

xLuna: a Real-Time, Dependable Kernel for Embedded Systems

ABSTRACT

While the GNU/Linux Operating System (OS) is gaining popularity in research and student communities as well as in the business world, its impact is still limited in all those application areas requiring hard real-time capabilities, extreme robustness and a minimal trusted computing base.

In this paper we present a software architecture featuring a portable user-mode version of Linux which runs on top of the Real-Time Operating System (RTOS) RTEMS. Our system, called *xLuna*, is composed of three parts: RTEMS, a user-mode version of Linux and a small software layer exposing virtual resources for Linux. We show that xLuna is able to provide real-time capabilities, predictability and reliability properties while maintaining the versatility, ease of use and large application base of Linux.

1. INTRODUCTION

Until recently, the vast majority of embedded systems has been based on simple processors mainly used for performing relatively simple control operations, while the rest of the system's functionalities has been implemented using dedicated hardware components. However, there is a strong trend from these "classical" embedded systems towards more powerful platforms. The major driving force lies in the demand for more sophisticated and flexible architectures. Considering the tight timing constraints imposed by today's market, such complexity can only be achieved by implementing a consistent part of the system's functionalities as software components.

As a consequence, such devices have peculiar requirements, quite different from those of classical embedded architectures: high demand for processing capabilities, and a standardized and well-known application programming interface (API). These requirements are well supported by contemporary desktop and server operating systems such as Linux, and it is not surprising that there is a strong trend towards their use in embedded systems. The disadvantage of these kernels is that they cannot usually guarantee response times

for their processes. Unfortunately, there are application areas (such as robotic devices, healthcare and military computers) where a hard real-time response is of critical importance. Hybrid operating systems, with both real-time and time-sharing components, are starting to appear on the market as a solution to these problems. They have many important benefits such as a comfortable, well-known interface compatible with lots of existing applications, and the ability to run real-time and standard non real-time applications at the same time on the same machine, without impairing the real-time predictability.

Another issue which limits the use of standard desktop OSes in embedded systems is that they are often employed in life-critical or mission-critical scenarios. While the reliability of Linux on desktops and servers is generally very high, this typically applies to systems with widely adopted configurations. Massive changes to the system configuration, as it is often necessary for an embedded system, inherently reduce stability and require a significant maturation process. At the same time, the critical parts of the system should not be affected by the other components which are only employed to support non-critical functions. Hybrid operating systems with separate addressing spaces address this problem.

In this paper we present *xLuna*, a reliable RTEMS/Linux kernel. This work was motivated by the need for a runtime environment targeting real-time and non-real-time applications and providing high reliability, real time behavior and the support for a large application base. Next to these features, standard programming interfaces (e.g. POSIX), an extensive set of libraries, and a familiar environment, requiring a minimal learning curve for either creating new applications or migrating/integrating existing components, are needed. By running the Linux kernel in unprivileged mode, this solution gathers the benefits of using a well known desktop OS while assuring fault isolation and real-time properties.

The xLuna Kernel is composed of two main sub-systems: RTEMS, running hard real-time tasks (HRTs) in "privileged-mode" and the Linux Kernel, running non real-time tasks (NRTs) and executing in restricted mode, i.e. "user-mode". This approach enables the Linux sub-system to be activated and deactivated at any time, and it also ensures protection from its erroneous behaviour. To minimize the dependencies from particular RTEMS versions, modifications of its source code were avoided. Indeed, most of the xLuna features are

made available by replacing the OS interrupt/trap handling routines at runtime. Memory protection requirements are achieved by a technique called *supervised memory management* [15]. This technique allows Linux to manage virtual memory, with every access through the Memory Management Unit (MMU) being monitored by xLuna. This way, memory protection of RTEMS from Linux is guaranteed even in the case of corruption of the Linux kernel.

This paper is organized as follows: Section 2 presents the state of the art in terms of hybrid operating systems. The xLuna kernel is described in detail in Section 3 and results about its performance are contained in Section 4. Finally, Section 5 concludes the paper and describes some additional future developments.

