CS 350 Operating Systems

Project Report

Kernel Configurations and Performance Assessment

Efe Atçalı, Arda Cirit, Eray Ocak

Instructor: Ismail Arı

Department of Computer Science

January 4, 2026

Contents

Abstract

This report extends the baseline boot time study from Part 1 by changing and evaluating optimizations. Experiments cover eight different Linux configurations including native x86_64 systems and an ARM64 Parallels virtual machine, for distros: Debian 12 and Arch Linux, and for storage USB and NVMe. The optimization range is intentionally limited to make it measurable and controllable across the full test matrix: the bootloader (**Loader**) and the Linux kernel (**Kernel**) as reported by `systemd-analyze`. Each change is evaluated using five cold-boot trials before and after modification, with results summarized using mean and sample standard deviation. Applied optimizations include simplifying the boot path by transitioning from GRUB to `systemd-boot`, reducing kernel overhead through configuration trimming and driver integration decisions (with USB boot-safety constraints), adjusting compression choices for the kernel/initramfs, and mitigating a platform-specific Parallels AHCI bottleneck. Firmware and userspace times are recorded for context and traceability, but they are not treated as primary optimization targets.

1 Introduction

1.1 Project Continuation and Objectives

In the previous report (Part 1), the baseline for boot times was established by measuring cold-boot performance for the platform environments that is decided. These measurements provided an idea on how distribution, architecture, and storage medium affect the boot process. This report builds upon this baseline and explores the optimization possibilities, implementation challenges, trade offs between the efficiency and system versatility among related matters.

The primary objective is to reduce boot time through kernel and boot-loader modifications that directly influence the early boot path, while keeping the experiment group and measurement method identical to Part 1. All improvements are evaluated using before/after cold-boot averages, and reported using absolute and normalized percentages to fair comparison across samples.

1.2 Scope of the Optimization Phase

Optimization phase is focused around the boot-loader and kernel times mainly because the firmware and user space times are not fundamentally reflects the boot performance as firmware times are heavily depends on the vendor, therefore not customizable and the user space startup times are often compromised due to desired functionalities such as daemons and display servers (Wayland, X11). Regardless, firmware time and userspace time are recorded for context and traceability, but they are not treated as primary optimization targets. This constraint is critical for fair comparison. Within this scope, the applied changes include simplifying the boot path by migrating from GRUB to **systemd-boot**, reducing kernel overhead through configuration trimming, compression strategy adjustments, and initramfs policy decisions that balance size reduction with boot safety, and mitigating platform-specific bottlenecks.

1.3 Measurement Methodology

For each test configuration, five independent cold-boot trials were conducted. The system was fully powered off between runs to eliminate residual state and caching effects. Boot timing was captured using `systemd-analyze` [9], which provides a consistent phase breakdown (e.g., firmware, loader when available, kernel, userspace) and the total time to reach the target.

After collecting five runs per configuration, the arithmetic mean and sample standard deviation were computed for each reported phase and for the total boot time. Reporting results as $\mu \pm \sigma$ reduces sensitivity to outliers and provides a transparent measure of variability, enabling fair comparison across platforms and storage media. All raw `systemd-analyze` outputs were preserved for traceability, and the averaged outcomes are summarized in the baseline and optimization tables in the following sections.

`systemd-analyze` was selected as the primary timing method because it is available and comparable across both native and virtual machine environments. Alternative approaches explored during the project were not consistently applicable inside VMs (due to hypervisor abstractions and missing low-level visibility), whereas `systemd-analyze` remains stable, reproducible, and phase-consistent across the full test matrix. One remark with the systemd-analyze is the distro specific behaviour where it displays the kernel and initramfs loading times separately in Arch Linux instances where as Debian includes the initrd times to kernel. It is considered to present the totals consistently with Debian results however it is kept as discrete results because it provided some additional traceability for those measurements.

2 Baseline Summary and Control Configuration

The baseline phase (Part 1) established a reproducible measurement framework across **eight test configurations** spanning distribution, architecture, and storage medium. Each configuration was measured using five cold boots under consistent setup controls, producing phase-level timing breakdowns from `systemd-analyze`. These baseline results serve as the control reference for all optimization comparisons in this final report.

Table 1: Test Configuration Summary (T1-T8)[1]

Test ID	Architecture	Distribution	Storage Type
T1	ARM (Parallels VM)	Debian 12	USB flash drive
T2	ARM (Parallels VM)	Debian 12	NVMe SSD
T3	ARM (Parallels VM)	Arch Linux	USB flash drive
T4	ARM (Parallels VM)	Arch Linux	NVMe SSD
T5	Intel x86 (bare metal)	Arch Linux	USB flash drive
T6	Intel x86 (bare metal)	Arch Linux	NVMe SSD
T7	Intel x86 (bare metal)	Debian 12	USB flash drive
T8	Intel x86 (bare metal)	Debian 12	NVMe SSD

[1]The test configurations T5 and T7 is measured again because the original installations were lost.

2.1 Summary of Baseline Findings

The baseline results highlighted clear system-level trends. NVMe-based configurations consistently achieved faster and more predictable startup behavior than USB-based configurations, reflecting lower storage latency and faster device readiness in early boot. In addition, distribution-level differences were observable in multiple configurations, with Arch-based setups generally exhibiting shorter kernel and userspace phases under comparable conditions.

Virtual machine measurements demonstrated that hypervisor abstractions and virtual storage behavior can dominate certain phases, sometimes masking native-storage effects. For that reason, baseline findings were interpreted primarily through phase breakdowns (kernel/userspace/loader when available) rather than relying only on total boot time.

Table 2: Stock-Average cold boot durations by test configuration (mean ± std. dev., in sec). ("N/A" show that a phase is not applicable in that environment.)

Test	**Firmware (s)**	**Loader (s)**	**Kernel (s)**	**Userspace (s)**	**Initrd (s)**	**Total (s)**
T1	N/A	N/A	2.54 ± 0.03	7.38 ± 0.50	N/A	9.92 ± 0.49
T2	N/A	N/A	2.94 ± 0.02	4.39 ± 0.20	N/A	7.33 ± 0.21
T3	0.72 ± 0.06	1.57 ± 0.45	2.06 ± 0.01	1.70 ± 0.15	1.39 ± 0.09	7.44 ± 0.59
T4	1.12 ± 0.16	0.21 ± 0.01	2.08 ± 0.01	1.45 ± 0.01	1.33 ± 0.06	6.19 ± 0.15
T5	7.15 ± 0.01	2.60 ± 0.13	0.78 ± 0.00	3.80 ± 0.24	6.70 ± 2.14	21.04 ± 1.99
T6	15.45 ± 0.05	1.17 ± 0.00	0.79 ± 0.00	3.35 ± 0.01	1.98 ± 0.28	22.74 ± 0.34
T7	16.894 ± 0.243	2.776 ± 0.061	44.719 ± 0.342	91.478 ± 0.347	N/A	155.868 ± 1.19
T8	3.41 ± 0.26	4.44 ± 0.32	3.53 ± 0.04	10.74 ± 0.18	N/A	22.12 ± 0.43

Since the USB was put into read-only mode new (see 4.2.3) USB was used with same hardware and therefore stock boot times of that setup are updated as shown in Table below.

Table 3: (T7) Stock but New Boot Time Breakdown — Intel Debian (USB)

Run	Firmware (s)	Loader (s)	Kernel (s)	Userspace (s)	Total (s)
1	16.904	2.794	44.590	91.577	155.867
2	17.231	2.681	45.102	91.204	156.218
3	16.587	2.842	44.231	92.019	155.679
4	17.018	2.755	45.004	91.463	156.240
5	16.732	2.809	44.667	91.128	155.336
Avg	16.894	2.776	44.719	91.478	155.868
Std. Dev.	0.243	0.061	0.342	0.347	1.19

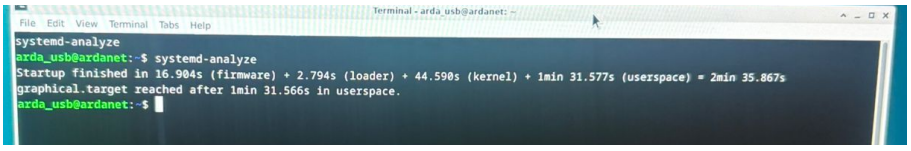

Figure 1: Stock but New Boot Time Breakdown — Intel Debian (USB)

2.2 Control Configuration

For each test ID, the Part 1 baseline measurements were treated as the **control state** (*pre-optimization*). All optimized results in this report are evaluated relative to that control using the same cold-boot procedure and reporting format (mean and sample standard deviation over five trials).

During the optimization phase, limited rework was required for a subset of configurations due to experimental constraints that affected traceability (e.g., removable-media lifecycle issues during UKI exploration). When re-measurement was necessary, the system was rebuilt to match the original Part 1 control envelope (same distribution version, same storage class, same boot path assumptions, and the same measurement discipline) however the boot times were completely different compared to first setup eventhough the same debian installation was done several times. These cases are explicitly noted in the relevant setup sections to ensure that before/after comparisons are consistent.

3 Experimental Environment and System Setup

The second phase of this project focused on the comprehensive optimization of the boot process, moving beyond the initial baseline measurements established in Part 1. This section details the hardware platforms, installation procedures, and the specific kernel and bootloader configurations used to achieve maximum performance while maintaining system stability across native environments.

3.1 Native Environment

The native environment was selected to measure the impact of boot optimizations on physical x86_64 hardware without hypervisor abstractions. Native runs provide a high-fidelity reference for changes that directly affect the `Loader` and `Kernel` phases reported by `systemd-analyze`.

3.1.1 Hardware Platform and Constraints

Measurements were performed across two generations of AMD Ryzen platforms to observe how CPU capability influences early-boot decompression and kernel execution:

- **Debian native configurations (USB and NVMe):** AMD Ryzen 6000-series.
- **Arch native configurations (USB and NVMe):** AMD Ryzen 7000-series.

