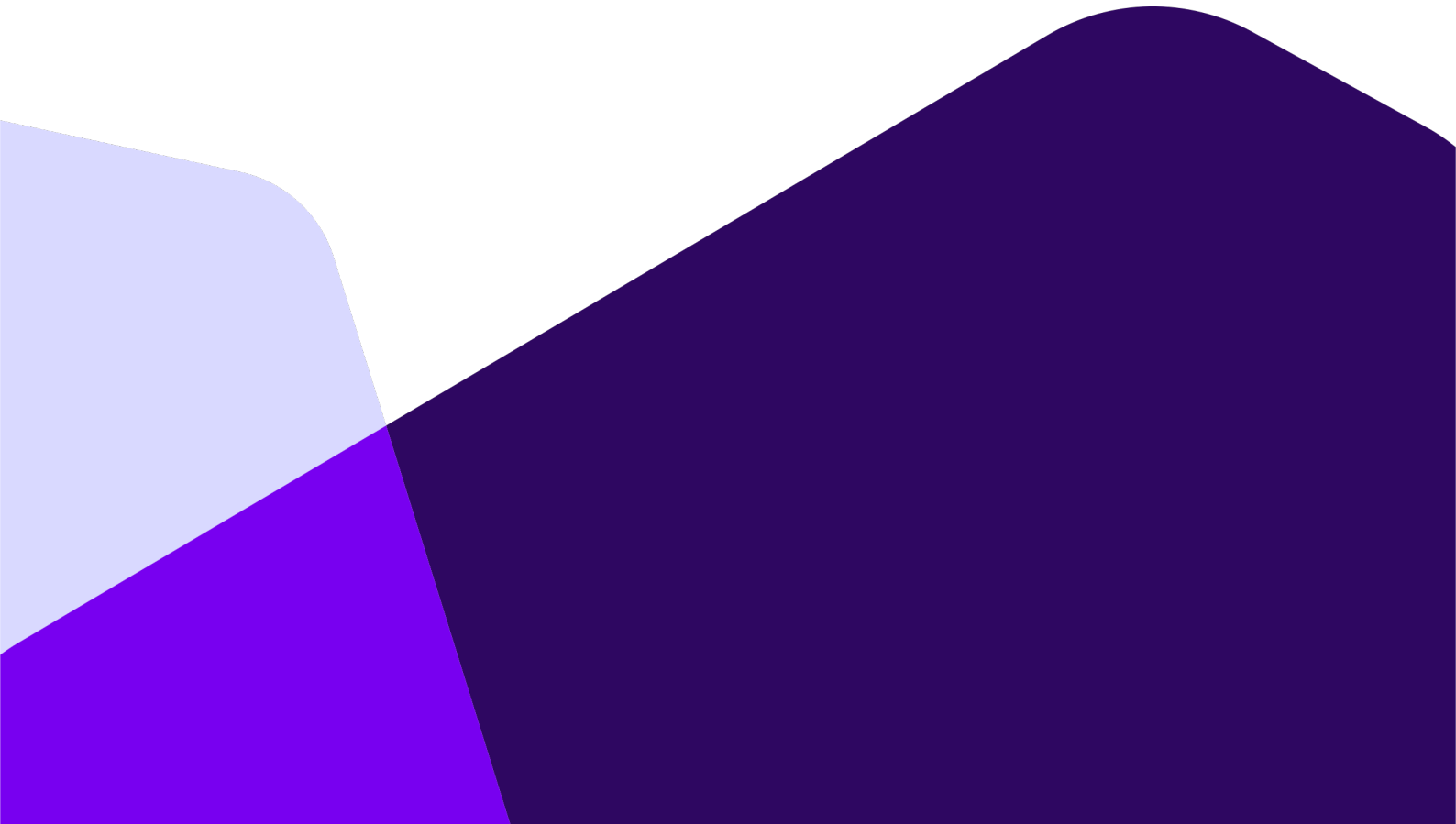


Kent S. ODDE / Candidate number: 8504

Analysis and Optimization of Real-Time Systems on Altera Cyclone V with Linux



University of South-Eastern Norway
Faculty of Technology, Natural Sciences, and Maritime Sciences
Department of Science and Industry Systems
PO Box 235
NO-3603 Kongsberg, Norway

<http://www.usn.no>

© 2024 Kent S. ODDE

This thesis is worth 60 study points



University of
South-Eastern Norway

Analysis and Optimization of Real-Time Systems on Altera Cyclone V with Linux

Master's Thesis in Computer Science

Kent S. ODDE

Academic Supervisor

Prof. António L. L. RAMOS

Industry Supervisor

Eivind JACOBSEN

University of South-Eastern Norway

Faculty of Technology, Natural Sciences and Maritime Sciences

Department of Science and Industry Systems

Campus Kongsberg

May 2024

Abstract

This thesis aims to enable a planned effort to port a legacy system to a new hardware and software platform. The target platform is an Altera Cyclone V running a custom Linux distribution shared across several products within the company where the author is employed. Although not an inherent real-time system, the legacy system has several real-time constraints that must be met on the new platform to function as intended. Despite being a general-purpose operating system, many approaches exist to improve the real-time capabilities of Linux, such as the PREEMPT_RT patchset. This study's primary focus is investigating the viability of porting the legacy system to the new platform and determining the best approach for meeting the system's temporal requirements.

The first part of the thesis focuses on benchmarking a generic Linux system with different kernel settings to evaluate the impact of using different preemption models and scheduling policies. The second part analyzes the legacy system's requirements and mechanics. Further, it evaluates, implements, and tests software mechanisms that mimic the legacy system's time-critical functionality on the new platform.

This study showed that the PREEMPT_RT patch can improve the real-time capabilities of Linux but also introduces issues that can negatively impact the system. Additionally, it shows that the mainline kernel performs reasonably well and can be satisfactory for many systems.

The thesis concludes that the porting effort is feasible, potentially without requiring particular measures. The results show that the real-time constraints of the legacy system are not stringent enough to warrant the use of PREEMPT_RT. However, a preemptible low-latency kernel and real-time scheduling policies are recommended to improve the system's reliability and stability. The thesis also presents several recommendations regarding the target application's design and mechanics. Future work includes porting and testing the legacy system on the new platform and implementing the recommended modifications to the system.

Acknowledgements

I would like to express my gratitude to my academic supervisor, Professor António L. L. Ramos, at the University of South-Eastern Norway. He has been a tremendous source of encouragement and has guided and supported me throughout the process of writing this thesis.

Many thanks to my industry supervisor, Eivind Jacobsen, for suggesting the topic of this thesis, as well as my line manager, Ricardo Marquez, for providing me with the necessary resources to work on it. Further, I would like to thank Eldor Rødseth, Arne Lie, and Helge Rustad for their help in shaping the problem statement and for sharing their knowledge. Although challenging, delving into this particular topic has proven to be an incredibly enriching and enlightening experience.

My sincere appreciation also goes to my classmates with whom I've shared the ups and downs of the past year, especially Stian Onarheim, who has been a great source of inspiration and motivation.

Thanks should also go to family and friends who have supported me throughout this process. However, a special acknowledgment goes to my partner, Kristine, for her endless patience, support, and encouragement. Not only throughout the thesis year but for my entire venture into higher education.

Lastly, I would like to thank my mother. For everything.

Kent S. Odde
Kongsberg, Norway, May 24, 2024

Contents

Abstract	i
Acknowledgements	ii
1 Introduction	1
1.1 Motivation and Problem Statement	1
1.2 Objectives and Deliverables	2
1.3 Intellectual Property Considerations	2
1.4 Outline	3
2 Background	4
2.1 Real-Time Systems	4
2.2 Linux	5
2.2.1 Embedded Linux	5
2.2.2 Scheduling	7
Deadline Scheduling	9
Real-Time Scheduling	9
Completely Fair Scheduler	9
Earliest Eligible Virtual Deadline First	10
Kernel Threads & CPU Modes	10
2.2.3 Real-Time Linux	11
Co-Kernel	12
Single-Kernel	13
2.3 Target System	14
2.3.1 Hardware	17
2.3.2 Software	18
2.3.3 Real-Time Constraints	18
3 PREEMPT_RT	20
3.1 Literature Review Methodology	20
3.2 High-Level Overview	22
3.3 Main Features	22
3.3.1 Fully Preemptible Kernel	23
3.3.2 High-Resolution Timers	25
3.3.3 Threaded IRQs	25
3.3.4 Priority Inheritance	28
3.3.5 Preemptible RCU	28
3.3.6 Full Tickless Operation and CPU isolation	29
3.4 Current State and Mainlining	30
3.5 Performance and Real-Time Capabilities	30

3.5.1	Latency	30
3.5.2	Determinism	31
3.5.3	Throughput Degradation	32
3.6	Tuning and Best Practices	32
3.6.1	Workload Analysis	33
3.6.2	Kernel Compile-Time Settings	33
3.6.3	Policies, Priorities, and Throttling	33
3.6.4	Measuring Time and Sleeping	34
3.6.5	Managing Memory	35
3.6.6	Broken Modules and Subsystems	36
3.6.7	Loadable Kernel Modules	36
3.7	Summary	37
4	Methodology and Design	38
4.1	Platforms and Tools	39
4.1.1	Hardware	39
4.1.2	Board Support Package	39
	Toolchain	39
	U-Boot	40
	Linux Kernel	40
	Root Filesystem	40
	Running the BSP	41
4.2	Test Setups	41
4.2.1	Comparative Setup Configurations	41
4.2.2	Stressors	42
4.3	Platform Baseline Tests	43
4.3.1	High-Resolution Timers Verification	43
4.3.2	Cyclictest	43
4.3.3	Throughput Test	44
4.3.4	Preemption Test	45
4.3.5	Memory Lock Test	45
4.4	Target System Analysis	46
4.4.1	<i>Real Time vs. Real Fast</i>	46
4.4.2	Periodic vs Aperiodic	46
4.4.3	Memory Management	48
4.4.4	Single-Threaded vs Multi-Threaded	48
4.4.5	Requirements	48
	Aperiodic	49
	Periodic	49
	Transmitting	50
4.4.6	Design of System Under Test	50
	Reception Pipeline	50
	Transmission Pipeline	53
4.4.7	Output	55
4.4.8	FPGA Design	56
4.5	Target System Experiments	57
4.5.1	Dynamic Memory Allocation	57
4.5.2	Periodic Execution Test	57

4.5.3	Shared Memory Interaction	61
4.5.4	Generic Netlink Interface	63
4.5.5	Kernel Thread Event Handling	65
4.5.6	Full Test	66
	Special Considerations for Transmission Test	67
4.5.7	Production Environment Test	67
4.6	Summary	68
5	Test Results and Recommendations	69
5.1	Platform Baseline Tests	69
5.1.1	High-Resolution Timers Verification	69
5.1.2	Cyclictest	69
5.1.3	Throughput Test	73
5.1.4	Preemption Test	77
	Interrupt Latency	81
	Kernel Preemption Test	81
5.1.5	Memory Lock Test	84
5.1.6	Discussion	85
5.2	Target System Tests	86
5.2.1	Dynamic Memory Allocation	86
5.2.2	Periodic Execution Test	88
5.2.3	Shared Memory Test	92
	512 Byte Packets	95
5.2.4	Generic Netlink Test	97
5.2.5	Kernel Thread Event Handling	99
5.2.6	Full Test RX	100
5.2.7	Full Test TX	103
	Setup 1	103
	Setup 2	105
	Setup 3	107
	Comparison	108
5.2.8	Production Environment Test	109
5.3	Recommendations for the Target System	111
5.3.1	<i>If it Ain't Broke, Don't Fix it?</i>	111
5.3.2	Real-Time Scheduling Policies	111
5.3.3	Memory	112
5.3.4	kmemleak	112
5.3.5	Preemption Model	112
5.3.6	Modifications to the Legacy Application	113
5.4	Summary	114
6	Conclusion and Future Work	115
6.1	Conclusion	115
6.2	Future Work	116
A	BSP Configuration	117
A.1	Crosstool-NG Configuration	117
A.2	Linux Configuration	117
A.3	Preemptible Mainline Kernel Configuration	120

A.4 Linux-stable-rt Configuration	121
B Stressor Configurations	122
C Additional Result Data	125
C.1 Baseline Tests	125
C.1.1 High-Resolution Timers Verification	125
C.1.2 Cyclictst	130
C.1.3 Throughput Test	133
C.1.4 Preemption Test	136
C.1.5 Memory Lock Test	138
C.2 Target System Tests	139
C.2.1 Dynamic Allocation	139
C.2.2 Periodic Thread Mechanisms	141
C.2.3 Shared Memory Test	152
C.2.4 Event-Driven vs Polled Kthread	160
C.2.5 Generic Netlink Test	161
C.2.6 Full Test RX	163
C.2.7 Full Test TX	168
C.2.8 Production Environment Test	178
D Miscellaneous	180
Bibliography	181

List of Figures

2.1	Embedded Linux boot sequence.	6
2.2	Illustration of kernel configuration through <i>menuconfig</i>	7
2.3	Simplified state machine for tasks.	8
2.4	Preemption of user mode vs kernel mode task.	12
2.5	Co-kernel vs single-kernel approach.	13
2.6	Target system.	14
2.7	High-level architecture of the target system.	15
2.8	HPS-FPGA shared memory interface.	16
2.9	Simplified Altera Cyclone V block diagram [38] Copyright ©2017, IEEE.	17
3.1	Structured literature search.	21
3.2	Refining search results.	22
3.3	Preemptible kernel.	23
3.4	Preemption models.	24
3.5	Fully preemptible kernel.	24
3.6	Traditional interrupts.	26
3.7	Top- and bottom-half interrupt processing.	26
3.8	Threaded bottom-half.	27
3.9	Forced threading.	27
4.1	Design of reception pipeline.	51
4.2	Design of transmission pipeline.	53
4.3	Quartus high-level platform overview.	56
4.4	Main routine of shared memory test.	62
4.5	Generic Netlink throughput test.	64
5.1	Cyclictest results, no load.	70
5.2	Cyclictest results, Stress-NG 25 %.	71
5.3	Cyclictest with SCHED_FIFO.	71
5.4	Cyclictest on the company's production kernel.	73
5.5	Throughput comparison across different preemption models.	73
5.6	Hackbench throughput comparison across preemption models using Unix sockets and pipes.	74
5.7	Hackbench throughput comparison across preemption models using Unix sockets and pipes, scheduled under SCHED_FIFO.	75
5.8	Throughput comparison of Iperf across different preemption models.	76
5.9	Throughput comparison of 7z across different preemption models.	76
5.10	Throughput comparison of 7z across different preemption models measured in MIPS.	77
5.11	Preemption count of CPU-intensive task under SCHED_OTHER.	77
5.12	Preemption count of CPU-intensive task under SCHED_FIFO.	78

5.13	CPU migrations detected using eBPF.	80
5.14	CPU-intensive task preempting Iperf.	80
5.15	Preemptions of kernel tasks by Cyclicttest.	83
5.16	Preemptions of kernel tasks by Cyclicttest using PREEMPT_RT.	84
5.17	Latencies of soft page faults.	85
5.18	Dynamic memory allocation results.	87
5.19	Worst case latencies per dynamic memory operation.	88
5.20	Avg latency of periodic execution test, at 25 % load.	89
5.21	Max latency of periodic execution test, at 25 % load.	89
5.22	Latency distribution of <i>setitimer</i> with SCHED_OTHER and 25 % load.	90
5.23	Latency distribution of periodic mechanisms with SCHED_FIFO and 25 % load.	91
5.24	Average CPU load of periodic execution mechanisms.	92
5.25	Shared memory test, average CPU load.	93
5.26	Shared memory test, preemption count.	93
5.27	Shared memory test, max buffer size	93
5.28	Correlation between latency and max buffer size for each shared memory test configuration.	94
5.29	Shared memory test, 512-byte packets, average CPU Load.	95
5.30	Shared memory test, 512-byte packets, preemption count.	95
5.31	Shared memory test, 512-byte packets, max buffer size.	96
5.32	Shared memory test, execution times compared under SCHED_FIFO with Stress-NG 75 %.	96
5.33	Shared memory test, latencies compared under SCHED_FIFO and Iperf load.	97
5.34	Execution time of kernel thread communication over Generic Netlink.	98
5.35	Polled vs. event-Driven kernel worker, at different input event frequencies.	99
5.36	Average execution time of full RX pipeline.	100
5.37	Maximum execution time of full RX pipeline.	100
5.38	Maximum execution time of full RX pipeline under SCHED_FIFO.	101
5.39	Maximum buffer size of full RX pipeline.	101
5.40	Maximum buffer size of full RX pipeline, increased overhead.	102
5.41	Temporal data for full RX pipeline with increased overhead.	102
5.42	CPU load of full TX pipeline, setup 1, under Stress-NG 25 %	104
5.43	Minimum buffer size in TX pipeline, setup 1	104
5.44	Average CPU load of full TX pipeline, setup 2 under Stress-NG 75 %	105
5.45	Minimum buffer size of full TX pipeline, setup 2 with kernel thread under SCHED_OTHER.	106
5.46	Minimum buffer size of full TX pipeline, setup 2 with kernel thread under SCHED_FIFO.	106
5.47	Average CPU load of full TX pipeline, setup 3	107
5.48	Minimum buffer size of full TX pipeline, setup 3 with kernel thread under SCHED_FIFO.	107
5.49	Minimum buffer size of Full TX pipeline compared across setups with 75 % load.	108
5.50	Number of underruns on Full TX pipeline compared across setups with 75 % load.	108

5.51	Latency distribution of the company's production Linux system with no forced preemption.	109
5.52	Latency distribution of the company's production Linux system with preemptive kernels.	109
5.53	CPU load average of background workload on the company's production Linux distribution.	110
5.54	CPU utilization of background workload on the company's production Linux distribution.	111
C.1	Cyclictest results with Stress-NG 75 % and Iperf.	132
C.2	Duration of writing to dynamically allocated memory	139
C.3	Duration of freeing dynamically allocated memory	140
C.4	Latency distribution of periodic mechanisms with SCHED_OTHER and 25 % load.	145
C.5	Latency distribution of periodic mechanisms with SCHED_OTHER and 75 % load.	146
C.6	Latency distribution of periodic mechanisms with SCHED_FIFO and 75 % load.	147
C.7	Latency distribution of periodic mechanisms with SCHED_OTHER, 25 % load and PREEMPT_RT.	148
C.8	Latency distribution of periodic mechanisms with SCHED_FIFO, 25 % load and PREEMPT_RT.	149
C.9	Latency distribution of periodic mechanisms with SCHED_OTHER, 75 % load and PREEMPT_RT.	150
C.10	Latency distribution of periodic mechanisms with SCHED_FIFO, 75 % load and PREEMPT_RT.	151

List of Tables

4.1	Versions of relevant software provided by Buildroot.	41
4.2	Versions of tools for building BSP artifacts.	41
C.1	Cyclictest results, no load	130
C.2	Cyclictest results, Stress-NG 25 %	130
C.3	Cyclictest results, Stress-NG 75 %	131
C.4	Cyclictest results with Iperf load	131
C.5	Cyclictest results, 800 kernel	132
C.6	Throughput test initial results.	133
C.7	Throughput results, Hackbench	133
C.8	Throughput results, Hackbench SCHED_FIFO	134
C.9	Throughput test Iperf and p7zip.	135
C.10	Preemption test results.	136
C.11	Preemption test results, with load.	136
C.12	Migrations detected during throughput test	137
C.13	Page fault test results.	138
C.14	Periodic execution test results.	141
C.15	Shared memory test results, 4 KiB packets.	152
C.16	Shared memory test results, 4 KiB packets Stress-NG 25 %.	153
C.17	Shared memory test results, 4 KiB packets Stress-NG 75 %.	154
C.18	Shared memory test results, 4 KiB packets with Iperf.	155
C.19	Shared memory test results, 512 B packets.	156
C.20	Shared memory test results, 512 B packets with Stress-NG 25 %.	157
C.21	Shared memory test results, 512 B packets with Stress-NG 75 %.	158
C.22	Shared memory test results, 512 B packets with Iperf.	159
C.23	Event-driven vs polled kernel worker	160
C.24	Event-driven vs polled kernel worker, with Stress-NG 25 %	160
C.25	Generic Netlink test results with SCHED_FIFO on kernel thread, no load	161
C.26	Generic Netlink test results with SCHED_FIFO on kernel thread, with Stress-NG 25 %	161
C.27	Generic Netlink test results with SCHED_FIFO on kernel thread, with Stress-NG 75 %	161
C.28	Generic Netlink test results with SCHED_FIFO on kernel thread, with Iperf	162
C.29	Full test RX, no load	163
C.30	Full test RX, with Stress-NG 25 %	163
C.31	Full test RX, with Stress-NG 75 %	164
C.32	Full test RX, with Iperf	164
C.33	Full test RX, increased overhead, no load	165
C.34	Full test RX, increased overhead, with Stress-NG 25 %	166

