Tools and Techniques for Reducing Bootup Time

By
Tim Bird
Sony Corporation

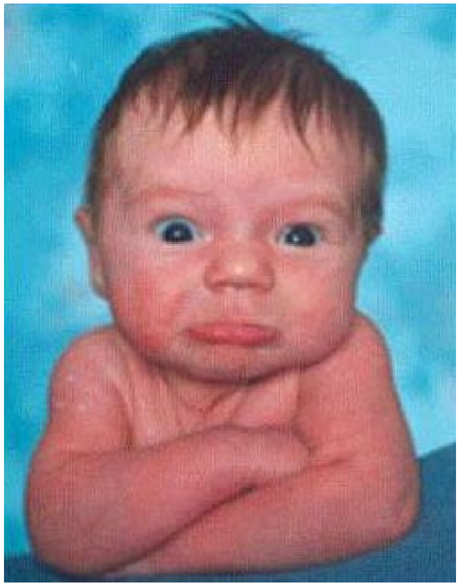
SONY

Agenda

- The problem
- Overview of boot phases
- Instrumentation
- Techniques for kernel
- Techniques for user space
- Final results / Conclusions
- Resources

The Problem

- Consumer electronics products require very fast boot times.
- Desired cold boot time for a digital still camera is less than 1 second.
  - Did I mention they have crummy, slow processors…?
- Consumer must catch the baby smiling!!



Overview of Boot Phases

- Firmware (bootloader)
  - Hardware probing
  - Hardware initialization
  - Kernel load and decompression
- Kernel execution
  - Core init (start_kernel)
  - Driver init (initcalls)
- User-space init
  - /sbin/init
  - RC scripts
  - Graphics start (First Impression)
- Application start
  - Application load and link
  - Application initialization
- First use

My hardware

- Old x86 desktop
  - X86 - Intel Celeron processor, running at 2 GHz.
  - 128 meg. of RAM and a 40G IDE hard drive.
  - Linux 2.6.27-rc7 from kernel.org
  - Sony distribution (CELinux) for x86.
- Some notes:
  - Most initcall and uptime timings are with NFS-root fs.
    - (IDE is present (probed and detected) but not mounted).
- Started at 4.91 seconds of uptime, at first shell prompt.

SONY

My Hardware (2)

- Old ARM Eval board
  - OSK - OMAP 5912 at 192 MHZ
  - 32 meg. RAM and 32 meg. of NOR Flash
  - MontaVista distribution (3.1 preview kit) for ARM
  - Linux 2.6.23.17 (with patches)
- Notes
  - Some tests done with Linux 2.6.27
- Started at uptime: 5.42
  - This is after RC scripts, but before first login

Instrumentation

Why Instrumentation

- Very important principle:

Premature optimization is the root of all evil.
- Donald Knuth

- Measure and find *the worst* problems first, or you just end up wasting a lot of time

Instrumentation

- System-wide:
  - Uptime (!!)
  - grabserial
- Kernel Measurement
  - Printk-times
  - initcall_debug
  - KFT
- User space measurement
  - Bootchart
  - Strace
  - Process trace - Tim’s quick hack
  - Linux Trace Toolkit

Uptime

- Easiest time measurement ever:
  - Add the following to /sbin/init or rc.local, or wherever you "finish" booting:

```
echo -n "uptime:" ; cat /proc/uptime
```

  - Note: Use of "echo" is wasteful, I'll get back to this later...
- Values produced are:
  - Wall time since timekeeping started
  - Time spent in the idle process (process 0)
- My x86 starting value:
  - For kernel, nfs fs mount, and short RC script
  - uptime: 4.91 3.04
- My ARM starting value
  - For kernel, flash fs mount, short RC script, some services:
  - uptime: 5.42 1.56

X86 user-space init overview

- On X86, /sbin/init is a shell script that:
  - Mounts /proc and /sys
  - Remounts root filesystem rw
  - Configures the loopback interface (ifconfig lo)
  - Runs /etc/rc.local
  - Starts syslogd, klogd, telnetd
  - Runs ‘free’
  - Runs a shell

