Fast Linux Bootup using Non-Intrusive Methods for Predictable Industrial Embedded Systems

Mikael Åsberg and Thomas Nolte
MRTC/Mälardalen University
Västerås, Sweden
{mikael.asberg,thomas.nolte}@mdh.se

Mikael Joki, Jimmy Hogbrink and Saher Siwani
Eskilstuna Elektronikpartner AB
Eskilstuna, Sweden
{mikael.joki,jimmy.hogbrink,saher.siwani}@eepab.com

***Abstract*—Fast kernel boot-time is one of the major concerns in industrial embedded systems. Application domains where boot time is relevant include (among others) automation, automotive, avionics etc. Linux is one of the big players among operating system solutions for general embedded systems, hence, a relevant question is how fast Linux can boot on typical hardware platforms (ARM9) used in such industrial systems. One important constraint is that this boot-time optimization should be as non-intrusive as possible. The reason for this comes from the fact that industrial embedded systems typically have high demands on reliability and stability. For example, adding, removing or changing critical source-code (such as kernel or initialization code) is impermissible.**

This paper shows the steps towards a fast-booting Linux kernel using non-intrusive methods. Moreover, targeting embedded systems with temporal constraints, the paper shows how fast the real-time scheduling framework ExSched can be loaded and started during bootup. This scheduling framework supports several real-time scheduling algorithms (user defined, multi-core, partitioned, fixed-priority periodic tasks etc.) and it does not modify the Linux kernel source code. Hence, the non-intrusive bootup optimization methods together with the un-modified Linux kernel and the non-patched real-time scheduler module offers both reliability and predictability.[1]

***Index Terms*—real-time systems, embedded systems, linux, hierarchical scheduling**

I. Introduction

So why use Linux in industrial embedded systems with requirements on predictable timing? Lets look at the alternatives; common alternative solutions include Microsoft Windows or pure Real-Time Operating Systems (RTOS) like VxWorks, FreeRTOS, µC/OS-II, OSE etc. Embedded system platforms vary a lot in hardware setup (processor, devices etc.) and requirements (power consumption, memory/disk usage, real-time responsiveness etc.). Add to this that new hardware devices and chipsets emerge in rapid pace which must be supported by OS developers making device-driver, architecture and Board Support Package (BSP) development a heavy burden. This variation in hardware/software is not well suited for General Purpose Operating Systems (GPOS) like Windows. RTOS are more fit for these kind of variations and they also offer a high degree of reliability. However, the lack of standardization (i.e., “roll-your-own” software) of software stacks and inability to keep up with emerging technologies makes it tough for these vendors as well. Linux has a modular kernel which can be scaled down to less than one megabyte. It has a solid reputation of being reliable and it has many developers around the world that evolve the code base every single day. Above all, it is open source, making all of this possible [1].

The favorable aspects of using Linux are many. The most important reasons are kernel stability and cutting down development costs of embedded systems [2]. These properties hold in Linux due to its supreme degree of distributed kernel development and rich software stack.

Fast bootup time is important in industrial embedded systems such as those found in vehicles, i.e., a car should be ready to start short after the user has inserted the key into the ignition. Real-time responsiveness is without a doubt an important property as well. Take for example the actuation of an Antilock Brake System (ABS) or the precise fuel and air injection in a combustion engine [3]. The reliability and stability of these systems are highly prioritized since these products are expected to run for a long time without errors [1] (the product brand is at stake which has a huge economical impact). Getting real-time responsiveness of native Linux solutions is equally important in mobile phones [2]. A handheld device is also expected to run forever without errors and to get good real-time performance from Linux for its multimedia and voice processing.

There is no doubt that real-time is needed in most embedded systems applications. However, the degree of real-time may of course vary since the term *real-time* could be considered relative. For example, for an airbag system in a car to be correct the airbag may never be released too early or too late. Compare this to the playback of a movie on a smartphone that should be perfect, although no human injury will occur if one or two movie frames are missed.

This paper presents non-intrusive methods for both fast booting of Linux, and in conjunction with support for timing predictability. The timing predictability is enabled by including a specialized real-time module called ExSched [4] in the boot process. We give a more detailed description of ExSched in Section II-B. This module does not modify Linux in any way and includes a rich variety of real-time scheduling algorithms. Hence, we have chosen to include this module since it is 100% non-intrusive and does not add significantly much to

[1]The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programmes SYNOPSIS and PRESS.

the overall boot time.

Note that non-intrusiveness is something that we prioritize since it has many advantages that are related to the main philosophy of Linux:

"Of course, you could also dive in and modify Linux to convert it into a real-time operating system, since its source is openly available. But if you do this, you will be faced with the severe disadvantage of having a real-time Linux that can't keep pace, either features-wise or drivers-wise, with mainstream Linux. In short, your customized Linux won't benefit from the continual Linux evolution that results from the pooled efforts of thousands of developers worldwide." [1].

