# 7

# Embedded Operating Systems

**Claudio Scordino[1], Errico Guidieri[1], Bruno Morelli[1], Andrea Marongiu[2,3], Giuseppe Tagliavini[3] and Paolo Gai[1]**

[1]Evidence SRL, Italy
[2]Swiss Federal Institute of Technology in Zurich (ETHZ), Switzerland
[3]University of Bologna, Italy

In this chapter, we will provide a description of existing open-source operating systems (OSs) which have been analyzed with the objective of providing a porting for the reference architecture described in Chapter 2. Among the various possibilities, the ERIKA Enterprise RTOS (Real-Time Operating System) and Linux with preemption patches have been selected. A description of the porting effort on the reference architecture has also been provided.

## 7.1 Introduction

In the past, OSs for high-performance computing (HPC) were based on custom-tailored solutions to fully exploit all performance opportunities of supercomputers. Nowadays, instead, HPC systems are being moved away from in-house OSs to more generic OS solutions like Linux. Such a trend can be observed in the TOP500 list [1] that includes the 500 most powerful supercomputers in the world, in which Linux dominates the competition. In fact, in around 20 years, Linux has been capable of conquering all the TOP500 list from scratch (for the first time in November 2017).

Each manufacturer, however, still implements specific changes to the Linux OS to better exploit specific computer hardware features. This is especially true in the case of computing nodes in which lightweight kernels are used to speed up the computation.

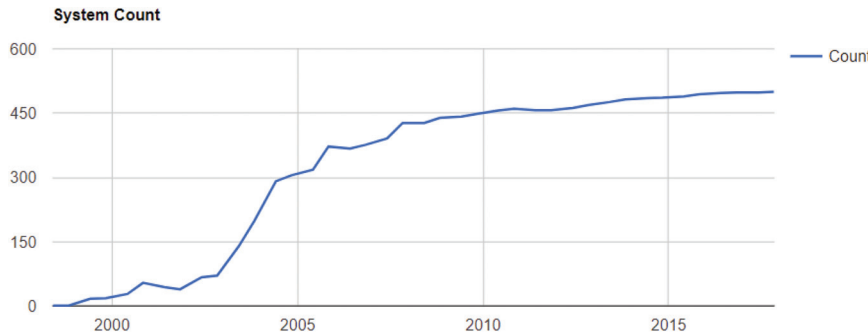


**Figure 7.1** Number of Linux-based supercomputers in the TOP500 list.

Linux is a full-featured OS, originally designed to be used in server or desktop environments. Since then, Linux has evolved and grown to be used in almost all computer areas – among others, embedded systems and parallel clusters. Linux currently supports almost every hardware processor, including x86, x86-64, ARM, PowerPC, MIPS, SuperH, IBM S/390, Motorola 68000, SPARC, etc. The programmability and the portability of code across different systems are ensured by the well-known "Portable Operating System Interface" (POSIX) API. This is an IEEE standard defining the basic environment and set of functions offered by the OS to the application programs.

Hence, the main reason for this success and popularity in the HPC sector is its excellent performance and its extreme scalability, due to very carefully designed data structures like Linux Read-Copy Update (RCU) [2]. This scalability, together with the high modularity, enables excellent performance on both a powerful parallel cluster made by thousands of cores and a small embedded microcontroller, as will be shown in the next sections.

Therefore, when designing the support for our predictable parallel programming framework, we started selecting Linux as the basic block for executing the target parallel applications. On the other hand, Linux alone is not sufficient for implementing the needed runtime support on our reference architecture: a solution needed to be found for the compute cores, where a tiny RTOS is needed in order to provide an efficient scheduling platform to support the parallel runtime described in Chapter 6.

This chapter in particular describes in detail how the scheduling techniques designed in Chapter 4 have been implemented on the reference architecture. The chapter includes notes about the selection of the tiny RTOS for the compute cores, with a description of the RTOS, as well as the solutions implemented to support Linux on the I/O cores with real-time performance.

This chapter is structured as follows. Section 7.2 describes the state of the art of the real-time support for the Linux OS and as well for small RTOSes. Section 7.3 describes the requirements that influenced the choice of the RTOS, which is described in detail in Section 7.4. Section 7.5 provides some insights about the OS support for the host processor and for the many-core processor. Finally, Section 7.6 summarizes the chapter.

## 7.2 State of The Art

### 7.2.1 Real-time Support in Linux

As noted in the Section "Introduction," in the last years, there has been a considerable interest in using Linux for both HPC and real-time control systems, from academic institutions, independent developers, and industries. There are several reasons for this rising interest.

First of all, Linux is an Open Source project, meaning that the source code of the OS is freely available to everybody, and can be customized according to user needs, provided that the modified version is still licensed under the GNU General Public License (GPL) [3]. This license allows anybody to redistribute, and even sell, a product as long as the recipient is able to exercise the same rights (access to the source-code included). This way, a user (for example, a company) is not tied to the OS provider anymore, and is free to modify the OS at will. The Open Source license helped the growth of a large community of researchers and developers who added new features to the kernel and ported Linux to new architectures. Nowadays, there is a huge number of programs, libraries, and tools available as Open Source code that can be used to build a customized version of the OS.

Moreover, Linux has the simple and elegant design of the UNIX OSs, which guarantees meeting the typical reliability and security requirements of real-time systems.

Finally, the huge community of engineers and developers working on Linux makes finding expert programmers very easy.

Unfortunately, the standard mainline kernel (as provided by Linus Torvalds) is not adequate to be used as RTOS. Linux has been designed to be a general-purpose operating system (GPOS), and thus not much attention has been given to the problem of reducing the latency of critical operations. Instead, the main design goal of the Linux kernel has been (and still remains) to optimize the average throughput (i.e., the amount of "useful work" done by the system in the unit of time). For this reason, a Linux program may suffer a high latency in response to critical events. To overcome these problems, many

approaches have been proposed in the last years to modify Linux in order to make it more "real-time." These approaches can be grouped in the following classes [4]:

1. Hard real-time scheduling through a Hardware Abstraction Layer (HAL);
2. Latency reduction through better preemption mechanisms and interrupt handling;
3. Proper real-time scheduling policies.

The following subsections describe each approach in detail.

#### 7.2.1.1 Hard real-time support

This approach consists in creating a layer of virtual hardware between the standard Linux kernel and the real hardware. This layer is called Real-Time Hardware Abstraction Layer (RTHAL). It abstracts the hardware timers and interrupts and adds a separate subsystem to run the real-time tasks. The Linux kernel and all the normal Linux processes are then managed by the abstraction layer as the lowest priority tasks — i.e., the Linux kernel only executes when there are no real-time tasks to run.

The first project implementing this approach was RTLinux [5]. The project started at Finite State Machine Labs (FSMLabs) in 1995. Then, it was released in two different versions: an Open Source version (under GPL license) and a more featured commercial version. An operation of patenting issued in US in 1999, however, generated a massive transition of developers towards the parallel project RTAI. Then, the commercial version was bought by WindRiver. Nowadays, both versions are not maintained anymore [5].