C.35 Full test RX, increased overhead, with Iperf	167
C.36 Full test TX, Setup 1, no load	168
C.37 Full test TX, Setup 1, with Stress-NG 25 %	168
C.38 Full test TX, Setup 1, with Stress-NG 75 %	169
C.39 Full test TX, Setup 1, with Iperf.	169
C.40 Full test TX, Setup 2, no load	170
C.41 Full test TX, Setup 2, with Stress-NG 25 %	171
C.42 Full test TX, Setup 2, with Stress-NG 75 %	173
C.43 Full test TX, Setup 2, with Iperf	174
C.44 Full test TX, Setup 3, no load	175
C.45 Full test TX, Setup 3, with Stress-NG 25 %	176
C.46 Full test TX, Setup 3, with Stress-NG 75 %	176
C.47 Full test TX, Setup 3, with Iperf.	177
C.48 Production environment latency results.	178
C.49 Production environment load results.	179

Listings

3.1	Struct timespec definition [92].	34
3.2	Example of manually pre-faulting heap memory.	36
4.1	Cyclictest baseline test configurations.	43
4.2	Throughput test workload parameters.	44
4.3	CPU intensive task, excerpt from preemption test.	45
4.4	Update function of reception pipeline.	52
4.5	Update function of transmission pipeline.	54
4.6	Command line dashboard of SUT.	55
4.7	Main routine of dynamic memory test.	57
4.8	Periodic task implemented with <code>setitimer</code>	58
4.9	Periodic task implemented with <code>nanosleep</code>	59
4.10	Periodic task implemented with deadline scheduling.	60
4.11	Periodic task implemented using condition variable with timeout.	61
4.12	Shared memory test.	63
4.13	Main routine of Generic Netlink throughput test.	64
4.14	Polling kernel worker.	65
4.15	Event-driven kernel worker.	66
5.1	Cyclictest on the company's production kernel.	72
5.2	Cyclictest on the company's production kernel, without <code>kmemleak</code>	72
5.3	Iperf configuration for throughput test.	75
5.4	7z benchmark.	76
5.5	eBPF program to detect preempting task.	78
5.6	eBPF program to detect task migration.	79
5.7	Interrupt latency caused by CPU-intensive <code>SCHED_FIFO</code> task.	81
5.8	eBPF program to detect when preempting task executing in kernel mode.	82
A.1	Crosstool-NG configuration.	117
A.2	Linux kernel configuration.	117
A.3	Preemptible mainline kernel configuration diff	120
A.4	<code>PREEMPT_RT</code> kernel configuration diff	121
B.1	Stressor: Stress-NG 25%.	122
B.2	Stressor: Stress-NG 75%.	123
B.3	Stressor: Iperf.	123
B.4	Stressor: Hackbench.	124
C.1	Output of <code>/proc/timerlist</code> on 6.6.14-rt21.	125
D.1	Linux kernel, number of commits in 2023.	180

List of Abbreviations

.dtb	Device Tree Blob
.rbf	Raw Binary File
ACP	Accelerated Coherency Port
ADC	Analog-to-Digital Converter
AXI	Advanced eXtensible Interface
BIOS	Basic Input/Output System
BSP	Board Support Package
CBS	Constant Bandwidth Server
CFS	Completely Fair Scheduler
CMA	Continuous Memory Allocator
CPU	Central processing unit
DAC	Digital-to-Analog Converter
DL	Deadline
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
EDS	Earliest Deadline First
EEVDF	Earliest, Eligible Virtual Deadline First
FIFO	First-In, First-Out
FPGA	Field-Programmable Gate Array
FPSoC	Fully-Programmable System on Chip
FSF	Free Software Foundation
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
HPS	Hard Processor System
HR	High-Resolution
HW	Hardware
IPC	Inter-Process Communication Mechanism
IP	Internet Protocol
IRQ	Interrupt Request
LKM	Loadable Kernel Module
NFS	Network File System
OSADL	Open Source Automation Development Lab
OS	Operating System
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RCU	Read Copy Update
ROM	Read-Only Memory
RR	Round-Robin
RTOS	Real-Time Operating System
RT	Real-Time

RX	Receive
SD	Secure Digital
SDRAM	Synchronous Dynamic Random Access Memory
SMM	System Manager Mode
SMP	Symmetric Multi-Processing
SUT	System Under Test
SW	Software
SoC	System on Chip
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TTY	Teletypewriter
TX	Transmit
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
WCET	Worst-Case Execution Time

Chapter 1

Introduction

Linux is a general-purpose operating system, meaning its design maximizes throughput [1]–[3]. Although this is desirable in most desktop and server systems, this is not true for embedded devices running tasks with real-time constraints. Instead, it implies that the operating system can have unbounded latency, resulting in a non-deterministic system where the response time to an event or the wake-up time of a cyclic task cannot be guaranteed [1], [2].

Several approaches and tools have evolved to enable the use of Linux in real-time systems. The PREEMPT_RT patch is the most notable, which modifies the kernel to increase its real-time capabilities [1], [3].

This thesis aims to enable a planned effort within the company at which the author is employed¹ to port a legacy real-time system to a new hardware (HW) and software (SW) platform. The new platform is an Altera Cyclone V running a custom Linux distribution shared across several of the company's products.

The target system must meet several real-time requirements to function as intended and achieve the desired performance. This thesis will consist of designing and implementing a HW/SW system that mirrors the target system's planned architecture and data flow. Then, it will analyze whether its real-time requirements can be met under different conditions with and without the PREEMPT_RT patch. Additional side-effects of enabling kernel preemption will also be analyzed.

1.1 Motivation and Problem Statement

This project aims to implement software that mirrors the data flow of the target system and then optimize it using available techniques. The focus will be on limiting the kernel interference with the real-time tasks, analyzing and optimizing Inter-Process Communication (IPC) mechanisms, and ensuring that any relevant Loadable Kernel Modules (LKM) do not perform system calls with unbounded latencies. This outcome will hopefully be a factor in the success of the target system.

A secondary aim of the project is to discover the potential implications of applying a PREEMPT_RT patched kernel to the company's other systems. A known side effect of applying this patch is that the overall throughput decreases [1]. However, the desire to

¹Henceforth referred to as *the company*

use a single OS constellation for all systems warrants investigating how the patch will affect existing systems.

The academic motivation for the project is that much of the research done within this field has focused on comparing PREEMPT_RT to different co-kernel approaches [4], [5]. The research on PREEMPT_RT often uses generic test suites [6]–[8] or focuses on particular uses or mechanisms [3], [4], [9]. Many researchers have also failed to explain how they have configured their systems [1]. The main interest has been to see what latencies it is theoretically possible to achieve using generic tools.

Rather than being a generic investigation into the capabilities of PREEMPT_RT, this thesis will investigate how PREEMPT_RT affects a concrete HW/SW architecture, both concerning the latencies of real-time tasks and the performance of other non-real-time tasks. The results will help others consider the suitability of PREEMPT_RT for their systems.

Although the System Under Test (SUT) mimics the data flow of the planned target system, this architecture is typical for HW/SW co-designs in high data rate systems, meaning that it can also benefit other applications in different fields.

1.2 Objectives and Deliverables

The objectives of this thesis work are:

1. A literature review on PREEMPT_RT
2. Implementation of a HW/SW co-design architecture, typical in high data rate systems
3. Performance and real-time capability analysis of PREEMPT_RT
4. Propose kernel compile-time and run-time configurations for meeting real-time and throughput requirements
5. Guidelines for real-time application development in the context of the SUT

The deliverables of this thesis are:

1. The final thesis report containing all the contributions mentioned above².
2. The HW/SW co-design comprising the SUT will be made open-source³.
3. Any custom test suites developed will be made open-source³.

1.3 Intellectual Property Considerations

In order to keep the results of the study open and available to the research community, this thesis does not utilize any intellectual property belonging to the company. The hardware used is a commercially available development kit, and the software used

²Throughout the process of writing this thesis, Grammarly [10] was used to check the grammar and spelling of the text. Grammarly's suggestions to rephrase text were adhered to occasionally.

³Available in [11]

is either developed from scratch or open source. In some limited cases, the Linux system used in company production environments is tested, but only to verify that the findings in the thesis apply to their systems.

The existing legacy target system is the company's proprietary intellectual property. Therefore, its descriptions will be limited to what is necessary in order to understand the thesis. However, the scope of this thesis is not to reverse-engineer the target system for a new platform but to analyze the real-time capabilities of Linux on a similar HW/SW architecture.

1.4 Outline

The rest of this thesis is outlined as described below:

- Chapter 2 introduces concepts and topics required to understand the main topics of this thesis and its problem statement. It provides a high-level overview of real-time systems and introduces Linux in the context of embedded systems. Further, it discusses scheduling concepts in Linux and briefly introduces the mechanics and requirements of the target system.
- Chapter 3 is a literature review with a focus on the PREEMPT_RT patchset. It places PREEMPT_RT into a historical context and discusses the motivation behind its development. Further, it examines the patchset's main features and current state before investigating its capabilities and limitations. Lastly, it identifies best practices and recommendations found in the literature and relevant documentation.
- In chapter 4, the research methods of the thesis are discussed. The chapter analyzes the target system further and gives an overview of the system under test. It also discusses the experiments designed for the thesis, including comparisons, measures, and expectations.
- The results are presented and discussed in chapter 5, while chapter 6 offers recommendations about possible future work and concludes the thesis.

Chapter 2

Background

This chapter discusses topics and concepts critical for comprehending the thesis' means, goals, and outcomes.

2.1 Real-Time Systems

A *real-time* system is defined as a system that, to be considered correct, must not only provide correct outputs but must do so at the correct time [12]. This means that a deadline is imposed on the system to deliver an output, either cyclically or after an event. The severity of breaking a deadline to produce an output is the usual way to classify real-time systems. A system in which a single missed deadline can lead to loss of human life or massive environmental damage is considered a *hard* real-time system. Other classifications include *firm* and *soft*, but the literature differs slightly in how these terms are defined and which of them they choose to include [1], [12]–[14].

This thesis will limit the classifications to *hard* and *soft* and use the following definitions:

- **Hard:** A system in which a single missed deadline will lead to a complete system failure [13], [14].
- **Soft:** A system in which one or more missed deadlines may only lead to degraded performance or reduced general usefulness [13], [14].

No response can come instantaneously. All effects of a cause will have some delay, termed *latency* in real-time literature. The variation in latency is called *jitter* [12]. Events in computer systems can be *periodic*, *aperiodic*, or *sporadic* [12]. While the meaning of the two former is obvious, sporadic events are aperiodic but with a lower or upper bound on how often they may occur. In real-time systems, constraints limit the permitted latency from an event occurring until its effects are observable. While one may be concerned with the average latency, one often cares more about the worst case.

Real-time is an overloaded term and is often mistakenly confused with performance [1], [15]. According to Stankovic - "..., the most important property of a real-time system should be predictability; that is, its functional and timing behavior should be as deterministic as necessary to satisfy system specifications." [15]. He goes on to clarify that while performance can aid in meeting real-time constraints, it cannot guarantee it. On the contrary, performance must often concede to meet the real-time constraints of the system [15],

[16]. Reghenzani et al. formulate the same in a more direct manner: [1] note: *“Real time does not mean computing as fast as possible, but as fast as required.”*

Real-time operating systems (RTOS) have evolved to make developing complex systems with real-time constraints easier than using a bare metal approach. [1]. An operating system’s job is to schedule how much time a task is given on the CPU, with the lowest possible overhead, while providing fundamental mechanisms for task synchronization and communication. An RTOS has the added requirement of being deterministic, guaranteeing the response times to an event. [1]

To achieve this, RTOSs are typically preemptive, provide mechanisms for prioritization of tasks, and implement priority inheritance or other solutions to priority inversion for synchronization primitives [17]

2.2 Linux

2.2.1 Embedded Linux

There is a general trend in the embedded computing domain toward placing more functional requirements on products and an increased focus on energy consumption [18]. At the same time, the complexity of hardware has increased, and the increasing competition in many sectors has led to a desire for an ever shorter time to market. This evolution has led to increased use of Commercial Off-The-Shelf (COTS) HW and SW in the industry [1], [13]. Multicore CPUs and several caching layers also mean that the requirements for simple RTOSs grow more extensive to enable systems to take advantage of the increased HW capabilities [13].

Because of this, the use of Linux in embedded devices has increased dramatically over the last two decades [1], [3], and it is today the most prevalent OS in the domain [19]. Linux is open source, has support for a wide range of architectures and hardware, as well as large amounts of existing software, toolchains, and community support [13], [14]. These advantages have led to reduced development costs and a shorter time to market for businesses [1], [3]. The company’s efforts to align its products to the same Linux-based SW platform highlight this fact.

Widely accepted knowledge forms the basis for the remainder of this subsection. Therefore, citations are limited, but the information adheres to [14] and [13].

The sum of everything required to build and run a custom Linux distribution on a hardware platform is called a Board Support Package (BSP) [13]. On ARM platforms, this includes the following:

- Toolchain
- Bootloader
- Linux Kernel
- Rootfs
- Device Tree

The toolchain includes a cross-compiler and standard libraries. It allows the developer to build the remainder of the BSP and any applications that should execute on the system.

Most modern SoCs capable of running Linux will have a small bootloader provided by the vendor placed in Read Only Memory (ROM). These bootloaders are limited and can not boot Linux on their own. They will only bring up the most essential pieces of hardware before handing over control to a more capable bootloader chosen by the developer. This bootloader loads the other BSP artifacts into memory before starting the Linux kernel.

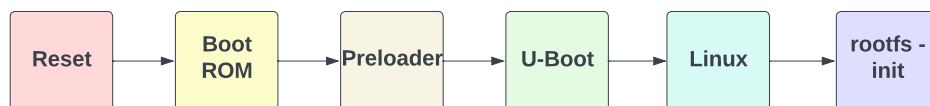


FIGURE 2.1: Embedded Linux boot sequence.

The kernel runs on all sorts of hardware. Typically, in x86 systems and similar, the kernel utilizes the basic input/output system (BIOS) to acquire information regarding the hardware constellation. Other hardware extensions are typically attached using USB, which offers run-time detection. In contrast, for embedded devices, the hardware platform and peripherals were previously configured at compile-time through header files included in the build, essentially making a compiled kernel only valid for a particular target.

The device tree, a data structure describing all the hardware and peripherals on a particular target, solves this problem, at least for ARM platforms. It is loaded into memory by the bootloader and parsed by the kernel during initialization. Based on the contents of the device tree, the kernel will load relevant drivers, configure the hardware, create virtual files for interaction with them, and set up SMP if running on a multicore system.

Before building the kernel, many compile-time parameters are configurable. These parameters include architecture-specific settings, the preemption model, debugging and functionality, and more. The kernel build system also compiles Loadable Kernel Modules (LKMs), either in-tree or out-of-tree. The concept of LKMs simplifies injecting custom hardware drivers and other software components that should execute in kernel space. In Linux, memory is divided into two parts, whereas one requires privileged access. Processes executing in kernel mode can access both parts, but processes executing in user mode do not have privileged access to ensure they will not interfere with the system or each other. When a user space process needs to access hardware, it makes a system call, which triggers a kernel module to perform the required work before the user space process can continue.

Linux is not a complete operating system in the sense that many view operating systems. It only provides the kernel. This aspect means that for a system to be able to do something, it needs something more. When the bootloader jumps to the kernel, it passes several parameters, including a root file system and a path to a binary in user

space for the kernel to execute after initialization. In theory, this could be any application; however, that would diminish most of the benefits of using Linux. Instead, it commonly provides a path to an init *daemon*¹ for starting and managing other applications throughout the system's run time.

There are no requirements regarding the structure of the root filesystem, but it should adhere to the Unix conventions concerning directory structure and location of standard utilities and libraries. Bundled with a Linux distribution, there are many tools a user might expect to find. Many originate from the GNU project, spearheaded by Richard Stallman and maintained by the Free Software Foundation (FSF). Because of the limited functionality of the kernel itself and its dependence on GNU utilities to be considered a complete operating system, Stallman has argued that Linux distributions should instead be referred to as GNU/Linux systems [20]. Pulling, building, and bundling up all packages and shared libraries for a custom Linux distribution can be tedious if done manually. Therefore, developers typically use automated build systems like Yocto and Buildroot to create their custom Linux distributions, entirely or partially.

Configuring the artifacts for a BSP is done through similar mechanisms. These repositories typically contain predefined configuration files often provided by the hardware vendor. Further refinement of the configuration files is possible through tools like *menuconfig*, illustrated in Figure 2.2.

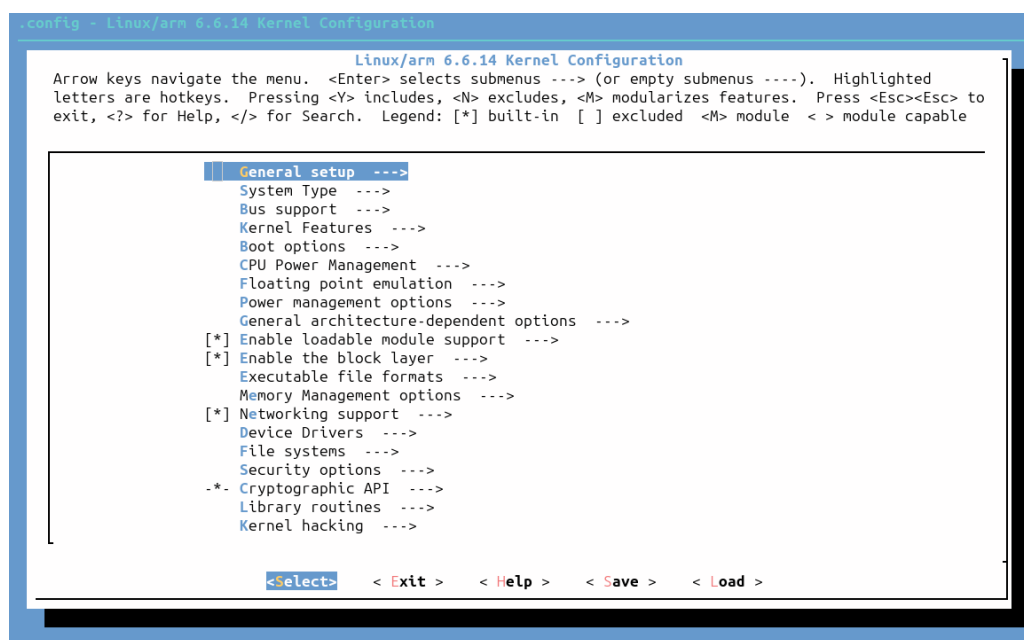


FIGURE 2.2: Illustration of kernel configuration through *menuconfig*.

2.2.2 Scheduling

This subsection discusses scheduling mechanisms in Linux, a key concept in this thesis. Unless otherwise stated, the information is widely accepted and based on [21] and [22].

¹Background service process

A *process* is an executing instance of a program that contains one or more threads. Threads within a process share a virtual memory space but have separate stacks, stack counters, and program counters. Linux schedules threads, but in real-time literature discussing scheduling, *task* is the common term. Therefore, a task and a thread can be considered synonymous within this thesis.

A task can be in any number of states, as seen from the scheduler's point of view. This thesis will confine its descriptions of a task's state to a simple and generic state machine, as seen in Figure 2.3. The model does not consider initialization states and cleanup states. A *ready* task is ready to execute on the CPU; a *blocked* task is either sleeping or waiting for a resource to become available, while a *running* task is currently executing.

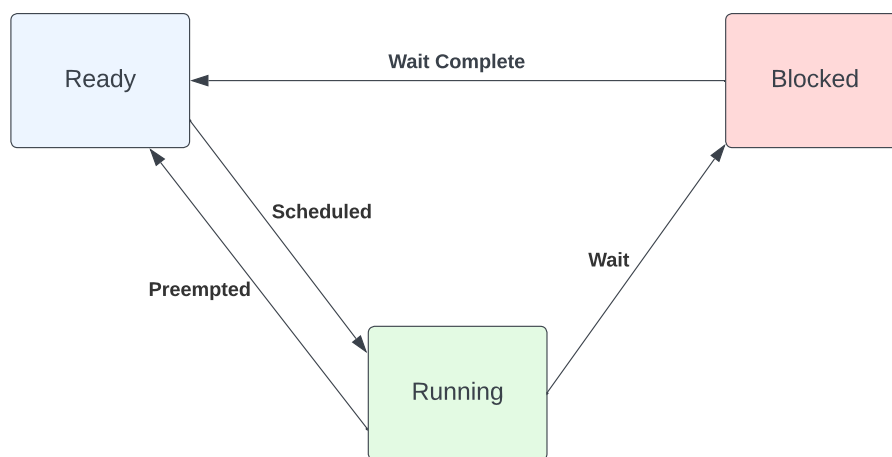


FIGURE 2.3: Simplified state machine for tasks.