Contribution The main contributions of this paper are:

1) We present a detailed description and evaluation of non-intrusive methods for achieving a fast boot-time of Linux in an industrial embedded-system setup.
2) This study also considers real-time aspects since we focus on a real-time adapted kernel by including the real-time scheduler module ExSched in the boot process. This real-time capability is of course non-intrusive since it does not alter the Linux kernel in any way.

Outline The outline of this paper is as follows: Section II presents the background and preliminaries, and Section III outlines the related work in the area of Linux boot-time reduction. Section IV shows the non-intrusive methods used for reducing the boot time, and finally, Section V concludes our work.

II. Preliminaries

This section explains the main steps of the boot process of Linux in order for the user to fully understand the enhancements that we have made for minimizing the boot time. We will also give a description of the ExSched framework.

A. The Linux boot process

We will give a description of the boot process in Linux and where, in this process, there are possibilities to reduce time [5], [6].

First of all, we will start by defining what we, in this paper, mean by boot time. It is not uncommon that embedded systems never power down, i.e., the system looks like it is off while it in fact is in low power-mode with the operational status saved in memory. When power-on is applied, it is really just a resume from a suspended state (referred to as warm boot). Some cell phones never do a full boot unless the battery is removed and inserted again. In this case, the operational status is lost (i.e., the suspend state is lost) so the device is forced to boot from non-volatile memory and conduct all initializations (both in software and hardware). When a device boots in this way we refer to it as a cold boot. We are interested in the total boot time from power-on, i.e., cold boot.

Upon pressing the power-on button, the hardware spends a small amount of time in waiting for stabilization of clocks etc. and after that the bootloader starts to execute immediately. The hardware finds the bootloader in non-volatile memory such as NOR or SD card flash. The user has to specify where the hardware should look for the bootloader. This is typically done by setting some pins on the hardware platform (or set it in the BIOS if the platform is a typical desktop computer). When the bootloader starts it copies itself from non-volatile memory to DRAM and continues to execute there. In DRAM the bootloader initializes the hardware (memory controller etc.). The bootloader is configured by the user where it should look for the Linux kernel image. When it finds the kernel image it copies it from the storage it is residing in to DRAM memory. After the copy is done the bootloader passes the control to the kernel by calling the Linux kernel function start_kernel(). If the kernel is in compressed form (GZIP, LZO etc.) then the kernel will uncompress itself (before start_kernel()) using a small bootstrap decompression loader that is appended to the compressed kernel. The bootstrap loader is appended to the image automatically at the end of the Linux kernel compilation process. Once the kernel starts it will initialize itself, load modules/drivers, locate and mount a root filesystem (where user initialization scripts reside) and execute these scripts. The kernel boot is the most dynamic step along the whole boot process, i.e., it is difficult to describe the boot steps after the kernel starts because there might be so many steps in the kernel boot. The amount of steps can also be very small, it depends on how many modules/packages the kernel is configured with. This relates to the kernel compilation step where the user can chose which modules that should reside in the kernel. Depending on how many modules that are chosen, the kernel image size can very from less than one megabyte to several megabytes.

Some basic steps that are important when optimizing the kernel boot time are described in the literature [5]:

1) Keeping the Linux kernel small is probably the most efficient way of decreasing the boot time. Smaller kernel images require less time to load to memory and they decompress faster. The kernel boot process will also be faster since less modules are needed to be loaded.
2) Keep the bootloader image size small since this will decrease the load time between non-volatile storage and memory. Also, it will initialize faster.
3) Having the kernel in compressed or uncompressed form has a big impact on the boot time. There is a tradeoff here because an uncompressed kernel takes more time to load from non-volatile storage to memory compared to a compressed version, however, the decompression step is completely removed when having an uncompressed kernel. Hence, it depends on the hardware and kernel size which of these two variants are faster. Hardware with slow processor and fast transfer time between storage and memory probably gains more by having an uncompressed kernel.
4) Remove initrd/initramfs from the kernel configuration since it is rarely appropriate to use it in embedded systems. It can support a wide variety of configurations by

providing various device drivers, hence, this makes the kernel more portable without the need to re-configure it. Embedded systems don't usually have this requirement so it can in most cases be left out.

5) During the kernel boot-process a timer loop-calibration is done which can take several hundreds of milliseconds. The loop counts how many loop iterations can be executed during a time unit called jiffy. This loop-iteration count is constant for each processor, hence, the user can simply observe this value during the boot and then just pass it as a boot argument at the next boot. The command lpj is used as boot argument and this will force the kernel to skip the calibration step.
6) The choice of filesystem algorithm used by the kernel affects the boot time. CRAMFS is usually recommended since it is read-only and hence compact and fast.

