

Methodology For Testing The Linux Process Scheduler

Bc. Aleš Tabarka

Master's thesis
2025



Tomas Bata University in Zlín
Faculty of Applied Informatics

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Aleš Tabarka
Osobní číslo: A23600
Studijní program: N0613A140022 Informační technologie
Specializace: Softwarové inženýrství
Forma studia: Kombinovaná
Téma práce: Metodika testování plánovače procesů v Linuxu
Téma práce anglicky: Methodology for Testing the Linux Process Scheduler

Zásady pro vypracování

- Vypracujte literární řešení na dané téma.
- Vysvětlete význam a funkci plánování procesů, cíle a kritéria plánovacích algoritmů.
- Popište princip používaných plánovačů procesů a uveďte vliv architektury CPU na plánovač procesů.
- Navrhněte metodiku testování plánovače procesů a způsob prezentace výsledků pro různé scénáře použití.
- Proveďte srovnání a testy plánovačů podle navržené metodiky.
- Zhodnoťte a konfrontujte výsledky.

Forma zpracování diplomové práce: **tištěná/elektronická**
Jazyk zpracování: **Angličtina**

Seznam doporučené literatury:

1. BHARADWAJ, Raghu. *Mastering Linux kernel development: a kernel developer's reference manual*. Birmingham: Packt, 2017. ISBN 978-1-78588-305-7.
2. LOVE, Robert. *Linux kernel development*. 3rd ed. Upper Saddle River: Addison-Wesley, c2010. ISBN 978-0-672-32946-3.
3. GREGG, Brendan. *Systems performance: enterprise and the cloud*. Second edition. Addison-Wesley professional computing series. Boston: Addison-Wesley, 2021. ISBN 978-0-13-682015-4.
4. THE KERNEL DEVELOPMENT COMMUNITY. *The Linux Kernel documentation*. Online. 2024. Dostupné z: <https://docs.kernel.org/scheduler/index.html>. [cit. 2024-10-22].
5. STOICA, Ion a HUSSEIN, Abdel-Wahab. *Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*. Online. CS Dpt., Old Dominion Univ. TR-95-22. 1996. Dostupné z: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=805acf7726282721504c8f00575d91ebfd750564>. [cit. 2024-10-22].
6. EKLEKTIX, Inc. *Kernel index: Scheduler*. Online. LWN.net. 2024. Dostupné z: <https://lwn.net/Kernel/Index/#Scheduler>. [cit. 2024-10-22].
7. RED HAT, Inc. *Performance Tuning Guide*. Online. Red Hat Documentation. 2024. Dostupné z: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/performance_tuning_guide. [cit. 2024-10-22].
8. VOŽÁR, Jirí. *Performance Testing of Linux Kernel Scheduler*. Online, Bakalářská práce. Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2019. Dostupné z: <https://www.fit.vut.cz/study/thesis-file/21469/21469.pdf>. [cit. 2024-10-23].

Vedoucí diplomové práce: **doc. Ing. Martin Sysel, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan



prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Tato práce se zaměřuje na návrh a realizaci metodiky pro testování plánovače procesů v operačním systému Linux. Pomocí opakovatelné testovací metodologie, specifických testů a unifikovaného prostředí jsou zkoumány vlastnosti plánovačů CFS a EEVDF. Zaměření testů pokrývá oblasti latence, propustnosti, spravedlivosti, škálovatelnosti, chování při přepínání kontextu a detekce hladovění procesu. Testování probíhá na vybraných platformách s cílem porozumět dopadům plánování na moderní architektury.

Klíčová slova: Plánovač, Linux, Testování, Jádro, CFS, EEVDF, Procesy, mkosi, Výkon, Latence, Prostupnost, Spravedlivost, Škálovatelnost, Hladovění, Přepínání kontextu, SMT, Hybridní architektura, Plánování procesů, Vícejádrové systémy, Testování plánovače

ABSTRACT

This thesis focuses on the design and implementation of a methodology for testing the Linux process scheduler. Using a reproducible methods, unified environment and defined test scenarios, it examines the behavior of CFS and EEVDF schedulers. The evaluation covers latency, throughput, fairness, scalability, context switching, and starvation detection across selected multi-core platforms, aiming to understand scheduler behavior in modern CPU architectures.

Keywords: Scheduler, Linux, Testing, Kernel, CFS, EEVDF, Process, mkosi, Performance, Latency, Throughput, Fairness, Scalability, Starvation, Context Switching, Benchmarking, Hybrid Architecture, SMT, Process Scheduling, Multicore, Scheduler Evaluation

I would like to express my sincere gratitude to my thesis supervisor, doc. Ing. Martin Sysel, Ph.D., for his guidance, feedback, and support throughout the development of this thesis.

I also thank my girlfriend Ivana for her unwavering support and encouragement during my studies. My appreciation extends to my friend Pavel, whose advice and help with the mkosi tool were invaluable during the setup of the testing environment.

I declare that the submitted printed version of the thesis and the electronic version uploaded to IS/STAG are identical.

Complying with rules for using modern technology in creative work, assistive software with elements of artificial intelligence was occasionally used during the preparation of this thesis. Specifically, the ChatGPT generative model, available at <https://chatgpt.com/>, and GitHub Copilot integrated into the Visual Studio Code development environment were utilized to support text revision, code suggestions and structural refinement.

All generated content was carefully reviewed, edited and validated by the author, who takes full responsibility for the final form, originality and accuracy of this thesis.

TABLE OF CONTENTS

THE INTRODUCTION	13
I THEORETICAL PART	14
1 TESTING OF THE LINUX PROCESS SCHEDULER.....	15
1.1 THEORETICAL BACKGROUND AND SCHEDULER ARCHITECTURE	15
1.2 PERFORMANCE TESTING METHODOLOGIES	15
1.3 BENCHMARKING TOOLS AND VISUALIZATION TECHNIQUES	16
1.4 SCHEDULING THEORY AND COMPLEXITY CONSIDERATIONS	16
1.5 INTEGRATION OF THEORY AND PRACTICE	17
2 FUNDAMENTS & OBJECTIVES OF PROCESS SCHEDULING	18
2.1 FUNCTION OF PROCESS SCHEDULING	18
2.2 OBJECTIVES OF SCHEDULING ALGORITHMS	18
2.3 EVALUATION CRITERIA FOR SCHEDULING ALGORITHMS	20
2.4 TRADE-OFFS AND SCHEDULER DESIGN	22
3 LINUX PROCESS SCHEDULERS.....	23
3.1 O(1) SCHEDULER.....	23
3.2 COMPLETELY FAIR SCHEDULER (CFS).....	24
3.3 EEVDF SCHEDULER	25
3.4 PROCESSOR AFFINITY AND SCHEDULING.....	26
3.5 CPU ARCHITECTURE IMPACT ON SCHEDULING	27
4 PRESENT LINUX SCHEDULERS AND EVALUATIONS	29
4.1 RATIONALE FOR KERNEL SELECTION.....	29
4.2 PERFORMANCE INSIGHTS FROM EARLY EVALUATION	29
4.3 HYPOTHESES FOR THE RESULTS EVALUATION	30
II PRACTICAL PART.....	32
5 PROCESS SCHEDULER TESTING METHODOLOGY	33
5.1 LINUX DISTRIBUTION SELECTION AND TEST ENVIRONMENT	33

5.2	SCHEDULER LATENCY TEST	34
5.3	SCHEDULER FAIRNESS LATENCY TEST	37
5.4	SCHEDULER THROUGHPUT TEST	40
5.5	SCHEDULER CONTEXT SWITCHING TEST	43
5.6	SCHEDULER SCALABILITY TEST	45
5.7	SCHEDULER STARVATION DETECTION TEST	48
5.8	SYSTEM ENVIRONMENT INFORMATION	50
5.9	SYSTEM INITIALIZATION	52
5.10	TEST PROCEDURE EXECUTION	53
5.11	CPU TEMPERATURE AND FREQUENCY LOGGER.....	56
6	TEST ENVIRONMENT SETUP AND DEPLOYMENT	58
6.1	INSTALLATION OF MKOSI.....	58
6.2	CONFIGURE SYSTEM IMAGE.....	58
6.3	BUILD AND DEPLOYMENT OF THE SYSTEM IMAGE.....	64
7	TESTING ON DIFFERENT HARDWARE PLATFORMS	65
7.1	INTEL HYBRID ARCHITECTURE (i7-1260P)	65
7.2	AMD ZEN+ (RYZEN 5 2600)	87
7.3	AMD ZEN3 (RYZEN 9 5950X)	107
8	TEST RESULTS EVALUATION.....	126
8.1	TEST ENVIRONMENT CONFIGURATION	126
8.2	LATENCY	126
8.3	FAIRNESS.....	129
8.4	THROUGHPUT	131
8.5	CONTEXT SWITCHES	133
8.6	SCALABILITY	135
8.7	STARVATION DETECTION.....	137
	CONCLUSION.....	141
	REFERENCES	143

LIST OF FIGURES	148
LIST OF TABLES	151
LIST OF LISTINGS	154
LIST OF ABBREVIATIONS	155
LIST OF APPENDICES	157

THE INTRODUCTION

Efficient process scheduling is a cornerstone of operating system performance. As hardware evolves, particularly with the rise of heterogeneous architectures combining performance and efficiency cores, the ability of the scheduler to balance competing demands becomes crucial.

The Linux kernel scheduler is central to managing task execution, and its development reflects the rising complexity of both general purpose and response time sensitive workloads. Assessment of the process scheduler performance requires a reliable, consistent, and well structured testing methodology. This thesis addresses that need by building a test framework designed to evaluate key process scheduling properties under varied conditions.

The initial chapters provide a foundation in scheduling theory, covering classical goals such as latency minimization, throughput maximization, and fairness. This is followed by an overview of major Linux scheduler designs, including the legacy $O(1)$ scheduler, the well established Completely Fair Scheduler, and the recent EEVDF algorithm. Their behavior is analyzed in the context of modern system features such as SMT, cache topology, and asymmetric core layouts.

A reproducible benchmarking methodology is then introduced, leveraging both established tools and custom scripting. This enables thorough evaluation of latency, fairness, throughput, context switching, and starvation. Using lightweight system images helps keep kernel and scheduler behavior consistent across platforms.

The framework is applied to three processor platforms: an Intel hybrid-core design and two AMD Ryzen architectures. Results reveal how scheduling strategies perform under different hardware constraints and workloads, uncovering strengths and weaknesses.

In conclusion, this thesis bridges theoretical scheduling concepts with practical, reproducible evaluation. It contributes a lightweight and portable testing methodology for examining Linux scheduler behavior across modern hardware.

I THEORETICAL PART

1 TESTING OF THE LINUX PROCESS SCHEDULER

The Linux kernel scheduler is a key component influencing operating system performance, particularly in terms of responsiveness, throughput and latency. To ensure the reliability and efficiency of evolving scheduler implementations, rigorous testing methodologies are essential.

1.1 Theoretical Background and Scheduler Architecture

The foundation for understanding the Linux process scheduler begins with the book *Linux Kernel Development* by Robert Love, which provides a comprehensive overview of kernel structures, particularly the design and implementation of the Completely Fair Scheduler. Adding to this, Raghu Bharadwaj's *Mastering Linux Kernel Development* offers a developer perspective on hacking and debugging the Linux kernel, with insights into modifying and analyzing the scheduler's behavior [1, 2].

In his book *Systems Performance: Enterprise and the Cloud*, Brendan Gregg introduces essential methodologies for performance analysis, including the USE and RED methods. These frameworks serve as the foundation for analyzing system bottlenecks and behavior under load, making them especially valuable for testing kernel components like the scheduler [3].

The online documentation maintained by the Linux kernel community provides the most authoritative and up-to-date reference on process scheduler implementation and design. To complement this, Brendan Gregg's *Systems Performance* offers to get practical insights into historical changes and current trends in Linux scheduling, including kernel tracing, observability tools, and scheduler-specific performance implications [3, 4].

1.2 Performance Testing Methodologies

Jiří Vozár's bachelor's thesis *Performance Testing of Linux Kernel Scheduler*, conducted at Brno University of Technology, is one of the most applied academic studies in this area.

Vozár proposes a methodology involving automated long-term testing, statistical analysis, and visualization, even exploring the use of machine learning for regression detection [5].

The *Red Hat Performance Tuning Guide* complements Vozár's work with enterprise-grade strategies for workload tuning, system monitoring, and benchmark execution in production environments [6].

The algorithmic principles of the Earliest Eligible Virtual Deadline First (EEVDF) scheduler are rooted in the 1996 paper by Ion Stoica and Abdel-Wahab Hussein, titled *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*. This work laid the foundation for EEVDF's integration into Linux 6.6 as a successor to CFS [7, 8].

1.3 Benchmarking Tools and Visualization Techniques

Gregg's second edition of *Systems Performance* provides extensive coverage of benchmark tools like perf, ftrace, bpftrace, and KernelShark, which are indispensable for the analysis of scheduler behavior. These tools allow fine-grained tracking of CPU usage, latency, and scheduling events [3].

Vozár's thesis emphasizes the importance of data visualization in long-term testing. Tools like Traceshark, developed by Viktor Rosendahl, facilitate the graphical interpretation of scheduling metrics over time, aiding in the identification of regressions and performance anomalies [5, 9].

1.4 Scheduling Theory and Complexity Considerations

From a theoretical standpoint, Peter Brucker's book *Scheduling Algorithms* presents a broad classification of scheduling problems, offering insights into their computational complexity and applicable optimization methods [10].

Michael Pinedo's *Scheduling: Theory, Algorithms, and Systems* serves as a cornerstone text in operations research, connecting deterministic and stochastic models to real-world

scheduling applications, including those found in operating systems [11].

The *Handbook of Scheduling* by Joseph Y-T. Leung and James H. Anderson further expand on models and performance criteria, particularly in multi-processor and real-time systems [12].

Frank Werner, Larysa Burtseva, and Yuri Sotskov contribute additional depth through two edited volumes: *Algorithms for Scheduling Problems* and *Exact and Heuristic Scheduling Algorithms*. These collections examine heuristic and exact techniques applicable to high complexity scheduling, which can be extended to CPU level task scheduling [13].

1.5 Integration of Theory and Practice

The connection between theory and practice is best seen in the combination of Gregg's instrumentation based methodology and the formal models presented by Brucker, Pinedo, and Werner. These sources together provide a solid theoretical basis for the practical testing of Linux schedulers.

The influence of CPU microarchitecture on scheduler efficiency is explored in John Paul Shen and Mikko Lipasti's book *Modern Processor Design: Fundamentals of Superscalar Processors*. The work highlights how instruction level parallelism, core configuration, and pipeline depth affect the kernel's ability to schedule and switch tasks effectively [14].

2 FUNDAMENTALS & OBJECTIVES OF PROCESS SCHEDULING

Process scheduling is a fundamental function of operating systems that enables the efficient, fair, and responsive execution of processes in a shared computing environment. The process scheduler determines the order and timing in which processes access the CPU, which is crucial in achieving performance goals, user experience, and system stability [11, 3].

2.1 Function of Process Scheduling

The primary purpose of process scheduling is to control which process is executed on a CPU core at a given time. In systems with more processes than cores, the scheduler ensures that all runnable processes receive access to CPU resources according to system policies and goals.

Schedulers operate in the kernel space and manage one or more queues of runnable tasks. Decisions are based on task priority, CPU affinity, I/O state, and historical runtime statistics. They must support diverse workload types: interactive applications, computation intensive tasks, background daemons, and real-time jobs [3, 12].

The architecture of the processor heavily influences scheduler behavior. For instance, in superscalar and out-of-order execution architectures, task switching can be costly due to pipeline flushing and cache invalidation, affecting context-switch efficiency. Moreover, with hybrid architectures like Intel's performance and efficiency cores, the scheduler must decide not just when to run a task, but also where to map tasks to the most suitable core type [15, 14].

2.2 Objectives of Scheduling Algorithms

Scheduling algorithms aim to fulfill different system goals. These often conflict with one another, and the choice of scheduling strategy depends on workload characteristics and system requirements [11, 10].

2.2.1 Maximizing Throughput

Throughput is the number of processes completed per unit time. High throughput implies better CPU utilization and is critical in server and high-performance computing workloads. However, architectures with shared caches or inter-core communication overhead (NUMA) can negatively affect throughput if the scheduler does not optimize for locality [14, 11].

2.2.2 Minimizing Latency

Latency refers to the time delay between a process becoming ready and starting execution. Interactive systems, such as desktop environments, prioritize latency to ensure responsive behavior. Algorithms like SRTF and variants of the CFS try to minimize latency for short tasks. CPU architectures with deep pipelines or high cache latencies may amplify perceived delays [3].

2.2.3 Fairness

Fairness ensures that all processes get fair CPU access, avoiding starvation and ensuring consistent behavior across users. Proportional share and lottery scheduling strategies are designed to increase fairness. On multi-core systems, fairness might also involve core usage balancing to prevent thermal throttling and unequal performance [3, 7, 12].

2.2.4 Energy Efficiency

Schedulers may aim to reduce CPU activity by consolidating tasks on fewer cores, allowing others to enter power-saving states. This is essential in mobile and embedded systems and has gained importance with hybrid architectures [15].

2.2.5 Scalability

Scalability involves maintaining scheduling efficiency as the number of CPUs increases. SMP and NUMA-aware process schedulers must consider inter-core communication, cache

coherence, and memory locality [14, 3].

2.2.6 Predictability

In real-time systems, schedulers must meet deterministic constraints. Objectives include bounded response times, minimal jitter, and deadline guarantees. Algorithms like EDF or RMS are often used [12, 3].

2.3 Evaluation Criteria for Scheduling Algorithms

Scheduling algorithms are evaluated using both theoretical and practical metrics. These criteria provide quantifiable measures to compare and assess scheduler performance under various workloads.

- **Turnaround Time:** The total time taken from process submission to completion. Shorter turnaround times are preferred in batch systems.
- **Waiting Time:** The time a process spends in the ready queue. Algorithms like FCFS tend to produce high waiting times for long jobs.
- **Response Time:** Time between submission and first execution. Essential in interactive environments.
- **CPU Utilization:** Percentage of time the CPU is actively executing tasks. High utilization suggests efficient scheduling, though too high may reduce responsiveness.
- **Context Switch Rate:** Frequent context switches degrade performance due to cache and TLB invalidation. Minimizing this rate is efficient beneficial unless required for responsiveness [3].
- **Starvation Avoidance:** Algorithms must ensure that no process is indefinitely denied access to the CPU.
- **Jitter:** Variability in scheduling latency, critical in multimedia and real-time systems.

- **Deadline Miss Rate:** Relevant in real-time workloads, the lower the rate, the more reliable the scheduler [12].
- **Load Balancing Efficiency:** Measures how evenly work is distributed across cores. Poor balancing leads to idle CPUs while others are overloaded [3].

The Linux kernel exposes many tuning options under `/proc/sys/kernel/sched_*` which influence these criteria. Key parameters include:

- `sched_latency_ns`: Target latency for a scheduling period, lower value improve responsiveness at the cost of throughput.
- `sched_min_granularity_ns`: Minimum time slice length, higher values leads in reduction of context switch frequency.
- `sched_wakeup_granularity_ns`: Controls preemption sensitivity, directly affects interactivity.
- `sched_migration_cost_ns`: Time threshold for considering task migration, higher values reduce inter-core hopping.
- `sched_nr_migrate`: Limits number of tasks migrated between CPUs.
- `sched_autogroup_enabled`: Enables or disables automatic grouping of tasks per terminal session, impacting fairness in multi-user systems.
- `sched_util_avg_enabled`: Enables average utilization estimation for energy-aware scheduling.
- `sched_energy_aware`: Activates or deactivates scheduling decisions based on energy models, crucial for heterogeneous cores.

These parameters allow system administrators to adapt scheduling behavior to match specific workload requirements [3, 6].

These criteria are not mutually exclusive. For instance, minimizing waiting time can increase context switching overhead. Therefore, system designers must carefully evaluate and choose the right optimization trade-offs according to the target environment deployment [10].

2.4 Trade-offs and Scheduler Design

Scheduler design involves analyzing complex trade-offs between conflicting performance goals. For example:

- Increasing fairness can reduce responsiveness.
- Lowering latency might increase CPU overhead due to more frequent task switching.
- Improving predictability often comes at the expense of throughput.

These trade-offs are influenced by workload characteristics and system architecture. Modern schedulers, especially those targeting hybrid CPUs and energy-efficient platforms, must balance performance, fairness, and power consumption. Kernel-level tunables described earlier provide mechanisms to shift these trade-offs dynamically at runtime [3].

3 LINUX PROCESS SCHEDULERS

3.1 O(1) Scheduler

3.1.1 Principle and Functionality

The O(1) scheduler was a significant evolution introduced in the Linux 2.6 kernel. Its name reflects its core feature: all primary operations enqueueing, dequeueing, and selecting the next task execute in constant time [3, 2].

It uses two scheduling runqueues per CPU: the *active* and *expired* runqueue (Figure 3.1). Each runqueue holds 140 lists (one for each static priority). Processes are placed into the appropriate queue based on priority and run one after another within that priority level. When a process's time slice expires, it's moved to the expired runqueue. Once the active runqueue is empty, the runqueues are swapped [2, 12].

Key advantages are simplicity and predictable runtime behavior. However, it fails to dynamic adaptation of task types or modern hardware (e.g., hybrid cores, NUMA domains). It also suffers from inefficiencies in priority recalculation and interactive task detection [3, 1].

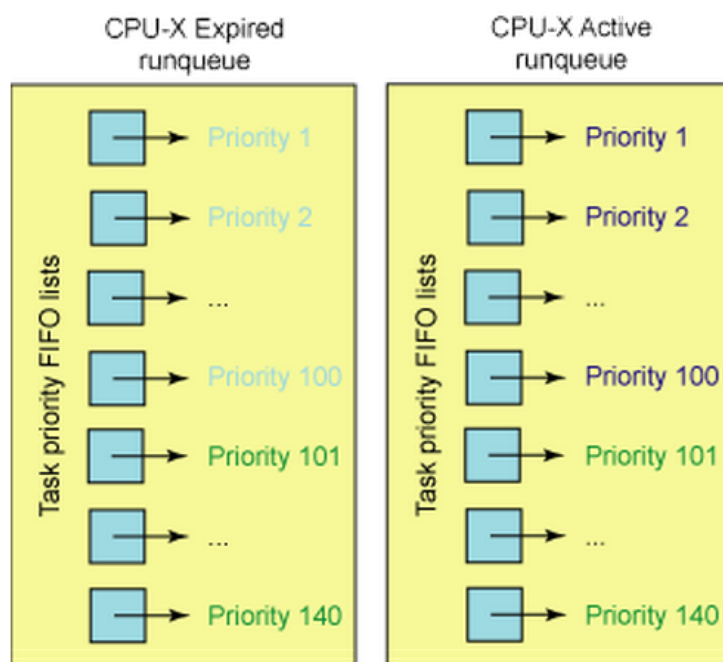


Figure 3.1 Active and Expired Runqueue
(source: [16])

3.2 Completely Fair Scheduler (CFS)

3.2.1 Principle and Functionality

CFS was introduced to improve fairness and latency control. It models an ideal multitasking processor, distributing CPU time proportionally using *virtual runtime* (vruntime), a counter tracking each task's consumed CPU time scaled by weight (derived from nice value) [4].

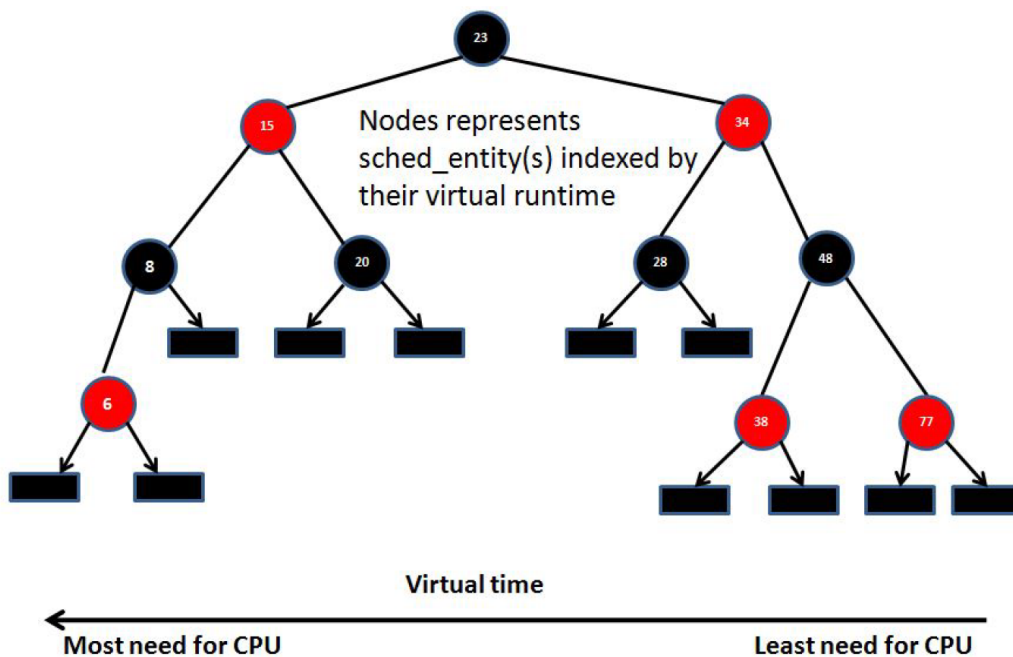


Figure 3.2 CFS red-black tree
(source: [16])

Tasks (sched_entity) are organized in a red-black tree (self-balancing binary search tree), sorted by vruntime (Figure 3.2). The task with the smallest vruntime is picked to run next. On each tick or wakeup, vruntime is updated, and tasks are reinserted in the tree.

This model avoids fixed priorities and allows CFS to handle CPU-bound and I/O-bound tasks more gracefully. Tasks that sleep or block do not suffer starvation thanks to vruntime compensation [12].

Unlike O(1), CFS provides soft fairness and avoids starvation. However, it does not provide explicit deadline support or latency guarantees. For real-time tasks, Linux uses separate RT scheduling classes [2]. CFS was integrated in Linux with version 2.6.23 [8].

3.3 EEVDF Scheduler

3.3.1 Principle and Functionality

The EEVDF (Earliest Eligible Virtual Deadline First) scheduler is an advanced proportional share process scheduling algorithm introduced to address some of the limitations of the CFS, especially in soft real-time performance and responsiveness. Originally proposed by Stoica and Hussein in 1996 [7], EEVDF enhances scheduling accuracy by integrating notions of fairness, service guarantees, and deadlines. Integrated into Linux with release 6.6 [8].

EEVDF assigns each task a virtual deadline, which represents the time by which the task should ideally be scheduled next. The algorithm ensures that the virtual deadlines are consistent with the task's allocated share of the CPU. At each scheduling decision point, the task with the earliest eligible virtual deadline (i.e., the one whose scheduled start time has passed) is chosen for execution [7].

Each task maintains the following properties:

- **Weight (w):** Represents the proportion of CPU time the task is entitled to.
- **Virtual Time (v):** A logical clock that tracks task progression based on CPU usage.
- **Eligible Time (e):** The earliest virtual time when the task is eligible to run again.
- **Virtual Deadline (d):** Calculated from the eligible time and the allocated share.

EEVDF enforces proportional fairness by attempting to meet each task's deadline, thus ensuring bounded latency for latency-sensitive applications without penalizing throughput. It also supports dynamic insertion and removal of tasks with a predictable impact on existing tasks [4].

In Linux, the EEVDF logic has been implemented as an extension of the fair scheduling class (SCHED_OTHER). Instead of vruntime tracking, each task's scheduling parameters are mapped to virtual deadlines and eligibility windows, improving latency and responsiveness for interactive workloads such as desktop GUIs, multimedia, or virtual machines [4].

EEVDF also decouples fairness enforcement from time slice size. Unlike the CFS algorithm, which implicitly binds time slice size to weight and load, EEVDF allows tighter control over latency by tuning deadline parameters independently [2].

3.4 Processor Affinity and Scheduling

Processor affinity, also known as CPU affinity, refers to binding or associating a process or thread to a specific CPU or subset of CPUs. This mechanism allows the operating system or user to control which processor a task is allowed to run on. Affinity can be statically set using user-space tools like taskset or cpuset, or dynamically managed by the kernel through APIs such as sched_setaffinity and control groups (cgroups) [6, 4].

Affinity is particularly relevant in modern multi-core and NUMA systems, where memory latency and CPU cache locality significantly affect performance. Maintaining affinity can improve cache reuse and reduce the overhead associated with task migration between cores.

3.4.1 Types of Affinity

- **Soft Affinity:** The kernel attempts to honor the affinity mask but may override it if necessary to balance the load.
- **Hard Affinity:** A strict binding where a task can only execute on the designated CPUs; enforced unless explicitly modified.

3.4.2 Kernel Implementation and Scheduling Domains

Linux uses "scheduling domains" and "load balancing groups" to define logical relationships among CPUs. These domains group CPUs by topology features, shared caches, memory locality, or SMT threads, and are the basis for balancing and affinity decisions [2, 3].

The process scheduler attempts to respect affinity settings when making task placement decisions, avoiding unnecessary migrations that would violate cache locality. Affinity masks are applied during task wakeups, fork operations, and load balancing passes. Overriding

affinity is only done in cases of imbalance, policy constraints, or resource starvation.

3.5 CPU Architecture Impact on Scheduling

Modern CPU architectures significantly influence how process schedulers perform. With the development of processors from basic single-core architectures to complex multi-core and heterogeneous configurations, the demands on the scheduler have increased accordingly. This subsection explores how architectural features like cache hierarchies, NUMA domains, simultaneous multithreading, and hybrid cores shape scheduling behavior and policy design [14].

3.5.1 Cache Hierarchy and Locality

Each core in modern CPUs typically has private L1 and L2 caches, with a shared L3 cache across multiple cores. Cache locality becomes critical: scheduling a task on the same core or within a group sharing a cache can significantly improve performance by reducing cache misses and memory latency. The Linux scheduler considers this in its load balancing heuristics and scheduling domains [14, 3].

3.5.2 NUMA Awareness

Non-Uniform Memory Access (NUMA) architectures divide memory into zones local to particular CPU sockets. Accessing remote memory is more expensive in terms of latency and bandwidth. Schedulers must prefer cores that are NUMA-local to the task's data, which can be influenced by memory allocation and processor affinity. NUMA balancing in Linux attempts to migrate memory pages or tasks to align them with optimal cores [3].

3.5.3 Simultaneous Multithreading

SMT, or Hyper-Threading in Intel terminology, enables multiple logical threads to share a single physical core. Schedulers must be aware of this sharing to avoid contention on shared

resources such as execution units and caches. Linux uses core sibling information from the CPU topology to avoid scheduling heavy tasks on sibling threads [2].

3.5.4 Heterogeneous Core Architectures

With the introduction of hybrid CPUs like the Intel Alder Lake processor series, which combines performance and efficiency cores, schedulers face the challenge of placing tasks on the most suitable cores. Performance cores are ideal for latency-sensitive tasks, while efficiency cores handle background or batch workloads. Linux is gradually integrating support for this through Energy-Aware Scheduling (EAS) and using feedback from hardware like Intel's Thread Director [15].

3.5.5 Scheduler Adaptations

To accommodate these architectural features, Linux implements scheduling domains, load balancing heuristics, and policies such as EAS and task affinity. The kernel exposes hardware topology via `/sys/devices/system/cpu/`, which helps tools like `lstopo` and `lscpu` gain visualization and insight. These tools are essential for tuning the system's scheduler to match hardware topology [3, 17].

Understanding these architectural aspects enables more efficient scheduler design and better tuning for both general-purpose and specialized workloads. The interplay between CPU architecture and scheduling policies will continue to grow in importance with increasing hardware heterogeneity.

4 PRESENT LINUX SCHEDULERS AND EVALUATIONS

Linux kernel versions used for evaluation are 6.1 for the CFS and 6.12 for the EEVDF scheduler. While many improvements have been made to the Linux kernel between these two versions, detailed patching of the kernel and back-porting scheduler implementation was considered outside the scope of this thesis. Nonetheless, these changes may influence results, and this is taken into account during final analysis.

Moreover, our evaluation considers comparisons with public benchmark results, particularly those reviewed in section 4.2.

4.1 Rationale for Kernel Selection

As described in Chapter 5, kernel version 6.1 was selected as the representative baseline for testing the CFS scheduler. At this stage, CFS was still the default scheduler.

Kernel version 6.12, in contrast, is the first stable release where EEVDF has been promoted to the default scheduler. Choosing this version ensures that the evaluation of EEVDF reflects its intended behavior under default Linux configurations.

The thesis does not cover the numerous optimizations introduced by upstream development beyond the scheduler, including enhancements in power management, I/O handling, and CPU topology awareness. Our objective is not to isolate EEVDF patches from the rest of kernel evolution, but rather to understand the realistic performance and behavior of CFS and EEVDF in their respective upstream contexts. The implications of these kernel-level differences are further examined in comparison to the results from established benchmarks in section 4.2.

4.2 Performance Insights from Early Evaluation

The Phoronix report on Linux 6.10, 6.11, and 6.12 focusing on AMD Zen 5 systems provides early public benchmarks comparing the performance of CFS and EEVDF [18]. The key findings include:

- Comparable or improved performance in many desktop and mixed-use workloads, tested by code compilation task. Result visible in Figure 4.1.
- Stable latency behavior under interactive conditions.
- Improvement performance in server oriented workloads, simulated by hackbench test. Results shown by Figure 4.2).

Timed LLVM Compilation 16.0

Build System: Ninja

ptsli
Phoronix.com

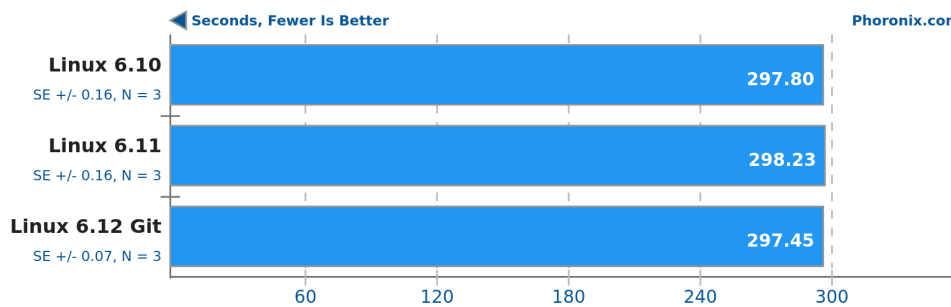


Figure 4.1 Many workloads simulated compilation task.

(source: [18])

Hackbench

Count: 32 - Type: Process

ptsli
Phoronix.com

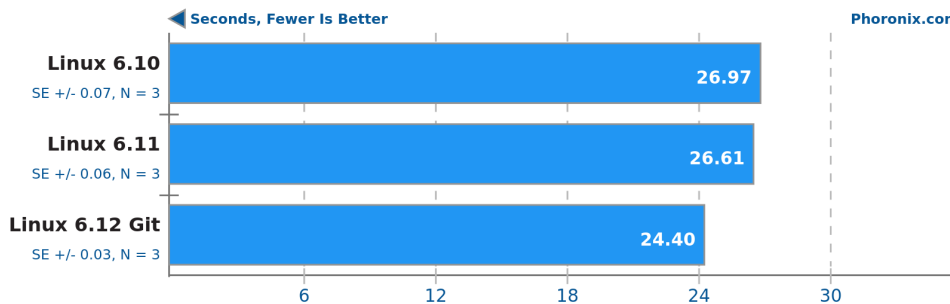


Figure 4.2 Scheduler benchmarks by hackbench.

(source: [18])

These results suggest EEVDF is highly compatible with modern heterogeneous computing environments that prioritize responsiveness and fairness across diverse workloads.

4.3 Hypotheses for the results evaluation

The objective of this thesis is to develop and apply a repeatable methodology for evaluating process schedulers in the Linux kernel. To guide the interpretation of measured data and

support meaningful conclusions, the following hypotheses have been formulated based on theoretical expectations and available benchmark data (4.2):

1. **EEVDF is expected to reduce maximum latency** under interactive workloads with comparison to CFS, due to its earliest eligible virtual deadline mechanism, which prioritizes tasks based on time sensitive eligibility [7, 4].
2. **EEVDF may incur more frequent context switches** than CFS under mixed-priority workloads, due to its aggressive preemption of overdue tasks. This increase may be a tradeoff for improving fairness and responsiveness [4].
3. **EEVDF should better preserve fairness** in CPU distribution of time among diverse workloads, particularly where tasks have varying weights, by more precisely managing their virtual deadlines [7].
4. **EEVDF is anticipated to scale better** across higher thread counts, retaining efficiency with increasing parallelism, due to its virtual time alignment and finer-grained task eligibility tracking [4].
5. **CFS may provide slightly higher throughput** on uniform, computation intensive workloads due to its lower overhead and mature red-black tree scheduling design [3].
6. **Starvation is not expected to occur** in either CFS or EEVDF. Both schedulers has implemented mechanisms to ensure long-term fairness and prevent indefinite task postponement, with CFS using minimum granularity and task aging, and EEVDF enforcing virtual deadlines to ensure bounded delay [4, 2, 7].

These hypotheses are evaluated through the experiments presented in Chapter 7, with results evaluated in detail in Chapter 8.

II PRACTICAL PART

5 PROCESS SCHEDULER TESTING METHODOLOGY

All the scripts described throughout this chapter can be found in Appendix 2, within the `test_scripts` directory.

5.1 Linux distribution selection and test environment

A stable, consistent operating system environment is fundamental for meaningful process scheduler evaluation. For this reason, the Debian GNU/Linux distribution was chosen due to its reputation for reliability, transparency in package management, and strong upstream alignment with the mainline Linux kernel [19].

Two recent Debian releases are used:

- **Debian Bookworm (12)** – A stable release with widespread adoption and long-term support, used to test CFS [20].
- **Debian Trixie (13)** – A development branch tracking newer kernel version 6.12, used to test EEVDF [21].

A minimal, reproducible test environment was built using `mkosi`, a tool for generating bootable Debian images with defined kernel versions and configurations [22].

`mkosi` was selected for:

- Its ability to construct custom root file systems directly from Debian source packages.
- Its integration with QEMU for virtualized, isolated testing.
- Support for injecting post-install scripts, essential for tests deployment.

Using `mkosi`, two images were built: one based on Debian Bookworm with Linux 6.1 (representing CFS), and one on Debian Trixie with Linux 6.12 (representing EEVDF). This setup ensures consistent userspace and system behavior across both test targets, with the kernel version as the only significant variable. Technical details are listed in Chapter 6.

5.2 Scheduler latency test

The latency test is designed to measure system process scheduling latency under load. At start, the script first detects the number of logical CPUs available by parsing `/proc/cpuinfo` and stores it in `threads` variable. Then it initiates a background load using the **hackbench** benchmarking tool, which stresses the scheduler by simulating multiple groups of concurrent tasks with specified parameters listed in Table 5.1 [23].

Table 5.1 Hackbench configuration for latency test

Parameter	Value	Description
-g	20	Number of groups of sender/receiver pairs to start.
-l	10000	Number of messages each sender will send.
-s	512	Size of each message in bytes.
-f	25	Number of file descriptors per sender.
-T	N/A	Use threads instead of processes.

(source: [23], values own)

After launching the hackbench workload in the background, the script introduces a 0.5 s delay to allow the system load to stabilize. Following this, the script uses the **cyclicttest** utility to perform latency measurements. Cyclicttest is executed with the same priority for all threads. It creates a thread on each CPU core, ensuring distributed latency measurement across cores. The test parameters are listed in Table 5.2, selected and configured to ensure a sufficient number of samples are collected for latency statistics. Figure 5.1 represents a flow chart of the test execution. However, if the background load task finishes earlier, cyclicttest logging is stopped too [24].

Table 5.2 Cyclicttest configuration for latency test

Parameter	Value	Description
-mlockall	N/A	Lock all memory to prevent paging.
-nsecs	N/A	Show results in in nanoseconds.
-priority	80	Set scheduling priority for all threads.
-interval	200	Interval between wake-ups in microseconds.
-distance	0	Disable spreading threads across sockets.
-loop	800000	Number of test loop iterations.
-threads	<code>\$threads</code>	Number of test threads equal to number of CPUs.
-affinity	<code>0-\$((\$threads-1))</code>	CPU affinity set, one thread per CPU

(source: [24], values own)

This test simulates a real-world loaded system and measures how predictably the system scheduler can wake up real-time threads under stress, which is critical for real-time and low-latency system evaluation.

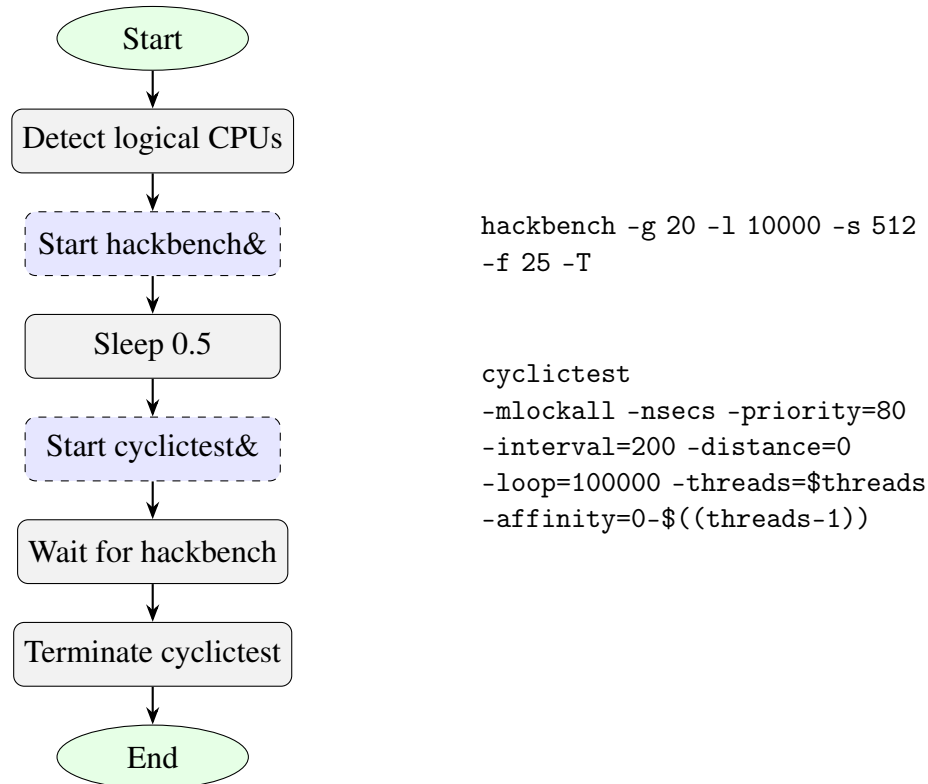


Figure 5.1 Latency test flow chart
(source: own)

5.2.1 Test output data format

Latency test results are printed on standard output as plain text from where they can be stored to a file or for different use. Output is from both hackbench and cyclicttest. Each thread's latency statistics are shown individually, followed by system-level load.

Example of test output for a system with 12 logical CPUs is shown by Listings 5.1.

- The hackbench section provides information about groups, descriptors, and thread (tasks) counts, configured based on Table 5.1.
- The cyclicttest section logs Min/Act/Avg/Max latency (in microseconds) for each thread (T), parameter (C) shows current cycle defined by parameter --loop described in Table 5.2 [24].

