Integration of Real-Time Services in User-Space Linux *

Gilles Chanteperdrix and Alexis Berlemont
Openwide
Paris, France
{chanteperdrix,berlemont}@openwide.fr

Dominique Ragot and Philippe Kajfasz
Thales
Colombes, France
{dominique.ragot,philippe.kajfasz}@fr.thalesgroup.com

Abstract

Among solutions to allow sub-millisecond determinism for real-time Linux applications, the co-kernel approach is found in many products and is probably to most widely known - and used - technique. In the past few years, this approach evolved, at least in the RTAI project, towards a better integration of the co-kernel deterministic services to Linux:

- the co-kernel interfaces were brought to user-space, allowing real-time applications to run under memory protection;
- any thread was allowed to use alternatively the two interfaces, suppressing the need for splitting the application into a Linux-based and real-time parts and for communication facilities between these two parts.

In this paper, we describe a versatile software architecture which, in our view, is the next step towards the seamless integration of real-time to the Linux user-space. Based on the Adeos technology, it improves:

- the determinism of a real-time task calling Linux services, by preventing events of lower priority from causing any disturbance;
- the integration of the two application programming interfaces, by redirecting Linux system calls to their real-time counterpart.

We present the mechanisms used to achieve these goals and that are included in a deterministic intensive computing (DIC) domain: interrupt dispatch control, thread management, services integration, system call handling mechanism. The integration of these real-time services in the POSIX API is also discussed. Finally, performance results for different architectures are presented.

1 Introduction

Real-time multiprocessor systems have been used for a long time in a number of domains ranging from medical imaging to aeronautics and telecommunications. The most widespread approach is to use MPP systems with distributed memory and high-speed links between processors. The software consists in a RTOS for each processor with inter-processor communication libraries highly dependent on communication infrastructure. These architectures are highly efficient and offer the best deterministic capabilities. However, with the increasing part of software in these applications, it is very difficult to reuse designs optimized for one architecture to another, making software migration costs very high in order to benefit from hardware improvements.

On the other hand, SMP systems, well-suited for enterprise application, have not been used for real-time applications. From a software develop-

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

ment methodology perspective, these systems offer a dramatic improvement over MPP systems at the expense of overall performance. For real-time systems, our motivation is to determine how the SMP architecture may be managed by a GPOS in order to provide real-time capabilities. We have chosen to extend the popular Linux operation system with specific real-time features under the constraints that they must have already been proved in other contexts and that they could be adapted to SMP without requiring a fork from the Linux codebase.

2 HYADES Requirements analysis

2.1 Real-time task profiles

The HYADES project [1] aims at supporting two major real-time task profiles:

P1 Time-critical data acquisition tasks. Guaranteed low interrupt and dispatch latencies (i.e. as forming the preemption latency) are required for these high-priority tasks which are expected to communicate with specialized hardware in a timely manner. Since the timeliness is critical, it has been admitted that such tasks may not be under the permanent control of the Linux kernel when operating, so that scheduling decisions could always be made regardless of the regular Linux kernel state. This exemption however calls for a specialized API for programming such tasks, independent from the regular Linux API. Additionally, ad hoc communication paths between those time critical tasks and the rest of the Linux system would be created for fast and efficient data exchange.

P2 Deterministic Intensive Computing (DIC) tasks. Whilst less prioritary than acquisition tasks, these activities still require bounded latencies, reliable execution determinism, and a strict priority management, so that their allotted runtime quanta are never significantly perturbated by non real-time activities from any existing processor. Such tasks must run as regular Linux processes according to the FIFO scheduling policy, so that the regular Linux programming model is kept.

2.2 Structural issues

The real-time sub-system envisioned for the HYADES project has a special characteristic: it must coexists on the same hardware with the general purpose Linux kernel and applications. This fact raises a number of structural issues which go well beyond the basic needs for a real-time operating system; generally speaking, Linux kernel internal design promotes fairness and throughput, at the expense of determinism. Techniques used to cope efficiently between fairness and short latency mainly tend to implement an acceptable trade-off, according to results obtained from experimentation.

In the merely unfair approach like a real-time system should implement, the fairness attempt is a major issue with respect to determinism. The net effect is that nobody can predict that every code path from the vanilla 2.6 Linux kernel will be preemptible soon enough for serving interrupts and/or rescheduling tasks. This might be acceptable for jitter-tolerant real-time activities, but still, time-critical ones (like stream-based acquisition systems) could certainly not cope with such a risk. If Montavista's preemptible kernel extension - which went mainstream since 2.6 - partly solves the kernel granularity issue by limiting the non-preemptible sections to those protected by SMP spinlocks, no formal guarantee exists that some unidentified code path could not create unacceptable jitter for time critical tasks. The consequences of such uncertainty could be made even worse whenever a "foreign" code (i.e. un-audited by the HYADES project) is integrated into the Linux kernel, as the usual maintenance of the codebase goes.

Short bounded preemption latency and high execution determinism in kernel 2.6 cannot be guaranteed in a way HYADES real-time tasks can operate safely. Fine grained Linux kernel preemptibility now available with 2.6 does not help prioritizing the interrupt load from a real-time point of view. This means that low-priority interrupt post-processing handlers (i.e. bottom-halves) could still preempt high-priority time critical Linux tasks under this model, thus introducing unbounded latencies.