2. RELATED WORK

Leslie et al. [8] present in detail the motivations which push towards an adoption of Linux in the embedded market. They also identify the main challenges which still prevent its wide diffusion for this kind of devices:

1. Embedded systems are mostly real-time, meaning that they have to respond to external events in a timely fashion. In spite of very significant progress in Linux's real-time responsiveness, it is not yet suitable for hard real-time systems
2. Embedded systems are often employed in life-critical or mission-critical scenarios. The high reliability necessary in these situations cannot be guaranteed by Linux which, even if fairly stable in standard environments, is not mature enough to be trusted after massive changes in its configuration
3. The need for a small *Trusted Computing Base (TCB)*: the set of components which must operate correctly to allow the proper behavior of the system has to be as small as possible. Since all kernel code has access to the overall system, the kernel has to be part of the TCB. Unfortunately, the Linux kernel is too large to establish the degree of trust that is required for many embedded applications

Wombat, a port of the Linux kernel on top of the L4 [14] microkernel, is presented as a solution to these issues. This system, like ours, is designed to support real-time applications. Linux programs can be executed in either native or compatibility mode: the former means that the applications share the address space with the subsystem providing trusted services (called Iguana). Such a policy is dangerous in that a problem in those programs can cause a failure of the whole Iguana. Linux processes are directly managed by L4 by using low scheduling priorities so that real time properties of critical tasks are maintained. The biggest drawback of this system consists in the system-call slow down, around 3.5 time with respect to a standard version of Linux. On the other hand, the performance of the context switch operation is not noticeably affected. Our system enforces a complete separation between Linux and RTEMS, meaning that the RTEMS scheduler is not aware of Linux tasks. We also do not share any memory portion between RTEMS, Linux and xLuna.

RTLinux [5] was the first project to provide real-time in Linux using two separate OSs: Linux as the general purpose OS and a newly developed executive running beneath Linux. A good example of the synergy between RTLinux and Linux for hard and soft real-time applications can be found in [9] where a Linux thread is executed as a constant bandwidth server (CBS) at the RT-Linux level, providing a guaranteed use of the CPU. Other works [7] go in the same direction by applying several patches to the Linux kernel in order to improve its real-time capabilities and reduce its latency, thus making it suitable for soft realtime applications.

RTAI (RealTime Application Interface for Linux) [4] is a Linux kernel modified to have real-time properties. A small real-time kernel is inserted under the standard Linux OS, treating Linux as the idle task that is scheduled only when no other real-time task is ready. The main issue of this work, which often create problems for its adoption in the embedded world are: (a) a high cyclomatic complexity [11] which complicates its testing, validation, certification and maintenance, and (b) robustness in front of misuse of the OS core routines. Also, the "Linux-as-idle-task" approach used by RTAI is protected by the RTLinux patent. To overcome this drawback, recent versions of RTAI implement the interrupt pipeline approach of the ADEOS nanokernel [1] thus being free from the RTLinux patent problem. xLuna does not suffer this issue as it follows a completely different approach, adding a general purpose operating system support to a RTOS.

As described in [8] and as implemented in xLuna, separate address spaces can be effectively used to guarantee that failures in the non-critical part of the system do not affect the TCB. The work presented in [13] determines the cost of having separate address spaces with respect to shared space systems. Former experiments compare L4RTL [12], a combination of the L4 microkernel and of Linux, with RTLinux. While the reliability of the whole system is consistently increased with L4RTL, the authors did not experience any significant slowdown due to the separation of the memory addresses.

Other attempts of combining Linux with a real-time kernel include MaRTE-OS [10], a real-time kernel for embedded applications providing both C and Ada interfaces, coupled with Linux in two different ports. In the first, MaRTE-OS executes as part of the Linux kernel, while in the second it is executed as a user process. Our work goes in the opposite direction in that we integrate Linux inside a RTOS and not viceversa.

In addition to maintaining real-time properties, as shown in the following, our system has complete independence and isolation of the Linux and RTEMS subsystems, something not completely true for the presented approaches ([8, 10]) This has been achieved by executing Linux as an RTEMS thread and by implementing tight memory protection mechanisms.

3. xLUNA

The solution introduced by xLuna (shown in Figure 1) is based on two sub-systems. The first one provides support for Hard Real-Time (HRT) tasks, the second one provides a

well-known programming interface with an extensive set of libraries. The advantage in having two isolated systems is that it is possible to have applications with different criticality levels running on the same system. Thus, non-critical components are quickly developed or simply ported from a previous implementation, resulting in a reduction of time-to-market and budget. Furthermore, because the HRT system is protected by the Memory Management Unit (MMU) from the Non-Real-Time (NRT) system, a heavy validation and verification test campaign on non-critical components could be reduced or even avoided. This separation allows an approach where the Linux sub-system can be activated and deactivated at any time and, at the same time, it ensures protection from erroneous behaviour of the Linux kernel. xLuna is currently built to support the LEON 2 processor and it uses Snapgear Embedded Linux [2] as the Linux sub-system, since this OS contains a port for an MMU enabled LEON processor.