- The output is sampled periodically and reflects the impact of background stress on thread wake up accuracy. Sampling rate is set by parameter `--interval` (Table 5.2) and in output shown as parameter (I).

```

Scheduler Latency Test
Current logical CPUs: 12
Starting hackbench...
Running in threaded mode with 20 groups using
50 file descriptors each (== 1000 tasks)
Each sender will pass 10000 messages of 512 bytes
Starting cyclicttest latency measurement...
/dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 100.83 41.57 17.09 497/1192 2830

T: 0(2819)P:80I:200C:1811Min:4370Act:5351Avg:6100Max:64088
T: 1(2820)P:80I:200C:3313Min:4649Act:7197Avg:6115Max:69492
T: 2(2821)P:80I:200C:1863Min:4395Act:5876Avg:5819Max:63685
T: 3(2822)P:80I:200C:692Min:4882Act:10253Avg:6595Max:73326
T: 4(2823)P:80I:200C:2838Min:4508Act:6187Avg:6561Max:56195
T: 5(2824)P:80I:200C:1458Min:4579Act:6532Avg:6271Max:40631
T: 6(2825)P:80I:200C:2790Min:4625Act:5944Avg:6296Max:61920
T: 7(2826)P:80I:200C:1226Min:4540Act:6030Avg:5807Max:45051
T: 8(2827)P:80I:200C:2813Min:4908Act:6362Avg:6523Max:53835
T: 9(2828)P:80I:200C:2711Min:4601Act:5345Avg:6280Max:39812
T:10(2829)P:80I:200C:353Min:4783Act:8638Avg:6260Max:36365
T:11(2830)P:80I:200C:2539Min:4543Act:6829Avg:5959Max:13183
[14Apolicy: fifo: loadavg: 125.51 47.67 19.19 481/1192 2830

T: 0(2819)P:80I:200C:2167Min:4370Act:5829Avg:6129Max:64088
T: 1(2820)P:80I:200C:3670Min:4649Act:5737Avg:6129Max:69492
T: 2(2821)P:80I:200C:2219Min:4395Act:4871Avg:5869Max:63685
T: 3(2822)P:80I:200C:1049Min:4616Act:5145Avg:6498Max:73326
T: 4(2823)P:80I:200C:3195Min:4371Act:5027Avg:6512Max:56195
T: 5(2824)P:80I:200C:1815Min:4579Act:7596Avg:6269Max:40631
T: 6(2825)P:80I:200C:3147Min:4625Act:5065Avg:6323Max:61920
T: 7(2826)P:80I:200C:1582Min:4540Act:5296Avg:5824Max:45051
T: 8(2827)P:80I:200C:3170Min:4859Act:5843Avg:6508Max:53835
T: 9(2828)P:80I:200C:3068Min:4505Act:6980Avg:6264Max:39812
...

```

Listing 5.1 Reduced sample latency test output (source: own)

5.3 Scheduler fairness latency test

The Scheduler Fairness Test assesses the fairness characteristics of the system's scheduler under varying priority conditions. Initially, the script determines the total number of logical CPUs by querying `/proc/cpuinfo` and also retrieves the system's clock ticks per second (CLK_TCK).

To simulate diverse priority levels, four groups of processes are launched concurrently, each running the **hackbench** benchmarking tool with specific parameters detailed in Table 5.3. Each group is assigned distinct nice levels (ranging from -5 to 18) to represent varied scheduler priorities:

- Group A: nice 18 (lowest priority)
- Group B: nice 10
- Group C: nice 5
- Group D: nice -5 (highest priority)

Table 5.3 Hackbench configuration for fairness test

Parameter	Value	Description
-s	512	Size of each message in bytes.
-l	90000	Number of messages each sender sends.
-g	\$threads	Number of S/R groups (based on logical CPU).
-f	6	Number of file descriptors per sender.
-P	N/A	Use processes instead of threads.

([23], values own)

Sleep of 100 ms is introduced to give the system time to spawn all hackbench processes for the lowest priority group. After initiating all groups, the script captures the process identifiers (PIDs) of spawned child processes from each group. It then enters a periodic monitoring loop, sampling at 100 ms intervals. During each iteration, the script records timestamps, process start times, current niceness values, and CPU usage statistics (user and system times) for each tracked process by parsing the `/proc/[pid]/stat` file [25].

The logging continues until all hackbench groups complete execution, producing a data set to analyze scheduler fairness across different priority conditions. Figure 5.2 represents a flow chart of the test execution.

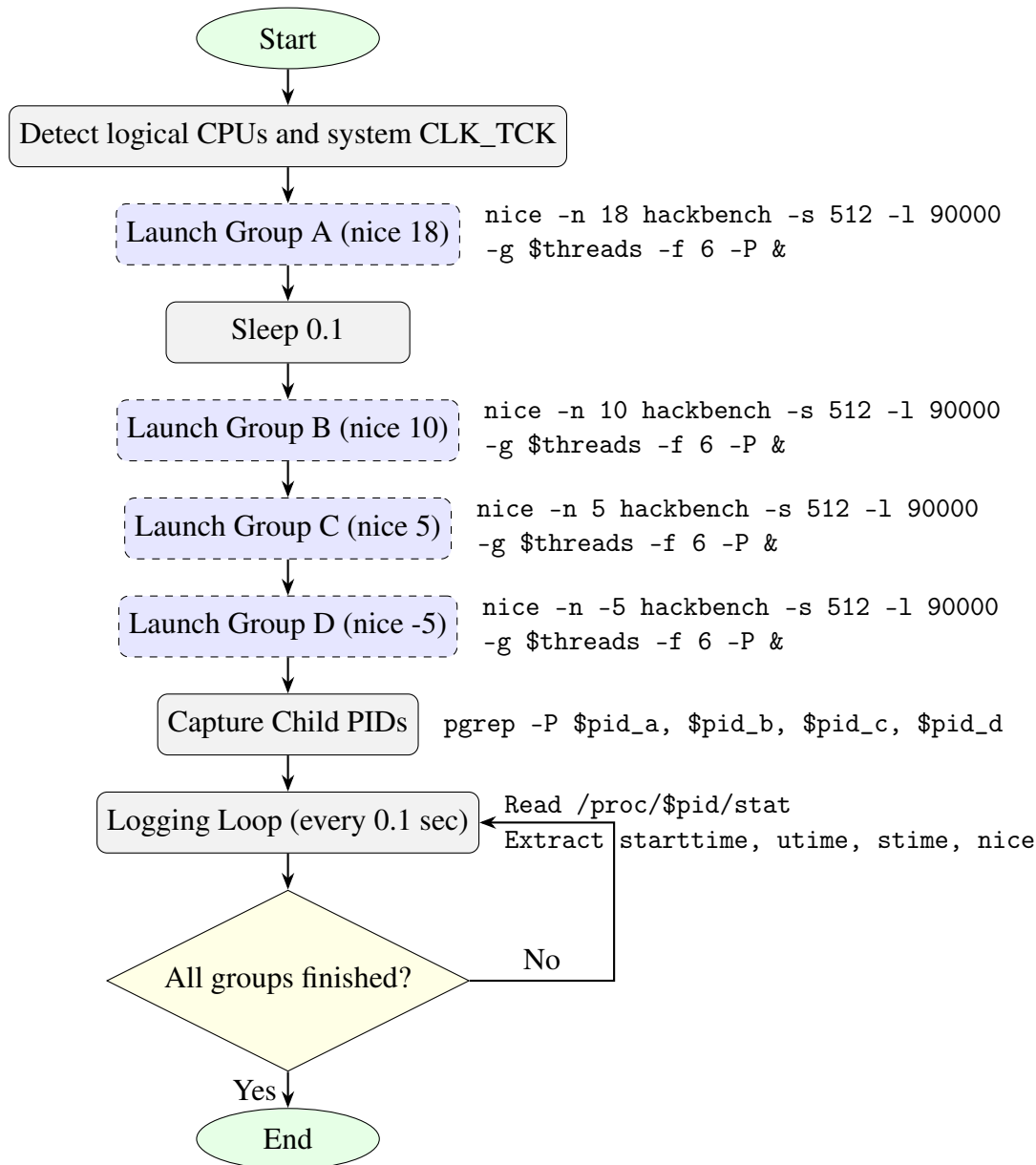


Figure 5.2 Fairness test flow chart
(source: own)

5.3.1 Test output data format

The fairness test output is printed on standard output in plain text format per-thread metrics sampled at 100 ms intervals throughout the test duration. Listings 5.2 show reduced example output.

At the beginning, the system prints the number of logical CPUs and the value of CLK_TCL, followed by output from the start of hackbench instances for each nice group. After that, started PIDs are captured and the sum of them is printed for back check possibility that all PIDs are logged.

Each row in the PID log output contains values for: timestamp with nanoseconds, process start time (in clock ticks since system boot), PID, process nice value, accumulated user and system time (in clock ticks).

```

=== Scheduler Fairness Test with PID Logging ===
Logical CPUs: 12
CLK_TCK: 100
Starting Group A (nice 18)
Starting Group B (nice 10)
Starting Group C (nice 5)
Starting Group D (nice -5)
Running in process mode with 12 groups using 12 file
descriptors each (== 144 tasks)
Running in process mode with 12 groups using 12 file
descriptors each (== 144 tasks)
Running in process mode with 12 groups using 12 file
descriptors each (== 144 tasks)
Running in process mode with 12 groups using 12 file
descriptors each (== 144 tasks)
Total PIDs being logged: 576
timestamp,starttime,pid,nice,utime,stime
Start logging
1746555053.031600157,59978,795,18,0,0
1746555053.031600157,59978,799,18,0,0
...
1746555088.295708863,59978,959,10,3,94
1746555088.295708863,59978,963,10,3,94
...
1746555088.295708883,59978,859,5,1,88
1746555088.295708883,59978,863,5,4,54
...
1746555088.295708893,59978,659,-5,1,44
1746555088.295708893,59978,663,-5,2,34
...

```

Listing 5.2 Reduced sample fairness test output (source: own)

5.4 Scheduler throughput test

The Scheduler Throughput Test is designed to evaluate the CPU scheduler's efficiency under varying load scenarios using the benchmarking utilities **sysbench** and **stress-ng**. The test comprises four distinct scenarios intended to measure throughput both with and without background load stress. Initially, the script determines the number of logical CPUs by inspecting `/proc/cpuinfo` [26, 27, 28]. The throughput test scenarios are executed as follows:

- **Scenario 1 (30 seconds, no background load):**
 - Executes **sysbench** with full CPU cores utilization for 30 seconds, measuring throughput without any additional load.
- **Scenario 2 (30 seconds, with stress-ng load):**
 - Starts **stress-ng** as a background process for 33 seconds (30 seconds plus a 3 second buffer).
 - After 2 seconds, runs **sysbench** concurrently, again utilizing all CPU cores for 30 seconds to measure performance under stress.
- **Scenario 3 (100 seconds, no background load):**
 - Runs **sysbench** independently for an extended duration of 100 seconds to assess throughput stability without additional load.
- **Scenario 4 (100 seconds, with stress-ng load):**
 - Initiates **stress-ng** in the background for 103 seconds (100 seconds with a 3 second buffer).
 - After a short delay of 2 seconds, executes **sysbench** simultaneously for 100 seconds, evaluating system throughput under prolonged stress conditions.

Between each scenario, a 5 second timeout is introduced to stabilize the system environment. This ensures the evaluation of scheduler performance, capturing throughput variations in both idle and heavily loaded states. Figure 5.3 represents a flow chart of the test execution.

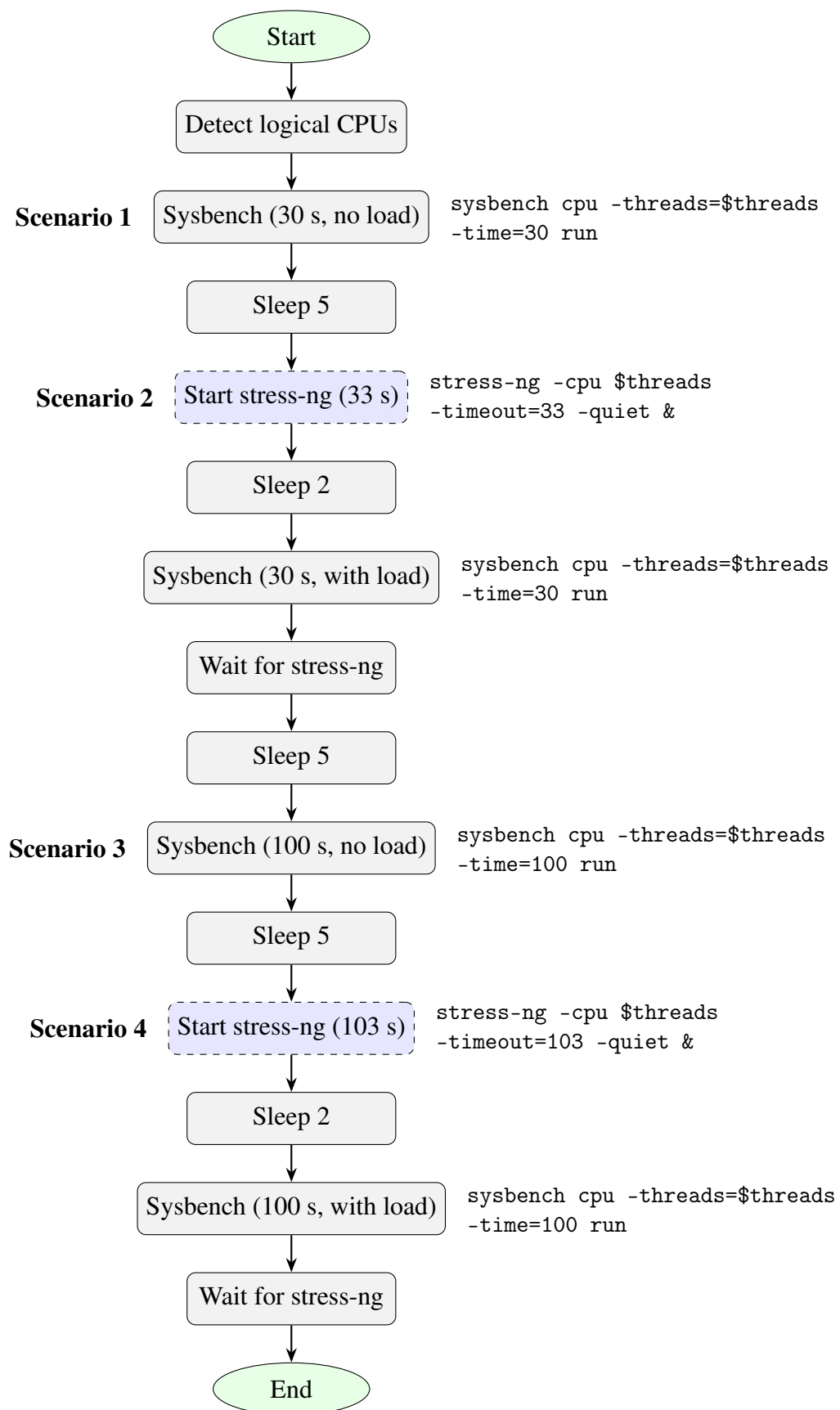


Figure 5.3 Throughput test flow chart
(source: own)

5.4.1 Test output data format

The throughput test logs the performance of sysbench CPU-bound workloads under different stress conditions. Output is segmented by scenario, each reflecting a specific test case described in section 5.4. Output is printed on standard output in plain text format in a similar fashion as previous tests. Listings 5.3 show reduced throughput test log output.

Each section starts with a scenario header, followed by detailed output from sysbench, including throughput metrics, latency statistics, and fairness indicators. For the process scheduler throughput evaluation, "**total number of events:**" value is evaluated for each scenario [27].

```

Current logical CPUs: 12
Scenario 1: Sysbench (12 threads, 30s, no background load)
Running the test with following options:
Number of threads: 12
Initializing random number generator from current time
Prime numbers limit: 10000
Initializing worker threads...
Threads started!

CPU speed:
  events per second: 11754.95
General statistics:
  total time:                               30.0010s
  total number of events:                   352676
Latency (ms):
  min:                                     0.54
  avg:                                     1.02
  max:                                     19.00
  95th percentile:                         1.03
  sum:                                     359893.39
Threads fairness:
  events (avg/stddev):       29389.6667/27.96
  execution time (avg):      29.9911

Scenario 2: Sysbench (12 threads, 30s, with
               stress-ng background load)
...

```

Listing 5.3 Reduced sample throughput test output (source: own)

5.5 Scheduler context switching test

The Context Switching Test is designed to analyze how the scheduler handles frequent thread switches under heavy CPU load. It uses **pidstat** to monitor involuntary context switches on a per-thread basis, while generating synthetic load using **stress-ng** [28, 29].

At the beginning of the test, the script determines the number of logical CPUs by parsing `/proc/cpuinfo`. It then starts a CPU bounded workload using **stress-ng**, configured to spawn a number of worker threads equal to the number of logical CPUs. Sleep for 2 seconds is introduced to enable the load to properly start. Configuration listed by table 5.4. The load runs for a slightly longer duration than the monitoring phase to ensure consistent system stress [26].

The context switch metrics are collected using the **pidstat** utility, which outputs detailed per-thread context switch statistics over time, including voluntary and involuntary switches. **Pidstat** configuration is shown by table 5.5. Sampling occurs every 1 second over 30 sample reports [29].

Table 5.4 Stress-ng configuration for fairness test

Parameter	Value	Description
-cpu	\$threads	Number of worker threads.
-timeout	33	Run duration in seconds.
-quiet	N/A	Suppress detailed output.

(source: [28], values own)

Table 5.5 Pidstat configuration for fairness test

Parameter	Value	Description
-w	N/A	Report task switching statistics.
-t	N/A	Display threads associated statistics.
interval	1	Amount of time in seconds between each report.
count	30	Total number of reports generated.

(source: [29], values own)

This test is critical in evaluating how efficiently the scheduler performs under conditions that force frequent thread preemption and wake-up activity, typical in highly parallel or latency-sensitive applications. Figure 5.4 represents a flow chart of the test execution.

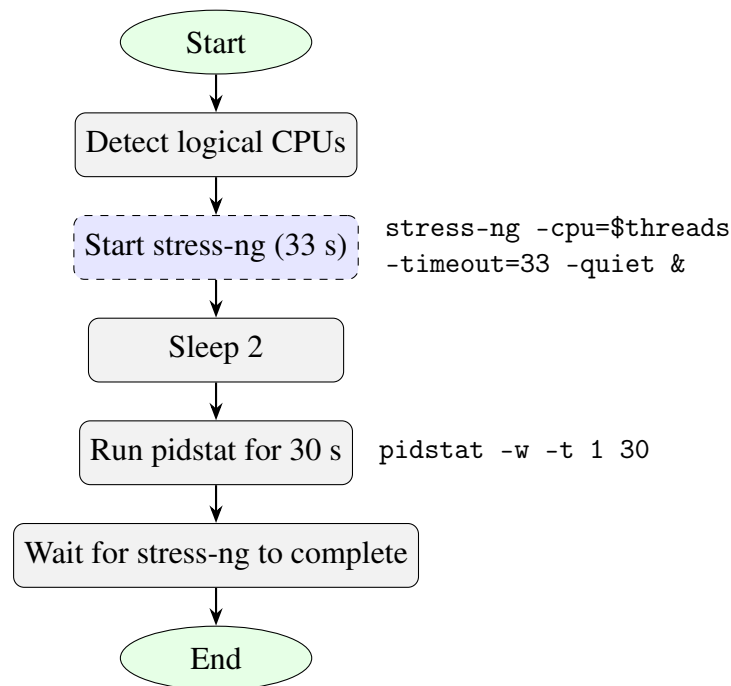


Figure 5.4 Context switching test flow chart
(source: own)

5.5.1 Test output data format

The context switching test output logs stress-ng context switch statistics over time using pidstat. The output consists of entries collected at 1 second intervals over a 30 second window (table 5.5), showing voluntary and involuntary context switches per thread.

Stress-ng configuration uses a CPU-bound worker type task by parameter `--cpu`, visible in table 5.4, which does not generate any voluntary context switches (cswch/s). Only context switches which are non-voluntary are present (nvcswh/s).

By measuring and analyzing this metric for stress-ng-cpu thread, evaluation of how the scheduler handles frequent thread switches under heavy CPU load can be performed [29]. Listings 5.4 show reduced test log output.

```

Current logical CPUs: 12
Test duration: 30 seconds
=== Scheduler Context Switching Test ===
>>> Measuring per-thread context switches with pidstat -w -t
Starting stress-ng again for detailed tracking...
--- pidstat -w -t output ---
  
```

```
Linux 6.1.0-34-amd64 (localhost) 05/06/25 _x86_64_ (12 CPU)
```

	UID	TGID	TID	cswch/s	nvcswch/s	Command
18:09:56	0	772	-	0.00	87.00	stress-ng-cpu
18:09:56	0	773	-	0.00	3.00	stress-ng-cpu
18:09:56	0	774	-	0.00	1.00	stress-ng-cpu
18:09:56	0	775	-	0.00	9.00	stress-ng-cpu
18:09:56	0	776	-	1.00	1.00	pidstat
...						
18:10:00	0	771	-	0.00	2.00	stress-ng-cpu
18:10:00	0	772	-	0.00	7.00	stress-ng-cpu
18:10:00	0	773	-	0.00	2.00	stress-ng-cpu
18:10:00	0	774	-	0.00	3.00	stress-ng-cpu
...						

Listing 5.4 Reduced sample context switching output (source: own)

5.6 Scheduler scalability test

The Scalability Test evaluates how well the system scheduler handles increasing thread concurrency by measuring throughput across different thread counts using the **sysbench** CPU workload. Used configuration parameters are shown by table 5.6 [27].

At first, the script determines the number of logical CPUs by parsing `/proc/cpuinfo`. The script iterates over a series of thread levels, based on a powers of two sequence (1, 2, 4, 8, 16, 32, 64, 128) and executes **sysbench** with each thread count if it is less than or equal to the number of available CPU threads.

Each configuration runs **sysbench** for 30 seconds, with a 5 second pause between runs to allow the system to settle. If the maximum number of detected logical CPUs is not already covered in the predefined thread levels, it is tested explicitly in an additional final run.

This test analyzes the scalability of the system's CPU scheduler under increasing workloads, identifying how performance scales with parallel thread execution. Figure 5.5 represents a flow chart of the test execution.

Table 5.6 Sysbench configuration for scalability test

Parameter	Value	Description
cpu	N/A	Run the CPU benchmark mode.
-threads	1 to \$threads	Number of threads used for run.
-time	30	Test duration in seconds.
run	N/A	Execute the test with given parameters.

(source: [27], values own)

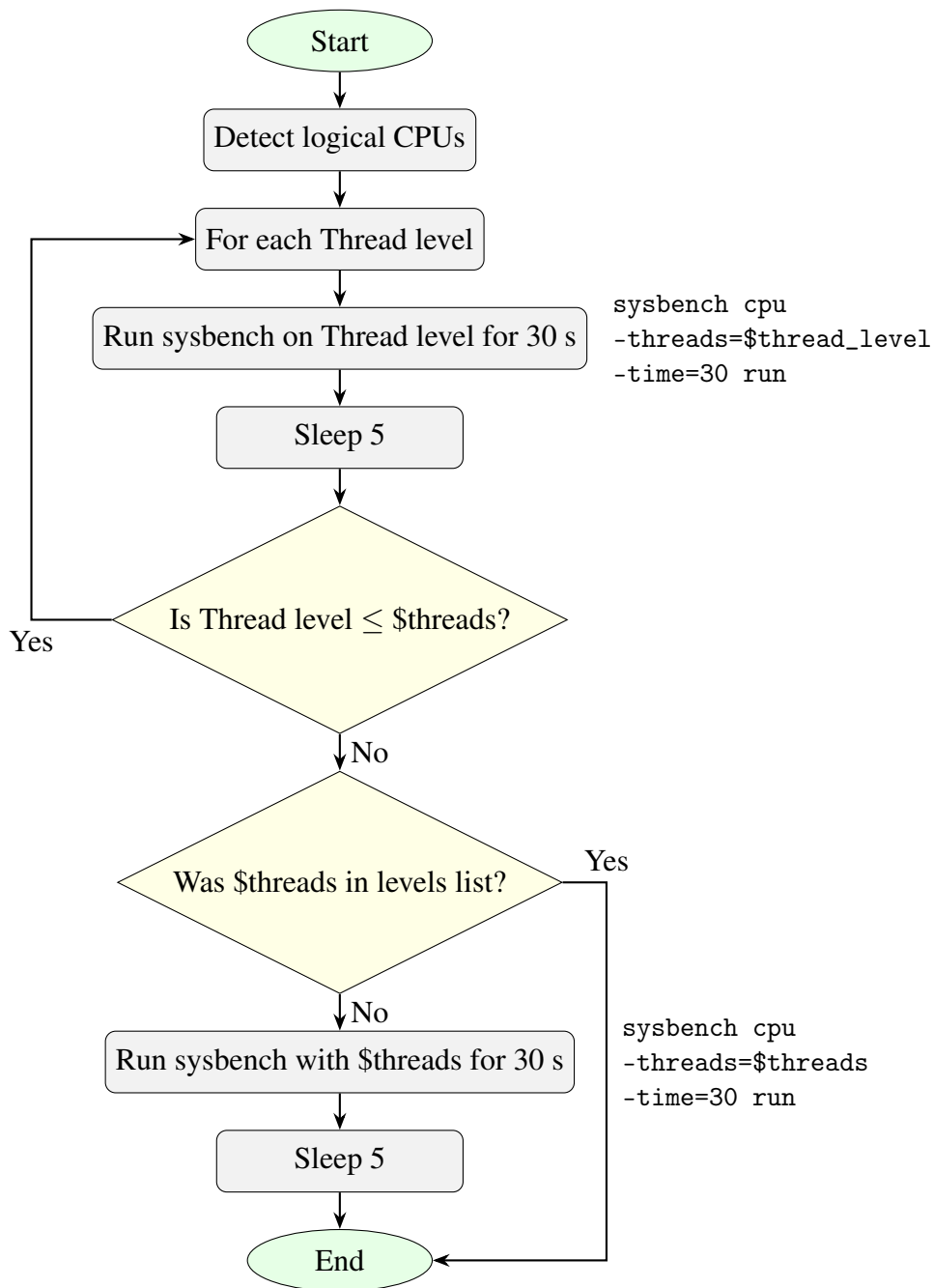


Figure 5.5 Scalability test flow chart

(source: own)

5.6.1 Test output data format

The scalability test prints out sysbench output for each thread level. Detailed functionality is described in 5.6. Listings 5.5 show reduced test log output.

Each section begins with a header indicating the thread count used, followed by raw output from sysbench, which consists of various metrics where "events per second:" is the one used for the analysis of system scalability performance.

```
Current logical CPUs: 12
=== Scheduler Scalability Test ===
Test duration per run: 30s

>> Running sysbench with 1 threads
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)
Running the test with following options:
Number of threads: 1
Initializing random number generator from current time
Prime numbers limit: 10000
Initializing worker threads...
Threads started!
CPU speed:
    events per second:  1926.94
...
>> Running sysbench with 2 threads
...
CPU speed:
    events per second:  3851.97
...
>> Running sysbench with 4 threads
...
CPU speed:
    events per second:  7702.36
...
>> Running sysbench with 12 threads (max detected)
...
CPU speed:
    events per second: 11615.80
...
=== Scalability test complete ===
```

Listing 5.5 Reduced sample scalability test output (source: own)

5.7 Scheduler starvation detection test

The starvation detection test is designed to verify whether low-priority CPU tasks are granted a fair share of CPU time when the system is under pressure from high-priority workloads. It simulates an imbalance by concurrently launching two groups of tasks using the `hackbench` utility: a large high-priority group and a small low-priority group. Figure 5.6 shows the execution flow of the test script.

The high-priority group is configured with a niceness of `-10` and launches an amount of processes dependent on the number of system logical CPUs. The low-priority group uses a fixed configuration of just 2 sender/receiver pairs equal to 16 processes total with a niceness of `19`. Each group uses the same number of loop iterations (120 k), file descriptors (8), and message size (512 B) for consistent load characteristics [30].

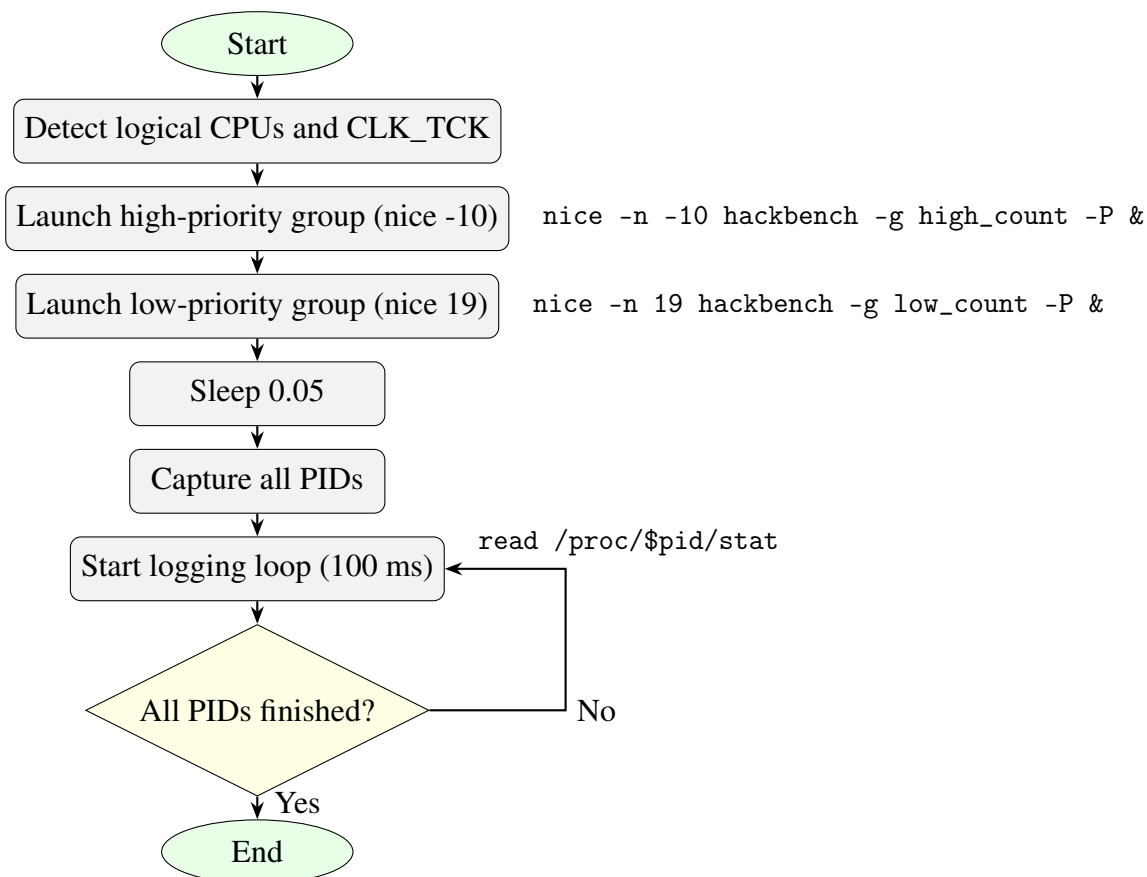


Figure 5.6 Starvation detection test flow chart

(source: own)

To accommodate process spawning overhead, the script inserts a short 50 ms delay after

launch. It then captures all child process IDs and begins logging CPU usage statistics by reading the `/proc/[pid]/stat` interface for each task. Samples are printed on standard output in a period of 100 ms. The loop continues until all processes are terminated [29, 23].

5.7.1 Test output data format

Each data line in the log is a comma-separated record representing a single process sample. Listings 5.2 show output example. It consists of the fields: `timestamp`, `starttime`, `pid`, `nice`, `utime`, `stime`. The timestamp is the wall-clock time when the sample was taken. The start time is the process start time in clock ticks since system boot, taken from `/proc/<pid>/stat`. The PID is the process ID, and `nice` indicates its scheduling priority. The fields `utime` and `stime` represent the amount of CPU time the process has spent in user mode and system mode respectively, also in clock ticks. Each sampling period ends with the line `"end sample"` which serves as a delimiter for parsing the log in chunks [29].

```
=== Scheduler Starvation Test ===
Logical CPUs: 32
CLK_TCK: 100
Starting high-priority load (nice -10)
Starting victim group (nice 19)
Running in process mode with 64 groups
    using 8 file descriptors each (== 512 tasks)
Each sender will pass 120000 messages of 512 bytes
Running in process mode with 2 groups
    using 8 file descriptors each (== 16 tasks)
Each sender will pass 120000 messages of 512 bytes
High-priority PIDs: 512
Victim PIDs: 16
Total PIDs being logged: 528
timestamp,starttime,pid,nice,utime,stime
1747331454.631334993,27596987,3164495,-10,3,38
1747331454.631334993,27596987,3164496,-10,1,39
...
1747331455.266989431,27596987,3164523,19,0,3
1747331455.266989431,27596987,3164525,19,0,3
end sample
...
```

Listing 5.6 Reduced sample starvation test output (source: own)

5.8 System environment information

Before any test is executed, it is essential to capture the hardware and kernel environment configuration to ensure reproducibility and assist with result interpretation. The script gathers system metadata relevant to scheduler behavior and performance. Figure 5.7 shows script execution flow.

- **CPU architecture, vendor, and core/thread layout** – collected using multiple `lscpu` queries [17].
- **Memory availability and usage** – obtained through a snapshot from the `free -h` utility showing RAM and swap state [31].
- **NUMA and core affinity topology** – captured using `lstopo-no-graphics`, which prints the physical CPU layout and NUMA domains [32].
- **Kernel version** – reported via `uname -r` to record active kernel version [33].
- **Scheduler parameters** – iterated directly from `/proc/sys/kernel/sched*` [34].

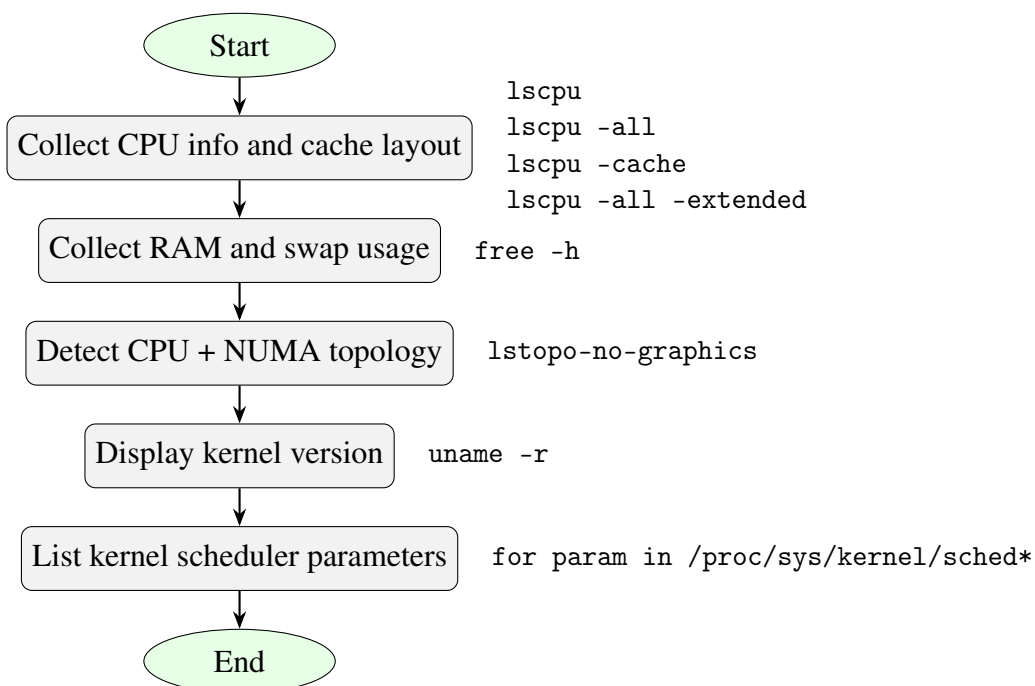


Figure 5.7 System environment collection flow chart
(source: own)

5.8.1 Script output data format

The script prints a structured summary to standard output divided into logical sections: CPU architecture and cache hierarchy, memory usage snapshot, CPU and NUMA topology, kernel version and all tunable kernel scheduler parameters. Each section is clearly labeled and presented in human-readable form shown by reduced output in Listings 5.7.

```

=== CPU Information ===
Architecture:                x86_64
Byte Order:                  Little Endian
CPU(s):                      16
Vendor ID:                   GenuineIntel
BIOS Vendor ID:              Intel(R) Corporation
Model name:                  12th Gen Intel(R) Core(TM) i7-1260P
L1d cache:                   448 KiB (12 instances)
L1i cache:                   640 KiB (12 instances)
L2 cache:                    9 MiB (6 instances)
L3 cache:                    18 MiB (1 instance)
...
=== RAM Information ===
      total    used    free shared    buff/cache    available
Mem:   46Gi    618Mi   46Gi  9.3Mi        99Mi        46Gi
Swap:    0B     0B      0B
=== CPU Topology ===
Machine (47GB total)
  Package L0
    NUMANode L0 (P0 47GB)
      L3 L0 (18MB)
        L2 L0 (1280KB)+L1d L0(48KB)+L1i L0(32KB)+CoreL#0
          PU L0 (P0)
          PU L1 (P1)
...
=== Kernel Version ===
6.1.0-34-amd64
=== Scheduler Settings (/proc/sys/kernel/sched*) ===
sched_autogroup_enabled: 1
sched_cfs_bandwidth_slice_us: 5000
sched_child_runs_first: 0
sched_deadline_period_max_us: 4194304
...

```

Listing 5.7 Reduced sample system environment log (source: own)

5.9 System initialization

Before running any scheduler tests and benchmarks, the system must be initialized into a known state to ensure consistent test conditions and to prepare the environment for data persistence.

- **Applies unified kernel scheduler settings** – A predefined set of settings is configured to ensure consistent scheduling behavior across test runs. These include settings such as enabling autogroup scheduling, defining real-time execution limits or tuning the CFS slice length. Complete list of parameters in Table 5.7.
- **Mounts a persistent output partition** – A separate disk partition labeled OUTPUT (details in 6.2.4) is mounted at /media/output. This ensures test results are stored in a consistent location across reboots or test re-executions.

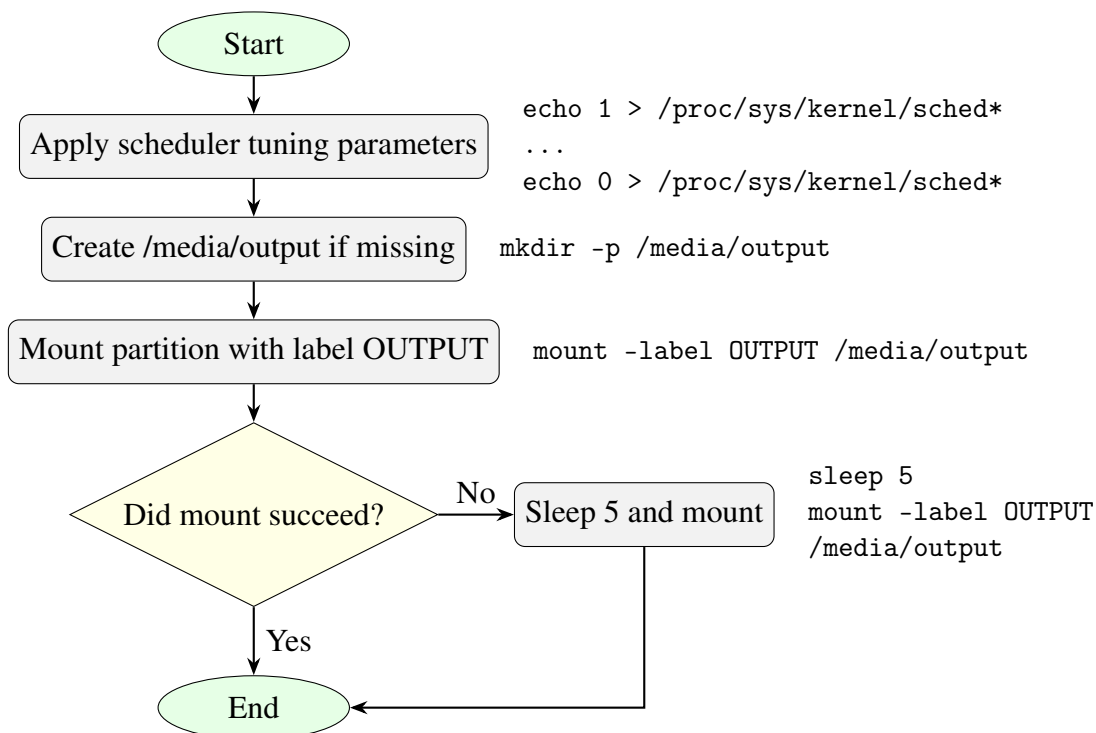


Figure 5.8 System initialization flow chart

(source: own)

Table 5.7 Unified scheduler tuning parameters

Parameter	Value	Description
<code>sched_autogroup_enabled</code>	1	Enables automatic task group scheduling for better interactivity in desktop and mixed workloads.
<code>sched_cfs_bandwidth_slice_us</code>	5000	Time slice (μ s) allocated for CFS bandwidth control.
<code>sched_child_runs_first</code>	0	Disables child-first execution; parent continues immediately after forking.
<code>sched_deadline_period_max_us</code>	4194304	Upper bound for deadline scheduling period (in microseconds).
<code>sched_deadline_period_min_us</code>	100	Minimum allowed period for tasks under deadline policy.
<code>sched_energy_aware</code>	0	Disables energy-aware scheduling.
<code>sched_itmt_enabled</code>	1	Enables Intel Turbo Mode-aware task placement on asymmetric frequency cores.
<code>sched_rr_timeslice_ms</code>	100	Defines timeslice (ms) for round-robin tasks.
<code>sched_rt_period_us</code>	1000000	Maximum real-time scheduling period (μ s).
<code>sched_rt_runtime_us</code>	950000	Runtime quota within RT period; prevents starvation by limiting RT CPU time.
<code>sched_schedstats</code>	0	Disables internal scheduler statistics to reduce runtime overhead during tests.

(source: [34])

5.10 Test procedure execution

The script automates the execution of all implemented scheduler evaluation tests. It provides a consistent and repeatable way to manage multiple test runs and capture all relevant outputs in an organized structure. Script parameter definition is listed in Table 5.8. Figure 5.9 visualizes the flow of the script execution.

The script performs the following steps:

- Run system init. script (5.9) to mount OUTPUT partition and apply scheduler settings.
- Creates a unique output directory for each test cycle results (under /media/output)

based on run index.

- Captures the environment configuration and saves it alongside each test set.
- Sequentially executes selected test scripts (or all if none are specified), storing their logs in per-run files. Ten seconds cool-down time is executed between each test run.
- For latency tests, a temperature/frequency logger is launched in parallel to monitor system thermal state.

Table 5.8 Test execution scrip parameters

Parameter	Value	Description
--help	N/A	Print a help.
--loops=	N	Number of times the full test cycle is repeated. Default is 1.
--tests=	[lat, fai, thr, sca, ctx, star]	Comma-separated list of test keys to execute. If omitted, all are run. Definition in table 5.9.

(source: own)

Table 5.9 Test execution script --tests parameter values

Value	Description
lat	sch_test_latency.sh Runs the latency evaluation test with background stress and cyclic timing.
fai	sch_test_fairness.sh Measures fairness across different nice-level tasks.
thr	sch_test_throughput.sh Runs sysbench with/without load to measure scheduling throughput.
sca	sch_test_scalability.sh Executes sysbench at various thread levels to evaluate scalability.
ctx	sch_test_contextswitch.sh Monitors per-thread context switching under heavy CPU load.
star	sch_test_starvation.sh Detects starvation of low-priority tasks under high-load conditions.

