Linux for High Performance and Real-Time Computing on SMP Systems[*]

Dominique RAGOT, Yulen SADOURNY
Thales, Colombes, France
{dominique.ragot,yulen.sadourny}@fr.thalesgroup.com

Denis FOUEILLASSAR, Philippe COUVEE
Bull, Grenoble, France
{denis.foueillassar,philippe.couvee}@bull.net

Léonard SIBILLE
CEA List, Fontenay aux Roses, France
leonard.sibille@cea.fr

Jean-Luc DEKEYSER, Philippe MARQUET, Eric PIEL, Julien SOULA
LIFL, University of Lille, France
{jean-luc.dekeyser,philippe.marquet,eric.piel,julien.soula}@lifl.fr

Hugo KOHMANN
Dolphin Interconnect, Oslo, Norway
hugo@dolphinics.no

Alexis BERLEMONT
Openwide, Paris, France
alexis.berlemont@openwide.fr

Abstract

Applications that require a combination of high-performance computing capabilities and real-time behavior, although pervasive (simulation, medicine, training, multimedia communications), often rely on specific hardware and software components that make them high performance but expensive, and quite difficult to develop, validate and moreover upgrade. The increasing performance of COTS and the volume of software developed for these applications lead to the consideration of incremental development schemes in addition to sole performance. In the ITEA Hyades project, industrial companies, research centres and academic departments, propose a complete set of software technologies aimed at adding real-time capabilities to multi-processor systems, with a strong commitment to standards. In this paper we present the application requirements with respect to real-time, the architectural model proposed, as well as the reasons for using the Linux operating system. Then, we introduce software components that have been selected to provide real-time needs, among which are Adeos and ARTiS, and their expected contribution to global performance. Finally we provide performance measurements for these elements.

[*]This work has been done in the scope of the Hyades project, ITEA 01010

1 Introduction

The integration of digital systems in many aspects of life is now a reality of every day. In many fields of activity: office, leisure, health, security, transportation, we are indeed communicating with computers, without having to know how this communication is managed. Terminals, computers and networks have simply to bring together the required services to the end-users in a seamless fashion. This integration requires infrastructure components that must deliver both functionality and performance. For a majority of systems, performance relates to throughput, but for a growing number of domains (video and audio contents delivery, virtual reality, manufacturing process control, sensor fusion) performance relates to timely execution. Such applications have usually required non-standard and costly hardware and software solutions. Their specificity had been for years the justification for the use of specific technology at all levels: specialized DSP processors, specialized operating systems lacking the support of standard APIs and requiring custom applications, and also specialised cluster interconnects.

Moreover the diffusion and utilization of parallel distributed systems based on COTS (components off the shelf) technology has widely increased in last years. Today, using COTS, it is possible to build up powerful clusters not only for number crunching but also for highly parallel commercial applications. Many computer manufacturers have adopted this approach, and now high performance computing systems are available at a price very low with respect to one decade ago.

Real-time capabilities for these systems have not reached a comparable level of maturity due to limited market size. In order to evaluate what level of performance could be reached, a multidisciplinary team [1] has designed and developed real-time extensions for parallel systems whose requirements, contents, and results are exposed in the following chapters.

2 Applications requirements

For complex applications, real-time constraints are expressed at several levels of interaction. When there is close man-system interactions (e.g in virtual reality applications), the constraints are expressed in relation to perception. On the other hand, for data acquisition systems, the receiver/emitter must not cause data to be lost due to lack of temporal control over some asynchronous events.

Because they are complex, these applications also make large usage of components that are not in dealing at all with real-time issues. For instance all back end processing such as classification, database access, global configuration and monitoring, typically rely on several legacy or third-party middleware and tools components. The underlying software architecture has to provide capabilities to integrate these components in a seamless fashion.

In order to assess the versatility of the proposed architecture for this class of applications, we have chosen the following two cases:

2.1 Virtual Reality

One application of the real-time kernel is the simulation of industrial parts in virtual reality. Industrialists currently use real-life mock-ups for assembly testing. This process takes a large amount of time and money. Virtual reality makes such testing easier and cheaper. Once converted into appropriate 3D computer models, industrial parts are integrated into a simulation framework which computes dynamics and collisions between parts. In addition, the simulation is connected to a force-feedback device which enables the user to feel collision forces, as shown on figure 1.