2.3 SMP issues

Like any SMP kernel architecture, Linux is not immune from high contention for shared resources. An application executing kernel services may be delayed while attempting to enter a critical section which is currently executed by another activity on a different processor. Under some circumstances, these delays caused by activities not even related to the real-time processing could unacceptably impede the performance of time-critical tasks. A significant example of this problem can be taken from the Virtual File System (VFS) implementation, particularly in the kernel support for the `ioctl()`service pro-

vided by most Linux drivers. The generic kernel code `sys_ioctl`)) holds a system-wide spinlock while executing the driver-provided `ioctl()`call upon request from a user-space task. The spinlock is held with interrupts enabled, which subsequently allows interrupts to preempt the `ioctl()`code. Since pending interrupt bottom-halves would run on the interrupt return path, the preempted code might be delayed for several hundreds of microseconds, and sometimes even induce millisecond-range delays, before the preempted task can resume after the bottom-halves have been executed (e.g. high networking pressure do produce such effects). Obviously, this behaviour would have the additional side-effect of delaying from the same amount of time any real-time task executing on a different processor trying to issue an `ioctl()`call. Unfortunately, `ioctl()`and other I/O support routines are the standard way for user-space tasks to communicate with I/O drivers in the Linux programming model, so this would be a significant problem for real-time tasks aimed at time-critical data acquisition. Whilst exacerbated, this example should not be considered as one of a kind. Even if the ioctl() lock could be locally dismissed under certain circumstances, there are various known locations in the kernel code where priority inversions could occur this way on other locks. Some lengthy non-preemptible locations are probably still unknown, not to speak of those which could appear as the mainline maintenance of the 2.6 codebase evolves. The induced jitter could probably remain acceptable with DIC tasks, but certainly not with data acquisition ones.

3 Adding real-time features to Linux

As a result of the above analysis, a plain vanilla Linux 2.6 kernel could not meet the requirements expressed by the HYADES project. However, different technical paths have already been experimented by third-parties to improve Linux's determinism, but at the time these lines are written, all of them are based on the older 2.4 architecture. This said, most if not all of these attempts are still compatible with (or even present in) the new 2.6 architecture, so analysing their pros and cons in this respect is still relevant.

3.1 Native kernel preemptability

The figure above illustrates how non-preemptible sections could cause priority inversions by preventing a high priority task from being scheduled for a long time. So, it is critical to reduce the amount of time spent in non preemptible sections by giving frequent opportunities for the kernel to reschedule its tasks. There are many ways to insert efficient scheduling points, kpreempt proposes a systemic solution, also know as "involuntary preemption": each time a thread unlocks a spinlock a scheduling procedure is launched. It is a simple and clever way to disseminate scheduling points in kernel code considering the number of spinlock handlings executed in Linux. The kpreempt solution has been natively added the kernel 2.6, thus compared to the 2.4 architecture, we obtain a finer granularity for the rescheduling procedure call.

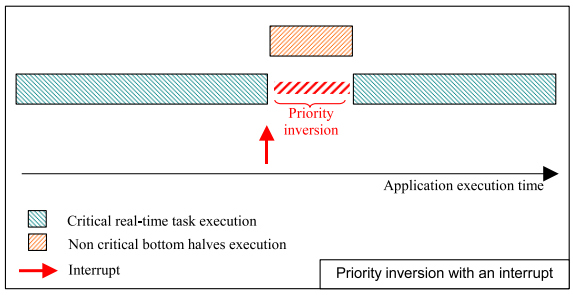

FIGURE 1: *Priority inversion*

Recently, a second approach to providing a finer granularity to the Linux kernel has been developed by RedHat's engineer and main Linux contributor Ingo Molnar known as the "voluntary preemption" scheme. Only available as development-level patches for the x86 platform as of now, this scheme adds explicit rescheduling points in the 2.6 kernel at places which have been otherwise designated as being potentially sleeping points of the code for debugging purposes. New patches are routinely issued to fix instabilities which have been introduced, or add new explicit preemption points to solve latency spots which have found experimentally. The voluntary preemption patches are mainly tested by the Linux audio development community right now. The main problem this solution faces comes from the never ending variety of code which could add lengthy non-preemptible sections of code, such as network drivers, not to speak of the network protocol stack itself.

3.2 Processor shielding

Concurrent Computer Corporation [6] has developed a Linux-based real-time system leveraging SMP hardware to reach sub-milliseconds guaranteed response time. This company has initially developed this technology during the past ten years on proprietary UNIX SVR4 systems, before porting it to Linux around 1999 and improving it since then. The shielded processor concept dedicates selected processors in a symmetric multi-processing system for real-time components of an application.

In their Linux variant called RedHawk, high-priority tasks and high-priority interrupts are bound to a set of dedicated processors, which are thus said "shielded" from negative interaction caused by non real-time activities occuring on other processors.

But sub-millisecond guarantee using the CPU shielding technique also implies locking out most of the interrupts on the shielded processor (i.e. to prevent bottom-halves from perturbating the timing), including the standard periodic timer. Unfortunately, doing so requires to specialize interrupts on a per-processor basis, so that real-time applications can still receive a selected set of interrupt sources in order to operate properly. Even if interrupt routing can be done dynamically at the programmable interrupt controller level to fulfill these needs when the application starts, the overall scheme remains mostly static by dedicating a pre-defined set of CPUs to the real-time duties, since interrupt programming cannot be a real-time operation. This induces two major flaws: first, CPUs dedicated to real-time are mostly under-utilized; and secondly, since a non real-time CPU cannot be converted to a real-time one on-the-fly as the current need for awakening a real-time application may require, the scalability of such scheme is rather problematic. Additionally, the CPU shielding technique does not address the following SMP-specific priority inversion problem caused by mutual exclusion constructs, when a non real-time activity running on a CPU blocks a real-time task running on another CPU for an unbounded amount of time, because it has been preempted by some IRQ handler(s) while holding a contended lock.