(source: own)

The script expects that all test scripts listed above, along with the temperature logger (5.11), the environment info collector (5.8), and the system init. (5.9) are present in the system's \$PATH variable. This is ensured by the system image definition described in Chapter 6.

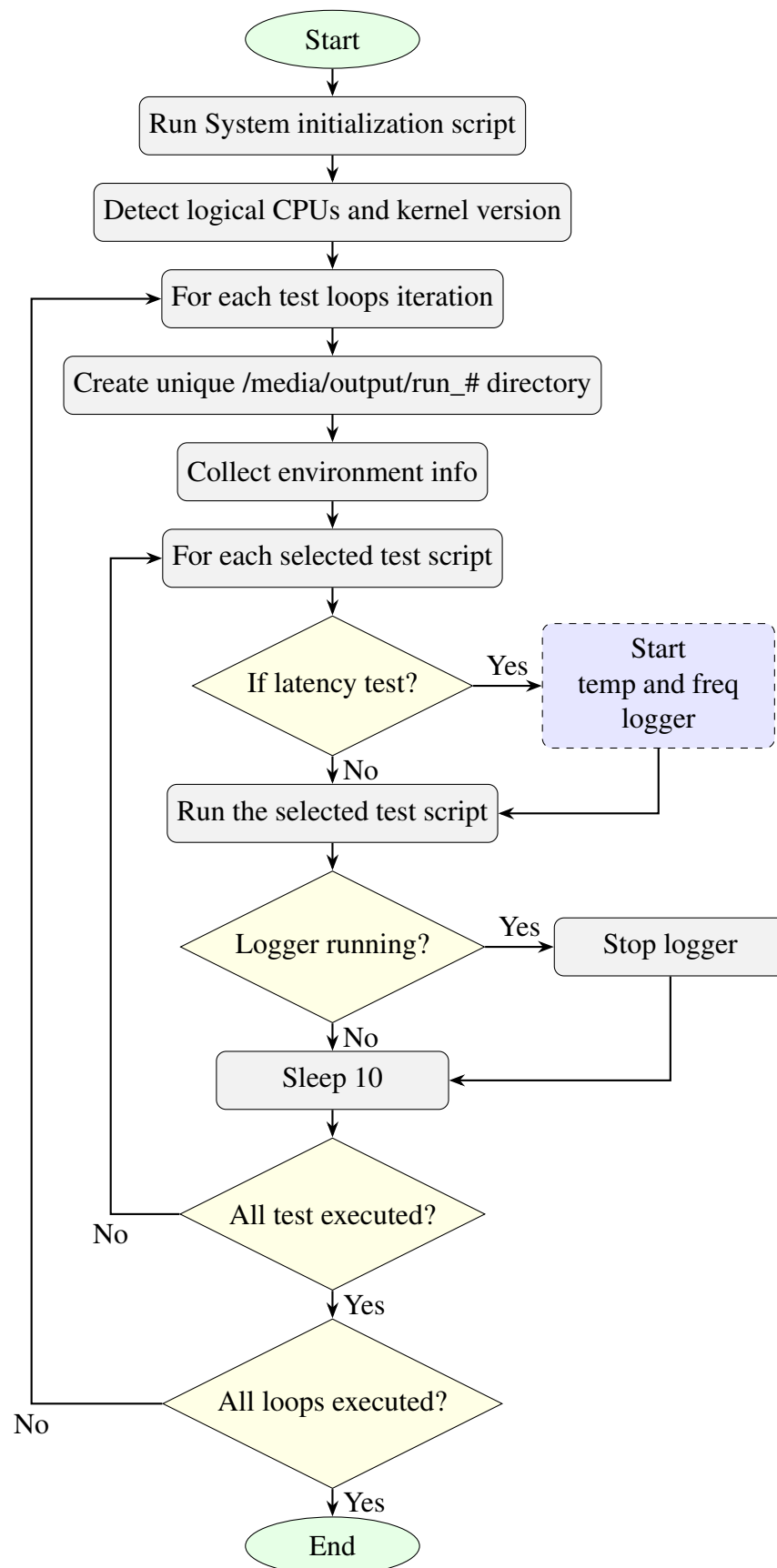


Figure 5.9 Test procedure execution flow chart
(source: own)

5.11 CPU temperature and frequency logger

To monitor thermal behavior and CPU scaling under load, the script continuously logs CPU package temperature and per-core frequencies using native Linux sysfs interfaces [35].

Functionality:

- Detects the temperature input file based on the CPU's monitoring chip (k10temp or coretemp) [36, 37].
- Prints time-stamped logs with temperature in degrees Celsius and current frequency (in kHz) for each logical CPU.
- Supports a configurable logging interval provided as the CLI parameter.
- Prints to standard output (stdout), suitable for redirection by the main test runner script.

Figure 5.10 shows script execution flow. Script is typically used during the latency test to assess if CPU frequency throttling affects scheduling behavior [38].

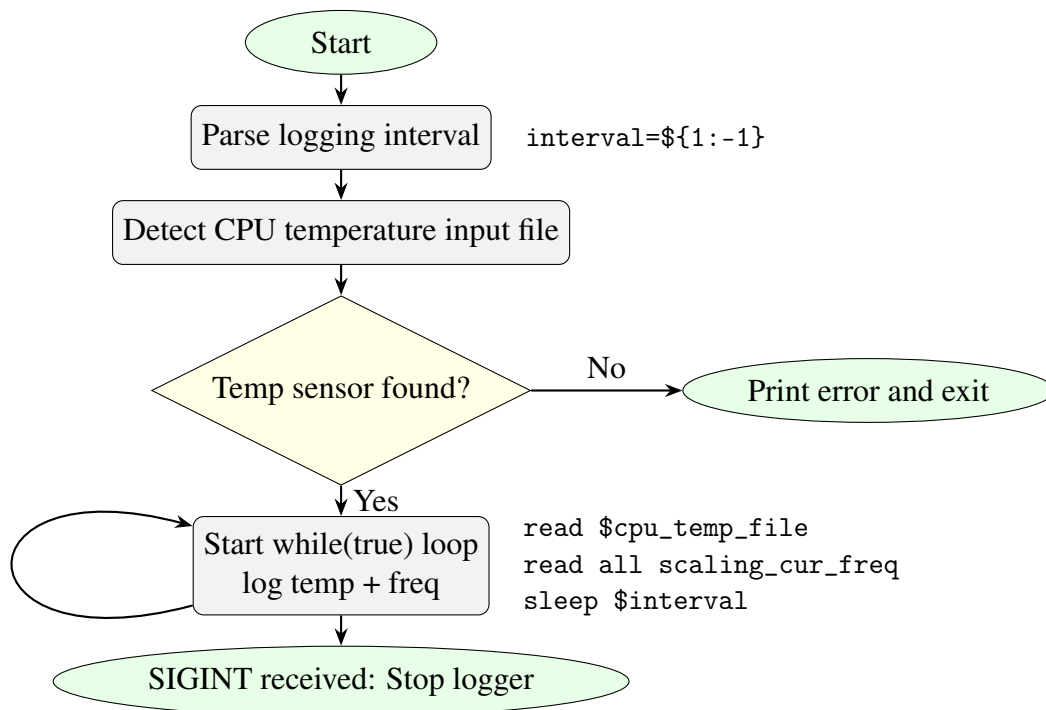


Figure 5.10 Temp and CPU freq logger flow chart
(source: own)

5.11.1 Script output data format

Each output line represents a single time sample and includes:

- Sample number and UNIX timestamp
- CPU temperature in degree of Celsius
- Current frequency (kHz) for each logical core

```

1747100305  LOGGER STARTED
Sample 1, Timestamp: 1747100305, CPU_Temperature: 50C,
  cpu0: 3707714kHz, cpu1: 3434351kHz, cpu10: 400000kHz,
  cpu11: 400000kHz, cpu12: 400000kHz, cpu13: 400000kHz,
  cpu14: 400000kHz, cpu15: 400000kHz, cpu2: 2912370kHz,
  cpu3: 3800000kHz, cpu4: 400000kHz, cpu5: 399873kHz,
  cpu6: 400194kHz, cpu7: 400000kHz, cpu8: 399856kHz,
  cpu9: 417414kHz,
Sample 2, Timestamp: 1747100305, CPU_Temperature: 50C,
  cpu0: 3800000kHz, cpu1: 3800009kHz, cpu10: 2899978kHz,
  cpu11: 2899980kHz, cpu12: 2877630kHz, cpu13: 2900093kHz,
  cpu14: 2900008kHz, cpu15: 2899992kHz, cpu2: 3800009kHz,
  cpu3: 3800009kHz, cpu4: 3800000kHz, cpu5: 3799990kHz,
  cpu6: 3800009kHz, cpu7: 3800000kHz, cpu8: 2881556kHz,
  cpu9: 2851122kHz,
Sample 3, Timestamp: 1747100306, CPU_Temperature: 83C,
  cpu0: 3799990kHz, cpu1: 3800000kHz, cpu10: 2799987kHz,
  cpu11: 2799994kHz, cpu12: 2799994kHz, cpu13: 2800011kHz,
  cpu14: 2799997kHz, cpu15: 2799992kHz, cpu2: 3800000kHz,
  cpu3: 3799990kHz, cpu4: 3799990kHz, cpu5: 3800000kHz,
  cpu6: 3800000kHz, cpu7: 3800000kHz, cpu8: 2800007kHz,
  cpu9: 2800001kHz,
Sample 4, Timestamp: 1747100308, CPU_Temperature: 86C,
  cpu0: 3799990kHz, cpu1: 3800000kHz, cpu10: 2799975kHz,
  cpu11: 2799988kHz, cpu12: 2799983kHz, cpu13: 2800003kHz,
  cpu14: 2799993kHz, cpu15: 2799990kHz, cpu2: 3800009kHz,
  cpu3: 3800000kHz, cpu4: 3799990kHz, cpu5: 3700000kHz,
  cpu6: 3699990kHz, cpu7: 3756932kHz, cpu8: 2799989kHz,
  cpu9: 2799988kHz,
...

```

Listing 5.8 Reduced sample of CPU temp and freq log output (source: own)

6 TEST ENVIRONMENT SETUP AND DEPLOYMENT

To ensure reproducible scheduler testing, a controlled environment is built using `mkosi`. This tool allows for minimal host dependencies and easy deployment of custom Linux images, supporting multiple distributions and kernel variants.

6.1 Installation of `mkosi`

As a mandatory precondition, these packages have to be installed on the host system:

- `python3`, `pipx` : As `mkosi` is Python based tool.
- `systemd-repart`, `systemd-boot` : Mandatory for bootable image and partitioning.
- `mtools` : Package for managing FAT partitions.

Python method of `mkosi` installation:

```
pipx install git+https://github.com/systemd/mkosi.git
```

Listing 6.1 `mkosi` python install command (source: [22])

6.2 Configure system image

Each system image project has to be set up in a dedicated directory. Where the configuration files and directories are set up. The main definition of the image is `mkosi.conf` configuration file, which is located in the root of the project directory. If a parameter is not configured explicitly, `mkosi` reuses the definition from the host system, for example, the list of package sources for the package manager [22].

6.2.1 mkosi project directory

Listing 6.2 shows the content of the test environment system image root directory, which is parsed during the mkosi build process execution [22].

Definition for test environment system image consists of two mkosi configuration directories:

- `mkosi.extra`: Content of this directory is copied into system image during the mkosi build. Details about directory content in 6.2.3.
- `mkosi.repart`: This directory contains `systemd-repart` configuration files, which defines the partitions, they are passed to `systemd-repart` during build of system image. Details of partition configuration in 6.2.4.

```
mkosi_image/  
|-- mkosi.extra/  
|   |-- usr/  
|   |   |-- bin/  
|   |   |   |-- sch_test_env_init.sh  
|   |   |   |-- sch_test_env_info.sh  
|   |   |   |-- sch_test_fairness.sh  
|   |   |   |-- sch_test_latency.sh  
|   |   |   |-- sch_test_contextswitch.sh  
|   |   |   |-- sch_test_starvation.sh  
|   |   |   |-- sch_test_scalability.sh  
|   |   |   |-- sch_test_throughput.sh  
|   |   |   |-- sch_test_exec.sh  
|   |   |   |-- sch_test_cpu_temp.sh  
|-- mkosi.repart/  
|   |-- 00-esp.conf  
|   |-- 05-bios.conf  
|   |-- 10-root.conf  
|   |-- 60-output.conf  
|-- mkosi.conf
```

Listing 6.2 mkosi project directory (source: own)

6.2.2 mkosi configuration file

Configuration file for mkosi `mkosi.conf` is structured into configuration sections in which configuration parameters are defined [22]. Definitions for CFS and EEVDF test environment system images are listed in Listings 6.3 and 6.4.

Both configurations consist of the following sections and parameters [22]:

- **[Distribution]:**

Contains parameters for system distribution configuration.

- **Distribution:**

Define target distribution for the image. Value from the mkosi supported list.

- **Release:**

Definition of distribution release. Value from the mkosi supported list.

- **[Output]:**

Configuration section for output image definition.

- **Format:**

Define a image format type to generate. Set up to "Disk" value which creates a block device image with a GPT partition table.

- **Output:**

Configures a name of the output image file.

- **[Content]:**

Section for definition and configuration of the content of the output image.

- **Bootable:**

Enables or disables generation of a bootable image. If enabled, mkosi will install an EFI bootloader.

- **Packages:**

Comma-separated list of packages which will be installed in the image.

- RootPassword:

Configure a password for the root user.

- **[Validation]:**

Section for validation and security configuration parameters.

- Checksum:

Generate a SHA256 checksum file of all created artifacts after the build process.

- **[Include]:**

Section for extra configuration.

- Include:

mkosi-vm configuration value used for virtual machine compatibility for easier test of the output system image.

```
1 [Distribution]
2 Distribution=debian
3 Release=bookworm
4 [Output]
5 Format=disk
6 Output=image
7 [Content]
8 Bootable=yes
9 Packages=systemd,systemd-boot,systemd-sysv,udev,dbus,sysbench
10    linux-image-6.1.0-32-amd64
11    openssh-client
12    vim
13    rt-tests
14    linux-perf
15    sysstat
16    hwloc
17    procps
18    stress-ng
19
20 RootPassword=password
21 [Validation]
22 Checksum=yes
23 [Include]
24 Include=mkosi-vm
```

Listing 6.3 mkosi.conf of CFS image (source: own)

```
1 [Distribution]
2 Distribution=debian
3 Release=trixie
4 [Output]
5 Format=disk
6 Output=image
7 [Content]
8 Bootable=yes
9 Packages=systemd,systemd-boot,systemd-sysv,udev,dbus,sysbench
10     linux-image-6.12.27-amd64
11     openssh-client
12     vim
13     rt-tests
14     linux-perf
15     sysstat
16     hwloc
17     procps
18     stress-ng
19
20 RootPassword=password
21 [Validation]
22 Checksum=yes
23 [Include]
24 Include=mkosi-vm
```

Listing 6.4 mkosi.conf of EEVDF image (source: own)

6.2.3 mkosi.extra directory

Content of the `mkosi.extra` directory as is mentioned in 6.2.1 is copied into the system image [22].

In the path `mkosi.extra/usr/bin`, all mandatory scripts for test execution have to be present, as it is mentioned in Listing 6.2.

All referenced scripts are available in Appendix 2.

Copying test scripts into path `/usr/bin` will ensure their availability in the system `PATH` variable for easier execution.

6.2.4 mkosi.repart directory

As mentioned in 6.2.1 `mkosi.repart` directory is used as the source of configuration for the `systemd-repart` partition manager. In total, four partition configuration files are included for the test environment system image [22, 39].

- `00-esp.conf`: Partition for EFI bootable image.
- `05-bios.conf`: Grub BIOS bootable image.
- `10-root.conf`: Root system partition.
- `60-output.conf`: User output partition for storage of test results.

`Esp`, `bios`, and `root` partition are a reuse of the default `mkosi` configuration and are listed in Appendix 1.

Output partition is added due to the necessity of transferring test output data from the test environment system for further processing. Complete configuration file is listed in the Listing 6.5.

```
1 [Partition]
2 Type=usr
3 Format=vfat
4 SizeMinBytes=1G
5 SizeMaxBytes=2G
6 Label=output
```

Listing 6.5 `60-output.conf` (source: own)

6.3 Build and deployment of the system image

After the configuration and ensuring the package sources are available, execution of mkosi image build is a fairly simple process. There are several possibilities, simple build execution Listing 6.6 or build with direct execution of the output image in QEMU virtual machine Listing 6.7.

In both cases, parameter `--force` is used to ensure the execution of the `mkosi clean` command to remove the output of the previous build if necessary [22].

The deployment of the system image to the USB stick drive is done by the Linux DD tool for copying raw data. Set up of the command is shown by Listing 6.8, usage of `sudo` depends on the setup of the host system. During the definition of the target disk `of=/dev/xxx` special attention should be paid; an error in the definition could lead to data loss [40].

```
mkosi --force
```

Listing 6.6 mkosi build execution (source: [22])

```
mkosi --qemu-smp 2 qemu --force
```

Listing 6.7 mkosi build with direct run in virtual machine (source: [22])

```
sudo dd if=image of=/dev/XXX conv=fsync oflag=direct  
status=progress bs=4M
```

Listing 6.8 Copy of image to USB stick drive by dd (source: [40])

Both mkosi configurations are available in the Appendix 2 in directory `mkosi_images`.

7 TESTING ON DIFFERENT HARDWARE PLATFORMS

The data analyzed in this chapter is available in the Appendix 2 in directory `test_results`.

7.1 Intel hybrid architecture (i7-1260P)

The 12th Generation Intel® Core™ i7-1260P processor, part of the Alder Lake family, introduces a hybrid architecture that combines performance and efficiency cores to optimize computing tasks. This processor features a total of 12 cores, comprising 4 high-performance cores and 8 energy-efficient cores, supporting up to 16 threads. This configuration allows for improved multitasking and responsiveness, adapting to various computing needs.

Furthermore, the i7-1260P includes 18 MB of Intel® Smart Cache [41].

7.1.1 DUT HW specification

Device under test is Lenovo ThinkPad T14 Gen 3 laptop with Intel i7-1260P processor (Type number: 21AH-0094CK), detailed specifications relevant for process scheduler testing are listed in table 7.1 and CPU topology is shown by Figure 7.1 [42].

Table 7.1 DUT: i7-1260P - System Hardware Specification

Component	Specification
CPU Model	12th Gen Intel® Core™ i7-1260P
Architecture	x86_64
CPU Cores (Physical)	12 (4 Performance cores + 8 Efficient cores)
Threads per Core	2 (Performance cores), 1 (Efficient cores)
Total Logical CPUs	16
Base Frequency	400 MHz
Max Turbo Frequency	4700 MHz
L1 Cache (Data + Instruction)	448 KiB (Data), 640 KiB (Instruction)
L2 Cache	9 MiB
L3 Cache	18 MiB
NUMA Nodes	1 (CPUs 0-15)
RAM Installed	46 GiB
Swap Space	0 B (no swap configured)

(source: [17, 31, 32, 41])

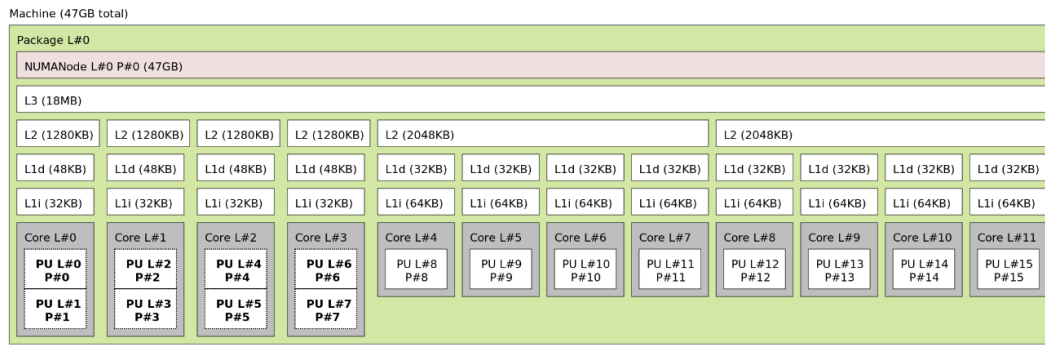


Figure 7.1 i7-1260P CPU topology

(source: [32])

7.1.2 CFS latency test result

Results of latency test (5.2) executed on the i7-1260P CPU using the CFS scheduler. The results show the minimum, maximum, and average latencies recorded per thread running on the CPU. Table 7.2 presents detailed per-thread latency statistics, while Table 7.3 summarizes the overall latency measurements across all threads. Figure 7.2 illustrates the average latency of the CPU thread throughout the defined cycles.

Table 7.2 i7-1260P CFS latency per CPU

Thread	Min [ns]	Max [ns]	Avg [ns]	Thread	Min [ns]	Max [ns]	Avg [ns]
T0	1766	2763180	9277.0	T8	2359	2609037	12303.0
T1	1774	2622423	9879.0	T9	2453	2652532	11821.0
T2	1771	2707716	8960.0	T10	2492	2702922	11867.0
T3	1841	2689493	9221.0	T11	2351	2663055	11902.0
T4	1863	2636852	9060.0	T12	2995	2849568	11786.0
T5	1956	2680034	9243.0	T13	2285	2666573	11982.0
T6	1893	2640146	9257.0	T14	2732	2676289	11965.0
T7	1866	2665035	9056.0	T15	2406	2658548	12048.0

(source: own)

This latency measurement reveals a significant performance disparity between performance (T0 to T7) and efficiency (T8 to T15) cores, which correlates with observed differences in CPU temperature and operating frequency. Higher average temperatures and lower CPU core frequencies, particularly on efficiency cores, suggest thermal throttling may be influencing scheduler behavior and execution latency. This indicates that thermal conditions and dynamic frequency scaling have a substantial impact on real-time latency measurements clearly visible in Figure 7.3. Summary of measured metrics is listed in Table 7.4.

Table 7.3 i7-1260P CFS overall latency summary

Metric	Value
Min [ns]	1766
Max [ns]	2849568
Avg [ns] (all)	10601.69
Avg [ns] (T0–T7)	9244.13
Avg [ns] (T8–T15)	11971.75
Iterations (C)	200000

(source: own)

Table 7.4 i7-1260P CFS frequency and temperature summary

Metric	Value
Temperature Min [°C]	51
Temperature Max [°C]	88
Temperature Avg [°C]	60.84
CPU0–7 Avg Freq [GHz]	1.57
CPU8–15 Avg Freq [GHz]	1.30

(source: own)

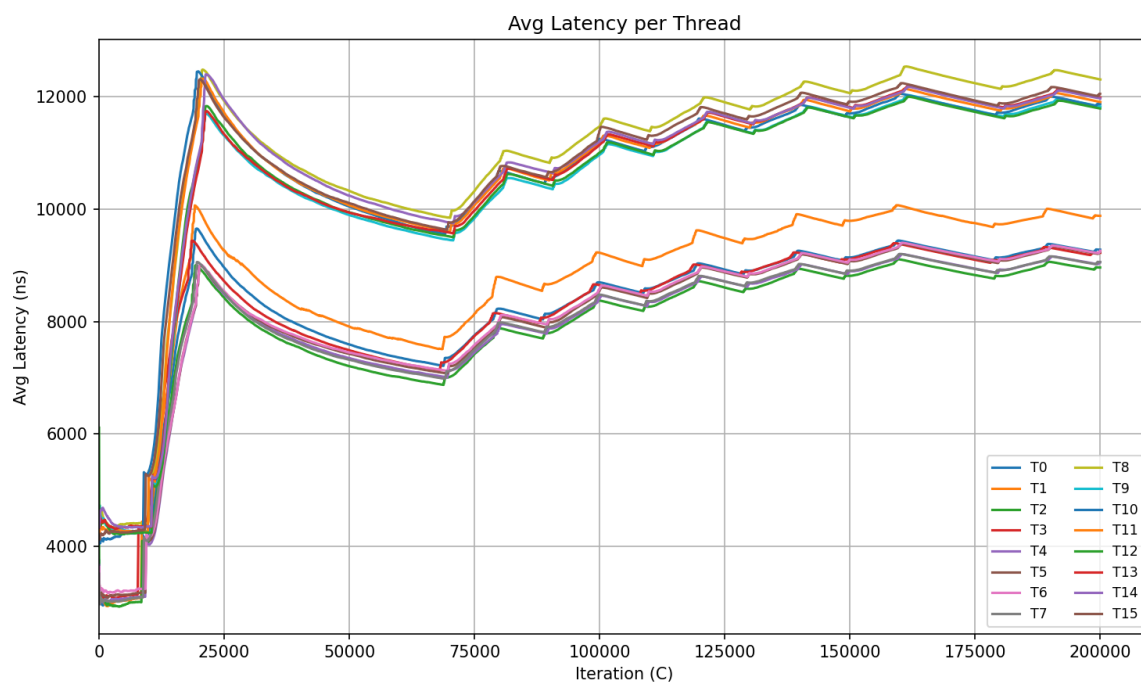


Figure 7.2 i7-1260P CFS average latency per CPU thread

(source: own)

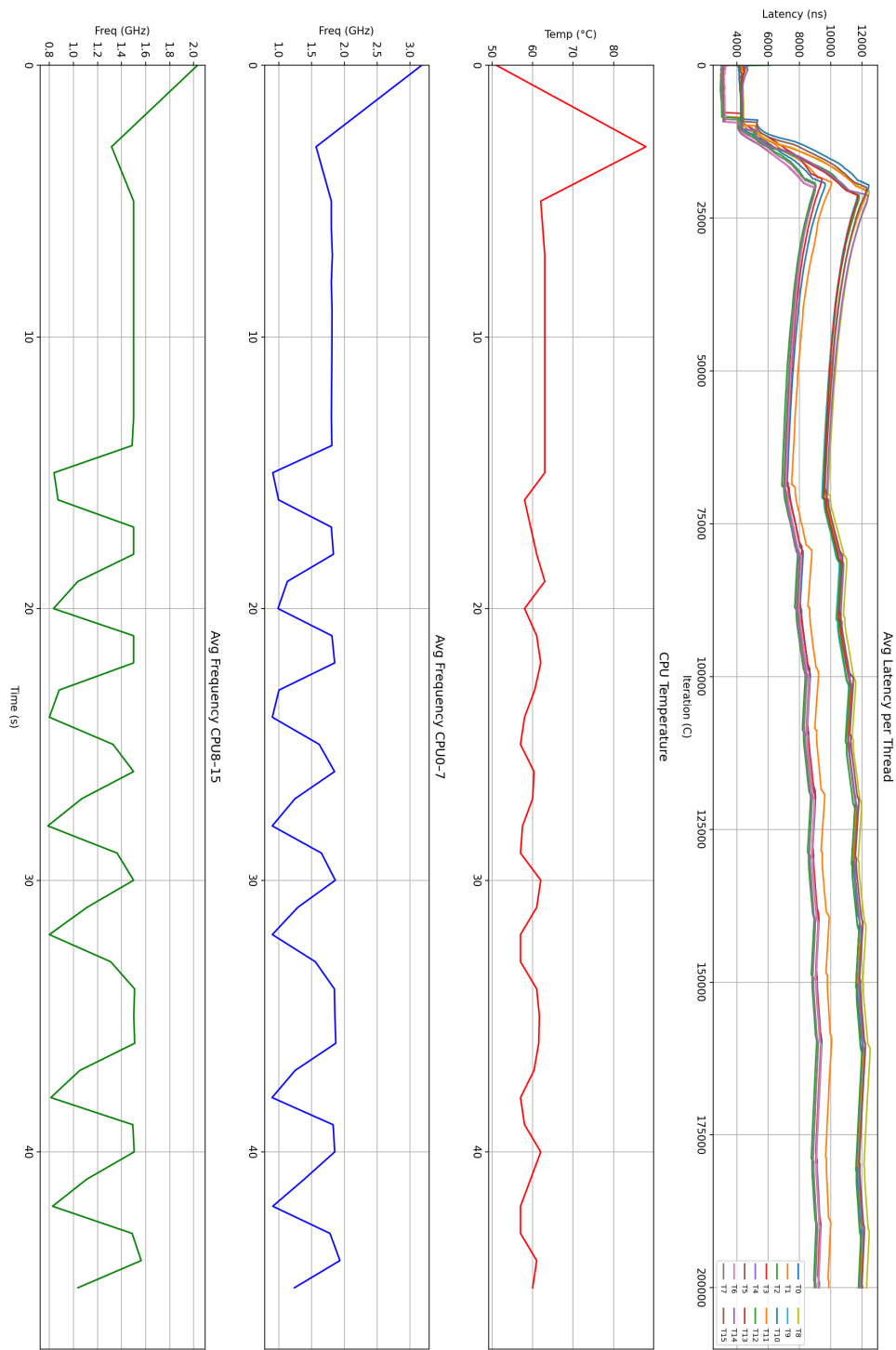


Figure 7.3 i7-1260P CFS average latency with CPU temp and freq
(source: own)

7.1.3 EEVDF latency test result

Results of the latency test (5.2) executed on the i7-1260P CPU using the EEVDF scheduler. Measured metrics are the same as with the previous test for CFS (7.1.2). Table 7.5 displays the detailed per-thread latency statistics, while Table 7.6 summarizes the overall latency measurements. Figure 7.4 illustrates the average latency of CPU threads through the defined cycles.

Table 7.5 i7-1260P EEVDF latency per CPU (single column)

Thread	Min [ns]	Max [ns]	Avg [ns]
T0	1755	2676066	11044.0
T1	2059	2679264	10057.0
T2	2093	2691642	9748.0
T3	2044	2701792	9806.0
T4	2070	2699323	9625.0
T5	1976	2698333	9296.0
T6	1933	2684540	9467.0
T7	1977	2705328	9605.0
T8	3382	2652948	13060.0
T9	3498	2656663	13176.0
T10	3229	2664540	12672.0
T11	3295	2649320	12311.0
T12	3503	2682061	13098.0
T13	3430	2698212	12985.0
T14	3187	2706936	12924.0
T15	3423	2666125	12921.0

(source: own)

Table 7.6 i7-1260P EEVDF overall latency summary

Metric	Value
Min [ns]	1755
Max [ns]	2706936
Avg [ns] (all)	11362.19
Avg [ns] (T0–T7)	9843.00
Avg [ns] (T8–T15)	12943.38
Iterations (C)	200000

(source: own)

A similar thermal effect as observed in the previous CFS test (7.1.2) is also visible under EEVDF, with latency behavior showing sensitivity to the core architecture. Figure 7.5 shows CPU thread latency in relation to CPU temperature and frequency. Data summary is listed in Table 7.7.

Table 7.7 i7-1260P EEVDF frequency and temperature summary

Metric	Value
Temperature Min [°C]	51
Temperature Max [°C]	85
Temperature Avg [°C]	60.33
CPU0–7 Avg Freq [GHz]	1.53
CPU8–15 Avg Freq [GHz]	1.27

(source: own)

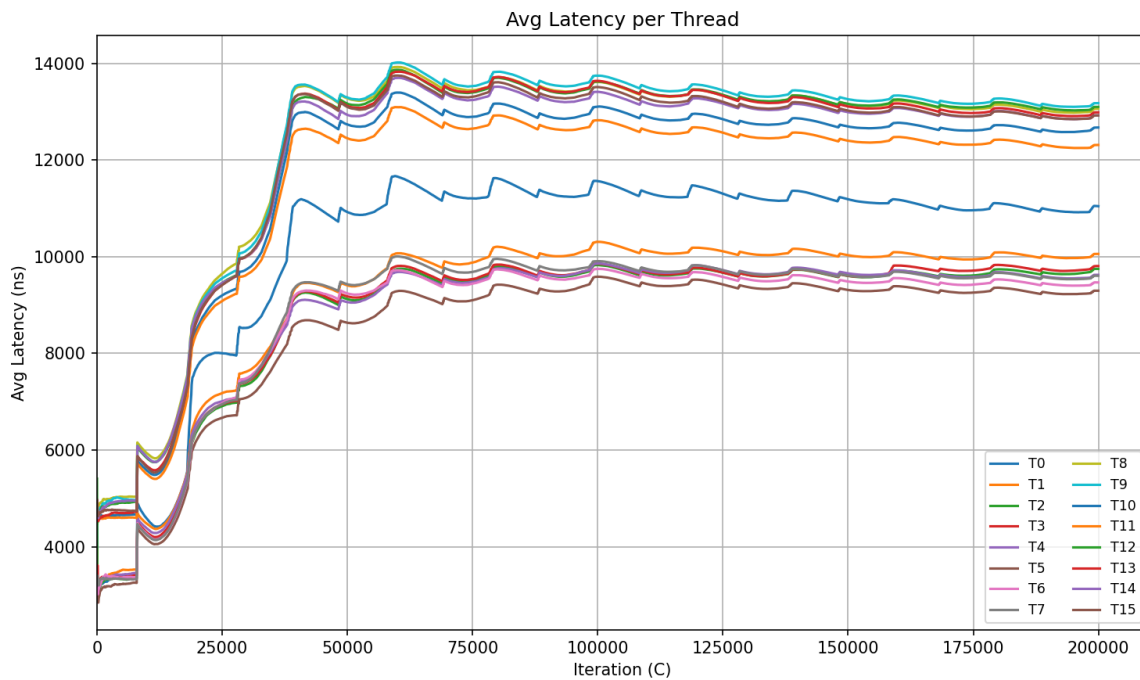


Figure 7.4 i7-1260P EEVDF average latency per CPU thread

(source: own)

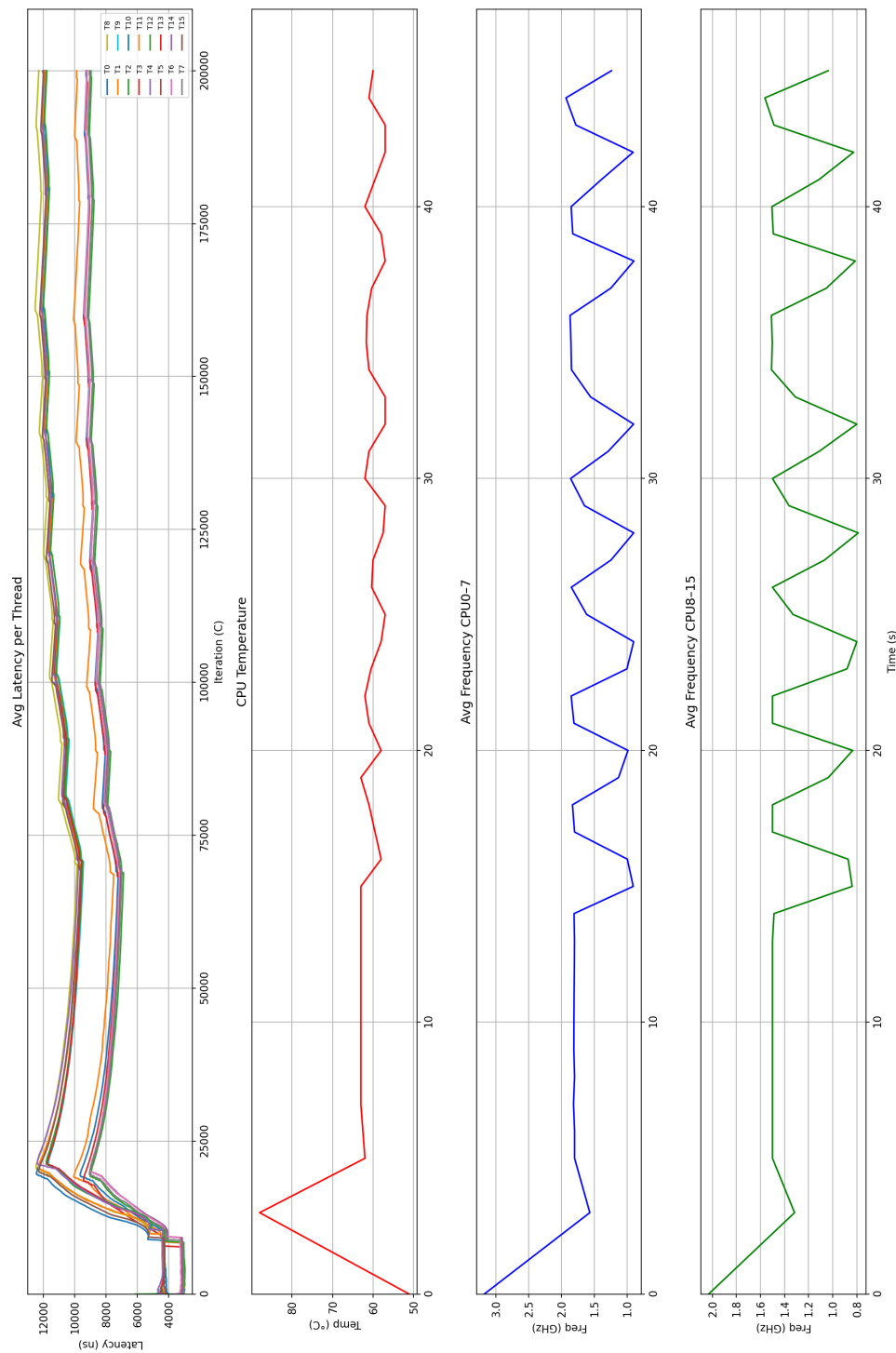


Figure 7.5 i7-1260P EEVDF average latency with CPU temp and freq
(source: own)

7.1.4 CFS fairness test result

The fairness test described in 5.3 evaluates how evenly the CFS scheduler distributes CPU time among processes grouped by different `nice` values.

Figure 7.6 shows the normalized CPU utilization for each group. This metric represents how much CPU capacity was consumed by each group over the test duration. The scheduler grants more CPU time to higher-priority processes, which is reflected in the results: the `nice = -5` group reached over 81% utilization, while `nice = 18` remained under 28%.

Figure 7.7 provides a breakdown using a box plot. In this visualization, the central box indicates the interquartile range (IQR), showing the middle 50% of CPU time values. The median is marked by a horizontal line within the box. Whiskers extend to 1.5 times the IQR, and any points beyond are shown as outliers. This helps assess both the consistency and spread of CPU time allocation within each group. Notably, the `nice = 10` and `nice = 18` groups exhibit narrower boxes and smaller standard deviations, confirming tighter clustering of values, as also shown in Table 7.9 [43].

Table 7.8 provides an overview of key fairness metrics. These include average CPU time received per process, average wall time (i.e., the total elapsed time during process execution), CPU efficiency, and CPU utilization. CPU efficiency quantifies how much of a process's runtime was spent actively executing on the CPU, computed using Equation 7.1. CPU utilization is calculated per group to reflect the fraction of total available CPU time used, according to Equation 7.2.

Each group's performance is detailed individually in Table 7.9, listing not only means and medians, but also total wall and CPU times, standard deviation of CPU usage, and the measured lifetime span. These values contextualize the bar and box plots and highlight group-level behavior.

It is important to note that total wall time values reported in Table 7.9 do not represent the runtime of a single process. Instead, the total wall time is calculated as the sum of individual wall times across all processes within a given `nice` group. Since processes are scheduled

and executed concurrently across multiple logical CPUs, wall time should not be interpreted as a linear or sequential time span. Rather, it reflects the aggregate runtime footprint of the group, which grows proportionally with the number of parallel processes and their individual execution durations.

$$\text{CPU Efficiency (\%)} = \left(\frac{\text{Avg CPU Time}}{\text{Avg Wall Time}} \right) \times 100 \quad (7.1)$$

$$\text{CPU Utilization (\%)} = \left(\frac{\text{Total CPU Time}}{\text{Lifetime Span} \times \#\text{Logical CPUs}} \right) \times 100 \quad (7.2)$$

Table 7.8 i7-1260P CFS fairness summary

Nice	Avg CPU Time [s]	Avg Wall Time [s]	CPU Efficiency [%]	CPU Utilization [%]	Processes
−5	0.49	5.17	9.58	81.86	192
5	0.71	15.91	4.44	44.22	192
10	0.85	26.82	3.16	35.63	192
18	0.83	34.92	2.37	27.94	192

(source: own)

Table 7.9 i7-1260P CFS fairness detailed metrics

Metric	nice = −5	nice = 5	nice = 10	nice = 18
Processes	192	192	192	192
Avg CPU Time [s]	0.49	0.71	0.85	0.83
Median CPU Time [s]	0.48	0.73	0.84	0.83
Std Dev CPU Time [s]	0.09	0.07	0.04	0.04
Avg Wall Time [s]	5.17	15.91	26.82	34.92
Total Wall Time [s]	991.97	3053.78	5149.80	6705.08
Total CPU Time [s]	95.03	135.68	162.50	158.97
Lifetime Span [s]	7.26	19.18	28.50	35.56
CPU Efficiency [%]	9.58	4.44	3.16	2.37
CPU Utilization [%]	81.86	44.22	35.63	27.94

(source: own)

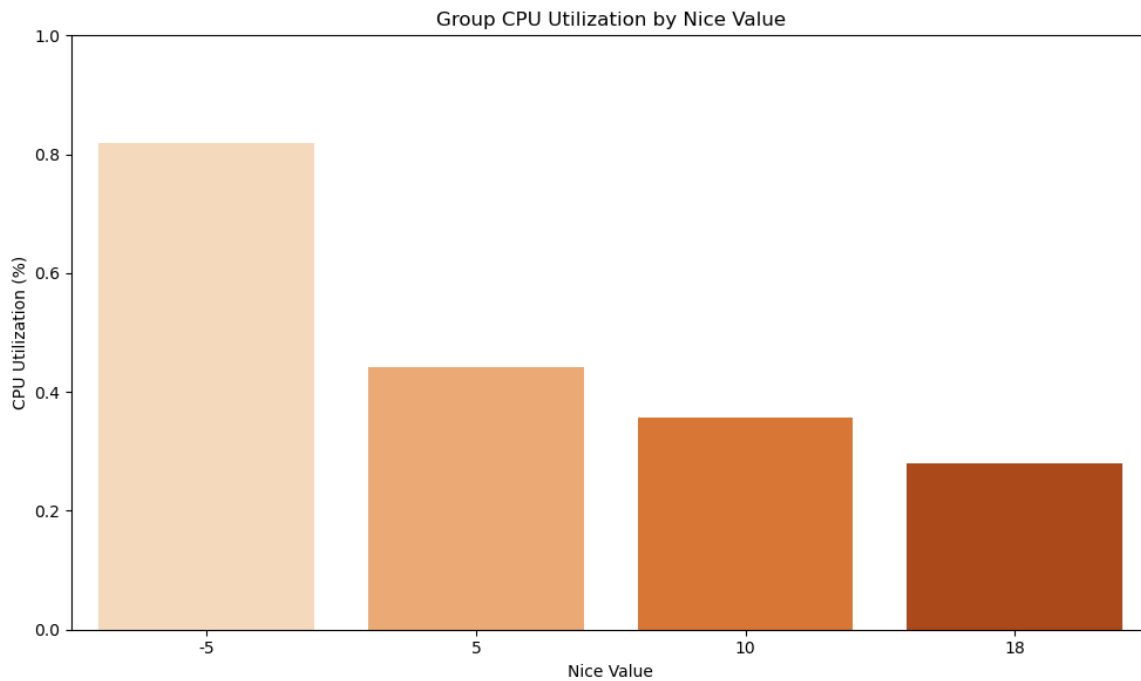


Figure 7.6 i7-1260P CFS fairness CPU utilization
(source: own)

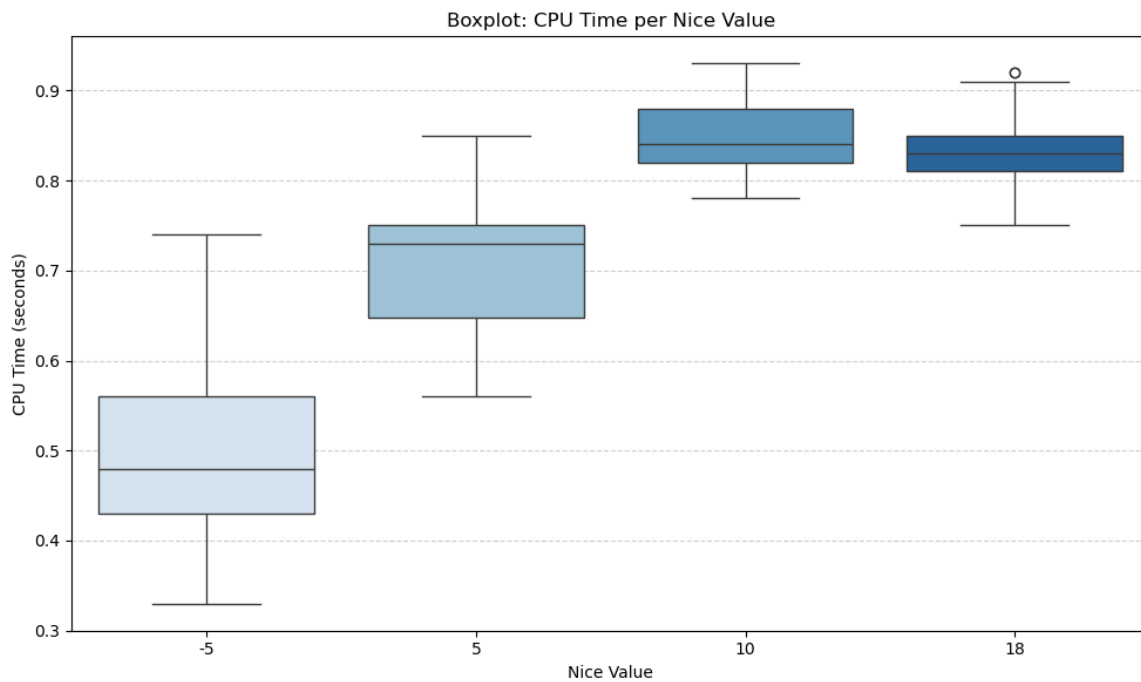


Figure 7.7 i7-1260P CFS fairness CPU time
(source: own)

7.1.5 EEVDF fairness test result

EEVDF fairness scheduler test follows the same evaluation described in Section 5.3 and results interpretation used for CFS analysis 7.1.4). Figure 7.8 presents the CPU utilization distribution across nice groups, while Figure 7.9 visualizes the distribution of CPU time using a box plot.