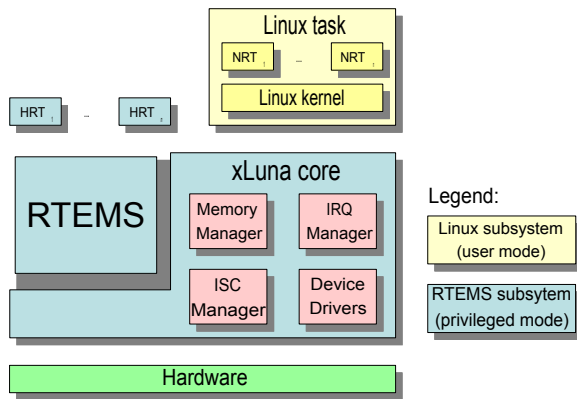


Figure 1: Overview of the xLuna architecture: xLuna and RTEMS are run in the same privileged environment, while Linux is executed in user-mode as the RTEMS idle thread

xLuna, as described in the Section 3.3.1, organizes and manages the memory for the whole system. The 4GB space addressable by the LEON processor is allocated to different devices, including RAM, PROM, bus I/O, on-chip registers, etc. The available RAM space is, in turn, divided into two parts, one for RTEMS kernel and its tasks and the other for the use by Linux kernel and Linux processes. RTEMS occupies the first chunk of memory with a fixed configurable size and the rest is allocated to Linux.

The communication and synchronization of HRT tasks with NRT processes takes place through the Inter-Systems Communication (ISC) module, as described in Section 3.3.4. This module provides synchronous/asynchronous bi-directional communication by using message queues. RTEMS tasks interact with it through direct calls to xLuna primitives, while Linux processes read/write to a character device.

3.1 Linux Subsystem

The modifications that have been performed to the Linux kernel source code are reduced to a minimum to ensure portability between different versions. All the implemented patches are mainly due to the fact that, in our system, Linux is not executed on bare hardware and with supervisor priv-

ileges. Instead, it runs inside a software shell (xLuna) as a user-mode process. The main challenge of this approach is to maintain, through the virtualization of the underlying hardware, the low-priority Linux system unaware of its execution environment.

In particular, to guarantee the real time properties of RTEMS and to ensure that failures in the Linux kernel do not affect the rest of the system, all direct hardware accesses was replaced with communication events triggering the hardware management primitives of the xLuna and RTEMS subsystems. This way, it is only necessary to make sure that xLuna routines behave as expected in order to guarantee that all operations on the hardware are properly performed. In addition, memory is not directly managed by the Linux kernel but, to achieve memory protection, Linux only reads/writes to the RAM portion which is owned by Linux itself. This can be achieved using the mechanisms described in Section 3.3.1.

As previously described, communication among the two worlds takes place using the Inter-Systems Communication module. To allow Linux processes to access to the ISC, a new kernel device was created: user processes can access the interface exported by this driver and easily perform inter-systems communication through regular file operations.

At system startup, the Linux subsystem is not started automatically but it is launched by calling an xLuna primitive inside RTEMS applications. This approach was adopted in order to allow the freedom of determining when Linux boots. Furthermore, in case of Linux kernel panic, a previously registered handler can simply restart the NRT environment.

Apart from these modifications, all the usual functionalities and libraries of Linux remain untouched and, from the point of view of the normal application developer, the modifications are unnoticeable. In particular the standard glibc library and the POSIX-Thread interface are available for writing applications. Additional libraries can also be added as in a normal Linux distribution.

3.1.1 Scheduling Policies

Linux tasks are scheduled by the Linux kernel using its original unmodified scheduler. This because Linux is executed as an RTEMS idle thread: the whole kernel and its processes are physically executed on the processor only when no HRT tasks are running, as shown by Figure 2.

3.2 RTEMS Subsystem

RTEMS (Real-Time Executive for Multiprocessor Systems) [3] is a free open source real-time operating system designed for embedded architectures; it is mainly targeted to critical and military applications. RTEMS does not provide any form of memory management or support for processes; in POSIX terminology, it implements a single process in a multithreaded environment. This is reflected in the fact that RTEMS provides nearly all POSIX services other than those which are related to memory mapping, process forking, or shared memory.

