ENEA

White Paper

Enabling Real-Time in Linux

By Patrik Strömblad, Enea

Linux has a strong momentum in the embedded software industry and has in the past years become the prevalent choice as operating system for new platforms. However, standard Linux is designed for general throughput processing rather than for real-time, and consequently it needs to be modified or complemented to meet high demands on interrupt latency, determinism and low-overhead multitasking. In parallel, advancements and trends within the semiconductor industry are driving an evolution towards multi- and manycore devices. This scaling to more cores challenges Linux in terms of its capability to provide a cost-efficient and real-time capable OS platform, and therefore we need to constantly evaluate our alternatives to enable real-time acceleration in Linux.

The embedded OS application domain has historically been dominated by classic home-made or proprietary real-time operating systems on embedded solutions. Nowadays Linux is ported to most multicore SoC:s by the HW vendors, and in most use cases where hard real-time, safety or low footprint is not required, Linux can replace an existing RTOS. Over the past years Linux has also evolved to better meet moderate real-time requirements, but there is still a magnitude of difference to what an RTOS can provide. In this paper we describe some fundamentals of a real-time system and why real-time capabilities are hard to achieve with the standard Linux kernel, followed by an analysis of different approaches for improving real-time behavior in a Linux OS solution.

Introduction

Trends in semiconductor technology evolution are constantly enabling processors with an increasing number of cores. The workload on a multicore processor will most likely become even more multiprogrammed to support a mixture of best effort, high throughput O&M and control applications, together with low-level packet processing or control loop applications with potentially high real-time requirements. Such systems must support applications with different requirements for throughput and determinism at the same time on the same processor device, which is hard to achieve since those characteristics are usually contradictive. What is equally important is also to be able to provide usable tools for performance analysis such as non-intrusive event tracing, execution profiling, resource monitoring, etc. Many "custom" solutions often end up in a system where the critical real-time application becomes a "black box", i.e. it becomes not possible to observe at all. The initial perceived simplicity will, as the software complexity grows, transform into higher development and maintenance cost over time.

Linux is today the preferred operating system for embedded systems, and it will always be chosen when possible from a characteristics perspective. Since standard Linux usually does not meet real-time requirements, the Linux kernel must be modified and/or complemented to gain real-time capabilities if the system has real-time requirements.

Adding real-time capabilities to embedded Linux has until recent years been a question of how to improve the deterministic behavior on single-core processors. Now, when the multicore technology starts to dominate also the embedded Linux market, new opportunities emerge to make use of multicore technology to simply avoid the resource conflicts that can occur in the existing Linux kernel design. Given the fact that most new processors will have multiple cores, the Linux real-time research should focus on the best ways to enable scalable multicore solutions for real-time applications by using some kind of *vertical partitioning*.

The most famous and common alternative approach to improve real-time in Linux has been to mitigate effects of potential resource conflicts in the kernel by applying the *kernel preemption patch*. This method is not using any kind of vertical partitioning principle, it more resembles the "paravirtualizing" principle.

However, on multicore architectures you can also try to vertically "isolate" one or several cores for real-time processing and by this avoiding non-deterministic side effects caused by the Linux kernel. There are fundamentally two ways to implement such a core isolation:

- *Linux user space partitioning*, where we use various features in Linux to isolate a core, or a core cluster, from load balancing, ticks, etc, so that we arrive as near a "bare-metal" thread as possible.
- *Dual-OS partitioning*, where Linux runs as Main OS on a set of cores, but where a number of cores are "unplugged" from Linux scheduler and instead run an SMP real-time executive based "RT accelerator".

This white paper describes and compares the three mentioned alternatives and provides a brief evaluation of functionality and grade of real-time capability for each alternative.

Definitions and Fundamentals

Before looking into the different real-time Linux solution alternatives, let us start with some background defining the key characteristics of a real-time system and which role the operating system has in such a system.

What is Real-Time?

The term "Real-time" is often and incorrectly interpreted as a synonym to "high performance". Real-time systems might be high-performing, but strictly speaking, real-time characteristics are expressed in other terms than performance. The performance of a system depends on the system design and obvious factors such as CPU clock frequency, cache and memory bandwidth, and the OS API overhead.

Response time is the time elapsed between an external event and the system response to the event. Providing a real-time response is the same as having a *deterministic response time*.

Real-time systems have well-defined operational deadlines from event to system response, with strict requirements on maximum and average response time. Usually a real-time system requires a response time of microseconds to milliseconds. A system may though still be a real-time one also when the allowed response time is in the range of seconds or higher, as long as the maximum response time can be granted.

Non-real-time systems cannot guarantee the response time in any situation and are often optimized for high throughput performance with best effort.

Hard, Firm, or Soft Real-Time

Real-time systems are often classified as hard, firm or soft. Missing a deadline in a *hard real-time system* means total system failure, and the impact can be critical. *Firm real-time systems* can tolerate infrequent misses although QoS deteriorates quickly when misses occur. More forgiving are *soft real-time systems* where misses slowly degrade QoS, but the system is still possible to use.