All native tests used two storage classes:

- **External removable storage:** USB 3.2 Gen1 flash drives.
- **Internal baseline storage:** NVMe SSD.

During the UKI evaluation stage, the primary Kingston DataTraveler 3.2 flash drive, which was used for Intel-Debian-Usb setup in Part 1, entered to hardware-level read-only (write-protected) state. After several hours of fixing trials, setup was recreated. To preserve measurement integrity, the USB setup was migrated to a same hardware setup but with USB 3.2 Gen1 device under the same control assumptions used in the baseline.

3.1.2 Linux Installation Procedure

All native systems were installed from official distribution ISO images to ensure a consistent starting point. Manual partitioning was applied using:

- a 500 MB EFI System Partition (FAT32),
- an ext4 root partition.

To reduce unnecessary I/O overhead during startup:

- swap was disabled to avoid paging behavior over the USB interface during boot.

3.1.3 Kernel Build Environment

A custom kernel build workflow was used to enable controlled configuration changes and reproducible before/after comparisons. For the native optimization track, Linux Kernel **6.12.6** was used as the working kernel baseline for tuning and validation.

3.1.4 Bootloader Setup and Configuration

Bootloader work targeted the `Loader` phase. Native optimized setups transitioned from GRUB to **systemd-boot** [10], with a minimal and explicit entry configuration (kernel, initramfs, and command line) to reduce variability and pre-kernel overhead.

3.2 Virtual Machine Environment

3.2.1 Virtualization Platform and Architecture

Virtual machine measurements were conducted using **Parallels Desktop** running on Apple Silicon hardware (MacBook Pro M4). This environment was selected to evaluate boot optimization strategies under hypervisor abstractions and to observe how virtual hardware presentation affects kernel initialization behavior.

Each guest VM was allocated 2 CPU cores and 4096 MB of RAM. The hypervisor presents virtual hardware through UEFI firmware (EDK II), and systemd correctly detects the Parallels virtualization context during boot. Both **Arch Linux ARM (aarch64)** and **Debian 12 (arm64)** were tested as guest operating systems to assess distribution-level differences under identical virtual hardware conditions.

3.2.2 Virtual Storage and Firmware Abstractions

The virtual storage topology in Parallels Desktop introduces platform-specific behavior that proved critical to kernel boot time optimization. Guest disk images are exposed through an **AHCI platform controller**, identified in the kernel as ACPI device `PRL4010:00`. This controller reports 6 SATA ports as implemented (port mask `0x3f`), even though only a single virtual disk is attached.

During kernel initialization, the AHCI/libahci driver enumerates all six ports (`ata1` through `ata6`). While `ata1` successfully links to the virtual disk, ports `ata2` through `ata6` report "link down" after sequential probing. This per-port scanning behavior consumed

a significant portion of the kernel boot budget in the baseline configuration.

Additionally, the AHCI controller advertises the **Staggered Spin-up (SSS)** capability flag. In physical systems, SSS is used to manage power draw during drive initialization by sequencing port power-up. However, in this virtualized environment, the SSS flag triggers serialized scanning behavior in the Linux AHCI driver, amplifying the time cost of probing unused ports. This platform-specific characteristic became the primary target for kernel-level optimization and is discussed in detail in Section 4.3.

3.2.3 Guest OS Installation and Configuration

Both guest operating systems were installed from official ARM64 distribution images using manual partitioning. Each VM used a 500 MB EFI System Partition (FAT32) and an ext4 root partition. To reduce unnecessary variability during boot measurements, swap was disabled in both configurations.Virtual disk images were hosted on internal NVMe SSD storage and a high-performance USB 3.2 flash drive.

3.2.4 Kernel Build Environment

Custom kernel builds for the ARM guest VMs followed a minimization strategy tailored to the Parallels virtual hardware contract. The baseline configuration ensured that virtualization-specific drivers remained available:

- **AHCI platform support:** Required for the `PRL4010:00` device.
- **VirtIO subsystem:** Retained for GPU, network, balloon, and vsock devices exposed by the hypervisor.
- **EFI and ACPI support:** Required for UEFI boot and device enumeration.

The trimming process removed unnecessary subsystems (e.g., physical hardware drivers not present in VMs, legacy protocols, unused filesystems) while preserving the minimal driver set required for stable operation under Parallels. For Arch Linux ARM, kernel version **6.18.2** was used as the working baseline for configuration tuning.

4 Optimization Methodology

Optimization focused on measurable and directly controllable boot phases: **Loader** and **Kernel**. Each optimization was evaluated using before/after cold-boot averages (five trials per state), and improvements are reported both as absolute reductions and as normalized ratios (e.g., $\Delta L/L_{\text{total}}$, $\Delta K/K_{\text{total}}$, L:loader, K:kernel) to support fair comparison across platforms and storage media.

4.1 Kernel-Level Optimizations

Kernel-level work aimed to reduce boot time by trimming unnecessary features/drivers and by enforcing deterministic early-boot support where the storage interface requires it (especially USB). The kernel configuration phase was treated as a structured *reduction and integration* problem rather than complete useless minimization. The objective was to

minimize early-boot work while preserving functional correctness across all experimental setups. To achieve this, broad areas of kernel functionality that were unlikely to be required in the target environments were identified first, prioritizing options that plausibly increase kernel image size, initramfs complexity, or driver probing during boot.To validate configuration decisions and understand module/driver purpose, the Linux Kernel Driver Database (LKDDB) was used as the primary reference [8].

4.1.1 Kernel Version and Configuration Approach

The optimization workflow was applied to Linux Kernel **6.12.6** through selective configuration changes designed to reduce kernel footprint and early initialization work.Based on baseline observations and Linux boot internals, potential optimizations were grouped into high-level areas: legacy hardware support, unused networking subsystems, non-essential filesystem drivers, debugging/tracing infrastructure, virtualization features not applicable to the execution context, and the placement of critical storage and graphics drivers across the kernel image, initramfs, and loadable modules. This grouping provided a logical way to reason about trade offs instead of disabling options in isolation. A total of **38 candidate kernel configuration changes** were identified and organized into functional groups as mentioned above.

For native **SSD/NVMe** configurations, the full **38-change set** was applied, since storage discovery is direct and early-boot dependencies are less fragile. For native **USB** configurations, only **20 changes** were applied; the remaining items were intentionally excluded to preserve removable-media boot safety and portability (e.g., retaining EFI-relevant filesystems and avoiding aggressive driver trimming that could break early root discovery). This staged approach allows the performance gains of broader trimming on SSD to be captured while maintaining stability on USB-boot systems.

Table 4: Identified kernel optimization targets (38 total) grouped by category.

Category	Kernel config targets	Count
Legacy hardware	CONFIG_PCCARD, CONFIG_PCMCIA, CONFIG_CARDBUS, CONFIG_PARPORT, CONFIG_BLK_DEV_FD	5
Networking	CONFIG_ATM, CONFIG_CAN, CONFIG_HAMRADIO, CONFIG_ISDN, CONFIG_INFINIBAND	5
Filesystems	CONFIG_JFS_FS, CONFIG_XFS_FS, CONFIG_BTRFS_FS, CONFIG_F2FS_FS, CONFIG_NTFS3_FS, CONFIG_NTFS_FS, CONFIG_HFS_FS, CONFIG_HFSPLUS_FS	8
Debugging / tracing	CONFIG_DEBUG_BUGVERBOSE, CONFIG_DEBUG_KERNEL, CONFIG_MAGIC_SYSRQ, CONFIG_DEBUG_FS, CONFIG_SLUB_DEBUG, CONFIG_LOCKUP_DETECTOR, CONFIG_SCHED_INFO, CONFIG_FTRACE	8
CPU / virtualization	CONFIG_X86_NATIVE_CPU, CONFIG_HYPERVISOR_GUEST, CONFIG_PARAVIRT, CONFIG_XEN, CONFIG_KVM_GUEST	5
Built-in drivers	CONFIG_NVME_CORE, CONFIG_BLK_DEV_NVME, CONFIG_MMC, CONFIG_MMC_BLOCK, CONFIG_DRM_AMDGPU, CONFIG_DRM_NOUVEAU, CONFIG_VIDEO_DEV, CONFIG_MD, CONFIG_RAID6_PQ, CONFIG_RAID6_PQ_BENCHMARK	7
	Total	**38**

4.1.2 Compression Strategy: Transition to LZ4

Kernel and initramfs compression were transitioned from GZIP to **LZ4**. The goal was to reduce early-boot wall-time by prioritizing decompression speed:

- **Performance rationale:** LZ4 can produce a larger compressed artifact than stronger compressors, but manual trimming (removing unnecessary driver batches) helps keep overall payload size bounded.
- **USB relevance:** on USB, early boot performance is sensitive to both I/O throughput and decompression overhead; faster decompression reduces time spent in the critical early pipeline.

4.1.3 Initramfs Configuration and Stability

Initramfs policy was interface-dependent:

- **NVMe/SSD systems:** remained stable with `MODULES=dep`, producing a leaner initramfs aligned with direct PCIe storage discovery.
- **USB systems:** attempts to use `MODULES=dep` resulted in a boot drop to an initramfs BusyBox shell due to missing early-boot bridging drivers. USB systems were stabilized using `MODULES=most` to ensure the root device is discoverable during early boot.

4.1.4 Driver Integration and Built-in Requirements

For USB boot safety, critical drivers were changed from modules to **built-in** (`=y`) so the kernel can mount the root filesystem without depending on late module availability:

- USB XHCI host controller support,
- USB mass storage support,
- ext4 filesystem support.