3.3 Threaded interrupt model and sleep mutex-based locking

TimeSys [10] has developed a Linux kernel variant implementing a new interrupt model and shared resource locking scheme to reduce the overall latencies. Basically, this approach boils down to having interrupt service routines managed as schedulable objects (i.e. kernel threads), and SMP spinlocks replaced by "sleep mutex" objects supporting priority inheritance.

This approach has at least two strong advantages over the native preemptible kernel. First, interrupt handlers have a scheduling priority just like any regular Linux process. This allows to assign critical tasks higher priorities than the interrupt handlers's themselves, instead of assigning these handlers an arbitrary high priority level, thus causing unwanted priority inversions. Second, the sleep mutex-based locking scheme dramatically improves the thread concurrency, since contention on entry of critical sections are handled on a per-lock level, instead of locking out the rescheduling procedure on a per-CPU level, like the vanilla 2.6 support for kernel preemption does. As often illustrated in the TimeSys litterature, vanilla kernel preemption management is like using a single traffic-light to control the traffic of an entire city since disabling preemption during any given critical section totally locks out rescheduling opportunities for other threads. On the other hand, sleep mutexes only stop threads which would otherwise enter the critical section they specifically protect. Coupled with priority inheritance management, sleep mutexes prevent priority inversions, at the expense of a careful use to prevent deadlocks.

However, Timesys's approach has one major drawback: it only addresses the preemptibility and concurrency issues on uniprocessor systems by carefully modifying the interrupt sub-system and the kernel locking scheme in ways that remain compatible with Linux's overall kernel design. Going beyond these changes to address SMP determinism issues would require dramatic modifications to the kernel, likely leading to a fork from the original codebase. Albeit a Linux variant should still be able to track the mainline updates, radical changes such as design updates would not allow it anymore. Since the 2.6 branch is quite young, it seems wiser to keep a direct feed available from the mainline kernel tree so that Linux community's fixes are available to the Hyades project at a reasonable integration cost, especially for the non-mainstream ia64 architecture.

3.4 Concurrent real-time kernel

The most well-known approach for adding hard real-time capabilities to Linux consists of embedding a dedicated scheduler aimed at managing time-critical tasks into the kernel. The RTLinux [7] and RTAI [8] projects are the major representatives of this co-kernel approach.

3.4.1 RTLinux

The implementation is two-fold: first, control of the Linux interrupt sub-system is taken over by the co-kernel using a technique known as the "software PIC". Basically, this technique allows the co-kernel to virtualize Linux's access to the processor-based interrupt mask, so that the co-kernel is always able to receive external interrupts, regardless of the perceived status of such mask on the Linux side. Secondly, a simplistic FIFO scheduler dispatches the real-time application tasks embodied in dynamically loadable kernel modules. Under this scheme, the Linux kernel in its entirety becomes the lowest pri-

ority pseudo-task, and gets scheduled when no real-time task is ready to run.

Since the Linux kernel code is fully interruptible by interrupts directed at the co-kernel, the preemption latency is close to the hardware limits. Latency figures lower than a handful of microseconds are commonly obtained with commodity x86 hardware. Ports of co-kernel to ARM, PPC, Etrax or MIPS cores already exists, thus demonstrating the adaptability of such approach to various hardware platforms. SMP x86 systems are also supported.

Real-time tasks are implemented using a specific API since the regular Linux kernel services cannot be invoked on behalf of their context. Since the normal Linux I/O infrastructure cannot be reused to develop drivers, the latter must be reimplemented using non-standardized ad hoc support, for each data source which needs to be accessed under real-time constraints.

Using such systems also requires a careful design for applications, in order to separate the real-time duty from the non real-time activities. The former will be embodied into kernel modules, the latter will live in user-space as regular Linux processes. It is therefore required to implement non real-time communication paths between both worlds, so that they can exchange information as needed. For instance, in a data acquisition application, the acquisition loop will go to the kernel, and the archiver and display processes will be implemented as regular Linux processes, receiving data end sending commands from/to the kernel side by means of shared memory or FIFO channels. Obviously, this particular dual-sided approach has a significant impact on the overall design of the final application.

3.4.2 RTAI

RTAI started from the same approach as RTLinux, but uses a different interrupt virtualization technique, based on the Adeos layer. On top of Adeos, a hardware abstraction level further insulates the generic RTOS code from the machine-dependent core.

In order to reduce the impact of real-time constraints on the usual application design, the RTAI project has developed a technology called "LXRT" allowing the real-time tasks to run in user-space, under the usual memory protection scheme available to regular Linux processes. The hard real time environment execution can not be the Linux domain so LXRT enables a mechanism of task migration. A common task can be stolen from the Linux scheduler and inserted in the RTAI LXRT scheduler. After managing memory isssues, the current task is able to run in the hard real time context provided by RTAI. Nonetheless, this process can only use RTAI syscalls: asking for Linux fonctionnalities automatically sends the task back to the Linux context.

This means that real-time tasks can almost be developed using the regular Linux programming model. However, these tasks cannot re-enter the Linux kernel while operating in hard real-time mode, losing their real-time property when invoking a regular Linux system call, until they finally exit the Linux kernel code.

A recent evolution of the RTAI technology, fusion, leverages the achievements of the LXRT scheme by re-implementing a compatible technology over a new nanokernel technology, which is also extended by a deeper integration between the regular Linux and hard real-time execution models.

