Methods to Improve Bootup Time in Linux

Tim R. Bird
Sony Electronics
tim.bird@am.sony.com

Abstract

This paper presents several techniques for reducing the bootup time of the Linux kernel, including Execute-In-Place (XIP), avoidance of `calibrate_delay()`, and reduced probing by certain drivers and subsystems. Using a variety of techniques, the Linux kernel can be booted on embedded hardware in under 500 milliseconds. Current efforts and future directions of work to improve bootup time are described.

1 Introduction

Users of consumer electronics products expect their devices to be available for use very soon after being turned on. Configurations of Linux for desktop and server markets exhibit boot times in the range of 20 seconds to a few minutes, which is unacceptable for many consumer products.

No single item is responsible for overall poor boot time performance. Therefore a number of techniques must be employed to reduce the boot up time of a Linux system. This paper presents several techniques which have been found to be useful for embedded configurations of Linux.

2 Overview of Boot Process

The entire boot process of Linux can be roughly divided into 3 main areas: firmware, kernel, and user space. The following is a list of events during a typical boot sequence:

1. power on
2. firmware (bootloader) starts
3. kernel decompression starts
4. kernel start
5. user space start
6. RC script start
7. application start
8. first available use

This paper focuses on techniques for reducing the bootup time up until the start of user space. That is, techniques are described which reduce the firmware time, and the kernel start time. This includes activities through the completion of event 4 in the list above.

The actual kernel execution begins with the routine `start_kernel()`, in the file `init/main.c`.

An overview of major steps in the initialization sequence of the kernel is as follows:

- `start_kernel()`
  - init architecture
  - init interrupts
  - init memory
  - start idle thread
  - call `rest_init()`
    - start 'init' kernel thread

The `init` kernel thread performs a few other tasks, then calls `do_basic_setup()`, which calls `do_initcalls()`, to run through the array of initialization routines for drivers statically linked in the kernel. Finally, this thread switches to user space by `execve`-ing to the first user space program, usually `/sbin/init`.

- `init` (kernel thread)
  - call `do_basic_setup()`
    - call `do_initcalls()`
      - init buses and drivers
  - prepare and mount root filesystem
  - call `run_init_process()`
    - call `execve()` to start user space process

3 Typical Desktop Boot Time

The boot times for a typical desktop system were measured and the results are presented below, to give an indication of the major areas in the kernel where time is spent. While the numbers in these tests differ somewhat from those for a typical embedded system, it is useful to see these to get an idea of where some of the trouble spots are for kernel booting.

3.1 System

An HP XW4100 Linux workstation system was used for these tests, with the following characteristics:

- Pentium 4 HT processor, running at 3GHz
- 512 MB RAM
- Western Digital 40G hard drive on hda
- Generic CDROM drive on hdc

3.2 Measurement method

The kernel used was 2.6.6, with the KFI patch applied. KFI stands for "Kernel Function Instrumentation". This is an in-kernel system to measure the duration of each function executed during a particular profiling run. It uses the `-finstrument-functions` option of `gcc` to instrument kernel functions with callouts on each function entry and exit. This code was authored by developers at MontaVista Software, and a patch for 2.6.6 is available, although the code is not ready (as of the time of this writing) for general publication. Information about KFI and the patch are available at:

`http://tree.celinuxforum.org/pubwiki/moin.cgi/KernelFunctionInstrumentation`

3.3 Key delays

The average time for kernel startup of the test system was about 7 seconds. This was the amount of time for just the kernel and NOT the firmware or user space. It corresponds to the period of time between events 4 and 5 in the boot sequence listed in Section 2.

Some key delays were found in the kernel startup on the test system. Table 1 shows some of the key routines where time was spent during bootup. These are the low-level routines where significant time was spent inside the functions themselves, rather than in subroutines called by the functions.

