Visualizing Resource Usage During Initialization of Embedded Systems

Matthew Klahn, Senior Software Engineer
Moosa Muhammad, Software Engineer

Motorola, Inc., Mobile Devices Sector
Purpose

• Time from power-on to “usable system” is a critical user satisfaction component for consumer electronic devices

• Quantitative analysis of system initialization is useful for many tasks (i.e. analyzing kernel initialization w/ printk-times patch)

• Qualitative analysis is “fuzzier”, but still useful
Why Visual Presentation of Resource Usage Data?

• User space initialization is somewhat complicated - straight numeric analysis could be difficult, without a full understanding of boot process

• All time spent from creation of init task to “usable system” will be considered the user space portion of system boot for this talk
bootchart is Born

- bootchart tool written by Ziga Mahkovec in answer to the challenge on the Fedora mailing lists in November, 2004 by Owen Taylor:
  - “The challenge is to create a single poster showing graphically what is going on during the boot, what is the utilization of resources, how the current boot differs from the ideal world of 100% disk and CPU utilization, and thus, where are the opportunities for optimization.”

- Allows system architects to see where system resources are being utilized, and where opportunities for optimization are available

- Linux distros such as Knoppix use bootchart to reduce boot time significantly through iterative improvements, starting with “low hanging fruit”, and re-examination of their boot process (e.g. pre-loading pages from CD-ROM all at once)
bootchart Design

- Replaces init w/ data collection shell script
- Reads resource usage data from /proc filesystem
- Writes (well, copies) data to files in /tmp/ bootchart.XXXXXX tempdir
- Data collection stops when trigger application is found to be running -- that is system is usable (e.g. gdm, xdm, X server, getty, etc.)
- Image creation done in separate step, after all data collection is completed by parser-renderer application
Awesome!

Let’s try this on our embedded target system and see where it’s spending all its time!
Not so good, eh?

- Using bootchart increased system boot time by at least order-of-magnitude

- What worked on a 1.5 GHz CPU w/ lots ‘o RAM & a fast HD doesn’t work so well on embedded system w/ limited resources
Causes of Performance Problems

Though the data collection tasks are “simple”, a shell script is an inappropriate choice for implementation of data collector

- Using shell commands cause each action to require a fork() / exec()

- When using commands like cat to copy files, each read() or write() requires a corresponding open() & following close()

- “polls” process list to find exit trigger event (uses pidof, which reads from /proc)

- These problems get much worse as:
  - # of processes increase (/proc/<pid> dirs are read)
  - sampling rate increases
How to Solve These Performance Problems?

Problems

1) fork()/exec() for each sample?
2) File I/O overhead too high?
3) Exit trigger event search too costly?
4) Poor performance scalability?

Solutions

Do all data collection in a single process
open() files once, read/write many times.
Use deterministic trigger.
Minimize impact of reading /proc files & directory.
embootchart Design Principles, I

- Meant to be an open source tool: tight and simple design & code
- Reuse the bootchart parser-renderer
  - Less code to write/maintain
  - Performance not an issue, so why bother?
  - This application is quite well done!
- Adds a restriction: embootchart must now be data compatible w/ bootchart (no format changes of output data files)
embootchart Design Principles, 2

• Performance is absolutely critical to reduce skewing the results

• Because embedded systems should generally have shorter boot times, may need to increase sampling rate

  • bootchart default: 5 samples/sec.

  • embootchart default: 20 samples/sec.
How to Achieve Higher Performance

- Single compiled, multi-threaded application
- Tasks divided one-per-thread
- Input & output files open()’d one time, read() & write multiple times
  - Output files use open(..., O_WRONLY | O_APPEND)
  - Input files use open(..., O_RDONLY) and lseek(fd, 0, SEEK_SET) to refresh data before sampling (/proc files!)
- Use asynchronous notification to stop data collection: signal from init script or modified application (e.g. gdm)
Because the data collector is easy to separate into discrete tasks, OO seemed a logical design choice.