3.4.3 Shortcomings of the strict co-kernel approach

Whilst sufficient to run simple to moderately complex acquisition and/or control systems, mere co-kernel solutions in kernel space have failed so far delivering a viable environment for running complex applications with real-time requirements spanning all over their implementation. Most of the time, the co-kernel approach requires that some trade-offs between determinism and feasibility are made for the time critical part to live in isolation into the Linux kernel space. This leads to constraints being arbitrarily relaxed for parts of the application that could only fit into the Linux process space (e.g. because of the availablity of some required operating system services), and usually suboptimal data communication paths must be set between the real-time kernel space handling the time critical work, and the regular Linux process mates running without strict real-time guarantees. This approach leads to highly constrained design choices aimed at setting the proper border line between critical and non-critical processings, so that one could then separate them "organically", with the additional burden of using two different programming models for their respective implementation (i.e. kernel programming for time-critical tasks and regular Linux process programming for soft-realtime and non-realtime tasks). Obviously, such intrusive re-engineering process is not something which could be undergone easily and at low cost with large pre-existing real-time application codebases such as the ones envisioned for the HYADES project.

4 Development methodology

The HYADES project is conducted using a three-fold approach with respect to development and testing:

- Development of the core functional parts (DIC) initially took place on SMP/x86 systems. Validation, testing and preliminary performances measurements have been conducted on x86 systems, so that further work on SMP/IPF platforms starts with a reasonably stable codebase.
- IPF-specific software needed to port the work previously achieved on SMP/x86 platforms have been initially written on HP's SKI ia64 instruction-set simulator. The preliminary validation phase of each component has also been conducted in this simulated environment, so that many ia64-specific bugs have been trapped and fixed more easily than on real hardware, thanks to the ability to reproduce them systematically. Adeos/ia64 and other architecture-dependent components of the HYADES system are among the software which has been initially developped using a simulation approach.
- According to our internal project milestones, simulation-validated components have been periodically tested on real SMP/IPF hardware, while the were progressively refined on SMP/x86. Recently, the HYADES project has eventually entered its global validation phase where the whole system is being tested, fixed and tuned on SMP/IPF.

5 HYADES execution model

In order to achieve a high degree of determinism while keeping the Linux programming model available to the real-time tasks in user-space, it has been decided to implement a hybrid approach, segregating the real-time constraints according to the requirements with respect to scheduling latency and execution determinism, thus defining distinct performance modes. Each performance mode provides its own set of services, but the transition of any real-time task between each mode is a transparent and time-bounded operation. At any point in time, real-time tasks in a HYADES system operate in one of the two following performance modes:

5.1 Micro-second, PRIMARY level performance mode

In this mode, a real-time task is guaranteed a very low scheduling latency, and it cannot be delayed by the regular operations of the Linux kernel in any case. Here, the worst-case scheduling jitter is typically bounded to a few tenths of micro-seconds, while still running in the memory management context of a user-space Linux process.

A real-time task executing in the primary performance mode has access to a set of extended HYADES-specific system calls which implementation can efficiently operate in such context. In order to ease the application programming, selected regular Linux services can even be impersonated by HYADES system calls, providing a high performance replacement of the former, while still keeping the standard POSIX call interface at the application level.

Any call to a regular Linux system call from this mode begets a transition to the secondary performance mode, so that the Linux kernel is properly re-entered to execute the requested service.

5.2 sub milli-second, SECONDARY level performance mode

In this mode, a real-time task is guaranteed a bounded scheduling latency only limited by the granularity of the Linux kernel internals, and it cannot be preempted by any regular kernel activity including Linux interrupt handlers, batched RCU work or other non-HYADES tasks. This performance mode is obtained by the combination of the preemptible Linux kernel features, which HYADES extends by additional protections against priority inversions.

A real-time task executing in the secondary performance mode has access to the full set of regular Linux services. This said, only the services having a complexity compatible with the real-time requirements of the applications should be used. For this reason, the preemptible kernel configuration option should be enabled for the Linux kernel embedding the HYADES RTOS, and the newest NPTL thread interface should be used, since it is far more efficient performance-wise than the older LinuxThreads implementation.

Any call to a extended HYADES system call from this mode begets a transition to the primary performance mode, which does not incur any delay.

5.3 Rationale

This two-fold execution model is motivated by the following issues:

The Linux kernel essentially follows a GPOS design, and as such, many portions of its implementation favour throughput over preemptibility. The

implementation of the I/O, network, virtual memory management and interrupt sub-systems to name a few illustrate this clear choice, where interrupt processing and context switches could be delayed by an unbounded amount of time in order to bring the current operation to completion first, regardless of any priority considerations. Because the latter option is strictly incompatible with the real-time duty of the HYADES RTOS, simply relying on a modification of the Linux scheduling algorithm falls short of bringing the deterministic behaviour we need; the issue of obtaining the proper kernel granularity with respect to the real-time requirements of the HYADES applications must be addressed instead. The primary performance mode (i.e. micro-second level) has been designed to fullfil the most stringent requirements of real-time tasks usually performing closed loop processing using a minimalistic set of system services.

Augmenting the preemptibility by adding involuntary or voluntary preemption points in the kernel in order to reduce the average time between two rescheduling points is a fundamental improvement of the Linux 2.6 series over the older 2.2 and vanilla 2.4 series. This said, it is usually admitted by the main Linux developers that some non-preemptible execution paths must remain in the standard kernel, either because introducing some form of conditional rescheduling there would go against the throughput of the global system, or because this might even cause unstability. Additionally, and maybe most importantly, these optimisations are to be considered in the average case only, which is not sufficient to grant the worst case guarantee real-time performance requires. The secondary performance mode (i.e. sub milli-second) has been designed to leverage the continuous improvement of the Linux kernel granularity and the richness of its native API, while removing the known causes of execution jitter in a way which is not currently addressed by the mainline development branch.