RTAI [6] (which stands for "Real-Time Application Interface") is a project started as a variant of RTLinux in 1997 at Dipartimento di Ingegneria Aerospaziale of Politecnico di Milano (DIAPM), Italy. The project is under LGPL license, and it was supported by a large community of developers, based on the Open Source model. Although the project initially started from the original RTLinux code, it has been completely rewritten over time. In particular, the RTAI community has developed the Adaptive Domain Environment for Operating Systems (ADEOS) nanokernel as an alternative for RTAI's core, to get rid of the old kernel patch and exploit a more structured and flexible way to add a real-time environment to Linux [4]. The project mainly targets the x86 architecture and is currently maintained (even if less popular than it used to be in the past).

Xenomai [7] was born in 2001 as an evolution of Fusion, a project to run RTAI tasks in the user space. With Xenomai, a real-time task can

execute in user space or in kernel space. Normally, it starts in kernel space (i.e., "primary domain"), where it has real time performance. When the real-time task invokes a function belonging to the Linux standard API or libraries, it is automatically migrated to the user-level (i.e., "secondary domain"), under the control of the Linux scheduler. In this secondary domain, it keeps a high priority, being scheduled with the SCHED FIFO or SCHED RR Linux policies. However, it can experience some delay and latency, due to the fact that it is scheduled by Linux. After the function call has been completed, the task can go back to the primary mode by explicitly calling a function. In this way, at the cost of some limited unpredictability, the real-time programmer can use the full power of Linux also for real-time applications.

Among the various projects implementing the hardware abstraction approach, Xenomai is the one which supports the highest number of embedded architectures. It supports ARM, Blackfin, NiosII, PPC and, of course, x86. Xenomai also offers a set of skins implementing the various APIs of popular RTOS such as Windriver VxWorks [8], as well as the POSIX API [9]. In version 3 of Xenomai, the project aims at working on top of both a native Linux kernel and a kernel with PREEMPT_RT [10], by providing a set of user-space libraries enabling seamless porting of applications among the various OS versions.

It is important to highlight the advantages of the approach of hardware abstraction. First of all, the latency reduction is really effective [4]. This allows the implementation of very fast control loops for applications like vibrational control. Moreover, it is possible to use a full-featured OS like Linux for both the real-time and the non-real-time activities (e.g., HMI, logging, monitoring, communications, etc.). Finally, the possibility of developing and then executing the code on the same hardware platform, considerably simplifies the complexity of the development environment.

Typical drawbacks of this approach – which depend on the particular implementation – are:

- Real-time tasks must be implemented using specific APIs, and they cannot access typical Linux services without losing their real-time guarantees.
- The implementation is very hardware-dependent, and may not be available for a specific architecture.
- The real-time tasks are typically executed as modules dynamically loaded into the kernel. Thus, there is no memory protection and a buggy real-time task may crash the whole system.

For these reasons, this approach is usually followed only to build hard real-time systems with very tight requirements.

#### 7.2.1.2 Latency reduction

"Latency" can be defined as the time between the occurrence of an event and the beginning of the action to respond to the event [4]. In the case of an OS, it is often defined as the time between the interrupt signal arriving to the processor (signaling the occurrence of an external event like data from a sensor) and the time when the handling routine starts execution (e.g., the real-time task that responds to the event). Since in the development of critical real-time control systems, it is necessary to account for the worst-case scenario, a particularly important measure is the maximum latency value.

The two main sources of latency in general-purpose OSs are task latency and timer resolution:

1. Task latency is experienced by a process when it cannot preempt a lower priority process because this is executing in kernel context (i.e., the kernel is executing on behalf of the process). Typically, monolithic OSs do not allow more than one stream of execution in kernel context, so that the high-priority task cannot execute until the kernel code either returns to user-space or explicitly blocks. As we will explain in the following paragraphs, Linux has been capable of mixing the advantages of a traditional monolithic design with the performance of concurrent streams of execution within the kernel.
2. The timer resolution depends on the frequency at which the electronics issues the timing interrupts (also called "tick"). This hardware timer is programmed by the OS to issue interrupts at a pre-programmed period of time. The periodic tick rate directly affects the granularity of all timing activities. The Linux kernel has recently switched towards a dynamic tick timer, where the timer does not issue interrupts at a periodic rate. This feature allows the reduction of energy consumption whenever the system is idle.

In the course of the years, several strategies have been designed and implemented by kernel developers to reduce these values. Among the mechanisms already integrated in the official Linux kernel, we can find:

- Robert Love's Preemptible Kernel patch to make the Linux kernel preemptible just like user-space. This means that several flows of kernel execution can be run simultaneously. Urgent events can be served regardless of the fact that the system is running in the kernel context.

Hence, it becomes possible to preempt a process at any point, as long as the kernel is in a consistent state. With this patch the Linux kernel has become a fully preemptive kernel, unlike most existing OSs (UNIX variants included). This feature was introduced in the 2.6 kernel series (December 2003).

- High Resolution Timers (HRT) is a mechanism to issue timer interrupts aperiodically – i.e., the system timer is programmed to generate the interrupt after an interval of time that is not constant, but depends on the next event scheduled by the OS. Often, these implementations also exploit processor-specific hardware (like the APIC on modern x86 processors) to obtain a better timing resolution. This feature was introduced in the 2.6.16 kernel release (March 2006).
- Priority inheritance for user-level mutex, available since release 2.6.18 (September 2006). Priority inheritance support is useful to guarantee bounded blocking times in case more than one thread needs to concurrently access the same resource. The main idea is that blocking threads inherit the priority of the blocked threads, thus giving them additional importance in order to finish their job early.
- Threaded interrupts by converting interrupt handlers into preemptible kernel threads, available since release 2.6.30 (June 2009). To better understand the effect of this patch, we have to consider that the typical way interrupts are managed in Linux is to manage the effect of the interruption immediately inside the so-called interrupt handler. In this way, peripherals are handled immediately, typically providing a better throughput (because thread waiting for asynchronous events are put earlier in the ready queue). On the other hand, a real-time system may have a few "important" IRQs that need immediate service, while the others, linked to lower priority activities (e.g., network, disk I/O), can experiences higher delays. Therefore, having all interrupt services immediately may provide unwanted jitter in the response times, as low-priority IRQ handlers may interrupt high-priority tasks. The threaded interrupt patch solves this problem by transforming all IRQ handlers into kernel threads. As a result, the IRQ handlers (and their impact on the response time) are minimized. Moreover, users can play with real-time priorities to eventually raise the priorities of the important interrupts, therefore providing stronger real-time guarantees.

PREEMPT_RT [10] is an on-going project supported by the Linux Foundation [11] to bring real-time performance to a further level of sophistication, by introducing preemptible ("sleeping") spinlocks and RT mutexes implementing Priority Inheritance to avoid priority inversion.

It is worth specifying that the purpose of the PREEMPT_RT patch is not to improve the throughput or the overall performance. The patch aims at reducing the maximum latency experienced by an application to make the system more predictable and deterministic. The average latency, however, is often increased.

#### 7.2.1.3 Real-time CPU scheduling

Linux systems traditionally offered only two kind of scheduling policies:

1. SCHED_OTHER: Best-effort round-robin scheduler;
2. SCHED_FIFO/SCHED_RR: Fixed-priority POSIX scheduling.

During the last decade, due to the increasing need of a proper real-time scheduler, a number of projects have been proposed to add more sophisticated real-time scheduling (e.g., SCHED_SOFTRR [12], SCHED_ISO [13], etc.). However, they remained as separate projects and have never been integrated in the mainline kernel.