Table 7.10 summarizes the aggregate statistics across all groups. Individual group-level statistics are provided in Table 7.11.

Table 7.10 i7-1260P EEVDF fairness test summary

Nice	Avg CPU Time [s]	Avg Wall Time [s]	CPU Efficiency [%]	CPU Utilization [%]	Processes
−5	0.45	1.43	31.64	158.45	192
5	0.48	9.99	4.81	50.40	192
10	0.50	15.13	3.28	37.27	192
18	0.49	20.30	2.43	28.63	192

(source: own)

Table 7.11 i7-1260P EEVDF fairness detailed metrics

Metric	nice = −5	nice = 5	nice = 10	nice = 18
Processes	192	192	192	192
Avg CPU Time [s]	0.45	0.48	0.50	0.49
Median CPU Time [s]	0.47	0.48	0.50	0.50
Std Dev CPU Time [s]	0.06	0.05	0.05	0.03
Avg Wall Time [s]	1.43	9.99	15.13	20.30
Total Wall Time [s]	274.14	1918.58	2904.65	3898.09
Total CPU Time [s]	86.73	92.33	95.22	94.69
Lifetime Span [s]	3.42	11.45	15.97	20.67
CPU Efficiency [%]	31.64	4.81	3.28	2.43
CPU Utilization [%]	158.45	50.40	37.27	28.63

(source: own)

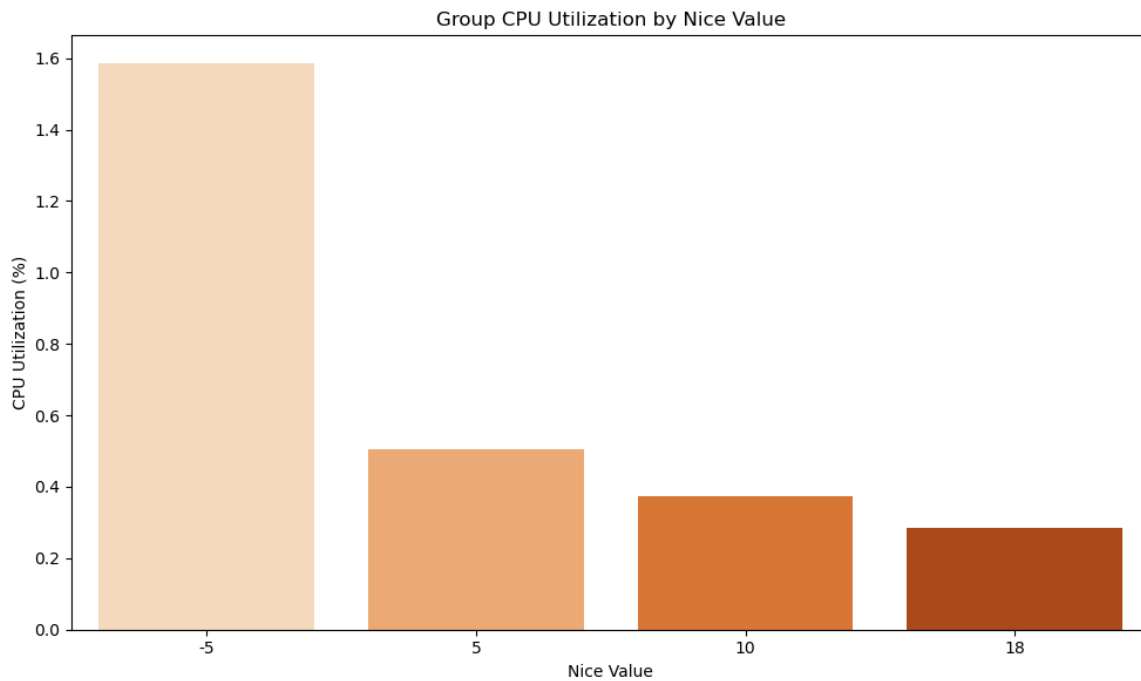


Figure 7.8 i7-1260P EEVDF fairness CPU utilization
(source: own)

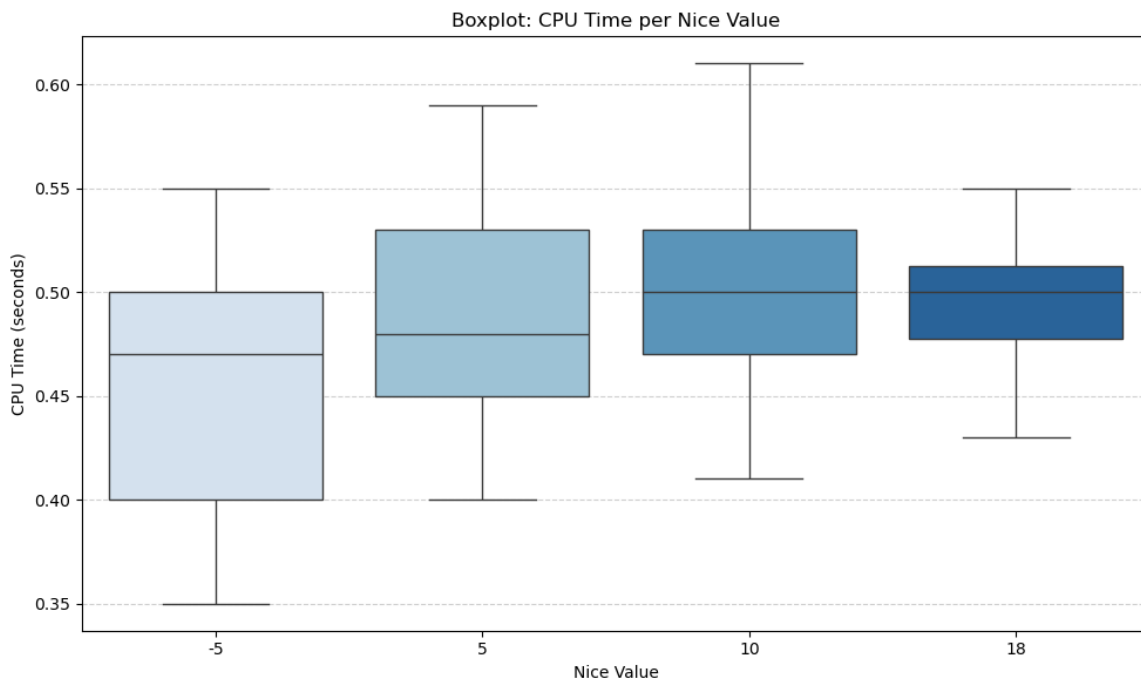


Figure 7.9 i7-1260P EEVDF fairness CPU time
(source: own)

7.1.6 CFS throughput test result

CFS throughput testing executed as described in 5.4. Figure 7.10 illustrates the total number of events completed in each scenario. The results show a clear impact of background load on throughput, particularly evident in longer test durations. Table 7.12 presents the detailed performance metrics for each scenario.

Table 7.12 i7-1260P CFS throughput results for all scenarios

Metric	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Description	30 s, no background	30 s, with background	100 s, no background	100 s, with background
Total Time [s]	30.00	30.02	100.00	100.01
Total Events	895,439	358,066	2,299,029	1,127,333
Events per Second	29,845.69	11,928.21	22,989.78	11,272.15

(source: own)

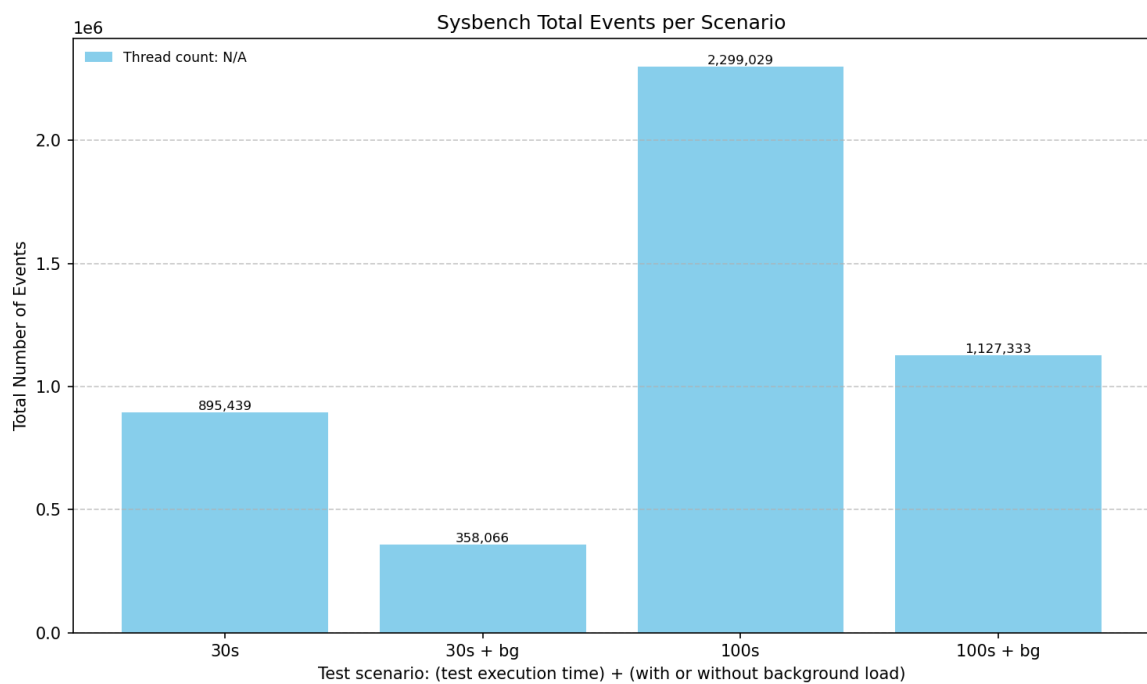


Figure 7.10 i7-1260P CFS throughput

(source: own)

7.1.7 EEVDF throughput test result

EEVDF throughput testing follows the same methodology as defined in Section 5.4. Figure 7.11 displays the number of completed events in each scenario. The results show a noticeable drop in throughput when background load is introduced. Detailed metrics for each test scenario are presented in Tables 7.13.

Table 7.13 i7-1260P EEVDF throughput results for all scenarios

Metric	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Description	30 s, no background	30 s, with background	100 s, no background	100 s, with background
Total Time [s]	30.00	30.00	100.00	100.00
Total Events	642,583	292,290	2,051,442	995,912
Events per Second	21,417.15	9,740.63	20,513.96	9,958.50

(source: own)

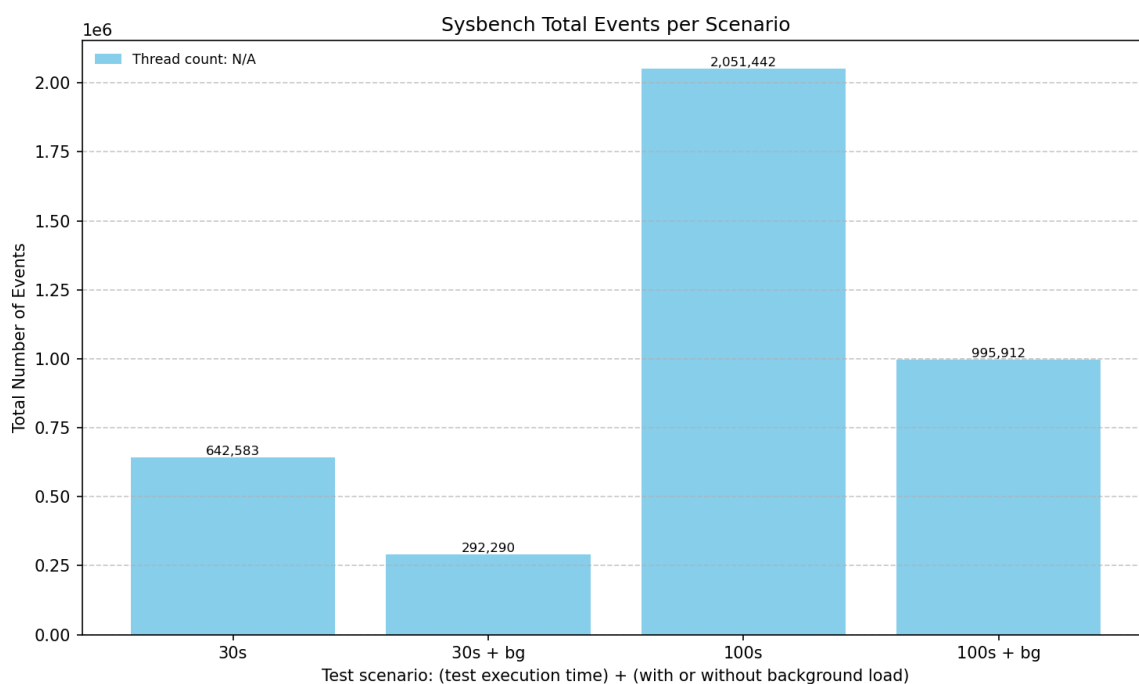


Figure 7.11 i7-1260P EEVDF throughput

(source: own)

7.1.8 CFS context switching test result

The context switching test for the i7-1260P system under the CFS scheduler was conducted following the methodology described in 5.5. Figure 7.12 visualizes the amount of context involuntary switches per second by each thread during the test period.

Table 7.14 presents the per-thread statistics for involuntary context switches per second (nvcschw/s), including total and average values. The cumulative metrics are summarized in Table 7.15.

Table 7.14 i7-1260P CFS per-thread context switching

Thread	Total [cswch/s]	Average [cswch/s]
T0	42.97	2.86
T1	95.97	3.84
T2	239.74	10.42
T3	131.97	6.60
T4	102.85	4.67
T5	287.80	9.59
T6	84.97	4.47
T7	181.88	6.06
T8	25.88	2.16
T9	90.97	5.05
T10	466.48	15.55
T11	22.97	2.09
T12	91.97	5.41
T13	41.97	2.62
T14	565.91	18.86
T15	63.80	3.54

(source: own)

Table 7.15 i7-1260P CFS context switching summary

Metric	Value
Total [nvcschw/s]	2538.10
Average per Thread [nvcschw/s]	6.49
Peak Rate (Max per-second nvcschw)	93.00
Std. Dev. of Per-Thread Averages	4.85
Thread Avg Max/Min Ratio	9.03

(source: own)

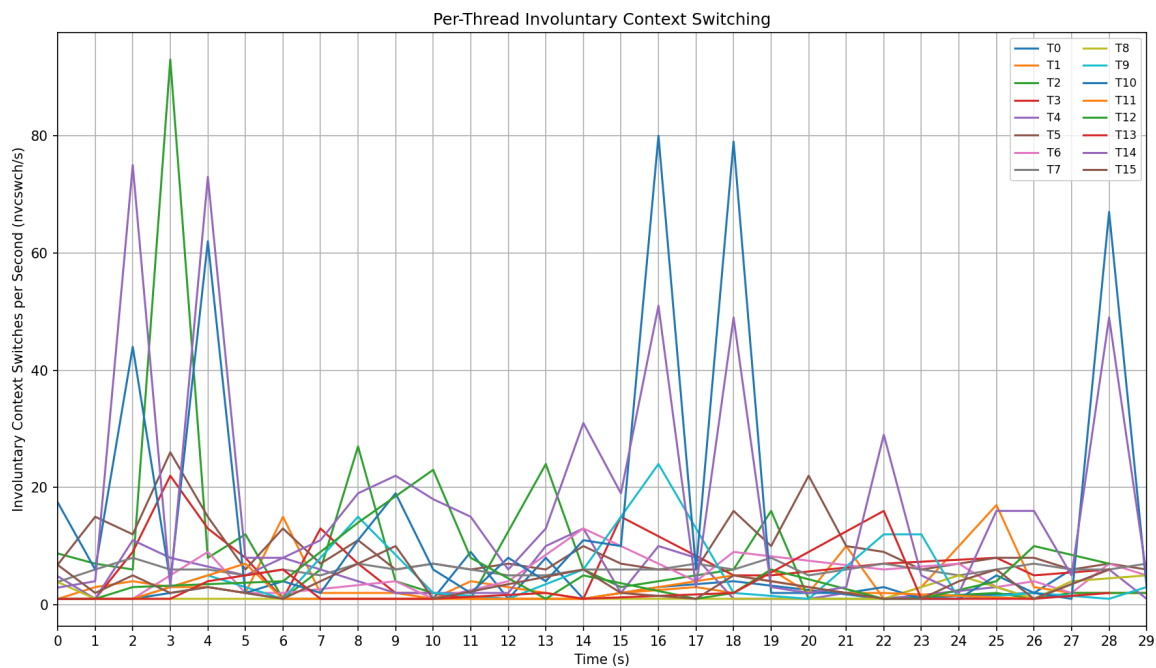


Figure 7.12 i7-1260P CFS context switching
(source: own)

7.1.9 EEVDF context switching test result

Figure 7.13 visualizes the amount of involuntary context switches per second. Table 7.17 presents the per-thread statistics. The cumulative metrics are summarized in Table 7.16.

Table 7.16 i7-1260P EEVDF context switching summary

Metric	Value
Total [nvcswh/s]	2256.93
Average per Thread [nvcswh/s]	5.94
Peak Rate (Max per-second nvcswh)	99.05
Std. Dev. of Per-Thread Averages	3.87
Thread Avg Max/Min Ratio	7.02

(source: own)

Table 7.17 i7-1260P EEVDF per-thread context switching

Thread	Total [cswch/s]	Average [cswch/s]
T0	162.62	7.39
T1	89.55	3.58
T2	25.93	2.16
T3	36.97	2.17
T4	33.92	2.42
T5	37.97	1.81
T6	24.93	2.08
T7	253.86	9.07
T8	212.82	12.52
T9	119.79	3.99
T10	150.71	5.58
T11	290.42	10.01
T12	342.95	12.70
T13	43.93	2.93
T14	292.05	9.74
T15	138.51	6.93

(source: own)

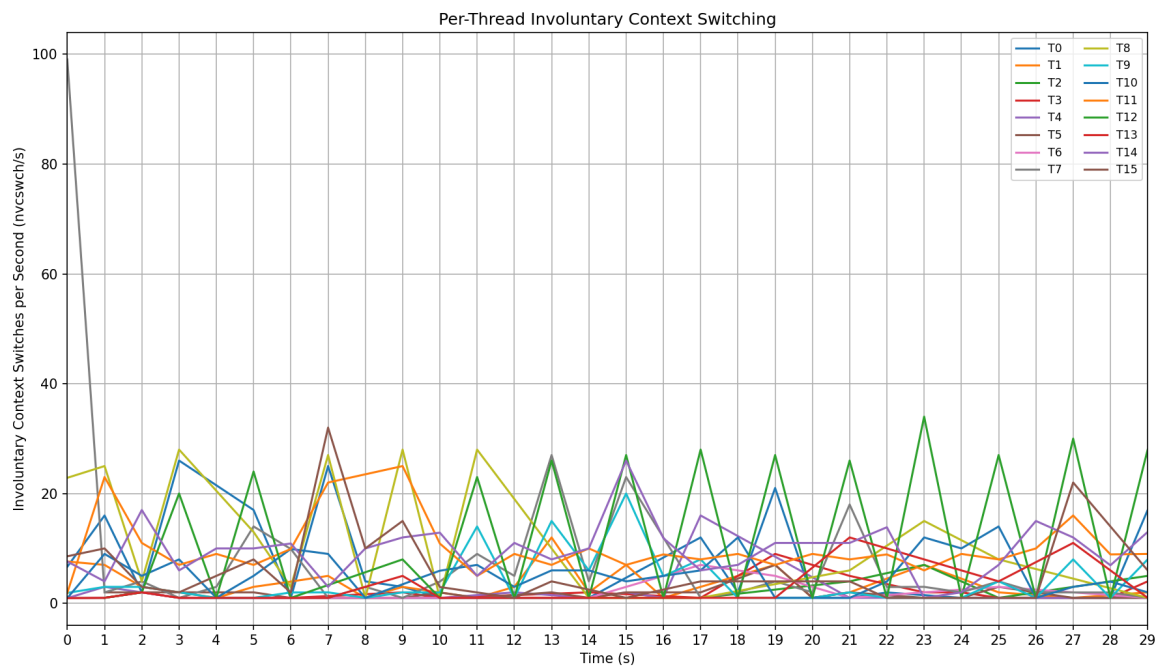


Figure 7.13 i7-1260P EEVDF context switching

(source: own)

7.1.10 CFS scalability test result

This section presents the results of the scalability test as described in 5.6, executed on the i7-1260P CPU using the CFS scheduler. The analysis evaluates how the throughput of the system measured in Events Per Second (EPS) evolves as the number of threads increases. Three core metrics are assessed: EPS Gain, Efficiency, and Efficiency Drop.

EPS Gain evaluates how performance scales with increasing thread counts, we define the events per second gain as shown in Equation 7.3. This metric expresses the percentage increase in throughput (events per second) compared to the previous thread level. A high EPS gain suggests effective parallel scaling.

$$\text{EPS Gain (\%)} = \left(\frac{\text{EPS}_{\text{curr}} - \text{EPS}_{\text{prev}}}{\text{EPS}_{\text{prev}}} \right) \times 100 \quad (7.3)$$

Efficiency measures the amount of throughput achieved per thread and is computed using Equation 7.4. It is calculated as the ratio of events per second to the number of threads used. This metric provides insight into how effectively the system utilizes each individual thread as parallelism increases.

$$\text{Efficiency} = \frac{\text{EPS}}{\text{Threads}} \quad (7.4)$$

Efficiency drop determines diminishing returns, we compute the efficiency drop between thread levels, defined in Equation 7.5. It quantifies the percentage decrease in efficiency compared to the previous level, indicating how much less performance is gained per next thread level. A growing drop highlights scalability limitations.

$$\text{Efficiency Drop (\%)} = \left(\frac{\text{Eff}_{\text{prev}} - \text{Eff}_{\text{curr}}}{\text{Eff}_{\text{prev}}} \right) \times 100 \quad (7.5)$$

Table 7.18 presents measured metrics for different thread counts. The table includes the number of threads, events per second (EPS), total number of events, EPS Gain percentage compared to the previous thread count, and relative efficiency drop.

Figure 7.14 provides a visual representation of the same data in the form of a histogram. The figure shows the EPS and total events for each tested thread count.

Table 7.18 i7-1260P CFS Scalability Metrics

Threads	EPS	Total Events	EPS Gain [%]	Eff Drop [%]
1	3994.81	119,850	—	—
2	6582.46	197,483	64.8	17.6
4	10547.57	316,443	60.2	19.9
8	17177.56	515,359	62.9	18.6
16	22844.16	685,378	33.0	33.5

(source: own)

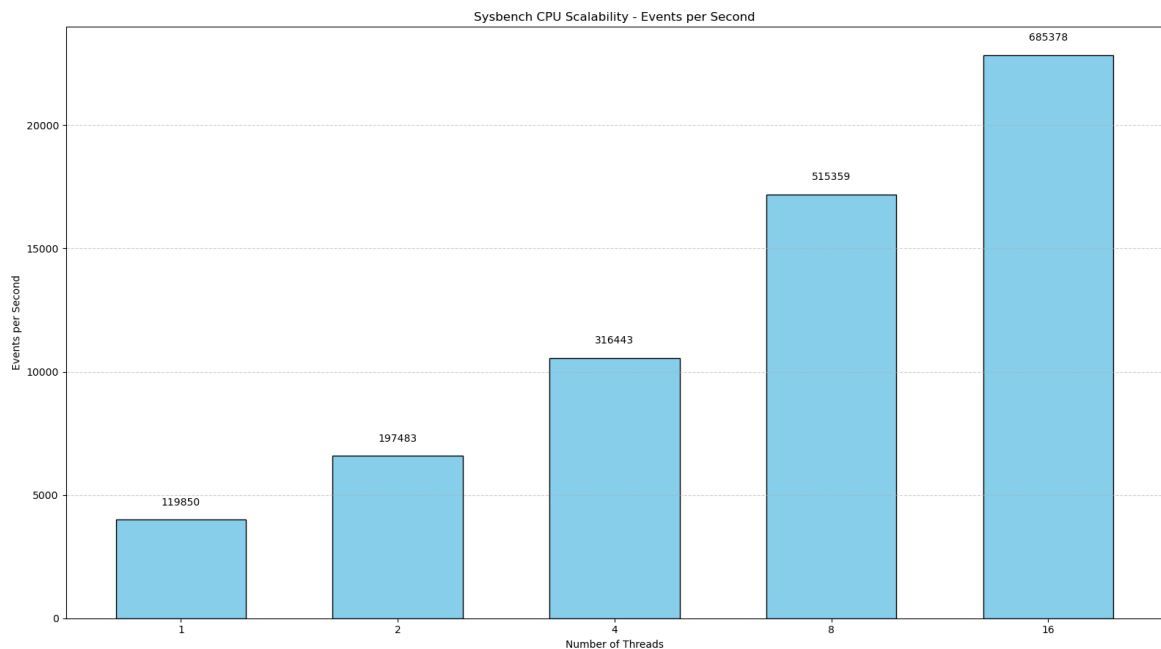


Figure 7.14 i7-1260P CFS scalability

(source: own)

7.1.11 EEVDF scalability test result

Scalability test results for the i7-1260P CPU using the EEVDF scheduler. Table 7.19 lists metrics including EPS, total events processed, percentage gain in EPS, and efficiency drop across increasing thread counts. The graphical representation is provided in Figure 7.15.

Interpretation and analysis follow the methodology and discussion outlined in 7.1.10.

Table 7.19 i7-1260P EEVDF scalability metrics

Threads	EPS	Total Events	EPS Gain [%]	Eff Drop [%]
1	3794.22	113,831	—	—
2	6449.54	193,495	70.0	15.0
4	10198.01	305,959	58.1	20.9
8	15724.12	471,755	54.2	22.9
16	20750.61	622,647	32.0	34.0

(source: own)

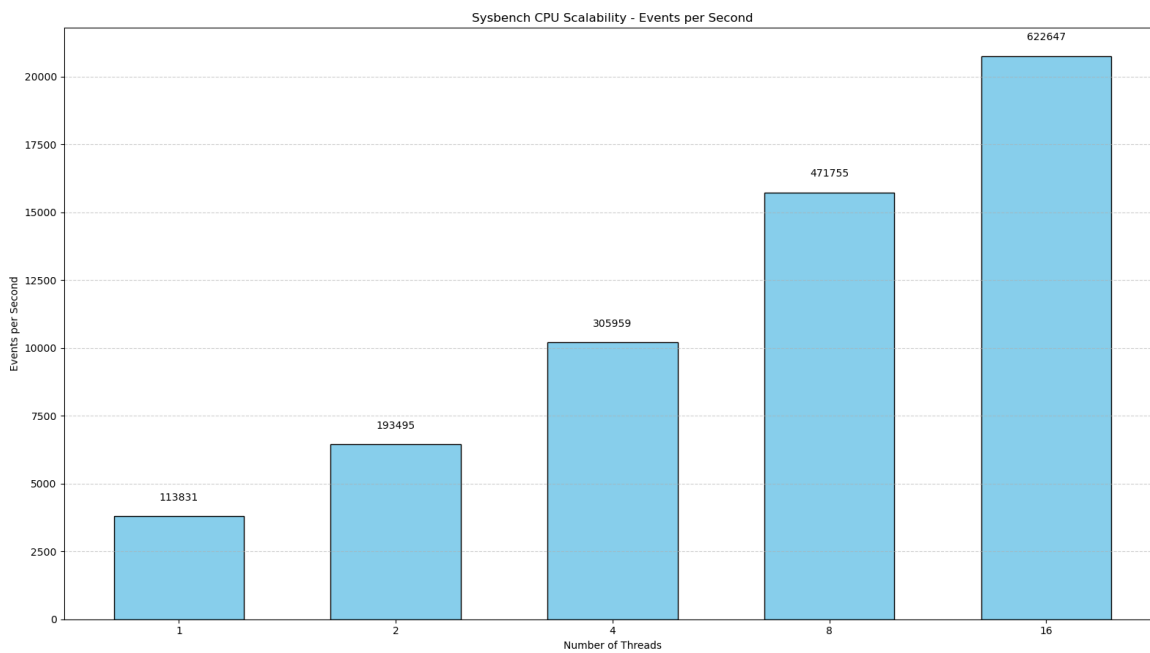


Figure 7.15 i7-1260P EEVDF scalability

(source: own)

7.1.12 CFS starvation detection test result

This test methodology is described in 5.7. Figure 7.16 illustrates the runtime lifespan of all threads over time, highlighting disparities in execution flow. Figure 7.17 presents a box plot of CPU time distributed across threads grouped [43]. Table 7.20 summarizes the statistical results for two groups: `nice = -10` and `nice = 19`.

Table 7.20 i7-1260P CFS starvation CPU time summary statistics

Nice	Samples	Avg CPU Time [s]	Min [s]	Max [s]	Std Dev [s]
-10	256	0.820	0.280	1.470	0.254
19	16	0.816	0.740	0.880	0.040

(source: own)

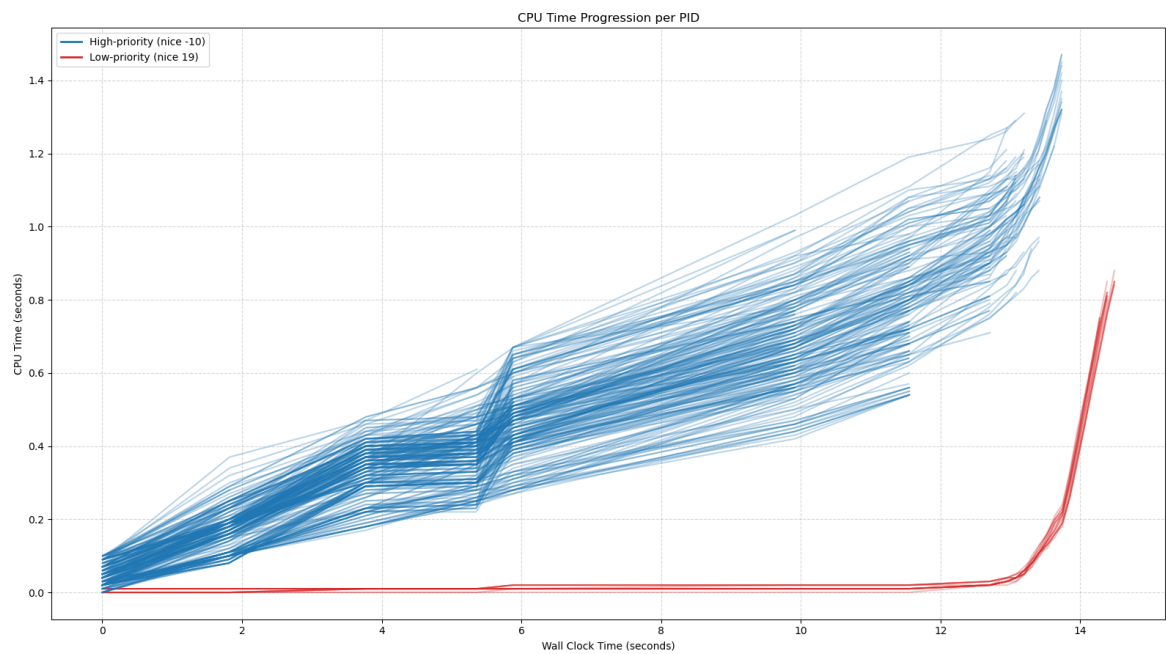


Figure 7.16 i7-1260P CFS starvation thread life

(source: own)

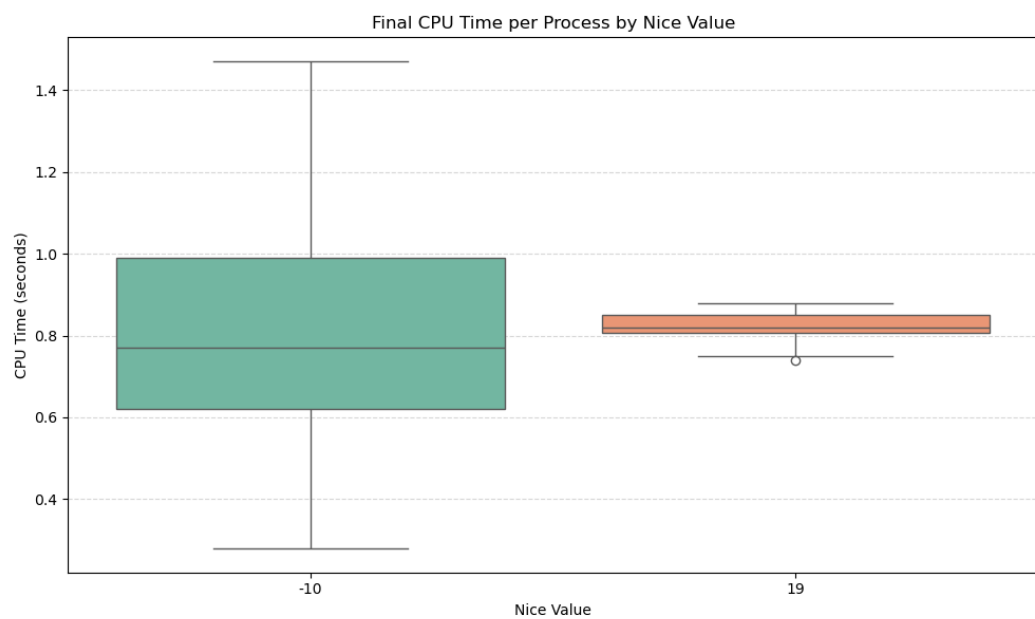


Figure 7.17 i7-1260P CFS starvation CPU time

(source: own)

7.1.13 EEVDF starvation detection test result

The EEVDF starvation test for i7-1260P was executed under the same methodology as outlined in 5.7. Figure 7.18 provides visualizations of the thread life progression.

Table 7.21 outlines the statistics, including the count of sampled threads, average, minimum, and maximum CPU times, and standard deviation.

Table 7.21 i7-1260P EEVDF starvation CPU time summary statistics

Nice	Count	Avg CPU Time [s]	Min [s]	Max [s]	Std Dev [s]
-10	256	0.678	0.270	1.000	0.175
19	16	1.303	1.240	1.390	0.049

(source: own)

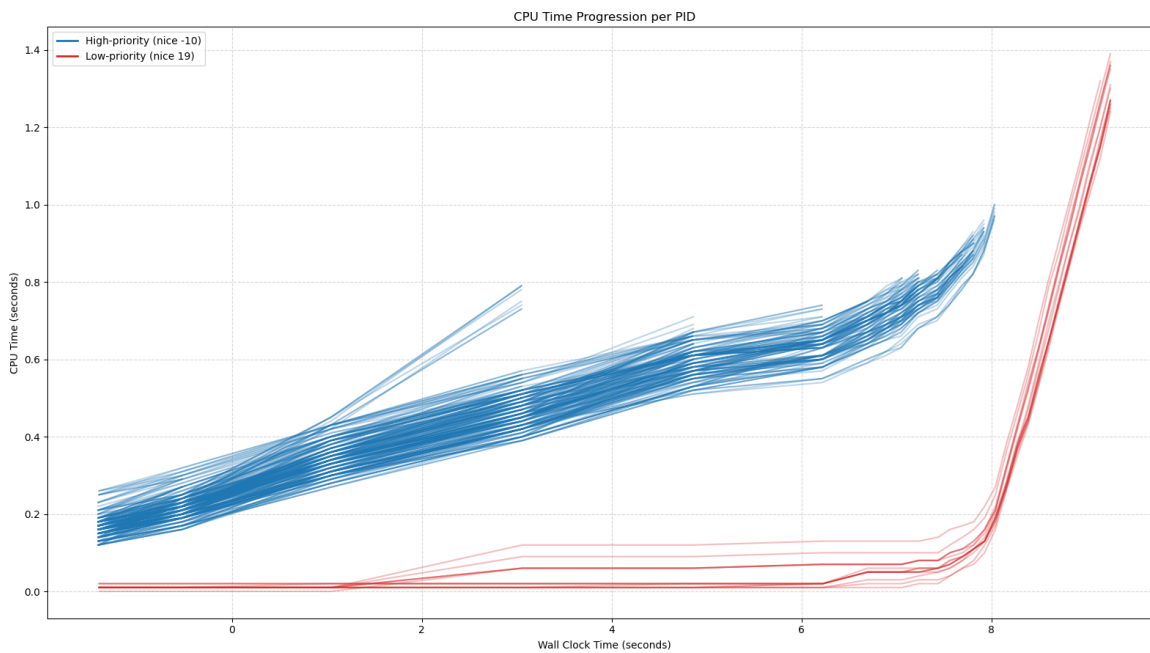


Figure 7.18 i7-1260P EEVDF starvation thread life

(source: own)

7.2 AMD ZEN+ (Ryzen 5 2600)

The AMD Ryzen[®] 5 2600 processor is based on the ZEN+ architecture and belongs to the second generation of Ryzen CPUs. It features 6 physical cores with support for SMT, enabling a total of 12 logical CPUs. The processor operates with a base clock of 3.4 GHz and can dynamically boost performance with turbo frequencies. With its 16 MB L3 cache and dual CCX design, it provides balanced performance for both single-threaded and parallel workloads [44].

7.2.1 DUT HW specification

The DUT is a desktop system equipped with the AMD Ryzen 5 2600 processor, intended to evaluate Linux scheduler behavior on a traditional homogeneous multi-core platform. Key hardware specifications relevant to the testing of process scheduling are provided in Table 7.22, and detailed CPU topology is shown by Figure 7.19.

Table 7.22 DUT: Ryzen 5 2600 - System Hardware Specification

Component	Specification
CPU Model	AMD Ryzen [®] 5 2600 Six-Core Processor
Architecture	x86_64
CPU Cores (Physical)	6
Threads per Core	2 (SMT)
Total Logical CPUs	12
Base Frequency	1550 MHz
Max Turbo Frequency	3400 MHz
L1 Cache (Data + Instruction)	192 KiB (Data), 384 KiB (Instruction) (6 instances)
L2 Cache	3 MiB
L3 Cache	16 MiB
NUMA Nodes	1 (CPUs 0-11)
RAM Installed	47 GiB
Swap Space	0 B (no swap configured)

(source: [17, 31, 32, 44])

7.2.2 CFS latency test result

Results of the latency test (5.2) executed on the Ryzen 5 2600 system using the CFS. These measurements capture minimum, maximum, and average latencies observed per thread.

Machine (47GB total)

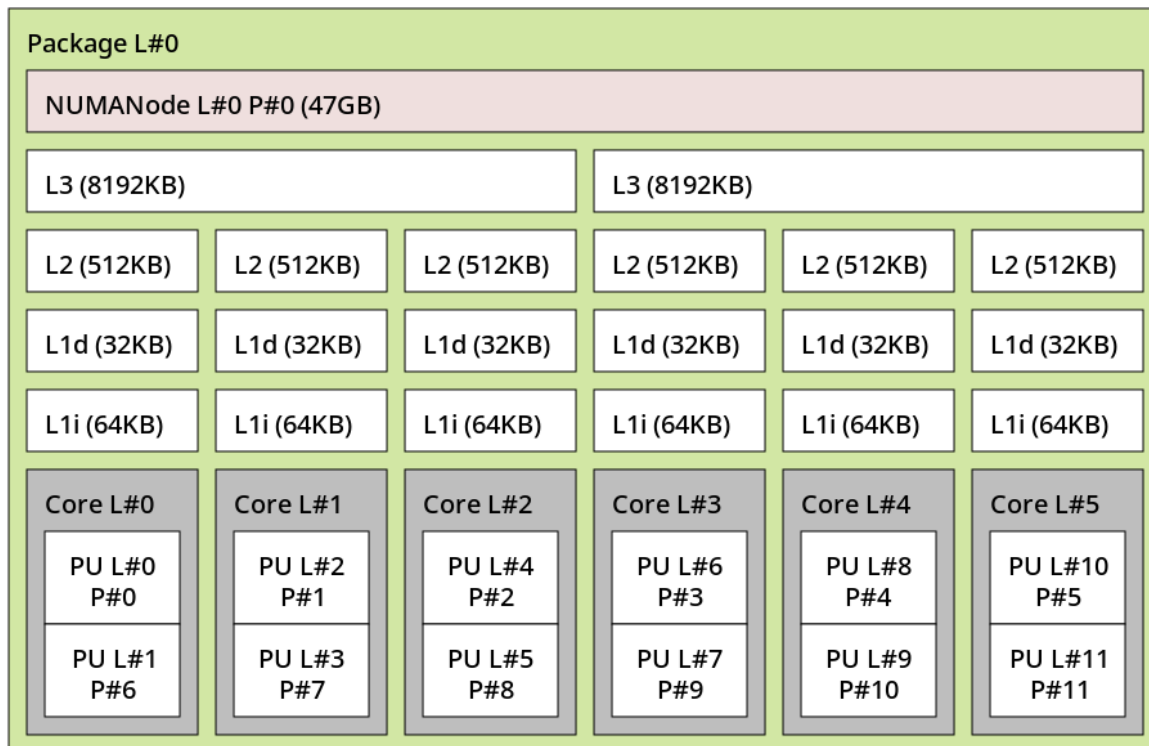


Figure 7.19 Ryzen 5 2600 CPU topology
(source: [32])

The detailed per-thread values are listed in Table 7.23, with an overall summary listed in Table 7.24. Figure 7.20 visualizes the per-thread and CPU average latency evolution across the measurement cycle.

