Lightweight RTAI for IA-32

Michael Heimpold and Robert Baumgartl
Chemnitz University of Technology
Department of Computer Science
09107 Chemnitz, Germany
{michael.heimpold, robert.baumgartl}@cs.tu-chemnitz.de

Abstract

Using the IA-32 architecture for time-critical tasks allows highest computing performance at reasonable hardware cost. Because many open source real-time operating systems are based on Linux, the comparatively high resource requirements of Linux also apply to these systems.

In this work we present a port of the RTAI API to a bare IA-32 machine which eliminates Linux' core subsystems. Instead of a full API reimplementation we used an established Linux/RTAI as a starting point and gradually eliminated the Linux functions. This methodology minimizes changes to the RTAI code and ensures easy maintenance of the resulting code base.

Our measurements show that the resulting system is as efficient as a conventional RTAI. The uncompressed memory footprint of the kernel is about 370 KiB. Hence, the resulting system is a small and efficient real-time kernel of clear design which provides the RTAI API and is ideally suited for embedded systems with low resources.

1 Introduction

For several reasons, the Intel IA-32 is not a traditional architecture for real-time systems. Its complex caching hierarchy, branch prediction unit and incomplete documentation makes worst case execution time (WCET) estimation difficult. On the other hand, its superior price/performance ratio and its very good support of the Linux operating system and the GCC toolchain lead to more and more IA-32-based embedded systems.

Although Linux in itself cannot be regarded as fully real-time, several projects exist to co-locate a real-time API with the classic Linux kernel. Usually, the Linux subsystem is executed as low-priority real-time task. Interrupt virtualization ensures that no part of the Linux kernel hogs the CPU for indefinite periods of time. Examples are the well-known RTLinux project [9], the Real-Time Application Interface (RTAI) [8], or Xenomai [10].

Because these real-time operating systems are based on Linux, the comparatively high resource requirements of Linux in terms of CPU power and memory size also apply to them. According to [5], at least 2 MiB of RAM are necessary to successfully run Linux. For hardware platforms below that boundary no open-source real-time operating system has been available.

Therefore, the idea has been coined to strip the real-time kernel from the Linux subsystem and thus create a very small and efficient operating system with a well-established real-time API. As a consequence, this project has been proposed and realized.

In typical real-time enabled Linux systems, the Linux subsystem is used for

- loading and booting the real-time kernel and
- non-real-time functions (e. g. logging, user interaction, . . .).

Hence, some way of establishing the system must be found. Additionally, if non-real-time-functionality is necessary, it must be reimplemented as low-priority real-time task.

The remainder of the paper is structured as follows: section 2 gives a short overview of similar projects and documents some of the earlier research work. In section 3, we present the design of Lightweight RTAI by looking at two major aspects: which concepts of Linux we identified to preserve and which concepts were deliberately omitted from LRTAI. The following section 4 presents some performance numbers and memory requirements and demonstrates that LRTAI is already very competitive. The paper closes with a short summary and a look on new projects.

2 Related Work

The first system realizing a low-resource real-time kernel based on open source software was Stand-Alone RTLinux by Esteve et al. [1]. The work was done by an incremental code migration from the RTLinux tree to the code base eliminating the Linux functionality in the process. It seems that this approach is disadvantageously in terms of maintainability of the code base.

Because of that drawback and because we wanted to provide the RTAI API instead of RTLinux, we decided to do a similar project, called Lightweight RTAI. It has been originally developed for a digital signal processor (DSP), the Texas Instruments TMS320C6x [4, 6]. In contrast to the project described in this paper, the port could not be realized by gradually removing functionality from a conventional Linux/RTAI system. Neither Linux nor the GNU Compiler Collection are available for that architecture. Instead, the main effort was to instrument the RTAI source code to compile with the proprietary Texas Instruments compiler.

After successfully finishing that project the question arose, whether it would be possible to adapt LRTAI to a conventional IA-32 architecture but it was postponed for some time because of the assumed difficulties adapting and porting the boot process and the Linux module management.

In the meantime, Masmano et al published a new version of Stand-Alone RTLinux which seems to improve on the initial approach [7].