B. ExSched

ExSched [4] is a loadable real-time scheduler framework designed to work with the POSIX-compliant SCHED_FIFO scheduling policy. ExSched is a scheduling framework which does **not** require any modifications to the Linux kernel. ExSched supports real-time scheduling capabilities which does not exist in the native Linux kernel. The frameworks base scheduler supports fixed-priority periodic tasks which can be scheduled in a preemptive manner. ExSched is composed of kernel modules and a user-space library for easy installation (Figure 2). ExSched supports both an interface to the users in user space, e.g., a task specific interface like `rt_wait_for_period()` (Figure 1), as well as in the kernel space.

```
main(timeval C, timeval T, timeval D, int prio, int nr_jobs) {
    .
    .
    .
    rt_set_server(1);
    rt_run(0);
    for (i = 0; i < nr_jobs; i++) {
        /* User's code. */
        rt_wait_for_period();
    }
    rt_exit();
}
```

Fig. 1. Example of a task using the ExSched API.

The kernel space API can be used by developers to develop their own schedulers. A user developed scheduler in ExSched is basically a kernel module which uses the API exported by the ExSched kernel module. This makes the framework modular without the need to actually modify anything in the ExSched source code and you get reusability of scheduling functionality through the API. User defined scheduling modules can of course also export an API to other modules.

ExSched has in total 7 scheduling algorithms and one module which can record task execution for debugging purposes. Three of these algorithms are multicore schedulers; global,

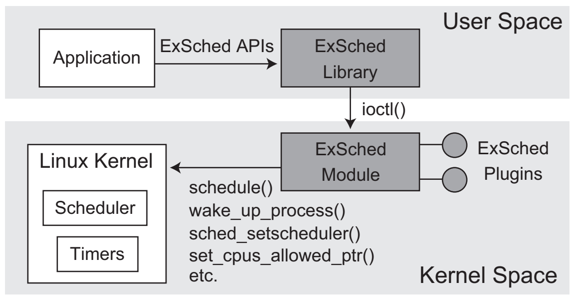

Fig. 2. The ExSched framework.

partitioned and semi-partitioned scheduling. Two of the schedulers can schedule tasks on uni-core; Fixed-Priority Preemptive Scheduling (FPPS) of periodic tasks and Earliest Deadline First (EDF) scheduling of periodic tasks. ExSched also has two diffrent two-level hierarchical schedulers (uni-core). Any scheduler (at any level) can schedule either FPPS or EDF. Our boot process includes loading and starting ExSched and the FPPS hierarchical scheduler. Hierarchical scheduling [7], [8], [9] is used when tasks (or groups of tasks) must be strictly scheduled in a encapsulated way. Each task or group of tasks may only execute in their pre-defined time-slots which gives a clear runtime separation in the time domain. This will protect tasks from using the CPU more than intended so that they do not impact other tasks or groups. This type of scheduling is mostly used where predictability is important. Figure 3 illustrates hierarchical scheduling. The $global$ scheduler is responsible for starting partitions (servers) and suspending them. The frequency at which partitions run and the time length of each run is defined in the partition $interface$. The priority of a partition is included in the interface in case of priority based global scheduling. A partition can consist of one or several tasks. The $local$ scheduler is responsible for scheduling tasks that reside in the same partition during the execution of a partition. The scheduling algorithm of the local schedulers can be arbitrary. Hierarchical scheduling can be found in ARINC653 [10], [11] compliant operating systems and these are commonly found in the avionics industry. The ARINC653 standard defines a time partitioning of applications (similar to ExScheds hierarchical schedulers) as well as memory partitioning. This will guarantee that applications will not interfere with eachother (in a unpredictable way) in terms of both CPU and memory. Hence, this will make the $system$ more predictable.

III. Related work

There has been work on non-intrusive techniques for minimizing the Linux boot time. In [5] the author describes both the Linux boot process in depth and general methods to reduce the boot time. A similar work to our paper is presented in [12]. They also focus on non-intrusive methods but their application is different. Their focus is on the more powerful ARM11 architecture which is more specialized for multimedia. The authors focus is on Android. They present a Linux kernel boot-time of 1.1 seconds and 10.1 seconds in total for the

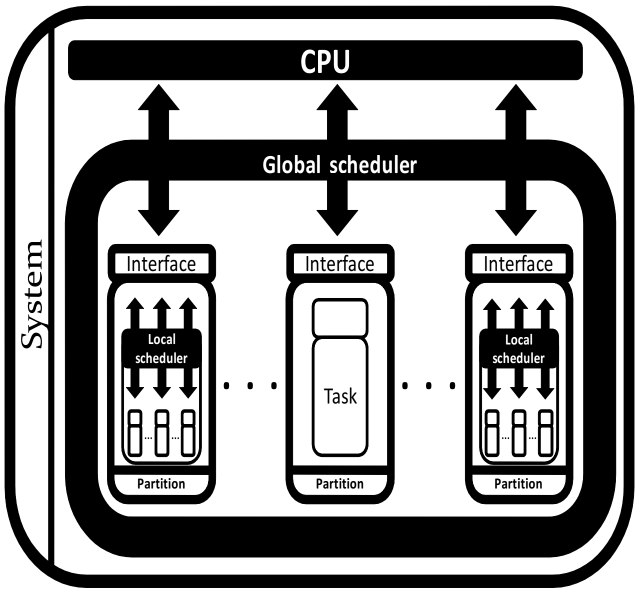

Fig. 3. Hierarchical/partitioned scheduling.