Kernel Function	No. of calls	Avg. call time	Total time
delay_tsc	5153	1	5537
default_idle	312	1	325
get_cmos_time	1	500	500
psmouse_sendbyte	44	2.4	109
pci_bios_find_device	25	1.7	44
atkbd_sendbyte	7	3.7	26
calibrate_delay	1	24	24

Note: Times are in milliseconds.

Table 1: Functions consuming lots of time during a typical desktop Linux kernel startup.

Note that over 80% of the total time of the bootup (almost 6 seconds out of 7) was spent busywaiting in `delay_tsc()` or spinning in the routine `default_idle()`. It appears that great reductions in total bootup time could be achieved if these delays could be reduced, or if it were possible to run some initialization tasks concurrently.

Another interesting point is that the routine `get_cmos_time()` was extremely variable in the length of time it took. Measurements of its duration ranged from under 100 milliseconds to almost one second. This routine, and methods to avoid this delay and variability, are discussed in section 9.

3.4 High-level delay areas

Since `delay_tsc()` is used (via various `delay` mechanisms) for busywaiting by a number of different subsystems, it is helpful to identify the higher-level routines which end up invoking this function.

Table 2 shows some high-level routines called during kernel initialization, and the amount of time they took to complete on the test machine. Duration times marked with a tilde denote functions which were highly variable in duration.

Kernel Function	Duration time
ide_init	3327
time_init	~500
isapnp_init	383
i8042_init	139
prepare_namespace	~50
calibrate_delay	24

Note: Times are in milliseconds.

Table 2: High-level delays during a typical startup.

For a few of these, it is interesting to examine the call sequences underneath the high-level routines. This shows the connection between the high-level routines that are taking a long time to complete and the functions where the time is actually being spent.

Figures 1 and 2 show some call sequences for high-level calls which take a long time to complete.

In each call tree, the number in parentheses is the number of times that the routine was called by the parent in this chain. Indentation shows the call nesting level.

For example, in Figure 1, `do_probe()` is called a total of 31 times by `probe_hwif()`, and it calls `ide_delay_50ms()` 78 times, and `try_to_identify()` 8 times.

The timing data for the test system showed that IDE initialization was a significant contributor to overall bootup time. The call sequence underneath `ide_init()` shows that a large number of calls are made to the routine `ide_delay_50ms()`, which in turn calls

```
ide_init->
 probe_for_hwifs(1)->
  ide_scan_pcibus(1)->
   ide_scan_pci_dev(2)->
    piix_init_one(2)->
     init_setup_piix(2)->
      ide_setup_pci_device(2)->
       probe_hwif_init(2)->
        probe_hwif(4)->
         do_probe(31)->
          ide_delay_50ms(78)->
           __const_udelay(3900)->
            __delay(3900)->
             delay_tsc(3900)
          try_to_identify(8)->
           actual_try_to_identify(8)->
            ide_delay_50ms(24)->
             __const_udelay(1200)->
              __delay(1200)->
               delay_tsc(1200)
```

Figure 1: IDE init call tree

```
isapnp_init->
 isapnp_isolate(1)->
  isapnp_isolate_rdp_select(1)->
   __const_udelay(25)->
    __delay(25)->
     delay_tsc(25)
   isapnp_key(18)->
    __const_udelay(18)->
     __delay(18)->
      delay_tsc(18)
```

Figure 2: ISAPnP init call tree

`__const_udelay()` very many times. The busywaits in `ide_delay_50ms()` alone accounted for over 5 seconds, or about 70% of the total boot up time.

Another significant area of delay was the initialization of the ISAPnP system. This took about 380 milliseconds on the test machine.

Both the mouse and the keyboard drivers used crude busywaits to wait for acknowledgements from their respective hardware.

Finally, the routine `calibrate_delay()` took about 25 milliseconds to run, to compute the value of `loops_per_jiffy` and print (the related) `BogoMips` for the machine.