3 System Design

3.1 Possible Approaches

Two different approaches can be distinguished to realize the formulated goal. The first one is to reimplement the RTAI API from scratch without relying on any Linux functionality. This seems straightforward enough and offers a very high degree of flexibility. From a developer's point of view, this approach is not too favorable. Testing the kernel cannot start before it is more or less complete. Additionally, implementation effort is high and will invariably lead to programming errors which are hard to track in the prospected environment. This might not pose a problem but could negatively influence the acceptance of the project by other parties who prefer a matured and well-tested code base.

The second approach is to start with an established RTAI/Linux system, identify all Linux dependencies and gradually removing unwanted parts respectively reimplementing needed functionality. Here, testing is performed in parallel with implementation work. By relying as much as possible on the very-well established RTAI code base one ensures both a maximum of code quality of the resulting operating system and a minimum of additional work. Of course, this strategy offers less degrees of freedom regarding the implementation, but from our point of view, the RTAI code seems more or less optimal. Therefore, we chose the latter approach to realize Lightweight RTAI.

Apart from the primary design prinziples, Lightweight RTAI was expected to be implemented with a minimum of code changes to existing RTAI code. New functions were consequently located in new source files. The goal was a simple patch against the latest RTAI source code tree to ease maintainability and portability to new versions of RTAI.

3.2 Taken-over Linux Concepts

Although the goal of this project is to separate and eliminate Linux from RTAI as thorough as possible, some Linux concepts were taken over to Lightweight RTAI.

3.2.1 Binary Image Layout

Traditional Linux/RTAI uses the Linux kernel module interface to establish the system. All RTAI components as well as real-time applications are loaded by *insmod*. As soon as the RTAI core module is loaded, the kernel relinquishes its execution control and is subsequently executed as a lowest-priority RTAI application. By stripping Linux from RTAI, this mechanism is not available anymore. Therefore, another way of booting the system must be conceived.

The simplest form of booting an operating system image is to use a bootloader to copy the image from file to main memory and transfer control by simply jumping to a predefined entry address. Most currently available loaders are able to perform that function. As a disadvantage, this mechanism doesn't allow to pass parameters between bootloader and kernel image. As this is clearly an important feature for LRTAI, we did not pursue that idea further.

Another option would be to define a new image layout and parameter passing mechanism. Although this approach is the most flexible and many bootloaders are open source, it requires the modification of any bootloader considered for booting LRTAI.

Instead, we chose to keep the Linux binary image layout for LRTAI. In this way, all bootloaders capable of booting Linux could be used for booting LRTAI, too; they detect a standard Linux image when loading LRTAI.

In this way, we could also preserve the traditional kernel command line interface. It consists of a simple command string which can be edited by the user during execution of the bootloader. After editing, the starting address of that string is written to a well-known memory location within the loaded kernel image. While booting, the kernel iterates over all elements of that command line. Typically, callback functions are invoked for every known element registered at compile time. By preserving that mechanism, we can provide a means to configure LRTAI at boot time.

3.2.2 Kernel Image Compression

To reduce the image size of traditional Linux kernels, compression has been employed for a long time. Because the usually used zlib algorithm does not work in-place, this concept requires more main memory: it must provide space for both the compressed and uncompressed image as well as the decompression code. On systems with very small main memory this could pose a problem. Additionally, the boot process is prolonged because the image decompression requires processing time.

We decided to include that feature in LRTAI because it allows to use smaller mass storage (flash memory or EEPROM) the kernel is loaded from. Even though flash memory capacity is increasing at a rapid rate, a small kernel footprint means more space for application programs. Of course, this evaluation depends on the price ratio between RAM and flash memory and must potentially be revised in the future.

3.2.3 Initcall Mechanism

When modules are loaded dynamically into or unloaded from the Linux kernel, standard functions to initialize and register respectively deregister that module are executed by the kernel. The same mechanism is used implicitly when modules are compiled statically into the kernel. To this aim, the so-called *initcall table* is used which holds pointers to initialization functions of different priority.

Because RTAI applications also are kernel modules and therefore frequently depend on that initialization mechanism it must not be removed. Currently, the *module_init* function of an RTAI implementation translates to the Linux implementation of *device_initcall*.

