Linux Plumbers Conference

Vienna, Austria | September 18-20, 2024

Initiatives in Boot Time Reduction – Boot time markers, Boot phases and Automated optimizations

Tim Bird
Sony Corporation

LINUX PLUMBERS CONFERENCE Vienna, Austria / Sept. 18-20, 2024

Abstract

This session is intended to present and discuss 3 different technology areas surrounding boot-time reduction for Linux systems: 1) boot time markers, 2) boot phases, and 3) automated boot-time optimizations. Boot markers is a proposed set of well-define measurement points in the Linux boot process, used for testing improvements and regressions in boot time. "Boot phases" refers to dividing the kernel boot process into two distinct phases: a time-critical phase and non-time-critical phase, and investigating how to initialize time-critical drivers and features, while still supporting full operation of a system in the long term. Finally, automated boot-time optimizations refers to utilizing run-time data from one instantiation of the kernel, to drive the optimization of subsequent instantiations, through things like an init data cache, that holds probed values, that can be incorporated into re-compilations of the kernel source to shorten boot times on dedicated or specialized hardware. Finally, I would like to discuss how to instantiate a working group of developers in the area of boot time reduction, when there is no centralized maintainer for this "feature" of the kernel.

Outline

- Boot markers
- Boot phases
- Automated optimizations
- Discussion
- Next Steps
- Resources

Boot Markers

Boot markers

- Boot time regression testing
- Rationale for standardization
- Proposed markers, tools and tests
  - Boot markers
  - Show_delta (v2) tool

Boot time regression testing issues

- There are lots of instrumentation and tools for measuring boot time, but no actual test of boot time
  - Regions selected for measurement are ad hoc
  - It's like a weight loss clinic that consists only of a mirror
- Need to pick a consistent set of time durations in the kernel to measure, and report those
- initcalls have too many, and are not comprehensive
- printks messages depend on config, and are not fixed strings

Rationale for standardization

- Need to standardize boot periods, in order to collaborate on working on them
- Agree on periods with:
  - Known regions across multiple machines and configs
  - Tractable number of data points
  - Good "cohesion" of data (routines within a measured region are related (e.g. by kernel sub-system or maintainer)

Proposed markers and methods

- Marker options:
  - Printk statements
  - Trace events
  - Some other output mechanisms (early serial or fixed gpio)
  - Use existing functions, using ftrace or grabserial to capture times
- Marker desired attributes:
  - Same "style" of markers for bootloader, kernel and user-space
  - Same markers can be used with multiple measuring tools
  - Markers can have small output length, if desired to reduce overhead
  - Markers are both human readable and automation-friendly

Boot markers

- Propose "boot markers" system,with the following attribute:
  - Markers have 3 parts: prefix, number, and string
  - Prefix is one of "B", "K", or "U"
  - Number is 1-9
  - String = initialization region name
- Rationale:
  - When limiting output bandwidth, just prefix and number can be used:
    - e.g. K1, U3
    - this can be converted to a short gpio signal
  - Region name gives a human-friendly string
- Add a function "boot_marker(int, char *)" to init/main.c
  - Call if from some places

Current proposed boot markers

```
[    0.000000] Boot Marker: K1 core init
[    0.000000] Boot Marker: K2 memory init
[    0.000333] Boot Marker: K3 output init
[    0.001315] Boot Marker: K4 network init
[    0.002223] Boot Marker: K5 arch init
[    0.034184] Boot Marker: K6 driver init
[    0.042816] Boot Marker: K8 initcall pure
[    0.043275] Boot Marker: K8 initcall core
[    0.046570] Boot Marker: K8 initcall postcore
[    0.048962] Boot Marker: K8 initcall arch
[    0.063728] Boot Marker: K8 initcall subsys
[    0.072041] Boot Marker: K8 initcall fs
[    0.083326] Boot Marker: K8 initcall device
[    0.174192] Boot Marker: K8 initcall late
[    2.869689] Boot Marker: K9 run init process
```

show_delta.py

- Existing tool in upstream tree (scripts/show_delta)
- Shows time delta for each line of timestamped kernel log messages
- Recent work:
  - Accept regular expressions to indicate regions to measure
    - start expressions, end expressions (regular expressions)
  - Output in KTAP format (see my presentation on KTAP benchmark support)

show_delta output

```
$ show_delta -s "Boot Marker" -t dmesg-bp1-*quiet*
KTAP 1.0
1..15
value Boot_Marker_K1_core_init_duration = 0.000000 s
unknown 1 Boot_Marker_K1_core_init
value Boot_Marker_K2_memory_init_duration = 0.000342 s
unknown 2 Boot_Marker_K2_memory_init
value Boot_Marker_K3_output_init_duration = 0.000468 s
unknown 3 Boot_Marker_K3_output_init
value Boot_Marker_K4_network_init_duration = 0.000889 s
unknown ...
```

Boot Phases

Boot phases

- Boot phases = divide boot into critical and non-critical phases
- Do only essential initializations during the critical boot phase
- Allowing deferred initializations
- Kernel already supports module loading after boot
- What else can be deferred?
  - initcalls
  - memory (partially)
  - what else?