This device, however, must be fed with force data at a very high rate (1kHz, typically). Failure to respect this rate results in jitter, and eventually makes the simulation crash. Today, the simulated 3D models can only consist of a few thousand polygons, because of this rate constraint. A SMP machine enables developers to isolate and make parallel the dynamics and collision processes, which should give dramatically better performance. The real-time patch will ensure the real-time constraint is enforced. All this should result in more detailed models, and better testing accuracy.

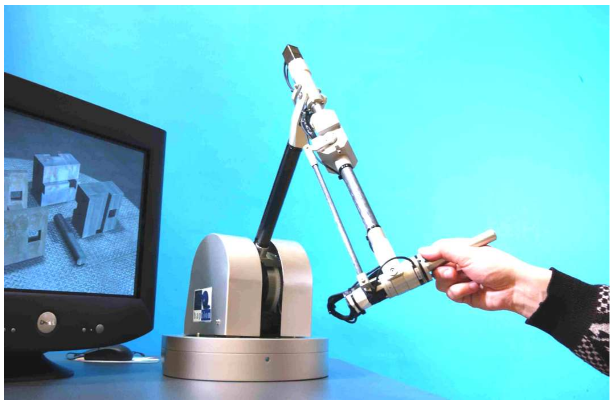

FIGURE 1: *Linking a haptic device to a 3D simulation*

2.2 Video proxy

The video proxy is an application located somewhere in the network between the server and the end-user. It is typically placed at the edge of a network, where the available bandwidth or the security requirements change (see figure 2). The purpose of a video proxy is mainly to adapt the video streams going through it, depending on the users' characteristics at the end of the delivery chain.

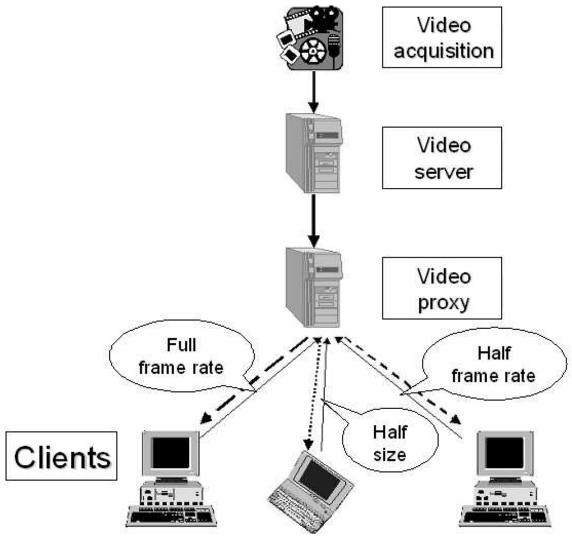

FIGURE 2: *Proxy in video distribution*

Description and Functionality The processing of a video stream during its transmission requires specialised modules, due to the high data rates involved. A video proxy allows to perform user authentication as well as filtering and logging on any traffic that traverses the proxy server. But its main and most heavy task consists of pure video-related processing, specifically at the edge of heterogeneous networks:

- Transcoding of video content, i.e. dynamic adaptation to ensure a certain quality-of-service. Some content formats are designed to optimise scalability, such as Motion JPEG 2000 or the upcoming MPEG-SVC, thus allowing to transcode streams without going through the entire encoding chain.

- Scalable encryption to ensure the confidentiality of critical data. This kind of encryption selects the byte chunks to cipher and allows to keep the structure of the video content intact. One of the main advantages of these techniques is to allow the transcoding of ciphered streams.

A generic video proxy can implement some traffic control, but does not contain any firewalling capability. In this way, it can be deployed behind a traditional firewall platform. Therefore, a typical use on a private network area can be the following: a main firewall accepting inbound traffic, determines which application is being targeted, and then hands off the traffic to an appropriate proxy server, e.g. videos to the video proxy. This way, such a dedicated proxy can be used to decrease the work load on the firewall and to perform more specialised processing that otherwise may be difficult or even impossible to perform on the firewall itself.

Application constraints Today, the two main limitations for video proxy modules are the low flexibility of content formats, although some standards are emerging, and the computing power of networks nodes, which have high performance for basic processes such as routing but are not optimised for more complex computation like transcoding.

The critical parameters for the machine when the proxy runs are the CPU-load and the achieved quality of service for the clients behind. The application performs a continuous, on-stream processing and must do the work in real-time, so that the video quality, resolution and frame-rate remain constant on the end-users' players.