Android application. The main difference from our work is that we focus on industrial embedded systems (ARM9) which has higher demands on predictability and this is something that is considered in our boot process. Moreover, we evaluate more methods and we show a more detailed evaluation of all the methods. [6] also describes many non-intrusive optimization techniques such as quiet, lpj, probing, Execute In Place (XIP) etc. The author also describes the Linux kernel boot activities in detail. XIP means that the kernel is never loaded into main memory but instead executes from non-volatile storage. The boot process is faster but the kernel will execute slower since it resides in disk. This is not a solution that is sufficient for our application.

There is also a lot of work on intrusive techniques for minimizing the Linux boot time. The work in [13], [14], [15] use the snapshot technique. An image of the kernel is saved during runtime and then saved to flash. At the next bootup, this image is used and this decreases the total boot time. The disadvantages are that it is time consuming to create images of the running kernel and this technique requires modifications to the bootloader and the kernel. A similar work [16] uses hibernation. This has been done in mobile handsets to improve the user perceived boot time.

The authors in [17] describe a unique technique for minimizing the Linux boot time. They develop a hybrid root-filesystem (combining CRAMFS and JFFS2) which gives performance improvements at runtime (in the aspect of disk usage) but it does not boot faster than CRAMFS.

IV. Boot-time improvements

We used a Freescale I.MX28 EVK board equipped with a 454 MHz ARM926-EJ core processor and 128 MB DRAM. We used the Linux kernel version 2.6.35.3 together with the U-boot (2009.08) bootloader. The kernel and the bootloader were stored in a SDHC memory card of class 2 (4GB) prior to the bootup. We used the Buildroot [18] cross-compiler tools to compile ExSched for our I.MX28 board and a Freescale version of the LTIB[2] tool for compiling the Linux kernel and the bootloader. A LTIB script was used to create a bootable SD card with our configured kernel and bootloader. We used a C-program[3] to measure the boot time. This tool is similar to Grabserial[4] which timestamps incoming UART messages from the microcontroller. There exists other options as well such as Kernel Function Instrumentation (KFT). However, this instrumentation method requires modifications to the kernel which we want to avoid. Another solution for measuring boot time is a kernel configuration option called `CONFIG_PRINTK_TIME` which can be found in the Kernel Hacking section. This option will append timestamps to `printk` log messages which are printed during the whole Linux boot process. However, one boot time optimization method called `quiet` (which we have used) can suppress boot log-messages and thereby save hundreds of milliseconds. Hence, if `quiet` is used then `CONFIG_PRINTK_TIME` will not have any effect.

All measured values presented in this paper represent the average value of three sampled values.

A. *Compressed versus uncompressed kernel image*

Several sources [5], [12], [17] report that uncompressed Linux kernels boot faster in the general case. During our experiments we found that the differences in boot time was enormous. However, it was the compressed kernel that was superior in fast bootup. The kernel size ($\sim$1MB compressed and $\sim$2MB uncompressed) is also a typical size found in the literature related to fast booting Linux systems. Hence, our setup is general regarding both hardware and software but our results contradict previous studies.

We used a non-optimized version of U-boot (we use an optimized version in our final results) in order to compare the boot time of an uncompressed and a Lempel-Ziv-Oberhumer (LZO) compressed kernel. The reason for using this bootloader version was because it displayed more log messages. This made it possible to determine the amount of time spend in the boot loader and in the decompression stage of the kernel. We are dependent on log messages from the microcontroller in order to measure the boot time since our host application timestamps incoming UART messages. We could conclude that only 80 ms of time was spend on decompressing the 918716 bytes large compressed kernel. The uncompressed version of Linux was 1804192 bytes large and the bootloader spend a staggering 555 ms more time (to transfer the image from disk to memory) than the compressed version. Hence, the uncompressed Linux version took in total 475 ms more time to boot than the LZO compressed image. We of course have to consider that this difference (555ms) would have been smaller if we would have used the optimized bootloader version. However, it would most likely not be smaller than 80 ms.

[2]LTIB http://ltib.org
[3]tstamp.c http://processors.wiki.ti.com/index.php/Measuring_Boot_Time
[4]Grabserial http://elinux.org/Grabserial

B. Kernel compression algorithms

The literature does not elaborate about the time reductions that can be done by choosing an efficient compression algorithm for the kernel. In this section we present our findings in this topic.