4.1.5 Stability Exclusions (USB Safety)

From the 38 tracked kernel targets, **18 were explicitly not applied** to USB-based native configurations to preserve stability and interoperability. Table 5 summarizes representative exclusions.

Table 5: Representative kernel optimizations excluded to maintain USB system stability.

Category	**Optimizations Excluded**	**Reason for Exclusion**
Filesystems	NTFS, NTFS3, F2FS, HFS, HFSPLUS	Cross-platform removable-media compatibility.
Boot Safety	FAT_FS, MSDOS_FS	Required to access the EFI System Partition.
USB Drivers	MMC, MMC_BLOCK	Risk of drive non-recognition at boot on some controllers/bridges.
Graphics	AMDGPU, NOUVEAU	Built-in GPU drivers may trigger early-boot stalls/timeouts on USB.
Portability	X86_NATIVE_CPU	Kept generic to avoid device-specific brittleness.

4.1.6 Investigating *make tinyconfig* and *make localmodconfig*

The source project of Linux kernel also offers presets and scripts for different types of kernel configurations. The whole lists can be seen in make help target, however among those targets, make tinyconfig and make localmodconfig aligns the most with the premise of this project that is optimizing the boot times by removing the unnecessary modules. By the target descriptions tinyconfig produces the smallest possible kernel with the downside

of unrelaiblity in most hardware. On the contrary the localmodconfig script looks at the loaded drivers and disables anything that is not presently used by the machine and kernel. In the testing both targets are observed to cut down on compile and installation(make modules-install and make install) times massively with *tinyconfig* being more prominent than other. Unfortunately, tinyconfig did not boot as expected therefore not used in the optimizations list. The localmodconfig target on the other hand did not brake anything that is already working expectedly and resulted a real boost but it didn't adopted to use in the project because even though it booted and runned normally it removed a lot of utility and feature from the OS. To name a few; bluetooth, ethernet port, audio input-output, HDI (human-device interface) drivers for mouse and keyboards were not available afterwards because those are not at the use when the localmodconfig is generated. Due to this findings both configurations didn't made it to the final list of optimizations.

4.2 Bootloader Optimization

To minimize pre-kernel latency, the bootloader path was simplified by transitioning from GRUB to **systemd-boot** [10]. Compared to GRUB, systemd-boot uses a smaller configuration surface and avoids script/module discovery overhead, providing a more direct EFI-to-kernel handoff.

4.2.1 Transition to systemd-boot

The optimized native setups used systemd-boot as the primary boot manager:

- **Efficiency:** eliminates GRUB module search and complex script processing overhead.
- **Deterministic behavior:** minimal boot entries (kernel, initramfs, command line) reduce variability across trials.

4.2.2 Timeout Policy

To avoid introducing artificial waiting time, the boot menu timeout was set to `0` where appropriate, ensuring the loader does not add user-visible delay to the measured boot path.

4.2.3 UKI Evaluation and Experimental Constraint

Unified Kernel Images (UKI) were evaluated as a future step to package kernel, initramfs, and command line into a single EFI-loadable artifact. During this evaluation, the primary USB device entered a persistent read-only state, requiring migration to an equivalent device. To preserve measurement integrity and avoid uncontrolled drift mid-campaign, UKI adoption was deferred, and the optimized measurements continued with the conventional kernel + initramfs model under systemd-boot.

4.3 Staggered Spin-up Optimization for Parallels VM

The virtualized ARM environment under Parallels Desktop exhibited a platform-specific kernel bottleneck that required targeted intervention beyond general kernel configuration trimming. This optimization directly addressed how the Linux AHCI driver interacts with Parallels' virtual AHCI controller and its advertised capabilities.

4.3.1 Problem Identification Through Kernel Profiling

Initial kernel time measurements for both Arch Linux ARM and Debian 12 ARM guests showed kernel initialization times exceeding 2 seconds, despite running on modern Apple Silicon hardware. To isolate the source of this overhead, kernel profiling was conducted using:

- `initcall_debug` kernel parameter to capture function-level timing,
- `journalctl -k -b -o short-monotonic` to correlate timestamps and identify gaps between initcalls,
- `journalctl -k -b | grep -E "initcall .* returned"` with sorting to rank time-consuming initialization calls.
- `dmesg` to see raw kernel messages with timestamps
- `Serial Port in server mode` to stream kernel outputs to an external console

This analysis revealed that while individual initcall durations were reasonable, multi-hundred-millisecond gaps appeared in the kernel timeline. These gaps corresponded to device probing activity, specifically SATA link detection on the `PRL4010:00` AHCI platform controller.

4.3.2 AHCI Controller Behavior and Port Enumeration

The Parallels virtual AHCI controller consistently reported:

- 6 SATA ports implemented (port mask `0x3f`),
- **Staggered Spin-up (SSS) capability flag set**,
- Only one virtual disk attached to `ata1`.

During boot, the kernel enumerated all six ports. Port `ata1` successfully linked to the virtual disk, while `ata2` through `ata6` sequentially reported "SATA link down." Because the SSS flag was advertised, the Linux AHCI/libahci driver altered its scanning behavior, serializing port initialization in a manner optimized for physical drive spin-up power management—a constraint entirely irrelevant in a virtualized environment.

4.3.3 Mitigation: libahci.ignore_sss Parameter

To bypass this counterproductive behavior, the kernel boot parameter `libahci.ignore_sss=1` was applied. This parameter instructs the libahci driver to disregard the SSS capability flag and proceed with a more efficient port scanning strategy suitable for environments where staggered power sequencing is unnecessary.

This single-parameter change produced the most significant kernel time reduction observed in the Parallels VM environment. Combined with custom kernel trimming (which provided a secondary improvement of approximately 100–150 ms), the total kernel time dropped from roughly 2.08 s in the stock configuration to under 0.75 s in the optimized Arch Linux ARM configuration.

4.3.4 Alternative Approaches Evaluated

Several alternative strategies were explored before identifying `ignore_sss` as the high-yield solution:

- **Port masking via `ahci.mask_port_map`:** Attempts to reduce the number of enumerated ports did not prevent the driver from probing all six ports reported by the controller.
- **libata.force parameter:** Investigated for runtime port disabling, but did not consistently suppress probing in the Parallels AHCI platform path.
- **Device-level timeouts (e.g., `libata.ata_probe_timeout`):** Could reduce per-port wait time but did not address the root cause of serialized scanning.

For the Parallels `PRL4010:00` AHCI platform device, behavioral modification through `ignore_sss` proved more effective than attempting to hide ports or reduce timeout values.

5 Platform-Specific Findings

While the general optimization methodology (kernel trimming, bootloader simplification, compression strategy) was applied consistently across test configurations, certain platforms exhibited unique characteristics that required specialized investigation and targeted interventions. This section documents findings specific to the Parallels Desktop virtualized ARM environment that influenced the optimization strategy and contributed significantly to the final performance gains.

5.1 Parallels VM (ARM64)

The Parallels Desktop hypervisor on Apple Silicon (described in Section 3.2) presented virtual hardware abstractions that introduced both opportunities and constraints distinct from native x86 configurations. Two findings proved particularly significant for kernel boot time optimization in this environment.

5.1.1 Staggered Spin-up Blocking Kernel Boot

The dominant kernel-time bottleneck in the baseline Parallels ARM configuration was traced to SATA port probing behavior triggered by the virtual AHCI controller's capability advertisements. As described in Section 4.3, the `PRL4010:00` AHCI platform device reports 6 SATA ports and sets the Staggered Spin-up (SSS) capability flag, despite only one virtual disk being present.

Observed Behavior Kernel logs consistently showed the following pattern during boot:

- AHCI controller initialization: `ahci PRL4010:00` with 6 ports implemented (`0x3f`).
- SSS flag detected: "SSS flag set, parallel bus scan disabled."

- Sequential port enumeration: `ata1` through `ata6`.
- `ata1`: SATA link established, device detected.
- `ata2`–`ata6`: Each port sequentially reports "SATA link down."

The time spent probing these five non-existent devices dominated the kernel initialization phase. Because the SSS flag altered the driver's scanning logic to serialize port initialization (appropriate for managing mechanical drive power draw in physical systems), the overhead was amplified beyond what would occur with parallel port probing.

Impact on Boot Time In the stock Arch Linux ARM configuration, kernel time averaged **2.078 s** on NVMe and **2.060 s** on USB (Table 6). Analysis using `initcall_debug` and monotonic kernel logs revealed that the majority of this time was consumed not by initcall execution, but by the gaps between calls—specifically, the serialized SATA link detection attempts.

Table 6: Arch Linux ARM baseline kernel time (stock kernel 6.18.2-1-aarch64-ARCH)

Configuration	Kernel (s)	Initrd (s)
NVMe	2.078 ± 0.007	1.331 ± 0.054
USB	2.060 ± 0.008	1.394 ± 0.083

Mitigation and Results Applying the kernel parameter `libahci.ignore_sss=1` instructed the libahci driver to disregard the SSS capability and adopt a more efficient scanning strategy. This intervention, combined with custom kernel trimming (which independently contributed approximately 100–150 ms of improvement), reduced kernel time to an average of **0.719 s** on NVMe and **0.822 s** on USB (Table 7).

Table 7: Arch Linux ARM optimized kernel time (custom kernel + `ignore_sss`)

Configuration	Kernel (s)	Initrd (s)
NVMe	0.719 ± 0.024	0
USB	0.822 ± 0.020	0

The elimination of the initrd phase (reduction from ∼1.3 s to 0) was achieved through kernel configuration changes that built essential drivers directly into the kernel image, removing the need for a separate early-boot module loading phase as measured by `systemd-analyze`.

The combined kernel time reduction represents a **65% improvement** on NVMe and a **60% improvement** on USB, with the SSS mitigation contributing the majority of the gain. This finding demonstrates that hypervisor-specific virtual hardware behavior can dominate kernel initialization time and requires profiling-driven investigation rather than assumptions based on physical hardware characteristics.