ARM user-space init overview

- OSK has a “real” /sbin/init that processes /etc/inittab
- Init runs /etc/init.d/rcS with:
    - Mounts /proc
    - Configured loopback interface (ifconfig lo)
    - Mounts /tmp
    - Touches a bunch of files in /tmp
    - Starts syslogd, klogd, inetd, thttpd
        - Also does a ‘sleep 1’ !!)
    - Creates /dev/dsp nodes
    - Then inittab spawns a console getty

grabserial

- Utility for watching serial console output
- Is run on host machine, not target
  - Captures serial output and echos it
  - Can apply a timestamp to each line seen
- Easy to use:
  - Ex: grabserial –t –d /dev/ttyUSB0 –m “Starting kernel”

grabserial Example Output

```
[   22.774152] ## Booting image at 10000000 ...
[   22.776073]    Image Name:   Linux-2.6.27-00002-g1646475-dirt
[   22.780302]    Image Type:   ARM Linux Kernel Image (uncompressed)
[   22.784842]    Data Size:    1321228 Bytes =  1.3 MB
[   22.787127]    Load Address: 10008000
[   22.791150]    Entry Point:  10008000
[   22.792627]    Verifying Checksum ... OK
[   24.068948] OK
[   24.069267]
[   24.069367] Starting kernel ...
[    0.001231]
[    0.001334] Uncompressing Linux.......................................
.......... done, booting the kernel.
[    5.434655] serial console detected.  Disabling virtual terminals.
[    5.437749] init started:  BusyBox v0.60.2 (2004.04.16-00:49+0000) multi
-call binary
[    5.607621] 3.17 0.28
[    5.787597] mount: Mounting /tmpfs on /tmp failed: Invalid argument
[    6.947394] mknod: /dev/dsp: File exists
[    7.072378] 4.64 0.28
[    8.268232]
[    8.268373] MontaVista(R) Linux(R) Professional Edition 3.1, Preview Kit
[    8.291287]
[    8.291381] (none) login: root
```

grabserial Notes

- Pros:
  - Doesn’t put any instrumentation on target
  - Doesn’t slow down target – only consumes host cpu cycles
- Cons:
  - Kernel queues up printk messages during very early init
  - To measure time of kernel bootup events, you have to have kernel messages turned on
    - (I will talk about this later)
  - Bit of a pain to install.
    - Grabserial is a python program. It requires the python serial.py module, which is not shipped with python by default

Kernel Measurement

- Printk-times
- Initcall_debug
- Kernel Function Trace

Printk times

- Method to put timestamp on every printk
- Is better with a good resolution clock
- How to activate (use one of the following):
  - Compile kernel with: CONFIG_PRINTK_TIMES=y
  - Use “time=1” on kernel command line
  - Or, to turn on dynamically:
    - “echo Y >/sys/module/printk/parameters/time”

Printk Times Example

- Try it right now
  - If you have a laptop (or are reading this presentation on a Linux desktop) try this:
    - su root
    - echo Y >/sys/module/printk/parameters/time
    - <plug in a USB stick>
    - dmesg
  - To see relative times (deltas):
    - Use ‘show_delta’ script
    - Located in ‘scripts’ directory in Linux source tree
    - dmesg | linux_src/scripts/show_delta /proc/self/fd/0
      - (OK – I should change show_delta to be a filter)

Printk Times Sample Output

On ARM:

```
[    0.000000] Linux version 2.6.23.17-alp_nl-g679161dd (tbird@crest) (gcc version 4.1.1) …
[    0.000000] CPU: ARM926EJ-S [41069263] revision 3 (ARMv5TEJ), cr=00053177
[    0.000000] Machine: TI-OSK
[    0.000000] Memory policy: ECC disabled, Data cache writeback
[    0.000000] On node 0 totalpages: 8192
[    0.000000]   DMA zone: 64 pages used for memmap
…
[    0.000000] OMAP GPIO hardware version 1.0
[    0.000000] MUX: initialized M7_1610_GPIO62
[    0.000000] MUX: Setting register M7_1610_GPIO62
[    0.000000]       FUNC_MUX_CTRL_10 (0xfffe1098) = 0x00000000 -> 0x00000000
[    0.000000]       PULL_DWN_CTRL_4 (0xfffe10ac) = 0x00000000 -> 0x01000000
[    0.000000] PID hash table entries: 128 (order: 7, 512 bytes)
[  715.825741] Console: colour dummy device 80x30
[  715.825999] Dentry cache hash table entries: 4096 (order: 2, 16384 bytes)
[  715.826490] Inode-cache hash table entries: 2048 (order: 1, 8192 bytes)
[  715.832736] Memory: 32MB = 32MB total
[  715.832832] Memory: 28052KB available (3852K code, 396K data, 124K init)
[  715.833493] SLUB: Genslabs=22, HWalign=32, Order=0-1, MinObjects=4, CPUs=1, Nodes=1
[  715.833595] Calibrating delay loop (skipped)... 95.64 BogoMIPS preset
[  715.834196] Mount-cache hash table entries: 512
[  715.836419] CPU: Testing write buffer coherency: ok
[  715.847232] NET: Registered protocol family 16
[  715.860679] OMAP DMA hardware version 1
[  715.860773] DMA capabilities: 000c0000:00000000:01ff:003f:007f
[  715.868239] USB: hmc 16, usb2 alt 0 wires
[  715.904668] SCSI subsystem initialized
```