During the last years, the real-time scheduler SCHED_DEADLINE [14, 15] originally proposed and developed by Evidence Srl in the context of the EU project ACTORS [16], has been integrated in the Linux kernel. It is available since the stable release 3.14 (March 2014). It consists of a platform-independent real-time CPU scheduler based on the Earliest Deadline Scheduler (EDF) algorithm [17], and it offers temporal isolation between the running tasks. This means that the temporal behavior of each task (i.e., its ability to meet its deadlines) is not affected by the behavior of the other tasks running in the system. Even if a task misbehaves, it is not able to exploit larger execution times than the amount it has been allocated. The scheduler only enforces temporal isolation on the CPU, and it does not yet take into account shared hardware resources that could affect the timing behavior.

A recent collaboration between Scuola Superiore Sant'Anna, ARM Ltd. and Evidence Srl, has aimed at overcoming the non-work-conserving nature of SCHED_DEADLINE while keeping the real-time predictability. This joint effort that replaced the previous CBS algorithm with GRUB has been merged since kernel release 4.13.

### 7.2.2 Survey of Existing Embedded RTOSs

The market of embedded RTOSs has been exploited in the past decades by several companies that have been able to build solid businesses. These companies started several years ago, when the competition from free OSs

was non-existent or very low. Thus, they had enough time to create a strong business built on top of popular and reliable products. Nowadays, the market is full of commercial solutions, which differentiate in the application domain (e.g., automotive, avionics, railway, etc.) and in the licensing model. Most popular commercial RTOSs are: Windriver VxWorks [8], Green Hills Integrity [18], QNX [19], SYSGO PikeOS [20], Mentor Graphics Nucleus RTOS [21], LynuxWorks LynxOS [22], and Micrium µc/OS-III [23]. However, there are some other interesting commercial products like Segger EmbOS [24], ENEA OSE [25], and Arccore Arctic core [26].

On the other hand, valid Open-Source alternatives exist. The development of a completely free software tool chain being our target, the focus of this subsection will be more on the free RTOSs available publicly. Some free RTOSs, in fact, have now reached a level of maturity in terms of reliability and popularity that can compete with commercial solutions. The Open-Source licenses allow the modification of the source code and porting the RTOS on the newest many-core architectures.

This section provides an overview of the free RTOSs available. For each RTOS, the list of supported architectures, the level of maturity and the kind of real-time support are briefly provided. Other information about existing RTOSs can be found in [27].

### FreeRTOS

FreeRTOS [28] is a small RTOS written in C. It provides threads, tasks, mutexes, semaphores and software timers. A tick-less mode is provided for low-power applications.

It supports several architectures, including ARM (ARM7/9, Cortex-A/M), Altera Nios2, Atmel AVR and AVR32, Cortus, Cypress PSoC, Freescale Coldfire and Kinetis, Infineon TriCore, Intel x86, Microchip dsPIC and PIC18/24/32 and dsPIC, Renesas H8/S, SuperH, Fujitsu, Xilinx Microblaze, etc.

It does not implement very advanced scheduling algorithms, but it offers a classical preemptive or cooperative fixed-priority round-robin with priority inheritance mutexes.

The RTOS is Open Source, and was initially distributed under a license similar to GPL with linking exception [29]. Recently the FreeRTOS kernel has been relicensed under the MIT license thanks to the collaboration with Amazon AWS. A couple of commercial versions called SafeRTOS and OpenRTOS are available as well. The typical footprint is between 5 KB and 10 KB.

### Contiki

Contiki [30] is an Open-Source OS for networked, memory-constrained systems with a particular focus on low-power Internet of things devices. It supports about a dozen microcontrollers, even if the ARM architecture is not included. The Open-Source license is BSD, which allows the usage of the OS in commercial devices without releasing proprietary code.

Although several resources include Contiki in the list of free RTOSs, Contiki is not a proper RTOS. The implementation is based on the concept of protothreads, which are non-preemptible stack-less threads [31]. Context switch can only take place on blocking operations, and does not preserve the content of the stack (i.e., global variables must be used to maintain variables across context switches).

Stack sharing is a useful feature, but the lack of preemptive support and advanced scheduling mechanisms made this OS not suitable to meet the needs of the parallel programming software framework we want to implement.

### Marte OS

Marte OS [32] is a hard RTOS that follows the Minimal Real-Time POSIX.13 subset. It has been developed by the University of Cantabria. Although it is claimed to be designed for the embedded domain, the only supported platform is the x86 architecture. The development is discontinued, and the latest contributions date back to June 2011.

### Ecos and EcosPro

Ecos [33] is an Open-Source RTOS for applications which need only one process with multiple threads. The source code is under GNU GPL with linking exception.

The current version is 3.0 and it runs on a wide variety of hardware architectures, including ARM, CalmRISC, Motorola 68000/Coldfire, fr30, FR-V, Hitachi H8, IA32, MIPS, MN10300, OpenRISC, PowerPC, SPARC, SuperH, and V8xx.

The footprint is in the order of tens of KB, which does not make it suitable for processing units with extremely low memory. The kernel is currently developed in a closed-source fork named eCosPro.

### FreeOSEK

FreeOSEK [34] is a minimal RTOS implementing the OSEK/VDX automotive standard, like Erika Enterprise. The Open-Source license (GNU GPLv3 with linking exception) is similar to the one of Erika Enterprise too. However,

it only supports the ARM7 architecture, the development community is small, and the project does not appear to be actively maintained.

### QP

Quantum platform (QP) [35] is a family of lightweight, open source software frameworks developed by company Quantum Leaps. These frameworks allow building modular real-time embedded applications as systems of cooperating, event-driven active objects (actors). In particular, QK (Quantum Kernel) is a tiny preemptive non-blocking run-to-completion kernel designed specifically for executing state machines in a run-to-completion (RTC) fashion.

Quantum platform supports several microcontrollers, including ARM Cortex-M, ARM 7/9/Cortex-M, Atmel AVR Mega and AVR32, Texas Instruments MSP430/TMS320C28x/TMS320C55x, Renesas Rx600/R8C/H8, Freescale Coldfire/68HC08, Altera Nios II, Microchip PIC24/dsPIC, and Cypress PSoC1.

The software is released in dual licensing: an Open-Source and a commercial license. The Open-Source license is GNU GPL v3, which requires the release of the source code to any end user. Unfortunately, the Open-Source license chosen is not suitable for consumer electronics, where the companies want to keep the intellectual property of their application software.

### Trampoline

Trampoline [36] is an RTOS which aims at OSEK/VDX automotive certification. However, unlike ERIKA Enterprise, it has not yet been certified.

Only the following architectures are supported: Cortex M, Cortex A7 (alone or with the Hypervisor XVisor), RISC-V, PowerPC 32 bits, AVR, ARM 32 bit.

The Open-Source license at the time the evaluation was made was LGPL v2.1. This license is not very suitable for consumer electronics because it implies that any receiver of the binary (e.g., final user buying a product) must be given access to the low-level and the possibility of relinking the application towards a newer version of the RTOS. The license was changed afterwards to GPL v2 in September 2015.