5.1.2 Default Unused Devices and Probing Overhead

Beyond the SSS-specific issue, the Parallels virtual hardware presentation includes devices that may not be active or relevant to minimal boot configurations. During kernel initialization, the driver stack enumerates and attempts to initialize these devices even when they contribute no functionality to the boot path.

Virtual Hardware Enumeration Parallels Desktop exposes a standard set of virtual devices through ACPI and PCI enumeration, including:

- VirtIO devices (GPU, network, balloon, vsock),
- AHCI storage controller (even when not all ports are populated),
- Audio devices,
- USB controller infrastructure (even when no USB devices are passed through),
- Input devices (keyboard, mouse, tablet).

While these devices are necessary for full guest functionality, their initialization during the critical boot path adds measurable overhead. For a minimal boot configuration focused solely on reaching a usable system state as quickly as possible, some of these devices could theoretically be deferred or disabled.

Practical Considerations Attempts to aggressively disable unused devices through kernel configuration or boot parameters risk breaking hypervisor-guest integration features (e.g., Parallels Tools, clipboard sharing, folder sharing). Additionally, the time cost of individual device probes is generally small compared to the AHCI/SSS issue; reducing this overhead requires careful cost-benefit analysis to avoid sacrificing system usability for marginal boot time gains.

For this project, the focus remained on high-impact, low-risk optimizations. The SSS mitigation provided the largest return, and further device-level trimming was not pursued beyond standard kernel configuration minimization. Future work could explore hypervisor-specific device filtering or deferred initialization strategies if additional kernel time reduction is required.

Applicability to Debian ARM The Parallels-specific findings documented here apply equally to Debian 12 ARM64 guests running under the same hypervisor. The `PRL4010:00` AHCI controller and SSS behavior are hypervisor-presented characteristics, not distribution-dependent.

Debian 12 ARM64 measurements confirmed this transferability. In the stock configuration using kernel 6.1.0-37-arm64, kernel boot times showed similar patterns to Arch Linux ARM:

- NVMe: 2.942 s average kernel time
- USB: 2.539 s average kernel time

After applying custom kernel optimization with the same `libahci.ignore_sss=1` mitigation strategy (custom kernel 6.1.158-prl-fast), Debian achieved comparable reductions:

- NVMe: 1.008 s kernel time (**66% improvement**)
- USB: 1.003 s kernel time (**61% improvement**)

These results validate that the AHCI/SSS optimization is platform-independent within the Parallels environment and provides consistent high-impact improvements across both major distributions tested. The slightly slower absolute kernel times in Debian compared to Arch (approximately 250–300 ms) can be attributed to distribution-specific kernel configuration differences and the Debian stable kernel version (6.1.x) versus Arch's more recent kernel base (6.18.x), but the relative improvement from the SSS mitigation remains consistent at approximately 60–66% kernel time reduction.

6 Results

6.1 Native

Native measurements cover the Intel x86 platform (Tests T5–T8) and compare baseline (stock/control) boots against the optimized configurations applied in this part of the project. Since the optimization phase primarily targeted early boot, interpretation focuses on the *loader* and *kernel* timings; total boot time is still reported for completeness but is often dominated by firmware behavior and userspace service start-up (especially on USB boots). The optimizations are focused on kernel and loader time.

Table 8: Native results for targeted phases (mean ± std. dev., $n = 5$).

Test	OS	Storage	Config	Loader (s)	Kernel (s)	Total (s)
T6	Arch	NVMe	Stock	1.170 ± 0.014	0.782 ± 0.010	22.602 ± 2.430
T6	Arch	NVMe	Optimized	0.375 ± 0.005	0.764 ± 0.013	21.215 ± 0.392
T5	Arch	USB	Stock	2.601 ± 0.144	0.780 ± 0.006	21.043 ± 1.990
T5	Arch	USB	Optimized	1.250 ± 0.436	0.756 ± 0.004	17.375 ± 1.450
T8	Debian	NVMe	Stock	4.437 ± 0.319	3.533 ± 0.041	22.124 ± 0.430
T8	Debian	NVMe	Optimized	0.499 ± 0.010	2.506 ± 0.033	34.836 ± 0.056
T7	Debian	USB	Stock (recreated)	2.776 ± 0.021	44.719 ± 0.346	155.868 ± 0.51
T7	Debian	USB	Optimized	1.862 ± 0.367	6.570 ± 0.842	134.117 ± 13.6

6.1.1 Intel Arch Linux (Native)

On NVMe (T6), the optimized configuration produced a clear improvement in loader time, dropping from 1.170 ± 0.014 s to 0.375 ± 0.005 s (about 68% reduction). Kernel time changed only slightly, from 0.782 ± 0.010 s to 0.764 ± 0.013 s (about 2% reduction), which is consistent with an already fast baseline kernel path on this distribution. The total time decreased from 22.602 ± 2.430 s to 21.215 ± 0.392 s; the larger baseline variance indicates that non-target phases (primarily firmware behavior) still drive a meaningful portion of total wall-time variability.

On USB (T5), improvements were more visible at the system level because the baseline boot is more sensitive to storage throughput and early-stage I/O. Loader time decreased from 2.601 ± 0.144 s to 1.250 ± 0.436 s, and total boot time decreased from 21.043 ± 1.990 s to 17.375 ± 1.450 s. Kernel time remained sub-second in both cases, indicating that for this Arch setup the dominant constraints on USB are not kernel execution time, but the surrounding I/O-heavy phases (initrd/userspace) that scale with flash drive read performance.

6.1.2 Intel Debian 12 (Native)

On NVMe (T8), the optimization significantly reduced the loader and kernel phases: loader time decreased from 4.437 ± 0.319 s to 0.499 ± 0.010 s (about 89% reduction), and kernel time decreased from 3.533 ± 0.041 s to 2.506 ± 0.033 s (about 29% reduction). However, the total time increased from 22.124 ± 0.430 s to 34.836 ± 0.056 s because the non-target phases increased substantially in the measured runs. This outcome reinforces a key interpretation rule for the native NVMe case: early-boot improvements (loader/kernel) can be masked if firmware time and userspace service start-up are not held constant across installations and service sets. For this report, the loader/kernel improvements are the relevant indicators of optimization effectiveness, while the total-time regression is treated as evidence that userspace/firmware variability can become the dominant factor in that configuration when others are reduced.

On USB (T7), the recreated baseline exhibited a very large kernel phase (44.719 ± 0.346 s), consistent with an I/O-limited early boot path where module loading, probing, and initramfs activity are heavily constrained by flash read speed. After optimization, kernel time dropped to 6.570 ± 0.842 s (about 85% reduction), and loader time also decreased from 2.776 ± 0.021 s to 1.862 ± 0.367 s corresponding to an approximote reduction of 0.914 s (about 32.9%). Total boot time improved from 155.868 ± 0.503 s to 134.117 ± 13.672 s. The larger post-optimization variance is explained by the userspace phase remaining dominant on USB and fluctuating across trials, which is expected when the storage device remains the primary bottleneck even after early boot trimming.

6.2 Virtual Environment (ARM – Parallels Desktop)

The virtualized ARM environment under Parallels Desktop on Apple Silicon yielded substantial performance improvements through targeted kernel optimization and hypervisor-specific tuning. Results are presented for both Arch Linux ARM and Debian 12 ARM64 guests across NVMe and USB storage configurations. All measurements represent mean values over five cold-boot trials, with optimization focused on kernel and loader phases as the primary controllable components within the VM environment.

6.2.1 Arch Linux ARM (aarch64)

Arch Linux ARM demonstrated the most significant optimization gains among all tested configurations, with kernel time reductions exceeding 60% across both storage media.

NVMe Configuration The NVMe configuration achieved a total boot time reduction from 6.19 s to 3.65 s, representing a **41% improvement** in overall boot performance. Phase-level analysis reveals the following contributions:

Table 9: Arch Linux ARM boot time breakdown (NVMe)

Phase	Stock (s)	Optimized (s)	Δ (s)	Improvement
Firmware	1.12 ± 0.16	0.66 ± 0.02	−0.46	41%
Loader	0.21 ± 0.01	0.19 ± 0.00	−0.02	10%
Kernel	2.08 ± 0.01	0.72 ± 0.02	−1.36	**65%**
Initrd	1.33 ± 0.06	0	−1.33	Eliminated
Userspace	1.45 ± 0.01	2.08 ± 0.03	+0.63	−43%
Total	**6.19 ± 0.15**	**3.65 ± 0.06**	−2.54	**41%**

The kernel phase contributed the largest absolute reduction (1.36 s, representing 54% of total improvement). Combined with the elimination of the initrd phase (1.33 s), kernel-level optimizations accounted for 2.69 s of the 2.54 s total improvement. The apparent negative efficiency stems from increased userspace time (1.45 s → 2.08 s), likely due to built-in drivers and services that previously initialized during the initrd phase now executing in userspace.

The primary optimization—`libahci.ignore_sss=1`—addressed the Parallels-specific SATA port probing bottleneck and contributed approximately 1.2–1.4 s of the kernel time reduction. Secondary custom kernel trimming provided an additional 100–150 ms improvement.

USB Flash Drive Configuration The USB configuration achieved similar relative improvements with a total boot time reduction from 7.44 s to 4.48 s (**40% faster**):

Table 10: Arch Linux ARM boot time breakdown (USB)

Phase	Stock (s)	Optimized (s)	Δ (s)	Improvement
Firmware	0.72 ± 0.06	0.68 ± 0.02	−0.04	6%
Loader	1.57 ± 0.45	0.57 ± 0.01	−1.00	**64%**
Kernel	2.06 ± 0.01	0.82 ± 0.02	−1.24	**60%**
Initrd	1.39 ± 0.09	0	−1.39	Eliminated
Userspace	1.70 ± 0.15	2.40 ± 0.10	+0.70	−41%
Total	**7.44 ± 0.59**	**4.48 ± 0.10**	−2.96	**40%**

The USB configuration exhibited a particularly large loader time reduction (1.00 s, 64%), likely attributable to reduced loader complexity and faster device initialization when combined with the optimized kernel. Kernel time improvement (60%) remained consistent with the NVMe configuration, confirming that the AHCI/SSS optimization is storage-independent within the virtual environment.