Printk Times Notes:

- On ARM, notice that timestamps are zero until clock is initialized
- On X86, timestamps are available immediately since it uses TSC, which is a built-in counter on the CPU
- On many embedded platforms, you need to fix the clock handling to get good timestamp values
  - Default printk_clock() returns jiffies
    - Only has 4 ms or 10 ms resolution)
  - Can call sched clock(), but you need to make sure not to call it too early
  - Newest kernel (2.6.27) uses cpu_clock()
  - For older kernels, I have a patch for some ARM platforms:
    - safe_to_call_sched_clock.patch

initcall_debug

- A good portion of bootup time is spent in 'initcalls'
- There's a flag already built into the kernel to show initcall information during startup
- On boot command line, use: initcall_debug=1
- After booting, do:
  dmesg -s 256000 | grep "initcall" | sed "s/\(.*\)after\(.*\)/\2 \1/g" | sort -n
- NOTE: It's a good idea to increase the printk log buffer size
  - Do this by increasing LOGBUF_SHIFT from 14 (16K) to 18 (256K)

Initcall_debug Example Output

```
24 msecs [    2.237177] initcall acpi_button_init+0x0/0x51 returned 0
28 msecs [    0.763503] initcall init_acpi_pm_clocksource+0x0/0x16c returned 0
32 msecs [    0.348241] initcall acpi_pci_link_init+0x0/0x43 returned 0
33 msecs [    0.919004] initcall inet_init+0x0/0x1c7 returned 0
33 msecs [    5.282722] initcall psmouse_init+0x0/0x5e returned 0
54 msecs [    2.979825] initcall e100_init_module+0x0/0x4d returned 0
71 msecs [    0.650325] initcall pnp_system_init+0x0/0xf returned 0
91 msecs [    0.872402] initcall pcibios_assign_resources+0x0/0x85 returned 0
187 msecs [    4.369187] initcall ehci_hcd_init+0x0/0x70 returned 0
245 msecs [    2.777161] initcall serial8250_init+0x0/0x100 returned 0
673 msecs [    5.098052] initcall uhci_hcd_init+0x0/0xc1 returned 0
830 msecs [    4.067279] initcall piix_init+0x0/0x27 returned 0
1490 msecs [    8.290606] initcall ip_auto_config+0x0/0xd70 returned 0
```

- Problem routines:
  - psmouse_init - unused driver!!
  - pnp_system_init - ??
  - pcibios_assign_resources - ??
  - ehci_hcd_init, uhci_hcd_init - part of USB initialization
  - serial8250_init - serial driver initialization
  - piix_init – IDE disk driver init
  - ip_auto_config - dhcp process

Kernel Function Trace (KFT)

- Instruments every kernel function entry and exit
- Can filter by time duration of functions
- VERY handy for finding boot latencies in early startup
- Unfortunately, this is a patch that was never mainlined
- See http://elinux.org/Kernel_Function_Trace
  - I would like to integrate KFT functionality into ftrace, but haven't had time yet
  - It's been on my "to do" list for years

KFT Trace Results Example

```
Entry  Delta  PID          Function                        Called At
    1      0    0              start_kernel  L6+0x0
   14   8687    0                setup_arch  start_kernel+0x35
   39    891    0              setup_memory  setup_arch+0x2a8
   53    872    0  register_bootmem_low_pages  setup_memory+0x8f
   54    871    0              free_bootmem  register_bootmem_low_pages+0x95
   54    871    0         free_bootmem_core  free_bootmem+0x34
  930   7432    0               paging_init  setup_arch+0x2af
  935   7427    0           zone_sizes_init  paging_init+0x4e
```

```
$ ~/work/kft/kft/kd -n 30 kftboot-9.lst
Function                  Count Time     Average  Local
------------------------- ----- -------- -------- --------
do_basic_setup                1  1159270  1159270       14
do_initcalls                  1  1159256  1159256      627
__delay                     156   619322     3970        0
delay_tsc                   156   619322     3970   619322
__const_udelay              146   608427     4167        0
probe_hwif                    8   553972    69246      126
do_probe                     31   553025    17839       68
ide_delay_50ms              103   552588     5364        0
isapnp_init                   1   383138   383138       18
```