3 The proposed architecture

Multiprocessor systems are well suited to provide the required processing power for such applications as well as a choice of operating systems and middleware tools, at least when excluding real-time issues. Including real-time capabilities directly usable by application designers dramatically reduces this choice and offers a limited set of solutions:

1. pure RTOS-based solutions are usually quite limited in terms of middleware and tools supported, and only a very few of them have support for multiprocessor systems. The application developer has usually no choice but to partition the number of processors available in two sets: one with RT capabilities running a RTOS, and the other running a GPOS with all applications. Communications within and between sets are done using MPI-like primitives. Besides having to statically define resources for RT and non-RT parts of the application, this solution requires that all communication software be developed in a way that is highly dependent on the underlying machine architecture.

2. mixed RTOS and GPOS solutions are interesting since they can overcome some of the problems raised above. In this approach, every processor in the system is able to operate in both RT and non RT modes, eliminating the constraint of a-priori partitioning. However, communications still have to be implemented using ad-hoc mechanisms. Furthermore the application code on every processor has to be rewritten in order to cope with RTOS-GPOS communication facilities and the specific, non-standard RTOS API.

3.1 Software and hardware requirements

The "ideal" solution would be the least invasive possible **from the application designer's/developer's viewpoint**. All the facilities that are commonly used to write RT programs have to be present and easily derived from the GPOS starting point. From this perspective, none of the solutions presented above are fully satisfactory. One would expect to program RTOS services within a GPOS infrastructure. To make this feasible, we need the following capabilities:

- A communication infrastructure that is as simple as possible and whose performance is compatible with RT issues. In this respect Symmetrical Multi-Processing is very appealing since the application developer does not have to care about passing data between processors. The underlying communication primitives are integrated in the kernel so minimal overhead can be obtained, which is very important for controlling latency. However, SMP has important limitations such as scalability. In such a case we need to have a way of improving scalability at the expense of communication primitives insertion while retaining low-latency capabilities.

- A GPOS compliant with POSIX standard, and extensible to Real-Time POSIX profiles. This allows easy porting non-RT applications as well as the integration of components with RT requirements.

- A high-end processor capable of delivering a good balance of CPU power and I/O throughput.

Meeting all these requirements has resulted in an architecture model that accommodates a good balance between ease of programming and scalability. The two-level communication infrastructure implied by this tradeoff has resulted in a two-level machine composition.

3.2 Machine architecture

Our reference machine is made of several SMP computing nodes linked together by SCI interconnects. The SMP node is a Bull Novascale system using 4 Intel Itanium2 CPUs and running the Mandrake Linux operating system.

3.2.1 SMP nodes

The Bull NovaScale® family includes modular, standard-based servers designed to run the most demanding business-critical and scientific applications. Bull NovaScale series are designed for transactional and decisional applications, as well as for consolidation and scientific/technical computing.

The Bull NovaScale family includes five series in order to meet a variety of customer requirements.

Bull NovaScale® servers provide the following advantages:

- High scalability with scale-up and scale-out configurations: SMP (Symmetrical Multi-Processing) systems up to 32 processors and clusters.

- High performance with large memory capacity, the EPIC (Explicitly Parallel Instruction Computing) technology, high bandwidth, floating-point processing.

- Mainframe-class Reliability/Availability/Service features with hardware redundancy, hot-swap, hot-plug capabilities, ECC and parity check and an integrated management console.

- Great flexibility with multi-operating systems support.

- Investment protection with upward hardware and software compatibility throughout the Intel® Itanium® Processor Family and Independent Software Vendors endorsement.

At the heart of high-end NovaScale servers, the FAME Scalability Switch (FSS), developed by Bull, federates the processors and optimizes memory and I/Os. The FSS, Intel® Itanium® 2 processors and the Intel® E8870 chipset, combined with the QBB (Quad Brick Block) provide an exceptional performance linearity and scalability to high-end NovaScale servers.

Bull NovaScale® servers set a world record in performance: Bull NovaScale® 5080 with 8 Intel®

Itanium®2 processors Performance: 175,366.24 tpmC.

It is worth noting the flexibility of this architecture model, which can be tailored to provide ease of programming either using larger SMP nodes (up to 32 CPUs per node) or scalability by increasing the number of nodes. In the latter, performance for real-time applications can be improved by increasing the connectivity of the SCI links from 1D to 2D and even 3D topologies.