Table I shows the compression size and the total boot-time of the Gnu ZIP (GZIP), LZO and Lempel-Ziv-Markov chain (LZMA) compression algorithms. The Linux image was 1804192 bytes large in an uncompressed format.

Algorithm	Image size (bytes)	Boot time (ms)
GZIP	845376	1160
LZO	918716	1128
LZMA	635276	1993

TABLE I
COMPRESSION ALGORITHMS.

It is interesting to note that the LZMA boot time is almost twice as long as LZO.

The LZO algorithm is known to be fast in decompression compared to GZIP so the results are not surprising. In total, there is a gain of 32 ms (with a small kernel) when using LZO instead of the default compression algorithm GZIP. It takes 80 ms of boot time to decompress the Linux kernel (of size 918716 bytes) using LZO, i.e, only 7% of the total boot time is spend on decompression which is a small part.

C. File systems

The choice of filesystem is well elaborated in the literature [2], [5], [12], [17]. All papers favor the filesystem CRAMFS. The reason for this is because it is compact due to that it is a read-only filesystem (fast boot time). We have experimented with different filesystem options using the LTIB configuration. We measured the total boot time using 5 different file systems. Table II shows the results.

File system	Boot time (ms)
CRAMFS	1128
JFFS2	1132
EXT2.GZ	1128
INITRAMFS	1132
NFS	1132

TABLE II
FILE SYSTEMS.

We did not note any significant difference in terms of boot time among these filesystems. However, we want to delimit these results since we have only tried one configuration tool (LTIB).

D. Results

The main goal of this work has been to reach 1 second total (cold) boot time. The smallest observed boot time that we obtained was 1.128 seconds. However, a "useful" kernel will most probably be larger than the one we obtained. The kernel that we configured lacks many packages, i.e., it is stripped down to a very small set of functionalities. Considering that more kernel packages might be needed, more optimizations must be done.

We used 7 non-intrusive methods for achieving a 1.128 second boot time:

1) Removing unnecessary packages from the kernel and converting necessary packages to loadable kernel modules (they can be loaded later after the kernel boot process). In total, 185 kernel configuration options were either removed or converted to loadable kernel modules. This is the most time consuming method but it is also the most efficient way to minimize the boot time.
2) Optimizing the bootloader saves a lot of time as well. Suppressing log messages from the bootloader is one way of decreasing the execution time.
3) The boot argument that minimized the boot time the most was without doubt the quiet option. It decreased the total boot time with a staggering 9%.
4) The second most efficient boot argument was the lpj option.
5) The default kernel memory allocation unit is called SLUB. It is actually a replacement of an older allocator called SLAB. It is difficult to claim any difference in performance between them since there is little documentation about these modules. However, the SLUB allocator is recommended by the literature [2], [12] because it shrinks the platform footprint. We experimented with both SLUB and SLAB and surprisingly found that (the non-default allocator) SLAB obtained about 32 ms less boot time than SLUB.
6) Linux allocates and pre-initializes all of the DRAM memory in its heap during the kernel bootup. Less memory to pre-initialize means less boot time. We used the mem boot argument to specify how much memory the kernel should allocate during the boot process. In our case, 16 megabytes was sufficient for the system to function properly and it decreased the boot time with 32 ms.
7) The third most efficient boot argument is init. The last step in the Linux kernel boot-process is to start the first task (called process in Linux terms). This task is called init and a subset of its source-code is shown in Figure 4. The function of init is to run initialization scripts (lines 11-14, Figure 4). These scripts will start various system processes and initialize serial ports, set the clock, check filesystems etc. However, the user can tell init to execute another process (lines 8-9) instead of its default processes (and hence skip all user-space initialization steps). This can be done with the init boot argument. We set init to execute an ash shell (init=/bin/ash) and this saves 31 ms.

Udev is a module that can detect devices and automatically load their drivers. The literature [5] points out that there is potential boot-time improvements that can be done related to minimizing its functionality. We found that removing this module did not affect the boot time at all. Hence, udev itself

```
/*
  * We try each of these until one succeeds.
  *
  * The Bourne shell can be used instead of init if we are
  * trying to recover a really broken machine.
*/

if (execute_command)
    run_init_process(execute_command);

run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");

panic("No init found. Try passing init= option to kernel.");
```

Fig. 4. Kernel source-code from init/main.c.

does not affect the boot time if no devices are detected.

The Read Copy Update (RCU) subsystem is a synchronization primitive that allows fast access to shared resources in the Linux kernel. It is recommended to use "UP-only-small-memory-footprint RCU" for small uni-processor systems instead of the default option since it is less resource demanding. This option reduced the kernel size with about 3000 bytes which is quite impressive. However, it does not give any improved boot times. On the contrary, it increased the boot time in the average case.