Because HYADES is targeted to medium to large IPF SMP systems, issues are raised by the increased contention of resources between CPUs. Among them, priority inversions due to contentions on common resources between real-time and non real-time activities running on different CPUs must be addressed, because they are known to be the major source of loss of determinism. Additionally, the specialisation of CPUs among the real-time and non real-time duties must not prevent the HYADES system from being scalable. In other words, any CPU which is busy with non real-time duties in the system should be able to switch to real-time activities as the real-time workload requires it, in a minimal and always bounded amount of time. To this end, the primary and secondary performance modes can be mixed and inter-operate in order to cope with the perturbations induced by the regular GPOS activity of the Linux kernel on the HYADES real-time tasks.

A pillar of the HYADES RTOS proposal is its ability to ease the migration of existing real-time applications over the real-time enabled Linux kernel. To this end, the preservation of the Linux programming model in user-space while keeping a high degree of determinism justifies the matter of segregating distinct levels of real-time requirements, so that the most adapted execution model is always used for the real-time tasks. The HYADES RTOS ensures that those levels are seamlessly integrated to the programming environment, and consistent with respect to the application interface.

To sum up the above points: transforming the Linux kernel in a pure RTOS implementation would lead to anything but Linux, likely losing the ABI compatibility or parts of the programming model in the same move, which would lead to the inability of recycling the existing Linux applications and kernel components like drivers. The evolution of the Linux kernel regarding fine-grain granularity support in the past years also showed that only fixing latency spots as soon as they are discovered experimentally in the vanilla kernel code - provided they could be fixed in the first place - just lowers the average interrupt and scheduling latencies, not the worst case figures we are interested by.

6 Components of the Hyades real-time/SMP system

Having the Linux GPOS and an efficient RTOS extension to coexist on the same SMP machine requires to solve the following issues:

- Worst case interrupt latency must be close to the hardware limit, so that the real-time events can be processed in a timely fashion without encurring unbounded delays, regardless of the current activity of the GPOS. For this reason, *interrupts must be prioritized*, so that the RTOS is granted an absolute priority for handling them before the GPOS kernel. The Adeos nanokernel is used to provide this feature.

- Due to the differences of the various IPF implementations, the machine-dependent services used to implement some real-time controls not provided by Adeos should go to a *hardware abstraction layer* (HAL), hiding the architecture-specific details under a normalized low-level programming interface.

- Since we aim at keeping the regular user-space programming model and API for the real-time applications, a mean must be found to prevent deterministic processing tasks running under Linux control from being preempted by asynchronous Linux activities such as interrupt handlers, software interrupts (e.g. bottom halves), or RCU callbacks which could induce unbounded execution latencies. This is especially important for SMP configurations, since such preemptions could cause priority inversions, would a resource contention arise between an interrupted processor holding a critical section while running a non-prioritary task, and another processor running a prioritary task which requires to enter the same critical section. An *interrupt shield* is required between Linux and lower-end components.

- Real-time services which cannot be provided by the regular Linux API must be made available from an additional system component aimed at supporting *Deterministic Intensive Computing* (DIC) applications. Additionally, some existing Linux services which could not be used in time-critical applications - mainly for performance reasons and/or lack of determinism - need to be seamlessly re-implemented by deterministic HYADES counterparts. An illustration of such service is the standard nanosleep() feature, whose timing precision depends on the current value of the periodical system tick. Since Linux/IPF uses a periodical time source rated at 1024Hz, the nanosleep() service cannot be used by HYADES applications demanding more accurate timings.

- Real-time events such as interrupts directed at HYADES real-time tasks must trigger the immediate rescheduling of those tasks, regardless of the current internal state of the Linux kernel. At that point, the real-time tasks have the opportunity either to run HYADES extended system calls or internal computations with no delay, or issue regular Linux system calls. In the latter case, the real-time tasks are migrated transparently under the control of the regular Linux scheduler and run as high-priority tasks, but still non-preemptible by any regular kernel activity including Linux interrupt handlers, batched work or other tasks.

Providing a seamless integration of the RTOS services into the original Linux ABI additionally requires to be able to intercept and redirect system calls issued by the regular applications. The resulting architecture for Hyades is shown in the figure below:

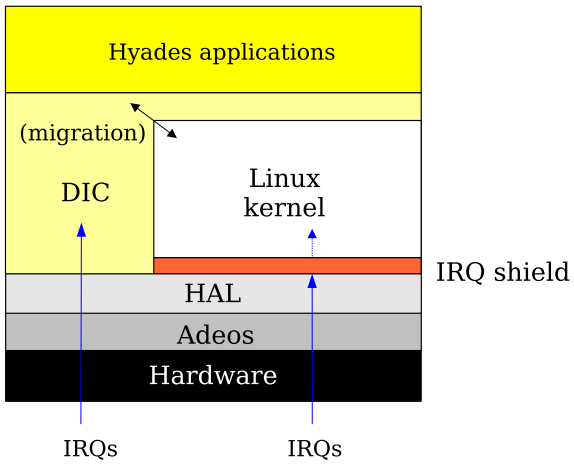

FIGURE 2: *The HYADES Architecture*

Since the Adeos nanokernel [5] is a mature Free Software technology offering all of these core facilities currently to x86, PowerPC and ARM platforms, it has been decided to port it to the IPF architecture [2]. Additionally, the porting experience gathered by the RTAI project [9] which already uses Adeos as its base technology should be of direct interest to the HYADES project.

6.1 the Adeos nanokernel

The purpose of Adeos is to provide a flexible environment for sharing hardware resources among multiple operating systems, or among multiple instances of a single OS. To this end, Adeos enables multiple kernel components called domains to exist simultaneously on the same hardware. None of these domains necessarily see each other, but all of them see Adeos. A domain could be a complete OS, but there is no assumption being made regarding the sophistication of what's in a domain. In its current development stage, Adeos allows to share hardware interrupts and system-originated events like traps and faults with the Linux kernel.