Typical hard real-time applications are pacemakers, process and engine controllers, robot control, and anti-lock brakes. 4G/5G Baseband processing and signaling in mobile network equipment and wireless modems are examples of firm real-time applications. Less time-critical processing like IP network control signaling, network servers, and live audio-video systems belong to the family of soft real-time applications.

The weakest part of the system determines the classification. In a hard real-time system, the response time for all possible combinations of code paths in the OS and application must be verified to prove that the system meets all worst-case deadlines. If not all cases can be verified due to the size and complexity of the OS kernel code or the number of possible combinations being too many, the system cannot be classified as a hard real-time system. Due to the clear definition of hard and soft real-time requirements, it is most common to distinguish between hard and soft real-time.

Latency and Jitter

When discussing real-time requirements in an operating system, the term *latency* is frequently used. There are however several definitions of latency, therefore we need to define how latency is used in this paper.

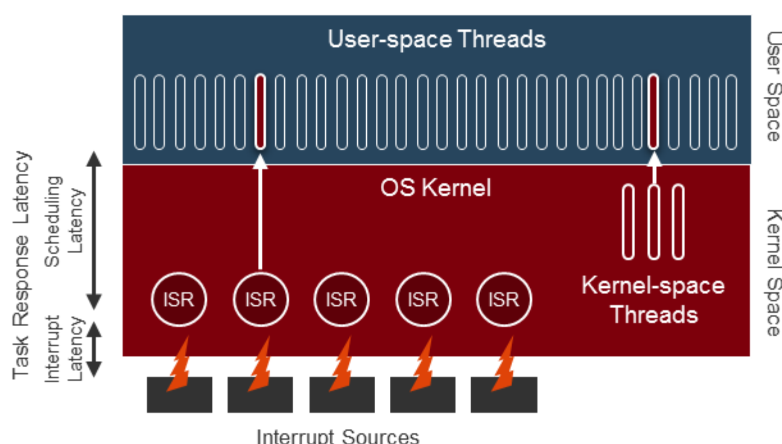

Figure 1. Task response latency from interrupt to user space

Interrupt latency – The time elapsed from the signaling of an interrupt request on hardware level until the kernel-level interrupt service routine (ISR) starts to execute. In Linux, the interrupt handling of an external event is usually split into one initial part (the kernel-level ISR triggered by the interrupt), and a second part referred for later execution (using the tasklet or work queue concept, or as a thread in kernel or user space as in RT-patch).

Scheduling latency – The time it takes from when a piece of driver code via some kernel API signals a thread (usually a user-space thread) to be scheduled, until the kernel scheduler resumes the actual thread, for example the returning of a blocking call on a device file. A number of sources can trigger the scheduling of a thread, for example an ISR routine in the kernel or another application thread.

Task response latency – The interrupt latency plus the scheduling latency. In most cases when “interrupt latency” is nowadays referred, we mean latency to a user thread (thus including the scheduling latency).

Jitter - The difference between minimum and maximum task latency.

Real-Time in Operating Systems

In a real-time system, the characteristic behavior of the operating system is very important. To start with, a *deterministic response* time from an external event until the application is invoked is what we normally refer to when talking about real-time characteristics of an operating system. However, since a chain is not stronger than its weakest link, it is also important to provide a deterministic runtime environment for the entire application execution flow so that it can respond within the specified operational deadline on system level. This implies that also the task scheduler and the resource handling API in the OS must behave in a deterministic way. The Operating System may preempt the multithreaded execution path in order to execute other OS services or some jitter is introduced due to asynchronous events before the output can be generated, and this defines a longer worst-case task response latency.

Naturally, the latency, OS jitter and performance throughput is also significantly affected by the characteristics of the caches, their size and topology, as well as memory bandwidth. This is however subject for another discussion.

When designing a system for high throughput performance, the goal is to keep down the average latency and OS overhead, while designing a real-time system aims to keep the

worst case latency under a specified limit. As a consequence, the design of a both high-performing and real-time capable system must take both average and maximum latency into account, as well as OS overhead.

Historically, the interrupt latency without scheduling, as defined above, is often what people refer to when talking about "latency" since leaving out the scheduling part presents the best figures. Those simplified figures are however becoming obsolete. Drivers are today usually implemented as POSIX applications running in user-space threads as they are easier to debug, maintain, and isolate from licensing issues compared to code in kernel space. The most important measure is thus the task response latency, which includes the scheduling latency. Note that it is becoming usual to refer to task response latency when talking about interrupt latency, so make sure that any latency discussion considers also the scheduling.

One of the most important requirements for a real-time capable operating system is a deterministic task response latency, where the scheduling delay is included in the latency definition.

Latency and Jitter in the Linux Kernel

To understand where a Linux system may suffer from a non-deterministic task response latency, we must first understand the kernel activities involved in the task response sequence.

