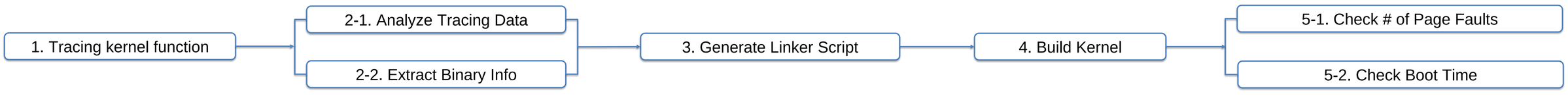
Reducing WebOS Boot Latency through Build and Boot-Time Optimization

연세대학교 스케일러블 시스템 소프트웨어 연구실
Scalable System Software Laboratory

팀명: BOOST
지도교수 : 정진규
2020199005 황종현　2020162029 박성현
2015147529 김창원　2020147529 임성윤

Overall Research Flow



1. Former Research and Limitation

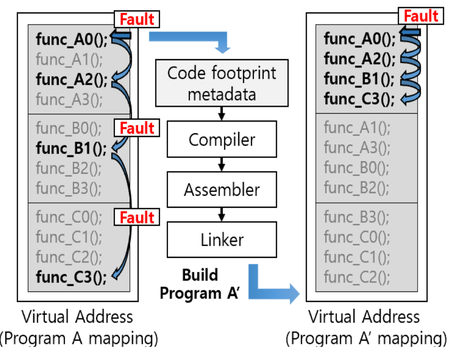
Former Optimization Research

- **Warm booting**: a method of restarting the system through software without turning off the power
- **Snapshot booting**: a method that either executes only essential units or restores a previously saved system state from a prior boot
- **Replacing bootloader and flie system:** a method that removes unnecessary decompression steps and adopts a hybrid file system to accelerate kernel loading and root filesystem mounting.

Limitation

- **Hardware & Software Dependency :** Optimizations rely on the underlying hardware and software stack — such as storage device, kernel configurations, init systems, etc — making them difficult to apply across different platforms or OS environments.
- **High Development Cost :** Fine tuning for each target platform requires expertise and development time.

2. Objective : Reducing Page Faults via Binary Relocation



Page fault: occurs when the processor requests a page that is not in physical memory, triggering disk I/O and causing significant delays that can slow the boot process.

Linker : A linker processes ELF object and libraries, resolves symbol references, and combines them into a single ELF executable by arranging code and data into their final memory segments

Linker script and ELF: A linker script defines how ELF sections and segments are organized—mapping them to memory addresses and grouping them into program segments—to control the final binary's layout.

We compiled with **`--ffunction-sections`** to place each function in its own ELF section, then link with **`--section-ordering-file`** (specified in our linker script) to arrange those sections in a predefined order—determined via `nm` based symbol and size analysis for optimized binary layout.

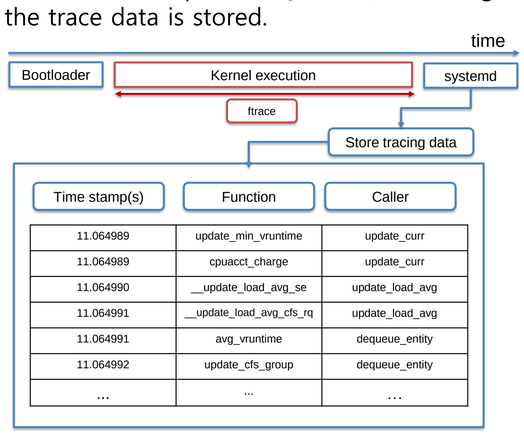
By defining the section order based on runtime profiling data and adding it into our linker script without modifying original source code, we tried to lay out frequently executed functions sequentially in the final ELF binary. This coalesced placement is intended to improve instruction cache efficiency and minimize page faults, with the goal of reducing start up and overall execution times.

3. Data Analysis & Implementation

Data Analysis

1. Collection of boot-time function traces

By enabling `ftrace` in the kernel boot argument, we captured the entire sequence of kernel function calls during the boot process(initial kernel execution ~ `systemd`). The diagram below shows our boot-time tracing workflow. `ftrace` is enabled throughout the kernel execution phase to capture each function call's timestamp, function name, and caller. Once init process(`systemd`) is running, the trace data is stored.