Figure 5 illustrates how much each optimization method (described previously) decreased the total boot time. As can be seen, the original boot time of the Linux 2.6.35.3 kernel was 6197 ms. Its interesting to note that the Linux kernel that was shipped with the I.MX28 board had a boot time of 26 seconds. Removing (and converting) 185 kernel packages resulted in a 49% decrease of the boot time. Improvements to the bootloader resulted in a 19% decrease of time while the 4 boot arguments quiet, lpj, mem and init together with the SLAB memory allocator resulted in a total improvement of 14%.

E. Booting ExSched

We compiled the ExSched framework including the FPPS hierarchical scheduler plugin (HSF-FP) for the Linux 2.6.35.3 ARM platform using the Buildroot [18] ARM cross-compiler. Figure 7 shows the boot log-messages that were displayed when we booted the system with the ExSched and HSF-FP scheduler modules. As can be seen, the HSF-FP scheduler creates an example system of 3 servers (which is the runtime definition of a partition) and their parameters are shown in Table III. The parameters show for example that the first partition (Server0), which has the highest priority, is started every 12 time units and that it executes for 4 time units at each release. A task (or a set of tasks) can be installed inside any of these 3 partitions which would give a runtime temporal protection against any task that would violate its assumed execution time.

The ExSched and HSF-FP module add only 80 ms to the bootup time, i.e., we get a total (cold) bootup time of 1208 ms. Figure 8 shows the execution trace of the 3 partitions scheduled by the HSF-FP hierarchical scheduler. The trace was visualized using the Grasp [19] tool and we used the HSF recorder [20] to record this trace on the I.MX28 platform. The native Linux trace-recorder Ftrace [21] can also be used to trace the task execution but we skipped this module in order to minimize the kernel size and hence also the boot time.

The server parameters (Table III) correspond to the execution pattern of Figure 8. For example, Server0 will often preempt the execution of Server2 since Server2 has lower priority. This example shows the applicability of a hierarchical scheduler in an industrial embedded-systems context and the predictability that it contributes to. Moreover, this real-time scheduler suite can be used in Linux without enforcing any modifications to the kernel unlike many other real-time solutions for Linux [22], [23], [24], [25], [26], [27], [28], [29], [30]. Last but not least, ExSched (and all of its real-time scheduler plugins) can be added to the Linux kernel bootup process with little extra overhead. Figure 6 shows the boot script used to load the ExSched modules (lines 3-4) and an ash shell (line 5) during the bootup process.

```
1. #!/bin/ash
2. # ExSchedBoot.sh
3. insmod /etc/exsched.ko
4. insmod /etc/hsf-fp.ko
5. /bin/ash
```

Fig. 6. Boot script (ExSchedBoot.sh) used to load ExSched and HSF-FP during bootup using the boot argument init=/ExSchedBoot.sh.

```
PowerPrep start initialize power
Battery Voltage = 4.25V
boot from battery. 5v input not detected
LLLCApr 16 201218:11:30
FRAC 0x92925552
memory type is DDR2
Wait for ddr ready 1power 0x00820616
Frac 0x92925552
start change cpu freq
hbus 0x00000003
cpu 0x00010001
start test memory accress
ddr2 0x40000000
finish simple test
Uncompressing Linux... done, booting the kernel.

EXSCHED: HELLO!
HSF-FP: HELLO! (4294938361)
server_create: 0 Server0 4
server_create: 1 Server1 4
server_create: 2 Server2 8

BusyBox v1.15.0 () built-in shell (ash)
Enter 'help' for a list of built-in commands.

/bin/ash: can't access tty; job control turned off
/ #
```

Fig. 7. Boot messages.

V. Conclusion

This paper concludes that the total (cold) boot time of (embedded) Linux on a typical industrial embedded systems platform (ARM9) can be decreased to an acceptable level. We