The Linux scheduler comprises several scheduling classes, each containing one or more scheduling policies. Although special classes do exist, the following are the ones under which a user can choose to have his task scheduled:

- Deadline (DL)
 - SCHED_DEADLINE
- Real-Time (RT)
 - SCHED_FIFO
 - SCHED_RR
- Completely Fair Scheduler (CFS)
 - SCHED_OTHER
 - SCHED_BATCH
 - SCHED_IDLE

Although Linux is not POSIX [23] compliant, instead based on the Linux Standard Base [24], according to Locke, “*from a practical standpoint, Linux exhibits a very high degree of POSIX compliance*” [25]. The scheduling policies SCHED_FIFO, SCHED_RR, and SCHED_OTHER originate from POSIX [23], underlining Locke’s statement.

The selected scheduling policy for a task decides a high-level priority and its conditions for preemption. When the scheduler cycles through the task run queue, it looks through the classes in the order given above and stops when it finds a runnable task. Consequentially, if a runnable task exists in the deadline class, it will have a higher priority than a runnable task in the real-time class. If no runnable tasks exist in any classes, the kernel will schedule the special *idle* task.

Deadline Scheduling

Deadline scheduling in Linux uses an implementation of the Earliest Deadline First (EDS) algorithm paired with Constant Bandwidth Server (CBS). [26] A user wanting to utilize this policy must specify the desired period, deadline, and Worst-Case Execution Time (WCET) of his task. The request will fail if the provided timing constraints for the task are not schedulable (i.e., achievable). Under this scheduling policy, tasks do not have an explicit priority. Instead, the scheduler will pick the task with the current shortest deadline.

A task that runs for longer than its registered WCET will get throttled. This trait means that the developer must calculate the WCET in advance for all EDS tasks in order for the system to function correctly. This process is both error-prone and tedious. When a task finishes its execution of the current cycle, calling `sched_yield()` will voluntarily give up the remaining CPU time.

Real-Time Scheduling

The real-time policies SCHED_FIFO and SCHED_RR have static priorities between 0-99. The scheduler prioritizes the two policies equally and places the highest prioritized runnable task on the CPU. If no higher-priority tasks are runnable, the current task will execute until it gives up the CPU voluntarily. However, if two SCHED_RR tasks have the same priority, they will be subject to round-robin time-slicing according to a preset CPU time, which defaults to 100 ms.

Section 3.6.3 discusses these policies further.

Completely Fair Scheduler

The Completely Fair Scheduler (CFS) is a time-slicing scheduler that tries to give all tasks a fair share of CPU time. The default scheduling policy for a task is SCHED_OTHER (as defined by POSIX [23]), also referred to in the context of Linux as SCHED_NORMAL [21]. These tasks default to a static priority of 120, which the developer can adjust up or down 20 increments through its *nice* setting. A high nice value indicates that the task is nice to other tasks, effectively lowering its priority.

According to [27, kernel/sched/core.c, Line 11534], nice levels are multiplicative, such that stepping the niceness up or down will result in an approximate 10% decrease or increase of time on the CPU.

The recommended use case of `SCHED_BATCH` is for CPU-intensive tasks, and the scheduler gives it a higher CPU time penalty than `SCHED_OTHER` tasks. `SCHED_IDLE` is the policy with the lowest priority, and these tasks will only run when the CPU is idle, meaning they might never run.

Earliest Eligible Virtual Deadline First

In Kernel version 6.6, an implementation of Earliest Eligible Deadline First (EEVDF) replaced CFS as the core scheduler for user space tasks [27, `kernel/sched/fair.c`, Line 857]. Although this change brings several benefits, overcoming latency issues with CFS is the primary justification [28].

Tasks with low execution times and high-frequency periods are particularly disadvantaged by CFS. Patches implementing latency nice levels have been around since 2019 [29]; however, CFS is, at its core, unfit for such an add-on, making the solution less than ideal [28].

EEVDF bases itself on the concepts of *lag* and *virtual deadline* [30]. Although the CPU tries to schedule all tasks fairly on a CPU, after a run-through of the queue, some tasks will have gotten more than their share, while others will have gotten less. The scheduler calculates the lag for all tasks, which is the difference between the theoretical fair CPU time and the actual CPU time received. The only tasks eligible to run next are those with a positive lag, prioritized by the highest lag time to the lowest. EEVDF prioritizes uneligible tasks by their virtual deadline, which is the task's *normal* time slice (as given by nice values) plus the time until it becomes eligible (i.e., time until its lag becomes zero or higher).

This scheme fits better with the concept of latency nice levels [28]. A task with a lower latency nice value will have a smaller time slice. This results in the same amount of CPU time but spreads across many shorter time slices on the CPU.

Kernel Threads & CPU Modes

The default policy of a kernel thread is `SCHED_OTHER` [27, `kernel/kthread.c`, Line 369]. For the most part, this default setting applies to all tasks created by the kernel on a typical Linux system. Exceptions to this include *migration* tasks that have a priority 99 under `SCHED_FIFO`. They have one instance per CPU and move threads to other CPUs if they detect imbalances across the run queues. In addition, the bottom halves of threaded IRQs run at priority 50, also under `SCHED_FIFO`. This topic is discussed further in sections 3.3.3 and 3.6.3.

The difference between a kernel thread and a user space thread is the CPU privilege level under which it executes. Most CPU architectures enforce at least two privilege levels: user and kernel mode. The former may restrict a task in many ways, such as the address space it can access and the instructions it can execute. In Linux, the mode of the thread determines when a thread is subject to preemption as well. System calls will temporarily escalate privileges for a user space thread, which means that the thread will execute in kernel mode for the duration of the system call.

It is important to note that while the priorities provided by the Linux documentation are used internally in the Kernel source code, they do not necessarily reflect how various user space tools choose to present task priorities. An example is *htop*, which presents real-time priorities between -2 and RT. The lowest priority, 2, is equivalent to 99 in the literature but is achieved by configuring a task with a priority of 1. Passing the value 98 to a system call gives -99, and passing 99 gives RT. CFS tasks are typically in tools like *htop* in the range between 0 and 40, directly representing the nice level.

2.2.3 Real-Time Linux

Parallel to the increased use of Linux in embedded systems, there has, since the mid-90s, been a desire to use Linux in real-time systems [31]. Since Linux is a general-purpose operating system, its primary goal is to ensure that all tasks collectively have the highest throughput possible, maximizing CPU utilization. Although there are mechanisms to prioritize specific tasks, the kernel is (or, more precisely, was) not preemptible.

The developer can set a preemption model when configuring the kernel at compile time. In most cases, this defaults to `PREEMPT_NONE` [27, `kernel/Kconfig.preempt`, Line 18], which means that the following situations will invoke the scheduler [32]:

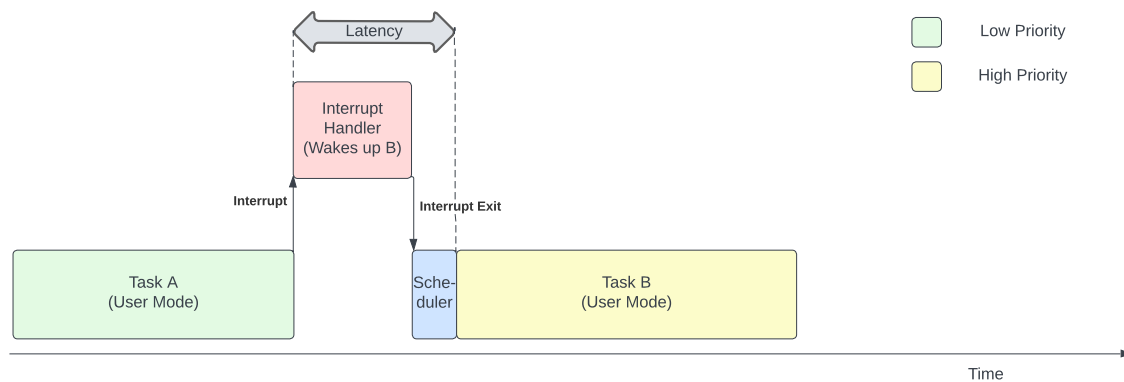
- When a task gives up the CPU voluntarily.
- When the CPU returns from hard interrupt context to a task executing in user mode.
- When a system call returns to a task executing in user mode.

In other words, a user space task cannot preempt a task executing in kernel mode, regardless of its scheduling policy and priority.

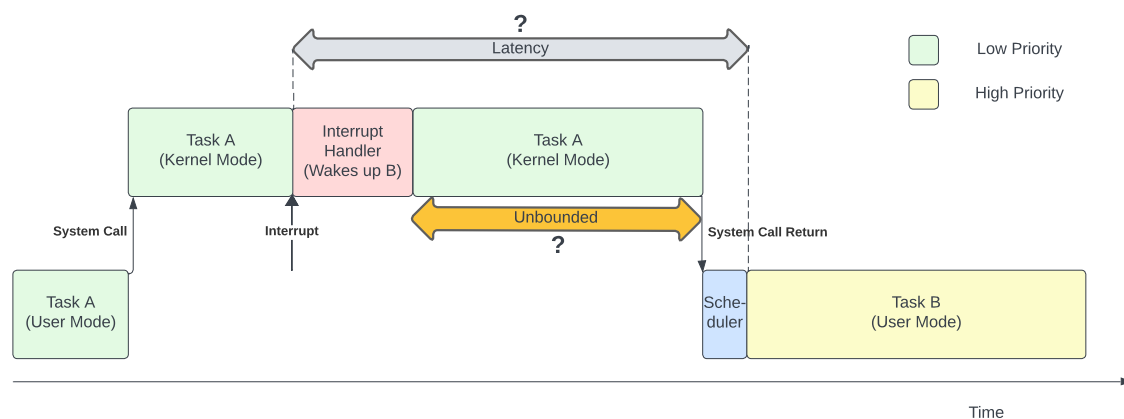
This characteristic means that scheduling latencies² are unbounded and that the kernel cannot guarantee that a particular task will execute within its given deadline.

Figure 2.4 illustrates this. In the first example, the low-priority task A runs when an event tries to wake up the high-priority task B. The kernel supports preemption of tasks executing in user mode, so the scheduler steps in and gives the CPU to task B. However, in the second example, task A performs a system call and executes in kernel mode when the event occurs. Because of this, task A can continue executing in kernel mode for as long as it likes, hindering the potentially critical task B from doing its job.

²Scheduling latency is the delay from when a task becomes runnable until it starts executing on the CPU [33].



(a) Task in user mode preempted.



(b) Task in kernel mode not preempted.

FIGURE 2.4: Preemption of user mode vs kernel mode task.

In addition, the kernel is a highly non-deterministic beast [34]. The many non-predictable execution paths a single system call can have may lead to high jitter. These issues and others are discussed further in Chapter 3.

Many approaches and projects have evolved to overcome these issues, and they broadly divide into two categories [1], [3]:

- Single-Kernel
- Co-kernel (microkernel)

Co-Kernel

In a co-kernel approach, a bare metal scheduler, or more commonly an RTOS, runs underneath Linux and manages it as any other task. Figure 2.5 illustrates this. This scheme allows critical tasks to run outside the context of Linux, and the Linux kernel is naturally always preemptible. Although providing provable *hard* real-time capabilities

[35], this approach diminishes some of the significant benefits of using Linux in the first place [1], [35]. The Linux kernel must be modified to run within the context of the microkernel. This approach often involves forked or outdated kernels, which do not benefit from regular mainline upstream updates and fixes. In addition, another layer of hardware abstraction must exist beneath the Linux kernel, adding complexity and required maintenance. Real-time tasks also lose the ability to utilize kernel drivers or other utilities [1].

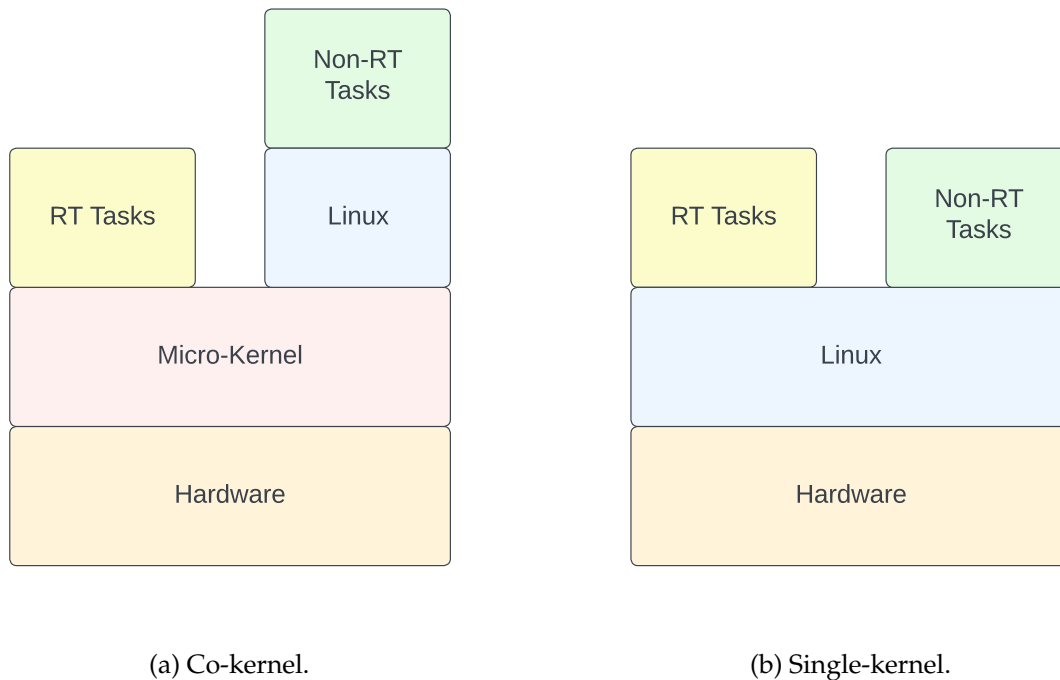


FIGURE 2.5: Co-kernel vs single-kernel approach.

Single-Kernel

While more straightforward in concept, the single-kernel approach is more ambitious and tries to transform the Linux kernel into an RTOS [2]. The PREEMPT_RT project is the most successful effort in this domain [1], [33]. It is backed by the Linux Foundation today, and most of the original contents of the patch have already merged into the mainline kernel³.

The approach of making Linux itself real-time capable means that one gets to keep all the initial benefits of using Linux. No special tools or libraries are required. It is more a matter of tuning the system with different kernel compile-time and run-time configurations [1]. However, it is not a *“silver bullet”* [36]. One still has to design a real-time application to meet its requirements as one would expect when using bare metal or RTOS approaches [2]. Also, some drivers and kernel mechanisms should be avoided as they have not necessarily been made real-time friendly yet [9].

Despite this, bare metal, RTOS, or co-kernel approaches are the only way to achieve provable hard real-time [1], [2]. Currently, the literature considers it impossible to

³Version 2.6.22.1-rt9 had 61595 lines of code, while 6.6.25-rt29 has 15494 lines of code

prove that a task will always meet its deadlines due to the complexity of the Linux Kernel [1], [37]. The intricacy of the embedded HW devices that typically run Linux makes this problem even more difficult. Modern computer architectures with several caching layers and mechanisms like instruction reordering mean that a computer is not as deterministic as it once was [2].

With PREEMPT_RT, estimating a system's capability of meeting its real-time requirements is typically done by experiment, where long-term tests are executed over weeks or months [1], [2].

2.3 Target System

This section attempts to describe the target system at a high level and discuss why it might benefit from running with a fully preemptible kernel while trying to maintain the integrity of the intellectual property belonging to the company.

The target system is a powerline modem and will run on a custom HW board with an Altera Cyclone V. The Altera Cyclone V is a Fully Programmable System on Chip (FPSoC), meaning that it has a Field Programmable Gate Array (FPGA) as well as a Hard Processor System (HPS), consisting of a dual-core ARM Cortex-A9. Figure 2.6 displays a simplified system architecture.

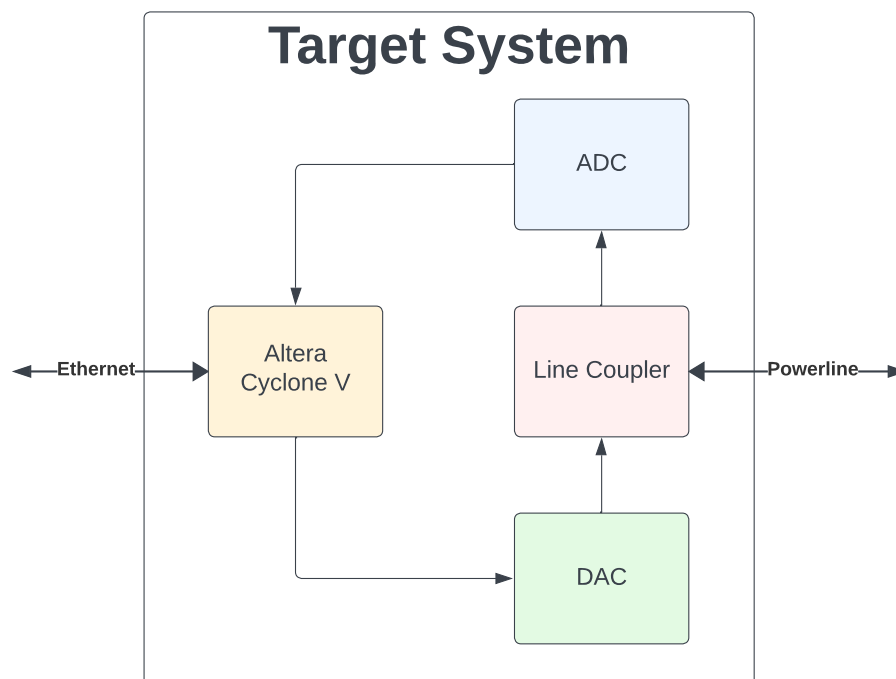


FIGURE 2.6: Target system.

The target system is not necessarily an inherent real-time system. Instead, specific mechanisms have real-time constraints associated with them. These constraints must

be met for the system to achieve the desired performance. Missing a deadline will not have fatal consequences but rather degrade the performance, meaning it fits within the definition of a soft real-time system.

When transmitting, the modem will use modulation algorithms to transform Ethernet frames into data that can be outputted to a powerline using a Digital-to-Analog Converter (DAC). For receiving data, it will drive an Analog-to-Digital Converter (ADC) to sample the powerline at a steady rate. The system decodes the samples to Ethernet frames and outputs them to the local network.

This project's scope does not include the modulation and demodulation algorithms that make up the system. Instead, it involves analyzing and optimizing the system's ability to move data across the SW and HW modules at the required rate without interference from the operating system.

Much work is offloaded to the FPGA, illustrated in Figure 2.7. The FPGA is responsible for driving the analog converters and applying filtering on the data as they pass through.

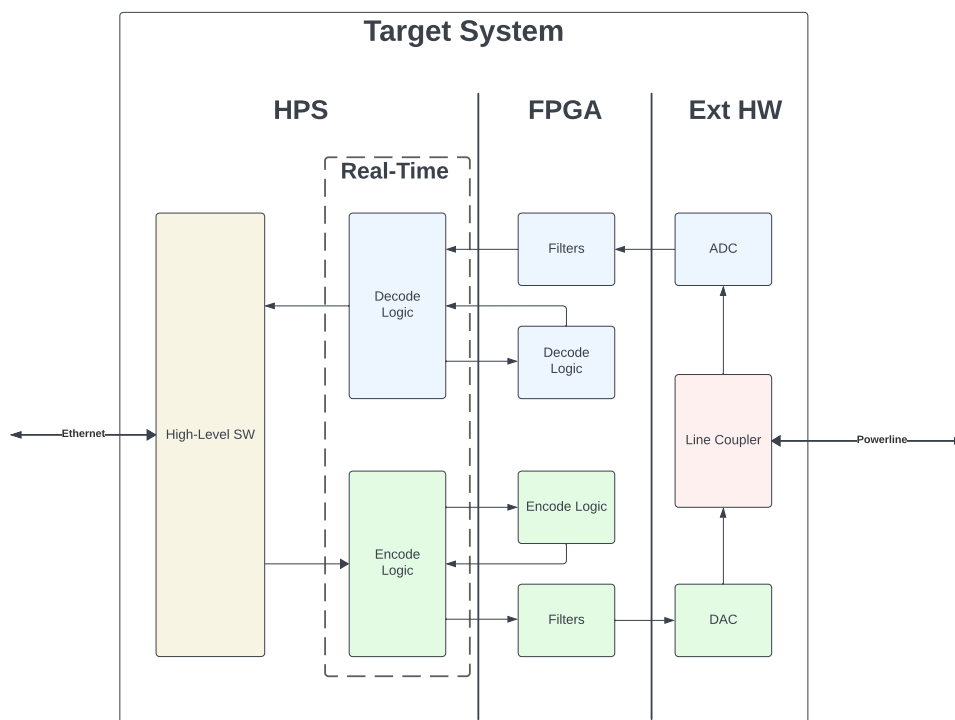


FIGURE 2.7: High-level architecture of the target system.