### RTEMS

RTEMS [37] is a fully-featured Open-Source RTOS supporting several application programming interfaces (APIs) such as POSIX and BSD sockets. It

is used in several application domains (e.g., avionics, medical, networking) and supports a wide range of architectures including ARM, PowerPC, Intel, Blackfin, MIPS, and Microblaze. It implements a single process, multi-threaded environment. The Open-Source license is similar (but not equal) to the more popular GPL with Linking Exception [29].

The footprint is not extremely small, and for the smallest applications, ranges from 64 to 128 K on nearly all CPU families [38]. For this reason, another project called TinyRTEMS [39] has been created to reduce the footprint of RTEMS. However, its Open-Source license is GPLv2, which is not suitable for development in industrial contexts.

### TinyOS

TinyOS [40] is an Open-Source OS specifically designed for low-power wireless devices (e.g., sensor networks) and mainly used in research institutions. It has been designed for very resource-constrained devices, such as microcontrollers with a few KB of RAM and a few tens of KB of code space. It's also been designed for devices that need to be very low power.

TinyOS programs are built out of software components, some of which present hardware abstractions. Components are connected to each other using interfaces. TinyOS provides interfaces and components for common abstractions such as packet communication, routing, sensing, actuation, and storage.

TinyOS cannot be considered a proper real-time OS, since it implements a non-preemptive thread model.

The OS is licensed under BSD license which, like GPL with linking exception, does not require redistribution of the source code of the application.

TinyOS supports Texas Instruments MSP430, Atmel Atmega128, and Intel px27ax families of microcontrollers. Currently, it does not support the family of ARM Cortex processors. The development of TinyOS has been discontinued since a few years.

### ChibiOS/RT

ChibiOS/RT [41] is a compact and Open-Source RTOS. It is designed for embedded real-time applications where execution efficiency and compact code are important requirements. This RTOS is characterized by its high portability, compact size and, mainly, by its architecture optimized for extremely efficient context switching. It supports a preemptive thread model but it does not support stack sharing among threads.

The official list of supported microcontrollers is mainly focused on the ARM Cortex-M family, even if a very few other processors (i.e., ARM7, AVR Mega, MSP430, Power Architecture e200z, and STM8) are supported as well. Some further microcontrollers are not officially supported, and the porting of the RTOSs has been done by individual developers.

The footprint of this RTOS is very low, being between 1 KB and 5.5 KB.

ChibiOS/RT is provided under several licenses. Besides the commercial license, unstable releases are available as GPL v3 and stable releases as GPL v3 with linking exception. Since version 3 of GPL does not allow "tivoization" [42], these Open-Source licenses are not suitable for industrial contexts where the manufacturer wants to prevent users from running modified versions of the software through hardware restrictions.

### ERIKA Enterprise v2

Erika Enterprise v2 [43] is a minimal RTOS providing hard real-time guarantees. It is developed by partner Evidence Srl, but it is released for free. The Open-Source license – GPL with linking exception (also known as "Classpath") [29] – is suitable for industrial usage because it allows linking (even statically) the proprietary application code with the RTOS without the need of releasing the source code.

The RTOS was born in 2002 to target the automotive market. During the course of the years it has been certified OSEK/VDX and it is currently used by either automotive companies (as Magneti Marelli and Cobra) or research institutions. ERIKA Enterprise v2 implements the AUTOSAR API 4.0.3 as well, up to Scalability Class 4.

Besides the very small footprint (about 2–4 KB), ERIKA Enterprise has innovative features, like advanced scheduling algorithms (e.g., resource reservation, immediate priority ceiling, etc.) and stack sharing to reduce memory usage.

It supports several microcontrollers (from 8-bit to 32-bit) and it has been one of the first RTOSs supporting multicore platforms (i.e., Altera NiosII). The current list of supported architectures includes Atmel AVR and Atmega, ARM 7 and Cortex-M, Altera NiosII, Freescale S12 and MPC, Infineon Aurix and Tricore, Lattice Mico32, Microchip dsPIC and PIC32, Renesas RX200, and TI MSP430. A preliminary support for ARM Cortex-A as well as the integration with Linux on the same multicore chip has been shown during a talk at the Automotive Linux Summit Fall [44] in October 2013.

Version 3 of ERIKA Enterprise has also been released recently [45]. The architecture of ERIKA Enterprise v3 has been directly derived as an evolution of the work described in this chapter, and is aimed to support full AUTOSAR OS compliance on various single and multi-/manycore platforms, including support for hypervisors.

### 7.2.3 Classification of Embedded RTOSs

The existing open-source RTOSs can be grouped in the following classes:

1. POSIX RTOSs, which provide the typical POSIX API allowing dynamic thread creation and resource allocation. These RTOSs have a large footprint due to the implementation of the powerful but complex POSIX API. Examples are: Marte OS and RTEMS.
2. Simil-POSIX RTOSs, which try to offer an API with the same capabilities of POSIX (i.e., dynamic thread creation and resource allocation) but at a lower footprint meeting the typical constraints of small embedded systems. Examples are: FreeRTOS, Ecos and ChibiOS/RT.
3. OSEK RTOSs, implementing the OSEK/VDX API with static thread creation but still allowing thread preemption. These RTOSs are characterized by a low footprint. Moreover, they usually also offer stack-sharing among the threads, allowing the reduction of memory consumption at run-time. Examples are: ERIKA Enterprise, Trampoline, and FreeOSEK.
4. Other minimal RTOSs, which have a low footprint and a non-preemptive thread model by construction. Usually, these RTOSs offer the stack-sharing capability. Examples are: TinyOS and Contiki.

This classification is shown in the following Table 7.1:

**Table 7.1** Classification of RTOSs

| | POSIX | Simil-POSIX | OSEK | Other Minimal |
|---|---|---|---|---|
| API | POSIX | Custom | OSEK/VDX | Custom |
| Footprint size | Big | Medium | Small | Small |
| Thread preemption | V | V | V | X |
| Thread creation | V | V | – | – |
| Stack sharing | – | – | V | V |
| Examples | MarteOS | FreeRTOS | ERIKA Enterprise | TinyOS |
| | RTEMS | Ecos | Trampoline | Contiki |
| | | ChibiOS/RT | FreeOSEK | QP |

## 7.3 Requirements for The Choice of The Run Time System

This section includes a short description of the main requirements that influenced the choice of the OS platform for the implementation of our parallel programming model.

### 7.3.1 Programming Model

The run-time system is a fundamental software component of the parallel programming model to transform the parallel expressions defined by the user into threads that execute in the different processing units, i.e., cores.

Therefore, the OS system must provide support to execute the run-time system that will implement the API services defined by the parallel programming model. In our case, the requirement is related to the fact that an UNIX environment such as Linux should be present, with support for the C and C++ programming languages.

### 7.3.2 Preemption Support

In single-core real-time systems, allowing a thread to be preempted has a positive impact on the schedulability of the system because the blocking on higher-priority jobs is significantly limited. However, in many-core systems, the impact of preemptions on schedulability is not as clear, since higher priority jobs might have a chance to execute on one of the many other cores available in the system. Nevertheless, for highly parallel workloads, it may happen that all cores are occupied by lower-priority parallel jobs, so that higher-priority instances may be blocked for the whole duration of the lower-priority jobs. In this case, a smart preemption support might be beneficial, allowing a subset of the lower-priority instances to be preempted in favor of the higher-priority jobs. The remaining lower-priority instance may continue executing on the remaining cores, while the state of the preempted instances needs to be saved by the OS, in order to restore it as soon as there are computing units available again.