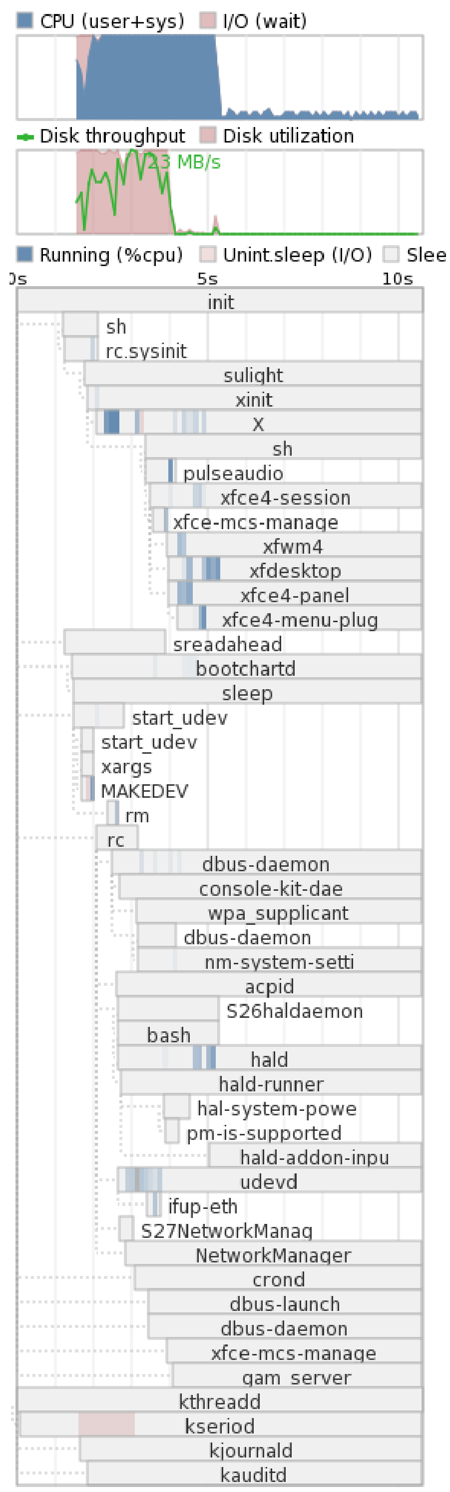
User-Space Measurement

- Bootchart
- Strace
- Process trace - Tim’s quick hack
- Linux Trace Toolkit

Bootchart

- Tool to display a nice diagram of processes in early boot
- Starts a daemon in early init
- Daemon collects information via /proc, and puts it into files in /var/log
- Has a tool to post-process the collected information, and prepare a nice diagram
  - PNG, SVG, or EPS
- Find it at: http://www.bootchart.org/

Bootchart Example Output



strace

- Strace can be used to collect timing information for a process
  - strace –tt 2>/tmp/strace.log thttpd …
- Can use to see where time is being spent in application startup
- Can also collect system call counts (-c)
- Can see time spent in each system call (-T)
- Great for finding extraneous operations
  - Eg. Wasteful operations, like scanning invalid paths for files, opening a file multiple times, etc.

strace Example Output

```
00:00:07.186340 mprotect(0x4001f000, 20480, PROT_READ|PROT_WRITE) = 0
00:00:07.200866 mprotect(0x4001f000, 20480, PROT_READ|PROT_EXEC) = 0
00:00:07.221679 socketcall(0x1, 0xbe842c70) = 3
00:00:07.235626 fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
00:00:07.248718 socketcall(0x3, 0xbe842c70) = -1 EPROTOTYPE (Protocol wrong type
 for socket)
00:00:07.264434 close(3)                = 0
00:00:07.286956 socketcall(0x1, 0xbe842c70) = 3
00:00:07.292816 fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
00:00:07.305603 socketcall(0x3, 0xbe842c70) = 0
00:00:07.327575 brk(0)                  = 0x24000
00:00:07.345397 brk(0x25000)            = 0x25000
00:00:07.360290 brk(0)                  = 0x25000
00:00:07.422485 open("/etc/thttpd/thttpd.conf", O_RDONLY) = 4
00:00:07.438049 fstat64(4, {st_mode=S_IFREG|0644, st_size=17592186044416, ...})
= 0
00:00:07.474121 old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONY
MOUS, -1, 0) = 0x40017000
00:00:07.490203 read(4, "#-------------------------------"..., 4096) = 1457
00:00:07.508544 read(4, "", 4096)       = 0
00:00:07.530151 close(4)                = 0
00:00:07.548675 munmap(0x40017000, 4096) = 0
00:00:07.561645 open("/etc/localtime", O_RDONLY) = -1 ENOENT (No such file or di
rectory)
00:00:07.585235 open("/etc/thttpd/throttle.conf", O_RDONLY) = 4
00:00:07.599182 gettimeofday({7, 603149}, NULL) = 0
00:00:07.613983 fstat64(4, {st_mode=S_IFREG|0644, st_size=17592186044416, ...})
= 0
00:00:07.637084 old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONY
MOUS, -1, 0) = 0x40017000
00:00:07.650604 read(4, "# thttpd 2.21b\n# Main throttle c"..., 4096) = 453
00:00:07.669586 read(4, "", 4096)       = 0
00:00:07.691589 close(4)                = 0
00:00:07.708099 munmap(0x40017000, 4096) = 0
```