3.2.2 Clustering

The clustering is made with SCI technology from Dolphin which is known to provide low latency for data transmission. The software stack available to developers is layered, as shown in figure 3, and its principal components are:

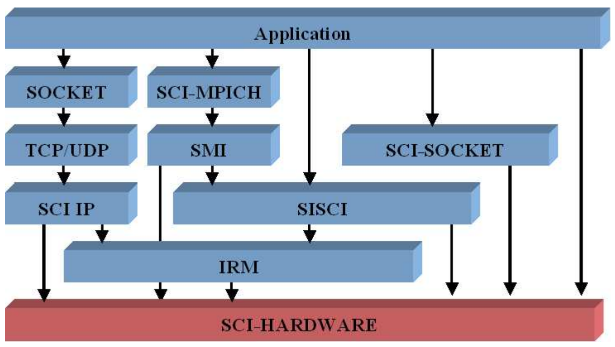

FIGURE 3: *SCI stack building blocks*

SCI Socket The SCI SOCKET software [2] provides a fast and transparent way for applications using Berkeley sockets - TCP/UDP/IP to use SCI as the transport medium. The major benefits are plug and play, high bandwidth and much lower latency than network technologies like Gigabit Ethernet, Infiniband and Myrinet. The SCI SOCKET uses SCI remote memory access to implement a fast and reliable connection. It enables standard sockets to utilize SCI as a transport without modifications. All drivers and software is Open Source, available under LGPL/GPL.

SISCI Userlevel API The SISCI SDK is a C system call interface to ease customer integration to Dolphins cluster interconnect. The SISCI software supports clusters of hundreds of nodes. Typical applications for SISCI includes:

- Bus bridging PCI-PMC-cPCI
- Remote access to IO Systems
- Reflective memory like clusters
- High Availability servers / Fast fail over
- Fat pipes / low latency messaging
- Bridging between heterogeneous systems (x86-PPC-SPARC-AMD64)
- Bridging between operating systems (Linux-Solaris-VxWorks-Lynx-Windows-NT/2000/2003/XP)
- MPICH

SISCI provides the customer with an API library to harness the power of a cluster of commodity personal computers or workstations. SISCI operates using system calls on SCI-descriptors and local and remote memory SCI-segments. With SISCI it is easily possible for your application running locally to operate on remote memory-segments in user space. By using SISCI, a customer application can bypass the limitations of traditional network solutions, avoiding time consuming operating system calls, and network protocol software overhead.

The relationship of the software and the hardware is shown in the following table 1:

Environment	Layers
Application:	Customer application code
User space:	SISCI API C library
Kernel:	SISCI driver,
	IRM driver
I/O bus:	PCI bus
Hardware:	Dolphin adapter cards
	with interconnect

TABLE 1: *SCI layered software*

Key product features:

- Enables transparent access to remote memory for ultra low latency access (shared memory).
- Provides memory-to-memory DMA transfers for low overhead data copying.
- Provides read/write block operations.
- Local and remote interrupt handling.
- Supports multiple adapters per host for increased fault tolerance or speed.
- Supports hot-pluggable links for high availability operation

- Error checking functions available to application for ensuring guaranteed data delivery or client notification.
- Programmers need not worry about normal parallel pitfalls such as race conditions and synchronization.
- Supports heterogeneous clusters consisting of multiple operating systems and hardware platforms.

SCI MPICH SCI-MPICH is an open source project adding native SCI support to the MP-MPICH software. The work is done by the University of Aachen, Germany. The main project web is found at `http://www.lfbs.rwth-aachen.de/users/joachim/SCI-MPICH`.

Performance measurement objectives In terms of performance measurement the Hyades project has the objective to make an evaluation of the different telecommunication stack possibilities and to conclude on the best solution according to architecture choice.

Solutions without source code adaptation like multithreaded for SMP machine, MPI compliant for cluster will be evaluated first.

Latencies and throughput will be measured according to packet size variations.

As such, this machine architecture provides a hardware frame for RT features, but does not offer the software facilities to 1) obtain RT behavior at the application level, and 2) accurately measure timings assessing proper RT system behavior.

4 RT behavior at application level

Although the recent Linux 2.6 kernel is widely known to be "preemptible", this capability remains insufficient when dealing with real-time. What is sufficient to play streaming audio and video on a client PC would prove unreliable under constrained timing requirements. As an example, the timer interrupt latency jitter ranges from 5 μs to more than 10 ms on a basic desktop PC.