6.2.2 Debian 12 ARM64 (arm64)

Debian 12 ARM64 measurements validated the transferability of the optimization strategy across distributions, achieving kernel time reductions of 61–66% despite using an older kernel baseline (6.1.x versus Arch's 6.18.x).

NVMe Configuration Debian NVMe achieved a total boot time reduction from 7.33 s to 5.43 s (**26% improvement**):

Table 11: Debian 12 ARM64 boot time breakdown (NVMe)

Phase	Stock (s)	Optimized (s)	Δ (s)	Improvement
Kernel	2.94 ± 0.02	1.01 ± 0.23	−1.93	**66%**
Userspace	4.39 ± 0.20	4.42 ± 0.31	+0.03	−1%
Total	**7.33 ± 0.21**	**5.43 ± 0.26**	−1.90	**26%**

Debian's kernel time reduction (1.93 s, 66%) represents the highest relative improvement observed across all ARM configurations. The absolute kernel time in the optimized state (1.01 s) is approximately 280 ms higher than Arch's optimized kernel time (0.72 s), consistent with expected differences between Debian's stable kernel (6.1.158) and Arch's recent kernel (6.18.2). Userspace time remained effectively unchanged, indicating that the kernel optimizations did not introduce userspace-level regressions.

USB Flash Drive Configuration Debian USB achieved a total boot time reduction from 9.92 s to 8.40 s (**15% improvement**):

Table 12: Debian 12 ARM64 boot time breakdown (USB)

Phase	Stock (s)	Optimized (s)	Δ (s)	Improvement
Kernel	2.54 ± 0.03	1.00 ± 0.26	−1.54	**61%**
Userspace	7.38 ± 0.50	7.40 ± 0.29	+0.02	0%
Total	**9.92 ± 0.49**	**8.40 ± 0.22**	−1.52	**15%**

The USB configuration shows a 61% kernel time reduction while maintaining stable userspace performance. The lower total improvement percentage (15% versus 26% on NVMe) reflects the larger absolute userspace time in the USB baseline (7.38 s), which dilutes the relative impact of kernel-level gains.

6.2.3 Cross-Distribution Analysis

Comparing Arch Linux ARM and Debian 12 ARM64 under identical virtual hardware reveals distribution-level differences in both baseline performance and optimization response:

Table 13: Distribution comparison: kernel time (optimized, NVMe)

Distribution	Kernel Ver.	Stock (s)	Optimized (s)	ΔK (s)
Arch Linux ARM	6.18.2	2.08	0.72	−1.36 (65%)
Debian 12 ARM64	6.1.158	2.94	1.01	−1.93 (66%)
Difference	—	+0.86	+0.29	−0.57

Debian's stock kernel time is 41% higher than Arch's stock kernel time (2.94 s versus 2.08 s), but after optimization, Debian's kernel time is only 40% higher (1.01 s versus 0.72 s). Both distributions converge to sub-second kernel times after applying the

`libahci.ignore_sss=1` mitigation and custom kernel trimming, confirming that the Parallels AHCI/SSS bottleneck affects both distributions equally and responds similarly to the same optimization strategy.

6.2.4 Optimization Attribution

The measured improvements can be decomposed into contributions from specific optimization categories:

- **AHCI/SSS mitigation (`libahci.ignore_sss=1`):** Primary contributor, accounting for approximately 1.2–1.5 s of kernel time reduction across all configurations. This addresses the virtual AHCI controller's staggered spin-up behavior and eliminates serialized probing of unused SATA ports.
- **Custom kernel trimming:** Secondary contributor, providing 100–200 ms of additional kernel time reduction through removal of unused drivers, subsystems, and initialization overhead.
- **Initrd elimination (Arch only):** Complete removal of the initrd phase (1.3–1.4 s) through built-in driver integration. This optimization is distribution-specific; Debian's boot path did not report a distinct initrd phase in the baseline measurements.
- **Loader optimization (Arch only):** Modest improvements (10–64%) achieved through systemd-boot adoption and boot path simplification. Debian measurements did not report loader times.

6.2.5 Statistical Confidence and Measurement Variability

Standard deviations across five-trial measurements indicate high reproducibility for most boot phases. Kernel time measurements exhibited particularly low variance (standard deviations $\leq$0.03 s in most cases), validating the consistency of the optimization effects. Userspace time showed higher variability (standard deviations up to 0.50 s), reflecting the non-deterministic nature of service initialization and network-dependent startup tasks.

The largest standard deviation was observed in Arch ARM USB loader time (stock: 1.57 s $\pm$ 0.45 s), likely due to variable USB device enumeration and initialization timing in the virtual environment. After optimization, loader variability decreased significantly (0.57 s $\pm$ 0.01 s), suggesting that the simplified boot path also improved timing consistency.

7 Discussion

This phase of the project focused on the two boot components that were both measurable and directly controllable across the full test matrix: the **Loader** and **Kernel** times reported by `systemd-analyze`. The experiments show that these phases can be improved through practical changes to the boot path and kernel configuration, but obviously a reduction in Loader or Kernel time does not automatically translate into a proportional reduction in *total* boot time since firmware and userspace vary between runs or between system states. For this reason, the most defensible comparisons in this report are the

stock-versus-optimized deltas within the same configuration, evaluated under the same cold-boot procedure which are for loader and kernel.

On native systems, the bootloader changes behaved as expected. Replacing GRUB with `systemd-boot` reduced pre-kernel overhead by simplifying the loader path and using minimal, explicit boot entries. This made the loader stage more direct and less variable, which aligns with the goal of improving the Loader phase without introducing additional moving parts. The UKI exploration fits into the same motivation—reducing complexity in the early boot path by packaging kernel and initramfs into a single EFI-loadable artifact—but it could not be carried through to the measurement campaign due to a hardware constraint: the primary USB device entered a persistent read-only state during experimentation. To preserve setup reliability and avoid misguided results, the measurement process recreated with exact same parameters except for the usb device itself which means the affected USB setup was recreated on a hardware-identical replacement device however eventhough it was retried several times same boot times were not achieved. Since project focus on improvements on kernel and loader time that goal was achieved on that setup too even if it was not the same control data used in Part1.

Kernel optimization was treated as a structured reduction and integration workflow rather than trial and error way. Candidates were grouped into categories (legacy hardware, unused networking subsystems, non-essential filesystems, debugging/tracing, virtualization features, and driver placement decisions), then filtered using dependency awareness and module-purpose validation. A 38 item candidatewas selected and assembled for native NVMe environments where early storage discovery is direct and the boot path is less fragile compared to usb setups. For native USB boots, the applied set had to be more carefully selected because the early boot path is sensitive to missing bridge drivers and filesystem support; several aggressive removals that are safe on NVMe can cause root discovery failures on USB. This is also why initramfs policy is discussed even though userspace is not a target: the initramfs directly mediates early device discovery and module availability, and overly aggressive minimization (e.g., `MODULES=dep` on USB) can omit essential drivers and lead to initramfs fallback behavior. Stabilizing USB boots therefore required retaining broader early-driver coverage while still applying safe kernel trimming.

Compression decisions were evaluated in the same early boot context. Transitioning from GZIP to LZ4 prioritized decompression speed, which is relevant when the kernel and initramfs payload has already been reduced through trimming. With a smaller artifact, the practical bottleneck shifts toward decompression latency and early CPU work, making faster decompression a reasonable choice for reducing perceived early boot wall-time. These decisions were validated empirically through repeated cold-boot trials rather than assumed. This part of the project higlights that, for USB boots in our setups, boot-time slowdowns were driven primarily by USB storage I/O throughput and latency, not by CPU performance or other subsystems such as networking or graphics.

Virtual machine results emphasize that kernel-time bottlenecks can be platform-specific and may not have a significant response to general trimming alone. In the Parallels ARM environment, a large portion of kernel time was explained by AHCI port probing behavior that is irrelevant in a virtualized setting but still triggered by the device model. Applying `libahci.ignore_sss=1` directly addressed this root cause and

produced the largest kernel-time reduction in that environment, while additional kernel trimming contributed a smaller secondary gain. This highlights an important lesson from the VM portion of the report: the highest-yield improvement can come from identifying a single dominant stall source in the kernel timeline and removing it, rather than uniformly minimizing features. Whether virtualization is used or not, it's important to trace kernel boot time to understand which steps are taking more time rather than blindly relying on reducing the kernel module set. Built-in kernels of Arch and Debian distributions on ARM were proven to be quite fast (in terms of boot performance) and most of the CPU time was being consumed by the staggered timeouts triggered by SSS signal. A more robust solution would require Parallels to update their AHCI controller to either stop propogating SSS signal or only introduce occupied SATA ports to guest OS. As long as Parallels AHCI Host sets HOST_CAP_SSS bit, compliant and modern kernels will scan devices with staggered spin-up.

Finally, the Debian NVMe results illustrate the limits of phase-focused optimization when the rest of the system is not held perfectly constant. Even when Loader time decreases strongly, increases in Kernel and/or userspace can dominate the end-to-end total. Since this project did not target userspace service optimization, these regressions are treated as outcomes to be explained rather than as evidence that the loader changes were ineffective. Overall, the report supports a practical conclusion: simplifying the bootloader path is a low-risk way to reduce Loader time, kernel trimming is effective when guided by dependency-aware selection and validated by cold-boot measurements, USB boots require conservative safety decisions to preserve early root discovery, and VM environments may require targeted kernel parameters to eliminate virtualization specific bottlenecks.