Adeos allows to control of the flow of hardware interrupts deterministically without any interference of the Linux kernel. Such control includes intercepting, masking and prioritizing thoses interrupts. Absolute preemptibility of any section of the Linux kernel code by interrupts, regardless of the perceived interrupt mask, is obtained by a common technique used in co-kernels based on the Optimistic Interrupt Protection scheme [3]. Adeos provides a facility to intercept Linux system calls, which in turn would allow to extend the regular programming API with dedicated HYADES system calls.

Generally speaking, Adeos forces Linux to share critical system resources with Adeos domains available as modules or statically linked into the kernel, according to a given priority scheme (e.g. such code might need to handle processor interrupts and being notified of kernel syscalls issued by any application before the Linux kernel code handles them).

6.1.1 Adeos fundamentals

The fundamental Adeos structure is the chain of client domains asking for event control. A domain is a Linux kernel-based software component which can ask the Adeos layer to be notified of:

- Every incoming hardware interrupt.
- Every system call issued by Linux applications.
- Other system events triggered by the Linux kernel code.

Adeos ensures that events are dispatched in an orderly manner to the various client domains, so it is possible to provide for determinism. This is achieved by assigning each domain a static priority.

This priority value strictly defines the delivery order of events to the domains. All active domains are queued according to their respective priority, forming the "pipeline" abstraction used by Adeos to make the events flow, from the most to the less prioritary domain. Incoming events (including interrupts) are pushed to the head of the pipeline (i.e. to the most prioritary domain) and progress down to its tail (i.e. to the less prioritary domain).

In order to defer the interrupts dispatching so that each domain has its own interrupt log which gets eventually played in a timely manner, Adeos implements the "Optimistic interrupt protection" scheme as described by [4].

6.1.2 Interrupt propagation

When a domain has finished processing all the pending interrupt it has received, it calls a special Adeos service which yields the CPU to the next domain down the pipeline, so the latter can process in turn the pending events it has been notified of, and this cycle continues down to the less prioritary domain of the pipeline. The stage of the pipeline occupied by any given domain can be "stalled", which means that the next incoming hardware interrupts will not be delivered to the domain's handler(s), and will be prevented from flowing down to the less prioritary domain(s) in the same move. While a stage is stalled, interrupts accumulate in the stalled domain's log, and eventually get played when this stage of the pipeline is unstalled. Adeos has two basic propagation modes for interrupts through the pipeline, which are defined on a per-domain, per-interrupt basis.

In the implicit mode, any incoming interrupt is automatically marked as pending by Adeos into each and every receiving domain's log accepting the interrupt source. In the explicit mode, an interrupt must be propagated "manually" if needed by the interrupt handler to the neighbour domain down the pipeline.

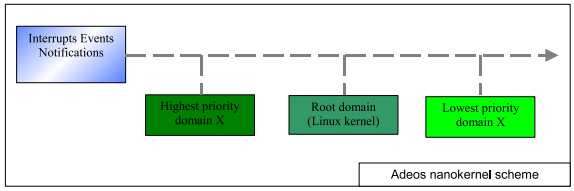

FIGURE 3: *Adeos domains*

As illustrated above, the interrupts flow down the pipeline from the highest prioritary domain to the lowest one. The Linux kernel is itself encompassed into an Adeos domain (i.e. the Root domain) as part of the initialization chores of an Adeos-enabled Linux system.

On a SMP system, each processor receives interrupts and generates exceptions. Adeos has to set up per-cpu stages inside the pipeline and instantiate one domain thread per cpu in order to process the events flowing through them.

6.2 The real-time HAL

Even if Adeos already provides an architecture-independent interface to the low-level resource management services, HYADES still needs an intermediate layer exporting strictly real-time oriented basic services. To this end, some architecture-dependent code is provided, aside of using the native Adeos capabilities. The extended programming of the IPF hardware timer source and its calibration are among the features provided by such layer.

The real-time HAL is provided as a removable kernel module which builds most of its services upon the Adeos nanokernel interface, and partly by providing an ad hoc implementation of real-time oriented services targeted to the IPF architecture. The HAL is a building block for the DIC domain.

6.3 The interrupt shield

Deterministic control of the execution latency which must remain under acceptable bounds is a key part of this proposal. Because integrating a RTOS domain into a GPOS environment definitely creates runtime perturbations in both, we need to make sure that real-time activities are not delayed by regular Linux

ones. Since we chose to keep the Linux programming model in user-space available for the HYADES applications, relying on the mere isolation of the RTOS domain in supervisor/kernel space to secure this property - a la RTLinux for instance - is not an option. However, keeping the real-time activities into the Linux realm opens a broad range of priority conflicts raised by non real-time asynchronous activities which could preempt the real-time tasks. Execution of interrupt bottom-halves and scheduled batches of work (e.g. RCU, work queues) can be triggered from a fair number of places in the Linux kernel, including from some critical places such as before returning control to the application after a system call has completed or an interrupt has been processed.

A typical priority inversion which occurs when a non real-time Linux task is preempted by some asynchronous Linux activities while it holds an interprocessor lock contended by a real-time Linux task. In such a case, the execution determinism of the real-time task is impacted by the duration of the asynchronous code, which delays the release of the interprocessor lock, thus blocking the execution of the real-time task for the same amount of time. Since inter-processor locks are almost everywhere in the Linux kernel, the odds to encounter such kind of priority conflict are high if the real-time tasks are allowed to run regular Linux services. Since we want the broader set of Linux services to remain available to the real-time tasks, it is just not relevant to switch those tasks out of the real-time realm when they enter a contention point: we must allow these tasks to go through the critical section as fast as possible without incurring priority inversion due to unwanted preemption.