The sequence above (figure 2.) illustrates a number of places where latency jitter (variance) due to resource conflicts may be added to the total amount of time spent inside the kernel. Some examples:

1. The external event triggers the generation of an asynchronous hardware exception in the CPU.
    - The exception can be delayed as interrupts may be disabled.
2. The kernel invokes the “top half” ISR whose task is to make either the user-space thread or a kernel “bottom half” thread ready.
    - The ISR can be delayed due to low-level kernel locks in the execution sequence from the ISR prologue to its epilogue when exiting.
3. After exit from interrupt level, the kernel enters the scheduler.
    - After returning from IRQ, there are several potential sources of jitter, e.g. pending interrupt or kernel threads on higher priority, RCU callbacks, or other threads holding resources needed for de-blocking/signaling. Most of the jitter usually derives from this step.
4. Other activities in the Linux kernel may also interrupt and preempt the threads involved in trying to respond to an event such as ticks, kernel services, or paging (if for example not all critical memory is locked in ram by mlock).

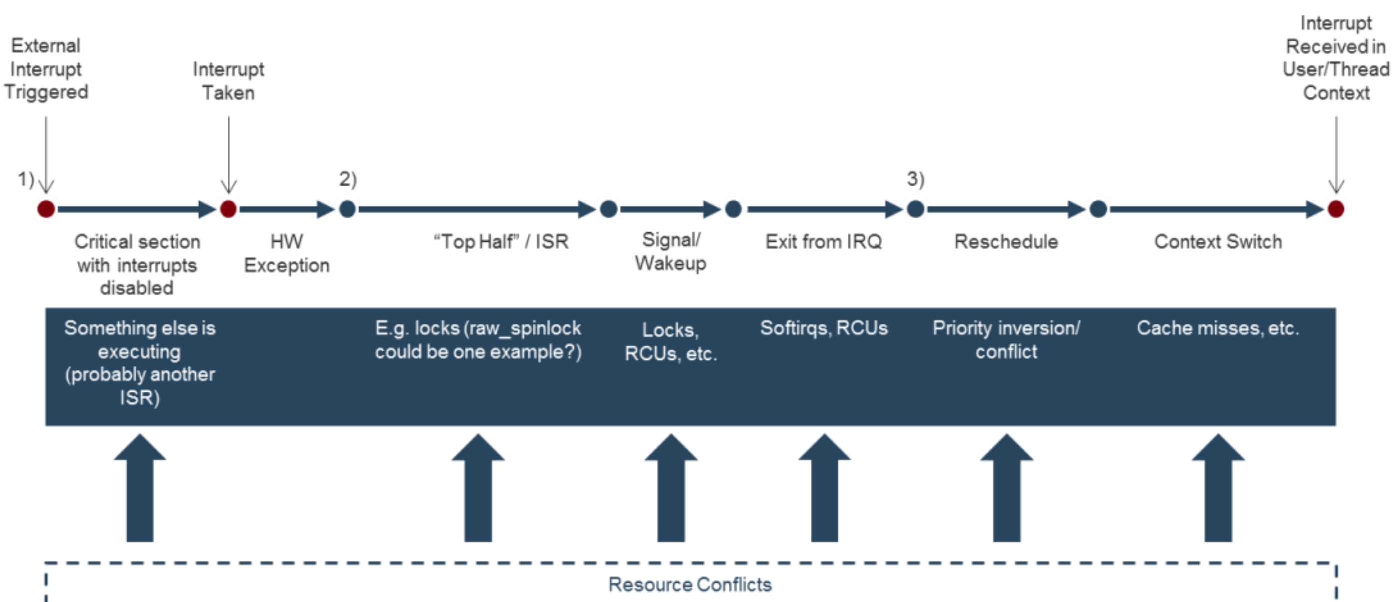

Figure 2. The Linux kernel contributions to latency and jitter

Most of the potential jitter in the sequence usually derives from the effects of resource conflicts in the scheduler and different priority-inversion scenarios. The research for making the Linux kernel real-time capable aims to either mitigate or completely avoid the effects from the resource conflicts in the kernel.

With this knowledge on how the Linux kernel can degrade the real-time response, we will look into three conceptual ways to add real-time capabilities to Linux.

Three Approaches to Enable Linux for Real-Time

There are basically two conceptual ways to enable a Linux system for real-time:

- *Minimize* the impact of possible resource conflicts in the Linux kernel by adding different levels of virtualization support on a "horizontal" level, or
- *Avoid* resource conflicts that would otherwise make Linux non-deterministic by embracing the fact that the Linux SMP kernel can utilize multiple cores or hardware threads and design the system "vertically".

This chapter elaborates the three real-time enabling design approaches mentioned in the Introduction chapter:

- The kernel preemption patch approach: Standard Linux can be modified with the PREEMPT_RT kernel patch. This can be considered as a "horizontal layering" design approach since it implements a kind of para-virtualization of the interrupt management and the kernel locking mechanisms.
- The user space partitioning approach: Linux can modified and configured to isolate a set of cores so that they run a single "pinned" POSIX thread completely without interference from the kernel. This thread (one per core) will be able to utilize the CPU to 100%, and is able to handle external events with a very low overhead in certain circumstances.
- The dual-OS partitioning approach: This takes the vertical partitioning a step further; the RT cores are entirely unplugged from Linux kernel ownership, and therefore they need to be managed by another OS instance. The cores "outside" Linux can either be allocated to a hypervisor guest, or simply run a native OS (Linux, real-time executive or bare-metal). A layer of middleware OS services (IPC, FS, etc) connects the different OS domains in order to create an integrated OS platform to the application layer.