During reception, the FPGA will write data into a shared buffer in the external memory, and it is the job of a real-time task running on the HPS to keep up with the FPGA and empty the buffer at a rate sufficient to avoid overruns. The same principle applies when transmitting. The FPGA will read a shared buffer at a fixed rate, and the software in the HPS has to ensure that underruns do not occur. This mechanism is highlighted in Figure 2.8.

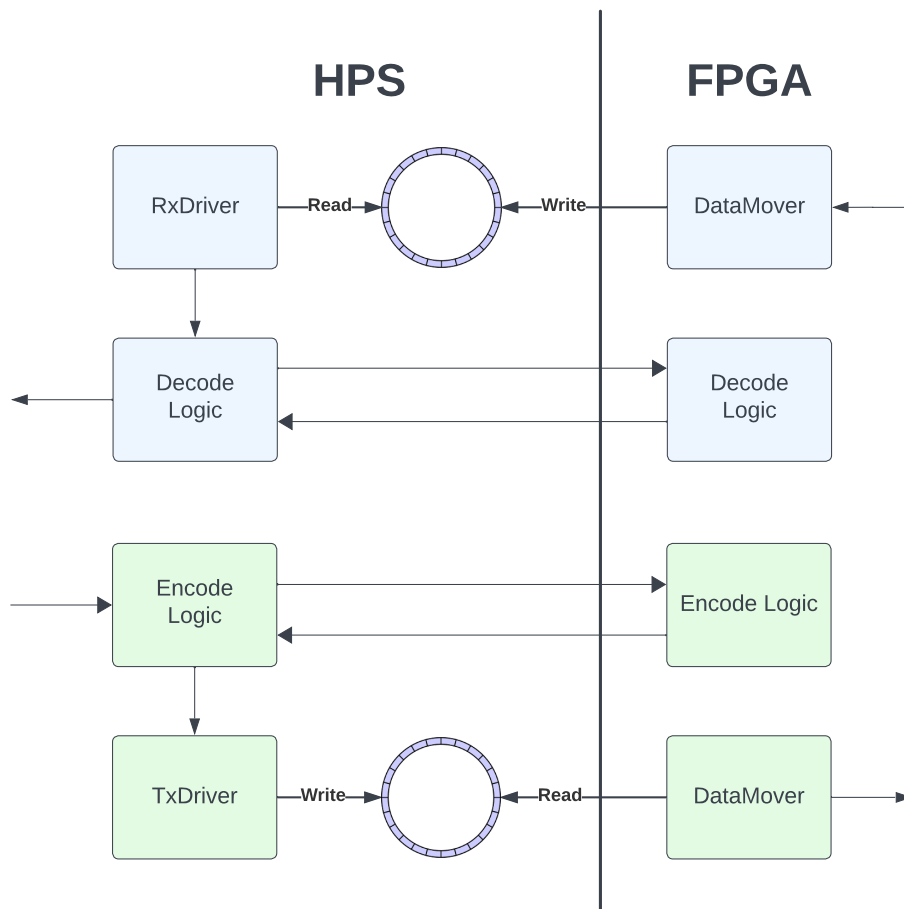


FIGURE 2.8: HPS-FPGA shared memory interface.

The modem's decoding and encoding logic also has a logic component placed in the FPGA to accelerate the algorithms. This means that data will also need to be written to and read from the FPGA during the signal processing, putting additional load on the system and the communication channels between the HPS and FPGA.

2.3.1 Hardware

As discussed in Chapter 1, the target system runs on an Altera Cyclone V, for which a simplified block diagram can be seen in Figure 2.9 [38, Copyright © 2017, IEEE].

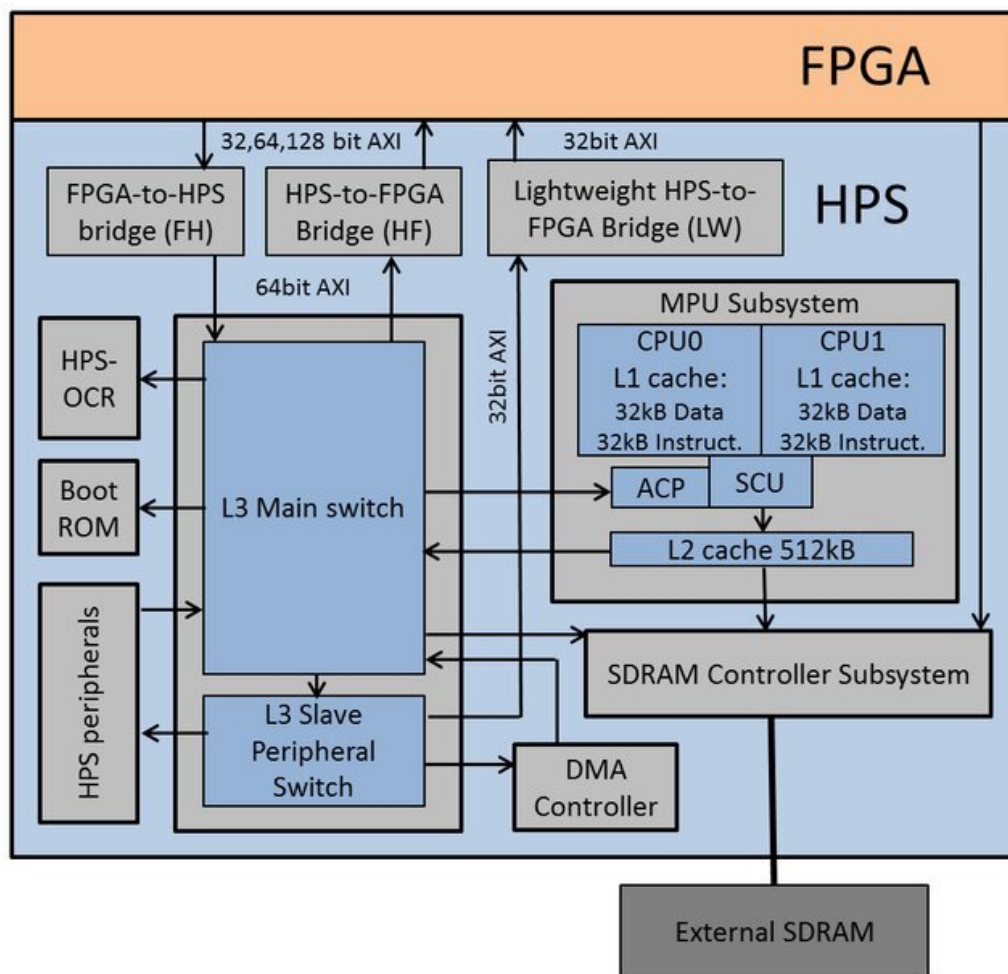


FIGURE 2.9: Simplified Altera Cyclone V block diagram [38] Copyright © 2017, IEEE.

The SoC has several mechanisms for communication between the CPU and the FPGA [39]. The bridges are AXI-buses with an overbuild called Avalon, a concept specific to Altera.

- **FPGA-to-HPS bridge:** Allows the FPGA to write up to 128-bit words directly to a peripheral in the HPS, accessible through the L3 Main Switch [39], [40].
- **HPS-to-FPGA bridge:** Allows the CPU to access any memory-mapped peripherals in the FPGA with up to 64-bit words. Originating in the L3 Main Switch [39], [40].
- **Lightweight HPS-to-FPGA bridge:** Same as the HPS-to-FPGA bridge, but strictly for 32-bit words, and passes through the L3 Slave Peripheral Switch instead of the L3 Main Switch. It is more lightweight, making it suitable for simple control registers [39], [40].

- **FPGA-to-SDRAM-Controller:** Allows the FPGA to write to SDRAM through the SDRAM-Controller Subsystem [39], [40].

The target system will use the HPS-to-FPGA bridge, configured for 32-bit words, to write control registers and to read and write data to the decode/encode logic. The FPGA will access the circular buffers in shared memory through the FPGA-to-HPS bridge, configured to 64-bit words, the L3 Main Switch, and the Accelerated Coherency Port (ACP). This design choice ensures that the L2 cache will be updated on writes from the FPGA, keeping it coherent with the L3 cache. Although this slightly increases the writing overhead from the FPGA's point of view, it will reduce the read time for the CPU.

2.3.2 Software

To keep the software as simple as possible and to avoid the context switch required when switching to kernel space, it is desirable to have the circular buffers in memory accessible from user space. A Continuous Memory Allocator (CMA) driver by Novickis et al. [40] will be used. This driver allows a user space task to allocate a continuous memory area and access it using a virtual memory address. Its interface allows cached or uncached memory access, and the software forwards the area's physical address to the FPGA logic at initialization time.

Although the low-level software responsible for modulation and demodulation will execute in user space, high-level Linux utilities running partially in kernel space will handle IP traffic toward the ethernet interface. The channel to this layer will be through a kernel driver, interfaced through Generic Netlink sockets, the modern communication mechanism between user space and kernel space [41]. These concepts will be explored further in section 4.4.

2.3.3 Real-Time Constraints

The FPGA will drive the ADC and DAC at 2.5 Ms/s with 12-bit samples. The decimation/interpolation in the filtering will vary by the product configuration and can be as low as 2 in the worst case. This attribute means that the DMA handler in the FPGA will read and write data to the circular buffer in shared memory at a rate of 1.25 Ms/s. For all practical purposes, each sample will consist of 2 bytes, meaning that the speed will be:

$$\begin{aligned}
 \text{samplesPerSecond} \cdot \text{bytesPerSample} &= \text{bitrate} \\
 1.25 \text{ Ms/s} \cdot 2 \text{ B} &= 2.5 \text{ MB/s} \\
 &\approx 2.4 \text{ MiB/s}
 \end{aligned}
 \tag{2.1}$$

Given that the circular buffers are 128 KiB large, this means that the filling/clearing rate will be

$$\frac{bitrate}{bufferSize} = bufferFillRate$$
$$\frac{2.4 \text{ MiB/s}}{128 \text{ KiB}} = 19.2 \text{ Hz} \tag{2.2}$$
$$\approx 20 \text{ Hz}$$

This means the buffers will be filled or cleared once every 50 ms by logic on the FPGA. The constraint on the software is to keep up with these rates and avoid underruns when writing and overruns when reading. The fear is that the non-deterministic nature of Linux will prohibit the system from reliably meeting these requirements and is the primary motivation for the problem statements of this thesis.

Chapter 3

PREEMPT_RT

This chapter is a literature review focusing on the PREEMPT_RT patchset for the Linux kernel. It provides an overview of PREEMPT_RT's history and main features while it examines relevant literature for key performance measures and documented real-world applications of the patchset. In addition, it summarizes tuning parameters and best practices extracted from the literature.

3.1 Literature Review Methodology

Kernel developers have traditionally not been the most prolific writers of academic papers. To cite Thomas Gleixner, co-developer and long-time maintainer of PREEMPT_RT: *"We are solving problems, comparing and contrasting approaches and implementations, but we are either too lazy or too busy to sit down and write a proper paper about it"* [42]. Instead, they participate in mailing lists and write documentation. Some might also write articles, blog posts, or give talks. Because of this, this thesis cites sources like Linux Weekly News, slide decks from Linux conferences, the Real-Time Linux Wiki page, and official Linux Kernel documentation. However, from non-peer-reviewed sources, no performance claims are cited. Reghenzani et al. [1] use a similar approach and justification.

For searching and gathering relevant literature, a structured approach [43] was used, querying the following databases:

- IEEE Xplore
- Springer
- ScienceDirect
- Web Of Science
- Wiley
- Mendeley

The following terms made up the basis for the search:

- (a) "PREEMPT_RT" || "PREEMPT RT"
- (b) "Linux" && "real-time" && "ARM" && (year >= 2019)
- (c) "Linux" && "real-time" && "Altera Cyclone V"

Figure 3.1 illustrates the structured search results, showing the number of articles found in each database for each search term. The findings provided by Google Scholar were so numerous that handpicking relevant articles from the first few result pages became the only viable approach. A citation manager pulled the results of the remaining databases in their entirety. The second search term gave too many results in several databases, so the keyword *ARM* was refined to *ARM Cortex-A9* to limit the results.

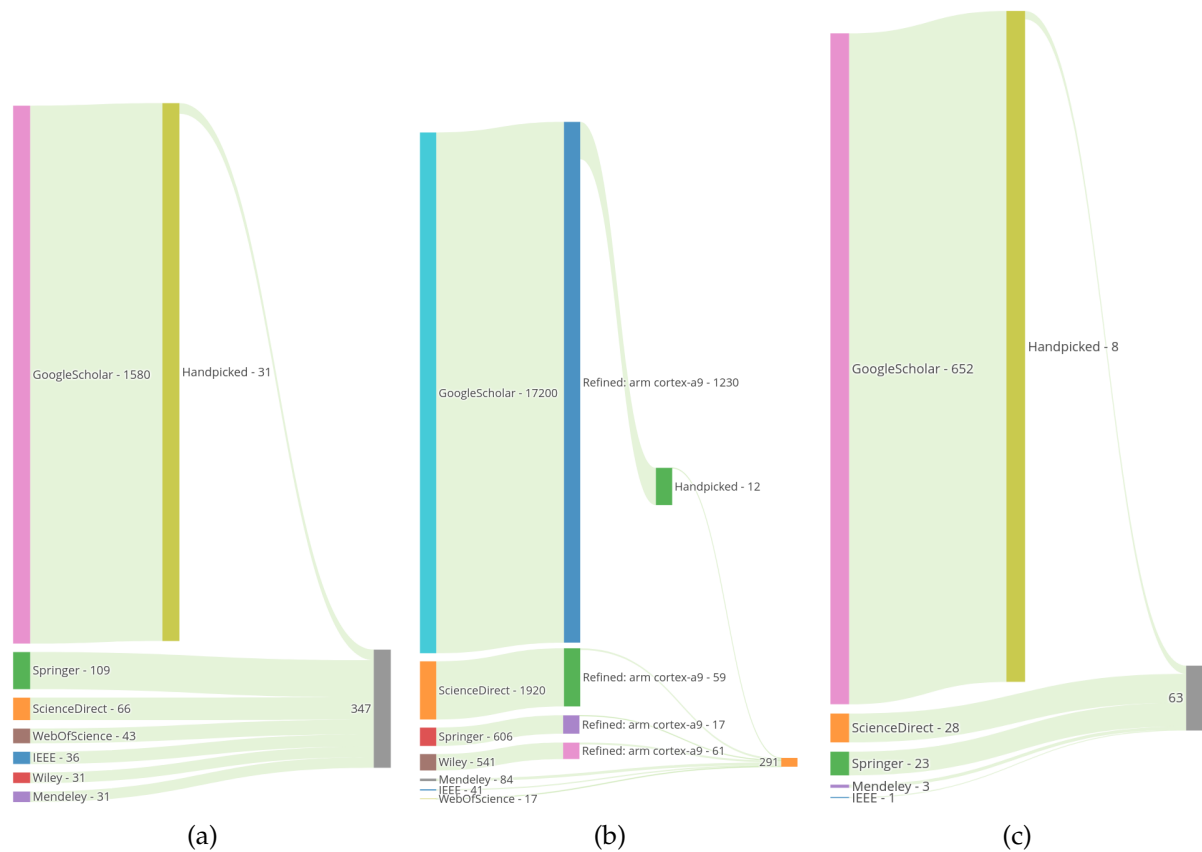


FIGURE 3.1: Structured literature search.

After dumping the findings into a reference manager, duplicates were removed, resulting in 547 unique articles, as seen in Figure 3.2. From there, 304 articles were discarded based on the title and abstract, and another 230 after skimming through the paper. This approach resulted in a total of 40 articles being deemed relevant.

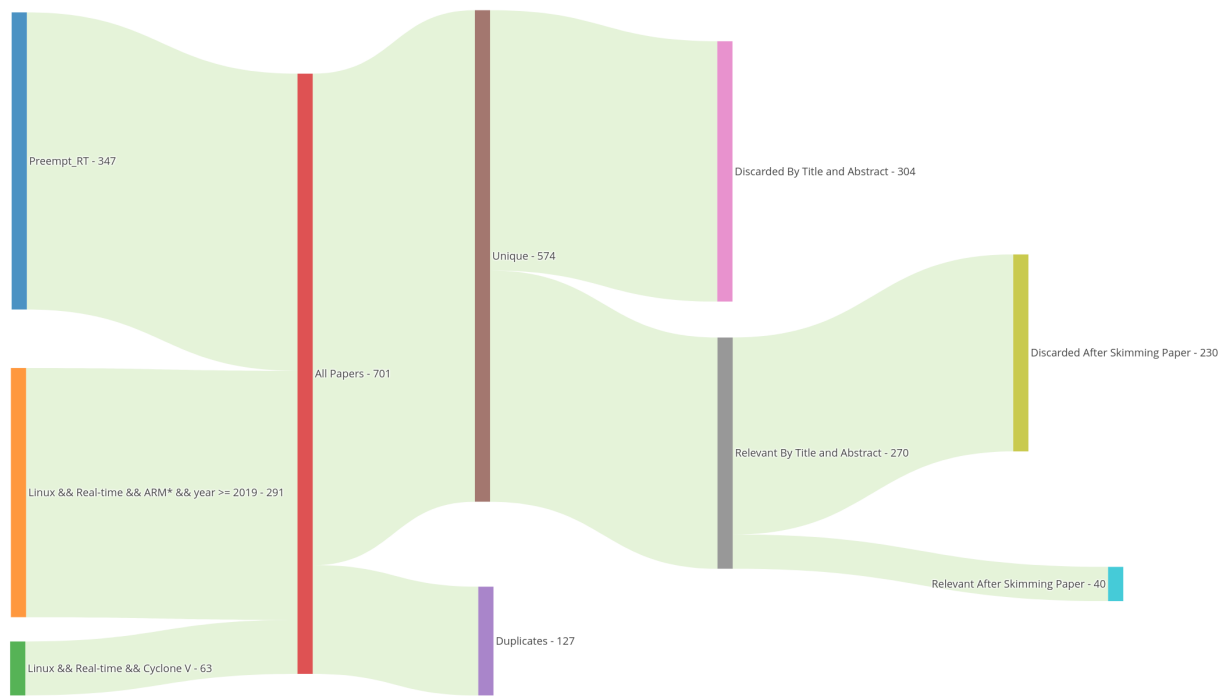


FIGURE 3.2: Refining search results.

Additionally, *snowballing*¹ was used to find more articles throughout the research period, resulting in the final literature list included in the bibliography.

3.2 High-Level Overview

PREEMPT_RT is a patch set for the Linux kernel published by Ingo Molnar and Thomas Gleixner in 2005 [44]. The patch rewrites the kernel in order to make it real-time capable. Major features introduced by the patch include:

- Fully Preemptible Kernel
- Threaded IRQs
- High-Resolution Timers
- Priority Inversion
- Full Tickless Operation

3.3 Main Features

This section explores the main features and modifications to the kernel introduced as part of PREEMPT_RT. Although all these are merged into the mainline kernel today, most are turned off or hidden on a mainline Linux system by default.

¹The process of following citations in the current collection of literature to find additional relevant studies.

3.3.1 Fully Preemptible Kernel

Today, the mainline Linux kernel offers three compile-time options for preemption [32]:

- No Forced Preemption (Server)
- Voluntary Kernel Preemption (Desktop)
- Preemptible Kernel (Low-Latency Desktop)

As discussed in section 2.2.3, the default compile-time configuration is *No Forced Preemption*, meaning that user space cannot preempt the kernel. *Voluntary Kernel Preemption* adds preemption points to the codebase [1]. Although this can limit the latency experienced by a high-priority task executing in user mode, this reduction in latency is linear to the number of preemption points in the source code, as is the overhead of the preemption points themselves. This characteristic could indicate that the solution might not scale well.