Table 7.23 Ryzen 5 2600 CFS latency per CPU

Thread	Min [ns]	Max [ns]	Avg [ns]	Thread	Min [ns]	Max [ns]	Avg [ns]
T0	4606	92174	6887.00	T6	4238	111499	6809.00
T1	4070	85047	6255.00	T7	4524	79998	7177.00
T2	3930	84144	6437.00	T8	4342	82146	6622.00
T3	4236	175958	7154.00	T9	4035	145538	6926.00
T4	4273	187010	6642.00	T10	3894	83473	6478.00
T5	4350	91424	6643.00	T11	4309	84369	6751.00

(source: own)

Temperature impact for the measurement is visible from Figure 7.21 in contradiction to latency measurement on Intel laptop system 7.1.2 CPU temperature factor has no visible influence. Values are summarized by Table 7.25.

Table 7.24 Ryzen 5 2600 CFS overall latency summary

Metric	Value
Min [ns]	3894
Max [ns]	187010
Avg [ns]	6731.75
Iterations	200000

(source: own)

Table 7.25 Ryzen 5 2600 CFS frequency and temperature summary

Metric	Value
Temperature Min [°C]	49
Temperature Max [°C]	64
Temperature Avg [°C]	62.54
CPU0–5 Avg Freq [GHz]	3.60
CPU6–11 Avg Freq [GHz]	3.63

(source: own)

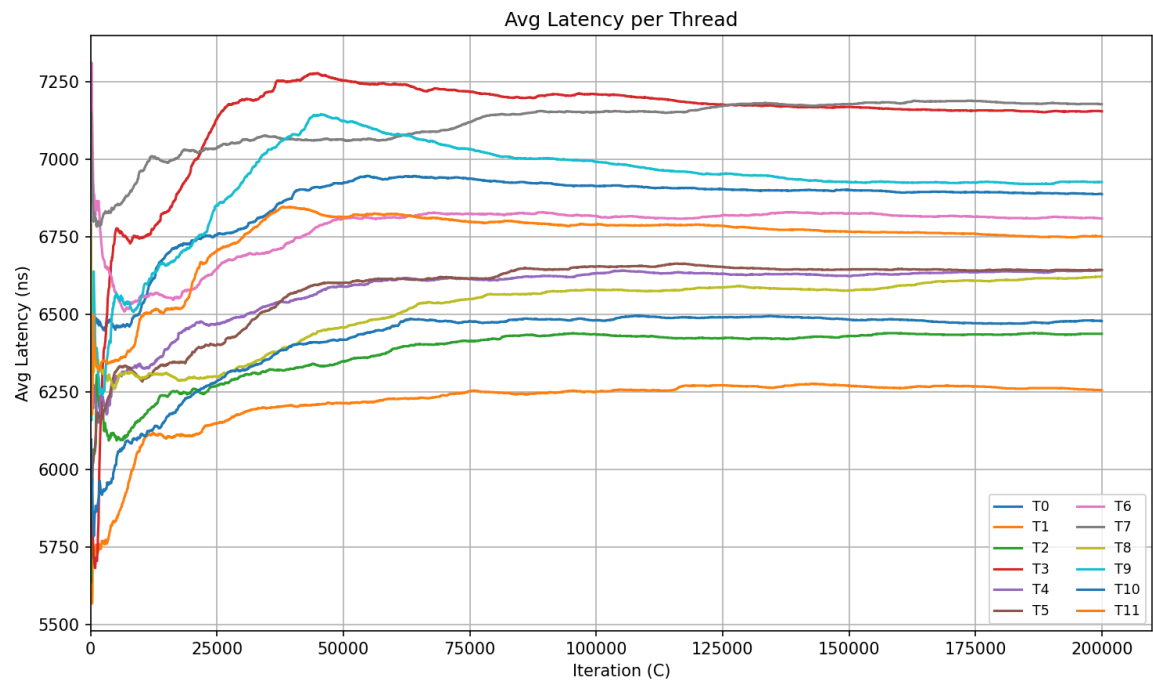


Figure 7.20 Ryzen 5 2600 CFS average latency per CPU thread

(source: own)

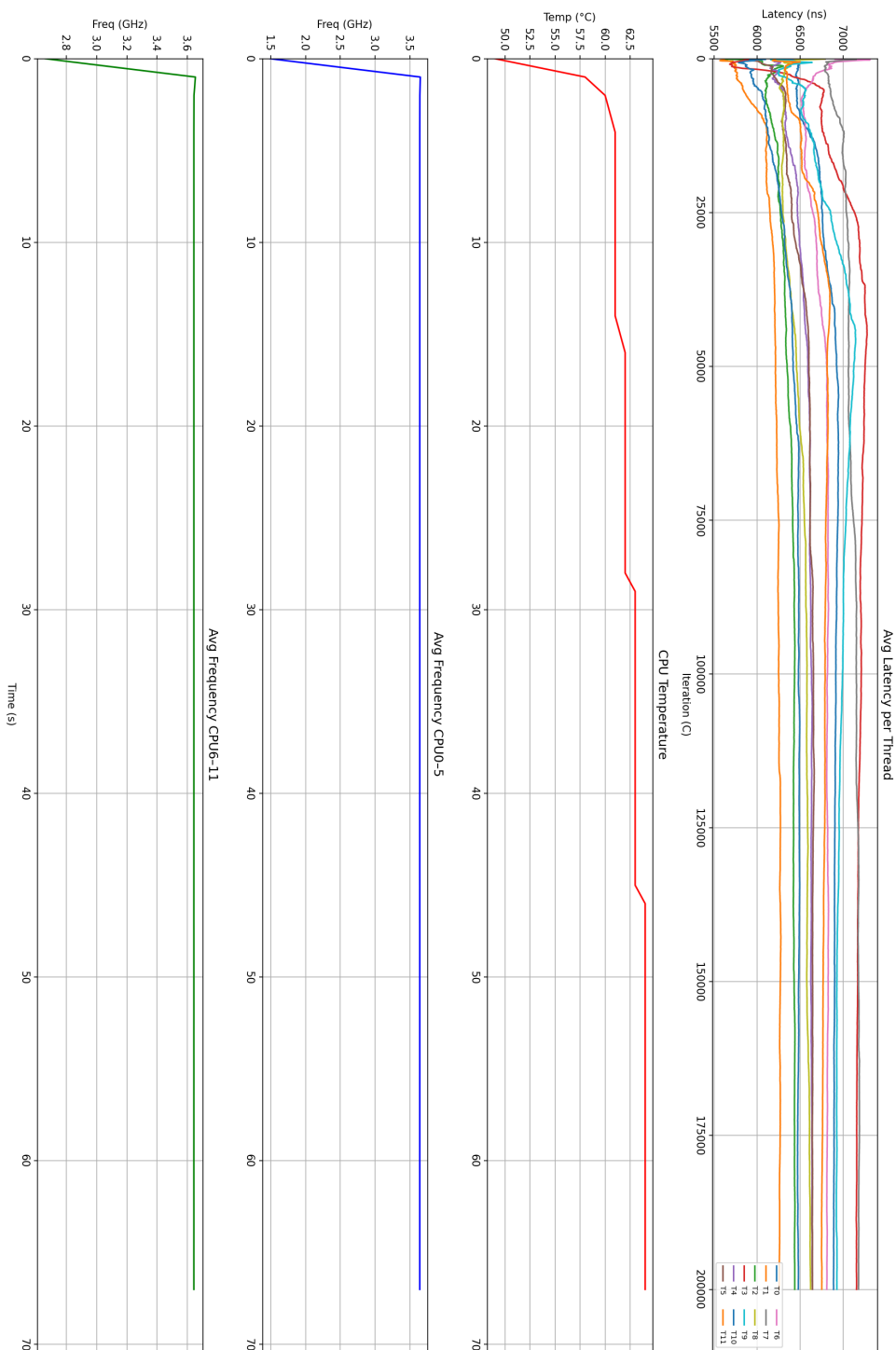


Figure 7.21 Ryzen 5 2600 CFS average latency with CPU temp and freq
(source: own)

7.2.3 EEVDF latency test result

Results of the latency test (5.2) conducted under the EEVDF scheduler on the Ryzen 5 2600. As with the CFS case, these results present latency characteristics per CPU thread. Table 7.26 provides thread-specific latency data, while Table 7.27 summarizes the overall latency, CPU temperature and frequency behavior. Figure 7.22 shows the latency trends across all active threads.

Table 7.26 Ryzen 5 2600 EEVDF latency per CPU thread

Thread	Min [ns]	Max [ns]	Avg [ns]
T0	4729	73816	6884.00
T1	4864	145963	7025.00
T2	4515	75749	6654.00
T3	4382	127365	6600.00
T4	4967	76906	7474.00
T5	4357	87850	6719.00
T6	4558	103421	6957.00
T7	4393	81754	6458.00
T8	4568	96761	6786.00
T9	4640	84560	6686.00
T10	4357	59093	6359.00
T11	4858	69159	6923.00

(source: own)

Table 7.27 Ryzen 5 2600 EEVDF overall measurement summary

Metric	Value
Min [ns]	4357
Max [ns]	145963
Avg [ns]	6793.75
Iterations (C)	200000
Temperature Min [°C]	48
Temperature Max [°C]	62
Temperature Avg [°C]	59.85
CPU0–5 Avg Freq [GHz]	3.62
CPU6–11 Avg Freq [GHz]	3.63

(source: own)

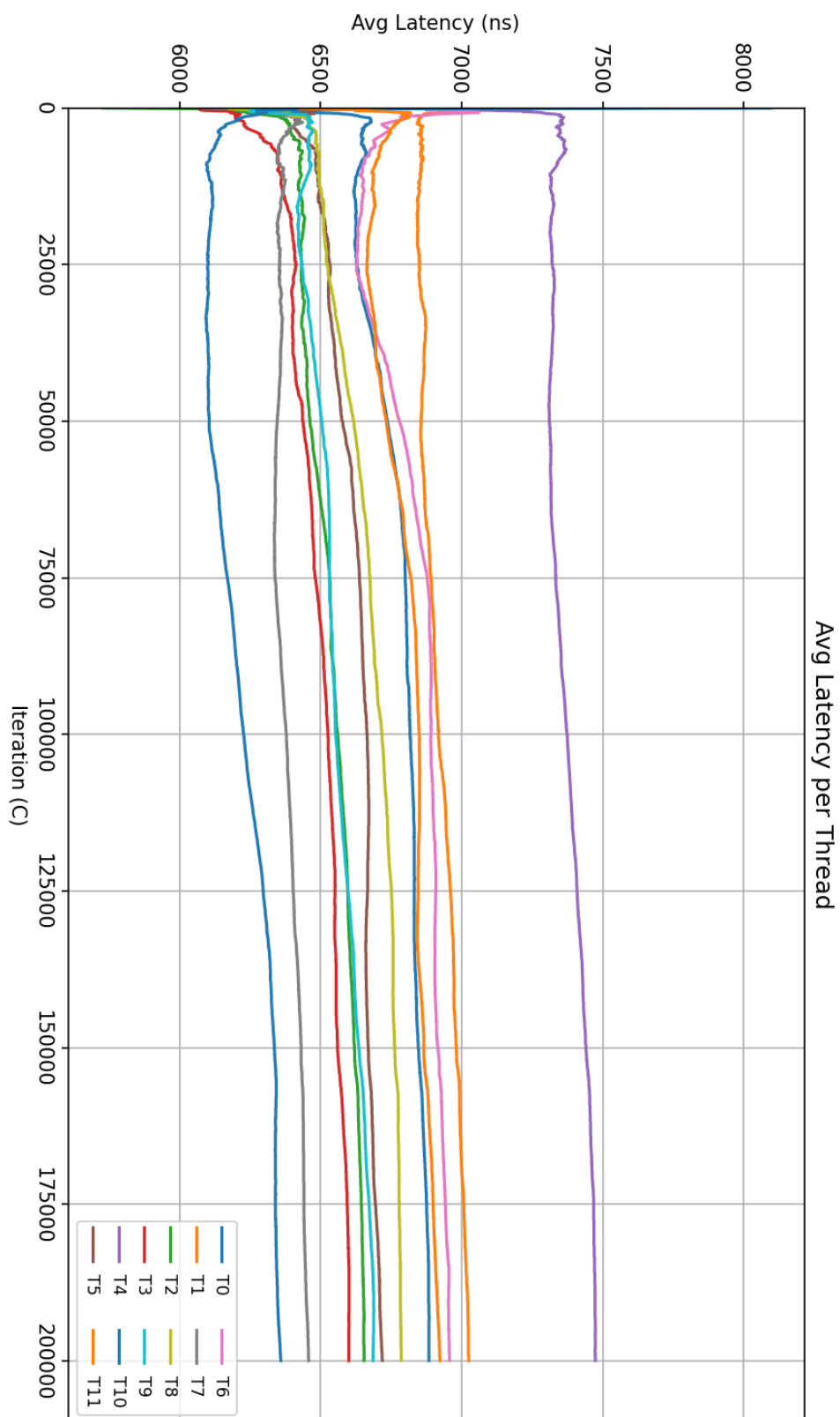


Figure 7.22 Ryzen 5 2600 EEVDF average latency per CPU
(source: own)

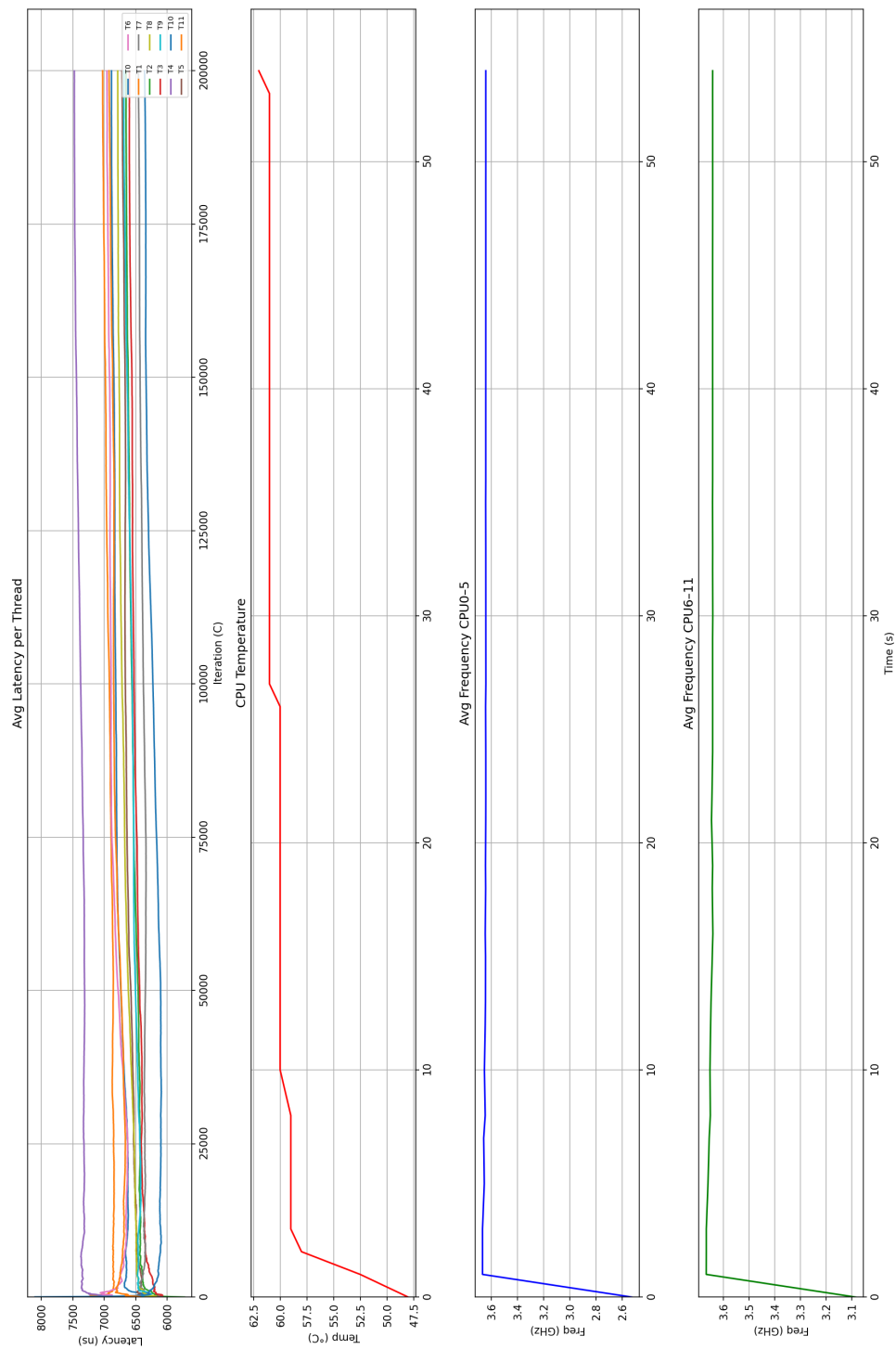


Figure 7.23 Ryzen 5 2600 EEVDF average latency with CPU temp and freq
(source: own)

7.2.4 CFS fairness test result

CFS fairness evaluation on the Ryzen 5 2600 system uses the same test as described in Section 5.3. Results for each nice group are visualized in Figure 7.24 and Figure 7.25. These charts illustrate the normalized CPU utilization and CPU time distribution across all processes within each group.

Table 7.28 summarizes the key metrics, including CPU efficiency and utilization. More detailed values for each priority group are available in Table 7.29.

Interpretation of measured data and analysis method is described previously in 7.1.4.

Table 7.28 Ryzen 5 2600 CFS fairness summary

Nice	Avg CPU Time [s]	Avg Wall Time [s]	CPU Efficiency [%]	CPU Utilization [%]	Processes
−5	0.43	5.29	8.10	83.11	144
5	0.61	15.89	3.83	47.20	144
10	0.71	26.78	2.64	37.27	144
18	0.69	34.90	1.99	29.13	144

(source: own)

Table 7.29 Ryzen 5 2600 CFS fairness detailed metrics

Metric	nice = −5	nice = 5	nice = 10	nice = 18
Processes	144	144	144	144
Avg CPU Time [s]	0.43	0.61	0.71	0.69
Median CPU Time [s]	0.42	0.61	0.71	0.70
Std Dev CPU Time [s]	0.07	0.06	0.04	0.04
Avg Wall Time [s]	5.29	15.89	26.78	34.90
Total Wall Time [s]	761.68	2287.99	3856.53	5025.70
Total CPU Time [s]	61.43	88.06	102.14	99.59
Lifetime Span [s]	6.16	11.78	13.48	14.62
CPU Efficiency [%]	8.10	3.83	2.64	1.99
CPU Utilization [%]	83.11	47.20	37.27	29.13

(source: own)

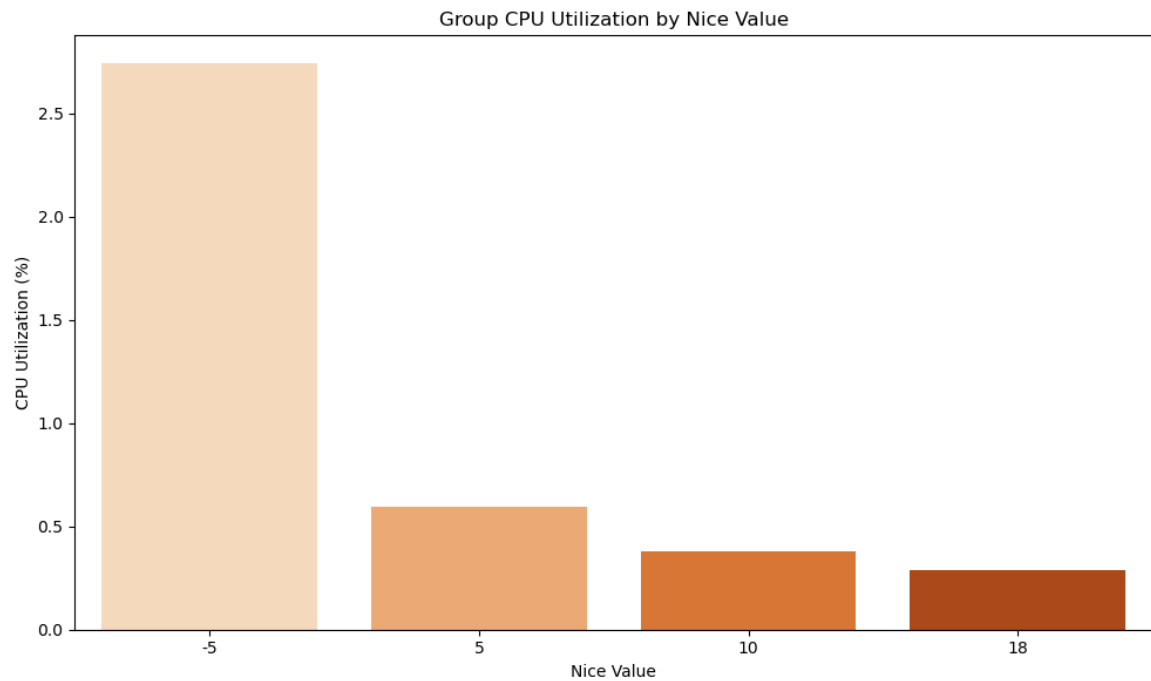


Figure 7.24 Ryzen 5 2600 CFS fairness CPU utilization
(source: own)

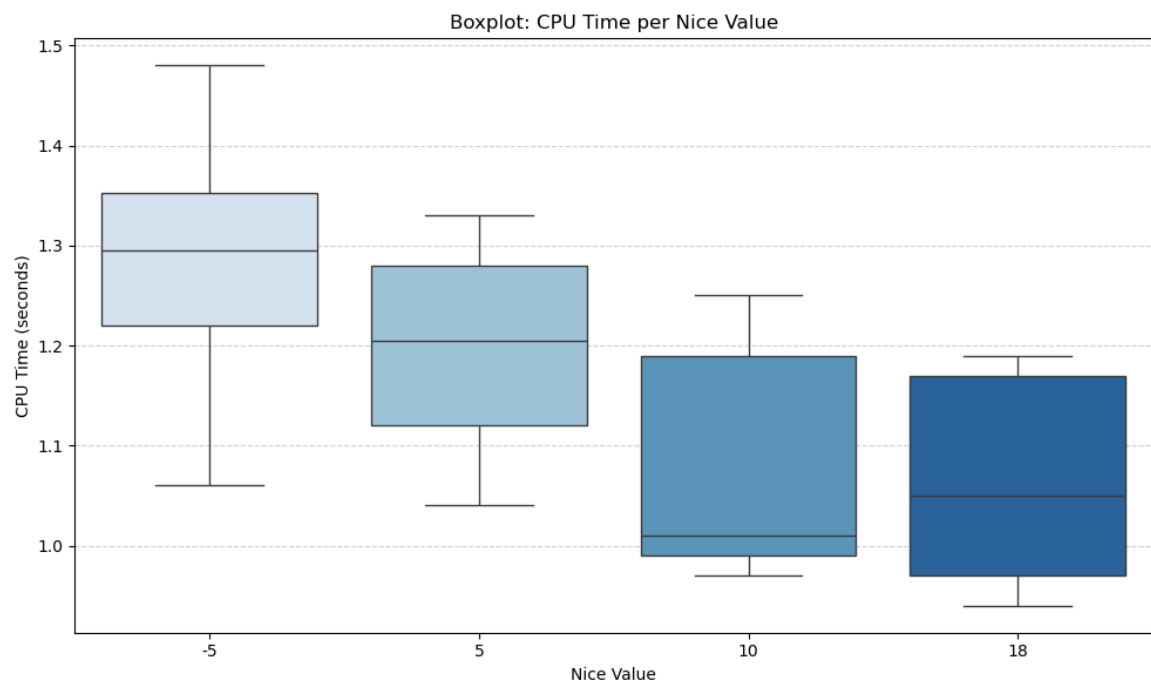


Figure 7.25 Ryzen 5 2600 CFS fairness CPU time
(source: own)

7.2.5 EEVDF fairness test result

EEVDF fairness test on the Ryzen 5 2600 system follows the same methodology as described in Section 5.3. Group-wise results are shown in Figure 7.26 and Figure 7.27. CPU utilization and time distribution for each nice value are depicted.

Table 7.30 summarizes group performance across key metrics, and per-group breakdowns are detailed in Table 7.31.

Interpretation of data as described in 7.1.4.

Table 7.30 Ryzen 5 2600 EEVDF fairness summary

Nice	Avg CPU Time [s]	Avg Wall Time [s]	CPU Efficiency [%]	CPU Utilization [%]	Processes
−5	1.01	4.89	20.75	195.80	144
5	1.07	22.27	4.81	55.19	144
10	1.07	32.86	3.26	38.31	144
18	1.08	43.90	2.47	28.94	144

(source: own)

Table 7.31 Ryzen 5 2600 EEVDF fairness detailed metrics

Metric	nice = −5	nice = 5	nice = 10	nice = 18
Processes	144	144	144	144
Avg CPU Time [s]	1.01	1.07	1.07	1.08
Median CPU Time [s]	0.99	1.08	1.07	1.10
Std Dev CPU Time [s]	0.08	0.06	0.07	0.09
Avg Wall Time [s]	4.89	22.27	32.86	43.90
Total Wall Time [s]	704.27	3206.54	4731.32	6320.94
Total CPU Time [s]	146.16	154.20	154.44	156.20
Lifetime Span [s]	6.22	23.28	33.59	44.98
CPU Efficiency [%]	20.75	4.81	3.26	2.47
CPU Utilization [%]	195.80	55.19	38.31	28.94

(source: own)

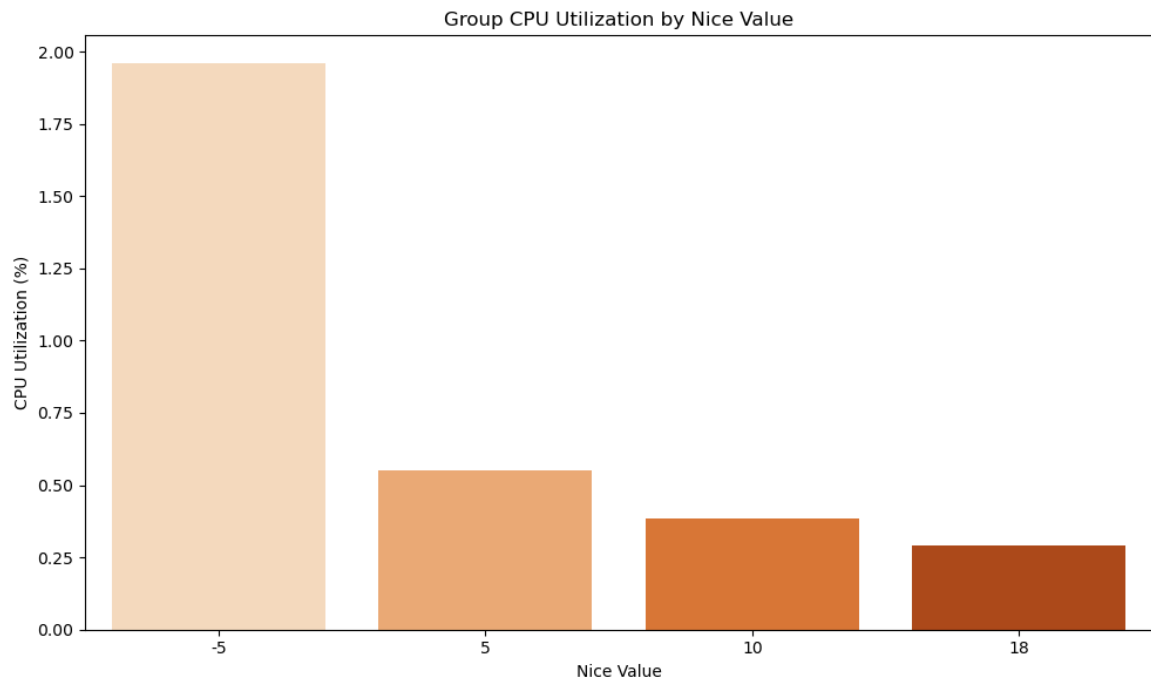


Figure 7.26 Ryzen 5 2600 EEVDF fairness CPU utilization
(source: own)

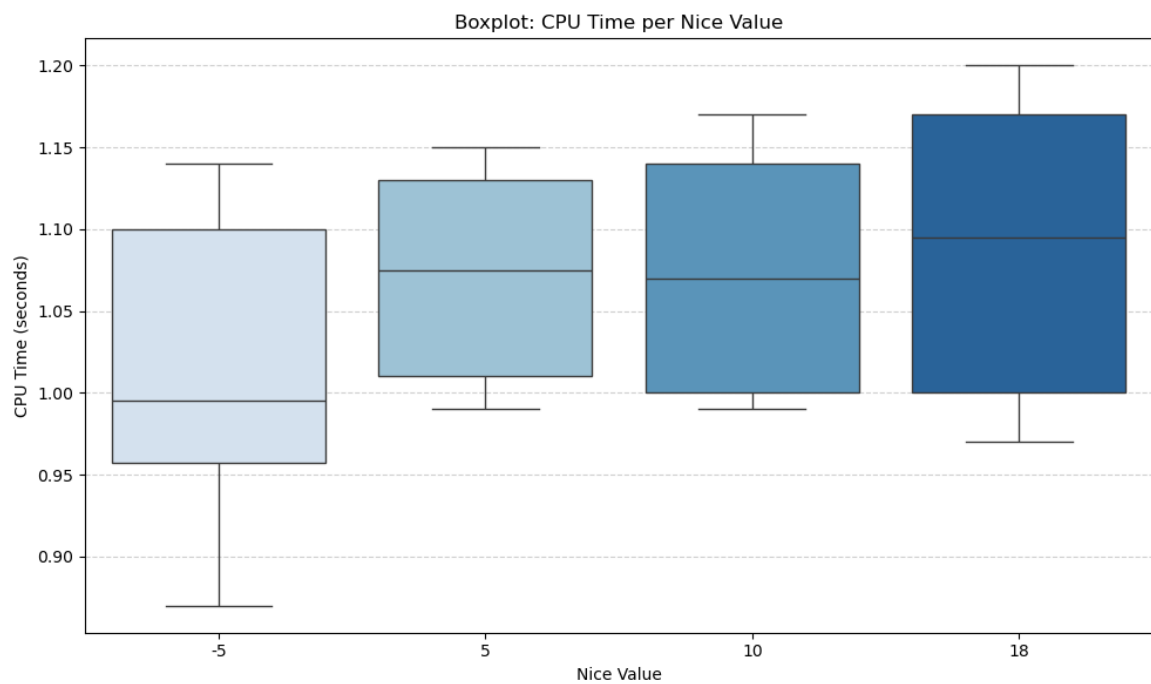


Figure 7.27 Ryzen 5 2600 EEVDF fairness CPU time
(source: own)

7.2.6 CFS throughput test result

The CFS throughput on the Ryzen 5 2600 was evaluated using the same methods outlined in 5.4. The resulting data are visualized in Figure 7.28. Detailed numerical results for each scenario are presented in Table 7.32.

Table 7.32 Ryzen 5 2600 CFS throughput results for all scenarios

Metric	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Description	30 s, no background	30 s, with background	100 s, no background	100 s, with background
Total Time [s]	30.001	30.0084	100.001	100.0164
Total Events	348,342	240,082	1,154,087	831,039
Events per Second	11,610.53	8,000.21	11,540.61	8,308.94

(source: own)

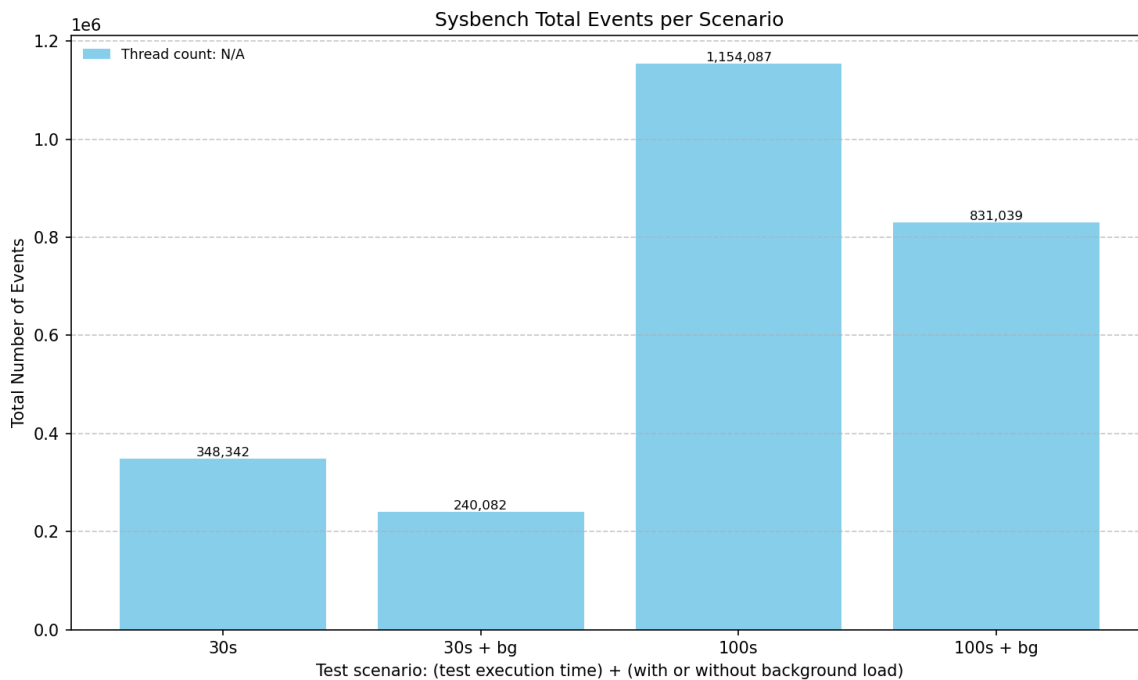


Figure 7.28 Ryzen 5 2600 CFS throughput

(source: own)

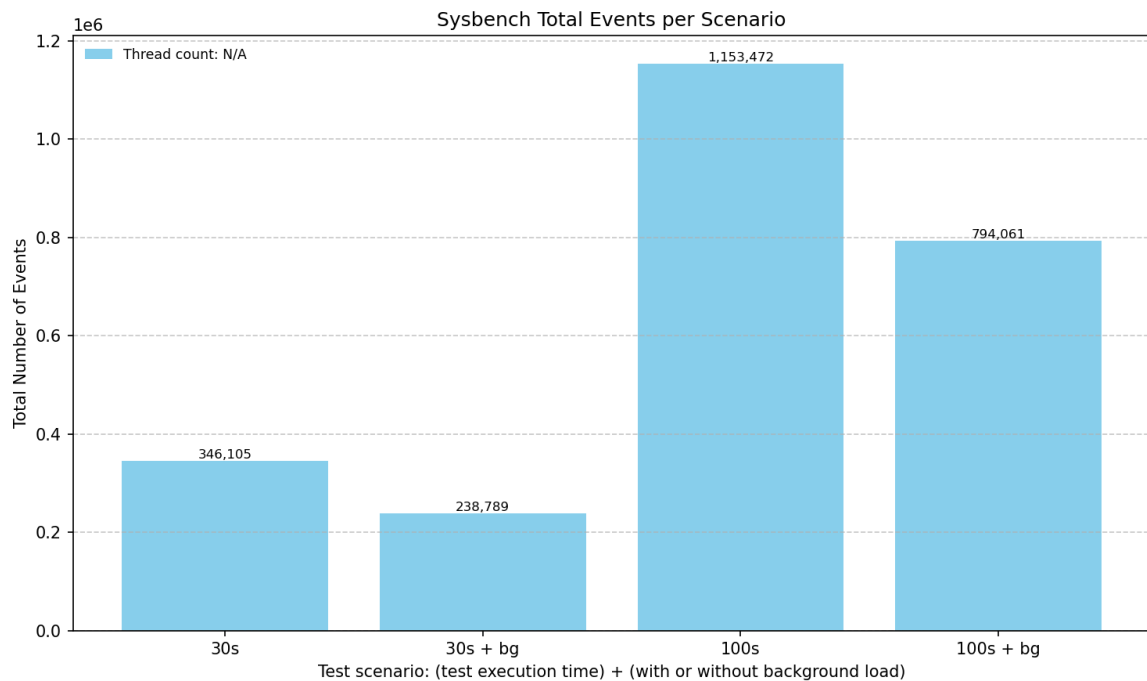
7.2.7 EEVDF throughput test result

The results are visualized in Figure 7.29. Detailed performance metrics for each of the four test scenarios are presented in Table 7.33.

Table 7.33 Ryzen 5 2600 EEVDF throughput results for all scenarios

Metric	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Description	30 s, no background	30 s, with background	100 s, no background	100 s, with background
Total Time [s]	30.00	30.00	100.00	100.00
Total Events	346,105	238,789	1,153,472	794,061
Events per Second	11,535.99	7,958.77	11,534.47	7,940.20

(source: own)

Figure 7.29 Ryzen 5 2600 EEVDF throughput
(source: own)

7.2.8 CFS context switching test result

The context switching test methodology described in 5.5. Figure 7.30 visualizes the amount of involuntary context switches per second by each thread.

Table 7.34 presents the per-thread statistics for involuntary context switches per second (nvcswh/s). The cumulative metrics are summarized in Table 7.35.

Table 7.34 Ryzen 5 2600 CFS per-thread context switching

Thread	Total [nvcswh/s]	Average [nvcswh/s]
T0	69.72	3.49
T1	120.77	4.03
T2	56.94	2.37
T3	33.89	2.61
T4	113.21	4.53
T5	59.94	3.15
T6	159.83	5.33
T7	191.49	7.09
T8	24.94	2.08
T9	214.66	7.16
T10	18.94	1.58
T11	87.83	3.66

(source: own)

Table 7.35 Ryzen 5 2600 CFS context switching summary

Metric	Value
Total [nvcswh/s]	1152.16
Average per Thread [nvcswh/s]	3.92
Peak Rate (Max nvcswh/s)	21.00
Std. Dev. of Per-Thread Averages	1.83
Thread Avg Max/Min Ratio	4.53

(source: own)

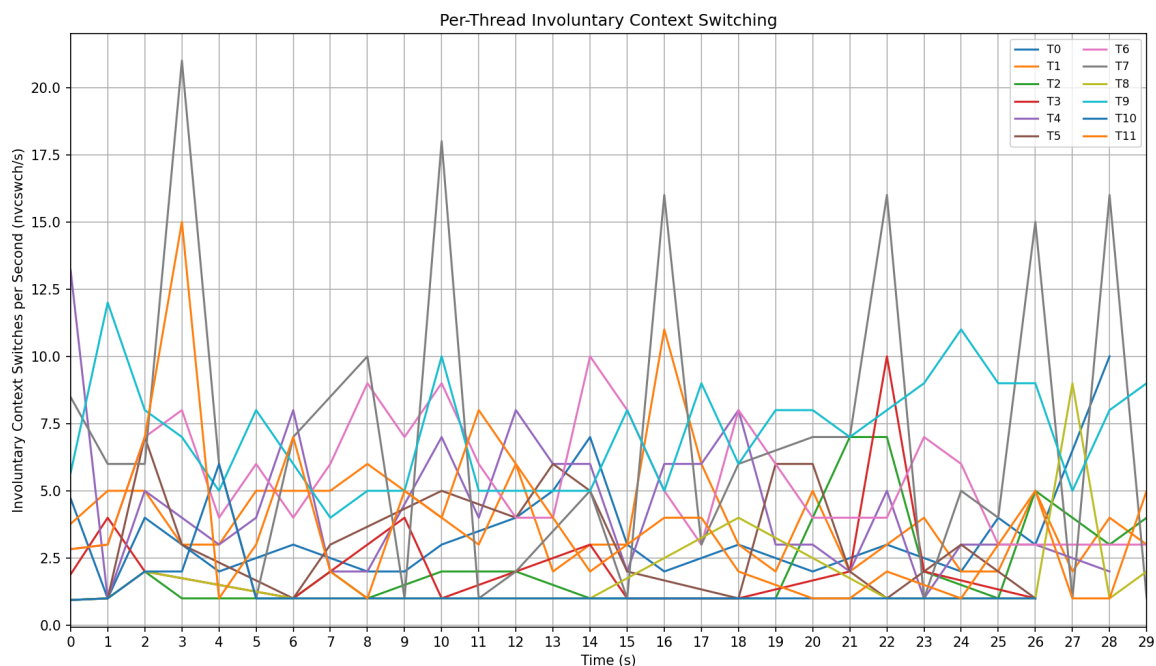


Figure 7.30 Ryzen 5 2600 CFS context switching

(source: own)

7.2.9 EEVDF context switching test result

Figure 7.31 visualizes the amount of involuntary context switches. Table 7.37 presents the per-thread statistics for involuntary context switches per second (nvcswh/s). The cumulative metrics are summarized in Table 7.36.

Table 7.36 Ryzen 5 2600 EEVDF context switching summary

Metric	Value
Total [nvcswh/s]	929.41
Average per Thread [nvcswh/s]	3.24
Peak Rate (Max nvcswh/s)	93.00
Std. Dev. of Per-Thread Averages	2.68
Thread Avg Max/Min Ratio	8.15

(source: own)

Table 7.37 Ryzen 5 2600 EEVDF per-thread context switching

Thread	Total [cswch/s]	Average [cswch/s]
T0	93.90	4.70
T1	176.54	5.88
T2	193.71	8.80
T3	10.99	1.10
T4	9.99	1.11
T5	21.98	1.37
T6	25.93	1.08
T7	143.46	4.78
T8	13.93	1.16
T9	175.49	6.05
T10	52.56	1.75
T11	10.93	1.09

(source: own)

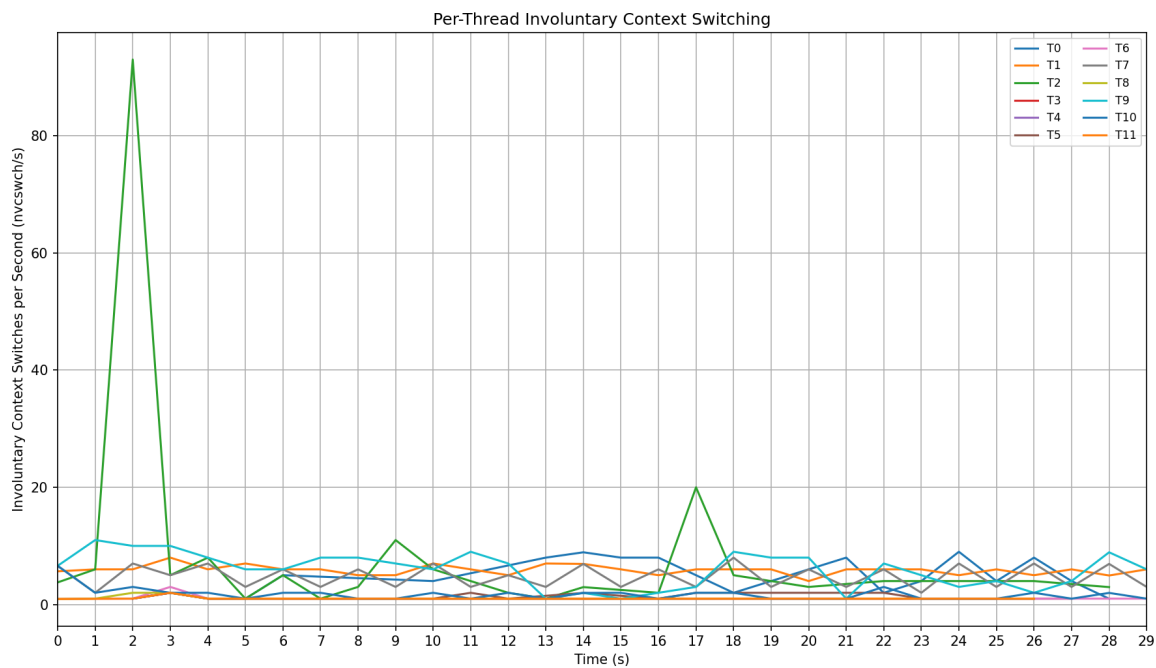


Figure 7.31 Ryzen 5 2600 EEVDF context switching
(source: own)

7.2.10 CFS scalability test result

Table 7.38 summarizes the results of the scalability test under the CFS scheduler. The corresponding visualization is shown in Figure 7.32, which illustrates the EPS across thread levels. Interpretation and analysis follow the methodology outlined in 7.1.10.