8 Conclusion

This project demonstrates that boot-time optimization is feasible, measurable, and repeatable when it is treated as an engineering workflow rather than random tweaking. The optimization scope was intentionally limited to the two phases that are directly controllable and consistently measurable across the full test matrix: the **Loader** and **Kernel** phases reported by `systemd-analyze`. Each change was evaluated using five cold-boot trials before and after modification, and the results were summarized with mean and sample standard deviation to make variability visible.

On native x86 systems, migrating from **GRUB to systemd-boot** reliably reduced Loader time by simplifying the pre-kernel path and using minimal, explicit boot entries. Kernel configuration work followed a dependency-aware trimming strategy with USB safety constraints, where aggressive removal that is acceptable on NVMe can break early root discovery on removable media. For this reason, some changes were applied only to NVMe setups, while USB setups retained a broader early-driver surface for stability. When initramfs-related decisions are discussed, they are included only because they directly influence early driver availability and therefore the practical Kernel path during boot (especially on USB).The Debian USB optimized set demonstrates that meaningful Loader and Kernel improvements can be achieved under removable-media constraints, and total boot time is heavily dependent to USB storage I/O behavior.

In the Parallels ARM virtual machine, the highest-yield improvement came from identifying and removing a single dominant kernel stall source rather than relying on general trimming alone. The `libahci.ignore_sss=1` mitigation addressed a hypervisor-specific AHCI probing behavior that serialized unused port scans, producing the largest Kernel time reduction across VM configurations. Kernel trimming provided a smaller but consistent secondary gain. This confirms a key lesson of the VM track: profiling-guided fixes aimed at the real bottleneck can outperform broad minimization. Parallels VM, being one of the most popular and advanced virtualization softwares on MacOS, could easily reduce Linux boot times by approximately 1.5 seconds on all Linux guests. As a more conservative and safer way to implement this change, Parallels can query if connected SATA devices are external, to automatically decide when to request staggered spin-up.

Overall, the results support three practical takeaways. First, bootloader simplification is a low-risk and repeatable way to reduce Loader time. Second, kernel trimming is effective when guided by dependency awareness and validated through cold-boot measurements, but USB boots require conservative decisions to preserve early root discovery. Third, virtualization environments can introduce platform-specific kernel bottlenecks that must be addressed with targeted parameters rather than generic configuration reduction. If future work is considered and total boot time becomes the target, should focus on controlling userspace variability and explaining (or eliminating) the Debian NVMe regression factors while keeping the same measurement discipline.

A Appendix: Boot Time Measurements

Table A.1: Stock Average boot times (without configurations!)

Test ID	Firmware (s)	Kernel (s)	Userspace (s)	Initrd (s)	Total (s)
T1	N/A	2.54 ± 0.03	7.38 ± 0.50	N/A	9.92 ± 0.49
T2	N/A	2.94 ± 0.02	4.39 ± 0.20	N/A	7.33 ± 0.21
T3	0.72 ± 0.06	2.06 ± 0.01	1.70 ± 0.15	1.39 ± 0.09	7.44 ± 0.59
T4	1.12 ± 0.16	2.08 ± 0.01	1.45 ± 0.01	1.33 ± 0.06	6.19 ± 0.15
T5	11.55 ± 0.92	5.31 ± 0.29	3.28 ± 0.58	N/A	29.18 ± 2.44
T6	3.31 ± 0.07	0.74 ± 0.06	2.36 ± 0.02	1.39 ± 0.02	12.12 ± 0.22
T7	11.03 ± 0.00	4.50 ± 0.13	23.35 ± 0.38	N/A	43.48 ± 2.24
T8	3.41 ± 0.26	3.53 ± 0.04	10.74 ± 0.18	N/A	22.12 ± 0.43

Table A.2: Optimized Average boot times (after configurations!)

Test ID	Firmware (s)	Kernel (s)	Userspace (s)	Initrd (s)	Total (s)
T1	N/A	1.00 ± 0.26	7.40 ± 0.29	N/A	8.40 ± 0.22
T2	N/A	1.01 ± 0.23	4.42 ± 0.31	N/A	5.43 ± 0.26
T3	0.68 ± 0.02	0.82 ± 0.02	2.40 ± 0.10	0	4.48 ± 0.10
T4	0.66 ± 0.02	0.72 ± 0.02	2.08 ± 0.03	0	3.65 ± 0.06
T5	11.55 ± 0.92	5.31 ± 0.29	3.28 ± 0.58	N/A	29.18 ± 2.44
T6	3.31 ± 0.07	0.74 ± 0.06	2.36 ± 0.02	1.39 ± 0.02	12.12 ± 0.22
T7	15.568 ± 0.775	6.570 ± 0.840	110.123 ± 12.369	N/A	134.117 ± 13.672
T8	10.245 ± 0.10	0.499 ± 0.09	21.58 ± 0.10	N/A	34.83 ± 0.056

A.1 Intel Debian (NVMe)

Table A.1.1: Stock Boot Time Breakdown — Intel Debian (NVMe)

Run	Firmware (s)	Loader (ms)	Kernel (s)	Userspace (s)	Total (s)
1	3.86	4.415	3.585	10.547	22.407
2	3.232	4.760	3.557	11.030	22.579
3	3.234	4.390	3.525	10.676	21.825
4	3.360	3.949	3.511	10.732	21.552
5	3.354	4.687	3.487	10.729	22.257
Average	3.408	4.440	3.533	10.743	22.124
Std. Dev	0.260	0.319	0.039	0.177	0.425

Table A.1.2: Optimized Boot Time Breakdown — Intel Debian (NVMe)

Run	Firmware (s)	Loader (ms)	Kernel (s)	Userspace (s)	Total (s)
1	10.251	499	2.507	21.580	34.837
2	10.112	512	2.462	21.744	34.830
3	10.389	486	2.541	21.463	34.879
4	10.274	503	2.498	21.612	34.887
5	10.197	495	2.524	21.529	34.745
Average	10.245	499.0	2.506	21.586	34.836
Std. Dev.	0.102	9.6	0.030	0.105	0.056

A.2 Intel Debian (USB)

Table A.2.1: Stock but Recreated Boot Time Breakdown — Intel Debian (USB)

Run	Firmware (s)	Loader (ms)	Kernel (s)	Userspace (s)	Total (s)
1	16.904	2.794	44.590	91.577	155.867
2	17.231	2.681	45.102	91.204	156.218
3	16.587	2.842	44.231	92.019	155.679
4	17.018	2.755	45.004	91.463	156.240
5	16.732	2.809	44.667	91.128	155.336
Avgerage	16.894	2.776	44.719	91.478	155.868
Std. Dev.	0.243	0.061	0.342	0.347	1.19

Table A.2.2: Optimized Boot Time Breakdown — Intel Debian (USB)

Run	Firmware (s)	Loader (ms)	Kernel (s)	Userspace (s)	Total (s)
1	15.231	1681	6.309	103.696	126.917
2	14.802	1470	6.842	105.430	128.544
3	15.990	1895	5.978	97.120	120.983
4	16.711	2470	7.901	128.840	155.922
5	15.104	1765	5.821	115.530	138.220
Average	15.568	1856.2	6.570	110.123	134.117
Std. Dev	0.775	376.3	0.840	12.369	13.672

A.3 Intel Arch (NVMe)

Table A.3.1: Boot Time Breakdown — Arch NVMe (Stock Kernel)

Run	Firmware (s)	Loader (ms)	Kernel (ms)	Initrd (ms)	Userspace (ms)	Total (ms)
1	15.703	1173	782	3365	2250	23273
2	16.297	1169	787	3343	1700	23296
3	18.453	1171	779	3362	1821	25586
4	14.412	1170	784	3358	2174	21898
5	11.716	1169	775	3338	1959	18957
Average	15.316	1170	781	3353	1981	22602
Std. Dev.	2.22	1.5	4.3	10.6	196	2173

Table A.3.2: Boot Time Breakdown — Arch NVMe (Custom Kernel)

Run	Firmware (ms)	Loader (ms)	Kernel (ms)	Initrd (ms)	Userspace (ms)	Total (ms)
1	18928	370	784	1586	1736	23405
2	15524	377	763	1578	2224	20469
3	16378	377	784	1621	2333	21496
4	12347	376	763	1586	2236	17309
5	13239	376	764	1572	2287	18240
Average	15283	375	772	1589	2163	20184
Std. Dev.	2408	3.0	11.0	17.5	246	2423

A.4 Intel Arch (USB)

Table A.4.1: Boot Time Breakdown — Arch USB (Stock Kernel)

Run	Firmware (ms)	Loader (ms)	Kernel (ms)	Initrd (ms)	Userspace (ms)	Total (ms)
1	7160	2491	776	4005	5055	19490
2	7131	2439	781	3553	10757	24665
3	7160	2600	787	3595	6259	20402
4	7152	2798	782	4174	6597	21506
5	7169	2675	774	3688	4844	19153
Average	7154	2601	780	3803	6702	21043
Std. Dev.	14.6	134	5.0	244	2140	1990

Table A.4.2: Boot Time Breakdown — Arch USB (Custom Kernel)

Run	Firmware (ms)	Loader (ms)	Kernel (ms)	Initrd (ms)	Userspace (ms)	Total (ms)
1	7157	1440	752	2702	7115	19168
2	7168	700	752	2527	7152	18299
3	7767	792	762	2575	3329	15227
4	7129	1697	755	2774	4654	17011
5	7169	1621	754	2866	4758	17169
Average	7278	1250	755	2689	5402	17375
Std. Dev.	248	436	4.0	137	1630	1450