On the chosen architecture, one could first try to bring RT functionality only within a SMP node. The main interest of this approach is to isolate what is used by the application programmer from what is really implemented in the kernel. Since the Linux operating system favors throughput and fairness, and since real-time is essentially a matter of latency and unfairness, we need to extend the kernel to provide the missing features. As a design choice, we chose not to re-engineer the whole kernel - that would be rather unrealistic anyway, given the amount of work required - but to develop add-ons that would be active beside normal (original) kernel mechanisms under certain conditions.

The aspects whose behavior can be adapted to meet our goals are:

Interrupt virtualization The most stringent requirement is noticeable at the lowest level; interrupts are being handled by Linux in a "best effort" scheme with no notion of criticity whatsoever. An add-on to the interrupt management is needed to provide interrupt prioritization. Since the 2.6 linux kernel makes systematic use of interrupt management functions in replacement of the hardwired `cli/sti` assembly instructions, it is possible to insert an underlying interrupt virtualization layer. To fit this purpose, the Adeos nanokernel [3] has been ported to the IA-64 processor architecture in SMP configuration. It provides the underlying support for interrupt management and for building a software barrier shielding the entire Linux kernel from non real-time interrupts when real-time activities are running.

Per processor scheduling By taking advantage of the multiprocessor architecture, it is possible to dedicate some CPUs to RT tasks by managing processor affinity. However, these CPUs are mostly idle. A new technique, ARTiS has been developed in order to allow preemptible sections of application to be executed on these RT CPUs, thus allowing a more well-balanced system while keeping RT behavior intact.

These properties defined, new capabilities specific to RT could be implemented in the kernel. They are detailed in the following paragraphs.

4.1 DIC: fast path for RT processing

Deterministic Intensive Computing is implemented as an Adeos domain and has the following capabilities:

- promote regular Linux tasks to high-priority DIC tasks.
- activate the interrupt shield to protect running DIC tasks from unwanted Linux preemption.
- trigger the immediate rescheduling of DIC tasks upon reception of a real-time event.
- provide support for very high resolution timers to the DIC tasks.

The consequence of the above is that the DIC domain must have a higher priority than Linux in the Adeos pipeline.

As such, the DIC domain derives from the RTAI/fusion technology [4] with enhancements specific to multi-processor machines such as: port to Intel Itanium2 processor and SMP support.

The application-level API for DIC is an extension of the usual Linux pthreads interface. Tasks created by the `pthread_create()` POSIX call are attached to the DIC domain `pthread_init_rt()`. High-precision timers are activated (resp. stopped) by the `pthread_start_timer_rt()` (resp. `pthread_stop_timer_rt()`) functions. Other functions such as `pthread_time_rt()` return the internal time as maintained by the DIC time source in nanoseconds. It is also possible to suspend thread execution for a specific amount of time and to schedule threads at a periodic rate.

Once attached to DIC, threads keep the ability to issue regular Linux system calls at the expense of losing their real-time properties until returning from the call. However, the interrupt shield protects them from unwanted asynchronous preemption from other non-DIC tasks.

In addition, some native Linux system calls are dynamically substituted by DIC counterparts when called from a DIC thread. This is the case for the `nanosleep()`, `getitimer()` and `setitimer()` system calls. Such substitution is performed at kernel level in order to keep the Linux ABI unchanged.

4.2 ARTiS: RT asymmetric scheduler

Among the numerous proposals of RTOS, one approach exploits the SMP architectures and relies on the shielded processors or **asymmetric multiprocessing principle**. On a multiprocessor machine, the processors are specialized being either real-time or non real-time: Real-time processors will execute real-time tasks while non-real-time processors will execute non-real-time tasks. Concurrent Computer Corporation RedHawk Linux variant [8, 7] and SGI REACT/pro, a real-time add-on for IRIX [10] follow this principle. However, since only real-time tasks are allowed to run on shielded CPUs, if these tasks do not consume all the available power then some CPU resources will be wasted. The ARTiS proposal enhances this basic concept of asymmetric real-time processing by allowing resource sharing between real-time and non-real-time tasks.

ARTiS promotes a programming model based on a user-space programming of the real-time tasks: The programmer uses the usual POSIX and/or Linux API to define his applications. These tasks are real-time in the sense that they are identified as high priority and are not perturbed by any non real-time activities. For these tasks, ARTiS targets a maximum response time below 300μs.