RTEMS was chosen as the basic block on top of which the whole xLuna infrastructure is built because it is a reliable operating system with well known efficiency, quality and stan-

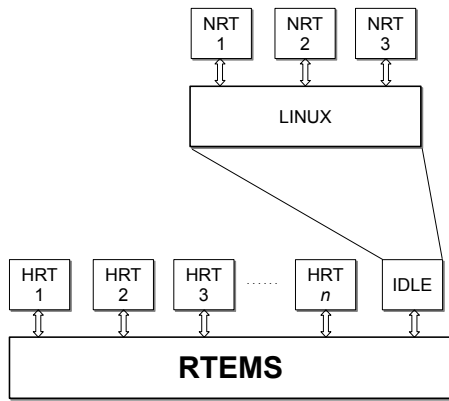


Figure 2: Task scheduling in xLuna

standard compliance. Its maintainability and robustness as well as its high performance, provide the necessary characteristics for the utilization in critical embedded systems.

The development on the RTEMS side focuses on the implementation of new modules that support the execution of the underlying low-priority Linux kernel (more details are contained in Section 3.3). To minimize RTEMS version dependency, the xLuna project avoids modifications to the RTEMS source code whenever possible. Indeed, most of the xLuna features are made available by replacing the interrupt/trap handling routines of the RTEMS kernel, only when explicitly required by a user task. The original RTEMS handlers can also be restored when necessary. This is possible since all the code in the RTEMS subsystem is running in privileged mode. This way, the original RTEMS features are kept intact when Linux is not needed or when the xLuna kernel is switched to a safe mode where only HRT functionalities of the system are guaranteed. Furthermore, with the adopted design, the xLuna kernel can support future RTEMS releases with minimum effort.

All RTEMS real-time characteristics are present in xLuna, as its kernel was not modified.

3.3 xLuna Subsystem

Figure 3 shows the xLuna subsystem which is mainly composed of four submodules, managing the interactions between RTEMS and Linux and between these sub-systems and the hardware. These are the *Memory Manager*, the *IRQ Manager*, the *Inter-Systems Communication (ISC) Manager* and the *Device Drivers*. These four modules are the heart of xLuna operating system and run in privileged mode along with RTEMS.

3.3.1 Memory management

Modern operating systems use, among other architecture-specific approaches, the paging mechanism in order to achieve memory sharing and memory protection. Paging maps virtual memory into physical memory on a page-by-page basis with different access rights for each page. This mechanism usually requires support from the underlying hardware, which in the xLuna case, is the *Memory Management Unit (MMU)* module. In particular xLuna implements a technique called *Supervised Memory Management* using the

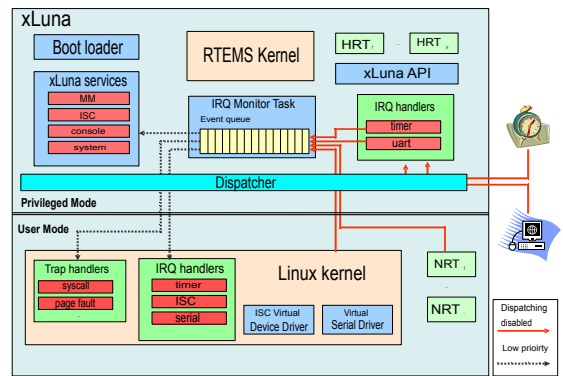


Figure 3: Internal structure of the xLuna subsystem

MMU. This technique allows Linux to manage virtual memory, with every MMU access being monitored by xLuna. This way, memory protection of RTEMS from Linux is guaranteed even in the case of corruption of the Linux kernel. To achieve memory protection, Linux can only access memory which is owned by Linux itself and, in the same way, the RTEMS kernel and its tasks can only access their own RAM portion. The xLuna extensions, running in protected mode, have access to both RTEMS-owned and Linux-owned memory. To this end, the virtual address space is partitioned into three sections, as shown in Figure 4:

- An RTEMS window for RTEMS and xLuna extensions to access all the physical RAM, ROM, and I/O addresses.
- A Linux kernel window for the Linux-owned physical RAM and ROM address spaces.
- A Linux process window to be used by Linux processes.

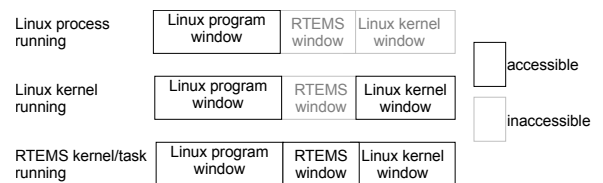


Figure 4: Virtual memory organization in the xLuna system

Different access rights for the different memory regions are used to enforce memory protection:

- The RTEMS window is always marked for kernel mode access, so that neither Linux processes nor the Linux kernel can have access to it.
- The Linux kernel window is marked as invalid when a Linux process is running, protecting the Linux kernel from its own processes. The invalid bit is restored when a trap/interrupt occurs. This way the Linux kernel, even though running with user mode privileges, still has the usual protection from user processes.