Deferment mechanisms

- Deferred initcalls
- Deferred memory initialization
- Init dependency re-ordering

Deferred initcalls

- This is an old patch, that was used in Sony products 10 to 15 years ago
- Patch recently updated to 6.11 kernel
- Changed to support runtime operation
  - Adds support for "initcalls_defer=<comma-separate-list>" kernel command line argument
  - Saves indicated initcall functions on a list
  - Sometime after initialization, user or system reads /proc/deferred_initcalls
    - Calls the functions on the deferred_initcalls list

Deferred initcalls - issues

- Frees init memory only after deferred initcalls are run
- Some initcalls may be depended on by other
- Should be safe to defer most initcalls in the 'late' stage
  - And maybe those in the 'driver' stage
- Needs testing to see which initcalls can be safely deferred, and what the impact is on other functions

Deferred memory initialization

- See talk "Deferred Memblocks Init for Boot Time Reduction"
  - by Sudarshan Rajagopalan, Qualcomm (ELC NA 2024)
  - https://elinux.org/images/f/f3/Deferred_Memory_Block_Init-EOSS-2024.pdf

Init dependency reordering

- Changing SysV init.rc ordering
  - Introduce a deferred init ordering
- Changing system unit ordering
  - Introduce a critical target
  - critical target is loaded (with dependencies, if any), before normal target

Automated Optimization

Automated boot-time optimizations

- Boot cache
- Statically compiled pre-initialization
- Boot-time tuning tool

Why automate pre-initializations

- Pre-initializations represent a specialization of the init code, that doesn't apply generally to other kernel users
- Changes are inappropriate for acceptance upstream
- Solution is to support automation of the specializations, so that individual developers can adjust their compiled kernels to match their hardware

Boot cache

- Idea: record values from one boot, and use them to reduce time in a subsequent boot
- This works with embedded, because in some products the hardware does not change

- Have a "record" phase
- And an "apply recordings" phase

Boot cache operations

- "Record" phase:
  - Add "boot_cache_record" to kernel command line
  - Add routines that emit the data that was probed or discovered in device tree
    - Probe, then print
    - Lookup, then print
- "Apply recording" phase:
  - Produce C code with hardcoded values to replace the probe or device-tree lookup code.
  - Apply automatically to the kernel and recompile
  - Use macros to hide whether we're in record or apply mode.

Boot-time tuning tool

- Read dmesg or run on target machine, and detect problems
- And then suggest solutions

- Example:
  - Detect if clk_disable_unused is in initcalls, and takes more than 0 usecs
    - Recommend adding 'clk_ignore_unused' to kernel command line
  - Ask if Bluetooth is required for critical boot
    - Recommend deferring bt module loading
  - Detect if network is:
    - Negotiating link speed -> recommend ethtool settings to reduce delay
    - Using dhcp to acquire address -> recommend static IP address assignment

Next Steps

Moving forward with boot time work

- There's no "boot time" maintainer in the kernel
- No ecosystem of boot time developers
- How to collaborate?

Resources

- https://elinux.org/Boot_Time
- Tim's presentation from ELC Europe 2008
  - https://elinux.org/images/d/d2/Tools-and-technique-for-reducing-bootup-time.pdf
- Andrew Murray presentation from ELC Europe 2010
  - https://elinux.org/images/f/f7/RightApproachMinimalBootTimes.pdf
- Alexandre Belloni presentation from ELC 2012
  - https://elinux.org/images/d/d1/Alexandre_Belloni_boottime_optimizations.pdf
- TI presentation from EOSS 2024
  - https://elinux.org/images/2/20/EOSS24-BootTimeOptimization_AS_ST.pdf

Thanks!

Questions or comments?
email: tim.bird@sony.com

Barriers to boot time optimization

- boot-time issues (like size, power, performance and realtime) are "death by a thousand cuts".
  - There's no single technical area that can be improved to achieve overall results
  - There's no single maintainer
- Boot time optimizations may interfere with other goals
  - Such as generality of the code, maintainability

Previous boot phase categories (sample 1)

- 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 or system
    - RC scripts to system units
    - Graphics start (First Impression)
  - Application start
    - Application load and link
    - Application initialization
    - First use

initcall-duration-test.py

- Reads kernel log for initcall data
- Reports duration of each initcall, in KTAP format
- Checks against references values
  - From a 'known good' run

- Issues:
  - There are lots of initcalls during startup
  - List of initcalls varies a lot based on machine and configuration
  - Initcall function durations can be affected by instrumentation method
    - serial console, log level, ftrace settings
  - Deferred operations are not accounted for by initcall_debug instrumentation