The core of the ARTiS solution is based on a strong distinction between real-time and non-real-time processors, and also, on migrating tasks which attempt to disable the preemption on a real-time processor. To provide this system ARTiS proposes:

- The partition of the processors into two sets;
- Two classes of RT processes;
- A specific migration mechanism;
- An efficient load-balancing policy;
- Asymmetric communication mechanisms.

The processors are partitioned into two sets A NRT CPU set (Non-Real-Time) and an RT CPU set (Real-Time). Each one has a specific 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 identified These are standard RT Linux processes, they just differ in their mapping:

- Each RT CPU has just one bound RT Linux task, called **RT0** (a real-time task of highest priority). Each of these tasks has the 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 ensures a lowest latency possible latency 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. These tasks are called **RT1+** (real-time tasks of priority 1 and below). They can use CPU resources efficiently if RT0 does not consume all the CPU time. To keep a low latency for RT0, the RT1+ processes are automatically migrated to a NRT CPU by the ARTiS scheduler when they are 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 a NRT CPU, it keeps its high priority above the Linux tasks.

A specific migration mechanism 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 a NRT CPU when they try to disable the preemption. One of the main changes which is required from the original Linux load-balancing mechanism is the removal of inter-CPU locks. To effectively migrate the tasks, a NRT CPU and an RT CPU have to communicate via queues. We have implemented a lock-free FIFO with one reader and one writer to avoid any active wait of the ARTiS scheduler based on [11].

An efficient load-balancing policy A dedicated 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. By definition, the RT0 tasks will never migrate. 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 performance for the global system, particular asymmetric load-balancing algorithms have been defined [9].

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 an RT1+ task will involve the migration of the RT1+ task before this latter takes on the lock, to avoid entering into 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.

Modification of the Linux kernel The ARTiS model is being implemented as a modification of the 2.6 Linux kernel and a version is already available at the ARTiS web-page [6]. Results of measurements of the response time latencies on this current implementation are available in the section 6.1, "ARTiS Performance Evaluation".

5 Tracing events in the system

5.1 High Resolution Timers for IA-64

One important point with regard to real-time systems is the provision of accurate time information to the developer as well as ability to trigger events at precise times. In the POSIX norm this action is possible via *timers*. In a real-time context a typical use of timers is the execution of a periodic code with high frequency: the real-time application sets a timer to a specified frequency and then it receives a signal periodically. Timers can also be used as watchdogs for real-time routines: a timer is set up at the beginning of the routine and later, if the given time limit has been reached, a special code is called and changes the operation to degraded mode.

The current Linux kernel on IA-64 already supplies time information with precision in the order of 1 μsec, which is enough for the application we are targeting. However, the current implementation does not provide timers with resolution better than 1 *m*sec which can lead to delays up to 3 *m*sec compared to the requested time. This is due to the fact that timers can only be checked at a clock tick, which happens only 1024 times per second on IA-64. The amount of time between two ticks is called a *jiffy*.

The "High Resolution Timers" project aims at solving this issue. It is currently maintained by MontaVista [5] for x86 processors. The adaptation for the IA-64 architecture is done inside the Hyades project [12]. The first part of the project, which consisted of implementing a POSIX compliant version of the timers inside the kernel, has already been inserted into the official kernel. The other part of the project is to modify the timers' implementation so that precision can be 10 to 1000 times better than now.

The implementation starts by introducing the subdivision of jiffies using *sub-jiffies*. The sub-jiffy unit is dependant on the hardware clock in use, nevertheless there is a known constant number of sub-jiffies in each jiffy. Thanks to this sub-division, it is possible to save with precise information the time at which a timer expires. The other and main idea of the project was to use the advanced features available in new clock sources to dynamically program the next tick. Instead of having a perfectly periodic clock, the kernel re-calculates after each tick the exact next time an interruption has to be triggered according to the next timer about to expire. Two cases are possible:

- one timer will expire between the actual tick and the next jiffy, then the clock is modified to generate an interrupt earlier, at the exact time of the timer.
- the next timer is after the next jiffy, then the next interrupt is at the next jiffy.

On IA-64, the clock source is a processor register. It allows the time to be read with accuracy up to the cycle, but this also implies that on multi-processor systems the registers have to be synchronized. Therefore, on multi-processor computers only 500 cycle precision can be guaranteed. With all the additional overheads, the resolution is approximately 10ms. This is about 100 times better than the original implementation and it is sufficient for the applications on which the Hyades project focuses. This second part of the "High Resolution Timers" project is still currently in development, both for the x86 part and for the IA-64 port.