strace Miscellaneous Notes

- Strace can follow children
- Strace adds of overhead to the execution of the program
  - Good for relative timings, not absolute
- Can’t get counts for a program that doesn’t end
  - If someone knows how to do this, let me know!
- I couldn’t figure out how to trace the whole system init
  - I tried replacing /etc/init.d/rcS in /etc/inittab with “strace –f -tt 2>/tmp/strace.log /etc/init.d/rcS ”
  - It didn’t work

Process Trace

- Process trace = Tim's own quick-and-dirty tracer
- Why?
  - Bootchart has problems:
    - Too much overhead for embedded
    - Reads lots of /proc frequently during boot
- I envisioned a kind of "Bootchart lite"
- Adds printks to fork, exec and exit in the kernel - Very simple
- Adds a script to process the dmesg output

SONY

Process Trace Example Output

```
[   37.162963] exec: 1 -> /sbin/init
[   38.189819] fork: 1 -> 15
[   38.203155] exec: 15 -> /etc/init.d/rcS
[   38.244598] fork: 15 -> 16
[   38.251708] exec: 16 -> /bin/mount
[   38.300262] exit: 16 - real 0.056 user 0.007 sys 0.039 nonrun 0.009
[   38.302429] fork: 15 -> 17
[   38.309509] exec: 17 -> /bin/cat
[   38.331481] exit: 17 - real 0.029 user 0.007 sys 0.015 nonrun 0.006
[   38.333312] fork: 15 -> 18
[   38.340362] exec: 18 -> /bin/mount
[   38.464355] exit: 14 - real 1.481 user 0.000 sys 0.390 nonrun 1.091
[   38.464752] fork: 18 -> 19
[   38.466979] exit: 18 - real 0.134 user 0.015 sys 0.007 nonrun 0.110
```

- Result of: “linux_src/scripts/procgraph –s d /target/tmp/bootprocs.msg”

```
PPid  Pid   Program                      Start    Duration Active   Idle
----- ----- --------------------- -------- -------- -------- --------
    0     1            /sbin/init    0.000    0.000    0.000    0.000
    0     2               unknown    0.000    0.000    0.000    0.000
    2     3              unknown*    0.000    0.000    0.000    0.000
. . .
   15    23           /bin/touch   38.644    0.106    0.014    0.092
   15    31       /usr/sbin/inetd   39.216    0.122    0.046    0.076
   15    18           /bin/mount   38.333    0.134    0.022    0.112
   15    25           /bin/touch   38.775    0.228    0.116    0.112
    1    14       [worker_thread]   36.983    1.482    0.390    1.092
    1    15       /etc/init.d/rcS   38.190   33.367    0.116   33.251
```

Process Trace Notes

- ‘procgraph’ script is badly named
  - It doesn’t produce a graph
  - I intended to copy Arjan’s bootgraph program, but didn’t have time
  - May be finished sometime soon
- It doesn’t replace bootchart, since it doesn’t show cpu or I/O utilization
- It’s good enough to find some problems
  - Like unexpected fork and execs

Linux Trace Toolkit

- Very nice tool for tracing “major” system events
- Good for showing process startup and interaction
- Has been out-of-mainline for many years
- See http://ltt.polymtl.ca/

Other Trace Systems

- trace_boot (being worked on right now)
  - Similar to process trace, but more comprehensive
    - Watches process schedules also
  - See fastboot git tree
  - Also search for "fastboot" subject lines on LKML
- SystemTap
  - Requires kernel loadable modules
  - Requires module insertion (user space must be up)
  - Should be easy to write a process trace tapset

Techniques for Reducing Bootup Time

Reduction Techniques for the Kernel

Reduction Techniques for the Kernel

- quiet console
- Eliminate unused drivers and features
- Deferred module initialization
- Reducing probing delays
- Filesystem tricks
- async initcall

quiet console

- Kernel spends significant time outputing chars to serial port during boot
- Can eliminate with simple runtime switch
- Add “quiet” to kernel command line
- Savings:
  - X86 savings: 1.32 seconds
  - ARM savings: 0.45 seconds
- Can still see all messages after booting with ‘dmesg’
- May also affect VGA console, but I haven’t measured