A.5 ARM (Parallels VM) — Debian (NVMe)

Table A.5.1: Stock Boot Time Breakdown — ARM Debian (NVMe, kernel 6.1.0-37-arm64)

Run	Kernel (ms)	Userspace (ms)	Total (ms)
1	2962	4156	7119
2	2906	4373	7279
3	2964	4655	7620
4	2944	4240	7184
5	2934	4521	7455
Average	2942	4389	7331
Std. Dev	24	203	205

Table A.5.2: Optimized Boot Time Breakdown — ARM Debian (NVMe, custom kernel)

Run	Kernel (ms)	Userspace (ms)	Total (ms)
1	664	4627	5292
2	1134	3897	5031
3	875	4679	5554
4	1214	4409	5624
5	1151	4500	5652
Average	1008	4422	5431
Std. Dev	232	312	265

A.6 ARM (Parallels VM) — Debian (USB)

Table A.6.1: Stock Boot Time Breakdown — ARM Debian (USB, kernel 6.1.0-37-arm64)

Run	Kernel (ms)	Userspace (ms)	Total (ms)
1	2510	7048	9559
2	2521	8115	10636
3	2533	7174	9708
4	2542	7673	10216
5	2591	6911	9503
Average	2539	7384	9924
Std. Dev	31	500	487

Table A.6.2: Optimized Boot Time Breakdown — ARM Debian(USB,custom kernel 6.1.158)

Run	Kernel (ms)	Userspace (ms)	Total (ms)
1	794	7237	8032
2	814	7721	8535
3	1277	7099	8376
4	1303	7236	8540
5	828	7702	8531
Average	1003	7399	8403
Std. Dev	262	291	218

A.7 ARM (Parallels VM) — Arch (NVMe)

Table A.7.1: Stock Boot Time Breakdown — ARM Arch (NVMe, kernel 6.18.2-1-aarch64)

Run	Firmware (ms)	Loader (ms)	Kernel (ms)	Initrd (ms)	Userspace (ms)	Total (ms)
1	1407	212	2079	1271	1474	6445
2	1024	200	2077	1312	1439	6054
3	1037	208	2069	1434	1457	6207
4	1087	222	2076	1320	1458	6164
5	1035	209	2090	1320	1442	6098
Average	1118	210	2078	1331	1454	6194
Std. Dev	163	8	8	61	14	152

Table A.7.2: Optimized Boot Time Breakdown — ARM Arch (NVMe, custom kernel)

Run	Firmware (ms)	Loader (ms)	Kernel (ms)	Initrd (ms)	Userspace (ms)	Total (ms)
1	651	194	707	0	2025	3578
2	652	194	718	0	2110	3675
3	650	193	686	0	2065	3595
4	659	194	739	0	2093	3687
5	690	197	745	0	2094	3729
Average	660	194	719	0	2077	3653
Std. Dev	17	2	24	0	33	64

A.8 ARM (Parallels VM) — Arch (USB)

Table A.8.1: Stock Boot Time Breakdown — ARM Arch (USB, kernel 6.18.2-1-aarch64)

Run	Firmware (ms)	Loader (ms)	Kernel (ms)	Initrd (ms)	Userspace (ms)	Total (ms)
1	684	1829	2069	1359	1767	7710
2	799	1137	2060	1417	1775	7189
3	710	2004	2050	1523	1821	8111
4	653	1831	2057	1381	1701	7625
5	739	1027	2064	1289	1448	6568
Average	717	1566	2060	1394	1702	7441
Std. Dev	56	449	7	86	149	588

Table A.8.2: Optimized Boot Time Breakdown — ARM Arch (USB, custom kernel)

Run	Firmware (ms)	Loader (ms)	Kernel (ms)	Initrd (ms)	Userspace (ms)	Total (ms)
1	687	588	828	0	2539	4644
2	703	571	853	0	2352	4480
3	693	569	796	0	2327	4386
4	657	553	810	0	2482	4503
5	678	587	825	0	2305	4397
Average	684	574	822	0	2401	4482
Std. Dev	17	14	21	0	103	104