In order to develop the proper OS mechanisms, it is necessary to support the kind of preemption needed by the scheduling algorithms described in Chapter 4, with particular reference to the hybrid approach known as “limited preemption,” and to the store location of the preempted threads context. In order to implement such techniques, the OS design needs to take into account which restrictions will be imposed on the preemptability of the threads,

whether by means of statically defined preemption points, or by postponing the invocation of the scheduling routine by a given amount of time.

### 7.3.3 Migration Support

In migration-based multicore systems, a preempted thread may resume its execution on a different core. Migration support requires additional OS mechanisms to allow threads to be resumed on different cores. Different migration approaches are possible:

- Partitioned approach: Each thread is scheduled on one core and cannot execute on other cores;
- Clustered approach: Each thread can execute on a subset (cluster) of the available cores;
- Global approach: Threads can execute on any of the available cores.

### 7.3.4 Scheduling Characteristics

Real-time scheduling algorithms are often divided into static vs. dynamic scheduling algorithms, depending on the priority assigned to each job to execute. Static algorithms assign a fixed priority to each thread. Although they are easier to implement, their performance could be lower than with more flexible approaches that may dynamically change priorities of each thread. Depending on the scheduling strategy, fixed or dynamic, different OS kernel mechanisms will be needed.

Another design point concerns the policies for arbitrating the access to mutually exclusive shared resources. Depending on the adopted policy, particular synchronization mechanisms, thread queues, and blocking primitives may be needed.

### 7.3.5 Timing Analysis

In order for the timing analysis tools to be able to compute safe and accurate worst-case timing estimates, it is essential that the RTOS manages all the software/hardware resources in a predictable manner. Also, it is crucial for the timing estimates to be as tight as possible because subsequently these values (like the worst-case execution time of a task or the maximum time to read/write data from/to the main memory) will propagate all the way up and will be used as basic blocks in higher-level analyses like the schedulability analysis. Deriving tight estimates requires that all the OS mechanisms that

allocate and arbitrate the access to the system resources are thoroughly documented and do not make use of any procedure that involves randomness or based on non-deterministic parameters.

**Task-to-thread and thread-to-core mapping:** The allocation of the tasks to the threads and the mapping of the threads to the cores must be documented; ideally, it should also be static and known at design time. If the allocation is dynamic, i.e., computed at run-time, then the allocation/scheduling algorithm should follow a set of deterministic (and fully documented) rules. The knowledge of where and when the tasks execute considerably facilitates the timing analysis process, as it allows for deriving an accurate utilization profile of each resource and then uses those profiles to compute safe bounds on the time it takes to access these resources.

**Contract-based resource allocation scheme:** Before executing, each application or task has a "contract" with every system resource that it may need to access. Each contract stipulates the minimum share of the system resource (hardware and software) that the task must be allowed to use over time. Considering a communication bus shared between several tasks, a TDMA (Time Division Multiple Access) bus arbitration policy is a good example of a contract-based allocation scheme: the number of time-slots dedicated to each task in a time-frame of fixed length gives the minimum share of the bus that is guaranteed to be granted to the task at run-time. When the resource is a core, contract-based mechanisms are often referred to as reservation-based scheduling. Before executing, an execution budget is assigned to every task and a task can execute on a core only if its allocated budget is not exhausted. Technically speaking, within such reservation-based mechanisms, the scheduling algorithm of the OS does not schedule the execution of the tasks as such, but rather it manages the associated budgets (i.e., empties and replenishes them) and defines the order in which those budgets are granted to the tasks. There are many advantages of using contract-based mechanisms. For example, they provide a simple way of protecting the system against a violation of the timing parameters. If a task fails and starts looping infinitely, for instance, the task will eventually be interrupted once it runs out of budget, without affecting the planned execution of the next tasks. These budgets/contracts can be seen as fault containers. They guarantee a minimum service to every task while enabling the system to identify potential task failure and avoid propagating the potentially harmful consequences of a faulty task through the execution of the other tasks.

**Runtime budget/contract reinforcement:** Mechanisms must be provided to force the system resources and the tasks to abide with their contract, e.g., a task is not allowed to execute if its CPU budget is exhausted or if its budget is not currently given to that task by the scheduler. This mechanism is known in the real-time literature as "hard reservation."

**Memory isolation:** The OS should also provide mechanisms to dedicate regions of the memory to a specific task, or at least to tasks running on a specific core.

**Execution independence:** The programs on each core shall run independent of the hardware state of other cores.

## 7.4 RTOS Selection

Considering the architecture of the reference platform (i.e., host processor connected to a set of accelerators, similarly to other commercially available many-core platforms), we decided to use two different OSs for the host and the many-core processors.

### 7.4.1 Host Processor

Linux has been chosen for the host processor, due to its excellent support for peripherals and communication protocols, the several programming models supported, and the popularity in the HPC domain.

Given the nature of the project and the requirements of the use-cases, soft real-time support has been added through the adoption of the PREEMPT_RT patch [10].

### 7.4.2 Manycore Processor

For the manycore processor, a proper RTOS was needed. The selected RTOS should have been Open-Source and lightweight (i.e., with a small footprint) but providing a preemptive thread model. For these reasons, only the RTOSs belonging to columns 2 (i.e., Simil-POSIX) and 3 (i.e., OSEK) of Table 7.1 could be selected. Moreover, the selected RTOS must be actively maintained through the support of a development community.

Ecos has been discarded due to the big footprint (comparable to the one of POSIX systems). FreeOSEK has been discarded because the project is not actively maintained and because it does not offer any additional feature

with respect to ERIKA Enterprise. ChibiOS/RT, Trampoline, and QP, instead, have been discarded for the too restrictive open-source license, not suitable for industrial products.

The only RTOSs that fulfilled our requirements, therefore, were ERIKA Enterprise [43] and FreeRTOS [28]. The project eventually chose to use ERIKA Enterprise due to its smaller footprint, the availability of advanced real-time features, and the strong know-how in the development team.

## 7.5 Operating System Support

### 7.5.1 Linux

As for the Linux support, we started with the Linux version provided together with the reference platform. In particular, the Kalray Bostan AccessCore SDK included an Embedded Linux version 3.10, and on top of it we assembled and configured a filesystem based on the Busybox project [46] produced using Buildroot [47].

The Linux version provided included Symmetric Multi-Processing (SMP) support (which is a strong requirement for running PREEMPT_RT [10]), and included the PREEMPT_RT patch.

### 7.5.2 ERIKA Enterprise Support

We have successfully ported the ERIKA Enterprise [43] on the MPPA architecture, supporting its VLIW (Very Large Instruction Word) Instruction Set Architecture (ISA) and implementing the API used by the off-loading mechanism. The following paragraphs list the main challenges we had during the porting, and the main choices we addressed, together with some early performance results.