Eliminate Unused Features

- "The fastest code is the code you don't run!"
- Linux kernel defconfigs include lots of features not needed in product
  - They try to include all features of a platform or board
- Should eliminate as much as possible from kernel using CONFIG options
- This helps two ways:
  - Reduces the amount of initialization in the kernel
  - Reduces the kernel size, which reduces the time it takes the bootloader to load the kernel image
- Can look at Linux-tiny pages for ideas of things you can safely eliminate
- Also, use initcall_debug to see lots of modules you probably don't need

Example Unused Feature

- I found that CONFIG_HOTPLUG was turned on in the X86 defconfig
- Booting my X86 machine with Linux v2.6.27, there are 809 calls to execve /sbin/hotplug, during boot
- I eliminated these by turning CONFIG_HOTPLUG=n
- X86 savings: 1.34 seconds

Deferred module initialization

- Deferred module loading
  - Compile drivers as modules, and insmod after main boot
- Deferred initcall
  - Statically link modules (CONFIG_FOO=y)
  - Change module init routine to be run later, on demand
  - Add trigger for deferred initcalls, after main boot sequence
  - Patch is available to provide support for this feature

Deferred initcall Howto

- Find modules that are not required for core functionality of product
  - Ex: USB on a camera – uhci_hcd_usb, ehci_hcd_init
- Change module init routine declaration
  - module_init(foo_init) to
  - deferred_module_init(foo_init)
- Modules marked like this are not initialized during kernel boot
- After main init, do:
  - echo 1 >/proc/deferred_initcalls
- Deferred initcalls are run
- Also .init section memory is freed by kernel

Using deferred_module_init() (USB modules)

- Used deferred_module_init() on ehci_hcd_init and uhci_hcd_init
- Changed:
  - module_init(ehci_hcd_init) to
  - deferred_modle_init(echi_hcd_init)
  - Same change for uhci_hcd_init
- X86 savings: 530 ms

Using deferred_module_init() (piix_init)

- Used deferred_module_init(piix_init)
- X86 savings: 670 ms

- Total savings from just these three deferred_module_init()s = 1.2 seconds

Reduce Probing

- Reduce probe delays
  - Can often reduce probe delay on known hardware
  - Example: IDE probe, especially for flash devices masquerading as IDE block devices
  - It makes no sense to wait 50 milliseconds for the disk to respond on a solid state disk
- Eliminate probes for non-existent hardware
  - Look at kernel command line options for drivers you use
    - USB, IDE, PCI, network
  - Pass operational parameters directly to driver, which causes driver to bypass probing
  - See Documentation/kernel-parameters.txt

Reducing Probing Delays

- Preset LPJ – next page
- Network delays for IP autoconfig
- Passing device parameters from firmware

Preset-lpj

- Time to calibrate “loops_per_jiffy” can be long
- Can specify the value for lpj on kernel command line
  - This bypasses the runtime calibration
- How much time is saved depends on platform, CPU speed, HZ, etc.
  - ARM savings: 192 ms
  - X86 savings: 19 ms

Preset-lpj howto

- Example on ARM:
  - On target (example on ARM):
    - $ dmesg | grep -A 2 Bogo >/tmp/boot.txt
  - On host:

```
$ linux/scripts/show_delta /target/osk/tmp/boot.txt
[715.833569 < 715.833569 >] Calibrating delay loop... 95.64 BogoMIPS (lpj=478208)
[716.025733 < 0.192164 >] Mount-cache hash table entries: 512
[716.027787 < 0.002054 >] CPU: Testing write buffer coherency: ok
```

  - Add to kernel command line: "lpj=478208"

  - New timings:

```
[715.833595 < 715.833595 >] Calibrating delay loop (skipped)... 95.64 BogoMIPS preset
[715.834196 < 0.000601 >] Mount-cache hash table entries: 512
[715.836419 < 0.002223 >] CPU: Testing write buffer coherency: ok
```

SONY

Reducing Network Delays

- Example of finding a bogus delay and shortening it
- Generic mainline code has to work with every conceivable crummy piece of hardware
- Delays are often too long for specific hardware
- Patch on next page shows reduction in delay for IP autoconfig
- X86 savings: 1.4 seconds