- The Linux process window is always accessible from all the sub-systems

Figure 5 shows the memory map and the ownership of each memory region. The memory region visible in the RTEMS-owned window contains all of the memory-mapped physical resources. The RTEMS kernel and its tasks use this address space to access the RTEMS-owned RAM portion and all the other hardware resources, as if they were running in an environment without MMU (RTEMS is designed for such an environment). The RTEMS window is also used by the xLuna extensions to access the Linux-owned memory portion. The Linux kernel window is mapped to all the Linux-owned memory address space but it does not allow access to any other memory region. This design protects the global address space from Linux so that this sub-system is unable to access physical devices directly. The Linux process window is used by the Linux kernel to allocate virtual address space for its processes, and each of them sees a different memory window as normal.

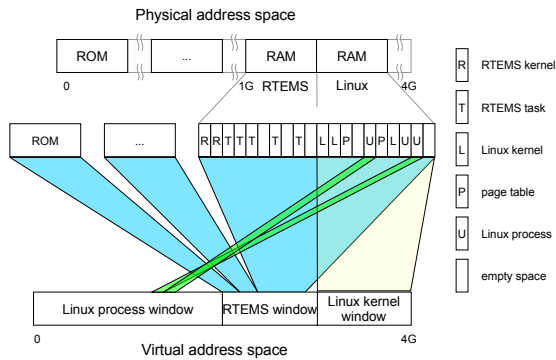


Figure 5: Detailed mapping from virtual address space to physical address space

An important factor in the memory management design is the placement and management of the page tables, which implements the above address space division and access rights management. Allowing Linux to manage all page tables might compromise system integrity as uncontrolled changes could make any physical memory space present in the hardware platform visible. This means that without employing any privileged CPU instruction, failures in the Linux kernel can lead to memory corruption in the RTEMS space. However, it is also not desirable to port the whole Linux memory management code into RTEMS, as the page table management logic in Linux is very complex and it is highly coupled with other Linux kernel functions like file system and task scheduling. The approach proposed here is to let Linux manage page tables, but under the supervision of RTEMS. In this approach, Linux memory management logic remains unchanged, but the memory containing page tables is made read-only to Linux. Low-level page table interfaces, that change the virtual to physical memory mapping, are re-implemented to invoke xLuna services that verify the validity of the modifications and apply them.

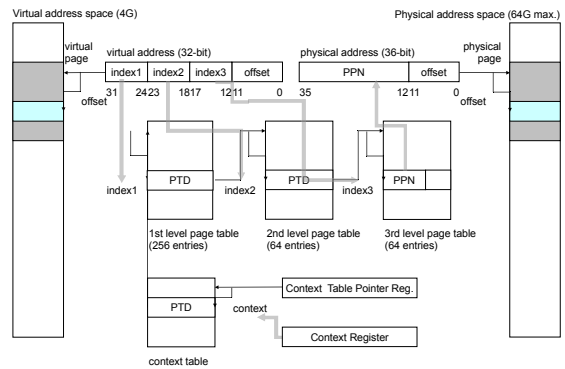


Figure 6: Memory layout and mapping in the xLuna architecture

3.3.2 Interrupt handling

As previously described, Linux runs as an RTEMS regular task with low privileges and no direct connection to the underlying hardware. This situation is contrary to the basis with which that OS is designed and it introduces some difficulties, mainly catching hardware interrupts and traps generated by user processes.

The xLuna kernel provides *bridging services* in order to connect the Linux interrupt handlers with the physical interrupt/trap sources. Since during interrupt and trap handling the dispatch of Hard Real-Time Tasks is disabled, the amount of time spent for this process is kept to a minimum. As a result, only a minimal amount of work is done inside the interrupt service routine of xLuna. The actual work is deferred and carried out later by a low priority task called Dispatcher.

More in detail, the main features provided by the *bridging service* module are:

- Managing of the incoming hardware interrupts by intercepting them and by pipelining them to the corresponding RTEMS handlers;
- Catching of all traps from the Linux subsystem and redirecting to the correct handlers. These include Linux system calls made by Linux processes, xLuna system calls made by the Linux kernel, and other traps like page faults.
- Allowing Linux kernel to virtually disable interrupts during its execution: in this case xLuna will not deliver the disabled interrupt to the Linux subsystem.
- Managing time synchronization for the Linux subsystem;
- Providing an RTEMS API for managing the Linux subsystem

All illegal traps from the NRT system (Linux kernel and applications) are handled by xLuna kernel so that HRT system remains protected. Hardware interrupts are first processed

by xLuna and all interrupts for the Linux kernel are queued and passed to it later on.