Reasonably high performance & small code size (~350 LoC, 77kb binary, dynamically linked).

libstdc++ functionality for file I/O, string manipulation, etc. reduces LoC I need to write/maintain, and is higher performance than handwritten code while shielding complexity.

Simplest way to get there from here.
System-wide info:
- CPU type,
- hostname,
- kernel version, etc.

Singleton

Main thread:
main() of executable

Thread abstraction:
1 Collector per output file

Per-process data collector:
Specialization of Collector, reads /proc/<pid>/stat files

Input file reader:
Opens one /proc file & uses lseek() to refresh contents, then reads out data

Timestamp info:
Calc'd from # of jiffies consumed by system since boot
Singleton
embootchart Data Acquisition Sequence

- embootchart launched
  - Run pre-initialization executable/script (optional)
  - fork(): run “real” init (i.e. /sbin/init) in parent process (pid 1), child process goes on to do data collection
  - Start & initialize data collector threads & let them collect data; main thread waits for exit signal
  - When exit signal (SIGSTOP) is rec’d, stop all data collector threads
  - Run post-processing executable/script (optional)
- embootchart exits
Pre-init script

Responsible for special set-up for data collection, not normally done for system boot

- Mount tmpfs for temp r/w of data output files
- Mount /proc fs for data acquisition
- Pre-init hardware setup (console setup, device symlinking, etc.)
- May replace a script already run on system (e.g. linuxrc)
Post-processing script

Responsible for packaging data & setting up resources for data export from target system

- tar & gzip datafiles, which is how parser-renderer expects them to be
- Set up networking or NFS filesystem for data export
- Any cleanup of data files, etc. after export, if your system is going to continue to run
embootchart Customization

- Modify Makefile to customize gross functionality
  - Whether to collect disk usage statistics (2.6.x kernels or higher)
  - Whether to run pre-init script
  - Whether to run post-init script
- Modify Config.hh file to fine-tune parameters at compile time
  - Sampling rate (default: 20 samples/sec.)
  - Filepath to “real” init process
  - Filepaths to pre-init and post-processing scripts
  - Location of rw filesystem to output data files
Contrived Example: Mainstone Reference Platform

- 200 MHz Intel XScale (PXA270, iwmmxt_le) CPU
- 16-bit memory bus
- 32 entry text & data TLB caches
- 64 MB RAM
- 32 MB Tyax Flash (NOR)
- Based on MontaVista 3.1 CEE Linux distribution (glibc 2.3.2, gcc-3.2)
- Linux 2.6.10 kernel (upgrade over MVLCEE 3.1)
- busybox 1.00-rc3
Warning!

I said “contrived” on the previous slide because this is not a real-world target system. Due to the simplicity of its purpose (i.e. show some bootcharts!), there are not as many low-hanging fruit as I would like for demonstration purposes.
Boot chart for (none) (Thu Jan 01 00:00:28 UTC 1970)
uname: Linux version 2.6.10_dev-mainstone_pxa27x (fishburn@virgil) (gcc version 3.4.3 (MontaVista 3.4.3-21.0.0.custom 2005-04-13)) #10 Fri May 20 17:55:02 CDT 20
release: MontaVista(R) Linux(R) Consumer Electronics Edition 3.1
CPU: (0)
kernel options: root=/dev/mtdblock2 console=ttyS0,115200 mem=64M
time: 0:33
What Next?

• Reduce need for external script/executable for pre-init & post-processing stages by writing common tasks (e.g. mounting filesystems) as C++ modules

• Extend data collection to cover memory usage

• Attempt to identify further performance improvements in existing code (e.g. remove the user of /proc all together?!)
Great!
Where do I get it?

- Process started to open-source this project & distribute to CELF & RotW
- When approved, embootchart will be released under GPL (most likely)
- Will publicize on CELF mailing list
- This should happen fairly quickly (weeks, not months)
Copyright, © 2006 by Matthew Klahn, Motorola, Inc.