The *Preemptible Kernel* mostly solves the issue illustrated in Figure 2.4 by running the scheduler after all interrupts², which means that the kernel is preemptible, except when in critical sections [1]. Figure 3.3, illustrates this.

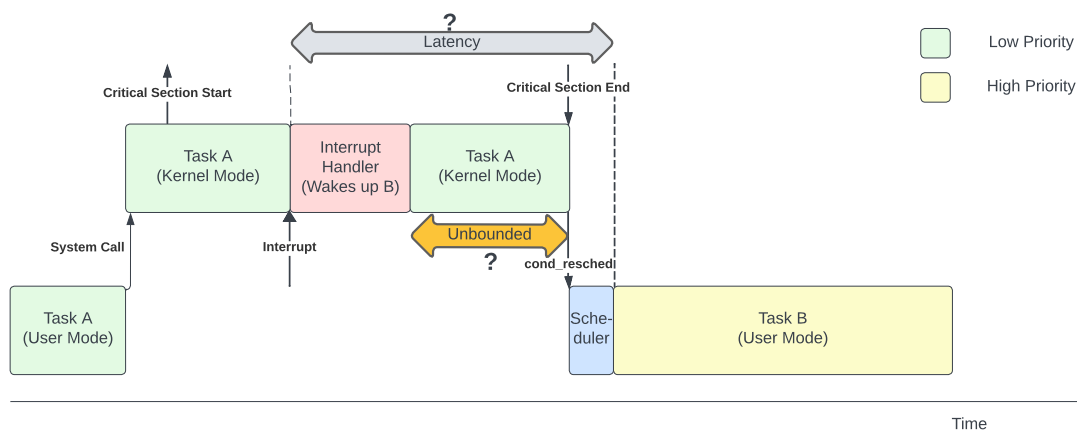


FIGURE 3.3: Preemptible kernel.

In this case, the system call is in a critical section when the event that tries to wake up task B occurs. In contrast to the previous scenario, the unbounded latency is limited to the length of the critical section and no longer the complete duration of the system call. The high-priority task will be scheduled with bounded latency if the system call is not in a critical section when the event occurs.

PREEMPT_RT enables a fourth option, which in the mainline kernel is hidden by the unset `ARCH_SUPPORTS_RT` flag [27, /kernel/Kconfig.preempt, Line 72]:

- Fully Preemptible Kernel (Real-Time)

²The implementation for ARM platforms is available in [27, arch/arm/kernel/entry-armv.S, Line 223]

This option makes the kernel preemptible in all critical sections, except for code running in hard interrupt context and sections protected by a *raw spinlock* [1].



FIGURE 3.4: Preemption models.

Kernel code waiting for mutually exclusive access to a resource typically uses a spinlock, where the CPU will busy-wait for the lock instead of blocking. Using a spinlock reduces the overhead of waiting for the lock since the lock will typically be available within a few cycles. However, as illustrated, a task holding a spinlock is not preemptible. To also make these sections preemptible, all spinlocks in the kernel are replaced by *RT-mutexes* when full preemption is enabled [45]. Raw spinlocks are still not preemptible, but they are only used in the most critical sections or when the duration of the section is negligible.

Assuming a raw spinlock does not protect the critical section illustrated in the previous example, Figure 3.5 illustrates how using PREEMPT_RT would reduce the scheduling latency of the critical task from the previous example.

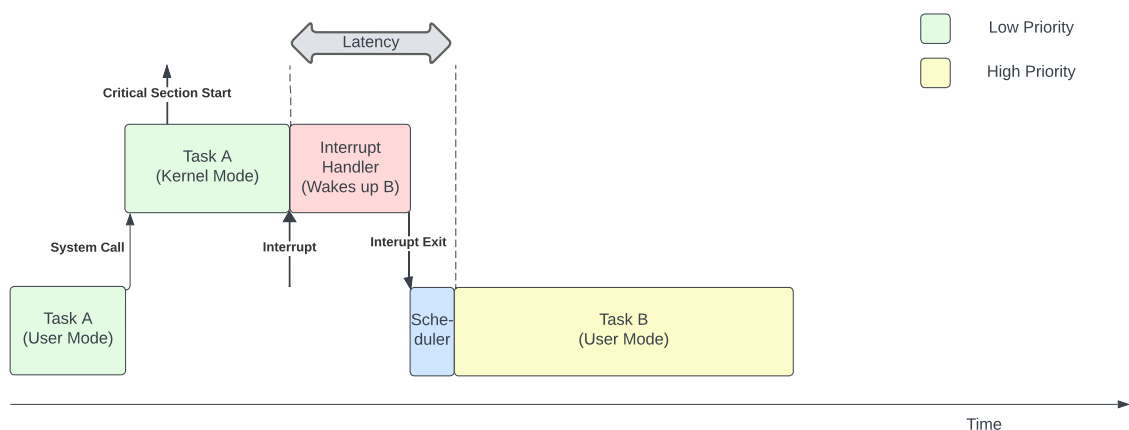


FIGURE 3.5: Fully preemptible kernel.

Traditionally, the preemption model was a compile-time parameter, but in 2021, a new option `CONFIG_PREEMPT_DYNAMIC` was introduced [46], allowing for reconfiguring the preemption model at run-time. This feature requires the architecture to support `static_call_inline`, which is unavailable for ARM [27]. Due to this, it will not be explored further in this thesis.

3.3.2 High-Resolution Timers

In Linux, the systick interval, known as a *jiffy*, is configurable at compile-time [47]. The frequency of the systick is typically 100 Hz on embedded systems and gives a jiffy of 10 ms. Before high-resolution timers made it into the kernel, the most fine-grained time resolution possible to measure was one jiffy, i.e., between 1 and 10 ms [1]. Naturally, this was a significant limitation and inadequate for many systems.

As described by Gleixner & Niehaus [48], the new implementation required creating a new timer subsystem. There was a great deal of architecture-dependent code with heavy duplication. The previous implementation was tied heavily to the periodic systick and had no general abstraction for the timers.

High-resolution timers are configurable at compile-time after the merge to the mainline kernel in version 2.6.16 [48]. The default settings for ARM platforms enable high-resolution timers³.

However, the underlying hardware still needs to support the granularity required by the system [1]. Tools like `Cyclictest`, part of *rt-tests*, an open-source suite of real-time tests [49], can validate this [50].

3.3.3 Threaded IRQs

The simplest way of handling interrupts is in *hard interrupt context*. The CPU stores its current state and jumps to a predefined address to handle the event, regardless of what it previously did. After processing the event, the CPU will restore its state and continue as if nothing happened. As illustrated in Figure 3.6, the potentially critical task already running can experience large latencies [51]. Issues such as this have led to the widely accepted notion that interrupt routines should be short, regardless of the operating system. Linux also disables interrupts while in hard interrupt context, meaning it cannot handle a new event for the duration of handling the first [51].

³All configuration files bundled with the Linux kernel for ARM enable high-resolution timers, as seen in [27, `arch/arm/configs/socfpga_defconfig`, Line 2].

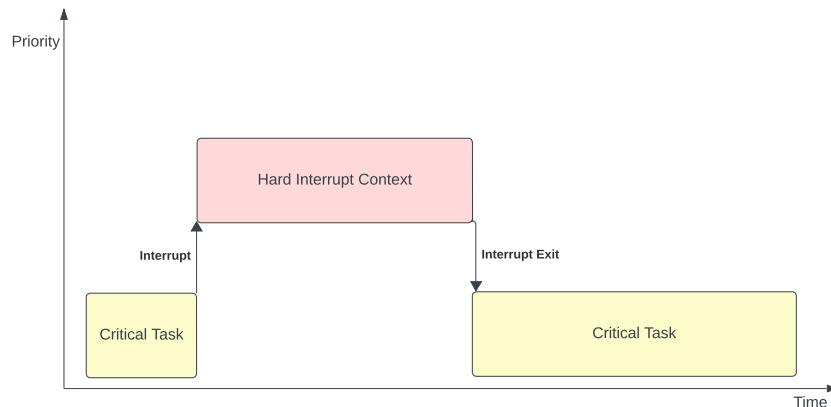


FIGURE 3.6: Traditional interrupts.

To handle these issues and to provide more flexibility, Linux has introduced several mechanisms for deferring work outside of hard interrupt context. These include *softIRQs*, *tasklets*, and *workqueues*. Common for them all is that they divide the handling into a *top-half* and a *bottom-half*. The *softIRQs* and *tasklets* typically execute immediately after the hard interrupt [51], run in atomic context, and are not allowed to sleep [52]. Concerning the critical task in the previous example, these mechanisms will not improve its experienced latency, as seen in Figure 3.7. Work queues, on the other hand, can defer the work to a later time, but they have significant overhead and may defer the work for too long, making them unsuitable for many types of events [51]

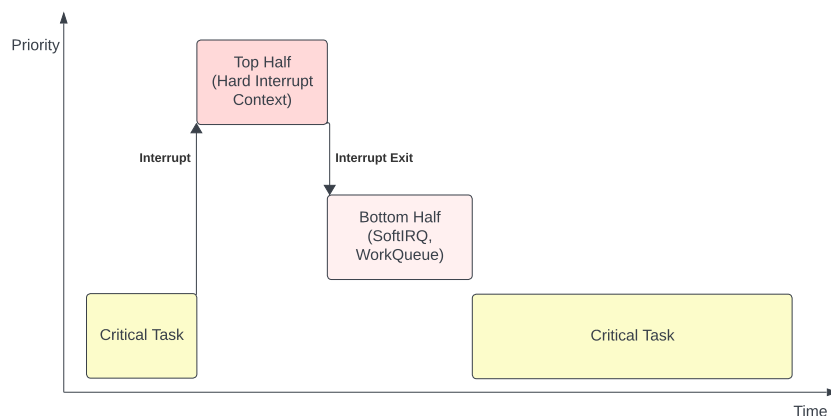


FIGURE 3.7: Top- and bottom-half interrupt processing.

Kernel version 2.6.30 introduced threaded interrupts as a new option for deferring work to a later time [1]. Although similar to work queues, they reduce the overhead, make debugging more manageable, and provide much more flexibility concerning prioritization [51], [52]. During system initialization, the kernel spawns a thread to handle

a particular interrupt's bottom half. This is a regular thread with an assigned scheduling policy and priority, meaning that if our critical task has a higher priority, it will only be delayed for the execution time of the interrupt's top half, as seen in Figure 3.8.

The division of work between the top and bottom half, as well as the mechanism used for deferring work, is decided at the discretion of the developer of a particular driver [51]. This leads different drivers and modules to have widely different impacts on the latency of a critical task.

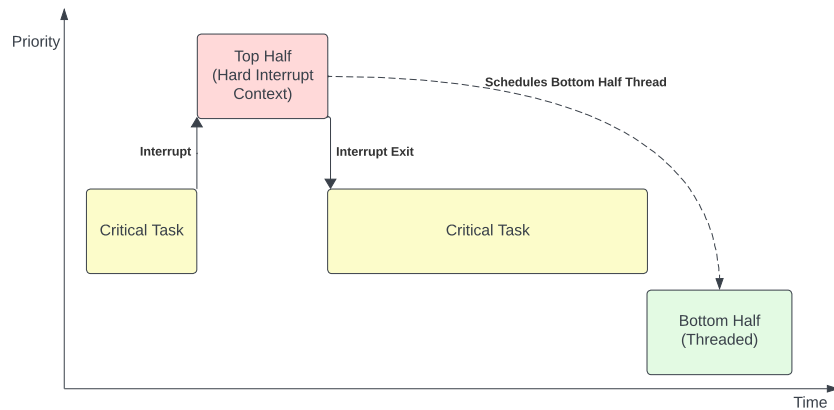


FIGURE 3.8: Threaded bottom-half.

To further improve the situation for the critical task in all conditions, PREEMPT_RT enables forced threading of all interrupts [53]. This means that all top halves are executed in a kernel thread, as seen in Figure 3.9, keeping the execution time of the hard interrupt handler as short as possible. The mechanism executing deferred work is still at the discretion of the developer.

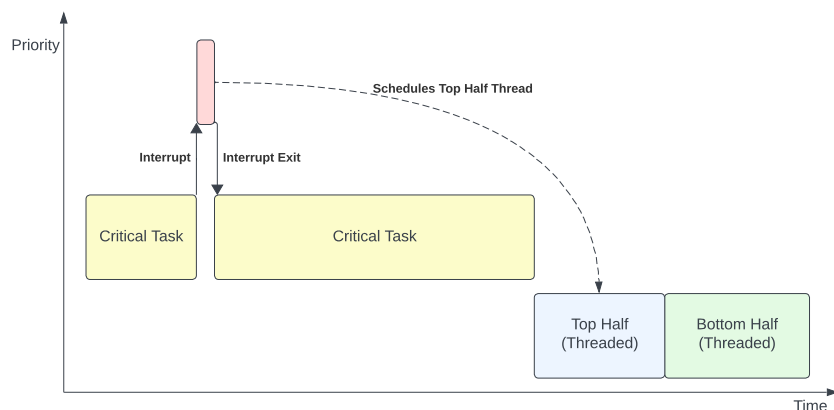


FIGURE 3.9: Forced threading.

Because of this, the latency experienced by the critical task shrinks to an absolute minimum [51]. However, a CPU-hungry task with a higher priority than a threaded interrupt handler can starve the interrupt, leading to an unstable system.

3.3.4 Priority Inheritance

Priority inheritance [54] solves the classical problem of priority inversion [55]. Priority inversion is when a high-priority task A blocks because it waits for a resource held by a low-priority task C. If B, a task prioritized higher than C but lower than A, becomes runnable, it will block C and keep executing for as long as it likes. Effectively, the priorities of A and B have become inverted. The most common solution is to boost the priority of the lower priority task C to the priority of the task it blocks. When C inherits the priority of task A, task B can no longer block C. Task C can quickly finish its work with the resource before being preempted by task A [55].

Before the PREEMPT_RT patch, the only priority inheritance implemented in the kernel was for fast user space mutexes [56], which can be configured during initialization by the user if needed. However, as the kernel was not preemptible, priority inversion was not an issue in kernel space before PREEMPT_RT [57].

According to [58], implementing support for priority inheritance in the kernel was not uncontroversial as it is very complex and easy to get wrong. However, Rostedt et al. maintain that it is well-tested and considers it proven in use.

3.3.5 Preemptible RCU

Read, Copy, Update (RCU) is a synchronization mechanism merged into the kernel with version 2.5 [59]. It solves the problem of having multiple readers and a single writer operating on a data structure in parallel on several CPUs. It has little overhead and favors readers over writers [59].

When an object is subject to deletion in a data structure like a linked list, if the writer removes the object and frees the memory immediately, there may be readers holding references to the now deleted object. The reader enters a *read-side critical section* to protect itself from accessing deleted memory, a state change that requires no overhead for the task. For the writer, deleting the object is divided into two actions: *removal* and *reclamation*. The removal phase involves removing all references to the object from the data structure. A grace period follows the removal phase, during which the writer waits until all CPUs have reported an RCU quiescent state, meaning that they are not currently in a read-side critical state. When this happens, the writer can safely free the memory, knowing that no readers hold any references to it. Registering a callback with the kernel, invoked when all CPUs have reported a quiescent state, is also possible. This feature ensures the writer does not have to block while waiting to delete the object. For this mechanism to work, it is crucial for the readers not to block while in a read-side critical state, as this can extend the grace period indefinitely. This also means that a reader can not allow preemption, which requires a different implementation of this mechanism under PREEMPT_RT [59].

The initial implementation of preemptible RCU, relying on priority inheritance, came in 2005. However, it had limitations and a high overhead implementation on the read

side. It was reworked in 2007 to use a multi-stage grace period detection algorithm [60].

A challenge with preemptible RCU is that the grace period can extend indefinitely if a high-priority task preempts a low-priority task while in a read-side critical section, effectively leading to an out-of-memory situation [61]. By default, *PREEMPT_RT* kernels enable `RCU_BOOST` [62, kernel/rcu/Kconfig, Line 197], which boosts the priority of tasks in a read-side critical section if blocked for more than a configurable amount. `CONFIG_RCU_NOCB_CPU` is an alternative approach that redirects RCU callbacks to a specified CPU, which should be less susceptible to a high-priority task that does not yield the CPU sufficiently. According to [63], this can significantly reduce the system's throughput due to the increased overhead but can make sense in certain situations, for instance, when using CPU isolation.

3.3.6 Full Tickless Operation and CPU isolation

The official Linux kernel documentation regarding CPU isolation and tickless operation is limited [64]. The Linux Foundation's documentation on real-time Linux provides some information [65] but redirects those interested to a blog post series on the topic published by OpenSUSE [66]. This has formed the basis for the information presented here.

For critical real-time tasks that require low disturbance, it is possible to isolate a CPU core for that particular task and have it run independently of the scheduler. However, as mentioned, the `systick` is an interrupt that fires on all cores at a fixed rate. This interrupt wakes the kernel to do maintenance tasks like handling timers, RCU callbacks, and CPU-time accounting, potentially interfering with the critical task. To overcome this, one can enable full tickless operation, which turns off this tick. Accommodating this required reworking several of the mechanisms relying on the `systick`.

In standard systems, CPU-time accounting checks the CPU context on each `systick` and infers how long the CPU has spent in each context. In tickless systems, the kernel stores a timestamp on each context switch. The kernel will use this timestamp to calculate how long the CPU ran in the previous context. Although this leads to more exact numbers, it dramatically increases the overhead of transitioning between the idle task, kernel space, and user space. A side effect is that the CPU can only report RCU quiescent state when executing from a user space context. This ramification means that each CPU spends less time in the quiescent state, increasing the length of the RCU grace period. Both factors reduce the system's overall throughput.

Because of this, CPU isolation with tickless operation is most efficient for tasks executing primarily in user space and requiring a limited number of system calls. At least one CPU is required to maintain a `systick`. The load on this CPU will increase with the added responsibilities from the other CPUs, like timer callbacks and work queues, reducing the throughput of remaining tasks. To utilize this, one must enable `NO_HZ_FULL` at compile-time and pass a parameter to the kernel at boot-time, specifying which CPUs are to run tickless. It is also possible to set the IRQ affinity, ensuring no unwanted interrupts trigger on a particular CPU.

Prohibiting the scheduler from putting tasks on the CPU was previously done with the deprecated `isolcpu` boot-time parameter and by setting CPU affinity for tasks, but should today be controlled through `CPU-set`.

It is only possible to run one task scheduled with CFS or `SCHED_RR` on an isolated CPU at any time. Although running more tasks in parallel will work with other policies, running only one is recommended.

3.4 Current State and Mainlining

Linux has many stakeholders using the kernel for different things. It runs on devices ranging from clusters of supercomputers to tiny embedded devices. The kernel is also a moving target, with about 385 daily commits in 2023⁴. Due to this, maintaining the patch out-of-tree requires significant resources from the real-time team [44]. Conversely, a mainline kernel supporting real-time could imply more strain on other kernel developers, as they cannot break the real-time capabilities. Naturally, the design of these features, their maintainability, and how they affect or block other efforts are crucial for the mainlining venture [44].

Different companies have supported the `PREEMPT_RT` work throughout the years, depending on their respective need for real-time capabilities. At times, funding for the project has been hard to maintain. In 2015, the Linux Foundation started the Real-Time Linux Collaborative Project and secured funding for a limited team to dedicate themselves fully to the project [44].

From the inception of `PREEMPT_RT`, the goal has always been to merge it with the mainline kernel [44]. Many predictions have been made over the years, projecting this to transpire imminently. However, almost 20 years later, this has yet to come about. Still, most of the features have made it in, and today, the patch only contains about fifteen thousand lines⁵, most related to latencies in `printk` [67], [68].

This fact means that one still needs the patch to enable `PREEMPT_RT_FULL` as the preemption model at compile-time, but the day when `PREEMPT_RT` is no longer a patchset but rather a compile-time setting is close.

Recently, the major distributions have started to ship versions of their images with patched kernels [69], [70]. This development indicates a growing demand for `PREEMPT_RT` and a high community anticipation of the delayed merge to mainline.

3.5 Performance and Real-Time Capabilities

3.5.1 Latency

Much of the existing literature on the capabilities of `PREEMPT_RT` concerns itself with x86 platforms [5], [33], [36], [71]. The hardware plays a significant role in determining the achievable latencies of the system [1], [2], so the results are not directly transferable to embedded systems. Also, the underlying hardware on x86 systems is known

⁴Based on Listing D.1 in Appendix D.