3.3.3 Device drivers

The device drivers module allows Linux to access additional hardware that is not managed neither by the memory manager, nor by the IRQ manager. For now, they provide access to timer and UART. These two devices are shared between RTEMS and Linux and xLuna is in charge of queuing the interrupts and of distributing them to the correct subsystem.

Hardware devices can also be exclusively assigned to Linux using one of two mechanisms: (a) direct access to the device address space in case they are passive devices and (b) management through xLuna for active devices.

3.3.4 Inter-Systems Communication

In xLuna, communication between HRT and NRT tasks is possible through a message queue, as depicted in Figure 7. This message queue is implemented by the Inter-Systems Communication (ISC) manager. It provides synchronous and asynchronous bi-directional communication. Actually, this message queue consists of an RTEMS message queue. Communication on the RTEMS side is implemented directly using message queue primitives. On the Linux side, a kernel device driver is provided for inter-systems communication, which uses system calls to access the xLuna kernel services, that in turn, call the RTEMS message queue management routines. No direct access to the queue is allowed from Linux to avoid corruption of the RTEMS side of the system. Linux user processes can access the interface exported by the kernel device driver and perform inter-systems communication through regular file operations.

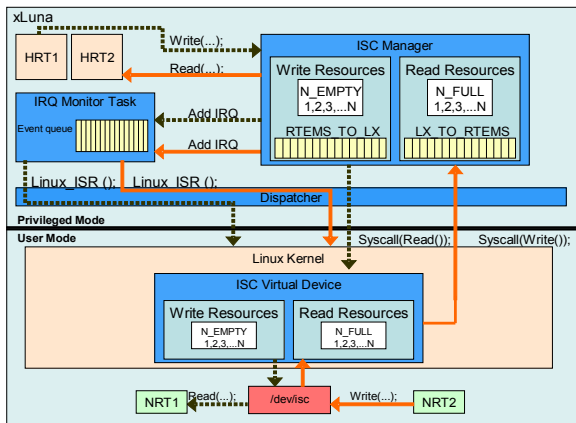


Figure 7: Detailed architecture of the Inter-System Communication mechanism

As Linux is regarded as a task of RTEMS, blocking a Linux process waiting for some message from RTEMS could naively block the whole Linux kernel from rescheduling. This is certainly not a desired behaviour since there might be other Linux processes ready to run. In xLuna, system calls from the Linux subsystem to read/write messages are only issued when resources for these operations are actually available. Algorithm 1 explains in detail this behavior.

```

begin linux_task
  ...
  xluna_read_isc ()
  ...
end

begin xluna_read_isc
  if rtems_check_queue () then
    rtems_read_isc ()
  else
    suspend current Linux process
    call Linux scheduler
  end
end

begin rtems_read_isc
  if queue_ready then
    read queue
  else
    suspend calling RTEMS thread
  end
end
end

```

Algorithm 1: Inter-System Communication mechanism

The initial plan was to support transparent semaphores, message queues and shared memory for HRT/NRT communication. Later, this approach was dropped to enforce isolation between both sub-systems and a specific message queue system to support isolation was developed and implemented on the ISC module. With respect to this approach, various alternatives were explored prior to its implementation. In particular we investigated where the communication resources had to be stored: if the communication resources are placed in the RTEMS memory space, the Linux kernel can only access them indirectly through system calls. If the communication resources are placed in the Linux memory space, both RTEMS tasks and Linux kernel can access them, but mutual exclusion to this memory area is required and this is non-trivial for tasks inside different subsystems. Although both approaches are possible, xLuna implements the first approach, guaranteeing higher reliability and clear separation among the HRT and NRT subsystems.

3.4 Putting It All Together

Deploying such a complex system on the target hardware architecture is not a simple task and it requires particular steps to be taken, as shown in Figure 8. The RTEMS and Linux kernel images are created separately and loaded in different physical memory areas. Then, the binary of the final xLuna system is generated by linking these two kernel images together with a small bootstrapper. To simplify the implementation of the memory protection mechanisms, RTEMS and Linux run in separate physical address spaces. Nevertheless this separation is not mandatory, since such protection, as explained in Section 3.3.1, could be achieved by only using MMU.