The different design approaches make different trade-offs in the areas of performance, latency, and functionality. An overview of each approach is given in the figure below, and in subsequent sections we will describe and compare the different approaches.

A: The Kernel Preemption Patch Approach – PREEMPT_RT

The PREEMPT_RT patch provides several modifications to get real-time support in the Linux kernel by mitigating the effects of resource conflicts. This is the well-known and "standard" approach to introduce a level of "soft" real-time capabilities to the Linux kernel. The RT patch was

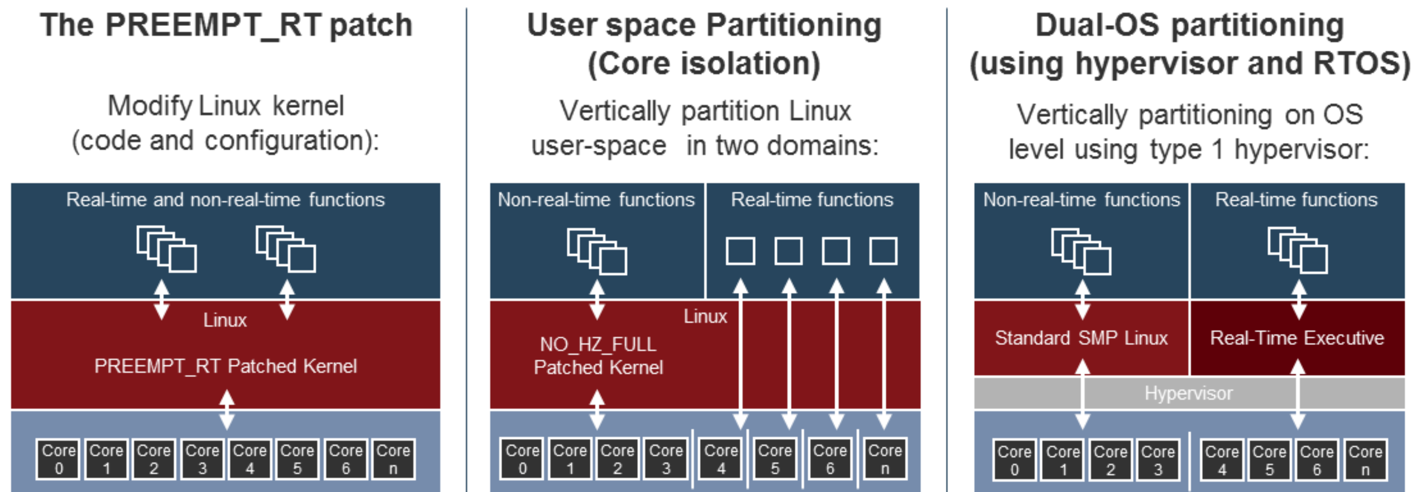

Figure 3. Three approaches to enable real-time in Linux systems

developed by Ingo Molnar, Thomas Gleixner and others, and is maintained today by Linux Foundation. Many of the modifications done throughout the years within the scope of the RT patch were good improvements that have been contributed to the mainline Linux, but the PREEMPT_RT configuration option does still contain a number of major changes that are maintained outside mainline Linux.

The benefit with the RT patch is that the full POSIX programming API will be available as is. The application has access to the regular, feature rich API provided by the Linux kernel and GCC SDK.

The changes in the RT patch include amongst other things the method to replace the kernel spinlock mechanisms to enable full preemption where otherwise the Linux design states a non-preemptible region. On a more detailed level, regular spinlocks are replaced with mutexes with priority inheritance, thus many interrupt handlers need to be migrated into kernel threads to become fully preemptible. In a way, PREEMPT_RT could also be seen as a kind of horizontal partitioning, since it is a patch that "para-virtualizes" and redefines the kernel locking design of the standard Linux kernel.

The RT patch adds overhead to the kernel execution overall and will decrease the throughput performance to some degree, depending on the kind application. Since spinlock regions are replaced with mutex regions for the purpose of enabling preemption, all eventual scalability penalty due to memory contention in spinlocking will be hidden and instead transformed to throughput penalty on thread level. The PREEMPT_RT patch also requires adoptions in driver code and other kernel code, which is not originating from kernel.org.

Although the RT patch improves the worst-case task interrupt response latency significantly, there are still a lot of things that can affect the overall event handling latency time. One example is taking page faults; avoiding these requires use of huge pages and potentially locking a lot of the working set memory in RAM.

When evaluating Linux with the PREEMPT_RT patch enabled and a low to moderate load, we will be able to achieve a worst-case task response latency down to around ~5-50 us. For many application scenarios this is sufficient.

As long as Linux systems used single-core processors, the kernel preemption approach was more or less the only reasonable choice for the embedded market. However, since it has also gained acceptance as a multicore alternative, Enea provides a Linux solution including the PREEMPT_RT patch.

