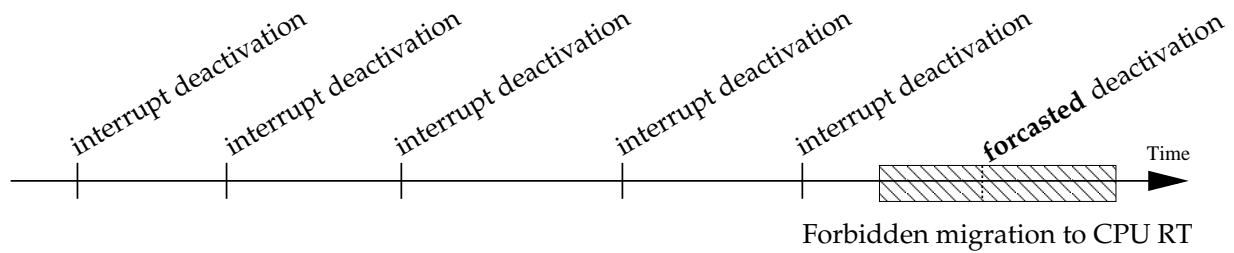


Figure 3.2: *Period of forbidden migration (hatched rectangle). The period is deducted from the study of the previous behavior of the given task.*

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. 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 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 3.2.

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 these conditions is available in [10].

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.

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 processors 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.

Chapter 4

Performance Evaluation

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

Nevertheless, this measurement is only one possible among others. For instance, other measurements could consist in observing the difference of execution time of a real-time task, or compare the efficacy of the load-balancing mechanism.

4.1 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 4.1):

- 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.

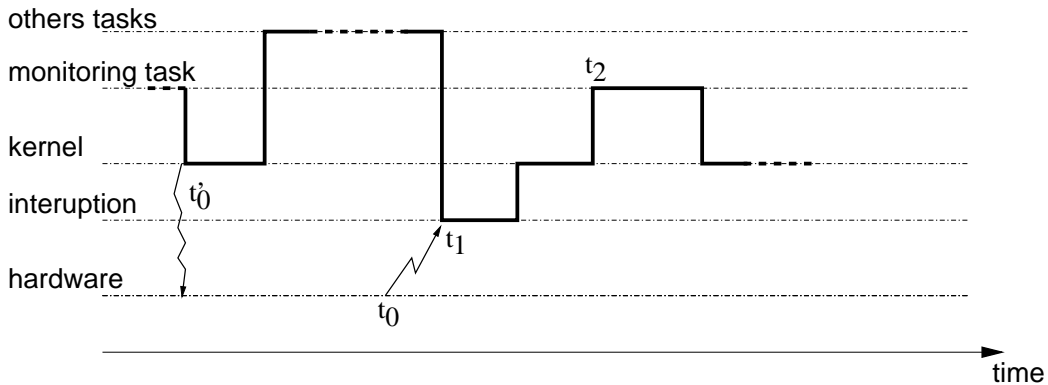


Figure 4.1: Chronogram of the tasks involved in the measurement code.

We conducted the experiments on a 4-way Itanium II 1.3GHz machine. It ran on an instrumented Linux kernel version 2.6.7 as given by Bull (version 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 [6]).

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, making each test composed of approximately 300 million measures. Given such duration, the results are reproducible.

4.2 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 (`pfm_interrupt_handler()` in our case). This is the latency that a driver would have if it was written as a kernel module following the usual design method.
- The **user latency**, $t_2 - t_0$, 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 a real-time application would have if it was written entirely in the user-space. In order to have the lowest latency, the application was notified via a blocking system call (a `read()`).

The real-time tasks designed to run in user-space are programmed using the usual and standard POSIX interface. This is one of the main advantage that ARTiS provides.

Therefore, within the ARTiS context, user latency is the most important latency to study and analyze.

4.3 Measurement Conditions

The measurements have been conducted under different conditions. We have identified two unrelated parameters which affect the interrupt latencies:

- **The load.** The machine can be either idle (without any load) or highly loaded (all the programs described below are executed concurrently).
- **The kernel preemption.** When activated, this new feature of the 2.6 Linux kernel allows tasks to be rescheduled even if kernel code is being executed. This configuration of the Linux kernel corresponds to the so-called “preemptible Linux kernel”.

Three configurations are presented, they were selected for their relevance toward latency. First of all, the Bull kernel (preemption is activated) was measured without and with load. Then, a similar kernel but with the ARTiS implementation was measured. This last configuration is only presented with load because, when idle, the results were extremely close to the Bull kernel configuration.

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. To achieve this goal, 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 by reading and writing at random positions in a large array.

4.4 Observed Latencies

The tables 4.1 and 4.2 summarize the measurements for the different tested configurations. Three values are associated to each latency type (kernel and user). “Maximum” corresponds to the highest latency noticed along the 8 hours. The two other columns display the maximum latency of the 99.999% (respectively 99.999999%) best measures. For this experiment, this is equivalent to not counting the 3000 (resp. 3) worse case latencies.

Table 4.1: Kernel latencies of the different configurations.

Configurations		Kernel		
		99.999%	99.999999%	Maximum
Linux with preemption	idle	84 μ s	93 μ s	94 μ s
Linux with preemption	loaded	76 μ s	100 μ s	108 μ s
ARTiS	loaded	3 μ s	14 μ s	16 μ s

Table 4.2: User latencies of the different configurations.

Configurations		User		
		99.999%	99.999999%	Maximum
Linux with preemption	idle	92 μ s	103 μ s	151 μ s
Linux with preemption	loaded	678 μ s	23ms	42ms
ARTiS	loaded	54 μ s	114 μ s	119μs

Although the study of an idle configuration does not bring very much information by itself, it 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. Additionally, the 99.999% does have much lower latencies than the maximum, 678 μ s for the user latencies, this is good for soft real-time applications but not for hard real-time applications where even one missed deadline is unacceptable. Let’s also note that for all the measurement configurations the average user latency was under 4 μ s.

In the ARTiS configuration, both kind of latencies are very significantly lowered, with a maximum of 119 μ s for user latencies. This is below the specified maximum latency of 300 μ s. Consequently, the system can be considered as a hard real-time system, insuring real-time applications very low interrupt response.

Bibliography

- [1] Stephen Brosky. Symmetric multiprocessing and real-time in PowerMAX OS. White paper, Concurrent Computer Corporation, Fort Lauderdale, FL, 2002.
- [2] 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.
- [3] Simon Derr and Sylvain Jaugey. CPUSETS for Linux home page. <http://www.bullopen-source.org/cpuset/>.
- [4] 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, December 1994. (In French).
- [5] Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille. ARTiS home page. <http://www.lifl.fr/west/artis/>.
- [6] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel : Design and Implementation*. Prentice-Hall, 2002.
- [7] Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. Load-balancing for a real-time system based on asymmetric multi-processing. In *16th Euromicro Conference on Real-Time Systems, WIP session*, Catania, Italy, June 2004.
- [8] Silicon Graphics, Inc. REACT: Real-time in IRIX. Technical report, Silicon Graphics, Inc., Mountain View, CA, 1997.
- [9] John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, October 1994.
- [10] Éric Piel. *Équilibrage de charge pour systèmes temps-réel asymétriques sur multi-processeurs*. Mémoire de DEA, Université des sciences et technologies de Lille, Lille, France, June 2004.