3.2.4 Memory Management

After a successful boot of the Linux kernel, some memory regions which are only needed during bootstrap (e. g. the initialization functions of a compiled-in driver) and which must be explicitly marked by the programmer are freed by the system. Although LRTAI is stripped down to a minimum and therefore potentially provides only a small amount of memory to be freed after bootstrap, this mechanism has been preserved to further reduce memory consumption. We feel that this will lead to more memory savings in later project stages with LRTAI providing more functionality.

Linux provides a preliminary memory management which is solely used during bootstrap, the so-called *Bootmem Allocator*. It statically initializes page tables and performs a linear mapping between available memory and address space. It is limited to 8 MiB of RAM. Because Linux' low-level memory management is very complex, we decided to retain the bootmem allocator and use it in LRTAI. The upper bound of 8 MiB should not pose a problem, because with such an amount of memory traditional Linux/RTAI will run.

The second level of memory management is traditionally provided by Jeff Bonwick's slab allocator. Because of the code size and complexity of that allocator, a simple and small replacement exists for embedded systems, the so-called *slob* allocator. It is more memory-efficient but at the same time more susceptible to fragmentation [3]. Because real-time applications usually do not make heavy use of the heap, we deemed the slob allocator ideally suited for LRTAI and adapted it accordingly to use the bootmem allocator. It solely provides *kmalloc()*, *ksize()* and *kfree()* and is used during the boot process and in the case of LRTAI applications requesting private heap.

3.3 Removed Linux Concepts

As a general-purpose operating system, Linux offers a lot of features. Many of them would be useful if included in LRTAI, some features are actually required by RTAI while sharing the existing code (e. g. spinlocks). However, including features leads to a large memory footprint in the end. Therefore, shrinking the system as much as possible for small-scale systems means omitting code and functionality. This section briefly discusses some related aspects.

3.3.1 File System Layer

The RTAI API does not provide a file abstraction. If file support is needed, it is accomplished by the Linux subsystem without real-time guarantees (e. g. , when loading a new RTAI module). As a consequence, we removed the file system layer from LRTAI. Of course this eliminates the possibility to store data to or read data from mass storage as well as dynamically loading LRTAI code.

3.3.2 Device Driver Infrastructure

Stripping Linux from RTAI eliminates functions typical device drivers rely on. Therefore, with the exception of the real-time aware driver for the serial interface provided by RTAI itself and the drivers for the interrupt/timer circuit, LRTAI does not provide any device drivers nor infrastructure for them. Hence, porting drivers from Linux to LRTAI may require some effort but this is the case for classical RTAI, too.

Additionally, the code concerning management of tasks in user space, such as the scheduler or signal management as well as user/kernel space transitions (system calls) has been evicted. Likewise, Linux' memory management has been dropped. This code is quite complex and provides many unneeded functions. Further, it is strongly interwoven with the file system layer.

This concludes our discussion of system design aspects of LRTAI. A more thorough analysis as well as many implementational details can be found in [2].

4 Performance Evaluation

4.1 Testing Environment

The following system configuration has been used for all measurements:

- AMD K6, 200 MHz,
- 128 MiB RAM installed, but only 8 MiB used by LRTAI,
- Linux kernel version 2.6.17,
- ADEOS IPIPE version 1.3-08,
- RTAI version 3.4,
- Linux timer interrupt running at 100 Hz.

Because of the missing device driver and file system support, it is impossible to log timing values into a file. Instead, the test system transmitted its measured data via the serial console to another host which redirected the received data into a file. Additionally, no (potentially complex) driver functions would spoil our measurement.

The somewhat ancient configuration was chosen because it was equipped with a serial interface. Typical x86-based embedded systems provide a similar computing power, therefore we deemed our measurement platform adequate.

We tested our LRTAI against a conventional RTAI/Linux installation on the same machine based on the Debian Sarge distribution.

4.2 Scheduling Latencies

To evaluate scheduling performance, we used the latency measurement module of RTAI and adopted it slightly to fit both environments. This was necessary, because in the original code, a user-space process reads the measurement data from a FIFO and logs them. In our version, measurement data is simply output by the kernel module via *rt_printk()* instead.