B: The User Space Partitioning Approach – Core Isolation

The strategy with the user space partitioning approach is to utilize the fact that SMP Linux can be configured to control and limit what is executed on a core (to a certain extent). We can create kernel resource isolation "barriers" between sets of cores, "core clusters", so that different applications can run on the same processor simultaneously without affecting the characteristics of each other. We can call this "CPU core isolation". The goal is to isolate a shielded set of cores into a real-time domain such that its local per-CPU schedulers are not in any way affected by the potential massive load that the applications outside this real-time domain generate in terms of kernel resource locking scenarios or thread preemptions. (Note, the execution time will of course be affected if the different domain uses the same level of cache, but in this paper we concentrate on avoiding the resource conflicts on the OS level, which may be substantially higher.)

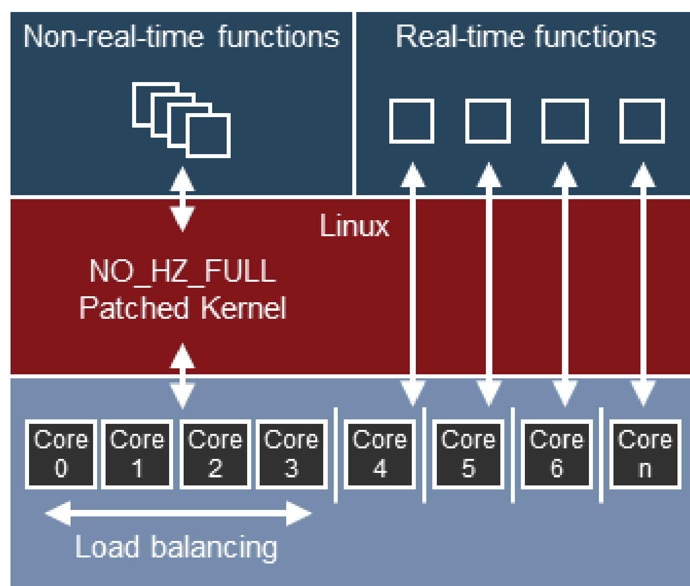

Figure 4. Vertical partitioning with CPU core isolation

The figure above exemplifies a Linux SMP kernel that is divided into two types of execution partitions or domains. Non-critical applications, e.g. O&M and control applications, are located to the non-shielded cores in the "normal", non-real-time partition. High-performance applications that require real-time task response latency and a deterministic behavior of the execution environment are located to the isolated cores in the real-time domain, where each core has exactly one "pinned" Linux thread in running state. In the state-of-the-art Linux of today, it is not possible to have more than one thread on a core if you

want to completely avoid kernel overhead (such as tick), this thread must not do any kernel calls to use services like high-resolution timeouts, or others that result in preemption or scheduling.

Irrespective of whether a core is isolated or not, the Linux kernel shall be the one and only resource management layer for both OS resources and HW resources, and the functionality level in the parts of the POSIX execution model that handles such resources directly or indirectly should be preserved.

Some changes in the kernel and system configuration needs to be done to achieve the real-time partition. We for example need to:

- Disable the regular load balancing in the Linux kernel for the isolated cores, i.e. so that threads are not automatically moved over to isolated cores by kernel.
- Explicitly bind IRQs that belong to the real-time application to that specific core (if polled, no IRQ!)
- Move all other IRQs and kernel threads not belonging to real-time applications to the non-shielded cores.
- Disable the local timer interrupts (removes the ability for the local scheduler to enforce time sharing and to do some book-keeping of resources). I.e. enable the "nohz_full" boot mode and to set scheduler tick deferment to indefinite time.

In summary, the end goal is to remove all sources of potential resource conflicts or system event that may add non-deterministic latency or overhead to the pinned threads on each isolated core.

It is important to note that the full feature level of the POSIX programming API will still be present on the isolated cores, since it is likely needed in startup and configuration scenarios. However, later when the application mode turns to "traffic", the use of the POSIX API will affect both soft and firm real-time characteristics, i.e. the ability to meet deadlines in time and with minimal jitter.

On the set of isolated cores in the real-time partition, the pinned single-threaded application design must explicitly avoid any use of the normal POSIX API calls that may potentially trigger resource conflicts in the kernel call implementations or kick-off kernel resource management or ticks. Omitting the use of OS API would lead to a so called "bare-metal" environment, which basically will be equal to a "standalone", or "bare-metal" C library.

The "bare-metal" variant is much debated and discussed in the industry. It has clearly some valid use cases, such as in High Performance Computing, where zero-overhead Linux can make a big difference in total calculation time since the jitter between the different core's execution time can be kept to a minimum. Also in cases of fast-path packet processing, where a polled loop completely handles a packet from enter to exist, very good performance can be gained by running in user space.

The drawback is however, that applications of the single-threaded polled loop type cannot expect to be able to use any kind of OS services such as task management, high-resolution timeout management, or IPC communication. In many cases, customers end up being forced to design some kind off a "light-weight OS" layer on top and adapt to a then rather poor and non-standard runtime API.