Table 7.38 Ryzen 5 CFS scalability metrics

Threads	EPS	Total Events	EPS Gain [%]	Eff Drop [%]
1	1911.28	57,345	—	—
2	3852.36	115,584	101.6	-0.8
4	7555.16	226,682	96.1	1.9
8	11155.25	334,681	47.7	26.2
12	11544.79	346,368	3.5	31.0

(source: own)

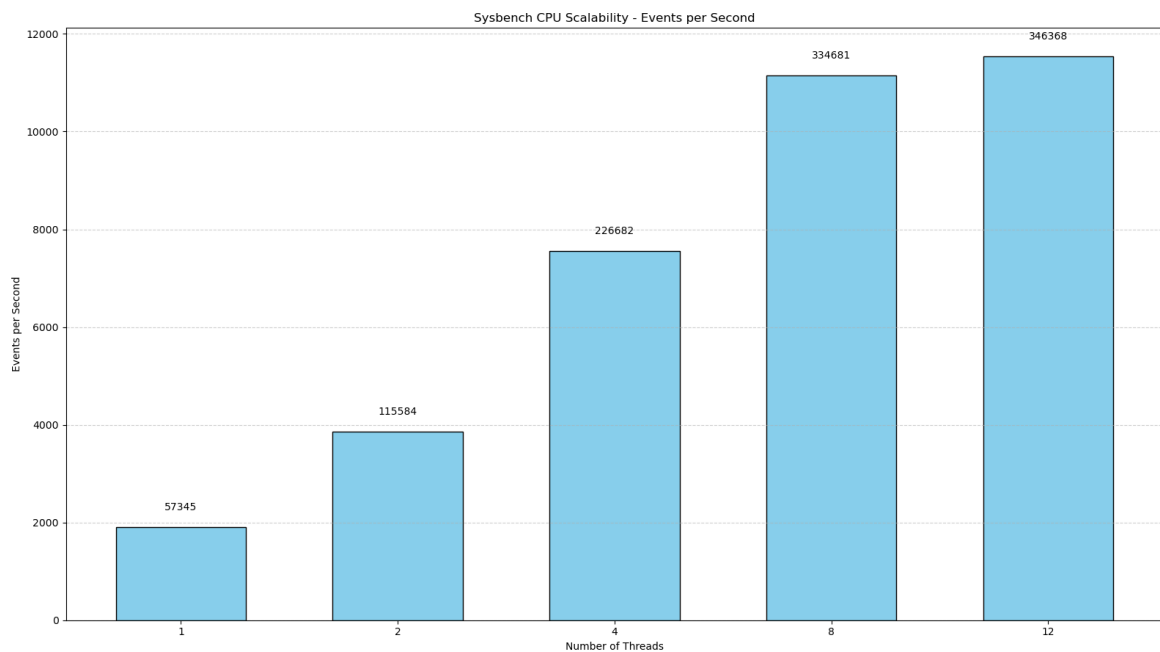


Figure 7.32 Ryzen 5 CFS scalability
(source: own)

7.2.11 EEVDF scalability test result

Table 7.39 summarizes the results of the scalability test under the EEVDF scheduler. The corresponding visualization is presented in Figure 7.33, showing the throughput in EPS across thread counts. Data interpretation and analysis are consistent with the approach described in 7.1.10.

Table 7.39 Ryzen 5 EEVDF scalability metrics

Threads	EPS	Total Events	EPS Gain [%]	Eff Drop [%]
1	1920.62	57,626	—	—
2	3850.74	115,536	100.5	-0.2
4	7644.28	229,342	98.5	0.7
8	11175.93	335,300	46.2	26.9
12	11586.13	347,608	3.7	30.9

(source: own)

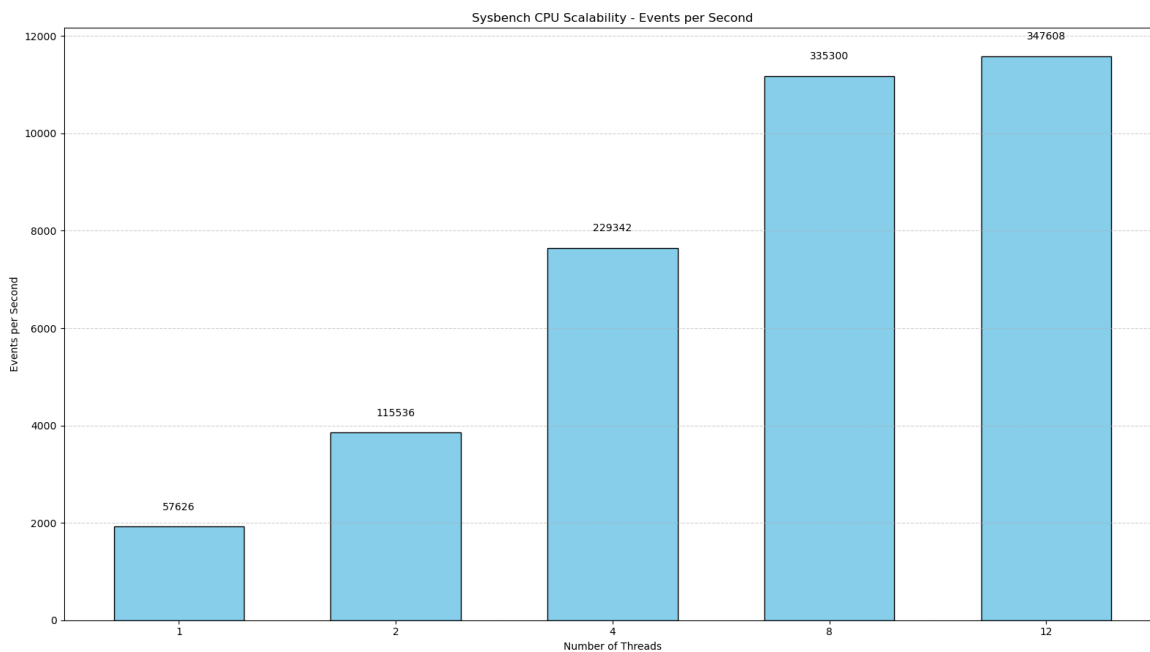


Figure 7.33 Ryzen 5 EEVDF scalability
(source: own)

7.2.12 CFS starvation detection test result

Figure 7.34 illustrates the progression of thread activity over time. Figure 7.35 displays a box plot of CPU time distribution for threads grouped by nice value. Table 7.40 summarizes the CPU time statistics for both priority groups [43].

Table 7.40 Ryzen 5 CFS starvation CPU time summary statistics

Nice	Samples	Avg [s]	Min [s]	Max [s]	Std Dev [s]
-10	191	1.273	1.070	1.540	0.081
19	16	0.987	0.880	1.100	0.109

(source: own)

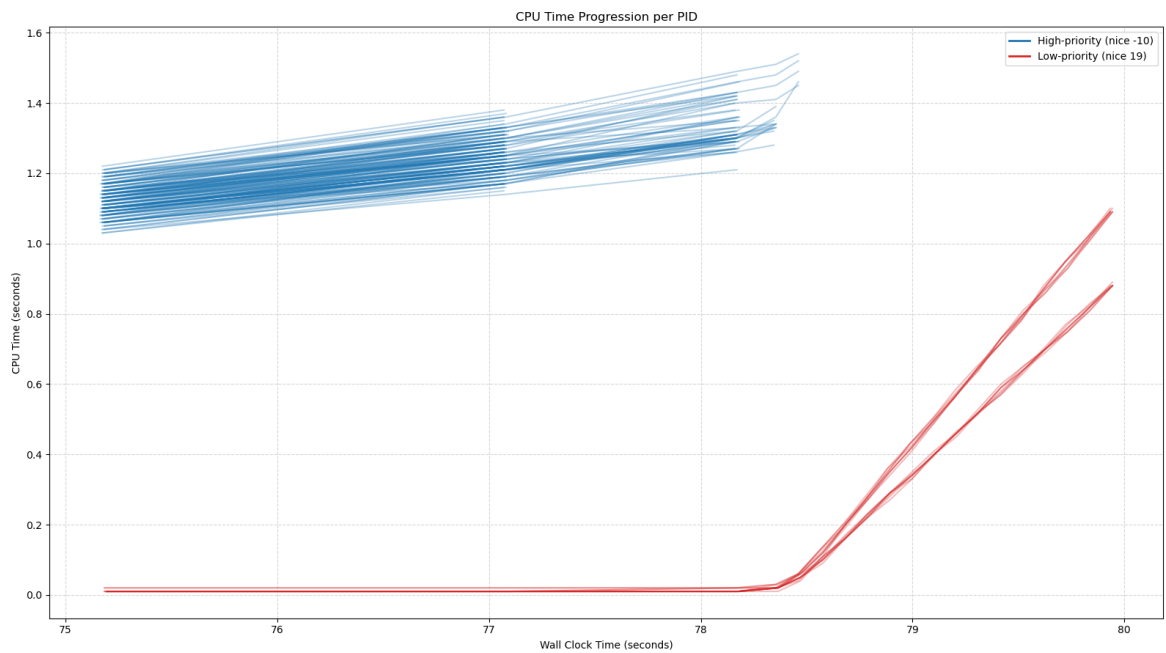


Figure 7.34 Ryzen 5 CFS starvation thread life
(source: own)

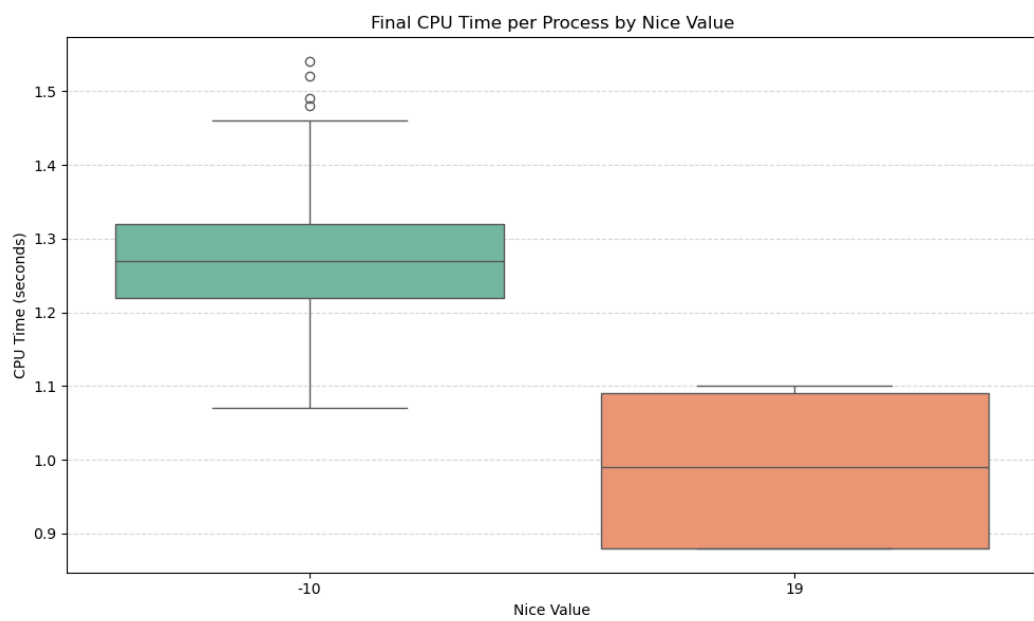


Figure 7.35 Ryzen 5 CFS starvation CPU time
(source: own)

7.2.13 EEVDF starvation detection test result

Figure 7.36 shows the thread life progression during the starvation test.

Table 7.41 summarizes CPU time statistics for both the high-priority and low-priority groups.

Table 7.41 Ryzen 5 EEVDF starvation CPU time summary statistics

Nice	Processes	Min [s]	Max [s]	Avg [s]	Std Dev [s]
-10	192	0.800	1.040	0.943	0.075
19	16	0.850	1.020	0.936	0.078

(source: own)

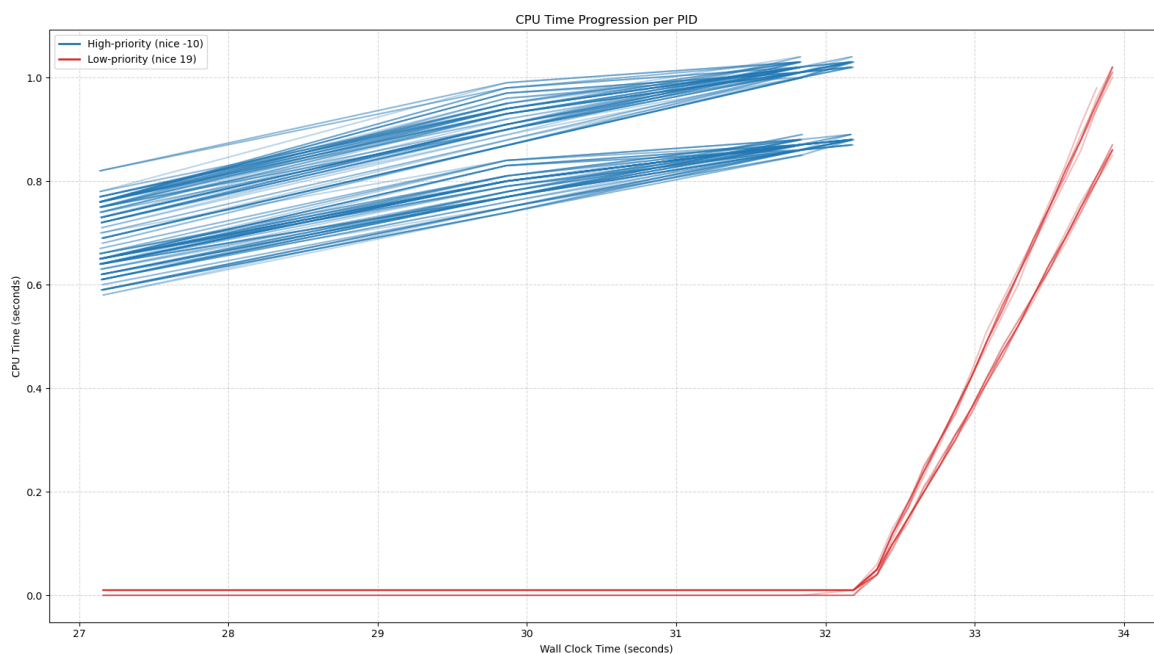


Figure 7.36 Ryzen 5 EEVDF starvation thread life

(source: own)

7.3 AMD ZEN3 (Ryzen 9 5950X)

The AMD Ryzen™ 9 5950X processor, based on the Zen 3 architecture, delivers high-end desktop performance with an efficient multi-core design. It features 16 physical cores and 32 threads, utilizing Simultaneous Multi-Threading (SMT) technology to maximize parallel processing capability. The CPU is organized using AMD's chiplet architecture, combining two core complexes (CCXs), each with up to 8 cores and shared access to a large 32 MB L3 cache segment, resulting in a total of 64 MB L3 cache. With base clock speeds starting at 550 MHz (under power-saving) and boosting up to 5084 MHz, the Ryzen 9 5950X offers outstanding single-thread and multi-thread performance. Its architecture is optimized for both heavy multitasking workloads and demanding real-time applications, making it highly suitable for scenarios requiring predictable scheduling and low-latency processing [45, 46].

7.3.1 DUT HW specification

Device under test is a desktop computer with a Ryzen 9 5950X processor, specifications relevant for process scheduler testing are listed in Table 7.42 and CPU topology is shown by Figure 7.37.

Table 7.42 DUT: Ryzen 9 5950X - System Hardware Specification

Component	Specification
CPU Model	AMD Ryzen® 9 5950X
Architecture	x86_64
CPU Cores (Physical)	16
Threads per Core	2 (SMT)
Total Logical CPUs	32
Base Frequency	550 MHz
Max Turbo Frequency	5084 MHz
L1 Cache (Data + Instruction)	512 KiB (Data), 512 KiB (Instruction)
L2 Cache	8 MiB
L3 Cache	64 MiB
NUMA Nodes	1 (CPUs 0-31)
RAM Installed	125 GiB
Swap Space	0 B (no swap configured)

(source: [17, 31, 32, 45])

Machine (126GB total)

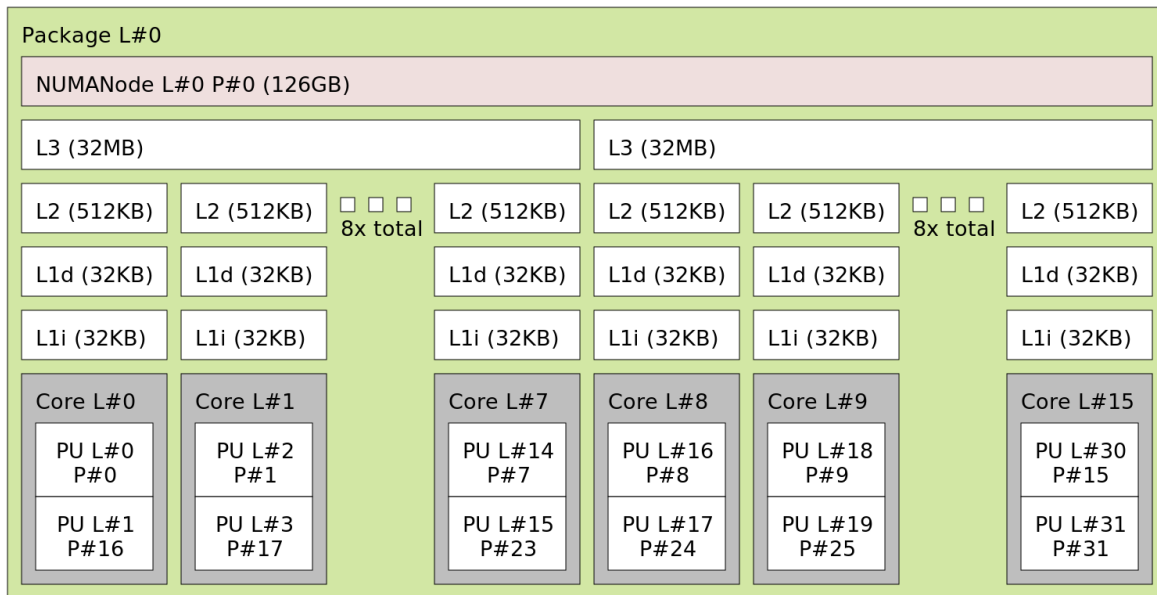


Figure 7.37 Ryzen 9 5950X CPU topology
(source: [32])

7.3.2 CFS latency test result

Results of the latency test (5.2) executed on the Ryzen 9 5950X system using the CFS scheduler. These measurements observe latencies per thread, values are listed in Table 7.44, with an overall summary shown in Table 7.43. Figure 7.38 visualizes the per-thread average latency, while Figure 7.39 presents the latency trend in relation to temperature and frequency over the measurement cycle.

Table 7.43 Ryzen 9 5950X CFS overall measurement summary

Metric	Value
Min [ns]	2305
Max [ns]	110196
Avg [ns]	5228.59
Iterations	200000
Temperature Min [°C]	38.5
Temperature Max [°C]	62
Temperature Avg [°C]	60.45
CPU0–15 Avg Freq [GHz]	4.34
CPU16–31 Avg Freq [GHz]	4.34

(source: own)

Table 7.44 Ryzen 9 5950X CFS latency per CPU

Thread	Min [ns]	Max [ns]	Avg [ns]	Thread	Min [ns]	Max [ns]	Avg [ns]
T0	2598	103540	5169.00	T16	2750	56094	5089.00
T1	2801	54839	4867.00	T17	3093	60667	5887.00
T2	2305	60262	4641.00	T18	2742	109955	5296.00
T3	2821	52765	5173.00	T19	2887	56044	5391.00
T4	2814	55019	5112.00	T20	2829	62723	5400.00
T5	2857	79933	5286.00	T21	2694	54444	5118.00
T6	3159	53356	5459.00	T22	3131	110196	5174.00
T7	2697	53919	5078.00	T23	2960	56979	5289.00
T8	2660	62065	5260.00	T24	2587	59062	5154.00
T9	2845	67790	5444.00	T25	2760	56344	5174.00
T10	2921	60658	4931.00	T26	2733	59906	5636.00
T11	2926	57450	5034.00	T27	2740	84077	5460.00
T12	2867	59373	5195.00	T28	2839	57800	5355.00
T13	2866	64887	5077.00	T29	2739	80044	5294.00
T14	2637	51064	5407.00	T30	2870	52973	5022.00
T15	2786	55898	5259.00	T31	2951	51651	5184.00

(source: own)

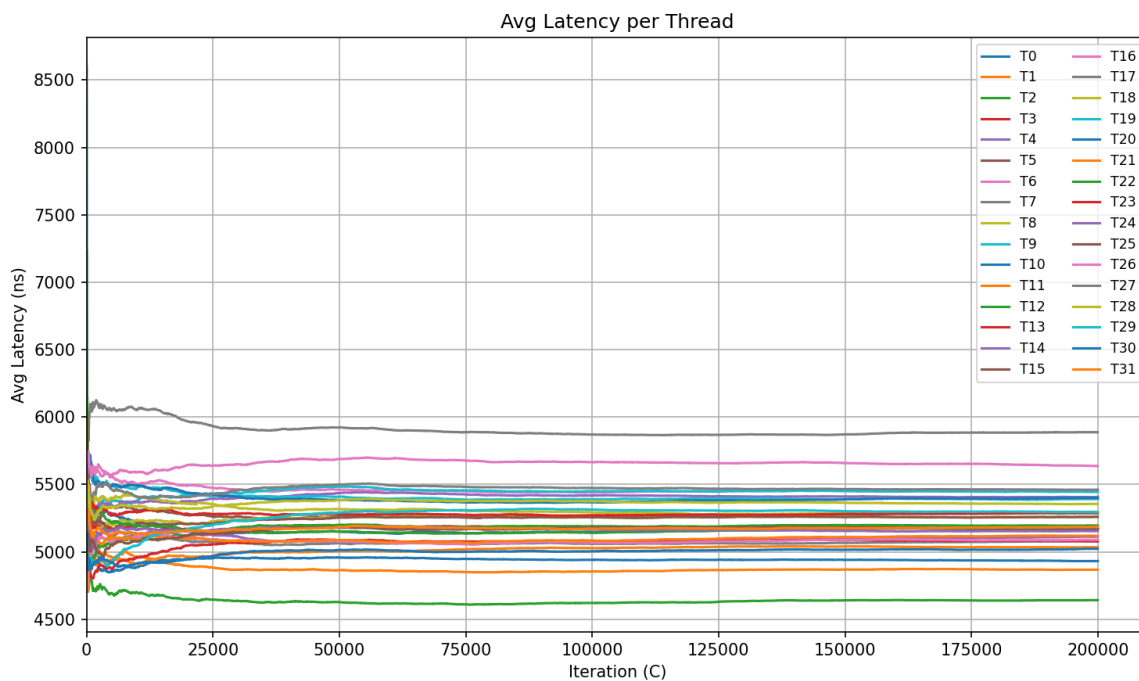


Figure 7.38 Ryzen 9 5950X CFS average latency per CPU thread

(source: own)

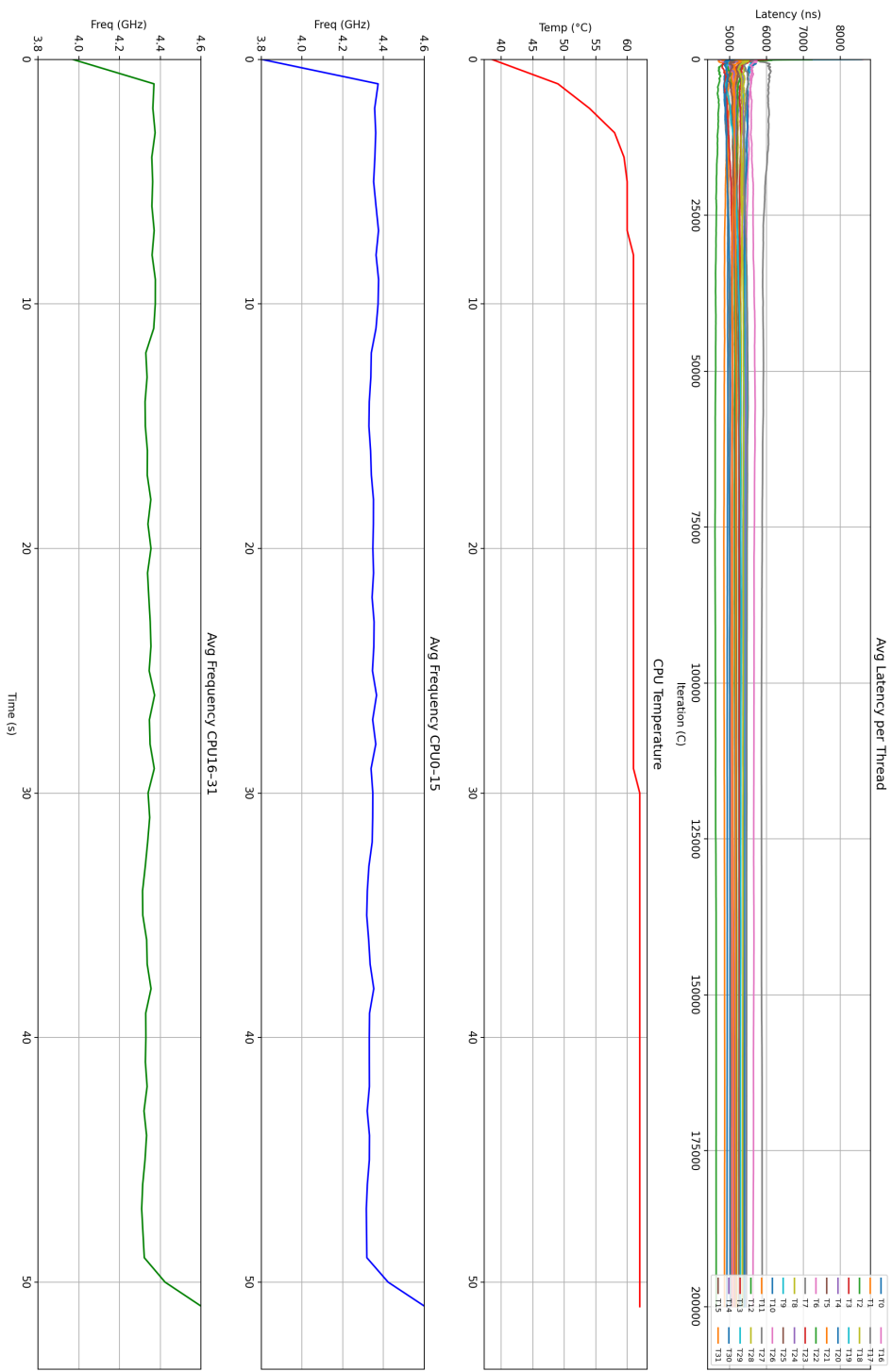


Figure 7.39 Ryzen 9 5950X CFS average latency with CPU temp and freq
(source: own)

7.3.3 EEVDF latency test result

These measurements observe latencies per-thread values are listed in Table 7.46, with an overall summary shown in Table 7.45. Figure 7.40 presents the latency trend in relation to temperature and frequency over the measurement cycle.

Table 7.45 Ryzen 9 5950X EEVDF overall measurement summary

Metric	Value
Min [ns]	2554
Max [ns]	82852
Avg [ns]	5075.19
Iterations	200000
Temperature Min [°C]	37.5
Temperature Max [°C]	62
Temperature Avg [°C]	59.99
CPU0–15 Avg Freq [GHz]	4.34
CPU16–31 Avg Freq [GHz]	4.34

(source: own)

Table 7.46 Ryzen 9 5950X EEVDF latency per CPU

Thread	Min [ns]	Max [ns]	Avg [ns]	Thread	Min [ns]	Max [ns]	Avg [ns]
T0	2807	53816	4981.00	T16	2582	55047	5121.00
T1	3395	65562	5003.00	T17	3211	60345	5140.00
T2	3296	48990	5086.00	T18	3419	49415	5142.00
T3	2971	47082	5058.00	T19	2975	52904	5275.00
T4	2843	49061	4821.00	T20	2706	48424	4758.00
T5	2770	52765	4623.00	T21	3473	67414	5550.00
T6	3204	46003	4867.00	T22	3278	56033	5032.00
T7	3081	63211	4911.00	T23	3127	64852	5491.00
T8	2884	76725	5501.00	T24	3017	57854	4944.00
T9	2636	49759	5271.00	T25	2936	48852	4896.00
T10	2849	50745	5263.00	T26	2912	59271	5203.00
T11	2654	51447	5050.00	T27	2795	42437	4921.00
T12	2789	49234	4844.00	T28	2589	52961	4972.00
T13	2554	80753	5346.00	T29	2842	50799	4936.00
T14	2818	72553	5367.00	T30	2742	55076	4900.00
T15	2949	49288	5295.00	T31	2682	82852	4838.00

(source: own)

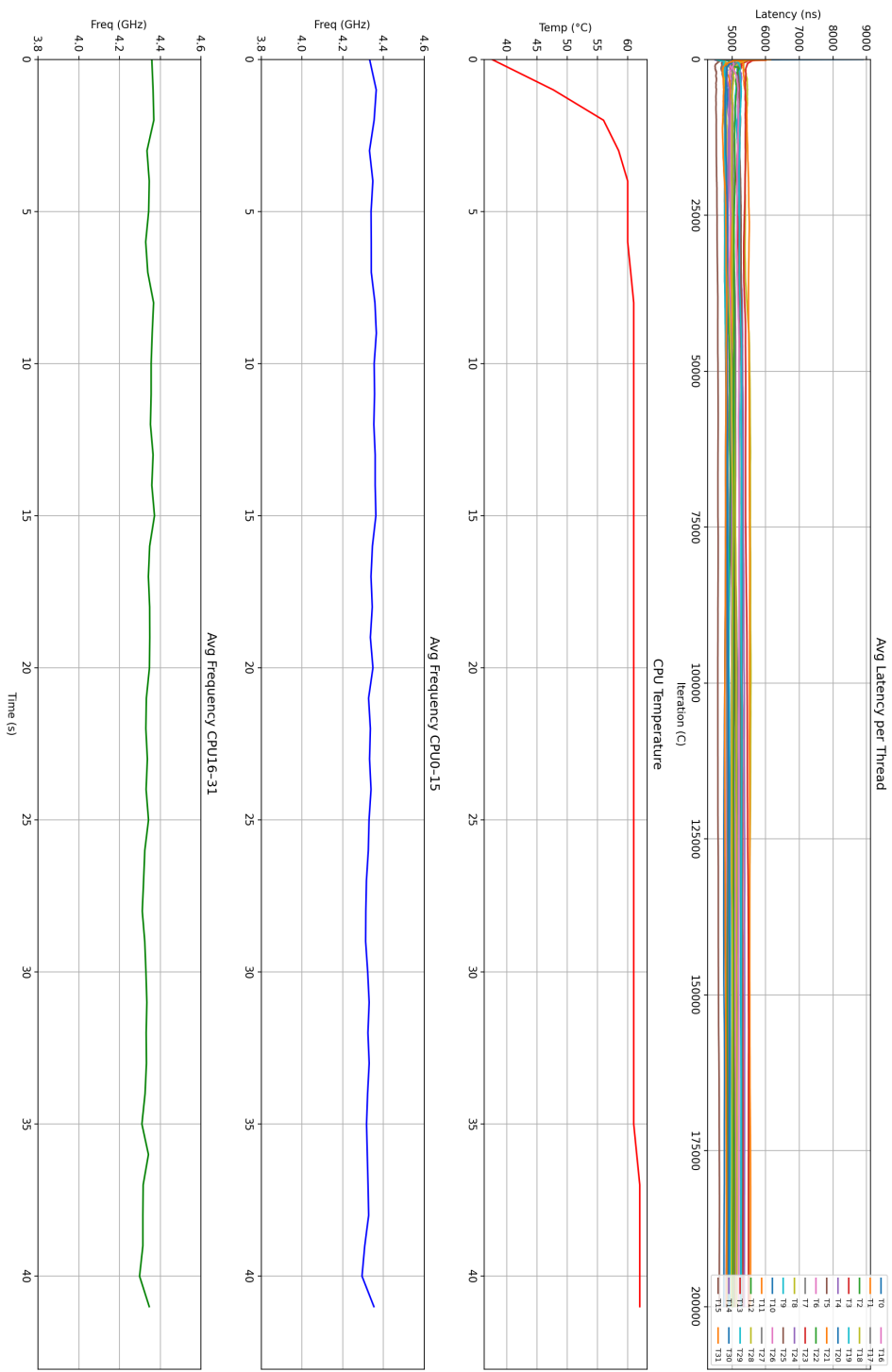


Figure 7.40 Ryzen 9 5950X EEVDF average latency with CPU temp and freq
(source: own)

7.3.4 CFS fairness test result

CFS fairness evaluation on the Ryzen 9 5950X system uses the same test as described in 5.3. Results for each nice group are visualized in Figure 7.41 and Figure 7.42. These charts illustrate the normalized CPU utilization and CPU time distribution across all processes within each group. Table 7.47 summarizes the key metrics, including CPU efficiency and utilization. More detailed values for each priority group are available in Table 7.48.

Interpretation of measured data and analysis method is described previously in 7.1.4.

Table 7.47 Ryzen 9 5950X CFS fairness summary

Nice	Avg CPU Time [s]	Avg Wall Time [s]	CPU Efficiency [%]	CPU Utilization [%]	Processes
−5	1.08	1.31	82.74	272.38	384
5	0.83	14.52	5.73	57.02	384
10	0.83	22.41	3.68	40.29	384
18	0.84	29.91	2.81	32.38	384

(source: own)

Table 7.48 Ryzen 9 5950X CFS fairness detailed metrics

Metric	nice = −5	nice = 5	nice = 10	nice = 18
Processes	274	384	384	384
Avg CPU Time [s]	1.08	0.83	0.83	0.84
Median CPU Time [s]	1.08	0.83	0.84	0.82
Std Dev CPU Time [s]	0.09	0.05	0.05	0.04
Avg Wall Time [s]	1.31	14.52	22.41	29.91
Total Wall Time [s]	358.21	5575.73	8606.91	11485.25
Total CPU Time [s]	296.38	319.26	317.07	322.34
Lifetime Span [s]	3.40	17.50	24.59	31.11
CPU Efficiency [%]	82.74	5.73	3.68	2.81
CPU Utilization [%]	272.38	57.02	40.29	32.38

(source: own)

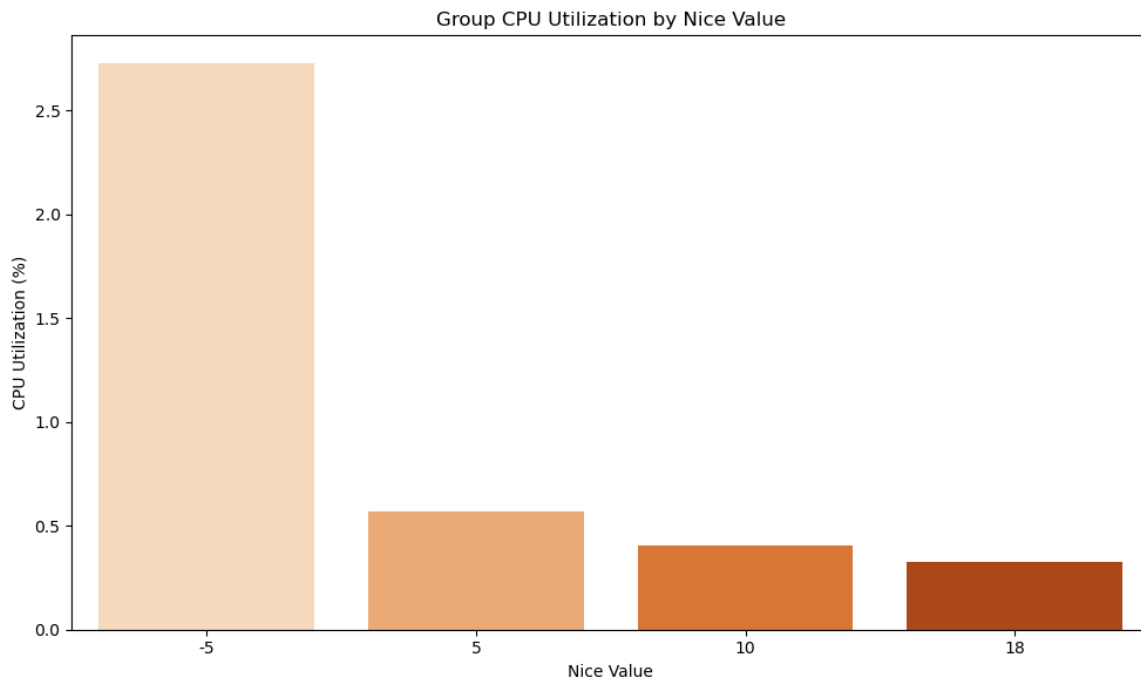


Figure 7.41 Ryzen 9 5950X CFS fairness CPU utilization
(source: own)

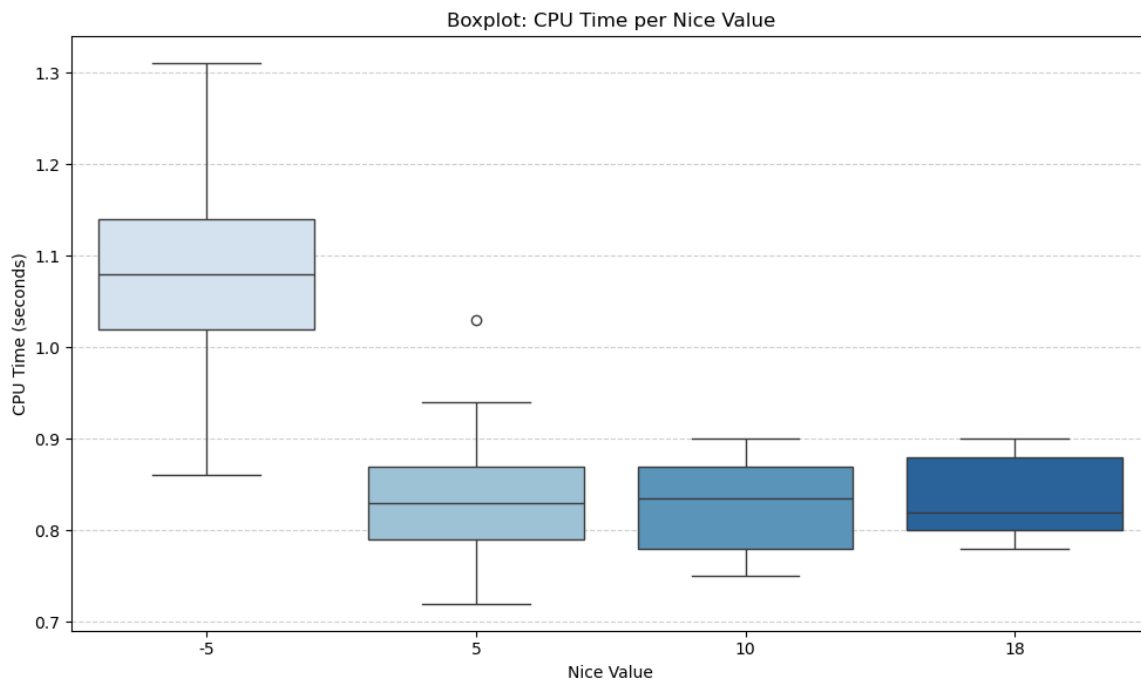


Figure 7.42 Ryzen 9 5950X CFS fairness CPU time
(source: own)

7.3.5 EEVDF fairness test result

Results for each nice group are visualized in Figure 7.43 and Figure 7.44. These charts illustrate the normalized CPU utilization and CPU time distribution across all processes within each group.

Table 7.49 summarizes the key metrics, including CPU efficiency and utilization. More detailed values for each priority group are available in Table 7.50.

Interpretation of measured data and analysis method is described previously in 7.1.4.