The remaining sections of this paper discuss various specific methods for reducing bootup time for embedded and desktop systems. Some of these methods are directly related to some of the delay areas identified in this test configuration.

4 Kernel Execute-In-Place

A typical sequence of events during bootup is for the bootloader to load a compressed kernel image from either disk or Flash, placing it into RAM. The kernel is decompressed, either during or just after the copy operation. Then the kernel is executed by jumping to the function `start_kernel()`.

Kernel Execute-In-Place (XIP) is a mechanism where the kernel instructions are executed directly from ROM or Flash.

In a kernel XIP configuration, the step of copying the kernel code segment into RAM is omitted, as well as any decompression step. Instead, the kernel image is stored uncompressed in ROM or Flash. The kernel data segments still need to be initialized in RAM, but by eliminating the text segment copy and decompression, the overall effect is a reduction in the time required for the firmware phase of the bootup.

Table 3 shows the differences in time duration for various parts of the boot stage for a system booted with and without use of kernel XIP. The times in the table are shown in milliseconds. The table shows that using XIP in this configuration significantly reduced the time to copy the kernel to RAM (because only the data segments were copied), and completely eliminated the time to decompress the kernel (453 milliseconds). However, the kernel initialization time increased slightly in the XIP configuration, for a net savings of 463 milliseconds.

In order to support an Execute-In-Place con-

Boot Stage	Non-XIP time	XIP time
Copy kernel to RAM	85	12
Decompress kernel	453	0
Kernel initialization	819	882
Total kernel boot time	1357	894

Note: Times are in milliseconds. Results are for PowerPC 405 LP at 266 MHz

Table 3: Comparison of Non-XIP vs. XIP bootup times

figuration, the kernel must be compiled and linked so that the code is ready to be executed from a fixed memory location. There are examples of XIP configurations for ARM, MIPS and SH platforms in the CELinux source tree, available at: `http://tree.celinuxforum.org/`

4.1 XIP Design Tradeoffs

There are tradeoffs involved in the use of XIP. First, it is common for access times to flash memory to be greater than access times to RAM. Thus, a kernel executing from Flash usually runs a bit slower than a kernel executing from RAM. Table 4 shows some of the results from running the `lmbench` benchmark on system, with the kernel executing in a standard non-XIP configuration versus an XIP configuration.

Operation	Non-XIP	XIP
stat() syscall	22.4	25.6
fork a process	4718	7106
context switching for 16 processes and 64k data size	932	1109
pipe communication	248	548

Note: Times are in microseconds. Results are for lmbench benchmark run on OMAP 1510 (ARM9 at 168 MHz) processor

Table 4: Comparison of Non-XIP and XIP performance

Some of the operations in the benchmark took significantly longer with the kernel run in the XIP configuration. Most individual operations took about 20% to 30% longer. This performance penalty is suffered permanently while the kernel is running, and thus is a serious drawback to the use of XIP for reducing bootup time.

A second tradeoff with kernel XIP is between the sizes of various types of memory in the system. In the XIP configuration the kernel must be stored uncompressed, so the amount of Flash required for the kernel increases, and is usually about doubled, versus a compressed kernel image used with a non-XIP configuration. However, the amount of RAM required for the kernel is decreased, since the kernel code segment is never copied to RAM. Therefore, kernel XIP is also of interest for reducing the runtime RAM footprint for Linux in embedded systems.

There is additional research under way to investigate ways of reducing the performance impact of using XIP. One promising technique appears to be the use of "partial-XIP," where a highly active subset of the kernel is loaded into RAM, but the majority of the kernel is executed in place from Flash.

5 Delay Calibration Avoidance

One time-consuming operation inside the kernel is the process of calibrating the value used for delay loops. One of the first routines in the kernel, `calibrate_delay()`, executes a series of delays in order to determine the correct value for a variable called `loops_per_jiffy`, which is then subsequently used to execute short delays in the kernel.