What is often forgotten when discussing so called "bare-metal" design, where applications run entirely in user space with run-to-completion, is the need for a high-speed inter-process communication (IPC) channel between these applications and the code "outside" the bare-metal environment, in the O&M/control part. Such communication can be needed for either control synchronization and/or data exchange, but in any case, the IPC mechanism needs to be very fast and scalable.

The integrated OS solution must implement such a high-speed IPC channel between the real-time partition and the "outside"non real-time partition by using polling techniques and lock-less user-space queues. This IPC "backplane" does not only provide a channel to or from the real-time domain, it also provides a high-speed IPC network between applications spread over the cores of the entire SMP Linux processor. It is a challenge to design such an IPC backplane for high speed as well as scalability, yet not affecting the real-time characteristics in the real-time domain.

An important aspect of the "bare-metal" design pattern in Linux User space is that the ability to debug such an application becomes much more limited, it becomes a "black-box". The normal bookkeeping activity that the kernel performs is not taking place on an isolated core.

When Linux is configured for core isolation and a benchmark is run that is triggered by a single, exclusive interrupt, it is possible to achieve a worst-case task response latency down to around ~3-30 us.

Enea provides a Linux solution with the support for core isolation.

C: The DUAL-OS Partitioning Approach – A Linux/real-time executive AMP Solution

The alternative that will provide the best characteristics of the two worlds, Linux and real-time executives, is of course an OS solution where these are combined side by side on a homogeneous multicore processor cluster:

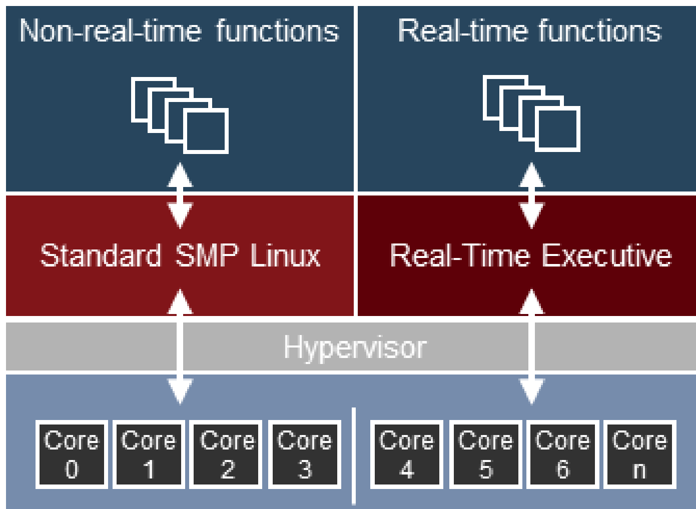

Figure 6. Dual OS partitioning – Linux and a real-time executive as a virtualized guest

This basic conceptual model above provides a Linux-based OS solution with a real-time accelerated SMP POSIX execution environment that has a number of shared OS services and functions in common with Linux, such as IPC, shared file system, point-to-point IP, and a command shell.

The main goal with such an integrated RTOS/POSIX runtime is to provide a programming environment with extremely low OS call/scheduling overhead and very low and deterministic worst-case latency, similar to native "bare-metal" characteristics. The integration of a real-time executive with Linux can be implemented in a couple of ways:

- Native AMP; start Linux on a number of cores, and then start a real-time executive on the rest of the cores. In order for this to work, the physical resources has to be possible to be shared or explicitly divided between the domains. This alternative will not provide any protection between the domains.
- Virtualized AMP; Using a type 1 Hypervisor, Linux is started as the root guest and then a realtime guest is created. This solution will provide a high level of protection between the domains, and it will also provide a higher degree of (non-realtime) HW resource sharing. However, for HW devices used by traffic application this solution uses a direct access, bypassing the Hypervisor. An example of such a Hypervisor can be either Xen or Jailhouse.

On the application and system level, both variants will provide a portable OS service layer regardless of the number of cores since SMP Linux runs on one side, and a SMP real-time executive runs on the other side. Regardless of if virtualization is used or not, or of the choice of Hypervisor implementation, the API to shared services such as IPC and IP will remain portable, as well as the shared POSIX file system and shell functionality.

It is important to consider that the initial effort to develop and integrate a combined BSP for Linux and the real-time executive guest so that the hardware resources are shared in a consistent manner can be quite large. The size of the BSP work will always depend on the actual use case need, but is more of a "one-time" cost. The long-term upside is that such a dual SMP OS solution will be agnostic to the hardware architecture and silicon vendor.