#### 7.5.2.1 Exokernel support

The development on the platform directly supports the Kalray "exokernel," which is a set of software, mostly running on the 17th core of each cluster (the resource manager core), used to provide a set of services needed to let a cluster appear "more like" a SMP machine. Among the various services, the exokernel includes communication services and inter-core interrupts. The exokernel API is guaranteed to be maintained across chip releases, while the raw support for the resource manager core will likely change with newer chip releases.

#### 7.5.2.2 Single-ELF multicore ERIKA Enterprise

One of the main objectives during the porting of the ERIKA RTOS has been the reduction of the memory footprint of the kernel, obtained by using a Single-ELF build system.

The reason is that the multicore support in ERIKA was historically designed for hardware architectures which did not have a uniform memory region, such as Janus [48]. In those architectures, each core had its own local memory and, most importantly, the view of the memory as seen by the various cores was different (that is, the same memory bank was available at a different address on each core). This imposed the need for a custom copy of the RTOS for each core. Other architectures had a uniform memory space, but the visibility of some memory regions was prevented by the Network on Chip. On Altera Nios II, for example, addresses differentiating by only the 31st bit referred to the same physical address with or without caching. This, again, implied the need for separate images (in particular, you can refer to the work done during the FP6 project FRESCOR, D-EP7 [49]). More modern architectures like Freescale PPC and Tricore AURIX allowed the possibility of single-ELF, but the current multi-ELF scaled relatively well on a small number of cores, reducing the need for single-ELF versions of the system.

In manycore architectures such as Kalray, the multi-ELF approach showed its drawback: the high number of cores, in fact, required avoiding code duplication to not waste memory. Moreover, each core has the visibility of a memory region, and the addressing is uniform across the cores. For this reason, after an initial simple single-core port of ERIKA on the Kalray MPPA, the project decided to eventually design a single-ELF implementation; this activity required a complete rewrite of the codebase (named ERIKA Enterprise v3). The new codebase is now in production and sponsored through a dedicated website [45] in order to gather additional comments and feedbacks. The next paragraphs include a short description of the main design guidelines, which are also described in a specific public document [50].

#### 7.5.2.3 Support for limited preemption, job, and global scheduling

The ERIKA Enterprise RTOS traditionally supported partitioned scheduling, where each core has a set of statically assigned tasks which can be individually activated.

In order to support the features requested by the parallel programming framework, the ERIKA Enterprise scheduler has been modified to allow the following additional features:

- **Limited preemption scheduler** – ERIKA Enterprise has been improved to allow preemptions only at given instants of time (i.e., at task scheduling points, see Chapters 3 and 6). The main advantage is related to performance, because the preemption is implemented in a moment that has a limited performance hit on the system.
- **Job activation** – In ERIKA Enterprise, each task can be individually activated as the effect of an *ActivateTask* primitive. In the new environment, the OS tasks are mapped onto the OpenMP worker threads (see Chapter 6). Those threads are activated in "groups" (named here "jobs"), because their activation is equivalent to the start of an OpenMP offload composed by N OS tasks on a cluster. For this reason, ERIKA Enterprise now supports "Job activation," which allows activating a number of tasks on a cluster. Typically, those tasks will have all the same priority (as they map the execution of an OpenMP offload).
- **Global scheduling** – In order to obtain the maximum throughput, ERIKA implemented a work conserving global scheduler, which is able to implement migration of tasks among cores of the same cluster. The migration support also handles contention on the global queue in case there are two or more cores idle.

#### 7.5.2.4 New ERIKA Enterprise primitives

The implementation of ERIKA Enterprise required the creation of a set of *ad hoc* primitives, which have been included in a new kernel explicitly developed for Kalray. The new primitives are described below:

**CreateJob:** This primitive is used to create a pool of OS tasks which are coordinated for the parallel execution in a cluster. A "Job" is composed by a maximum number of tasks which is equal to the cluster size (16 on Kalray MPPA). It is possible to specify how many tasks should be created, and on which cores they should be mapped in case of partitioned scheduling. All tasks which are part of a Job have the same priority, the same entry point, the same stack size. Finally, they all have an additional parameter which is used by the OpenMP workers to perform their job.

**ReadyJob** and **ActivateJob:** These two primitives are used to put in the ready queue (either global or partitioned depending on the kernel configuration) the tasks corresponding to a specific mask passed as parameter (the mask is a subset of the one passed previously to CreateJob). In addition to this, ActivateJob adds a preemption point on the calling site and issues inter-core interrupts in full preemptive configuration.

**JoinJob:** This is a synchronization point at the termination of all tasks of a Job. It must be called on a task which has lower priority than the Job task priority.

Synchronization primitives are also provided to allow the implementation of use-level locks and higher-level programming model synchronization constructs for the OpenMP runtime library (discussed in Chapter 6).

**SpinLockObj** and **SpinUnlockObj:** These primitives provide a standard lock API, and are directly based on spinlock primitives provided by the Kalray HAL. At the lowest abstraction level, the lock data structure is implemented as a 32-bit integer, which could be allocated at any memory-mapped address. Using this approach, the lock variables can be statically allocated whenever it is possible, and when more dynamism is required, lock data structures can be initialized via standard malloc operations on a suitable memory range.

**WaitCondition** and **SignalValue:** These primitives provide a synchronization mechanism based on WAIT/SIGNAL semantics. ERIKA supports four condition operators (equal, not equal, lower than, greater than) and three different wait policies:

1. BLOCK_NO – The condition is checked in a busy waiting loop;
2. BLOCK_IMMEDIATELY – The condition is checked once. If the check fails (and no other tasks are available for execution) the processor enters sleep mode until the condition is reached. A specific signal is then used to wake-up the processor.
3. BLOCK_OS – Informs the OS that the ERIKA task (i.e., the OpenMP thread mapped to that task) is voluntarily yielding the processor. The OS can then use this information to implement different scheduling policies. For example, the task can be suspended and a different task (belonging to a different job) can be scheduled for execution.

#### 7.5.2.5 New data structures

Addressing the single-ELF image implementation in the end required a restructuring of the kernel data structures.

The initial version of ERIKA Enterprise used a set of global data structures (basically, C arrays of scalars) allocated in RAM or ROM. Each core had its own copy of the data structures, with the same name. Data which is shared among the cores is defined and initialized in one core referred to as the *master* core. The other cores are called *slave* cores. Afterwards, when compiling the *slave* cores' code, the locations of the shared data are appended to each

core's linker scripts (see also [48]). Figure 7.2 shows the structure of the two ELF files, highlighting the first core (*master*), which has everything defined, and the subsequent *slave* cores, which have the shared symbols addresses appended in the linker script.

The single-ELF approach required a complete restructuring of the binary image. The complete system is compiled in a single binary image, and the data structures are designed to let the cores access the relevant per-CPU data. The main guidelines used when designing the data structures are the following:

- All data is shared among all cores.
- The code must be able to know on which core it is running. This is done typically using a special register of the architecture that holds the CPU number.
- Given the CPU number, it is possible to access "private" data structures to each core (see Figure 7.3). Note that those "private" data structures can be allocated in special memory regions "near" each core (for example, they could be allocated in sections which can be pinned to per-core caches).
- Clear distinction between Flash Descriptor Blocks (named *DB) and RAM Control Blocks (named *CB). In this way the reader has a clear idea of the kind of content from the name of the data structure.
- Limited usage of pointers (used to point only from Flash to RAM), to make the certification process easier.