Table 7.49 Ryzen 9 5950X EEVDF fairness summary

Nice	Avg CPU Time [s]	Avg Wall Time [s]	CPU Efficiency [%]	CPU Utilization [%]	Processes
−5	0.77	0.13	588.58	1234.90	384
5	0.80	12.70	6.29	67.97	384
10	0.82	20.58	4.00	44.53	384
18	0.82	28.69	2.84	33.33	384

(source: own)

Table 7.50 Ryzen 9 5950X EEVDF fairness detailed metrics

Metric	nice = −5	nice = 5	nice = 10	nice = 18
Processes	240	384	384	384
Avg CPU Time [s]	0.77	0.80	0.82	0.82
Median CPU Time [s]	0.76	0.80	0.82	0.82
Std Dev CPU Time [s]	0.03	0.02	0.01	0.02
Avg Wall Time [s]	0.13	12.70	20.58	28.69
Total Wall Time [s]	31.35	4876.63	7902.07	11016.96
Total CPU Time [s]	184.52	306.94	316.09	313.28
Lifetime Span [s]	0.47	14.11	22.18	29.37
CPU Efficiency [%]	588.58	6.29	4.00	2.84
CPU Utilization [%]	1234.90	67.97	44.53	33.33

(source: own)

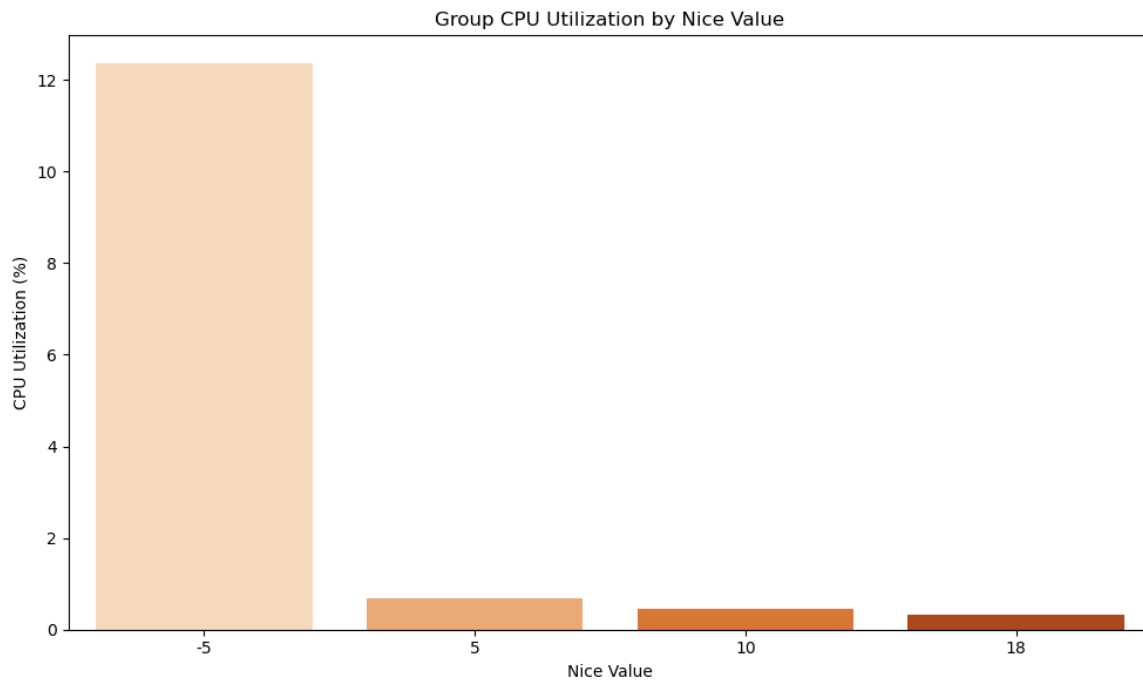


Figure 7.43 Ryzen 9 5950X EEVDF fairness CPU utilization
(source: own)

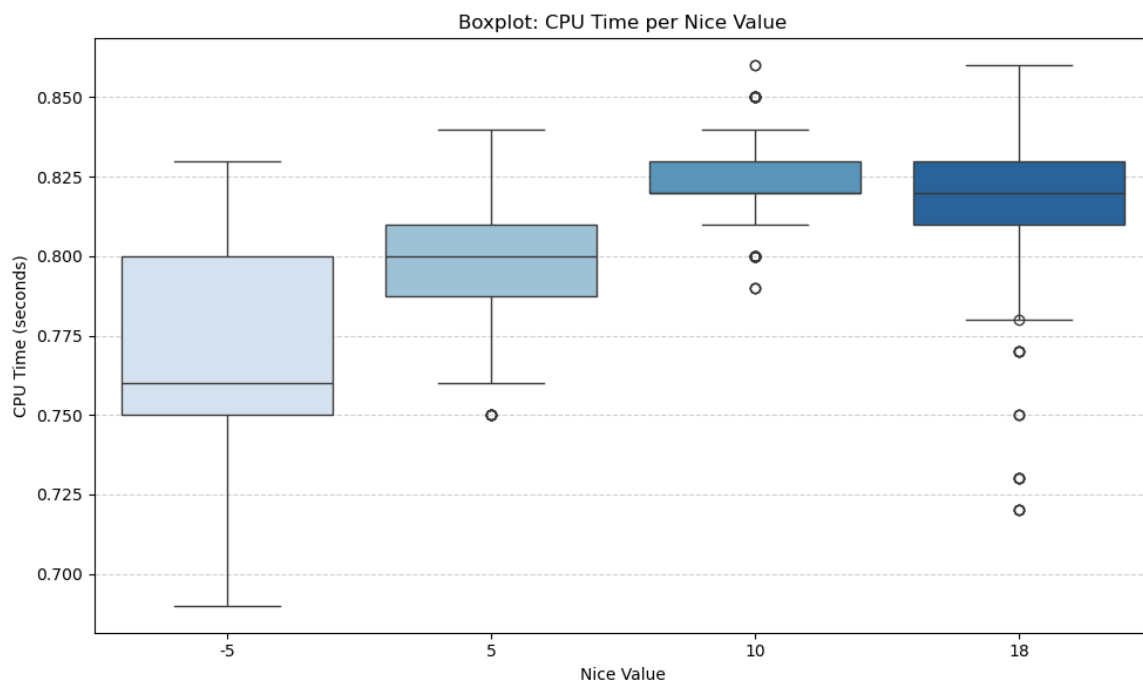


Figure 7.44 Ryzen 9 5950X EEVDF fairness CPU time
(source: own)

7.3.6 CFS throughput test result

CFS throughput performance on the Ryzen 9 5950X was evaluated following the same methodology outlined in 5.4. The resulting data are visualized in Figure 7.45. Detailed numerical results for each scenario are presented in Table 7.51.

Table 7.51 Ryzen 9 5950X CFS throughput results for all scenarios

Metric	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Description	30 s, no background	30 s, with background	100 s, no background	100 s, with background
Total Time [s]	30.0003	30.012	100.0004	100.0162
Total Events	2,740,530	1,654,890	9,123,572	5,527,875
Events per Second	91,348.93	55,139.68	91,235.00	55,269.56

(source: own)

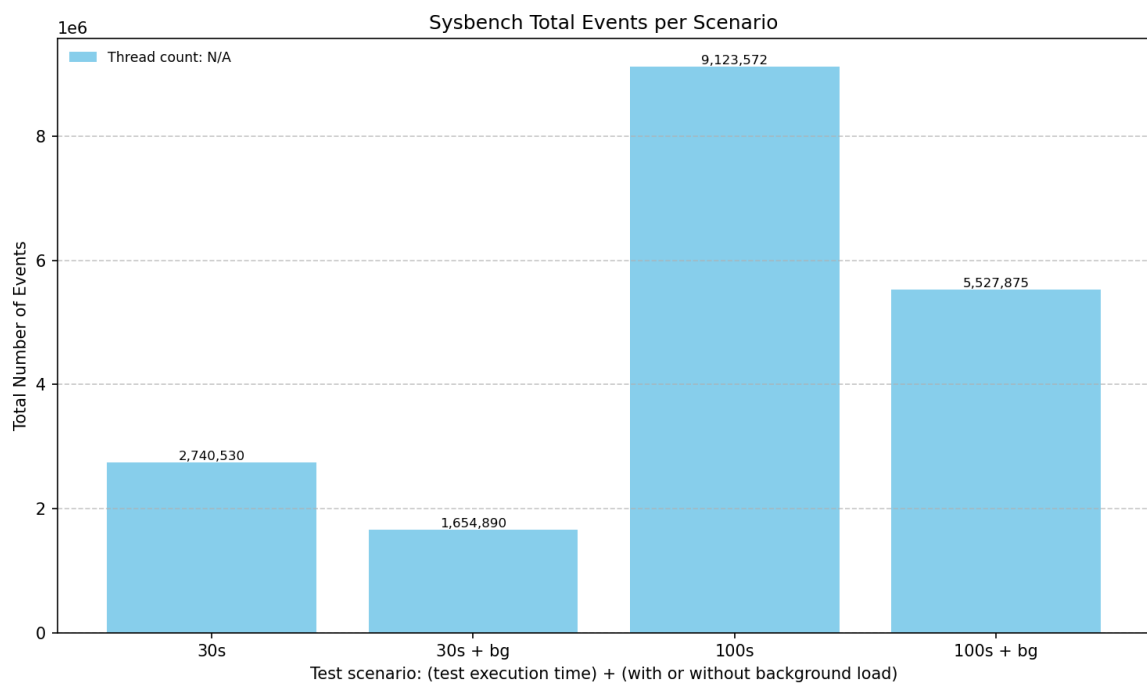


Figure 7.45 Ryzen 9 5950X CFS throughput

(source: own)

7.3.7 EEVDF throughput test result

The resulting data are visualized in Figure 7.46. Detailed numerical results for each scenario are presented in Table 7.52.

Table 7.52 Ryzen 9 5950X EEVDF throughput results for all scenarios

Metric	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Description	30 s, no background	30 s, with background	100 s, no background	100 s, with background
Total Time [s]	30.0003	30.0042	100.0004	100.0042
Total Events	2,766,596	1,753,440	9,195,635	5,697,721
Events per Second	92,217.45	58,438.58	91,955.59	56,974.56

(source: own)

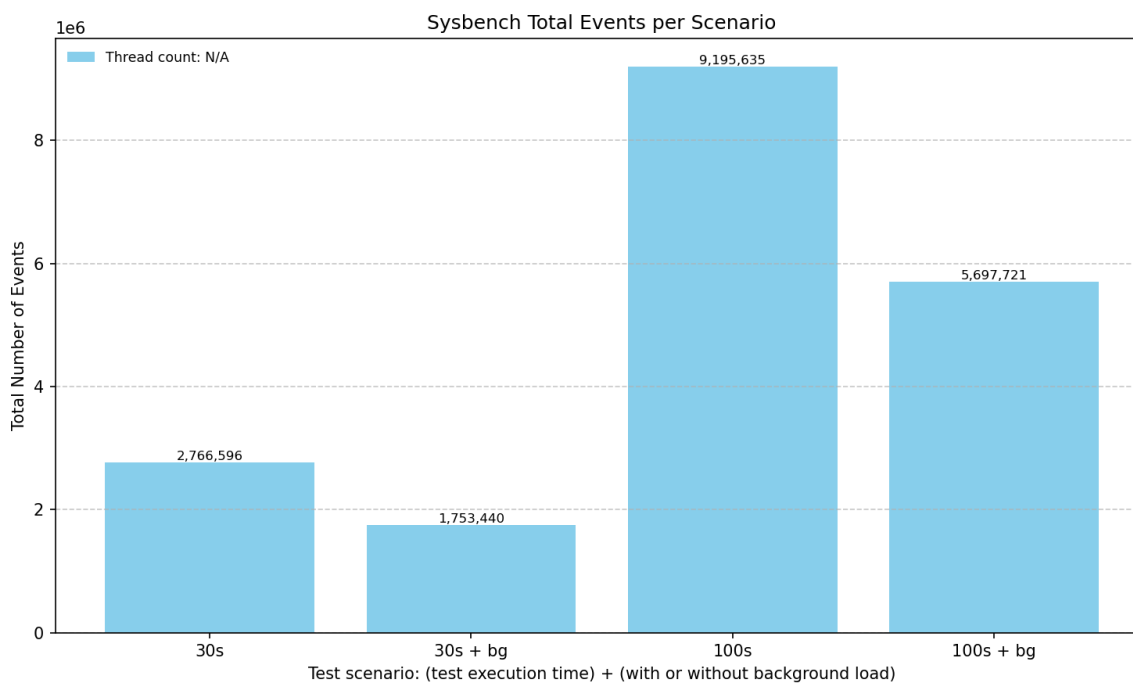


Figure 7.46 Ryzen 9 5950X EEVDF throughput
(source: own)

7.3.8 CFS context switching test result

The context switching test methodology described in 5.5. Figure 7.47 visualizes the amount of involuntary context switches per second by each thread. Table 7.54 presents the per-thread statistics for involuntary context switches per second (nvcswh/s). The cumulative metrics are summarized in Table 7.53.

Table 7.53 Ryzen 9 5950X CFS context switching summary

Metric	Value
Total [nvcswh/s]	1495.66
Average per Thread [nvcswh/s]	2.47
Peak Rate (Max nvcswh/s)	104.00
Std. Dev. of Per-Thread Averages	1.88
Thread Avg Max/Min Ratio	9.63

(source: own)

Table 7.54 Ryzen 9 5950X CFS per-thread context switching

Thread	Total [nvcswh/s]	Avg [nvcswh/s]	Thread	Total [nvcswh/s]	Avg [nvcswh/s]
T0	39.93	2.66	T16	17.93	1.28
T1	21.00	1.62	T17	31.00	2.21
T2	148.67	5.13	T18	22.93	1.64
T3	39.93	2.10	T19	32.93	2.06
T4	12.93	0.99	T20	40.93	2.41
T5	28.00	2.15	T21	25.93	2.16
T6	18.00	1.50	T22	31.93	1.52
T7	24.00	1.50	T23	19.93	1.53
T8	26.87	1.79	T24	48.54	2.55
T9	25.93	2.16	T25	172.00	8.19
T10	16.80	1.29	T26	33.87	1.61
T11	31.00	2.58	T27	287.48	9.58
T12	26.93	1.58	T28	32.93	2.53
T13	15.00	1.25	T29	115.87	4.14
T14	18.93	1.26	T30	29.00	1.93
T15	19.74	1.65	T31	38.80	2.59

(source: own)

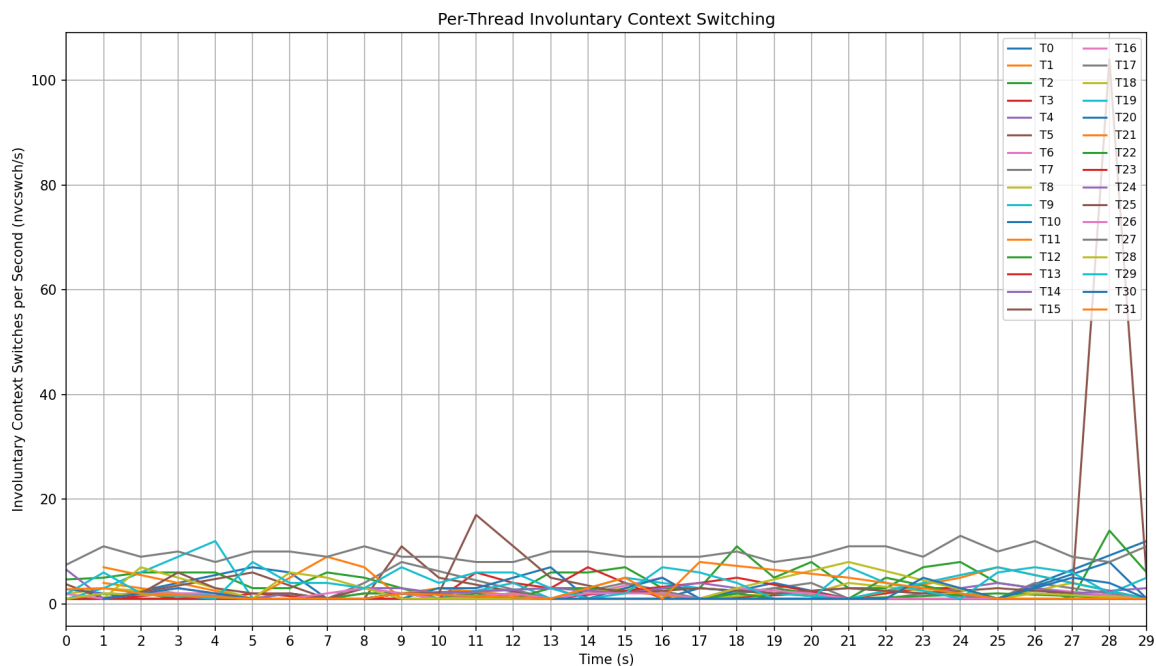


Figure 7.47 Ryzen 9 5950X CFS context switching
(source: own)

7.3.9 EEVDF context switching test result

The context switching test methodology described in 5.5. Figure 7.48 visualizes the amount of involuntary context switches per second by each thread. Table 7.56 presents the per-thread statistics for involuntary context switches per second (nvcschw/s). The cumulative metrics are summarized in Table 7.55.

Table 7.55 Ryzen 9 5950X EEVDF context switching summary

Metric	Value
Total [nvcschw/s]	1208.81
Average per Thread [nvcschw/s]	1.93
Peak Rate (Max nvcschw/s)	85.00
Std. Dev. of Per-Thread Averages	1.71
Thread Avg Max/Min Ratio	7.14

(source: own)

Table 7.56 Ryzen 9 5950X EEVDF per-thread context switching

Thread	Total [nvcswh/s]	Avg [nvcswh/s]	Thread	Total [nvcswh/s]	Avg [nvcswh/s]
T0	25.85	1.52	T16	40.93	2.56
T1	82.93	4.61	T17	9.93	0.99
T2	8.93	0.99	T18	8.93	0.99
T3	8.93	0.99	T19	44.93	1.50
T4	18.93	1.46	T20	9.93	0.99
T5	8.93	0.99	T21	29.93	1.30
T6	212.41	7.08	T22	9.85	1.09
T7	35.41	2.36	T23	9.93	0.99
T8	7.93	0.99	T24	10.93	1.09
T9	8.93	0.99	T25	8.93	0.99
T10	9.93	0.99	T26	8.93	0.99
T11	83.93	2.80	T27	8.93	0.99
T12	7.93	0.99	T28	8.93	0.99
T13	9.93	0.99	T29	199.93	6.66
T14	7.93	0.99	T30	134.26	5.16
T15	38.85	1.34	T31	85.93	4.30

(source: own)

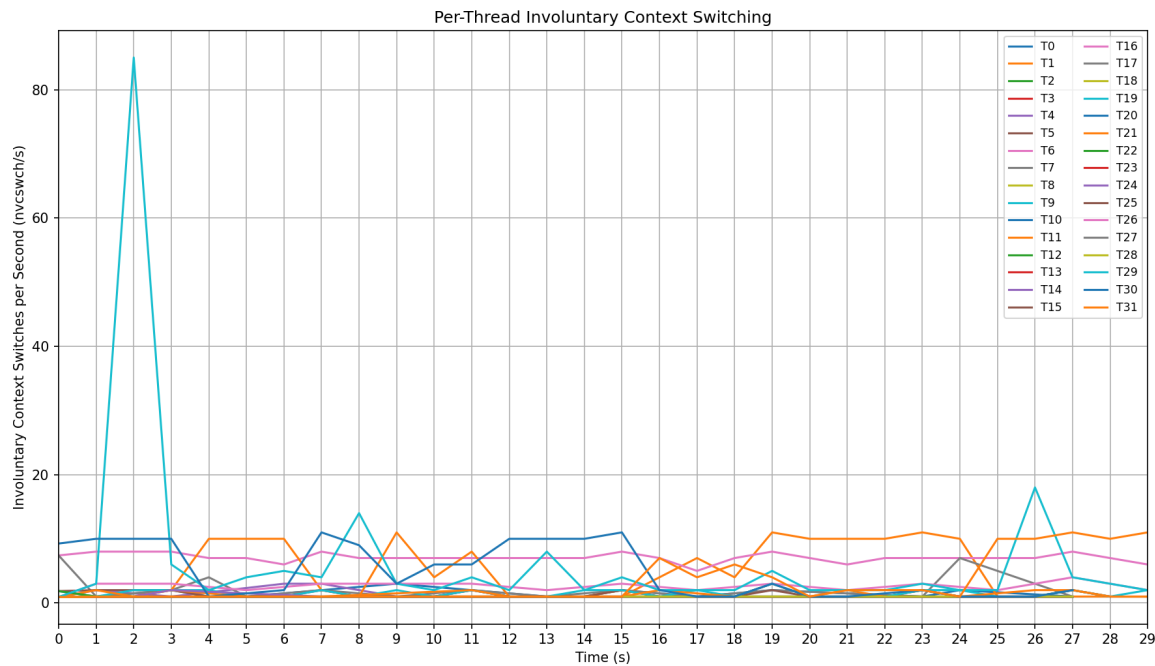


Figure 7.48 Ryzen 9 5950X EEVDF context switching
(source: own)

7.3.10 CFS scalability test result

Table 7.57 summarizes the results of the scalability test under the CFS scheduler. The corresponding visualization is shown in Figure 7.49, which illustrates the EPS across thread levels. Interpretation and analysis follow the methodology outlined in 7.1.10.

Table 7.57 Ryzen 9 5950X CFS scalability metrics

Threads	EPS	Total Events	EPS Gain [%]	Eff Drop [%]
1	5419.72	162,597	–	–
2	10824.30	324,735	99.7	0.1
4	21338.44	640,169	97.1	1.4
8	42215.59	1,266,498	97.8	1.1
16	81631.11	2,448,989	93.4	3.3
32	91516.08	2,745,556	12.1	43.9

(source: own)

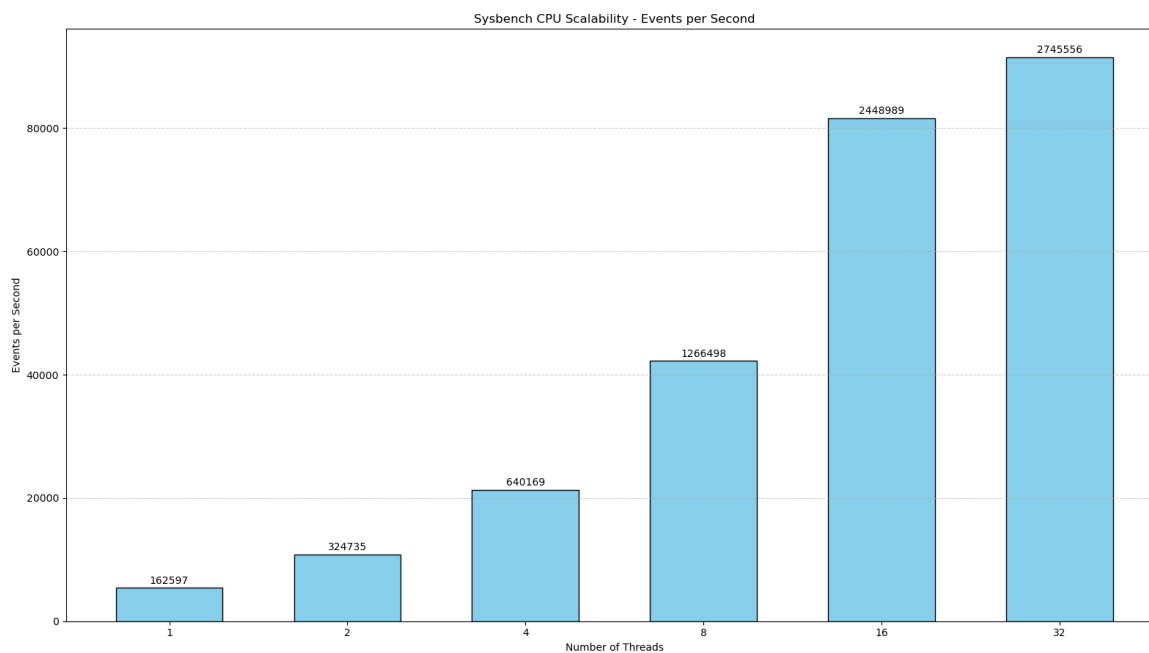


Figure 7.49 Ryzen 9 5950X CFS scalability

(source: own)

7.3.11 EEVDF scalability test result

Table 7.58 summarizes the results of the scalability test under the EEVDF scheduler. The corresponding visualization is shown in Figure 7.50, which illustrates the EPS across thread levels. Interpretation and analysis follow the methodology outlined in 7.1.10.

Table 7.58 Ryzen 9 5950X EEVDF scalability metrics

Threads	EPS	Total Events	EPS Gain [%]	Eff Drop [%]
1	5748.93	172,472	–	–
2	11437.52	343,134	99.0	0.5
4	21858.09	655,759	91.1	4.4
8	42875.07	1,286,279	96.2	1.9
16	82737.58	2,482,183	93.0	3.5
32	92239.14	2,767,248	11.5	44.3

(source: own)

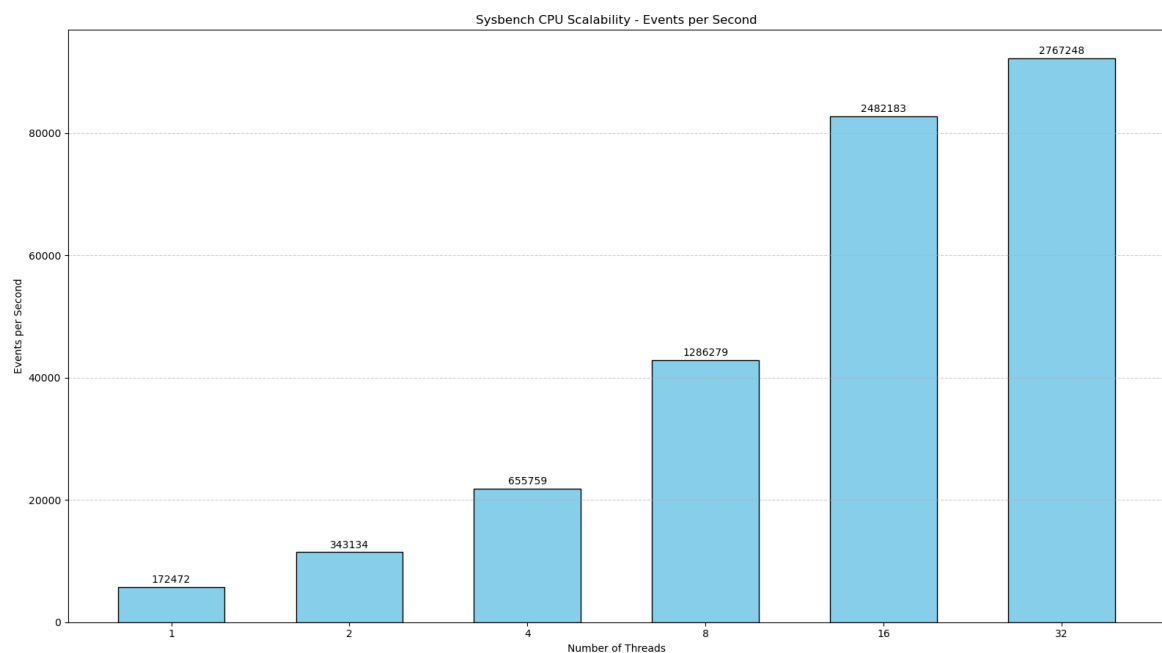


Figure 7.50 Ryzen 9 5950X EEVDF scalability

(source: own)

7.3.12 CFS starvation detection test result

Figure 7.51 illustrates the progression of thread activity over time. Figure 7.52 displays a box plot of CPU time distribution for threads grouped by nice value. Table 7.59 summarizes the CPU time statistics for both priority groups [43].

Table 7.59 Ryzen 9 5950X CFS starvation CPU time summary statistics

Nice	Samples	Avg [s]	Min [s]	Max [s]	Std Dev [s]
-10	512	1.090	0.550	1.700	0.212
19	16	0.619	0.550	0.680	0.050

(source: own)

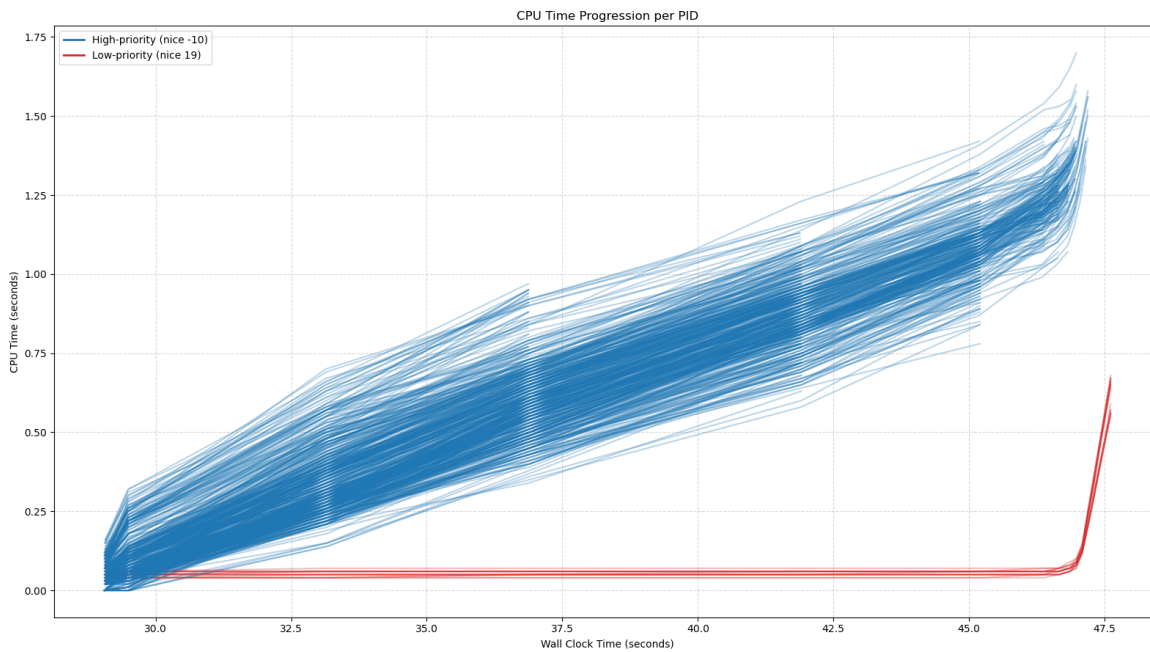


Figure 7.51 Ryzen 9 5950X CFS starvation thread life

(source: own)

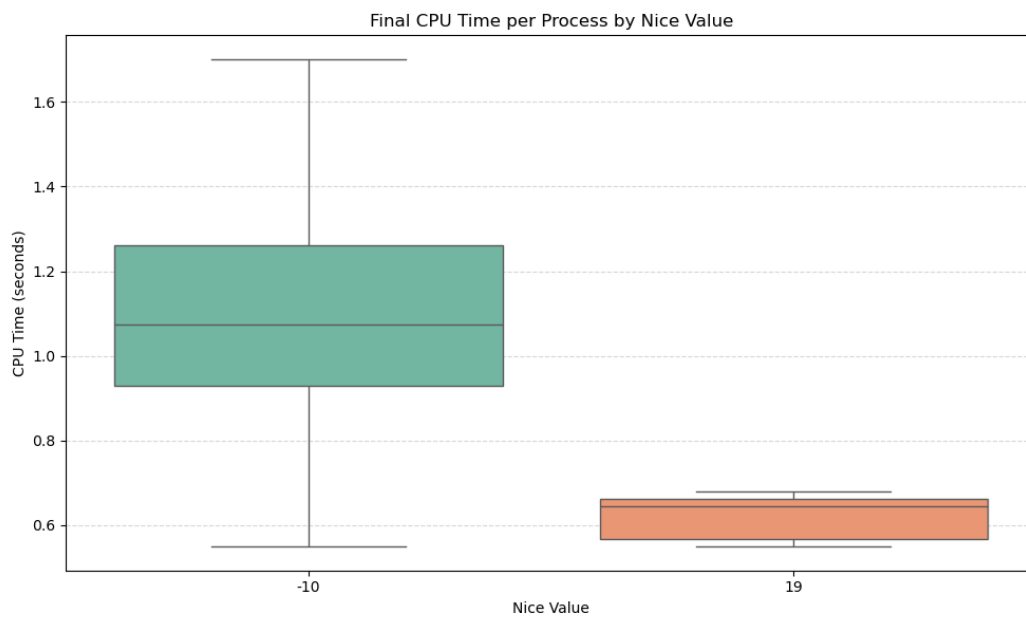


Figure 7.52 Ryzen 9 5950X CFS starvation CPU time

(source: own)

7.3.13 EEVDF starvation detection test result

Figure 7.53 illustrates the progression of thread activity over time. Table 7.60 summarizes the CPU time statistics for both priority groups.

Table 7.60 Ryzen 9 5950X EEVDF starvation CPU time summary statistics

Nice	Samples	Avg [s]	Min [s]	Max [s]	Std Dev [s]
-10	512	0.697	0.520	0.760	0.047
19	16	0.606	0.570	0.640	0.028

(source: own)

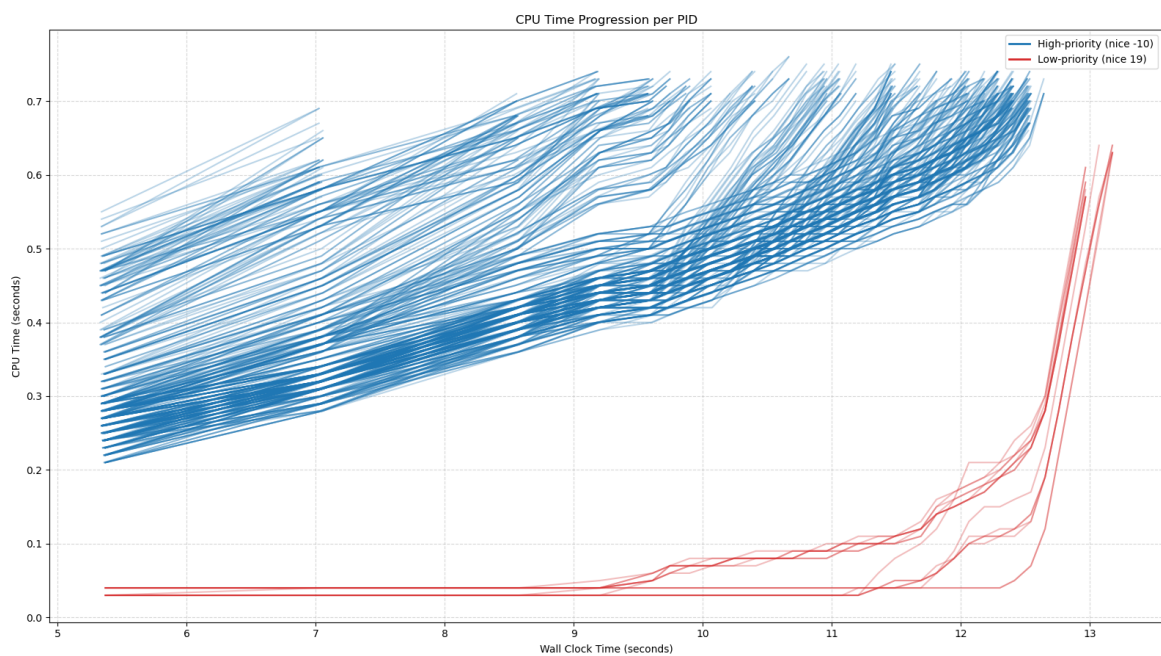


Figure 7.53 Ryzen 9 5950X EEVDF starvation thread life

(source: own)

8 TEST RESULTS EVALUATION

8.1 Test environment configuration

All process scheduler tests were conducted on three platforms: Intel i7-1260P detailed specification in 7.1.1, AMD Ryzen5 2600 specification in 7.2.1 and AMD Ryzen9 5950X details listed 7.3.1.

The kernel scheduler behavior was unified across all systems using a fixed set of tuning parameters, listed in Table 5.7. Ensuring consistent and reproducible behavior of both CFS and EEVDF across platforms. Linux testing environment image with kernel versions 6.1 (CFS) and 6.12 (EEVDF), configured as described in 6.2.

8.2 Latency

8.2.1 Intel Core i7-1260P

The Intel i7-1260P platform combines 8 efficiency cores and 4 performance cores, forming a hybrid design that introduces asymmetry in core performance. Latency testing from 7.1 revealed that the CFS scheduler delivers slightly lower average latency compared to EEVDF, with a +7.17% increase in average latency when using EEVDF.

Table 8.1 Intel i7-1260P latency comparison

Metric	CFS	EEVDF	Δ EEVDF vs CFS [%]
Min Latency [ns]	1766	1755	-0.62
Max Latency [ns]	2849568	2706936	-5.00
Avg Latency [ns]	10601.69	11362.19	+7.17

(source: own)

From the latency plot in Figure 7.2, the CFS test showed a sharp latency increase around iteration 20,000, followed by stabilization. This coincided with a spike in CPU temperature (>85 °C) and a drop in frequency for both performance and efficiency cores visible from the plot in Figure 7.3. The performance cores plateaued near 1.5 GHz, while efficiency cores dropped to 1.3 GHz, suggesting CPU frequency throttling played a major role in latency evolution [38].

In contrast, the EEVDF plot in Figure 7.4 showed a more stable trend with consistent latency increases across threads. Despite higher average latency, thread latency curves were more uniform, and temperature and frequency curves from the plot in Figure 7.3 indicate EEVDF's scheduling favored steady-state operation and fairness, at the cost of per-core reactivity.

8.2.2 AMD Ryzen 5 2600

The Ryzen 5 2600 is a 6-core processor with 12 logical CPUs due to SMT. In this case, testing from 7.2 discovered that both schedulers performed very similarly. Average latency under EEVDF was only +0.92% higher than CFS.

Table 8.2 Ryzen 5 2600 latency comparison

Metric	CFS	EEVDF	Δ EEVDF vs CFS [%]
Min Latency [ns]	3894	4357	+11.88
Max Latency [ns]	187010	145963	-21.95
Avg Latency [ns]	6731.75	6793.75	+0.92

(source: own)

Based on thermal and frequency plots for CFS in Figure 7.21 and EEVDF in Figure 7.23, the CPU maintained stable thermal conditions and frequency across the cores (about 3.6 GHz), with no observed throttling. The latency curves were consistent and tightly grouped across all threads, indicating scheduler-induced variations were minimal.

8.2.3 AMD Ryzen 9 5950X

Ryzen 9 5950X is a 16-core with 32 logical CPUs based on a homogeneous SMT design. Under this architecture, EEVDF slightly outperformed CFS in latency, with a reduction of -2.93% in average latency, based on test results from 7.3.

Table 8.3 Ryzen 9 5950X latency comparison

Metric	CFS	EEVDF	Δ EEVDF vs CFS [%]
Min Latency [ns]	2305	2554	+10.80
Max Latency [ns]	110196	82852	-24.79
Avg Latency [ns]	5228.59	5075.19	-2.93

(source: own)

The EEVDF latency plot in Figure 7.40 shows tight clustering of thread latencies, with minimal thread divergence. The CFS plot from Figure 7.38 displays slightly more dispersion among thread latencies but remains within a narrow band, indicating consistent performance.

Temperature and frequency plots Figures 7.39 and 7.40 confirm excellent thermal stability and consistent operating frequencies at an average of 4.34 GHz for all cores. This enables EEVDF to leverage fairness benefits without suffering penalties, unlike on thermally limited mobile platforms.

8.2.4 Cross-platform Latency comparison

Table 8.4 compares average latencies measured across all test platforms under both the CFS and EEVDF schedulers. This comparison highlights how scheduling behavior varies depending on system architecture. While EEVDF shows a latency improvement on the Ryzen 9 5950X, it results in slightly higher average latencies on the Intel i7-1260P and Ryzen 5 2600. These differences are essential for interpreting platform-specific scheduler trade-offs under uniform test loads.

Table 8.4 Cross-platform average latency comparison

Platform	CFS Avg [ns]	EEVDF Avg [ns]	Δ EEVDF vs CFS [%]
Intel i7-1260P	10601.69	11362.19	+7.17
Ryzen 9 5950X	5228.59	5075.19	-2.93
Ryzen 5 2600	6731.75	6793.75	+0.92

(source: own)

Table 8.5 Cross-platform maximum latency comparison

Platform	CFS Max [ns]	EEVDF Max [ns]	Δ EEVDF vs CFS [%]
Intel i7-1260P	2849568	2706936	-5.01
Ryzen 9 5950X	110196	82852	-24.79
Ryzen 5 2600	187010	145963	-21.95

(source: own)

Comparison from Table 8.5 enables a direct evaluation of the Hypothesis 1 stated in 4.3, which suggests: *EEVDF is expected to reduce maximum latency under interactive workloads with comparison to CFS.*

As shown in Table 8.5, EEVDF consistently reduces maximum latency across all evaluated platforms: by 5% on the Intel i7-1260P, 25% on the Ryzen 9 5950X, and nearly 22% on the Ryzen 5 2600. This confirms Hypothesis 1, demonstrating that EEVDF fulfills its goal of reducing worst case latency in time sensitive workloads compared to CFS.

8.3 Fairness

8.3.1 Intel i7-1260P

EEVDF demonstrates clearer enforcement of scheduling weights compared to CFS. The average CPU time among nice groups under EEVDF in Table 7.11 is closely distributed (0.45 to 0.50 s), CFS in Table 7.9 which shows increasing values with decreasing priority (0.49 to 0.83 s). CPU efficiency for high-priority processes improves substantially from 9.58% (CFS) to 31.64% (EEVDF), and CPU utilization jumps from 81.86% to 158.45% in favor of EEVDF listed in Table 7.8 and 7.10.

Figures 7.6 and 7.8 show a redistribution of CPU time that more strongly favors nice = -5 under EEVDF, confirming a bias toward responsiveness. Figure 7.9 reveals reduced variance in CPU times per group, supporting a more deterministic or fair behavior. Comparison summarized by Table 8.6.

Table 8.6 Intel i7-1260P summary of average CPU time per nice level

Nice Level	CFS Avg CPU t [s]	EEVDF Avg CPU t [s]	Δ [%]
-5	0.49	0.45	-8.16
5	0.71	0.48	-32.39
10	0.85	0.50	-41.18
18	0.83	0.49	-40.96

(source: own)

8.3.2 Ryzen 5 2600

CFS results (Table 7.29) show a drop in CPU efficiency from 8.10% to 1.99% across the nice scale. EEVDF, however, maintains tighter CPU time distribution, with higher overall CPU utilization (195.80% for nice = -5 vs. 83.11% for CFS) and increased efficiency (20.75% vs. 8.10%).

The visualization in Figures 7.26 and 7.27 indicates a more proportional and predictable distribution compared to CFS, especially in terms of lower variance in CPU time allocation.

Table 8.7 Ryzen 5 2600 summary of average CPU time per nice level

Nice Level	CFS Avg CPU t [s]	EEVDF Avg CPU t [s]	Δ [%]
−5	0.43	1.01	+134.88
5	0.61	1.07	+75.41
10	0.71	1.07	+50.70
18	0.69	1.08	+56.52

(source: own)

8.3.3 Ryzen 9 5950X

EEVDF provides significant fairness gains on this high-end platform shown in summary Table 8.8. The Table 7.50 reports very tight average CPU times across all nice levels (0.77 to 0.82 s), compared to wider variation under CFS (1.08 to 0.84 s) in Table 7.48. The improvement in CPU efficiency for nice = −5 is dramatic from 82.74% to 588.58% listed in Table 7.47.

CPU utilization also reflects this, increasing from 272.38% (CFS) to 1234.90% (EEVDF) listed in Table 7.49. This suggests that EEVDF aggressively prioritizes the high-priority group, confirming its virtual deadline model’s responsiveness to process weight (details in 3.3).

Table 8.8 Ryzen 9 5950X summary of average CPU time per nice level

Nice Level	CFS Avg CPU t [s]	EEVDF Avg CPU t [s]	Δ [%]
−5	1.08	0.77	-28.70
5	0.83	0.80	-3.61
10	0.83	0.82	-1.20
18	0.84	0.82	-2.38

(source: own)

8.3.4 Cross-platform Fairness Comparison

To compare fairness more clearly across platforms, Table 8.9 summarizes CPU utilization for nice = −5 processes, as a proxy for how strongly each scheduler enforces priority.

Table 8.9 CPU Utilization comparison for nice = -5 group across platforms

Platform	CFS Util. [%]	EEVDF Util. [%]	EEVDF Δ vs. CFS [%]
Intel i7-1260P	81.86	158.45	+93.57
Ryzen 5 2600	83.11	195.80	+135.52
Ryzen 9 5950X	272.38	1234.90	+362.99

(source: own)

EEVDF consistently improves fairness by delivering more CPU time to high-priority groups. The highest gain is observed on the Ryzen 9 system, where EEVDF allocates over four times the CPU time to high-priority tasks than CFS.

These results confirm Hypothesis 3 from 4.3, which states that *EEVDF should better preserve fairness in CPU distribution among diverse workloads*. The virtual deadline mechanism more accurately tracks task eligibility and improves responsiveness across all systems.

Therefore, EEVDF outperforms CFS in fair CPU distribution, especially in multi-threaded environments with varying priority weights.