Patch to Reduce Network Delay

```
diff --git a/net/ipv4/ipconfig.c b/net/ipv4/ipconfig.c
index 42065ff..e42d83f 100644
--- a/net/ipv4/ipconfig.c
+++ b/net/ipv4/ipconfig.c
@@ -86,8 +86,10 @@
 #endif

 /* Define the friendly delay before and after opening net devices */
-#define CONF_PRE_OPEN          500     /* Before opening: 1/2 second */
-#define CONF_POST_OPEN         1       /* After opening: 1 second */
+/*#define CONF_PRE_OPEN                500     /* Before opening: 1/2 second */
+/*#define CONF_POST_OPEN               1       /* After opening: 1 second */
+#define CONF_PRE_OPEN          5       /* Before opening: 5 milli seconds */
+#define CONF_POST_OPEN         10      /* After opening: 10 milli seconds */

 /* Define the timeout for waiting for a DHCP/BOOTP/RARP reply */
 #define CONF_OPEN_RETRIES      2       /* (Re)open devices twice */
@@ -1292,7 +1294,7 @@ static int __init ip_auto_config(void)
                return -1;

        /* Give drivers a chance to settle */
-       ssleep(CONF_POST_OPEN);
+       msleep(CONF_POST_OPEN);

        /*
         * If the config information is insufficient (e.g., our IP address or
```

Passing Device Params from Firmware

- Have firmware initialize hardware
  - It can sometimes do it faster because it doesn’t probe so much
- Have firmware pass information to kernel, for kernel driver to avoid probing and initializing hardware
- Sony used this in “snapshot boot”
- This is very firmware and hardware-specific
  - Don’t count on mainlining your work
- Devicetree does this in a general way??

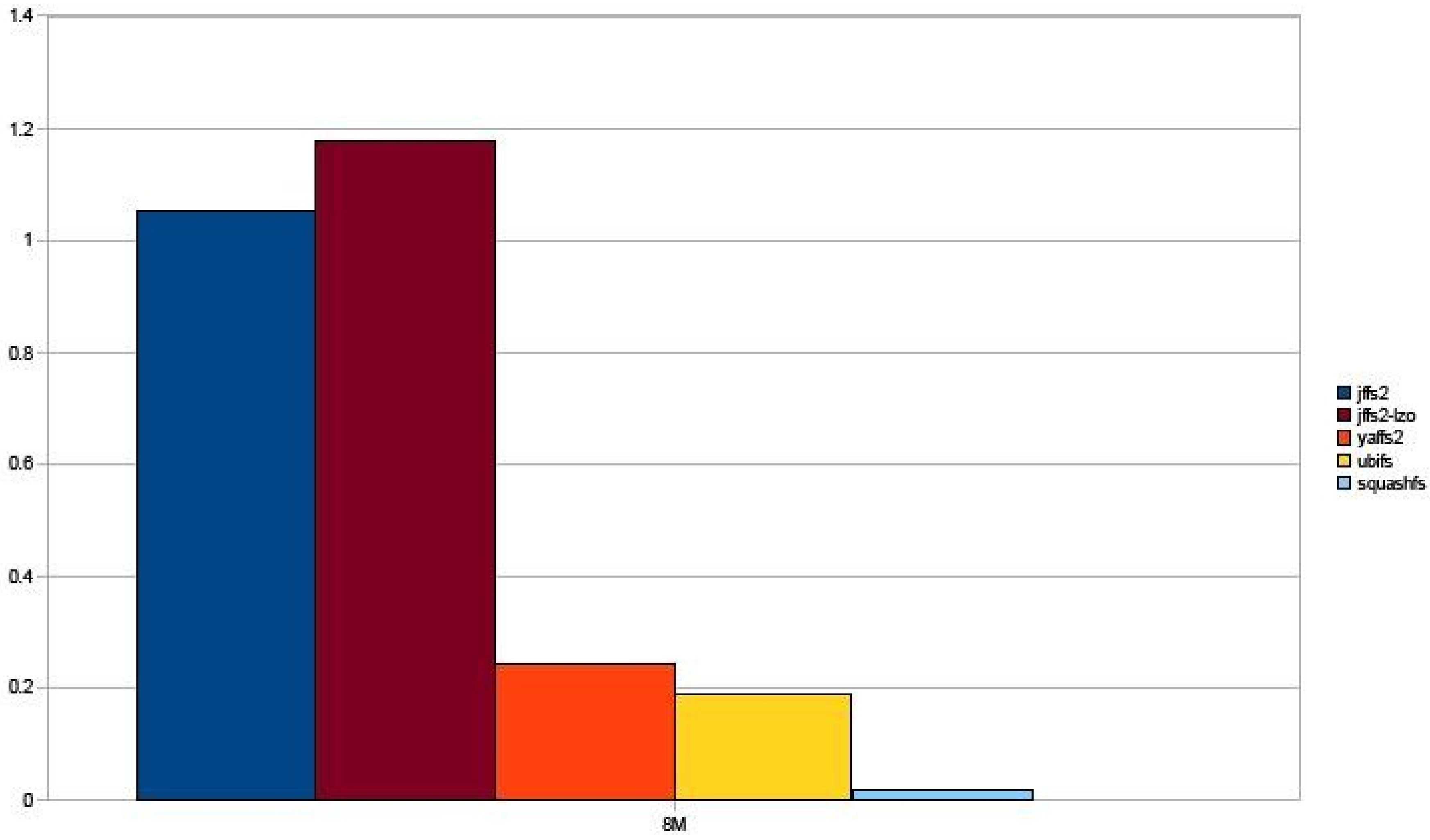
Filesystem Tricks

- Partition filesystem into read-only portion and read/write portion
  - Read-only file systems mount faster
- Mount filesystem faster:
  - Ex: Use UBIFS
  - Ex: Use CONFIG_JFFS2_SUMMARY