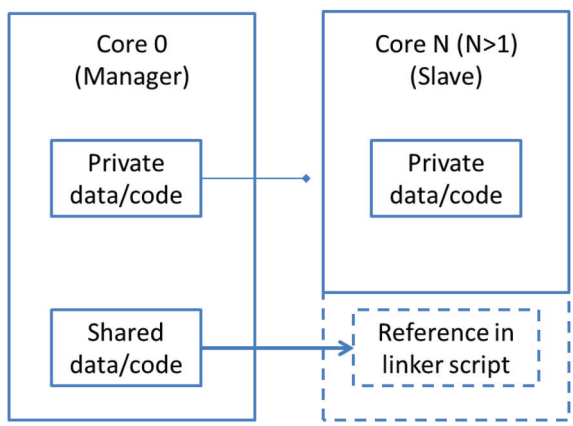


**Figure 7.2** Structure of the multicore images in the original ERIKA Enterprise structure.

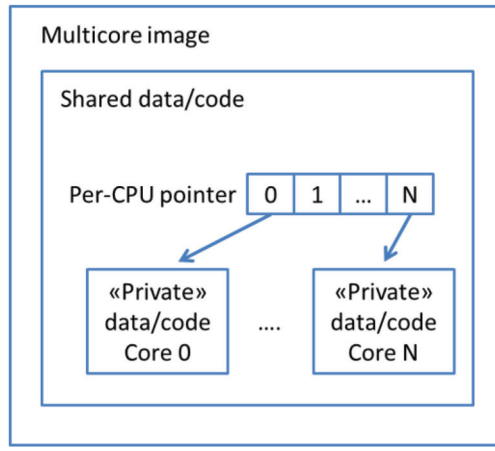


**Figure 7.3** Structure of the Single-ELF image produced by ERIKA Enterprise.

#### 7.5.2.6 Dynamic task creation

In the original version of ERIKA, RTOS tasks were statically allocated by defining them inside an OIL file. In the new version of ERIKA, we allowed a pre-allocation of a given number of RTOS tasks, which can be afterwards "allocated" using a task creation primitive. In this way, the integration with the upper layers of OpenMP becomes simpler, as OpenMP makes the hypothesis of being able to create as many threads as needed using the underlying Linux primitive pthread_create.

In addition to the changes illustrated above, we also took the opportunity for making the following additional changes to ease future developments.

#### 7.5.2.7 IRQ handlers as tasks

The original version of ERIKA handled interrupts in the most efficient way in the case of no memory protection among tasks. When memory protection comes into play, treating IRQs as special tasks has the advantage of simplifying the codebase.

In view of the future availability of multi-many cores with memory protection we implemented the possibility for an IRQ to be treated as a task. A special fast handler is called upon IRQ arrival, which has the main job of activating the "interrupt task."

This approach also simplified the codebase by allowing a simpler context change primitive, which in turn simplifies the implementation in VLIW chips such as Kalray.

#### 7.5.2.8 File hierarchy

For the new version of ERIKA, we adopted a new file hierarchy which aims to a simplification of the codebase. In particular, the main changes of the new codebase are the following:

- In the old version, CPU (the specific instruction set, such as PPC, Cortex-MX, etc.), MCU (the peripherals available on a specific part number), Boards (code related to the connections on the PCB) were stored in directories under the "pkg" directory. With the growing number of architectures supported, this became a limitation which also made the compilation process longer. The new version of the codebase includes MCUs and Boards under the CPU layer, making the dependencies in the codebase clearer.
- We adopted a local self-contained flat (single directory) project structure instead of a complex hierarchy. All needed files are copied once in the project directory at compilation time, leading to simpler makefiles.
- We maintained the RTOS code separated from the Application configuration. This is very useful to allow the deployment of pre-compiled libraries; moreover it allows partial compilation of the code.

#### 7.5.2.9 Early performance estimation

Before implementing the Single-ELF version of ERIKA on Kalray, we performed an initial implementation of the traditional single-core porting of ERIKA in order to get a reference for the evaluation of the subsequent development. Please note that the evaluation of the new version of ERIKA has been done on a prototype implementation (not the final one). However, the numbers are good enough to allow a fair comparison of the two solutions.

Table 7.2 summarizes an early comparison between the old and the new implementation of ERIKA, for a simple application with two tasks on a single core. The purpose of the various columns is the following:

- The comparison between the second and the third column gives a rough idea of the difference in the ISA on a "reasonably similar" code on another (different) architecture, Nios II.
- The comparison between the third and the fourth column gives a rough idea of the impact of the changes of the new version of ERIKA over the old version. The values show an increase of the code footprint. This increase, however, is less than indicated by the table: the old version of ERIKA, in fact, does contain the support for multiple task activations (which has not been compiled) and dynamic task creation (which was

**Table 7.2** ERIKA Enterprise footprint (expressed in bytes)

| Description | Old Version (*) | Old Version | New Version Single-core | New Version, Multicore with Services for Supporting Libgomp |
|---|---|---|---|---|
| Platform | Nios II | Kalray MPPA | Kalray MPPA | Kalray MPPA |
| Code footprint (**) | About 800 | 1984 | 2940 | 4156[1] |
| Code footprint related to multicore (***) | | – | – | 4156 + 502 for RM |
| Flash/Read-only | 164 | – | – | – |
| RAM | | 192 | 216 | 216 + 128 for each core[2] |

(*) Numbers taken from D-EP7v2 of the FRESCOR project [49]. These numbers can be taken as a reference for the order of magnitude for the size of the kernel and may not represent the same data structures. We considered these numbers as the current implementation on ERIKA has roughly a similar size and they can be used as a reference for comparing "similar" implementations.
(**) The code footprint includes the equivalent of the following functions: *StartOs*, *ActivateTask*, *TerminateTask*
(***) Code related to the handling of the multicore features (remote notifications, inter-processor interrupts, spin locks, and code residing) on the Resource Manager Core (see Chapter 2).

not available). Moreover, we have to consider that the old version of ERIKA needed 1,984 bytes for each core. The new version of ERIKA, instead, needs 2,940 bytes, regardless of the number of cores. This means that with just two cores, the amount of memory needed by the new version of ERIKA Enterprise is less than using the old version of the RTOS.

- The comparison between the fourth and the fifth column gives a rough idea of the impact of the multicore support. The increase of the code footprint is mainly due to additional synchronization primitives (i.e., spinlocks) needed for distributed scheduling – i.e., to allow the "group activation" done by the Resource Manager on behalf of OpenMP. Therefore, this increase is specific to the Kalray architecture, and it is missing on other (e.g., shared-memory) architectures. Note that the footprint takes into account only the kernel part with the services for supporting the OpenMP runtime library; it does not include the library itself.

[1] The footprint takes into account only kernel and support for the OpenMP runtime library; it does not include the library itself.

[2] 128 = 44 (core data structures) + 84 (idle task).