The bootstrapper is the entry point to the entire xLuna system. Its purpose is mainly to define the initial MMU mapping, i.e. the RTEMS window and the Linux kernel window. After switching on the MMU, it jumps to the virtual memory space of the RTEMS entry point and gives control to the RTEMS kernel. The raw physical addresses are used only at the very beginning of the booting process, then the virtual addresses of the RTEMS window are considered.

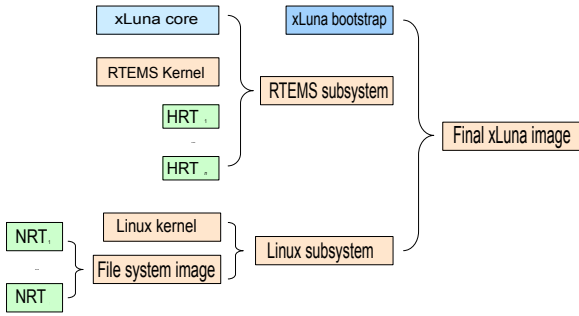


Figure 8: Linking order of the different sub-systems in order to produce the executable image of the whole architecture

The bootstrap process depends on how the final xLuna image is linked together. Figure 8 gives an idea of the different stages that are used to produce the final image. During linking, RTEMS is relocated to the virtual address in the RTEMS window, and similarly, the Linux subsystem is relocated to the virtual address of the Linux window.

4. PERFORMANCE

In this Section we show some performance measures on the xLuna system. In particular we demonstrate how the real-time responsiveness of RTEMS is not affected by the Linux subsystem. Since memory is a precious resource in embedded systems, we also present the system’s memory requirements.

4.1 Performance

The system’s performance was measured using TSIM[6] configured as a LEON 2 processor running at 50 MHz, 4-way set associative cache with 16 KB per way, UART, Timer and an AHB bus connecting all the components.

Figure 9 shows the interrupt response time of RTEMS as the number of tasks running in the system increases. In our experiments, Linux executes a benchmark (the queens problem) and some mathematical tasks, while RTEMS performs some floating point operations interleaved with random length wait states. The measurements were taken with an increasing number of RTEMS tasks with a constant the number of Linux tasks and viceversa. Interrupts were generated at random instants during simulation. As shown, xLuna does not affect the real-time behavior of RTEMS: the interrupt response time remains constant and independent of the number of tasks running on the system.

Figure 10 shows the runtime of the *queens* Linux benchmark, as measured while varying the number of both NRT and HRT tasks. As expected, the execution time scales linearly with the number of Linux tasks with the same priority. On the other hand, when increasing the number of RTEMS tasks, the execution time tends to saturate. This is due to the fact that RTEMS tasks interleave idle time with their execution (as normal in a real-time system), and exceeding a certain number of tasks, the cumulative idle time is sufficient to terminate the queens benchmark. This proves that the performance of Linux tasks is not excessively affected even when running a large number of real-time tasks.

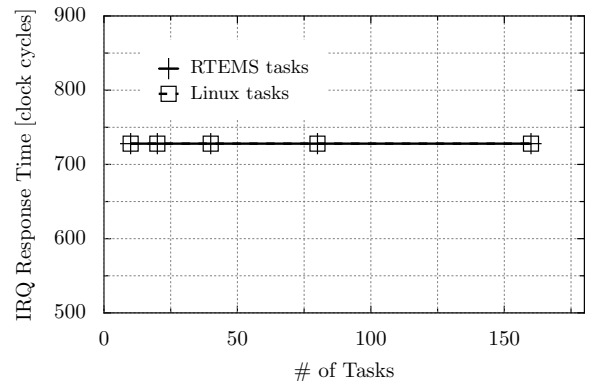


Figure 9: Interrupt response latency of the RTEMS sub-system with respect to the number of tasks in the system

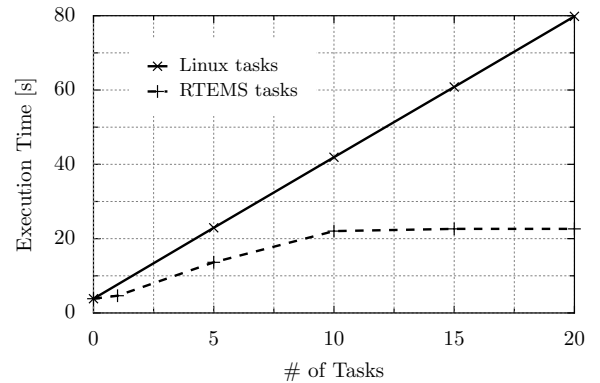


Figure 10: Execution time of the NRT queens benchmark with respect to the number of tasks in the system