Filesystem Mount Time Comparison

- Next slide stolen shamelessly from Michael Opdenacker's presentation comparing flash filesystems
- JFFS2 mount of 8M filesystem partition is over 1 second
- UBIFS mount of 8M partition is under .2 seconds
- Squashfs mount of 8M partition is under 50 milliseconds (it looks like)

Zoom - Mount time (seconds) - 8M



async initcall

- Arjan Van de Ven wrote a system to allow aynchronous calling of initcalls
  - Allows initcalls in parallel
- Patches were put into fastboot.git tree
- Unfortunately, patch was rejected for 2.6.28 merge – Linus didn’t like it
- Back to the drawing board
  - Point of story: Likely there will be some new async capabilities for module initialization sometime soon

Reduction Techniques for User-space

Reduction Techniques for User-space

- Adjusting user-space init
  - Use custom init
  - Refactor RC scripts
  - Use builtins with busybox ash
- Readahead
- Prelinking
- Execute In Place (XIP)

Use custom init

- Should use busybox ‘init’ program instead of full-blown ‘init’
  - Probably already doing this, but good thing to check
- Kernel executes /sbin/init by default
  - Can change on kernel command line:
    - Try this sometime: “init=/bin/sh”
- Can have /sbin/init be a shell script
  - No login prompt, no getty, etc.
- Even better, make it a compiled program
  - No interpreter at boot time

SONY

Refactor RC scripts

- If you must use RC scripts, at least take out the junk
- Use ‘set –x’ to see all commands run
- Remove conditional code
- Eliminate unneeded actions
  - e.g. echo of completion status
- Start sub-processes in parallel so that idle and busy portions of applications can intersect
  - Watch out for load order dependencies

Use builtins with busybox ash

- Old versions of busybox did fork and exec for commands from shell interpreter
  - Ex. Echo “foo” – overhead 33 ms
- Newer busybox has support to execute echo, test and ‘[‘ directly
  - The commands are in the same binary
  - No need to instantiate another instance of busybox for these commands
- Process trace will tell you if you are exec’ing ‘echo’ or ‘[‘
- To fix this, use latest busybox and set config:
  - busybox 1.10 has (in shell/Config.in):
    - CONFIG_ASH_BUILTIN_ECHO
    - CONFIG_ASH_BUILTIN_TEST

Readahead

- Basic idea:
  - While system is doing other stuff, read blocks that will soon be needed
- In Arjan's testing, Readahead cut boot time from 7 seconds to 5 seconds
- sReadahead code is now included in moblin, I'm told
  - See moblin.org

Prelinking

- A good portion of application initialization time is spent resolving symbols to dynamic libraries
- Can reduce the time spent during dynamic linking
- Use prelink on applications to reduce linking time
- Philips reported 30% reduction in application load time, per application

XIP = Execute In Place

- Kernel XIP reduces bootloader time
- Application XIP is to reduce application load times
- New flash filesystem: AXFS
  - See Jared Hulbert’s presentation and YouTube videos from OLS
  - http://ols.fedoraproject.org/OLS/Reprints-2008/hulbert-reprint.pdf
  - http://www.youtube.com/watch?v=fu6Yj7iKEiA
  - http://www.youtube.com/watch?v=HUqFrA4FYd

Final Results

- X86 final boot time:
  - User-space init complete: .90 seconds
- ARM final boot time:
  - User-space init complete: 3.25 seconds
- Sony record:
  - Time for kernel boot (to userspace) in 110 milliseconds on a 192 MHZ ARM processor

Conclusions

- 1 second boot is within reach
  - Depending, of course, on your application initialization time
  - Should be able to boot embedded kernel in under 1 second
- It takes a lot of elbow grease, but it's getting easier

Resources

- Arjan Van de Ven's talk at LPC
  - "LPC: Booting Linux in 5 Seconds"
    http://lwn.net/Articles/299483/
- New fastboot git tree:
  - "What's in fastboot.git for 2.6.28"
    http://lwn.net/Articles/299591/
- Christopher Hallinan's talk at MV Vision
- elinux wiki – Boot Time development portal
  - Stuff for this presentation:
    http://elinux.org/Tims_Fastboot_Tools