To this end, the HYADES RTOS uses the underlying Adeos support to build a software barrier which shields the entire Linux kernel from non real-time interrupts when real-time activities are running under its control.

6.4 The DIC domain

The core of the HYADES real-time system is implemented in an Adeos domain called DIC (i.e. Deterministic Interrupt Computing), embodied in a regular module inside the Linux kernel. This domain has four major responsabilities :

Promoting regular Linux tasks to high-priority HYADES DIC tasks. Providing support for very high-resolution timers to the DIC tasks, with 50 us worst-case precision. This support does not rely on the native high-resolution timer support for 2.6 which lacks determinism and even stability for SMP systems. Operating the interrupt shield to protect the running DIC tasks from unwanted preemption. Ensuring the immediate rescheduling of the DIC tasks upon receipt of a real-time event.

To this end, the DIC domain has a higher priority than Linux in the Adeos pipeline, so that external interrupts and system calls which need to be handled in real-time mode have absolute priority over regular Linux activities. Those real-time events are then immediately handled by the DIC domain directly. The core part of the DIC domain in charge of the most critical operations is called the controller. It implements an autonomous scheduling system able to react to the real-time events aside of the regular Linux machinery. The controller is able to schedule regular Linux tasks in hard real-time mode, and ensure the proper synchronization between its own services and those of the Linux kernel. The DIC controller has been adapted from the *fusion* technology available within the RTAI project. The RTAI/fusion technology, originally aimed at porting traditional RTOS interfaces from the embedded world over the RTAI system, has been converted as follows:

- Architecture-dependent port to IPF.
- SMP support.
- Adaptation from Linux kernel 2.4 to 2.6.

7 Implementation of the DIC controller

7.1 Execution modes

Since it is based on RTAI/fusion's core implementation, the DIC controller implements the primary and secondary operation modes as follows:

- The primary (or hardened) mode guarantees very low latencies by yielding control of the real-time task to the co-scheduler whose operations cannot be delayed by any regular Linux activity, including interrupt masking. In this mode, the task appears to the kernel as being aslept in TASK_INTERRUPTIBLE state, but it actually runs within its original MMU context under the control of the co-scheduler. The task enters this mode after initialization, and each time it calls a blocking native HYADES services. It leaves it to enter the secondary mode only when it issues a regular Linux syscall. Typical worst-case scheduling latencies in this mode are currently about 50 μs under high load on mid range x86 hardware.

- The secondary (or relaxed) mode still guarantees low latencies but with higher worst-case

figures though, depending on the underlying granularity of the Linux kernel which controls it. In this mode, the real-time shadow thread is suspended at the co-scheduler level, so that only Linux is allowed to alter the flow of control of the mapped task. This mode additionally defers interrupts to be processed by Linux as long as the real-time task is running. The latter technique greatly improves predictability with respect to execution time needed by CPU-bound tasks running in secondary mode, without impacting the interrupt latency for other tasks operating in primary mode.

Linux tasks controlled by the DIC controller are transparently and automatically switched between the two operating modes during their lifetime, according to the level of real-time service they ask for, either strict HYADES (primary) or shielded Linux (secondary). Real-time priorities are consistently kept across those migrations, so that a task issuing Linux syscalls could still have a higher priority than the ones managed by the co-scheduler.

7.2 Cooperating schedulers

The HYADES co-scheduler which manages the Linux tasks when operating in primary mode is implemented as a loadable kernel module. It defines a thread abstraction called a "real-time shadow" which is mapped to pre-existing Linux task contexts (i.e. task_struct objects). Real-time shadows are the co-scheduler's basic schedulable objects, representing their respective Linux task counterparts when operating in primary mode. In this respect, shadows do not have their own stack or set of registers, but share those of the original Linux task, since both schedulers operate in a mutually exclusive manner.

Shadow priorities are directly inherited from the mapped Linux tasks, and dynamically tracked upon change. Since Linux tasks must belong to the SCHED_FIFO scheduling policy in order to have a shadow counterpart, the priority range of real-time HYADES tasks is [1..99]. This way, consistency is kept across operation modes with respect to priorities, and the HYADES scheduler complies with the original Linux priority scheme.

The above principles allow both Linux and HYADES schedulers to cooperate fully, separately handling any given real-time task context according to the current operating mode.

7.3 Use of Adeos domains

The HYADES system is built over the Adeos layer for prioritizing hardware interrupt processing, and further implementing the means of cooperation between the DIC controller and the Linux kernel.

Three Adeos domains are defined in the kernel environment running a HYADES-enabled system:

- The DIC controller domain is in charge of processing the interrupt flow and system calls directed at the activities running in primary mode. Since it is at the highest position in the Adeos pipeline, those events cannot be masked by lower domains, and are thus immediately processed by the HYADES co-scheduler that lives in this domain. As a result of receiving a prioritary interrupt, the co-scheduler may preempt any less prioritary activity of the standard kernel to resume a hardened Linux task.

- The interrupt shielding domain is next down the Adeos pipeline, and provides a mean for deferring the normal interrupt flow directed at the standard kernel while Linux tasks are running in secondary mode. See below.

- The root or Linux domain is last in the pipeline; it is the place in the Adeos scheme where the standard kernel lives. It receives interrupts and system calls which have been propagated by upper domains, after they have been eventually processed by more prioritary activities.