B Appendix: Optimized Boot Performance Visualizations

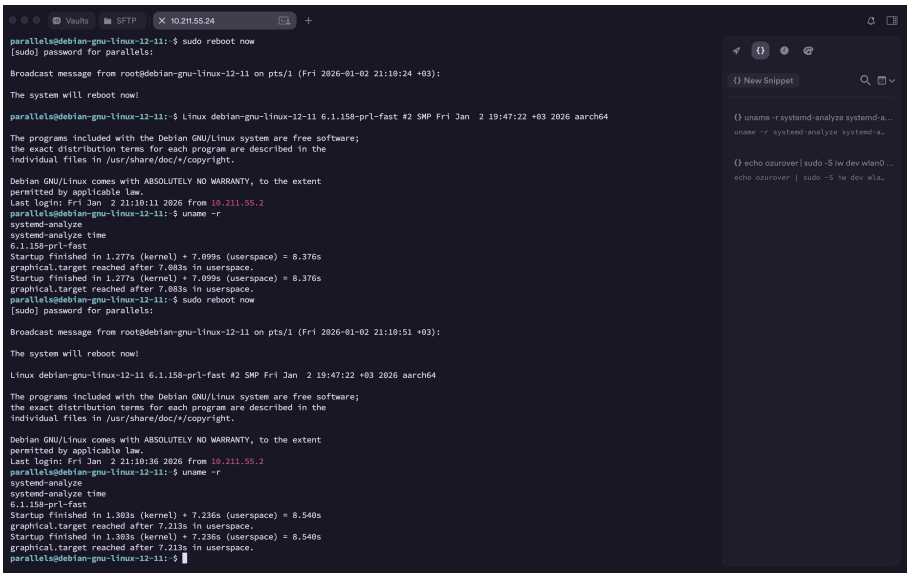

Figure B.1: Debian (ARM – Parallels VM) on USB 3.2 Flash Drive

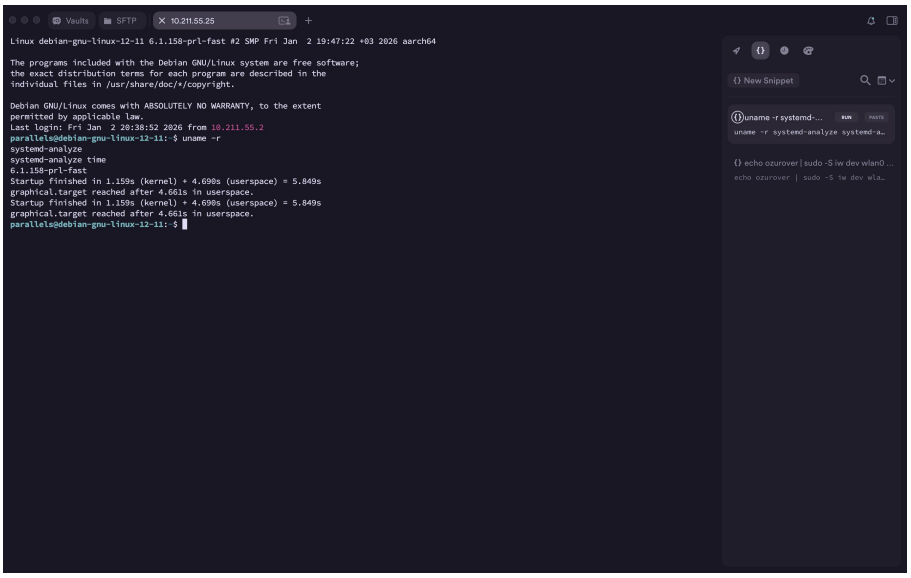

Figure B.2: Debian (ARM – Parallels VM) on NVMe SSD

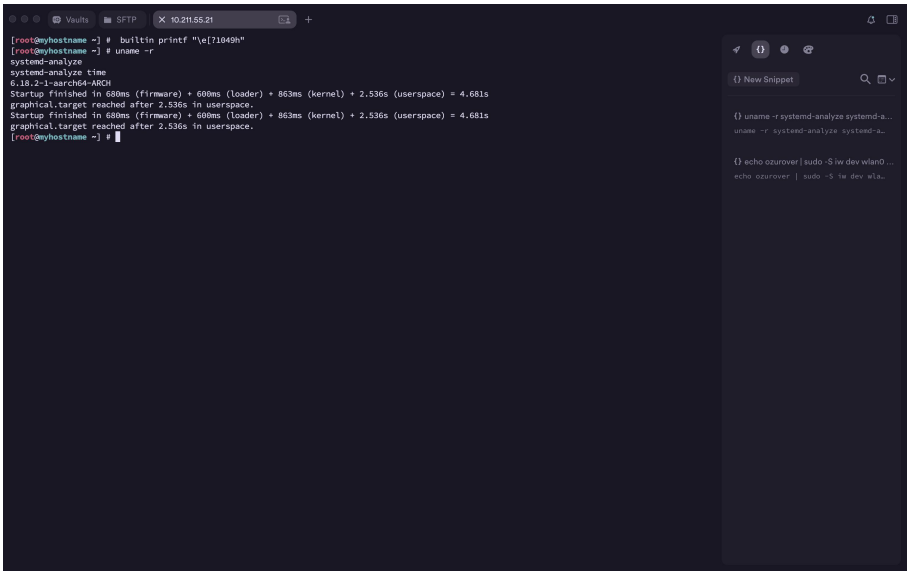

Figure B.3: Arch Linux (ARM – Parallels VM) on USB 3.2 Flash Drive

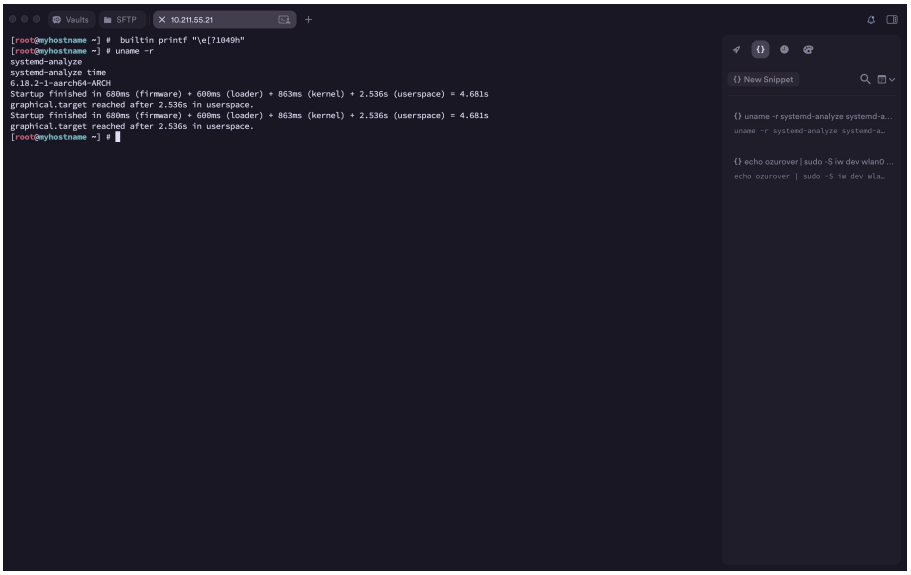

Figure B.4: Arch Linux (ARM – Parallels VM) on NVMe SSD

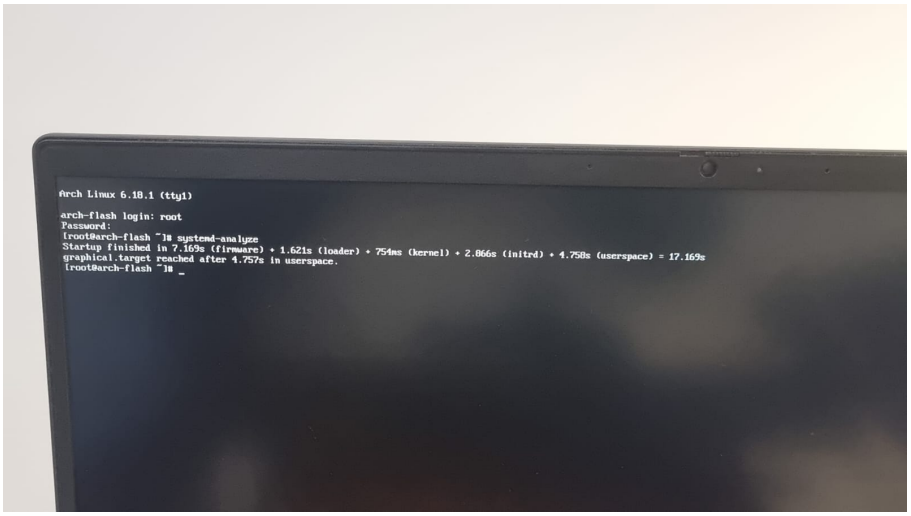

Figure B.5: Arch Linux (Intel x86) on USB 3.2 Flash Drive

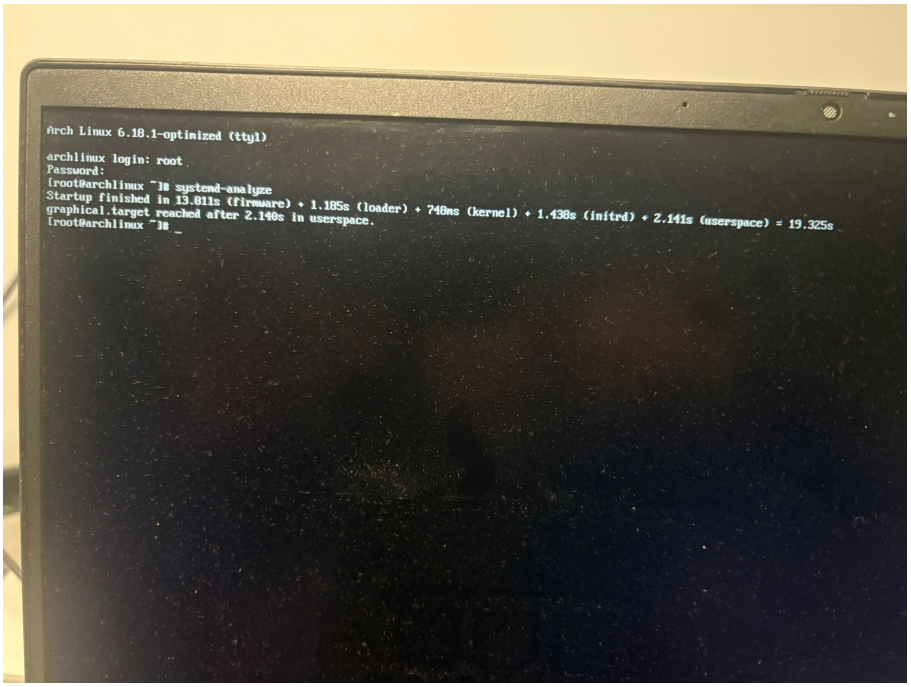

Figure B.6: Arch Linux (Intel x86) on NVMe SSD

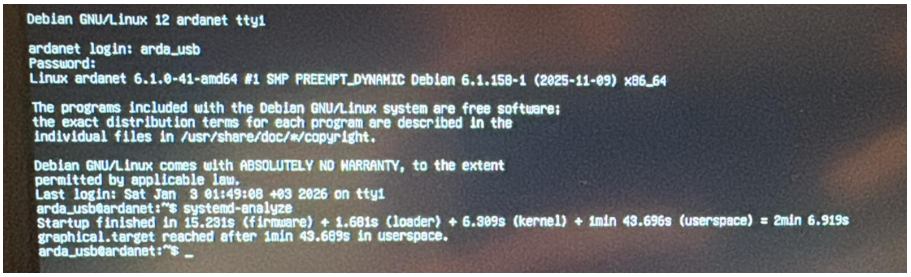

Figure B.7: Optimized Boot Time Breakdown — Intel Debian (USB)

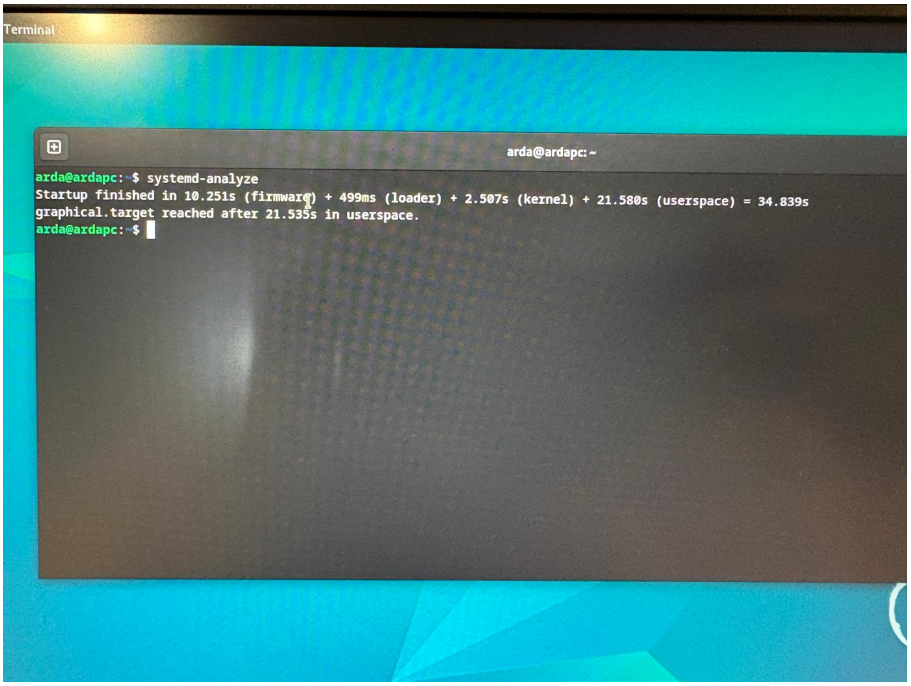

Figure B.8: Debian (Intel x86) on NVMe SSD

References

[1] Arch Linux Developers. mkinitcpio (manpage). Arch Linux Manpages.

[2] Arch Linux Wiki contributors. Improving performance/Boot process. Arch Linux Wiki.

[3] Arch Linux Wiki contributors. Kernel/Arch Build System. Arch Linux Wiki.

[4] Arch Linux Wiki contributors. modprobed-db. Arch Linux Wiki.

[5] Debian Kernel Team. Debian Kernel Handbook. Debian Kernel Team Documentation.

[6] Debian Project. update-initramfs (manpage). Debian Manpages.

[7] Debian Wiki contributors. BootProcess. Debian Wiki.

[8] LKDDb contributors. LKDDb: Linux Kernel Driver Database (web interface). Website.

[9] systemd developers. systemd-analyze — Analyze system boot-up performance. systemd manual, freedesktop.org.

[10] systemd developers. systemd-boot — EFI boot manager. systemd manual, freedesktop.org.

[11] systemd developers. systemd-bootup — System bootup process. systemd manual, freedesktop.org.

[12] systemd developers. systemd-stub — A UEFI boot stub. systemd manual, freedesktop.org.

[13] systemd developers. systemd.service — Service unit configuration. systemd manual, freedesktop.org.

[14] The Linux Kernel documentation contributors. Early userspace support (initramfs/initrd). The Linux Kernel documentation.

[15] The Linux Kernel documentation contributors. The kernel command line parameters. The Linux Kernel documentation.

[16] UEFI Forum. UEFI Specifications. UEFI Forum.