2. Page access analysis

Using `nm`, we analyzed the kernel binary to extract each function's address and then computed the number of memory pages accessed based on the location and size of each function.

```
ffffffc080010008 000000000000004c t bcm2835_handle_irq
ffffffc080010060 0000000000000070 t bcm2836_arm_irqchip_handle_irq
ffffffc0800100e0 00000000000000c8 t gic_handle_irq
ffffffc0800101b8 0000000000000130 t gic_handle_irq
ffffffc0800102f8 0000000000000028 T __do_softirq
ffffffc080010800 0000000000000800 T vectors
ffffffc080011000 000000000000007c t __bad_stack
ffffffc080011080 0000000000000068 t el1t_64_sync
ffffffc0800110e8 0000000000000068 t el1t_64_irq
ffffffc080011150 0000000000000068 t el1t_64_fiq
```

We used these data to analyze memory utilization of the code section, count of accessed pages during booting. We found the code section to span across 5,375 pages, 964 of which were accessed during booting. However, the functions executed during booting only consumes 1,231,168 bytes total, which can theoretically be compacted into 301 pages

```
Page range: 0xF_FFFF_FC08_0010, 0xF_FFFF_FC08_150E
Page count: 5375
Pages accessed: 964
Minimum required: 1,231,168 B -> 301 pages
```

Implementation

In Linux, since kernel memory is non-swappable, once a page fault has occurred on a page, revisiting that page will not trigger another fault. Therefore, the exact placement of functions is not critical so long as those which are accessed at least once are coalesced together. Thus, we only need to compile with –section-ordering-file to define the relative order of functions.

Extract Binary Info

- Using nm, extract each symbol's address and size from ELF binary
- Match each traced function with its address and size from the nm output to its location in binary

Generate Linker Script

- Compile with –ffunction-sections so each function has its own section
- Based on trace data, create an ordering file for functions in the .text region.

Build & Benchmark

- Rebuilding the kernel with section-ordering-file
- Verify the ELF binary(vmlinux) has changed.
- Gather and compare the page related metrics.

4. Result & Conclusion

Page faults & Binary Size

For each kernel binary(vmlinux), we examined the virtual address range and pages accessed at boot time of the code section. Lastly, we checked the on-disk sizes of both vmlinux(uncompressed) and Image(compressed) to measure the impact of our reordering

	Original	Optimized	Difference
Page Range	0xFFFFFFC080010~ 0XFFFFFFC08150E	0XFFFFFFC080010~ 0XFFFFFFC0814FE	-
Total # of Pages	5375	5359	-16 pages
Accessed # of Pages	964	322	-66.6%
Theoretical Minimum Accessed # of Pages	301		-

Table 1. Comparison of Code Section Memory Page Metrics. (Total # of Page: total pages spanned by code section, Accessed # of Pages : pages actually accessed during kernel execution, Theoretical Minimum Accessed # of pages: smallest # of page set covering all access.

From the result, we verify that the `.text` section's range is changed. `.text` section still starts at the same address (`0xFFFFFFC080010`), but end of address is slightly decreased from `0xFFFFFFC08150E` to `0xFFFFFFC0814FE`. That is, by reordering functions in the vmlinux, we reduced its code section footprint from 5,375 to 5,359 pages. More importantly, we cut the number of pages actually used at boot time by 66.6%, from 964 to 322 pages.

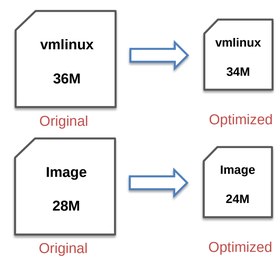


Figure 1. Size reduction of kernel binary and image after optimization

- **vmlinux** : the kernel ELF binary size shrinks from 36 MB to 34 MB (5.5% reduction) by compacting code layout
- **Kernel Image**: the stripped and bootable kernel image is cut from 28 MB to 24 MB (14% reduction)

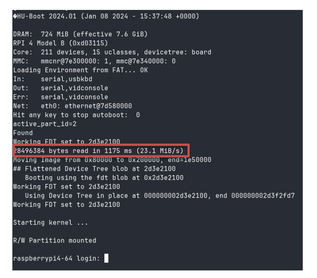


Figure 2. U-Boot log showing system initialization and the SD card read speed, as highlighted in red box

A U-Boot log shows a read speed of 23.1MB/s from the SD card. Since the bootloader reads data at the same speed for both the original and optimized images, the 4MB reduction in Image size would result in a load time reduction of approximately 0.173s.