7.4 Interrupt shielding

A key requirement with respect to highly deterministic computing is predictability of the execution time once CPU-bounds task. Since such predictability could be jeopardized by asynchronous Linux operations, like interrupt handling (e.g. top-halves and bottom-halves/softirqs, RCU batches), a mean has to be found for deferring such activities until the stringent real-time processing has ended.

To this end, the DIC controller engages a specialized "shield" blocking the interrupt flow before it reaches the Linux kernel each time a real-time task operating in secondary mode is scheduled. This shield is obtained using a specific Adeos domain which lives between the DIC domain and the Linux one. The interrupt flow is locked at this stage when the shield is engaged, and unstalled when disengaged.

Additionally, the interrupt shielding prevents priority inversions caused by mutual exclusion constructs on SMP configurations, e.g. when a non real-time activity running on a CPU blocks a real-time task running on another CPU for an unbounded amount of time, because it has been preempted by some IRQ handler(s) while holding a contended lock.

7.5 System call impersonation

Via dynamic interception using the proper Adeos support, i.e. adeos_catch_event(), some of the regular Linux system calls can be impersonated by pure HYADES counterparts, when doing so provides a real benefit with respect to enforcing determinism. For instance, the regular nanosleep call is wired to HYADES's high-precision timer for sub-HZ accuracy and very low scheduling latency. The task keeps running in (or possibly switches to) primary mode when such substituted services are called.

Impersonation is made at the DIC controller domain level, by a special handler filtering all regular Linux system calls before they are processed by the standard kernel. This handler further decides if a substitution with a HYADES-specific implementation must take place and applies it, or simply propagate the call to the root domain for execution by the standard kernel.

8 Performance measurements

8.1 Tests used

latency The latency test measures the latency of the HYADES aperiodic timer set at 10kHz frequency.

cruncher The cruncher test measures the execution jitter of a computation intensive loop running with or without the HYADES environment. Number crunching executes for a duration of 11 ms and then sleeps for 500 μs, in a loop.

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)

8.2 Results

The tests were run on a quad 2.4 GHz Xeon, the cruncher thread being run with and without the HYADES environment.

From figure 4 we can see that the worst-case jitter is below 60 μs, which is good in absolute timings compared to what Linux provides in user-space under heavy load [11] but barely sufficient compared to the timer frequency. The bimodal shape of the latency measurements is very noticeable and is not yet explained. The limits of what an SMP architecture can do for real-time in user-space are reached.

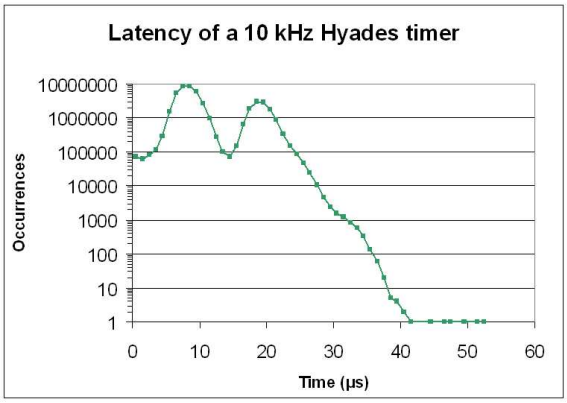

FIGURE 4: *Latency of a 10 kHz HYADES timer*

From figure 5 we can see that the Hyades system outperforms Linux by roughly an order of magnitude. As expected, the Hyades system is not subject to perturbations occurring from Linux activity, even when the system is left free to dispatch tasks to any processor available on the system. The shield is working very efficiently, even on a multiprocessor system.

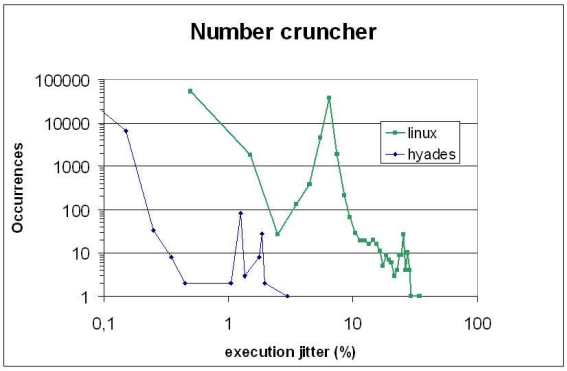

FIGURE 5: *Execution jitter*

9 Conclusion

It is possible for applications having both real-time requirements and heavy number crunching computations to be executed on SMP systems with very good overall performance with respects to these constraints. For systems where determinism is an important requirement, although non-critical, this solution offers a good alternative to high-end solutions using dedicated RTOS. Software development and maintenance is facilitated by the effective reuse capabilities for the real-time parts of the applications and by the tremendous integration facilities provided by the Linux GPOS over any RTOS. It results from this that systems developed using this approach could benefit

from the mass market of computer servers and providing to their customer a better efficiency in terms of performance/price ratio.

References

[1] *http://www.hyades-itea.org*

[2] D. Mosberger, S. Eranian, 2002, *IA-64 Linux Kernel*, Prentice Hall PTR, ISBN 0-13-061014-3.

[3] Stodolsky, Chen, and Bershad, 1993, *Fast Interrupt Priority Management in Operating System Kernels*, USENIX

[4] Stodolsky, Chen, and Bershad, *http://citeseer.nj.nec.com/stodolsky93fast.html*

[5] *http://www.adeos.org/*

[6] *http://www.ccur.com/*

[7] *http://www.fsmlabs.com*

[8] *http://www.rtai.org*

[9] *http://www.aero.polimi.it/rtai/*

[10] *http://www.timesys.com/*

[11] D.Ragot & al., *Linux for High Performance and Real-Time Computing on SMP Systems*, REAL-TIME LINUX WORKSHOP 2004