The module sets up a task with a period of 100 μs which calculates the difference between the expected and the true activation time. Each measurement collected 250.000 data values. The RTAI/Linux system was stressed by CPU and I/O load by using `cpuburn` and flood pinging the system. Due to the lack of driver support, only CPU load could be generated for LRTAI. This was achieved by finalizing system initialization with an endless loop instead of a call to *cpu_idle()*.

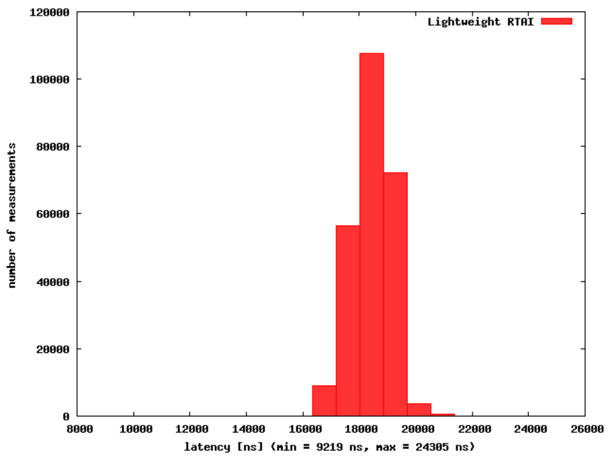

FIGURE 1: *Scheduling Latency of LRTAI in Oneshot Timer Mode*

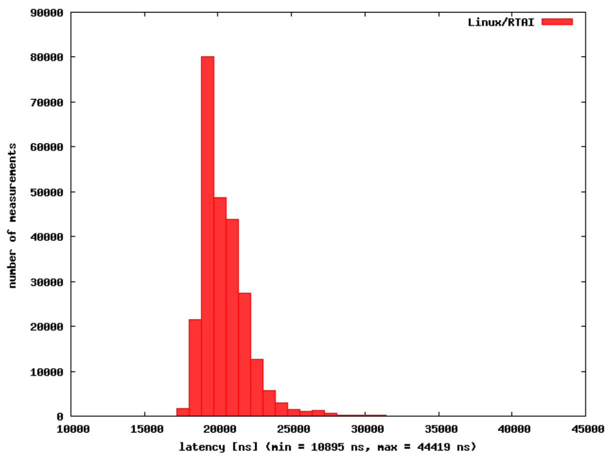

FIGURE 2: *Scheduling Latency of Linux/RTAI in Oneshot Timer Mode*

Figures 1 and 2 depict the measured scheduling latency in oneshot timer mode for LRTAI and standard Linux/RTAI, respectively. The first interesting fact to note is that the average latency is slightly smaller for LRTAI. This could be explained by the nonexistent I/O load in the LRTAI test case. Further, the worst case for LRTAI is 24.305 ns, whereas Linux/RTAI needs almost twice that time in the worst case (44.419 ns). Third, the timing variance is also smaller for LRTAI.

Figures 3 and 4 present the results for periodic timer mode. The performance is similar as in the previous experiment. Again, the worst cases both for minimum and maximum scheduling latency are smaller for LRTAI than for Linux/RTAI. Here, the timing variance of LRTAI is quite superior.