⁵As of 6.6.25-rt29

to trigger System Manager Mode (SMM) sporadically. SMM bypasses the operating system to ensure proper CPU cooling and similar, effectively lowering the predictability of the system [2]. Some research does exist on ARM platforms, most commonly the Raspberry Pi [3], [72], [73], but also on the Beaglebone Black [3] and even soft-core CPUs [74]. The Altera Cyclone V, as is under question in this thesis, was used by Huang and Yang in 2020 with kernel version 4.4 [6]. They all report similar results, with the worst-case latencies hovering around one hundred microseconds, varying slightly with the techniques used. For mainline kernels, this is in the order of tens of milliseconds [13].

Typically, researchers use Cyclictst [50] to measure the theoretical latencies achievable on a platform while putting the system under some simulated load [3], [5]–[8], [72], [75], [76]. Cyclictst is a program that implements one or more periodic tasks. When a task wakes up, it compares the current time with the expected time and outputs the difference as the scheduling latency. Although Cyclictst is a helpful tool, one should not necessarily draw significant conclusions from its results. Its simplicity has perhaps been a crutch for the scientific community, as Cerqueira et al. remark regarding Cyclictst: *"it should not be mistaken to provide a general measure of a system's 'real-time capability'; it can only show the lack of such capability under certain circumstances"* [33].

Many factors also contribute to differing latencies in a system. Adam et al. found that while the average latency was roughly the same, the max latency differed significantly using different distributions with different kernel versions [3]. They do not discuss whether the user space environment supplied by the distribution or the kernel version was the main contributor to these differences. However, based on the contents of the Linux real-time mailing list, one might conclude that specific RT kernels differ in capabilities. Many report that particular, still maintained versions have high latencies in particular subsystems [77]–[81].

Much of the research also fails to account for the configuration of their systems, reducing the relevance of their results [1].

The Open Source Automation Development Lab (OSADL) has hosted a QA farm for PREEMPT_RT for several years. There, they perform long-term tests of the latency and stability before releasing stable versions. They publish a list of hardware and setups currently under test and their results. In contrast to much of the research community, they have a more systematic approach and list all relevant configuration parameters for the systems they test [82]. At the time of writing, they are executing a long-term test of version 5.15.72-rt48 on an ARM Cortex-A9, which shows a max latency of 121 μ s [83].

There also exist examples in the literature where the latency requirements for a project were achievable without PREEMPT_RT, solely utilizing SCHED_DEADLINE on a mainline kernel [84]. Although they experienced some latency issues they were not able to resolve, PREEMPT_RT was not able to mitigate them. They suspected the issue was caused by cache flushing, either by the kernel or hardware.

3.5.2 Determinism

Despite all efforts and advances to reduce latencies and increase the Linux kernel's determinism, the literature consistently agrees that PREEMPT_RT is unsuitable for hard

real-time systems [2], [73]. However, others claim that a properly tuned system can meet hard real-time constraints, but outside the context of functional safety-critical systems [1]. On the other hand, Brown defines Linux with PREEMPT_RT to be “95 % *hard real-time* [35].

Concerning soft real-time systems, PREEMPT_RT makes Linux more than capable [2], [3], [36]. Others claim this also applies to mainline Linux kernels, out-of-the-box [2]. These assessments warrant the question: If Linux cannot meet hard real-time requirements with the patch and is capable of soft real-time without the patch, what is the purpose of PREEMPT_RT? Naturally, the claims regarding Linux being soft-real-time capable out-of-the-box do not apply in every context. There are many valid use cases for PREEMPT_RT, and for the most part, it depends on the requirements of the system. Real-time system classifications do not imply a specific range of latency requirements but rather the severity of missing one or more deadlines. But this leads to the interesting question of when *real-time* is required or recommended, which the thesis revisits in section 3.6.1

The reason for deeming Linux with PREEMPT_RT unsuitable for hard real-time systems is the impracticality/impossibility of verifying that it will meet all deadlines due to the kernel’s complexity [1], [37] and inherent non-deterministic properties [34]. However, this has not kept researchers from trying, and efforts to verify the kernel’s hard real-time capabilities are ongoing [85].

Currently, the method by which developers verify their system’s real-time capabilities is through experiments [37]. These tests are often long-term and can, in certain circumstances, justify use in critical environments. Justifications for this include showing that the probability of missing a critical deadline is lower than that of a random hardware fault and adequately mitigating the consequences [1].

3.5.3 Throughput Degradation

A known and obvious side-effect of enabling full preemption is that the overall throughput of the system decreases [1], [2], [16]. This degradation comes from several aspects discussed in section 3.3, with the primary factor being the increased context switching and scheduling overhead.

Literature that describes how severe degradation one can expect is limited, and the results vary significantly, indicating that it depends on the workload. In 2007, Jeong et al. found that PREEMPT_RT reduced the throughput by a factor of five using Hackbench [86]. The same year, McKenney found a time-consuming build workload to be 16.78 % faster on a mainline kernel than on a preemptible kernel [16]. Li et al. found a reduction of only 6 % in 2023 [76] but saw that the degradation increased with the number of CPU cores, leading to a 29 % reduction in throughput on a quad-core system [76].

3.6 Tuning and Best Practices

As mentioned above, the theoretical latencies produced by Cyclictst and similar tools can only tell us so much. Although PREEMPT_RT makes it easy to develop real-time applications for Linux with the same tooling as any other application, it is not a “*silver*

bullet" [36]. It is not just a matter of enabling PREEMPT_RT_FULL. Instead, a real-time system requires careful design, as with any other operating system. However, the literature contains several best practices and guidelines.

3.6.1 Workload Analysis

Paul McKenney distinguishes between *real-time* and *real-fast* systems. He describes *real-time* as "getting started as quickly as possible" and *real-fast* as "getting done quickly once started" [16]. Although this distinction seems trivial, it makes sense in the context of a patched Linux kernel with full preemption enabled. A preemptible kernel will lower the system's overall throughput. If the application does not have clear-cut critical real-time constraints, a workload analysis is essential to determine whether *real-time* will benefit a system.

According to [16], the duration of the work is the most critical factor when determining this. The positive effects of the real-time system's ability to start the work sooner diminish when the duration of the work extends past 10-20 milliseconds. Tasks with many system calls require more context switching, dramatically increasing the "*real-time average-overhead penalty*." In these cases, the workload duration limit before the positive effects of *real-time* diminish might be as low as a few milliseconds [16].

3.6.2 Kernel Compile-Time Settings

Full preemption should be enabled at compile time if the workload analysis deems *real-time* to benefit the system. Suppose the analysis shows that a *real-fast* approach is more suitable than a real-time approach; experimenting with other preemption models like *Low-Latency Desktop* or *Voluntary Preemption* might provide a good compromise. HR timers should be enabled and verified [1]; however, many platforms enable this by default. Full tickless operation can be enabled at compile-time if CPU isolation is relevant [65], and RCU settings should be tweaked if the system becomes unstable or if the latencies are still too high [63].

3.6.3 Policies, Priorities, and Throttling

Using real-time scheduling policies means that the scheduler respects the assigned priority of tasks. This submission by the scheduler to the user's choices means application developers should carefully consider the time they leave for non-critical tasks when designing systems. If a high-priority real-time task fails to forfeit the CPU for sufficient time, other tasks might never get to run. The system will, by default, impose real-time throttling, meaning that real-time scheduled tasks will not take up more than X amount of time per period P [87]. Throttling is a backup mechanism to ensure the system does not freeze entirely if a real-time task goes rogue. A real-time system design should not depend on this feature. The default setting for real-time throttling leaves five milliseconds out of every second for non-real-time tasks. This means that this mechanism can, at any time, give a latency of up to five milliseconds for all real-time scheduled tasks. The developer can modify the period and throttling limit or deactivate it. This is a kernel setting configurable at run-time through `sched_rt_period_us` and `sched_rt_runtime_us`, located in `/proc/sys/kernel`. Turning off real-time throttling requires extra care from developers to ensure that tasks with

real-time policies have a bounded and short duration. For CPU-intensive real-time scheduled tasks, the run-time option `RT_RUNTIME_GREED` can be enabled [88]. This means that throttling will not occur unless there are non-real-time tasks experiencing starvation.

For real-time scheduling policies, the static priorities range from 1 to 99. However, some critical kernel tasks have a priority of 99, and these should ideally still be allowed to preempt our real-time tasks. This means one should only use priority 99 except in exceptional circumstances after careful consideration [2], [89]. The kernel's default priority for threaded interrupts is 50 [2], [89]. Developers should consider this when setting task priorities and change them if required.

The Linux real-time community favors `SCHED_FIFO` over `SCHED_RR`. When two tasks with the same priority, scheduled with `SCHED_RR`, are runnable, the scheduler imposes time-slicing between them, leading to a less deterministic system. Although this might be fine (or even desired) for some use cases, one should be wary about the implications.

Neither the C++ standard library nor Boost supports setting scheduling policies and priorities through their thread APIs. However, they offer an API to retrieve the thread's native handle, making it possible to modify the POSIX thread attributes directly through the POSIX API after initialization. It is also possible to set the scheduling policy and priority of a process (and all threads it might spawn) during invocation through the `chrt` utility found in the *util-linux* package.

Depending on the system configuration, changing scheduling policies might require root access. One way to work around this is to use the kernel's *capabilities* interface to allow this for specific executables without requiring them to run as root [90].

3.6.4 Measuring Time and Sleeping

Measuring time with fine-grained granularity is critical for all real-time systems. After ensuring that the *hrtimers* subsystem is enabled and working, all system calls to get the current time should be to `clock_gettime()`, with the flag `CLOCK_MONOTONIC` [2], [91]. This flag ensures that the time read back is from a strictly increasing monotonic counter instead of the system clock. This function returns a `struct timespec`, as seen in Listing 3.1.

```
1 struct timespec
2 {
3     time_t tv_sec;
4     long int tv_nsec;
5 };
6
```

LISTING 3.1: Struct `timespec` definition [92].

Despite the limits of its datatype, the valid range for `tv_nsec` is $[0, 10^9)$ [92]. All system calls returning a `struct timespec` will abide by this and expect the same

when passing them a `timespec` parameter. Therefore, the user must normalize these values after arithmetic operations.

Tasks should use `clock_nanosleep()` to achieve the lowest possible scheduling latencies when voluntarily giving up the CPU [2]. The PREEMPT_RT developers recommend passing the absolute time as a wake-up parameter specified by the flag `TIMER_ABSTIME` [91]. This ensures that no overhead affects the wake-up time but requires the user to calculate this in advance.

Common C++ libraries, like the standard library and Boost, provide sleep mechanisms taking absolute or relative values. GCC's standard library implementation supports using `nanosleep` for this under the hood [93, `libstdc++-v3/include/bits/this_thread_sleep.h`, Line 80], as does *Boost* [94, `libs/thread/src/posix/thread.cpp`, Line 457] if configured correctly. However, one should verify that this is the case if in doubt.

Regarding kernel space, the most beneficial way to sleep depends on the situation. According to [95], when wanting to sleep for less than 20 ms in non-atomic situations, one should use `usleep_range`. It is based on high-resolution timers and provides a low-latency wake-up time. However, the wider the range provided, the more the kernel can optimize the wake-up time and the less of a performance hit the system will likely take.

3.6.5 Managing Memory

Swapping, a technique that writes the memory area of a process to disk to allow for more processes to execute concurrently, can negatively impact latencies [2]. Mitigating this can be done by locking the memory of a process or thread and prohibiting the operating system from swapping it. One way to do this is by calling `mlockall` during the initialization of the entire process or a real-time critical thread with the flags `MCL_CURRENT` and `MCL_FUTURE`. Even if swap memory is disabled, this call has the positive side-effect of pre-faulting current and subsequently initialized thread stacks [96].

When an application requests the kernel to create a thread, it is assigned a stack. A continuous virtual address space represents a thread's stack, but the kernel does not allocate the entire area to physical memory upon its creation. Instead, Linux will allocate a new page, typically 4 KiB, when needed. This occurrence is called a *page fault* and is a significant source of non-determinism for a task. Locking the memory of a process ensures that all of its threads' respective stacks will be deliberately pre-faulted immediately.

The literature conflicts on whether manual pre-faulting is required when locking memory. According to the official Linux Real-Time documentation, a call to `mlockall` will automatically pre-fault all memory [96]; however, according to Duval (2009), it is still recommended to pre-fault the stack memory [97]. [98] recommends manually pre-faulting heap memory as well.

The heap of a process is also subject to the same run-time page faulting latencies, meaning it should also be locked and potentially pre-faulted deliberately [96]. Pre-faulting can be done by dynamically allocating an area of memory and writing to at least one element per page, as seen in Listing 3.2. The process' expected dynamic memory usage determines the size of the area to touch during initialization.

```
1 uint8_t* buffer = malloc(EXP_HEAP_USAGE);
2
3 for( size_t i = 0; i < EXP_HEAP_USAGE; i += sysconf(_SC_PAGESIZE))
4 {
5     buffer[i] = i;
6 }
7
8 free(buffer);
9
```

LISTING 3.2: Example of manually pre-faulting heap memory.

Madden [2] underlines the risks involved when running with *overcommit_memory* enabled, which defaults to 150%. This means that the kernel assumes that all applications will not use all the memory requested simultaneously and that it is safe to allocate more memory than is available. This can lead to a scenario where the kernel kills a process to free up memory. In some systems, this might be acceptable; however, in others, it could be detrimental. It is important to note that by default, the kernel will not necessarily kill the process that caused the situation and could potentially kill a critical task [2].

According to *sysctl* documentation [99], it is possible to set *oom_kill_allocating_task* to ensure that the kernel will always kill the process that triggered the situation when running out of memory. Most configurations disable this feature by default, but enabling it can be good practice in certain situations. It is also possible to set *panic_on_oom* if it is desirable to crash the system when running out of memory [45].

3.6.6 Broken Modules and Subsystems

The Linux kernel consists of several subsystems and many drivers, and not all are well-behaved from a real-time perspective [2]. A general rule in real-time computing is to avoid dynamic memory allocation at run-time [96], [100]. This is also true when using Linux, as memory allocation operations might have unbounded latencies [2]. Some system calls might allocate memory dynamically, meaning they should be avoided if possible or reworked if required. The network stack is one such example. Because it dynamically resizes its buffers, the network stack can potentially reduce the system's determinism [9]. Operations like writing to disk or using the TTY subsystem layer are best avoided [67]. Developers should also inspect and verify that device drivers do not perform dynamic memory allocation before using them [2], [9].

3.6.7 Loadable Kernel Modules

For the same reason that specific existing kernel modules might lead to unacceptable levels of non-determinism in a system, so can any custom LKMs designed as part of that system. Therefore, it is vital also to consider the guidelines and best practices for these components [2]. As mentioned, dynamic memory allocation should be avoided at run-time. Disabling preemption is a big red flag and should only happen if the scheduler interacts with the module [101]. Normal spinlocks are preemptible under

PREEMPT_RT, meaning they are safe to use, but raw spinlocks are discouraged unless the module interacts with the scheduler or does interrupt-dispatching [101]. Although these mechanisms have valid use cases, they are exceptions and should rarely apply to a kernel module that supports an embedded real-time application.

3.7 Summary

Although it has been the facto standard for Real-Time Linux for several years and has many real-world use cases by prominent actors, PREEMPT_RT remains a niche. With the upcoming expected merge to mainline, new companies are curious to see what potential benefits they can reap from the capabilities it brings. Substantial research exists on the theoretical latencies achievable with Linux, but different hardware, kernel versions, and system requirements mean that existing results are not directly applicable. Cyclictst can only tell us so much, and benchmarks should preferably measure an actual system or something very similar to it, which is the goal of this thesis. The literature cannot predict how the target system might perform with PREEMPT_RT. It must be tested through experiments and evaluated carefully. The research on PREEMPT_RT capabilities on this project's hardware platform is also limited. Although some literature provides benchmark results on the reduced throughput caused by PREEMPT_RT, these measurements are pretty old, and in the end, the nature of the system determines the results.

This chapter has identified several techniques and best practices from the literature, but it is unknowable which, if any, will benefit the target system without putting them to use. Whether a *real-time* or *real-fast* system best serves the target application is a crucial question in this thesis.

As for the relevance of further research, Reghenzani et al., which performed a comprehensive survey on PREEMPT_RT in 2019, stated that: *"For academic and research purposes, PREEMPT_RT is a potential candidate for the development of both applications and test benches. In the first case, any type of application can be implemented and tested in a real-time environment with less effort rather than using complex RTOSs. In testbench cases, the experimenters can use Linux to test the performance of scheduling algorithms, IPC calls, and any other operating system mechanisms. It is important that researchers correctly configure the Linux system in order to have realistic latencies. This activity has been sometimes neglected and may lead to erroneous or unreliable conclusions."* [1].

Chapter 4

Methodology and Design

This chapter gives an overview of the research methods applied to the project. It goes into the concrete work required to build a platform to perform measurements on and the system under test's architecture. Finally, it describes the experiments designed for the thesis and explains how it evaluates the results.

This thesis is in the applied research domain and uses a combination of quantitative and design research methodologies. It does not utilize the company's intellectual property, ensuring all results are open and reproducible. All required SW and HW components are either based on open-source solutions or developed from scratch.

The project's first phase will be designing and implementing the SUT. This includes:

- FPGA Hardware design for simulating RX and TX and simulated application logic.
- Build and configure bootloader for the target platform.
- Build and configure mainline Linux kernel for the target platform.
- Build and configure Preempt RT patched Linux kernel for the target platform.
- Create a root file system with required libraries and utilities.
- Set up an environment where it is possible to modify HW, kernel version, and SW components and quickly deploy to the target, resulting in a short feedback loop and maximized efficiency.
- Design and implement platform baseline tests to understand the system's capabilities and limitations. This will, in some cases, allow us to verify the findings in the literature and, in other cases, clarify contradictory reports.
- Design and implement applications needed for achieving the desired data flow. This includes the interface between the HPS and the FPGA and between user and kernel space.
- Design and implement tests to perform on the mock implementation of the target system.

The second part will analyze and optimize the system's real-time capabilities and throughput. Relevant comparisons and activities include, but are not necessarily limited to:

- Preemption Models

- Different kernel configurations according to best practices found in the literature
- Scheduling policies and priorities
- Optimizing RT tasks
- Optimizing IPC mechanisms
- Applying CPU load with various stress tools
- Developing custom test suites where applicable

Using CPU isolation was considered but abandoned due to the limited number of cores on the target platform.

4.1 Platforms and Tools

The target system will run on custom hardware, integrating an Intel Altera Cyclone V. This thesis performs all tests on a commercially available development kit to avoid tying academic efforts to the company's intellectual property. The vendor's official kit, DK-DEV-5CSCX6N was chosen [102], with the SoC variant 5CSXFC6D6F31C6.

The project will only depend on hardware contained internally in the SoC, but the SoC requires external RAM to function, so this will affect the results. The development kit includes a 1 GB DDR3 SDRAM with a 32-bit bus clocked at 400 MHz. This results in a theoretically achievable bandwidth of 25.6 Gbps, according to the reference manual [102].

4.1.1 Hardware

The vendor's standard tool for FPGA development, Quartus Lite 22.1, is used to develop the necessary hardware logic for the project. The lite version of Quartus requires no license. It facilitates the configuration of both the HPS and FPGA and includes various scripts for generating artifacts required by the BSP. Specifically, this includes a raw binary file (.rbf) to configure the FPGA, a compiled device tree blob (.dtb) describing the hardware to the Linux kernel, and header files required by the bootloader.

4.1.2 Board Support Package

This section briefly describes and justifies the tools and versions used to build the minimal Linux system used in this thesis.

Toolchain

Crosstool-NG, a tool for generating toolchains for various platforms, generates the project's toolchain. Concretely, it is version 1.26.0 [103], which supports GCC 13.2. The content of the build configuration is available in Appendix A, Listing A.1.

U-Boot

Das U-Boot version 2023.07 [104] makes up the preloader and bootloader of the BSP. The build uses the default configuration for the Cyclone V, `socfpga_cyclone5_defconfig`, which comes bundled with U-Boot. Through Quartus, the Cyclone V is configurable in numerous ways. For U-Boot to configure the Cyclone V correctly during initialization, it depends on the contents of several board-specific header files. These are generated by Quartus Lite. The command line arguments passed to the kernel at boot-time contain no significant parameters concerning the outcomes of this thesis.