The cost of performing this calibration is, interestingly, independent of processor speed. Rather, it is dependent on the number of iter-

ations required to perform the calibration, and the length of each iteration. Each iteration requires 1 jiffy, which is the length of time defined by the HZ variable.

In 2.4 versions of the Linux kernel, most platforms defined HZ as 100, which makes the length of a jiffy 10 milliseconds. A typical number of iterations for the calibration operation is 20 to 25, making the total time required for this operation about 250 milliseconds.

In 2.6 versions of the Linux kernel, a few platforms (notably i386) have changed HZ to 1000, making the length of a jiffy 1 millisecond. On those platforms, the typical cost of this calibration operation has decreased to about 25 milliseconds. Thus, the benefit of eliminating this operation on most standard desktop systems has been reduced. However, for many embedded systems, HZ is still defined as 100, which makes bypassing the calibration useful.

It is easy to eliminate the calibration operation. You can directly edit the code in `init/main.c:calibrate_delay()` to hardcode a value for `loops_per_jiffy`, and avoid the calibration entirely. Alternatively, there is a patch available at `http://tree.celinuxforum.org/pubwiki/moin.cgi/PresetLPJ`

This patch allows you to use a kernel configuration option to specify a value for `loops_per_jiffy` at kernel compile time. Alternatively, the patch also allows you to use a kernel command line argument to specify a preset value for `loops_per_jiffy` at kernel boot time.

6 Avoiding Probing During Bootup

Another technique for reducing bootup time is to avoid probing during bootup. As a general technique, this can consist of identifying hardware which is known not to be present on one's machine, and making sure the kernel is compiled without the drivers for that hardware.

In the specific case of IDE, the kernel supports options at the command line to allow the user to avoid performing probing for specific interfaces and devices. To do this, you can use the IDE and harddrive `noprobe` options at the kernel command line. Please see the file `Documentation/ide.txt` in the kernel source tree for details on the syntax of using these options.

On the test machine, IDE `noprobe` options were used to reduce the amount of probing during startup. The test machine had only a hard drive on hda (ide0 interface, first device) and a CD-ROM drive on hdc (ide1 interface, first device).

In one test, `noprobe` options were specified to suppress probing of non-used interfaces and devices. Specifically, the following arguments were added to the kernel command line:

```
hdb=none hdd=none ide2=noprobe
```

The kernel was booted and the result was that the function `ide_delay_50ms()` was called only 68 times, and `delay_tsc()` was called only 3453 times. During a regular kernel boot without these options specified, the function `ide_delay_50ms()` is called 102 times, and `delay_tsc()` is called 5153 times. Each call to `delay_tsc()` takes about 1 millisecond, so the total time savings from using these options was 1700 milliseconds.

These IDE `noprobe` options have been available at least since the 2.4 kernel series, and are an easy way to reduce bootup time, without even having to recompile the kernel.

7 Reducing Probing Delays

As was noted on the test machine, IDE initialization takes a significant percentage of the total bootup time. Almost all of this time is spent busywaiting in the routine `ide_delay_50ms()`.

It is trivial to modify the value of the timeout used in this routine. As an experiment, this code (located in the file `drivers/ide/ide.c`) was modified to only delay 5 milliseconds instead of 50 milliseconds.

The results of this change were interesting. When a kernel with this change was run on the test machine, the total time for the `ide_init()` routine dropped from 3327 milliseconds to 339 milliseconds. The total time spent in all invocations of `ide_delay_50ms()` was reduced from 5471 milliseconds to 552 milliseconds. The overall bootup time was reduced accordingly, by about 5 seconds.

The ide devices were successfully detected, and the devices operated without problem on the test machine. However, this configuration was not tested exhaustively.

Reducing the duration of the delay in the `ide_delay_50ms()` routine provides a substantial reduction in the overall bootup time for the kernel on a typical desktop system. It also has potential use in embedded systems where PCI-based IDE drives are used.