5.2 LTT for IA-64

The Linux Trace Toolkit, more commonly known as LTT, is a fully-featured tracing system for the Linux kernel. It includes both the kernel components required for tracing and the user-level tools required to view the traces.

The following are some of LTT's main features:

- Linux kernel tracing capabilities with 48 unique trace points.
- Variable-length events minimizing overall trace size.
- Micro-second event time-stamps.
- Minimal performance overhead (< 2.5%).
- Completely configurable trace option during kernel build.
- Multi-platform support: i386, PowerPC, S/390, SuperH, ARM, and MIPS.
- Fully-featured graphical user interface with event graph, system and per-process analysis, and raw event descriptions.
- Single copy of traces between kernel-space and permanent storage.
- User-selectable and dynamically configurable event trace mask.
- Dynamic creation and logging of custom events both in kernel and in user space.
- Support for custom formatting of custom events.
- General hooking interface available within kernel trace facility.

Many of these features hide many sub-features which contribute to the completeness of the toolset provided.

6 Results

6.1 ARTiS Performance Evaluation

While implementing the ARTiS kernel, we conducted some experiments in order to evaluate the benefits of the approach in terms of interrupt latency. The measurement program is available on the ARTiS web-page [6]. We distinguished two types of latency, one associated with the kernel and the other one associated with user tasks.

Measurement method The experiment consisted of measuring the elapsed time between the hardware generation of an interrupt and the execution of the code concerning this interrupt. The measurement task sets up the hardware so it generates the interrupt at a precisely known time, then it gets unscheduled and waits 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 real-time applications: waiting for a hardware event to happen, processing data according to the new parameters, sending new information and returning to waiting mode. For one interrupt there are four associated times, corresponding to different locations in the executed code (Figure 4):

- 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.

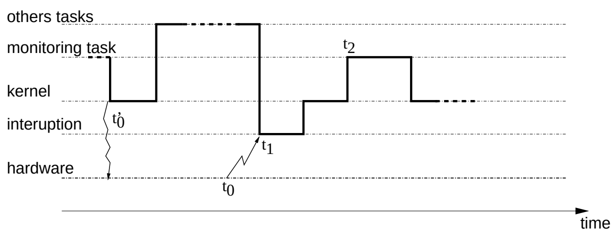

FIGURE 4: *Chronogram of the tasks involved in the measurement code.*

We conducted the experiments on a 4-way Itanium II 1.3GHz machine. It ran on an instrumented Linux kernel version 2.6.4. The `itc` (a processor register counting the cycles) is the timer on which all the measurements are based and the interrupt was generated with cycle accurate precision by the PMU (a debugging unit available in each processor [13]).

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 is composed of 300 million measures, making each test about 8 hours long.

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 an 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.

The current ARTiS configuration Although ARTiS is not yet complete, the current version already implements the detection of real-time endangering functions and can migrate the task from RT CPU to NRT CPU. The dedicated load-balancing mechanism is not yet present. However, the original load-balancing has been modified so as not to run on RT CPUs, in order to avoid inter-locking between CPUs. Migration from a NRT CPU to an RT CPU is still present so that RT CPUs can be used during their idle time. In respect to the latency measurements this configuration should give results very similar to the final version of ARTiS.

Measurement conditions The measurements have been conducted under different conditions. We have identified two unrelated parameters which effect 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".

Of the four possible configurations we will not present the combination of a preemptible kernel and an idle load because it is very similar to the normal kernel without load. However, we will present the measurements on an ARTiS kernel with load.

In the experiments, the system load consisted of busying the processors with 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, four types of program corresponding to four loading methods were used:

- **Computing load**: A task that executes an endless loop without any system call is pinned on each processor, simulating a computational task.
- **Input/output load**: The `iodisk` program reads and writes continuously on the disk.

Configurations		Kernel		User	
		99.999%	Maximum	99.999%	Maximum
standard Linux	idle	78μs	94μs	82μs	220μs
standard Linux	loaded	77μs	101μs	2.829ms	42ms
Linux with kernel preemption	loaded	76μs	101μs	457μs	47ms
ARTiS	loaded	71μs	101μs	91μs	120μs

TABLE 2: *Kernel/User latencies of the different configurations.*

- **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*.