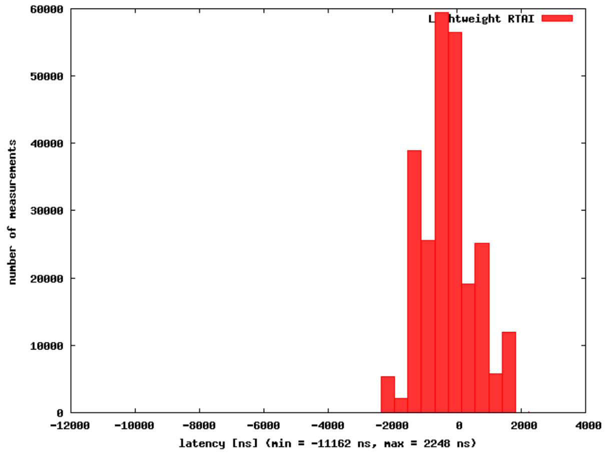

FIGURE 3: *Scheduling Latency of LRTAI in Periodic Timer Mode*

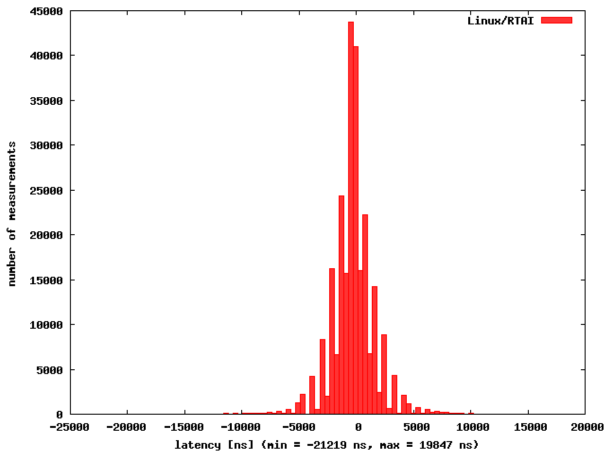

FIGURE 4: *Scheduling Latency of Linux/RTAI in Periodic Mode*

We can conclude that LRTAI is competitive in comparison to Linux/RTAI.

4.3 Image Size and Memory Footprint

The current kernel image size of the generated LRTAI *zImage* is about 122 KiB. The corresponding memory footprint is about 370 KiB. Therefore, it is possible to run LRTAI and quite some real-time applications on systems with less than 1 MiB of RAM installed. We believe that this boundary can even be lowered with ongoing project's progress.

5 Conclusions and Outlook

In this paper we have demonstrated that it is possible to port the RTAI API to a bare IA-32 machine. We started with a fully-fledged Linux/RTAI and gradually removed as much Linux components as possible. The resulting real-time kernel has an uncompressed memory footprint of approximately 370 KiB. As we have demonstrated by our measurements, the scheduling latency of the kernel is even a bit better than for Linux/RTAI. We can therefore conclude that Lightweight RTAI is a small and very efficient real-time kernel for systems with low resources.

LRTAI is not yet optimal. There are still some dependencies of the RTAI scheduler implementation which require access to some Linux code paths. This is because RTAI is capable to "steal" and return tasks from Linux' scheduler. Currently, the Linux internal implementation of these functions are directly called. This could be improved upon in further LRTAI versions and would result in a further footprint shrink. Likewise, the changes introduced in the build system could be minimized, improving compatibility to RTAI.

A second point of improvement are new features. As a first example, support for APICs could be implemented which would be the base for symmetric multiprocessing. By optimizing selected code paths by hand, it seems that the performance of the kernel, which is already better than that of traditional Linux/RTAI, could be improved even further.

A thorough comparison of Lightweight RTAI with Embedded RTLinux [7] would be very interesting. Both approaches potentially could be improved by learning from the adversary.

LRTAI is released under the GNU Public Licence and is available from our web site. Interested people are invited to join the development team.

References

[1] *Stand-Alone RTLinux-GPL.* Fifth Real-Time Linux Workshop, Valencia, 2003, Vicente Esteve, Ismael Ripoll, Alfons Crespo

[2] *Lightweight RTAI for IA-32.* Diploma Thesis, Chemnitz University of Technology, 2007, Michael Heimpold

[3] *Anatomy of the Linux Slab Allocator. http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/*, 2007, M. Tim Jones

[4] *Lightweight RTAI for DSPs.* First Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), 2005, Jens Kretzschmar, Robert Baumgartl

[5] *The Linux FAQ. Part III: The Kernel. http://tldp.org/FAQ/Linux-FAQ/kernel.html* David C. Merrill

[6] *Completing and Testing Lightweight RTAI/C6x.* Student Research Paper, Chemnitz University of Technology, 2006, Michael Luft

[7] *Embedded RTLinux: A New Stand-Alone RTLinux Approach.* Eighth Real-Time Linux Workshop, Lanzhou, China, 2006, Miguel Masmano et al

[8] *https://www.rtai.org/* (10/01/07)

[9] *http://www.fsmlabs.com/* (10/01/07)

[10] *http://www.xenomai.org/* (10/01/07)