Linux Kernel

For the mainline Linux kernel, version 6.6.14 from the master tree is used [27], and the PREEMPT_RT patched version uses version 6.6.14-rt21 from the Linux stable RT tree [62]. The reason behind selecting these versions is that they are the current LTS releases. Although the kernel used in production environments at the company derives from version 5.19, a newer kernel makes the results more relevant for others. The fact that these versions differ in their major number should not be significant. Torvalds himself has stated that he increments the major version "*when the minor version gets to around 20*" [105], not because anything fundamental has changed. Still, the company's kernel and Linux distribution are, in specific cases, used to verify that no significant discrepancies exist between the versions and their configurations.

All build configurations stem from the vendor-provided `socfpga_defconfig`, bundled with Linux. They are, however, slightly modified. `CONFIG_CMA` and `CONFIG_DMA_CMA` are enabled to provide the functionality required by the CMA driver discussed in section 2.3.2. The preemptible version of the mainline kernel enables `CONFIG_PREEMPT`, and the PREEMPT_RT patched tree enables `CONFIG_PREEMPT_RT`. `CONFIG_HIGH_RES_TIMERS` are enabled on all ARM platforms by default. The configuration file for the non-preemptible kernel is available in Listing A.2 in Appendix A. Differences between this config file and the ones used for the preemptible and fully preemptible kernels can be seen in Listing A.3 and A.4 in Appendix A.

The kernels in this thesis build the upstream drivers *in-tree* while building other kernel modules *out-of-tree*, leading to a tainted kernel at run-time.

Root Filesystem

Buildroot was chosen over Yocto to build the root filesystem. This is due to Buildroot's simplicity, as this thesis only requires a minimal setup. Although Buildroot can compile the toolchain, bootloader, and kernel, they were built separately for this project to maintain control and have a more flexible setup.

Version 2023.08.1 [106] was selected, from which the included `socrates_cyclone5_defconfig` made up the basis for the configuration. The final build configuration contains nothing relevant to this thesis's main topics and results and has been omitted from this document.

The versions of the tools provided by this particular version of Buildroot are summarized in Table 4.1.

TABLE 4.1: Versions of relevant software provided by Buildroot.

	Version
rt-tests	2.5
Stress-NG	0.15.07
Iperf3	3.14
htop	3.2.2
GNU Coreutils	9.3
p7zip	17.04
sysstat	12.6.1
libnl	3.7.0

Running the BSP

Normally, the Cyclone V boots from either flash memory or a micro SD card. To avoid flashing an SD card on each modification to the kernel or FPGA image during development, TFTP was used to have U-Boot download these artifacts from a TFTP server running on the host machine at boot time. The root filesystem was also placed on the host machine and loaded into RAM on the Cyclone V as a Network File System (NFS). This setup dramatically shortened the feedback loop during development but could lead to higher latencies. Therefore, a complete BSP stored on an SD card made up the platform upon which all benchmarks were executed.

The versions of the tools used to build the BSP artifacts are summarized in Table 4.2.

TABLE 4.2: Versions of tools for building BSP artifacts.

	Version
Quartus Lite	22.1
U-boot	2023.07
Buildroot	2023.08.1
Crosstool-ng	1.26.0
Linux	6.6.14
Linux-stable-rt	6.6.14-rt21

4.2 Test Setups

The following sections describe the experiments designed to evaluate the impact of kernel configurations, scheduling policies, and system architecture decisions on the target system's real-time capabilities and throughput. This section describes and justifies the overall setups and system configurations used in the experiments. If a particular experiment deviates from the configurations described here, the respective experiment's description explicitly states it.

4.2.1 Comparative Setup Configurations

Based on the findings in the literature, the target system might not require PREEMPT_RT to meet its requirements. Instead, it might even be considered harmful due to the

reduced throughput one should expect. Because of this, all tests were executed on the following preemption models:

Kernel Preemption Model

- Linux
 - No Forced Preemption (Server)
 - Preemptible Kernel (Low Latency Desktop)
- Linux Stable RT
 - Fully Preemptible Kernel (Real-Time)

The literature review did not uncover any comparisons between a preemptible and a fully preemptible kernel. This makes the comparison all the more interesting and the results more novel.

The following variations in scheduling policies and priorities are deemed attractive, but they are not necessarily all used in every experiment.

Scheduling Policies

- SCHED_OTHER - NICE level: 0
- SCHED_OTHER - NICE level: -20
- SCHED_FIFO - Priority: 40
- SCHED_FIFO - Priority: 60
- SCHED_FIFO - Priority: 99

SCHED_OTHER is used to determine a baseline, and using SCHED_FIFO with different priorities uncovers to what degree threaded interrupts influence latencies.

4.2.2 Stressors

The literature repeatedly finds that the impact of the preemption model becomes more pronounced when the system is under load. Because of this, the experiments are performed using different stressors to induce load on the system. This also makes for more realistic measurements.

The following stress scenarios are used in the experiments:

- Stress-NG - CPU load of 25 % and 75 %
- Iperf3 - Network load
- Hackbench - IPC load

The Stress-NG configurations induce 25 % and 75 % CPU load, respectively. Furthermore, they execute almost exclusively in user space. The loads induced by Iperf and Hackbench consist of many system calls, meaning that they primarily execute in kernel mode.

The exact configurations of the stressors and their resulting CPU load are available in Listings B.1, B.2, B.3, and B.4 in Appendix B. The stressors are scheduled under

SCHED_OTHER, meaning that they are subject to time-slicing. As the Iperf3 stressor executes a TCP benchmark that tries to run as fast as possible, the CPU load caused by Iperf will drop when executing alongside the respective benchmarking software of each test.

4.3 Platform Baseline Tests

This section describes and justifies several experiments and their design. They aim to evaluate the platform's real-time capabilities and the implications of applying different preemption models and scheduling policies.

4.3.1 High-Resolution Timers Verification

As the literature shows, high-resolution timers should be verified to be enabled. Moreover, the hardware should be verified to support their resolution. As recommended, this is done by inspecting the outputs of `/proc/timerlist`. In addition to this verification by analysis, they are also verified in practice using `Cyclictest` in the succeeding experiment.

4.3.2 Cyclictest

`Cyclictest` supports a wide range of options, and the configuration used for this experiment can be seen in Listing 4.1. The test uses the same periods as [3] to achieve comparable results.

```
1 #SCHED_FIFO - Priority 60
2 cyclictest --loops 1000000 \ # Number of iterations
3     --threads 1 \ # Number of threads
4     --interval 400 \ # Period of thread in us
5     --mlockall \ # Lock memory
6     --smp \ # Use all CPUs
7     --policy fifo \ # Use SCHED_FIFO
8     --priority 60 \ # Priority of 60
9     --default-system \ # Don't tune system
10    --quiet \ # Don't print to stdout
11
```

LISTING 4.1: `Cyclictest` baseline test configurations.

Three different tests are performed. The first test will run with SCHED_OTHER, the second with SCHED_FIFO and a priority of 40, and the third with SCHED_FIFO and a priority of 60. The reason for executing at 40 and 60 is that the default priority of threaded interrupts is 50, as described in earlier chapters. This facilitates a view of threaded interrupts' effects on the system's latency. In contrast to [3], the tests are not performed with a priority of 99, as the literature does not recommend using such a high priority. Additionally, the tests execute for twice the number of iterations to get more robust results. However, the number of iterations tested is still low compared

to what the literature recommends, but a compromise was necessary due to time constraints.

Further, this configuration means that Cyclictest will create one thread per CPU. The first thread will run with a periodic interval of 400 μ s, and since the default interval between threads is 500 μ s, the second thread will have a period of 900 μ s. The test will run until the most frequent task has reached 1 000 000 iterations, approximately 6.5 minutes. The memory is locked to avoid page faulting latencies, and Cyclictest is configured not to tune the system internally, ensuring that the test will most accurately reflect the system's capabilities.

4.3.3 Throughput Test

The literature shows that one must expect the system's throughput to degrade when using `PREEMPT_RT`. However, reports on the extent of this degradation are rare in the literature, and the literature review did not uncover any reports on the throughput reduction of a low-latency preemptible kernel. Due to these findings, this test measures and compares the system's throughput under different preemption models.

The test consists of running two different stressors, which will perform a finite set of operations as fast as possible. The total run time of the test will determine the extent of throughput degradation.

The tests utilize Stress-NG and Hackbench, and Listing 4.2 shows their respective configurations.

```
1 # Stress-NG workload
2 Stress-NG --vm 1 \
3     --vm-locked \
4     --vm-populate \
5     --vm-madvise nohugepage \
6     --vm-method gray \
7     --vm-bytes 128M \
8     --verify \
9     --metrics-brief \
10    --vm-ops 5000000
11
12 # Hackbench workload
13 hackbench --loops=50000
```

LISTING 4.2: Throughput test workload parameters.

The README of Stress-NG states that although it is possible to use it as a throughput benchmark, this is inadvisable as it was not designed for this purpose [107]. However, in an issue on the Stress-NG GitHub page [108], the author states that as long as the version of Stress-NG is the same across the benchmarks and that the several test executions give corroborative results, it should be considered sound. As explained in section 4.2.2, Stress-NG executes almost entirely in user mode, while Hackbench executes almost entirely in kernel mode, giving us coverage of both cases.

4.3.4 Preemption Test

The preemption test aims to determine the impact scheduling policies and preemption models have on the preemption count of a task. The test consists of a task that mimics a CPU-intensive task by busy waiting for 500 ms and then sleeping for 500 μ s, as seen in Listing 4.3.

```
1 while( true )
2 {
3     //Busy wait for 500 ms
4     struct timespec start = getTimespecNow();
5     while(timespecDiffUs(getTimespecNow(), start) < 500 * MICROS_PER_MILLI);
6
7     //Sleep for 500 us
8     usleep(500 * MICROS_PER_MILLI);
9 }
```

LISTING 4.3: CPU intensive task, excerpt from preemption test.

Each configuration of the test executes for 600 cycles, approximately 10 minutes. The system call *getrusage* allows a separate thread to query how many times the task has voluntarily given up the CPU and how many times another task has preempted it.

The expected outcome is that the scheduling policies will have similar preemption counts in idle CPU conditions but differ more when the system is under load. The preemption model should not impact the results regardless of the load, as it only dictates whether or not a task executing in kernel space is preemptible.

An additional test is also conducted to determine how often a user space task gains an advantage due to the kernel being preemptible. Cyclicttest will be executed with the same parameters as in section 4.3.2, using Iperf as a stressor. During this test, external software will record the number of times Cyclicttest preempts a task executing in kernel mode. Further details on this test are presented along with the results.

4.3.5 Memory Lock Test

The significance and implications of memory locking and prefaulting are somewhat contradictory reported in the literature. This test will investigate what operations are required to avoid page faults during run-time and measure their impact when memory is not locked and pre-faulted. The test will consist of SW that will allocate 64 KiB of memory, equivalent to 16 pages, and then touch it in a loop, measuring the worst-case latency experienced during a page fault. The test will be invoked 5000 times, resulting in a total sample size of 80 000 page faults.

The test is performed with static memory, memory allocated on the stack, and memory allocated on the heap to see if there are any differences among these.

4.4 Target System Analysis

The platform baseline tests provide an understanding of the system's capabilities. However, as the literature suggests, this is most likely insufficient to determine whether the target system's requirements are feasible on a Linux platform.

This section analyzes the target system based on findings in the literature and further refines its real-time constraints.

4.4.1 Real Time vs. Real Fast

The target system has, up till this thesis, been treated and designed as a *real-fast* system (in the context of McKenney's philosophy [16] described in section 3.6.1).

Parts of the application rarely perform system calls. As the IO interface towards the FPGA logic is strictly in user space, the only system call regularly used is the Generic Netlink interface towards the high-level side of the system. This might make the system suitable for a *real-time approach* [16].

According to [2], the workload of the target system can be considered asynchronous, as its start and completion time does not have hard constraints but rather a soft deadline spanning several cycles of execution. The target system might perform satisfactorily without real-time scheduling. However, Madden states several potential advantages it might gain from `PREEMPT_RT` and `SCHED_FIFO` [2]. Among these is the consistency of the task execution time and the fact that the time complexity of real-time scheduling policies is lower than that of CFS. In addition, real-time policies favor CPU-intensive tasks as found in the target system.

4.4.2 Periodic vs Aperiodic

As discussed, the existing target system is not periodic. Its main loop has a flag *RunMore*, which, if set, makes the loop execute again. Depending on the system's current state, various components throughout the application can set this flag. For instance, when reading incoming data from the circular buffer, if a chunk of 4 KiB was available, *RunMore* is set to one. This design choice leads to an inherent non-deterministic system. Depending on the execution time of the decoding logic, this can lead to at least two different scenarios:

- **Scenario 1:**

- Mainloop executes:
 - * 4 KiB is read from the buffer.
 - * RunMore is set to at least one.
 - * Packet is decoded.
- Mainloop executes again because of RunMore
 - * The buffer is found to have more than 4 KiB of data
 - * 4 KiB is read from the buffer.
 - * RunMore is set to at least one.
 - * Packet is decoded.
- Mainloop executes again because of RunMore
 - * The buffer is found to have more than 4 KiB of data
 - * ...
 - * ...

- **Scenario 2:**

- Mainloop executes:
 - * 4 KiB is read from the buffer.
 - * RunMore is set to at least one.
 - * Packet is decoded.
- Mainloop executes again because of RunMore
 - * The buffer is found to have less than 4 KiB of data
 - * The thread goes to sleep for a millisecond
- Mainloop wakes up after a millisecond
 - * The buffer is found to have more than 4 KiB of data
 - * ...
 - * ...

In scenario 1, the execution time of the main loop is longer than the time it takes the FPGA to write 4 KiB to the shared buffer, i.e., 1.62 ms. This means the main thread might never voluntarily yield the CPU while reading data. This is fine when Using a time-shared scheduling policy like SCHED_OTHER, but when using one of the real-time policies, all other tasks might be subject to starvation.

In the second scenario, the execution time of the main loop is shorter than 1.62 ms, for instance, 500 μ s. Again, under SCHED_OTHER, it might make sense not to actively sleep since the task might have already been swapped in and out on the CPU, meaning the buffer contains data again. With SCHED_FIFO, however, one can be sure the main

loop has not been preempted (at least not for long) and that running again is simply a waste of precious CPU time.

Although the extra overhead in scenario two is very slight, both scenarios imply that a periodic approach could benefit the system. Naturally, if the execution time were consistently above 1.62 milliseconds, the system would not function. However, it can still occur intermittently, for example, if the task is subject to heavy time-slicing under CFS or experiences throttling under a real-time policy.

4.4.3 Memory Management

For the most part, the target system uses statically allocated memory pools to avoid dynamic memory allocation, where the largest is about 9 MiB. However, lazy allocation might still lead to page faulting during run-time, introducing latencies into the system. The platform baseline test described in section 4.3.5 covers the implications of this.

However, the target application does perform some dynamic memory allocation when interfacing a kernel module through Generic Netlink. The memory lock test does not cover the impact of this, so a separate experiment is performed to investigate this.

4.4.4 Single-Threaded vs Multi-Threaded

The existing design of the target system is single-threaded for all practical purposes. Partitioning parts of the logic into separate threads could help appropriately separate real-time and real-fast workloads. However, this increases the complexity of the system. Since this is a porting effort, not a new venture, it might introduce significant risks.

The two outer points of the application, the FPGA interface and the part communicating with kernel modules using Generic Netlink, are the best candidates for extracting to separate threads. However, this might create bottlenecks in the system, where packets would pile up in queues, not solving anything.

The target system-specific experiments will investigate how the interface towards the kernel modules impacts the latencies and how the preemption model impacts the throughput of these mechanisms.

4.4.5 Requirements

As discussed in Chapter 2, the major real-time constraint placed on the target system is to keep up with the data rate required by the FPGA logic. As found in section 2.3.3, the target system must be able to read the entire buffer within 50 ms to avoid buffer overflow.

Due to DMA synchronization, the target system should always leave a safety margin in the buffer when the FPGA logic is actively writing. Taking this into account, as well as other uncertainties, one can assume that the target system should be able to read an entire buffer within 40 ms to be sure to avoid overflows.

Aperiodic

As discussed earlier, the design choices of the existing system resemble a *real-fast* approach. It is not periodic; instead, it continues to execute as long as there is work to do. This includes whether or not data is available from the circular buffer. When there is nothing to do, the application sleeps for one millisecond.

Assuming that the task is not preempted significantly during execution, one can assume that the thread can withstand up to 40 ms wake-up latency when going to sleep. However, this assumes that the thread gets sufficient CPU time in the short term after waking up to clear the buffer. If the task's scheduling policy is `SCHED_OTHER` and is subject to time-slicing, this is not guaranteed and depends on the system's background load. If the task uses a real-time scheduling policy, the task will only be preempted by more critical tasks. Still, if the execution time to clear the buffer grows too large, the task might be subject to real-time throttling.

This shows that determining the temporal constraints of the target system is not trivial and that the system might benefit from a *real-time* approach to increase the predictability of the system.

Periodic

If one instead assumes a periodic task, reading packets of 4 KiB from the circular buffer every millisecond, it might be possible to formulate more precise temporal constraints for the system.

Given these parameters, as well as a safety margin of 4 KiB, the software can read an entire buffer in

$$\frac{\mathit{bufferSize} - \mathit{safetyMargin}}{\mathit{readChunkSize}} \tag{4.1}$$

$$\frac{128 \text{ KiB} - 4 \text{ KiB}}{4 \text{ KiB}} = 31 \text{ cycles}$$

In other words, a period of 1 ms can be sufficient to keep up with the FPGA.

Assuming a perfect average scheduling latency of 0, one can perhaps accept a latency spike of up to 40-50 ms per 31 cycles.

Assuming no large spikes, one can calculate the maximum average latency required to avoid overruns. This is given by the fact that there is a 50 ms – 31 ms = 19 ms margin for latencies. This leads to a maximum average latency of

$$\frac{19 \text{ ms}}{31 \text{ cycles}} = 613 \mu\text{s} \tag{4.2}$$

A more realistic average scheduling latency of 50 μs would allow for an additional latency spike of up to

$$19 \text{ ms} - ((31 - 1) \cdot 50 \text{ } \mu\text{s}) = 17.5 \text{ ms} \quad (4.3)$$

Transmitting

The constraints for writing data to the circular buffer can be assumed to be the same if the buffer is always full when the DMA transaction starts. However, this might not always be a valid assumption. Suppose the DMA controller starts reading with only 8 KiB in the buffer. Assuming everything else is the same, the following two cycles will have an accumulated max latency of 8 ms. This gives a max latency of 7.95 ms, when accounting for an average latency of 50 μs

The target application will also apply modulation and demodulation algorithms on the data that passes through it. Determining the execution time of these algorithms in advance is not trivial and will include many assumptions.

4.4.6 Design of System Under Test

As mentioned, porting the target system to the new platform is a planned effort, and it is impossible to benchmark the actual system at this point. Due to this, critical components required to mimic the data flow of the target system will be implemented and benchmarked in isolation.

The actual system has an algorithm for swapping between transmission and reception and transmitting dummy packets if there is nothing to transmit. To avoid this logic, the SUT will consist of two modes: receiving or transmitting packets continuously.

The primary output of this thesis is the results of the experiments performed on the SUT, not the SUT itself. Therefore, the descriptions of its implementation are kept at a high level.

Reception Pipeline

The high-level design of the reception pipeline can be seen in Figure 4.1.

This pipeline represents the path the data will take through the SUT. Each component has an input and an output queue, making it easy to separate components into separate threads if desired. Although this might introduce overhead that is not present in the actual system, the target system has significant overhead through its business logic, such as the signal processing algorithms, which are not present in the SUT.

The *RX Driver* initializes and is responsible for enabling and disabling the *DataMover* component on the FPGA. Communication is done through the HPS-to-FPGA bridge, while the *DataMover* writes to the shared buffer through the FPGA-to-HPS bridge.