4.2 Memory footprint

The minimum space that must be allocated to hold the bootstrapper, xLuna core, RTEMS kernel and HRT tasks amounts to 256 KB. For this reason, to maximize the TLB performance, we used 256KB pages. Adding any HRT task to the “empty” system requires at least a new page, i.e. additional 256KB. Besides the Linux kernel and global data, the Linux image holds a file-system containing the NRT tasks, along with any other programs and libraries needed by the tasks themselves. The plain image without any NRT task is 1.1 MB , 1 MB being occupied by the kernel code and 87 KB by global data. Figure 11 shows how memory is distributed among the different parts of the system.

5. CONCLUSION

In this paper we presented xLuna, a software architecture featuring a portable user-mode version of the Linux OS, running on top of the Real-Time Operating System (RTOS) RTEMS. Concerning real-time tasks, xLuna provides all the features provided by RTEMS 4.6.6 since no modifications were performed to this kernel. For non-real-time task, Linux version 2.6.11 was used. The main objective of xLuna is the ability to merge the real-time, predictable, reliable and safe characteristics of RTEMS and the well known POSIX compliant Linux system in one system, providing a loosely

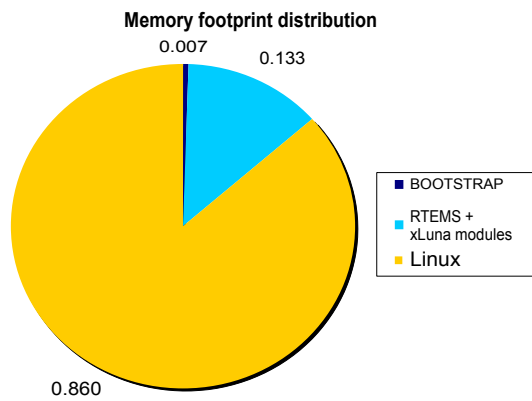


Figure 11: Memory utilization

coupled communication mechanism between the two. Even though minimalist, this loosely coupled mechanism helps maintaining the isolation between systems, protecting the Hard Real-Time system from the Non Real-Time one, and keeping the global system predictable and deterministic. Results show that xLuna does not affect RTEMS real time behaviour and interrupt response time, and that Linux tasks are not excessively slowed down even when running concurrently with a large number of RT tasks.

Future work include porting xLuna to other architectures, the increase of the number of message queues employed for Inter-Systems Communication, and the improvement of the synchronization mechanisms, with the aim of providing flexible and efficient communication among the two systems.

6. REFERENCES

- [1] Adeos Nanokernel, <http://home.gna.org/adeos/>.
- [2] SnapGear Embedded Linux Distribution, <http://www.snapgear.org/>.
- [3] A. Colin and I. Puaud. Worst-case execution time analysis of the RTEMS real-time operating system. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 191–198, 2001.
- [4] L. Dozio and P. Mantegazza. Real time distributed control systems using RTAI. In *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 11–18, 2003.
- [5] K. Fu. RTLinux: an interview with victor yodaiken. *Crossroads*, 6, 1999.
- [6] Gaisler Research. TSIM, <http://www.gaisler.com/>.
- [7] A. C. Heursch and H. Rzehak. Rapid reaction linux: Linux with low latency and high timing accuracy. pages 4–4, Oakland, California, 2001. USENIX Association.
- [8] B. Leslie, C. van Schaik, and G. Heiser. Wombat: A portable user-mode linux for embedded systems. 2005.
- [9] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: a new reclaiming algorithm for server-based real-time systems. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 211–218, 2004.
- [10] Masmano, Real, Ripoll, and Crespo. *Extending the*

Capabilities of Real-Time Applications by Combining MaRTE-OS and Linux, pages 144–155. 2004.

- [11] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32:1415–1425, 1989.
- [12] F. Mehnert. L4RTL: Porting RTLinux API to L4/fiasco. 1999.
- [13] F. Mehnert, M. Hohmuth, and H. Hartig. Cost and benefit of separate address spaces in real-time operating systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 124–133, 2002.
- [14] S. Ruocco. A real-time programmer’s tour of general-purpose L4 microkernels. *EURASIP Journal on Embedded Systems*, 2008:14, 2008.
- [15] S. Suzuki and K. G. Shin. On memory protection in real-time OS for small embedded systems. In *Proceedings - Fourth International Workshop on Real-Time Computing Systems and Applications*, pages 51–58, 1997.