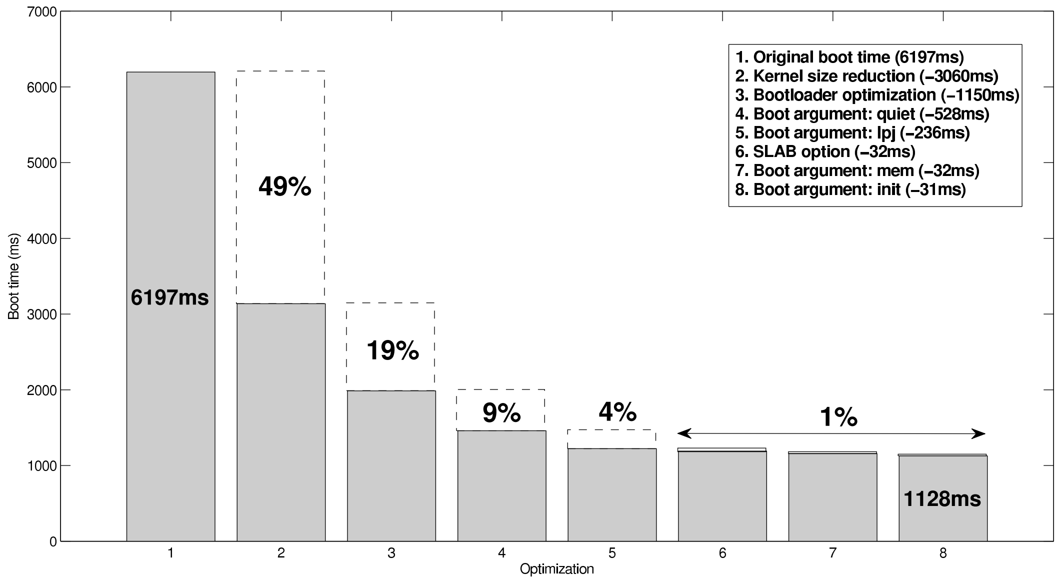

Fig. 5. Overview of boot-time optimizations.

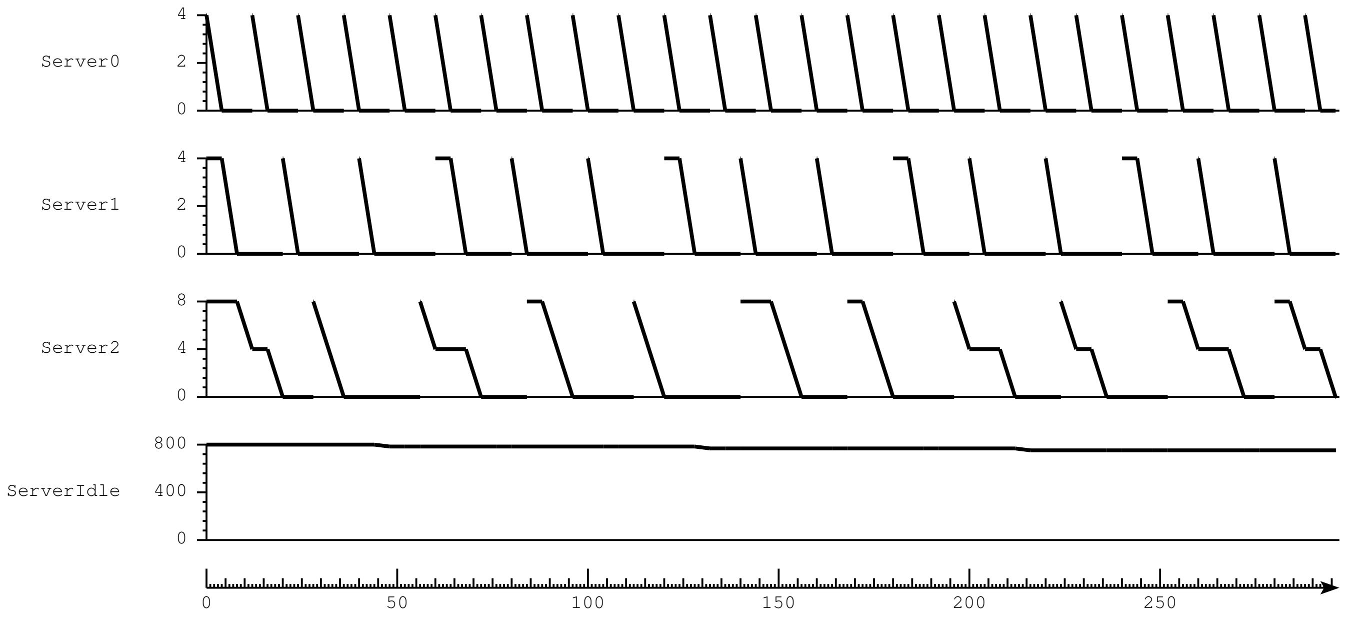

Fig. 8. Scheduling trace of the ExSched hierarchical scheduler running on the I.MX28 platform.

Server-name	*Period*	*Budget*	*Priority*
Server0	12	4	0
Server1	20	4	1
Server2	28	8	2

TABLE III
SERVER PARAMETERS.

managed to squeeze down the boot time to 1.128 seconds using only non-intrusive methods. These methods include optimizing the kernel and bootloader, adding the quiet, lpj, mem and init boot arguments, and using the SLAB kernel memory allocator instead of the default SLUB allocator. Some of these methods are uncommon in the literature, e.g., quiet, mem, init and the SLAB allocator. In fact, related work even recommends using the SLUB allocator.

The comparison of compressed and uncompressed kernels showed that compression was superior (at least in this setting) unlike previous statements in the literature which usually favor uncompressed kernels.