However, there are several issues with this modification that need to be resolved. This change may not support legacy hardware which requires long delays for proper probing and initializing. The kernel code needs to be analyzed to determine if any callers of this routine really need the 50 milliseconds of delay that they are requesting. Also, it should be determined whether this call is used only in initialization context or if it is used during regular runtime use of IDE devices also.

Also, it may be that 5 milliseconds does not represent the lowest possible value for this delay. It is possible that this value will need to be tuned to match the hardware for a particular machine. This type of tuning may be acceptable in the embedded space, where the hardware configuration of a product may be fixed. But it may be too risky to use in desktop configurations of Linux, where the hardware is not known ahead of time.

More experimentation, testing and validation are required before this technique should be used.

IMPORTANT NOTE: You should probably not experiment with this modification on production hardware unless you have evaluated the risks.

8 Using the “quiet” Option

One non-obvious method to reduce overhead during booting is to use the `quiet` option on the kernel command line. This option changes the loglevel to 4, which suppresses the output of regular (non-emergency) printk messages. Even though the messages are not printed to the system console, they are still placed in the kernel printk buffer, and can be retrieved after bootup using the `dmesg` command.

When embedded systems boot with a serial console, the speed of printing the characters to the console is constrained by the speed of the serial output. Also, depending on the driver, some VGA console operations (such as scrolling the screen) may be performed in software. For slow processors, this may take a significant amount of time. In either case, the cost of performing output of printk messages during bootup may be high. But it is easily eliminated using the `quiet` command line option.

Table 5 shows the difference in bootup time of using the `quiet` option and not, for two different systems (one with a serial console and one with a VGA console).

9 RTC Read Synchronization

One routine that potentially takes a long time during kernel startup is `get_cmos_time()`. This routine is used to read the value of the external real-time clock (RTC) when the kernel boots. Currently, this routine delays until the edge of the next second rollover, in order to ensure that the time value in the kernel is accurate with respect to the RTC.

However, this operation can take up to one full second to complete, and thus introduces up to 1 second of variability in the total bootup time. For systems where the target bootup time is under 1 second, this variability is unacceptable.

The synchronization in this routine is easy to remove. It can be eliminated by removing the first two loops in the function `get_cmos_time()`, which is located in `include/asm-i386/mach-default/mach_time.h` for the i386 architecture. Similar routines are present in the kernel source tree for other architectures.

When the synchronization is removed, the routine completes very quickly.

One tradeoff in making this modification is that the time stored by the Linux kernel is no longer completely synchronized (to the boundary of a second) with the time in the machine's realtime clock hardware. Some systems save the system time back out to the hardware clock on system shutdown. After numerous bootups and shutdowns, this lack of synchronization will cause the realtime clock value to drift from the correct time value.

Since the amount of un-synchronization is up to a second per boot cycle, this drift can be significant. However, for some embedded applications, this drift is unimportant. Also, in some situations the system time may be synchronized with an external source anyway, so the drift, if any, is corrected under normal circumstances soon after booting.

10 User space Work

There are a number of techniques currently available or under development for user space bootup time reductions. These techniques are (mostly) outside the scope of kernel development, but may provide additional benefits for reducing overall bootup time for Linux systems.

Some of these techniques are mentioned briefly in this section.

10.1 Application XIP

One technique for improving application startup speed is application XIP, which is similar to the kernel XIP discussed in this paper. To support application XIP the kernel must be compiled with a file system where files can be stored linearly (where the blocks for a file are stored contiguously) and uncompressed. One file system which supports this is CRAMFS, with the LINEAR option turned on. This is a read-only file system.

With application XIP, when a program is executed, the kernel program loader maps the text segments for applications directly from the flash memory of the file system. This saves the time required to load these segments into system RAM.