The major upside is that you will get the best of the two worlds:

- Standard POSIX API, C++11
- Future proofness (same API on all ARMv8 and x86 targets)
- Hardware platform independence
- Hypervisor vendor independence
- Independency of number of cores
- Deployment flexibility
- Runtime debug, trace, profiling
- High performance, low OS overhead
- High determinism (low latency and jitter)
- Safety (robust, high availability)
- Security

In this kind of integration, there is no need to tweak or patch the Linux kernel for real-time capabilities since the applications on the Linux side will not be real-time or performance critical. The applications that require very high performance, very low task response latency and very low OS overhead in scheduling and use of OS services such as POSIX timers, threads or mutexes, will be running in the RT domain. In the RT domain the application will use the set of POSIX API provided by the real-time executive in parallel with the actual device driver architecture provided by the real-time executive vendor. Whatever tools supported by the vendor in terms of

system debug, trace and profiling, will also be available, as well as the complete set of tools for the standard, unmodified Linux.

So why use a real-time executive as runtime in the real-time domain instead of running a "bare-metal" thread?

The answer is simply that everyone at the end of the day will need some or many of the regular OS services a POSIX API provides, there is a need to access to services like time and timeouts, task synchronization, memory allocation, event tracing, IPC, etc. The user needs to be able to debug the application. In the embedded industry there is a huge amount of legacy multithreaded POSIX application to port, but also in new designs the need for multithreading and OS services will arise. In order to utilize the processor in a cost-effient way, there must be a flexibility to utilize the "spare" capacity of the cores, i.e. run some applications or services in the background or do load balancing.

But at the same time, some applications will require the hard real-time capability provided by a "bare-metal" thread, i.e. a worst case interrupt latency to a thread in the magnitude of around ~1 us. So in order to get the best of two worlds, two different OS implementations needs to be combined, Linux and one real-time executive. There is no free lunch.

This kind of solution is very suitable and proven in use for high-demanding applications like 5G baseband processing, where the RLC/MAC/PHY processing needs to be very efficient and requires very low worst case latency, down to below 5 us. It may also be suitable for other types of applications that require both very high multithreaded performance (uses OS API) and low latency, for example:

- Fast-path IP processing; when the polling thread starts to need OS services, these are more deterministic and consumes much less overhead in an real-time executive than in Linux.
- In new use cases in automotive not covered by the "classic" certified domain.
- In Embedded Industry, hard real-time algorithms and control loops.
- In Finance trading, to achieve very low worst case latency and jitter on transaction handling.

Example: A Dual-OS AMP Solution Using Enea Real-Time Executive and Jailhouse Hypervisor

There are very many ways to provide a real-time-acceleration to Linux. This section will describe how such a solution can look like with the integration of Xilinx Linux, the Jailhouse Hypervisor and Enea Real-Time Executive on the Xilinx Ultrascale Cortex-A53 quad core SoC.

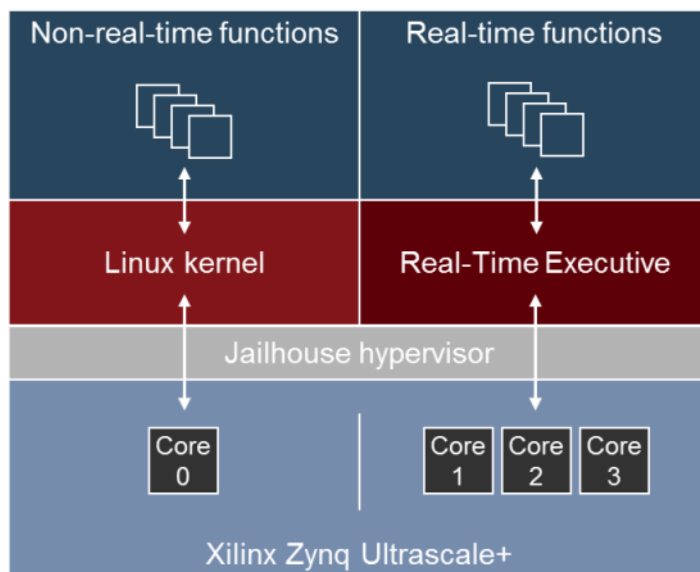

Figure 6. Dual OS partitioning – Linux on one core and OSE on three cores as a Jailhouse guest

In this reference implementation of a dual-OS solution, standard Linux is booted on all four cores initially, since this is the concept of using the Jailhouse Hypervisor, which is designed to start as a kernel module after the system is up. The Jailhouse Hypervisor is a Linux-based partitioning type-1 Hypervisor: https://github.com/siemens/jailhouse

The Jailhouse Hypervisor was chosen in this demonstration for a couple of reasons:

- It is open source.
- It is very small, a code base of around 7000 lines of code.
- It starts as an application with kernel module in Linux, which enables for a late and flexible configuration on the size of the real-time domain.
- It provides a direct memory-mapped access to hardware devices from guest level.
- It provides a true "bare-metal" execution speed and total absence of Linux interference.

After SMP Linux has been started on all four cores, there is a decision; to use two or three cores for the RT accelerator domain? The picture describes a choice to boot RT domain on three cores, and therefore the Jailhouse hypervisor is started as a kernel module with two cell configurations – Linux as root guest on core 0, and the real-time executive as guest on core 1-3. Note that Linux initially starts up in EL2 from uboot, but when the Jailhouse Hypervisor kernel module is started, it "migrates" the entire Linux environment to be a root guest running in Exception Level 1 (EL1) mode.