Boot Time

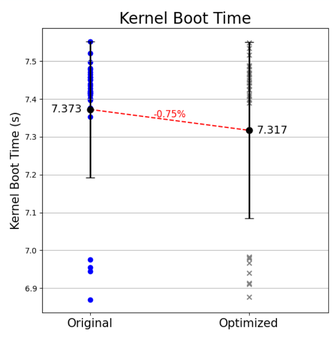


Figure 3. Kernel boot time comparison between original and optimized builds. This chart compares kernel initialization times across 30 runs for both original and optimized boot time on kernel portion.

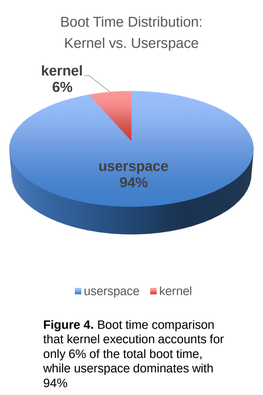


Figure 4. Boot time comparison that kernel execution accounts for only 6% of the total boot time, while userspace dominates with 94%

Figure 3 presents a comparison of kernel execution times during boot, based on 30 trials and their corresponding averages.
The result shows the optimized kernel achieved 0.76% reduction in kernel execution time on average.
The measurable gain at the kernel level demonstrates the effectiveness of function reordering in reducing page faults.
However, as shown in Figure 2, the kernel phase accounts for only a small fraction(6%) of the total boot time (approximately 7s out of 1m 53s).
Therefore, these results suggest potential for further optimization when extended to userspace components.

Conclusion

We traced kernel code execution and wrote a custom Python parser script to generate a function ordering file. Passing this file to the compiler and the linker, we produced an optimized ELF binary without modifying any kernel source (only altering the build process). As a result, we reduced the number of boot time accessed pages by 66.6%. This is especially meaningful given that kernel memory is unswappable, so minimizing initial page access directly reduces page faults. Additionally, the reordered layout led to a smaller final binary, which in turn reduced the size of the compressed kernel image. This helps the bootloader load the kernel faster. When we booted the system with the optimized kernel, the average kernel startup time reduced by 0.76%.

Limitation

Modest Impact in Kernel:
Since kernel memory is never swapped out in Linux, a page fault occurs only on its first access and revisiting it does not incur additional page faults. As a result, we only apply function reordering without page alignment, which simplifies the implementation, but offers limited reduction in page faults throughout the system runtime.

Suboptimal Compaction:
We unable to achieve the ideal 301 pages because some kernel functions had stricter requirements in their placements, leaving us with 322 pages instead of 301. However, further fine tuning may be possible.

Modest Impact in Overall Boot Time:
The boot time is roughly comprised of 7s (kernel) + 1m 46s (userspace). As the portion of time spent in kernel execution is low, the performance gains in the kernel had limited impact in the overall boot time. Nevertheless, it demonstrates the technique to be effective.

5. Further Study

Applying to user-space applications : Since user-space memory is swappable, hot code can be grouped in page units to minimize swap out. By organizing code in this way, the number of demand paging events during operation can be significantly reduced, which in turn enhances memory locality and minimizes page faults. In future work, we are planning to apply this technique to user applications like Node.js and Chromium, using `uftrace` to bring ftrace-style tracing into user space.

Optimization Section Reordering Algorithm Development: In our study, we divided the code section based on boot time usage. However, by incorporating function call relationships, we can develop a specialized reordering algorithm that accounts for page swap and invocation frequency. This can result in further optimized binary layouts for each user-space application tailored to its specific execution patterns.

6. Reference

[1] H. Jo, H. Kim, J. Jeong, J. Lee and S. Maeng, "Optimizing the startup time of embedded systems: a case study of digital TV," in IEEE Transactions on Consumer Electronics, vol. 55, no. 4, pp. 2242-2247

[2] K. Ho Chung, M. Sil Choi and K. Seon Ahn, "A Study on the Packaging for Fast Boot-up Time in the Embedded Linux," 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), Daegu, Korea (South), 2007, pp. 89-94

[3] http://korea.gnu.org/manual/release/ld/ld-sjp/