Platform	Speed	console type	w/o `quiet` option	with `quiet` option	difference
SH-4 SH7751R	240 MHz	VGA	637	461	176
OMAP 1510 (ARM 9)	168 MHz	serial	551	280	271

Note: Times are in milliseconds

Table 5: Bootup time with and without the `quiet` option

10.2 RC Script improvements

Also, there are a number of projects which strive to decrease total bootup time of a system by parallelizing the execution of the system run-command scripts ("RC scripts"). There is a list of resources for some of these projects at the following web site:

`http://tree.celinuxforum.org/pubwiki/moin.cgi/BootupTimeWorkingGroup`

Also, there has been some research conducted in reducing the overhead of running RC scripts. This consists of modifying the multi-function program `busybox` to reduce the number and cost of forks during RC script processing, and to optimize the usage of functions builtin to the `busybox` program. Initial testing has shown a reduction from about 8 seconds to 5 seconds for a particular set of Debian RC scripts on an OMAP 1510 (ARM 9) processor, running at 168 MHz.

11 Results

By use of the some of the techniques mentioned in this paper, as well as additional techniques, Sony was able to boot a 2.4.20-based Linux system, from power on to user space display of a greeting image and sound playback, in 1.2 seconds. The time from power on to the end of kernel initialization (first user space instruction) in this configuration was about 110 milliseconds. The processor was a TI OMAP 1510 processor, with an ARM9-based core, running at 168 MHz.

Some of the techniques used for reducing the bootup time of embedded systems can also be used for desktop or server systems. Often, it is possible, with rather simple and small modifications, to decrease the bootup time of the Linux kernel to only a few seconds. In the desktop configuration of Linux presented here, techniques from this paper were used to reduced the total bootup time from around 7 seconds to around 1 second. This was with no loss of functionality that the author could detect (with limited testing).

12 Further Research

As stated in the beginning of the paper, numerous techniques can be employed to reduce the overall bootup time of Linux systems. Further work continues or is needed in a number of areas.

12.1 Concurrent Driver Init

One area of additional research that seems promising is to structure driver initializations in the kernel so that they can proceed in parallel. For some items, like IDE initialization, there are large delays as buses and devices are probed and initialized. The time spent in such busywaits could potentially be used to perform other startup tasks, concurrently with the ini-

tializations waiting for hardware events to occur or time out.

The big problem to be addressed with concurrent initialization is to identify what kernel startup activities can be allowed to occur in parallel. The kernel init sequence is already a carefully ordered sequence of events to make sure that critical startup dependencies are observed. Any system of concurrent driver initialization will have to provide a mechanism to guarantee sequencing of initialization tasks which have order dependencies.

12.2 Partial XIP

Another possible area of further investigation, which has already been mentioned, is "partial XIP," whereby the kernel is executed *mostly* in-place. Prototype code already exists which demonstrates the mechanisms necessary to move a subset of an XIP-configured kernel into RAM, for faster code execution. The key to making partial kernel XIP useful will be to ensure correct identification (either statically or dynamically) of the sections of kernel code that need to be moved to RAM. Also, experimentation and testing need to be performed to determine the appropriate tradeoff between the size of the RAM-based portion of the kernel, and the effect on bootup time and system runtime performance.

12.3 Pre-linking and Lazy Linking

Finally, research is needed into reducing the time required to fixup links between programs and their shared libraries.

Two systems that have been proposed and experimented with are pre-linking and lazy linking. Pre-linking involves fixing the location in virtual memory of the shared libraries for a system, and performing fixups on the programs of the system ahead of time. Lazy linking consists of only performing fixups on demand as library routines are called by a running program.

Additional research is needed with both of these techniques to determine if they can provide benefit for current Linux systems.

13 Credits

This paper is the result of work performed by the Bootup Time Working Group of the CE Linux forum (of which the author is Chair). I would like to thank developers at some of CELF's member companies, including Hitachi, Intel, Mitsubishi, MontaVista, Panasonic, and Sony, who contributed information or code used in writing this paper.

Proceedings of the Linux Symposium

Volume One

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*