**Table 7.3** Timings (expressed in clock ticks)

| Feature | Time on ERIKA |
|---|---|
| *ActivateTask*, no preemption | 384 |
| *ActivateTask*, preemption | 622 |
| An IRQ happens, no preemption | 585 |
| An IRQ happens, with preemption | 866 |

Table 7.3 provides basic measurements of activation and pre-emption of tasks on a single-core:

Tables 7.4 and 7.5 provide some timing references to compare ERIKA Enterprise (which is a RTOS) with NodeOS on MPPA-256, taken using the Kalray MPPA tracer. Since NodeOS does not support preemption (and therefore a core can execute only one thread) we have configured ERIKA Enterprise to run only one task on each core as well. Then, we have measured footprint and execution times. In particular, Table 7.4 provides a rough comparison of the footprint for ERIKA and NodeOS on Kalray MPPA. For ERIKA, the footprint also takes into account the per-core and per-task data structures in a cluster composed of 17 cores. This footprint can be reduced by using a static configuration of the RTOS. Table 7.5 provides a comparison between the thread creation time on NodeOS and the equivalent inter-core task activation on ERIKA.

**Table 7.4** Footprint comparison between ERIKA and NodeOS for a 16-core cluster (expressed in bytes)

| Description | ERIKA New Version, Multicore with Services for Supporting OpenMP | NodeOS |
|---|---|---|
| Code footprint | 4484[3] | 10060 |
| RAM | 2184 | 2196 |

**Table 7.5** Thread creation/activation times (expressed in clock ticks)

| Inter-core Task Activation on ERIKA | Thread Creation on NodeOS |
|---|---|
| 1200 | 3300 |

[3]The footprint takes into account only kernel and support for libgomp; it does not include the whole libgomp library.

## 7.6 Summary

This chapter illustrated the state of the art of the OSs suitable for the reference parallel programming model. After reviewing the main requirements that influenced the implementation, the selection of the RTOS for the reference platform has been described for both the host processor and the manycore accelerators. Furthermore, a description of the main implementation choices for the ERIKA Enterprise v3 and Linux OS have been detailed. As can be seen, the result of the implementation provides a complete system which is capable of addressing high-performance workloads thanks to the synergies between the general-purpose OS Linux and the ERIKA Enterprise RTOS.

## References

[1] *Top500, Linux OS*. Available at: http://www.top500.org/statistics/details/osfam/1
[2] *RCU*, available at: http://en.wikipedia.org/wiki/Read-copy-update
[3] *GNU General Public License (GPL)*, available at: https://www.gnu.org/copyleft/gpl.html
[4] Lipari, G., Scordino, C., *Linux and Real-Time: Current Approaches and Future Opportunities, International Congress ANIPLA*, Rome, Italy, 2006.
[5] *RTLinux*, available at: http://en.wikipedia.org/wiki/RTLinux
[6] *RTAI – the RealTime Application Interface for Linux*, available at: https://www.rtai.org/
[7] Xenomai, *Real-Time Framework for Linux*. Available at: http://www.xenomai.org/
[8] Windriver, *VxWorks RTOS*. Available at: http://www.windriver.com/products/vxworks/
[9] *POSIX IEEE standard*, available at: http://en.wikipedia. org/wiki/POSIX
[10] *The Real Time Linux project*, available at: https://wiki.linuxfoundation.org/realtime/start
[11] *The Linux Foundation*, available at: https://www.linuxfoundation.org/
[12] Libenzi, D., *SCHED SOFTRR Linux Scheduler Policy*, available at: http://xmailserver.org/linux-patches/softrr.html
[13] Kolivas, C., *Isochronous class for unprivileged soft RT scheduling*. Available at: http://ck.kolivas.org/patches/
[14] *SCHED_DEADLINE Linux Patch*, available at: http://en.wikipedia.org/wiki/SCHED_DEADLINE

[15] Lelli, J., Scordino, C., Abeni, L., Faggioli, D., Deadline scheduling in the Linux kernel, Software: Practice and Experience, 46, pp. 821–839, 2016.
[16] *ACTORS European Project*, available at: http://www.actors-project.eu/
[17] *Earliest Deadline First (EDF)*, available at: http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling
[18] Green Hills, *Integrity RTOS*. Available at: http://www.ghs.com/products/rtos/integrity.html
[19] *QNX RTOS*, available at: http://www.qnx.com/
[20] *SYSGO PikeOS*, available at: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/
[21] Mentor Graphics, *Nucleus RTOS*. Available at: http://www.mentor.com/embedded-software/nucleus/
[22] *LynuxWorks LynxOS*, available at: http://www.lynuxworks.com/rtos/
[23] *Micrium µc/OS-III*, available at: http://micrium.com/
[24] *Segger EmbOS*, available at: http://www.segger.com/embos.html
[25] *ENEA OSE*, available at: http://www.enea.com/ose
[26] *Arccore Arctic Core*, available at: http://www.arccore.com/products/
[27] Wikipedia, *List of Real-Time Operating Systems*, available at: http://en.wikipedia.org/wiki/List_of_real-time_operating_systems
[28] *FreeRTOS*, available at: http://www.freertos.org/
[29] *GPL Linking Exception*, available at: http://en.wikipedia.org/wiki/GPL_linking_exception
[30] *Contiki*, available at: http://www.contiki-os.org/
[31] Wikipedia, *Protothreads*. Available at: http://en.wikipedia.org/wiki/Protothreads
[32] *Marte OS*, available at: http://marte.unican.es/
[33] *Ecos RTOS*, available at: http://ecos.sourceware.org/
[34] *FreeOSEK RTOS*, available at: http://opensek.sourceforge.net/
[35] Quantum Leaps, *QP$^{TM}$ Active Object Frameworks for Embedded Systems*, available at: http://www.state-machine.com/
[36] *Trampoline RTOS*, available at: http://trampoline.rts-software.org/
[37] *RTEMS*, available at: http://www.rtems.org/
[38] *Footprint of RTEMS*, available at: http://www.rtems.org/ml/rtems-users/2004/september/msg00188.html
[39] *Tiny RTEMS*, available at: https://code.google.com/p/tiny-rtems/
[40] *TinyOS*, available at: http://www.tinyos.net/
[41] *ChibiOS/RT*, available at: http://www.chibios.org/
[42] *Tivoization*, available at: http://en.wikipedia.org/wiki/Tivoization

[43] *Erika Enterprise RTOS*, available at: http://erika.tuxfamily.org
[44] *Automotive Linux Summit Fall*, available at: http://events.linuxfoundation.org/events/automotive-linux-summit-fall
[45] *ERIKA Enterprise v3*, available at: http://www.erika-enterprise.com
[46] *Busybox*, available at http://www.busybox.net/, last accessed March 2016.
[47] *Buildroot*, available at http://buildroot.uclibc.org/, last accessed March 2016.
[48] Ferrari, A., Garue, S., Peri, M., Pezzini, S., Valsecchi, L., Andretta, F., and Nesci, W., "The design and implementation of a dual-core platform for power-train systems." In *Convergence 2000*, Detroit, MI, USA, 2000.
[49] *FRESCOR FP6 D-EP7v2*, available at http://www.frescor.org/ and also http://www.frescor.org/ and also http://erika.tuxfamily.org/wiki/index.php?title=Altera_Nios_II, last accessed March 2016.
[50] Evidence, *ERIKA Enterprise Version 3 Requirement Document*, available at ERIKA Enterprise website: http://erika.tuxfamily.org/drupal/content/erika-enterprise-3