After the Linux kernel has become a root guest, a script instructs the Hypervisor layer to start the real-time executive in the guest cell. The Hypervisor kernel module copies the executive binary image to the range of memory set aside to the guest, and then it starts the binary by doing a SMC call. The real-time microkernel then boots up in EL1 mode, establishes an IPC channel to the Linux side, and then performs a secondary boot stage where a number of real-time executive program binaries are loaded from the Linux file system. In this stage, the OS services like virtual Ethernet and shared file system is also started.

This boot stage fires up the drivers for the HW devices that we decided to allocate to RT domain. The Jailhouse Hypervisor allows the guest direct access to HW, the guest cell configuration defines a valid memory range that we defined as being outside Linux memory range in the Linux boot command string. All accesses outside the cell boundaries will generate a second-stage EL2 translation trap to the Hypervisor, the guest will halt awaiting further actions.

Due to the fact that the RT guest has direct access to the interrupt controller and other devices, we will get the same low latency to HW as we get when we run natively (without hypervisor).

In a Linux terminal window, its now possible to open the real-time executive shell command window. Using the IP connection, it is also now possible to run all the IP-based O&M Optima tools. Enea Real-Time Executive and the Optima Tools Suite has a wide range of tools for event tracing, performance profiling, and source code debugging.

What is important though is that even though there is an access to a feature-rich SMP POSIX API for threads on RT domain side, it is still very much possible to design demanding hard real-time functions that requires a worst case response latency of down to ~1 us.

Enea provides the knowledge and foundation to efficiently integrate a Linux-real-time AMP solution on ARMv8 and x86 targets.

Summary and Conclusions

To conclude, there are a number of ways to introduce real -time capabilities to Linux, and three of them have been outlined in this paper:

- *The Linux kernel RT patch approach*, which improves the real-time responsiveness to POSIX applications. This alternative is very suitable for systems with low to moderate real-time requirements, in the range of ~5-50 us.
- *The user space partitioning approach*, which uses the concept of CPU core isolation and an exclusive user level thread pinned to each RT core. This alternative is not very suitable for any kind of multithreaded POSIX program, it is more suitable for HPC-type or polled, fast path processing applications. If using pinned ISR:s, its possible to reach down to around ~3-30 us in worst case task response latency.
- *The dual-OS partitioning approach*, which complements standard SMP Linux with an SMP real-time executive based "RT accelerator" on a multicore processor. This solution can be implemented either natively, or using a type 1 Hypervisor such as for example Xen or Jailhouse. This solution has an initial higher cost to deploy, but provides in the long run a more future proof and portable platform since standard Linux is used. This alternative is suitable for hard real-time applications that needs very high processing capacity and determinism. An SMP real-time executive provides a worst-case task response latency in the range of ~1 us and a task switch overhead 10-15 times less than Linux.

The fact that the standard Linux kernel design continuously improves with regard to the kernel resource separation on core basis, i.e. the "space-partitioning capabilities", is a promising foundation for the vertical partitioning approach using dual-OS as being a good solution for next-generation Linux-based embedded multicore platforms.

Enea as an OS Solution Partner

Enea has a long and proven track record as provider of reliable, high-performance and scalable real-time solutions for the telecom market and real-time embedded industry.

Enea develops and provides multicore real-time operating systems and Linux solutions with real-time capabilities. Enea Real-Time Executive has a proven-in-use, lock-less design that provides a supreme performance and maintained low latency even on many cores.

Open source software has through the years been a natural part of the Enea product portfolio. Consequently, Enea provides OS solutions for all design approaches described in this paper:

- Enea Linux with PREEMPT_RT
- Enea Linux with user mode partitioning (Core isolation)
- Enea Linux including real-time executive accelerator

The extensive know-how of real-time, IPC, Linux, RTOS and multicore techniques at Enea brings valuable experiences for supplying the market with future efficient real-time enablers for Linux and RTOS. On www.enea.com you can read more about solutions from Enea.

Glossary

API:	Application Programming Interface
CPU:	Central Processing Unit
GPL:	GNU General Public License
IPC:	Inter-Process Communication
IRQ:	Interrupt Request
OS:	Operating System
Pthread:	POSIX thread
POSIX:	Portable Operating System Interface
QoS:	Quality of Service
RCU:	Ready-copy-update
ISR:	Interrupt Service Routine
O&M:	Operation & Maintenance
RT:	Real-Time
RTOS:	Real-Time Operating System
SMP:	Symmetric Multiprocessing
UIO:	User-space I/O

Enea develops the software foundation for the connected society with a special emphasis on reducing cost and complexity at the network edge. We supply open-source based NFVI software platforms, embedded DPI software, Linux and Real-Time Operating Systems, and professional services. Solution vendors, Systems Integrators, and Service Providers use Enea to create new networking products and services faster, better and at a lower cost. More than 3 billion people around the globe already rely on Enea technologies in their daily lives. For more information: www.enea.com

Copyright © 2018 Enea. All rights reserved. Enea and the Enea logo, are trademarks of Enea. Other names and brands may be claimed as the property of others.