8.4 Throughput

8.4.1 Intel i7-1260P

CFS overall achieves higher throughput as is shown in Table 8.10 throughput drops from 2.3 M to 1.1 M events when background tasks are added to the 100 second run. EEVDF also decreases throughput under background stress, delivering fewer events overall. Comparison is based on throughput test results from 7.1.

Table 8.10 i7-1260P Throughput Comparison

Scenario	CFS Events	EEVDF Events	EEVDF Δ vs. CFS [%]
30s, no background	895,439	642,583	-28.26
30s, with background	358,066	292,290	-18.37
100s, no background	2,299,029	2,051,442	-10.77
100s, with background	1,127,333	995,912	-11.65

(source: own)

8.4.2 Ryzen 5 2600

On Ryzen 5, both schedulers perform closely in terms of total events, with CFS slightly outperforming EEVDF across most scenarios. Table 8.11 shows that CFS gains more events overall. The gap widens under background stress, suggesting that CFS handles background interference with better throughput preservation. Comparison is based on throughput test results from 7.2.

Table 8.11 Ryzen 5 2600 Throughput Comparison

Scenario	CFS Events	EEVDF Events	EEVDF Δ vs. CFS [%]
30s, no background	348,342	346,105	-0.64
30s, with background	240,082	238,789	-0.54
100s, no background	1,154,087	1,153,472	-0.05
100s, with background	831,039	794,061	-4.45

(source: own)

8.4.3 Ryzen 9 5950X

Contrary to the trend on lower-core systems, the Ryzen 9 platform shows EEVDF delivering consistently higher throughput across all test scenarios. As displayed in Table 8.12. EEVDF surpasses CFS, especially under background load scenarios. Comparison is based on test results from 7.3.

Table 8.12 Ryzen 9 5950X Throughput Comparison

Scenario	CFS Events	EEVDF Events	EEVDF Δ vs. CFS [%]
30s, no background	2,740,530	2,766,596	+0.95
30s, with background	1,654,890	1,753,440	+5.96
100s, no background	9,123,572	9,195,635	+0.79
100s, with background	5,527,875	5,697,721	+3.07

(source: own)

8.4.4 Cross-platform Throughput Comparison

Cross-platform throughput trend, summarized by Table 8.13 presents the relative difference in total events completed across all scenarios. CFS marginally outperforms EEVDF on lower-core systems (i7-1260P, Ryzen 5), supporting Hypothesis 5 from 4.3. However, on

the Ryzen 9 platform, EEVDF consistently leads in all cases, indicating better scalability. This finding aligns with Hypothesis 4.

Table 8.13 Aggregate throughput difference: total events (CFS vs. EEVDF)

Platform	CFS Total Events	EEVDF Total Events	Δ [%]
Intel i7-1260P	4,680,867	3,982,227	-14.92
Ryzen 5 2600	2,573,550	2,532,427	-1.60
Ryzen 9 5950X	19,046,867	19,413,392	+1.92

(source: own)

These results confirm the Hypothesis 5: *CFS may provide slightly higher throughput on uniform, computation intensive workloads*. EEVDF, while trailing on simpler platforms, demonstrates strong scalability and potential advantages on high-core systems.

8.5 Context Switches

8.5.1 Intel i7-1260P Platform

The i7-1260P shows modest total context switching activity under both schedulers, with CFS generating slightly higher overall switches. As summarized in Table 8.14. EEVDF shows a more balanced per-thread distribution, confirmed by a lower standard deviation and max/min thread ratio. Comparison is based on test results from 7.1.

Table 8.14 Intel i7-1260P Context Switching Summary Comparison

Metric	CFS	EEVDF	EEVDF Δ vs. CFS [%]
Total [nvcswh/s]	2538.10	2256.93	-11.08
Avg per Thread [nvcswh/s]	6.49	5.94	-8.48
Peak Rate (Max per-second nvcswh)	93.00	99.05	+6.51
Std. Dev. of Per-Thread Averages	4.85	3.87	-20.21
Thread Max/Min Ratio	9.03	7.02	-22.26

(source: own)

8.5.2 Ryzen 5 2600 Platform

CFS produces slightly more context switches than EEVDF on the Ryzen 5 2600. As seen in Table 8.15, CFS has a total of 1152.16 nvcswh/s, while EEVDF has 929.41 nvcswh/s,

marking a reduction of 19.37%. However, EEVDF shows higher peak context switching and greater variability in per-thread switching (as indicated by a higher standard deviation and max/min ratio), suggesting uneven preemption pressure under EEVDF. Comparison is based on test results from 7.2.

Table 8.15 Ryzen 5 2600 Context Switching Summary Comparison

Metric	CFS	EEVDF	EEVDF Δ vs. CFS [%]
Total [nvcswh/s]	1152.16	929.41	-19.37
Avg per Thread [nvcswh/s]	3.92	3.24	-17.35
Peak Rate	21.00	93.00	+342.86
Std. Dev.	1.83	2.68	+46.45
Thread Max/Min Ratio	4.53	8.15	+79.91

(source: own)

8.5.3 Ryzen 9 5950X Platform

On this high-core system, EEVDF reduces total context switching activity (Table 8.16). EEVDF records 1208.81 nvcswh/s — 19.2% lower than CFS (1495.66 nvcswh/s). The decrease is also seen in the standard deviation and per-thread ratios, suggesting that EEVDF performs more uniform scheduling across cores with less contention. Comparison is based on test results from 7.3.

Table 8.16 Ryzen 9 5950X Context Switching Summary Comparison

Metric	CFS	EEVDF	EEVDF Δ vs. CFS [%]
Total [nvcswh/s]	1495.66	1208.81	-19.21
Avg per Thread [nvcswh/s]	2.47	1.93	-21.86
Peak Rate (Max per-second nvcswh)	104.00	85.00	-18.27
Std. Dev. of Per-Thread Averages	1.88	1.71	-9.04
Thread Max/Min Ratio	9.63	7.14	-25.87

(source: own)

8.5.4 Cross-platform Context Switch Comparison

Table 8.17 aggregates total context switching across all platforms. While EEVDF does not consistently increase context switching frequency, its spike behavior (notably seen in Ryzen 5) aligns with its preemptive design, supporting Hypothesis 2 only partly.

Table 8.17 Total Involuntary Context Switches Across Platforms

Platform	CFS Total [nvcswch/s]	EEVDF Total [nvcswch/s]	Δ [%]
Intel i7-1260P	2538.10	2256.93	-11.08
Ryzen 5 2600	1152.16	929.41	-19.37
Ryzen 9 5950X	1495.66	1208.81	-19.21

(source: own)

In conclusion, while EEVDF was hypothesized to cause more frequent context switches, this is not uniformly supported by measured data. Instead, its behavior varies by platform, showing fewer but more volatile switches indicating targeted, aggressive preemption during load spikes rather than continual over-activity.

8.6 Scalability

8.6.1 Intel i7-1260P

Figure 7.14 and Table 7.18 show that under CFS, EPS increases steadily up to 8 threads, then begins to flatten with only a +33% gain from 8 to 16 threads. Figure 7.15 and Table 7.19 show that EEVDF performs similarly, but exhibits a smoother efficiency decline, suggesting better scheduling uniformity under load. Comparison is based on test results from 7.1.

Table 8.18 Intel i7-1260P scalability comparison

Threads	CFS EPS	EEVDF EPS	CFS Eff Drop [%]	EEVDF Eff Drop [%]
1	3994.81	3794.22	–	–
2	6582.46	6449.54	17.6	15.0
4	10547.57	10198.01	19.9	20.9
8	17177.56	15724.12	18.6	22.9
16	22844.16	20750.61	33.5	34.0

(source: own)

8.6.2 Ryzen 5 2600

CFS scalability is shown by Figure 7.32 and listed in Table 7.38 and EEVDF Figure 7.33, Table 7.39 scales well up to 4 threads. After 8 threads, gains stagnate and efficiency drops significantly over 30% at 12 threads. Both schedulers show nearly identical performance,

highlighting platform limits over scheduling design. Comparison is based on test results from 7.2.

Table 8.19 Ryzen 5 2600 scalability comparison

Threads	CFS EPS	EEVDF EPS	CFS Eff Drop [%]	EEVDF Eff Drop [%]
1	1911.28	1920.62	–	–
2	3852.36	3850.74	-0.8	-0.2
4	7555.16	7644.28	1.9	0.7
8	11155.25	11175.93	26.2	26.9
12	11544.79	11586.13	31.0	30.9

(source: own)

8.6.3 Ryzen 9 5950X

This 32 logical CPU platform exhibits excellent scaling. CFS results in Figure 7.49 and Table 7.57, and EEVDF in Figure 7.50 and Table 7.58. Both process schedulers maintain strong EPS gains up to 16 threads, with a sharp drop in per-thread efficiency from 16 to 32 threads. EEVDF shows marginally better throughput and a smoother decline. Comparison is based on test results from 7.3.

Table 8.20 Ryzen 9 5950X scalability comparison (CFS vs. EEVDF)

Threads	CFS EPS	EEVDF EPS	CFS Eff Drop [%]	EEVDF Eff Drop [%]
1	5419.72	5748.93	–	–
2	10824.30	11437.52	0.1	0.5
4	21338.44	21858.09	1.4	4.4
8	42215.59	42875.07	1.1	1.9
16	81631.11	82737.58	3.3	3.5
32	91516.08	92239.14	43.9	44.3

(source: own)

8.6.4 Cross-platform Scalability Comparison

Table 8.21 compares maximum throughput (EPS), efficiency drop, and thread scaling across platforms. The last column shows the EPS delta between EEVDF and CFS at maximum thread count.

Ryzen 9 5950X shows the highest scalability, where EEVDF outperforms CFS slightly in throughput while maintaining similar efficiency. On Ryzen 5, both schedulers behave almost

Table 8.21 Cross-platform scalability summary

Platform	Scheduler	Max EPS	Eff Drop [%]	Threads	Δ EPS [%]
Intel i7-1260P	CFS	22844.16	33.5	16	–
	EEVDF	20750.61	34.0	16	-9.17
Ryzen 5 2600	CFS	11544.79	31.0	12	–
	EEVDF	11586.13	30.9	12	+0.36
Ryzen 9 5950X	CFS	91516.08	43.9	32	–
	EEVDF	92239.14	44.3	32	+0.79

(source: own)

identically, suggesting hardware limitation. Intel i7-1260P sees a noticeable drop under EEVDF, likely due to a combination of hybrid-core scheduling complexity and thermal limits. Additionally, energy-aware scheduling was disabled during the test, which may have prevented optimal task placement between performance and efficiency cores details in 8.1.

These findings support Hypothesis 4, showing that EEVDF maintains or slightly improves scalability on multi-core systems.

8.7 Starvation Detection

8.7.1 Intel i7-1260P

On the i7-1260P, CFS gives equal CPU time to all threads. Table 7.20 shows almost the same average time for both priority levels. Figure 7.16 confirms smooth, parallel execution. But it takes about 25 seconds to finish, showing how CFS prioritizes fairness and task priority over speed and throughput.

EEVDF low-priority threads begin receiving CPU time after approximately 2 seconds of time visible in Figure 7.18, while high-priority tasks are still running. This behavior is mainly due to EEVDF's combination of earliest eligibility and weight-based scheduling. Instead of strictly finishing high-priority tasks first as seen in CFS, EEVDF gradually allows low-priority threads to progress in parallel. This overlap improves CPU utilization and leads to faster total completion in under 9 seconds. As shown in Table 7.21 and 8.22, EEVDF low-priority threads receive more CPU time overall, but from extended execution following their delayed but parallel start.

Comparison is based on test results from 7.1.

Table 8.22 Intel i7-1260P Starvation Detection Comparison

Scheduler	nice level	Avg CPU Time [s]	Std Dev [s]
CFS	-10	0.820	0.254
	19	0.816	0.040
EEVDF	-10	0.678	0.175
	19	1.303	0.049

(source: own)

8.7.2 Ryzen 5 2600

On the Ryzen 5, both schedulers show similar behavior. As visible in Figures 7.34 and 7.36, low-priority threads remain inactive for most of the run and only start progressing near the end after the majority of high-priority threads are completed.

This results in long execution times around 34 s for CFS and 30 s for EEVDF and delayed service for low-priority tasks in both cases. Table 8.23 confirms close average CPU time across groups under EEVDF, but this fairness comes with late access. The earlier mixing observed on Intel is not seen here. Comparison is based on test results from 7.2.

Table 8.23 Ryzen 5 2600 Starvation Detection Comparison

Scheduler	Nice Level	Avg CPU Time [s]	Std Dev [s]
CFS	-10	1.273	0.081
	19	0.987	0.109
EEVDF	-10	0.943	0.075
	19	0.936	0.078

(source: own)

8.7.3 Ryzen 9 5950X

CFS on Ryzen 9 delays low-priority thread execution significantly. Figure 7.51 shows low-priority threads are inactive until high-priority ones nearly complete. This results in a long total test duration of about 48 seconds. Table 8.24 reflects this imbalance in average CPU time.

EEVDF performs similarly to the Intel test. Figure 7.53 shows low-priority threads begin around 9 seconds in, while high-priority threads are still active. The test finishes in about 13 seconds. Table 8.24 shows both groups receive comparable CPU time with lower variance. This shows that EEVDF enables early low-priority execution and faster test completion by overlapping group progress. Comparison is based on test results from 7.3.

Table 8.24 Ryzen 9 5950X Starvation Detection Comparison

Scheduler	nice level	Avg CPU Time [s]	Std Dev [s]
CFS	-10	1.090	0.212
	19	0.619	0.050
EEVDF	-10	0.697	0.047
	19	0.606	0.028

(source: own)

8.7.4 Cross-platform Starvation Comparison

Table 8.25 compares average CPU times across all systems. CFS avoids hard starvation, but introduces long runtime and delayed low-priority access. EEVDF provides more efficient CPU use by overlapping group execution. On Intel and Ryzen 9, EEVDF enables lower priority threads to start earlier and finish faster, leading to reduced total runtime. Ryzen 5 shows less overlap regardless of scheduler, suggesting core count impacts fairness and execution parallelism.

Ryzen 5 shows less overlap regardless of scheduler, suggesting that fewer cores limit how tasks can run in parallel. Both CFS and EEVDF need more idle CPUs to split work fairly or prioritize tasks effectively. With only 6 cores (12 logical CPUs via SMT), Ryzen 5 does not give EEVDF enough room to apply its early-deadline and weight logic fully. As a result, low-priority threads start late under both schedulers, and the behavior looks almost the same [4, 44].

These findings support Hypothesis 6, showing that both schedulers maintained a non-zero level of CPU time for low-priority threads visible from thread life figures. On top of that, EEVDF showed higher responsiveness and efficiency by enabling earlier low-priority thread execution and reducing overall test duration through improved CPU utilization and group overlap.

Table 8.25 Cross-platform starvation test: average CPU time [s] by nice value

Platform	Scheduler	nice level -10	nice level 19
Intel i7-1260P	CFS	0.820	0.816
	EEVDF	0.678	1.303
Ryzen 5 2600	CFS	1.273	0.987
	EEVDF	0.943	0.936
Ryzen 9 5950X	CFS	1.090	0.619
	EEVDF	0.697	0.606

(source: own)

CONCLUSION

The main goal of this thesis was to design and implement a methodology for evaluating the performance of Linux process schedulers. By using a minimal Linux environment and a consistent set of kernel settings (as described in 8.1), the approach ensured fair and repeatable comparisons across different hardware platforms and scheduler versions.

Chapters 2 and 3 laid out the process scheduling theoretical background and evolution of Linux schedulers, with a focus on the Completely Fair Scheduler and the newer Earliest Eligible Virtual Deadline First algorithm. These two schedulers were evaluated under a series of tests measuring latency, throughput, fairness, context switching, scalability, and starvation detection, as implemented in the testing methodology described in Chapter 5.

Experiments were executed on three representative processor platforms: Intel Core i7-1260P, AMD Ryzen 5 2600, and AMD Ryzen 9 5950X. Chosen to reflect hybrid-core, mid-range, and high-performance multicore architectures. Results of the experiments are described in Chapter 7.

To provide deeper context and measurable targets, Chapter 4 formulated concrete evaluation hypotheses regarding latency, throughput, fairness, context switching behavior, scalability, and starvation prevention. These guided the measurements analysis conducted in Chapter 8.

Across these platforms, it became evident that scheduler performance is deeply influenced by the characteristics of the underlying hardware. On the Intel Core i7-1260P, which features a hybrid architecture, EEVDF demonstrated stronger fairness and latency ceiling control compared to CFS. However, its average latency was slightly worse, and throughput under stress scenarios dropped by nearly 15%. These results are best interpreted in the context of the platform's thermal design. The observed throttling events during latency tests suggest that the scheduler's ability to maintain fair, steady-state operation comes at the cost of short-term responsiveness when thermal constraints begin to limit processor performance. Moreover, the hybrid core topology introduces asymmetric scheduling complexity that CFS, with its simpler model, handles more predictably under load.

In contrast, the Ryzen 5 2600, a conventional 6-core processor with symmetrical SMT, showed minimal divergence between CFS and EEVDF in most metrics. Average latencies, throughput and context switch rates stayed closely matched, with EEVDF offering modest gains in fairness. The absence of thermal throttling and the relatively small core count created a bounded environment where the additional scheduling sophistication of EEVDF could not fully express its advantages. Interestingly, the improved fairness also caused more variation in scheduling. This was seen as bursts of context switches and suggests that in systems with limited resources, EEVDF's scheduling may become less predictable, especially when many tasks are competing for the same cores.

The best results for EEVDF were recorded on the Ryzen 9 5950X, a high performance platform with 16 physical cores and substantial thermal margin. Here, EEVDF not only improved average and maximum latency but also outperformed CFS in throughput. Its fairness model gave balanced CPU time distribution across priority levels, while reducing starvation and enabling earlier low-priority thread execution. Even context switching was more efficient and uniform, confirming that the design of EEVDF aligns well with wide-core symmetric architectures. EEVDF scales effectively with high thread counts and performs best on architectures without thermal limitations.

These observations collectively support the conclusion that while CFS remains robust and efficient on simpler or thermally limited platforms, EEVDF demonstrates clear advantages on multicore systems with high thread availability and stable thermal behavior.

The testing methodology developed in this thesis offers a lightweight, fast, and portable foundation for benchmarking Linux process schedulers. Designed to run with minimal system overhead, it supports consistent evaluation across platforms and can be reused for analyzing new scheduling algorithms, tuning kernel behavior, or conducting regression tests. Its simplicity makes it especially useful for practical experimentation in both research and development settings, and it may assist future work in adapting scheduling strategies to evolving hardware and workload demands.

REFERENCES

- [1] BHARADWAJ, Raghu, 2017. *Mastering Linux kernel development: a kernel developer's reference manual*. Birmingham: Packt. ISBN 978-1-78588-305-7.
- [2] LOVE, Robert, 2010. *Linux Kernel Development*. Upper Saddle River: Addison-Wesley. 3rd edition. ISBN 978-0-672-32946-3.
- [3] GREGG, Brendan, 2021. *Systems Performance: Enterprise and the Cloud*. Addison-Wesley professional computing series. Boston: Addison-Wesley. 2nd edition. ISBN 978-0-13-682015-4.
- [4] THE KERNEL DEVELOPMENT COMMUNITY, 2024. *The Linux Kernel documentation*. Online. Available from: <https://docs.kernel.org/scheduler/index.html>. [cit. 2025-03-01].
- [5] VOZÁR, Jiří, 2019. *Performance testing of linux kernel scheduler*. Online, Bachelor's thesis. Brno University of Technology. Available from: https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=197220. [cit. 2025-03-01].
- [6] RED HAT, INC., 2025. *Performance Tuning Guide*. Online. Red Hat Documentation. Available from: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/performance_tuning_guide. [cit. 2025-03-01].
- [7] STOICA, Ion and HUSSEIN, Abdel-Wahab, 1996. *Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*. Online. CS Dpt., Old Dominion Univ. TR-95-22. Available from: <https://citeseerx.ist.psu.edu/document?doi=805acf7726282721504c8f00575d91ebfd750564>. [cit. 2025-03-01].
- [8] EKLEKTIX, INC., 2025. *Kernel index*. Online. LWN.net. Available from: <https://lwn.net/Kernel/Index/>. [cit. 2025-03-01].
- [9] ROSENDAHL, Viktor, 2025. *Traceshark*. Online. Available from: <https://github.com/cunctator/traceshark>. [cit. 2025-03-01].
- [10] BRUCKER, Peter, 2007. *Scheduling Algorithms*. Springer: Berlin. 5th edition. ISBN 978-3-540-69515-8.

- [11] PINEDO, Michael L., 2016. *Scheduling Theory, Algorithms, and Systems*. Springer International Publishing AG Switzerland. 5th edition. ISBN 978-3-319-26580-3.
- [12] LEUNG, Joseph Y-T., 2004. *Handbook of SCHEDULING: Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC computer and information science series. Boca Raton: Chapman & Hall/CRC. ISBN 1-58488-397-9.
- [13] WERNER, Frank; BURTSEVA, Larysa and SOTSKOV, Yuri, 2018. *Algorithms for Scheduling Problems*. MDPI, Basel, Switzerland. 5th edition. ISBN 978-3-03897-120-7.
- [14] SHEN, John Paul and LIPASTI, Mikko H., 2005. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, Inc. ISBN 978-1-4786-0783-0.
- [15] INTEL CORPORATION, 2025. *Performance Hybrid Architecture*. Online. Available from: <https://www.intel.com/content/www/us/en/developer/articles/technical/hybrid-architecture.html>. [cit. 2025-03-01].
- [16] CHENG, Gang, 2012. *A Comparison of Two Linux Schedulers*. Online, Master thesis. University of Oslo. Available from: <https://www.duo.uio.no/bitstream/handle/10852/9093/cheng.pdf>. [cit. 2025-03-01].
- [17] QIAN, Cai; ZAK, Karel and CARSTENS, Heiko, 2015. *lscpu - display information about the CPU architecture*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/stretch/util-linux/lscpu.1.en.html>. [cit. 2025-04-30].
- [18] LARABEL, Michael, 2024. *Early Linux 6.12 Kernel Benchmarks Showing Some Nice Gains On AMD Zen 5*. Online. Phoronix Media. Available from: <https://www.phoronix.com/news/Linux-6.12-Early-AMD-Zen-5>. [cit. 2025-04-04].
- [19] AGHERA, Sameer, 2024. *Debian vs Ubuntu: A Guide to Choosing the Right Linux Distribution*. Online. HorizonIQ. Available from: <https://www.horizoniq.com/blog/debian-vs-ubuntu/>. [cit. 2025-04-24].
- [20] DEBIAN WIKI TEAM, 2025. *LTS: Debian Long Term Support*. Online. Available from: <https://wiki.debian.org/LTS>. [cit. 2025-04-24].

- [21] DEBIAN DOCUMENTATION TEAM, 2025. *Release Notes for Debian 13 (trixie)*. Online. Available from: <https://www.debian.org/releases/trixie/release-notes/>. [cit. 2025-04-24].
- [22] MEYER, Daan De, 2025. *mkosi — Build Bespoke OS Images*. Online. Available from: <https://github.com/systemd/mkosi>. [cit. 2025-04-24].
- [23] WILLIAMS, Clark and SOMMERSETH, David, 2020. *hackbench - scheduler benchmark/stress test*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/testing/rt-tests/hackbench.8.en.html>. [cit. 2025-04-24].
- [24] GLEIXNER, Thomas and BOGANI, Alessio Igor, 2007. *cyclictst - High resolution test program*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/jessie/rt-tests/cyclictst.8>. [cit. 2025-04-24].
- [25] BOWDEN, Terrehon and BAUER, Bodo, 1999. *The /proc Filesystem*. Online. The kernel development community. Available from: <https://docs.kernel.org/filesystems/proc.html>. [cit. 2025-04-30].
- [26] THE LINUX INFORMATION PROJECT, 2006. *The /proc/cpuinfo File*. Online. Available from: https://www.linfo.org/proc_cpuinfo.html. [cit. 2025-05-14].
- [27] KOPYTOV, Alexey, 2024. *sysbench - multi-threaded benchmark tool for database systems*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/testing/sysbench/sysbench.1.en.html>. [cit. 2025-04-30].
- [28] KING, Colin Ian, 2020. *stress-ng*. Online. wiki.ubuntu.com. Available from: <https://wiki.ubuntu.com/Kernel/Reference/stress-ng>. [cit. 2025-04-30].
- [29] GODART, Sebastien, 2020. *pidstat - Report statistics for Linux tasks*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/bookworm/sysstat/pidstat.1.en.html>. [cit. 2025-04-30].
- [30] MACKENZIE, David, 2024. *nice - change process priority*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/testing/manpages-dev/nice.2.en.html>. [cit. 2025-04-30].

- [31] EDMONDS, Brian, 2023. *free - Display amount of free and used memory in the system*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/testing/procps/free.1.en.html>. [cit. 2025-04-30].
- [32] OPEN MPI PROJECT, 2025. *lstopo, lstopo-no-graphics, hwloc-ls - Show the topology of the system*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/experimental/hwloc/lstopo.1.en.html>. [cit. 2025-04-30].
- [33] MACKENZIE, David, 2024. *uname - print system information*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/testing/coreutils/uname.1.en.html>. [cit. 2025-04-30].
- [34] LINUX MAN-PAGES PROJECT, 2024. *sched - overview of CPU scheduling*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/testing/manpages/sched.7.en.html>. [cit. 2025-04-30].
- [35] MOCHEL, Patrick and MURPHY, Mike, 2011. *sysfs - The filesystem for exporting kernel objects*. Online. The kernel development community. Available from: <https://docs.kernel.org/filesystems/sysfs.html>. [cit. 2025-04-30].
- [36] MAREK, Rudolf, 2025. *Kernel driver coretemp*. Online. The kernel development community. Available from: <https://docs.kernel.org/hwmon/coretemp.html>. [cit. 2025-04-30].
- [37] LADISCH, Clemens, 2025. *Kernel driver k10temp*. Online. The kernel development community. Available from: <https://docs.kernel.org/hwmon/k10temp.html>. [cit. 2025-04-30].
- [38] INTEL CORPORATION, 2025. *What Is Throttling and How Can It Be Resolved?* Online. Available from: <https://www.intel.com/content/www/us/en/support/articles/000088048/processors.html>. [cit. 2025-04-30].
- [39] POETTERING, Lennart, 2024. *repart.d - Partition Definition Files for Automatic Boot-Time Repartitioning*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/testing/systemd/repart.d.5.en.html>. [cit. 2025-04-24].

- [40] RUBIN, Paul; MACKENZIE, David and KEMP, Stuart, 2022. *dd - convert and copy a file*. Online. manpages.debian.org. Available from: <https://manpages.debian.org/bookworm/coreutils/dd.1.en.html>. [cit. 2025-04-24].
- [41] INTEL CORPORATION, 2025. *Intel® Core™ i7-1260P Processor*. Online. Available from: <https://www.intel.com/content/www/us/en/products/sku/226254/intel-core-i71260p-processor-18m-cache-up-to-4-70-ghz/specifications.html>. [cit. 2025-04-24].
- [42] LENOVO TECHNOLOGY SDN. BHD. CO., 2025. *ThinkPad T14 Gen 3 (14" Intel)*. Online. Available from: [https://www.lenovo.com/my/en/p/laptops/thinkpad/thinkpadt/thinkpad-t14-gen-3-\(14-inch-intel\)/len101t0014](https://www.lenovo.com/my/en/p/laptops/thinkpad/thinkpadt/thinkpad-t14-gen-3-(14-inch-intel)/len101t0014). [cit. 2025-04-24].
- [43] WIKIMEDIA FOUNDATION, INC., 2025. *Box plot*. Online. Available from: https://en.wikipedia.org/wiki/Box_plot. [cit. 2025-05-17].
- [44] ADVANCED MICRO DEVICES, INC., 2025. *AMD Ryzen™ 5 2600 Desktop Processor*. Online. Available from: <https://www.amd.com/en/support/downloads/drivers.html/processors/ryzen/ryzen-2000-series/amd-ryzen-5-2600.html>. [cit. 2025-04-24].
- [45] ADVANCED MICRO DEVICES, INC., 2025. *AMD Ryzen™ 9 5950X Desktop Processor*. Online. Available from: <https://www.amd.com/en/products/processors/desktops/ryzen/5000-series/amd-ryzen-9-5950x.html>. [cit. 2025-04-24].
- [46] ADVANCED MICRO DEVICES, INC., 2025. *AMD "Zen" Core Architecture*. Online. Available from: <https://www.amd.com/en/technologies/zen-core.html>. [cit. 2025-04-24].

LIST OF FIGURES

Figure 3.1	Active and Expired Runqueue	23
Figure 3.2	CFS red-black tree	24
Figure 4.1	Phoronix compile test chart	30
Figure 4.2	Phoronix hackbench test chart	30
Figure 5.1	Latency flow chart	35
Figure 5.2	Fairness test flow chart	38
Figure 5.3	Throughput test flow chart	41
Figure 5.4	Context Switching test flow chart	44
Figure 5.5	Scalability test flow chart	46
Figure 5.6	Starvation detection test flow chart	48
Figure 5.7	System info collection flow chart	50
Figure 5.8	System init flow chart	52
Figure 5.9	Test procedure execution flow chart	55
Figure 5.10	Flow chart for CPU temperature and frequency logger script	56
Figure 7.1	i7-1260P CPU topology	66
Figure 7.2	i7-1260P CFS average latency	67
Figure 7.3	i7-1260P CFS average latency with CPU temp and freq	68
Figure 7.4	i7-1260P EEVDF average latency	70
Figure 7.5	i7-1260P EEVDF average latency with CPU temp and freq	71
Figure 7.6	i7-1260P CFS fairness CPU utilization	74
Figure 7.7	i7-1260P CFS fairness CPU time	74
Figure 7.8	i7-1260P EEVDF fairness CPU utilization	76
Figure 7.9	i7-1260P EEVDF fairness CPU time	76
Figure 7.10	i7-1260P CFS throughput	77
Figure 7.11	i7-1260P EEVDF throughput	78
Figure 7.12	i7-1260P CFS context switching	80
Figure 7.13	i7-1260P EEVDF context switching	81
Figure 7.14	i7-1260P CFS scalability	83
Figure 7.15	i7-1260P EEVDF scalability	84
Figure 7.16	i7-1260P CFS starvation thread life	85

Figure 7.17	i7-1260P CFS starvation CPU time.....	85
Figure 7.18	i7-1260P EEVDF starvation thread life	86
Figure 7.19	Ryzen 5 2600 CPU topology	88
Figure 7.20	Ryzen 5 2600 CFS average latency	89
Figure 7.21	Ryzen 5 2600 CFS average latency with CPU temp and freq	90
Figure 7.22	Ryzen 5 2600 EEVDF average latency	92
Figure 7.23	Ryzen 5 2600 EEVDF average latency with CPU temp and freq	93
Figure 7.24	Ryzen 5 2600 CFS fairness CPU utilization	95
Figure 7.25	Ryzen 5 2600 CFS fairness CPU time	95
Figure 7.26	Ryzen 5 2600 EEVDF fairness CPU utilization	97
Figure 7.27	Ryzen 5 2600 EEVDF fairness CPU time.....	97
Figure 7.28	Ryzen 5 2600 CFS throughput.....	98
Figure 7.29	Ryzen 5 2600 EEVDF throughput	99
Figure 7.30	Ryzen 5 2600 CFS context switching	100
Figure 7.31	Ryzen 5 2600 EEVDF context switching	102
Figure 7.32	Ryzen 5 CFS scalability	103
Figure 7.33	Ryzen 5 EEVDF scalability	104
Figure 7.34	Ryzen 5 CFS starvation thread life	105
Figure 7.35	Ryzen 5 CFS starvation CPU time	105
Figure 7.36	Ryzen 5 EEVDF starvation thread life.....	106
Figure 7.37	Ryzen 9 5950X CPU topology	108
Figure 7.38	Ryzen 9 5950X CFS average latency	109
Figure 7.39	Ryzen 9 5950X CFS average latency with CPU temp and freq	110
Figure 7.40	Ryzen 9 5950X EEVDF average latency with CPU temp and freq	112
Figure 7.41	Ryzen 9 5950X CFS fairness CPU utilization	114
Figure 7.42	Ryzen 9 5950X CFS fairness CPU time	114
Figure 7.43	Ryzen 9 5950X EEVDF fairness CPU utilization.....	116
Figure 7.44	Ryzen 9 5950X EEVDF fairness CPU time	116
Figure 7.45	Ryzen 9 5950X CFS throughput	117
Figure 7.46	Ryzen 9 5950X EEVDF throughput.....	118
Figure 7.47	Ryzen 9 5950X CFS context switching.....	120
Figure 7.48	Ryzen 9 5950X EEVDF context switching	121

Figure 7.49	Ryzen 9 5950X CFS scalability	122
Figure 7.50	Ryzen 9 5950X EEVDF scalability	123
Figure 7.51	Ryzen 9 5950X CFS starvation thread life	124
Figure 7.52	Ryzen 9 5950X CFS starvation CPU time	124
Figure 7.53	Ryzen 9 5950X EEVDF starvation thread life	125

LIST OF TABLES

Table 5.1	Latency test hackbench config	34
Table 5.2	Latency test cyclicttest config.....	34
Table 5.3	Fairness test hackbench config.....	37
Table 5.4	Stress-ng configuration for fairness test	43
Table 5.5	Pidstat configuration for fairness test.....	43
Table 5.6	Sysbench configuration for scalability test.....	46
Table 5.7	Unified scheduler tuning parameters	53
Table 5.8	Test execution scrip parameters	54
Table 5.9	Test execution script --tests parameter values	54
Table 7.1	DUT: i7-1260P - System Hardware Specification.....	65
Table 7.2	i7-1260P CFS CPU latency.....	66
Table 7.3	i7-1260P CFS overall latency	67
Table 7.4	i7-1260P CFS Temp. and Frequency.....	67
Table 7.5	i7-1260P EEVDF CPU latency	69
Table 7.6	i7-1260P EEVDF latency summary	69
Table 7.7	i7-1260P EEVDF Temp. and Frequency	70
Table 7.8	i7-1260P CFS fairness test summary statistics.....	73
Table 7.9	i7-1260P CFS fairness detailed metrics.....	73
Table 7.10	i7-1260P EEVDF fairness test summary statistics	75
Table 7.11	i7-1260P EEVDF fairness detailed metrics	75
Table 7.12	i7-1260P CFS Throughput Results for All Scenarios.....	77
Table 7.13	i7-1260P EEVDF Throughput Results for All Scenarios	78
Table 7.14	i7-1260P CFS per-thread context switching	79
Table 7.15	i7-1260P CFS context switching summary	79
Table 7.16	i7-1260P EEVDF context switching summary.....	80
Table 7.17	i7-1260P EEVDF per-thread context switching	81
Table 7.18	i7-1260P CFS Scalability Metrics	83
Table 7.19	i7-1260P EEVDF Scalability Metrics	84
Table 7.20	i7-1260P CFS starvation test summary	85
Table 7.21	i7-1260P EEVDF starvation test summary	86

Table 7.22	DUT: Ryzen 5 2600 - System Hardware Specification.....	87
Table 7.23	Ryzen 5 2600 CFS CPU latency	88
Table 7.24	Ryzen 5 2600 CFS overall latency	89
Table 7.25	Ryzen 5 2600 CFS Temp. and Frequency	89
Table 7.26	Ryzen 5 2600 EEVDF CPU latency	91
Table 7.27	Ryzen 5 2600 EEVDF overall measurement summary	91
Table 7.28	Ryzen 5 2600 CFS fairness summary statistics	94
Table 7.29	Ryzen 5 2600 CFS fairness detailed metrics	94
Table 7.30	Ryzen 5 2600 EEVDF fairness summary statistics	96
Table 7.31	Ryzen 5 2600 EEVDF fairness stats detailed metrics	96
Table 7.32	Ryzen 5 2600 CFS Throughput Results for All Scenarios	98
Table 7.33	Ryzen 5 2600 EEVDF Throughput Results for All Scenarios.....	99
Table 7.34	Ryzen 5 2600 CFS per-thread context switching	100
Table 7.35	Ryzen 5 2600 CFS context switching summary	100
Table 7.36	Ryzen 5 2600 EEVDF context switching summary.....	101
Table 7.37	Ryzen 5 2600 EEVDF per-thread context switching	101
Table 7.38	Ryzen 5 CFS Scalability Metrics.....	102
Table 7.39	Ryzen 5 EEVDF Scalability Metrics	103
Table 7.40	Ryzen 5 CFS starvation statistics	104
Table 7.41	Ryzen 5 EEVDF starvation statistics.....	106
Table 7.42	DUT: Ryzen 9 5950X - System Hardware Specification	107
Table 7.43	Ryzen 9 5950X CFS overall measurement summary.....	108
Table 7.44	Ryzen 9 5950X CFS CPU latency	109
Table 7.45	Ryzen 9 5950X EEVDF overall measurement summary	111
Table 7.46	Ryzen 9 5950X EEVDF CPU latency	111
Table 7.47	Ryzen 9 5950X CFS fairness summary statistics.....	113
Table 7.48	Ryzen 9 5950X CFS fairness detailed metrics	113
Table 7.49	Ryzen 9 5950X EEVDF fairness summary statistics	115
Table 7.50	Ryzen 9 5950X EEVDF fairness detailed metrics	115
Table 7.51	Ryzen 9 5950X CFS Throughput Results for All Scenarios	117
Table 7.52	Ryzen 9 5950X EEVDF Throughput Results for All Scenarios	118
Table 7.53	Ryzen 9 5950X CFS context switching summary	119

Table 7.54	Ryzen 9 5950X CFS per-thread context switching	119
Table 7.55	Ryzen 9 5950X EEVDF context switching summary	120
Table 7.56	Ryzen 9 5950X EEVDF per-thread context switching	121
Table 7.57	Ryzen 9 CFS Scalability Metrics.....	122
Table 7.58	Ryzen 9 5950X EEVDF Scalability Metrics	123
Table 7.59	Ryzen 9 5950X CFS starvation statistics	124
Table 7.60	Ryzen 9 5950X EEVDF starvation statistics	125
Table 8.1	Intel i7-1260P latency comparison.....	126
Table 8.2	Ryzen 5 2600 latency comparison	127
Table 8.3	Ryzen 9 5950X latency summary	127
Table 8.4	Cross-platform latency summary.....	128
Table 8.5	Cross-platform maximum latency summary	128
Table 8.6	Intel i7-1260P fairness summary	129
Table 8.7	Ryzen 5 2600 fairness summary.....	130
Table 8.8	Ryzen 9 5950X fairness summary	130
Table 8.9	Cross-platform fairness CPU utilization	131
Table 8.10	Throughput Comparison: i7-1260P.....	131
Table 8.11	Throughput Comparison: Ryzen 5 2600	132
Table 8.12	Throughput Comparison: Ryzen 9 5950X	132
Table 8.13	Aggregate Throughput Comparison Across Platforms	133
Table 8.14	Context Switch Comparison: i7-1260P.....	133
Table 8.15	Context Switch Comparison: Ryzen 5 2600.....	134
Table 8.16	Context Switch Comparison: Ryzen 9 5950X	134
Table 8.17	Aggregate Context Switching Comparison	135
Table 8.18	Intel i7-1260P Scalability Comparison	135
Table 8.19	Ryzen 5 2600 Scalability Comparison	136
Table 8.20	Ryzen 9 5950X Scalability Comparison.....	136
Table 8.21	Cross-platform scalability summary.....	137
Table 8.22	Starvation Comparison: i7-1260P.....	138
Table 8.23	Starvation Comparison: Ryzen 5 2600	138
Table 8.24	Starvation Comparison: Ryzen 9 5950X	139
Table 8.25	Cross-platform Starvation Summary	140

LIST OF LISTINGS

Listing 5.1	Reduced sample latency test output (source: own)	36
Listing 5.2	Reduced sample fairness test output (source: own)	39
Listing 5.3	Reduced sample throughput test output (source: own)	42
Listing 5.4	Reduced sample context switching output (source: own)	44
Listing 5.5	Reduced sample scalability test output (source: own)	47
Listing 5.6	Reduced sample starvation test output (source: own)	49
Listing 5.7	Reduced sample system environment log (source: own)	51
Listing 5.8	Reduced sample of CPU temp and freq log output (source: own)	57
Listing 6.1	mkosi python install command (source: [22])	58
Listing 6.2	mkosi project directory (source: own)	59
Listing 6.3	mkosi.conf of CFS image (source: own)	61
Listing 6.4	mkosi.conf of EEVDF image (source: own)	62
Listing 6.5	60-output.conf (source: own)	63
Listing 6.6	mkosi build execution (source: [22])	64
Listing 6.7	mkosi build with direct run in virtual machine (source: [22])	64
Listing 6.8	Copy of image to USB stick drive by dd (source: [40])	64

LIST OF ABBREVIATIONS

API	Application Programming Interface
CCX	Core Complex (AMD architecture unit)
CFS	Completely Fair Scheduler
CLI	Command Line Interface
CPU	Central Processing Unit
cmd	Command
EDF	Earliest Deadline First
EEVDF	Earliest Eligible Virtual Deadline First
EAS	Energy-Aware Scheduling
EFI	Extensible Firmware Interface
EPS	Events Per Second
FCFS	First Come First Served
freq	Frequency
GUI	Graphical User Interface
I/O	Input/Output
IRQ	interquartile range
LTS	Long Term Support
NUMA	Non-Uniform Memory Access
O(1)	Constant Time Scheduler
PID	Process identifier
QEMU	Quick Emulator
RED	Rate, Errors, Duration
RMS	Rate Monotonic Scheduling
RT	Real-Time
SMT	Simultaneous Multithreading
stdout	standard output
temp	Temperature
TLB	Translation Lookaside Buffer
UEFI	Unified Extensible Firmware Interface
USE	Utilization, Saturation, Errors
VNA	Vector Network Analyzer
VRUNTIME	Virtual Runtime

LIST OF APPENDICES

APPENDIX A I: mkosi.repart partition configuration files

APPENDIX A II: Contents of the attached ZIP archive

APPENDIX A I: MKOSI.REPART PARTITION CONFIGURATION FILES

This appendix presents the configuration files used with `mkosi .repart` for creating partition layouts in the test images.

```
1 [Partition]
2 Type=esp
3 Format=vfat
4 CopyFiles=/boot:/
5 CopyFiles=/efi:/
6 SizeMinBytes=512M
7 SizeMaxBytes=512M
```

ESP partition: EFI System Partition, FAT-formatted, used for boot files and UEFI firmware.

```
1 [Partition]
2 # UUID of the grub BIOS boot partition which grubs
3 # needs on GPT to embed itself into.
4 Type=21686148-6449-6e6f-744e-656564454649
5 SizeMinBytes=1M
6 SizeMaxBytes=1M
```

BIOS Boot partition: Needed for GRUB on BIOS systems with GPT partitioning.

```
1 [Partition]
2 Type=root
3 Format=ext4
4 CopyFiles=/
5 CopyFiles=/usr/
6 Minimize=guess
```

Root partition: ext4-formatted main system partition holding the root filesystem and core directories.

APPENDIX A II: CONTENTS OF THE ATTACHED ZIP ARCHIVE

This appendix lists the contents of the ZIP archive attached to the thesis, which includes the following files and directories:

- **fulltext.pdf** – Full version of the thesis in PDF/A format.
- **mkosi_images/mkosi_CFS/** – Directory containing mkosi configuration files for the CFS-based system image.
- **mkosi_images/mkosi_EEVDF/** – Directory containing mkosi configuration files for the EEVDF-based system image.
- **test_scripts/** – Directory containing all scripts used for the testing methodology.
- **test_results/** – Directory with test results analyzed in the thesis.