Observed latencies Table 2 summarizes the measurements for the different tested configurations. Two values are associated with each latency type (kernel and user). "Maximum" corresponds to the highest latency noticed throughout 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.

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. The kernel latencies are nearly unaffected by the load. However, the user latencies are several orders higher. This is the typical problem with Linux, simply because it was not designed with real-time usage in mind.

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 457μs instead of 2.829ms. Unfortunately, the maximum value of these user-space latencies is very similar, in the order of 40ms. This enhancement permits soft real-time with better results than the standard kernel but in no way does it allow for hard real-time, for which even one latency over the threshold is unacceptable.

The results given by the real ARTiS implementation are significantly different with a maximum of 120μs for user latencies. This is much lower than the limit we have fixed of 300μs. The system can be considered as a hard real-time system, insuring real-time applications very low interrupt response.

6.2 Adeos Performance Evaluation

The relevant test for Adeos is the measurement of the interrupt latency, also named above kernel latency due to the fact that all activity is in the kernel perimeter. The test consists in a periodic activation of a hardware interrupt at 1kHz frequency. It has been run on two different architectures

x86 a bi-processor 2.4GHz Xeon.

IA-64 a quad-processor 1.3GHz IA-64. For UP measurements, the kernel has been instructed at boot time to manage only one processor.

The tests were run under the following load conditions:

- a parallelised kernel compilation running in loop ;

- a network interrupt flood (using the ping -f command from another machine)

	1kHz UP/IA-64	
Load	Max jitter	Avg jitter
idle	40 μs	13 μs
loaded	60 μs	17.5 μs

TABLE 3: *interrupt latency on UP/IA-64*

	1kHz SMP/IA-64 4x	
Load	Max jitter	Avg jitter
idle	60 μs	13.5 μs
loaded	70 μs	17.5 μs

TABLE 4: *interrupt latency on SMP/IA-64*

	1kHz SMP/x86 2x	
Load	Max jitter	Avg jitter
idle	54 μs	8 μs
loaded	86 μs	17.5 μs

TABLE 5: *interrupt latency on SMP/x86*

Comparing results from tables 3 and 4 show that degradation from SMP architecture is less on a loaded system than on an idle system.

Similarly, comparing results from tables 4 and 5 show that performance is not depending so much on processor capabilities but more on the memory and cache subsystem. We can infer that this hardware is of better quality on a high-end server such as the IA-64 4-ways than on a middle-range one.

Overall, the latency jitter provided by Adeos on SMP systems is similar the one observed on UP systems. And in any case (cf. table 2) it represents an improvement over Linux worst-case latencies.

7 Conclusion and perspectives

The comparison between mono-processor and multi-processor latencies confirms the initial expectations. The real-time performance is better in a mono-processor configuration. But, surprisingly, the difference with SMP configuration show that degradation caused by concurrent accesses to memory, system bus and shared resources is relatively low.

Each of the above techniques has already given very promising results in improving determinism of execution for multi-processor systems. By examining real-time issues under different viewpoints, namely scheduling and interrupt management, we obtained significant improvements over the Linux 2.6 "preemptible" kernel. We expect to have confirmation of these improvements at application level with minimal software changes. This work could pave the way for an extended, more flexible Linux kernel configuration, whose capabilities are set up in order to better fit application requirements, thus enhancing the pervasiveness of Linux in multi-processor systems.

References

[1] ITEA Hyades home page; `http://www.hyades-itea.org`.

[2] Friedrich Seifert, Hugo Kohmann SCI Socket, a Fast Socket Implementation over SCI `http://www.dolphin.com`.

[3] Karim Yaghmour. Adaptative Domain Environment for Operating Systems `http://www.opersys.com`.

[4] Philippe Gerum RTAI/fusion `http://www.rtai.org`.

[5] George Anzinger. High Resolution Timers Home Page; `http://high-res-timers.sourceforge.net`.

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

[7] Stephen Brosky. Symmetric multiprocessing and real-time in PowerMAX OS. White paper, Concurrent Computer Corporation, Fort Lauderdale, FL, 2002.

[8] 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.

[9] 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.

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

[11] 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.

[12] Éric Piel. Linux Temps-Réel en environnement HPC - Mise en place de fonctionnalités temps-réel sur des systèmes multi-processeurs IA-64 sous Linux Internship report, Bull, Grenoble, France, February 2003.

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