Our comparison of filesystems also resulted in surprising results. Related work tends to favor the filesystem CRAMFS while our experiments showed that there were almost no difference between CRAMFS, JFFS2, EXT2.GZ, INITRAMFS and NFS.

We found no related work with regard to compression algorithms. Our experiments showed that LZO was 32 ms faster compared to the default algorithm GZIP.

Our experiments also showed that it took 80 ms to load the scheduler framework ExSched and a hierarchical scheduler during the bootup process. This resulted in a total (cold) boot time of 1208 ms. Finally, we demonstrated the usefulness of a hierarchical scheduler in an industrial embedded-systems environment.

Future work includes (at least) two sources of optimizations; remove more (unnecessary) kernel packages and use NOR flash instead of SD card flash (faster transfer between flash and main memory).

REFERENCES

[1] R. Lehrbaum, "Using Linux in Embedded and Real-Time Systems," *Linux Journal*, no. 75, 2000.

[2] B. Weinberg, "Mobile phones: The Embedded Linux Challenge," *Linux Journal*, no. 148, 2006.

[3] M. Åsberg, M. Behnam, F. Nemati, and T. Nolte, "Towards Hierarchical Scheduling in AUTOSAR," in *ETFA'09*, 2009.

[4] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar, "ExSched: An External CPU Scheduler Framework for Real-Time Systems," in *RTCSA'12*, 2012.

[5] C. Hallinan, "Reducing Boot Time in Embedded Linux Systems," *Linux Journal*, no. 188, 2009.

[6] T. Bird, "Methods to Improve Bootup Time in Linux," in *Japan Linux Symposium*, 2004.

[7] P. Goyal, X. Guo, and H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," in *OSDI'96*, 1996.

[8] Z. Deng and J. W.-S. Liu, "Scheduling Real-time Applications in an Open Environment," in *RTSS'97*, 1997.

[9] J. Regehr and J. A. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *RTSS'01*, 2001.

[10] ARINC, *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), 1996.

[11] ARINC/RTCA-SC-182/EUROCAE-WG-48, "Minimal Operational Performance Standard for Avionics Computer Resources." RTCA, Incorporated, 1828 L Street, NW, Suite 805, Washington D.C. 20036, 1999.

[12] G. Singh, K. Bipin, and R. Dhawan, "Optimizing the Boot Time of Android on Embedded System," in *ISCE'11*, 2011.

[13] I. Joe and S. C. Lee, "Bootup Time Improvement for Embedded Linux Using Snapshot Images Created on Boot Time," in *ICNIT'11*, 2011.

[14] H. Kaminaga, "Improving Linux Startup Time Using Software Resume," in *Japan Linux Symposium*, 2006.

[15] H. Jo, H. Kim, H.-G. Roh, and J. Lee, "Improving the Startup Time of Digital TV," in *IEEE Transactions on Consumer Electronics*, 2009.

[16] D. Fuji, T. Yamakami, and K. Ishiguro, "A Fast-Boot Method for Embedded Mobile Linux: Toward a Single-Digit User Sensed Boot Time for Full-Featured Commercial Phones," in *WAINA'11*, 2011.

[17] K. H. Chung, M. S. Choi, and K. S. Ahn, "A Study on the Packaging for Fast Boot-up Time in the Embedded Linux," in *RTCSA'07*, 2007.

[18] A. Sirotkin, "Roll Your Own Embedded Linux System With Buildroot," *Linux Journal*, no. 206, 2011.

[19] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems," in *WATERS'10*, 2010.

[20] M. Åsberg, T. Nolte, and S. Kato, "A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux," in *RTCSA'11*, 2011.

[21] T. Bird, "Measuring Function Duration with Ftrace," in *Japan Linux Symposium*, 2009.

[22] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems," in *MMCN'98*, 1998.

[23] V. Yodaiken, "The RTLinux Manifesto," in *Linux Conference*, 1999.

[24] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous, "RTAI: Real Time Application Interface," *Linux Journal*, no. 29, 2000.

[25] K. Yaghmour, "Adaptive Domain Environment for Operating Systems," *Opersys inc*, 2001.

[26] D. Faggioli and F. Checconi, "An EDF Scheduling Class for the Linux Kernel," in *Real-Time Linux Workshop*, 2009.

[27] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS$^{\mathrm{RT}}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *RTSS'06*, 2006.

[28] K. Lakshmanan and R. Rajkumar, "Distributed Resource Kernels: OS Support for End-To-End Resource Isolation," in *RTAS'08*, 2008.

[29] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA—adaptive quality of service architecture," *Softw. Pract. Exper.*, vol. 39, no. 1, pp. 1–31, 2009.

[30] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, "Hierarchical Multiprocessor CPU Reservations for the Linux Kernel," in *OSPERT'09*, 2009.