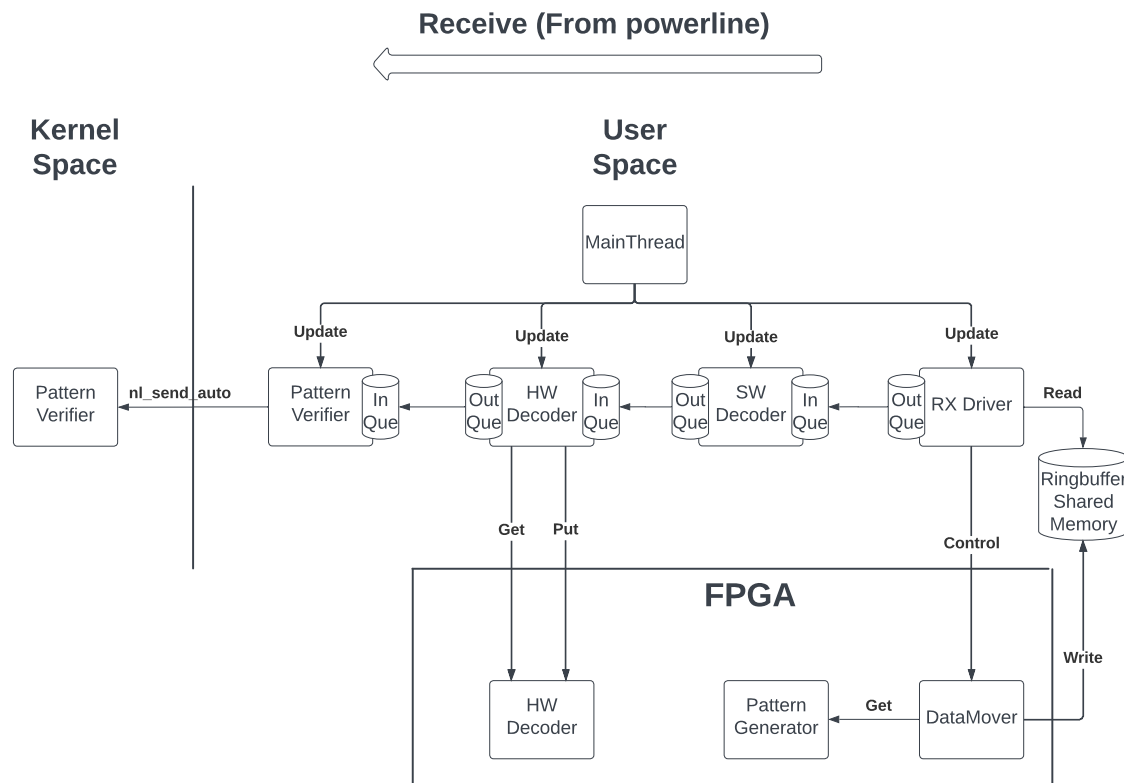


FIGURE 4.1: Design of reception pipeline.

Depending on its configuration, the main thread will implement a periodic or aperiodic routine and drive data through the pipeline. The main thread does this through *update* calls to the pipeline's respective components and by reading and writing to the packet queues on the input and output of each component.

The *SW Decoder* component will decimate the data by a factor of two, and the *HW Decoder* will decimate the data by a factor of four by passing it through a component in the FPGA.

This means that the pipeline has a total decimation of eight and that for every eight bytes read from the shared buffer, only one is written to the *Pattern Verifier*. The pattern verifier component takes the packages received in its input queue and writes them to a kernel module over Generic Netlink sockets. This kernel module is responsible for verifying that the data has not been corrupted or overflowed during transmission through the pipeline.

A slightly simplified implementation of the RX pipelines' main routine is available in Listing 4.4. This includes an example of the data passing between the components, showing the step-by-step decimation of the pattern generated by the FPGA.

```
1 void RxPipeline :: update ()
2 {
3     rxDriver . update () ;
4
5     // RX Driver Output:
6     // 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
7     // 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
8     // 0x03, ...
9     while ( rxDriver . packetAvailable () )
10    {
11        const auto packet = rxDriver . get () ;
12        swDecoder . put ( packet ) ;
13    }
14
15    swDecoder . update () ;
16
17    // SW Decoder Output:
18    // 0x01, 0x01, 0x01, 0x01, 0x02, 0x02, 0x02, 0x02
19    // 0x03, ...
20    while ( swDecoder . packetAvailable () )
21    {
22        const auto packet = swDecoder . get () ;
23        hwDecoder . put ( packet ) ;
24    }
25
26    hwDecoder . update () ;
27
28    // HW Decoder Output:
29    // 0x01, 0x02, 0x03, ...
30    while ( hwDecoder . packetAvailable () )
31    {
32        const auto packet = hwDecoder . get () ;
33        patternVerifier . put ( packet ) ;
34    }
35
36    patternVerifier . update () ;
37 }
```

LISTING 4.4: Update function of reception pipeline.

Transmission Pipeline

The transmission pipeline is similar to the reception pipeline and can be seen in Figure 4.2.

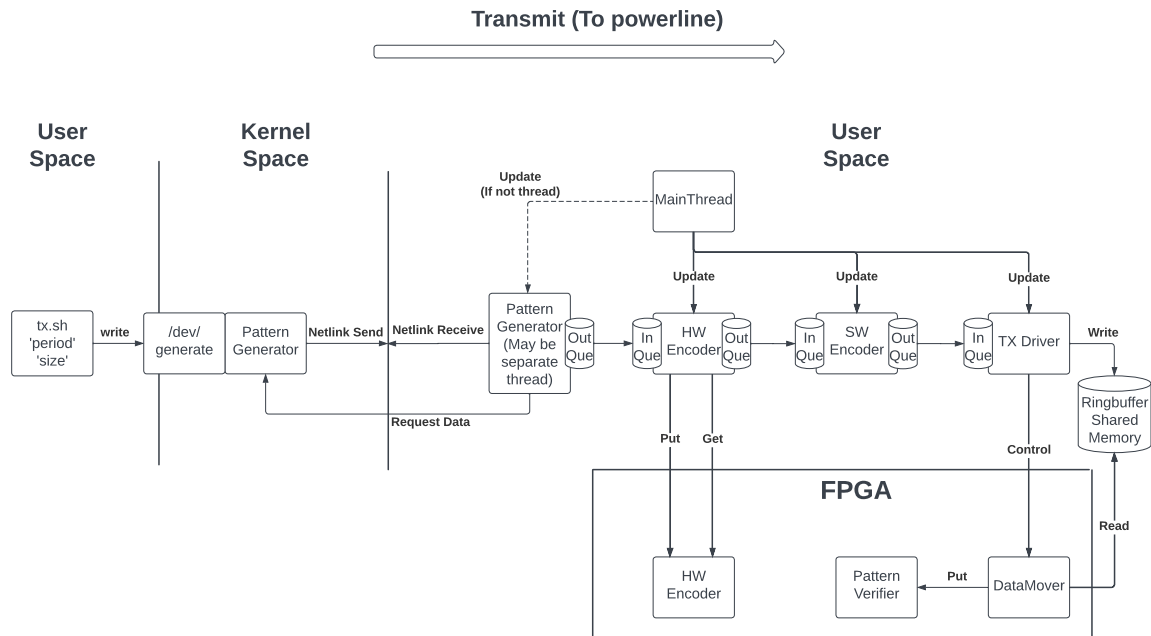


FIGURE 4.2: Design of transmission pipeline.

The kernel module *Pattern Generator* will have two interfaces. One is through a file descriptor, and the other is through Generic Netlink sockets. The file descriptor interface offers a mechanism for writing data into an internal queue, representing the ethernet frames transmitted from high-level Linux network services on the target system. When ready to transmit, the target application requests a package from the kernel module. The kernel module has a kernel thread that, upon receiving this request, transmits the newly generated package to the target system application through Generic Netlink. The justification for this design is to ensure that the kernel module handles congestion control and does not swarm the target system with packages.

Similarly to the reception pipeline, this pipeline comprises standalone components with input and output queues. However, in contrast to the reception pipeline, it will instead interpolate the data by a total factor of 8. The *HW Encoder* will interpolate by 4, using the FPGA similarly to the *HW Decoder*. The *SW Encoder* will interpolate by 2, and the *DataMover* will write the data to the shared buffer. This is illustrated in Listing 4.5, which shows a simplified version of the *TxDriver*'s `update` function.

In the reception pipeline, the shared buffer starts empty, and the software should try to keep its size as low as possible to avoid an overflow. In contrast, the transmission pipeline will start with the shared buffer at a size above a certain threshold and ensure that it will never underrun. Different thresholds can give different behaviors, and depending on the rate at which data will come through the Generic Netlink interface, it might be necessary to briefly disable the *DataMover* intermittently.

```
1 void TxPipeline :: update ()
2 {
3     if (shouldRequestMore ())
4     {
5         patternGenerator . request_message ();
6     }
7
8     // PatternGenerator Output:
9     // 0x01, 0x02, 0x03, ...
10    while (patternGenerator . packetAvailable ())
11    {
12        const auto packet = patternGenerator . get ();
13        hwEncoder . put (packet);
14    }
15
16    hwEncoder . update ();
17
18    // HW Encoder Output:
19    // 0x01, 0x01, 0x01, 0x01, 0x02, 0x02, 0x02, 0x02
20    // 0x03, ...
21    while (hwEncoder . packetAvailable ())
22    {
23        const auto packet = hwEncoder . get ();
24        swEncoder . put (packet);
25    }
26
27    swEncoder . update ();
28
29    // SW Encoder Output:
30    // 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
31    // 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
32    // 0x03, ...
33    while (swEncoder . packetAvailable ())
34    {
35        const auto packet = swEncoder . get ();
36        txDriver . put (packet);
37    }
38
39    txDriver . update ();
40 }
```

LISTING 4.5: Update function of transmission pipeline.

The actual application will transmit up to 1500-byte packages, quickly swapping between transmitting and receiving. However, the SUT will transmit or receive continuously, meaning it is not a perfect representation of the system but exhibits extreme behavior.

4.4.7 Output

A command-line dashboard was implemented to simplify debugging and development, where a separate thread gathers statistics from the other components and prints them to the console, as seen in Listing 4.6. This is silenced through command line arguments when running tests and replaced by writing a summarized version of the output to a log file.

```

1 #####
2           Begin
3 #####
4 00:00:16
5 Load Avg: 0.00 0.00 0.00 1/68 247
6 AppThread - Lat - Curr: 71 us Min: 65 us Max: 106 us Avg: 68 us
7 AppThread - Exe - Curr: 98 us Min: 6 us Max: 466 us Avg: 123 us
8 Missed Single Per 0, Missed Total Per 0
9 CPU - 246 - AppThread:
10     Curr - Usr 11.88, Sys 0.00, Tot 11.88
11     Avg. - Usr 7.07, Sys 0.00, Tot 7.07
12     Flts - Maj 0, Min 0
13     CtxS - Vol 16058, Non 0
14 CPU - 232 - GeneratorThread:
15     Curr - Usr 0.00, Sys 0.00, Tot 0.00
16     Avg. - Usr 0.00, Sys 0.00, Tot 0.00
17     Flts - Maj 0, Min 0
18     CtxS - Vol 3, Non 0
19
20
21 RX Pipeline - 1.25 Mibps
22 RxDr On - Sent: 11.30 MiB Rcv: 11.29 MiB Buf: 6.08 KiB Err: 0
23 SW Dec - Sent: 11.29 MiB Rcv: 5.64 MiB Err: 0
24 HW Dec - Sent: 5.64 MiB Rcv: 1.41 MiB Fif: Empty Err: 0
25 Verifier - Cur: 1.41 MiB Tot: 1.41 MiB Err: 0
26
27 QueSize - RxDr: 0 SwDIn: 0 SwDOut: 0 HwDIn: 0 HwDOut: 0 Veri: 0
28 QueMaxS - RxDr: 1 SwDIn: 1 SwDOut: 1 HwDIn: 1 HwDOut: 1 Veri: 1
29
30
31 TX Pipeline - 0.00 bps
32 Generate - Gen: 0.00 B Tot: 0.00 B Pac: 0 Msg: 0
33 HW Enc - Sent: 0.00 B Rcv: 0.00 B Fif: Empty Err: 0
34 TxDr Idl - Sent: 0.00 B Rcv: 0.00 B Buf: 0.00 B Err: 0
35 Underrun Flag: False, Underruns: 0
36
37 QueSize - Gen: 0 HwEIn: 0 HwEOut: 0 SwEIn: 0 SwEOut: 0 TxDr: 0
38 QueMaxS - Gen: 0 HeEIn: 0 HwEOut: 0 SwEIn: 0 SwEOut: 0 TxDr: 0

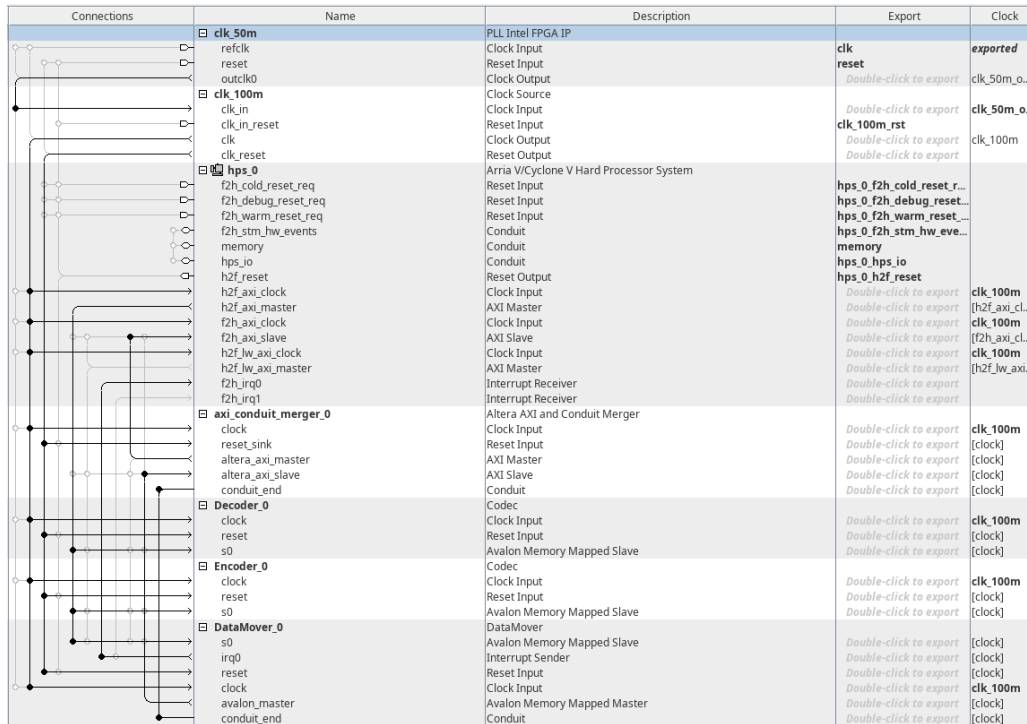
```

LISTING 4.6: Command line dashboard of SUT.

The components in the SUT design form a base for building smaller applications during the target-specific tests to evaluate the impact of different mechanisms in isolation.

4.4.8 FPGA Design

Figure 4.3 shows a high-level representation of the system configuration in Quartus' platform designer. The *DataMover*, *Encoder*, and *Decoder* are components implemented in VHDL to mock the actual FPGA logic. They connect to the HPS through various bridges as described in section 2.3.1.



Connections	Name	Description	Export	Clock
	clk_50m	PLL Intel FPGA IP		
	refclk	Clock Input	clk	exported
	reset	Reset Input	reset	
	outclk0	Clock Output	<i>Double-click to export</i>	clk_50m_o...
	clk_100m	Clock Source		
	clk_in	Clock Input	<i>Double-click to export</i>	clk_50m_o...
	clk_in_reset	Reset Input	clk_100m_rst	
	clk	Clock Output	<i>Double-click to export</i>	clk_100m
	clk_reset	Reset Output	<i>Double-click to export</i>	
	hps_0	Arria V/Cyclone V Hard Processor System		
	f2h_cold_reset_req	Reset Input	hps_0 f2h_cold_reset_r...	
	f2h_debug_reset_req	Reset Input	hps_0 f2h_debug_reset...	
	f2h_warm_reset_req	Reset Input	hps_0 f2h_warm_reset...	
	f2h_stm_hw_events	Conduit	hps_0 f2h_stm_hw_eve...	
	memory	Conduit	memory	
	hps_io	Conduit	hps_0 hps_io	
	h2f_reset	Reset Output	hps_0 h2f_reset	
	h2f_axi_clock	Clock Input	<i>Double-click to export</i>	clk_100m
	h2f_axi_master	AXI Master	<i>Double-click to export</i>	[h2f_axi cl...
	f2h_axi_clock	Clock Input	<i>Double-click to export</i>	clk_100m
	f2h_axi_slave	AXI Slave	<i>Double-click to export</i>	[f2h_axi cl...
	h2f_hw_axi_clock	Clock Input	<i>Double-click to export</i>	clk_100m
	h2f_hw_axi_master	AXI Master	<i>Double-click to export</i>	[h2f_hw_axi...
	f2h_irq0	Interrupt Receiver	<i>Double-click to export</i>	
	f2h_irq1	Interrupt Receiver	<i>Double-click to export</i>	
	axi_conduit_merger_0	Altera AXI and Conduit Merger		
	clock	Clock Input	<i>Double-click to export</i>	clk_100m
	reset_sink	Reset Input	<i>Double-click to export</i>	[clock]
	altera_axi_master	AXI Master	<i>Double-click to export</i>	[clock]
	altera_axi_slave	AXI Slave	<i>Double-click to export</i>	[clock]
	conduit_end	Conduit	<i>Double-click to export</i>	[clock]
	Decoder_0	Codec		
	clock	Clock Input	<i>Double-click to export</i>	clk_100m
	reset	Reset Input	<i>Double-click to export</i>	[clock]
	s0	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clock]
	Encoder_0	Codec		
	clock	Clock Input	<i>Double-click to export</i>	clk_100m
	reset	Reset Input	<i>Double-click to export</i>	[clock]
	s0	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clock]
	DataMover_0	DataMover		
	s0	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clock]
	irq0	Interrupt Sender	<i>Double-click to export</i>	[clock]
	reset	Reset Input	<i>Double-click to export</i>	[clock]
	clock	Clock Input	<i>Double-click to export</i>	clk_100m
	avalon_master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clock]
	conduit_end	Conduit	<i>Double-click to export</i>	[clock]

FIGURE 4.3: Quartus high-level platform overview.

4.5 Target System Experiments

As discussed in the previous section, the target system uses several concrete mechanisms that may introduce latencies into the system.

These will be implemented in isolation and benchmarked against other solutions the literature recommends. Essential factors for evaluating them include CPU load, latency, and determinism.

4.5.1 Dynamic Memory Allocation

One of the few places the target system will perform dynamic memory allocation at run-time is when interacting with the Generic Netlink interface. It will utilize *libnl* [109] for this, which requires dynamic memory allocation of packets. This allocation is explicit when transmitting and implicit when receiving. Because of this, this test aims to see how long it takes to allocate, touch, and free memory across the different preemption models.

The implementation of this can be seen in Listing 4.7.

```
1 void DynamicMemoryTest::update()
2 {
3     allocTimer.start();
4     auto* msg = nlmsg_alloc();
5     allocTimer.stop();
6
7     putTimer.start();
8     genlmsg_put(msg, NL_AUTO_PORT, NL_AUTO_SEQ, ...);
9     nla_put_u32(msg, SAI_LENGTH, 150);
10    putTimer.stop();
11
12    freeTimer.start();
13    nlmsg_free(msg);
14    freeTimer.stop();
15 }
```

LISTING 4.7: Main routine of dynamic memory test.

Separate timer class instances measure each operation and internally record the minimum, maximum, and average values. The routine has a period of 1 ms and executes for a total of 20 minutes, giving approximately 1 200 000 samples of allocations, touches, and frees. The test is invoked both with and without memory locking and heap pre-faulting.

4.5.2 Periodic Execution Test

The target system does not use sleep functions directly but blocks on a condition variable. This allows the main thread to wake up on an event; however, the likely path is for a timer to trigger the condition variable after a millisecond. This timer is implemented as a service that provides a callback after a given duration. On the previous

platform, which did not run Linux, this was internally based on a hardware timer, effectively providing the component with a millisecond tick. For debugging purposes, the legacy system could also partially run on a Linux platform, where this tick was implemented using a POSIX interval timer and *SIGALRM*. A simplified version of this can be seen in Listing 4.8.

```
1  static sem_t tick_sem;
2
3  static const struct itimerval interval
4  {
5      .it_value.tv_usec = 1000U,
6      .it_interval.tv_usec = 1000U
7  };
8
9  void* tick_thread_func(void* arg)
10 {
11     while(true)
12     {
13         sem_wait(&tick_sem);
14         ...
15         ...
16     }
17
18     return NULL;
19 }
20
21 void sig_callback(int signal)
22 {
23     if (signal == SIGALRM)
24     {
25         sem_post(&tick_sem);
26     }
27 }
28
29 int main()
30 {
31     ...
32     signal(SIGALRM, &sig_callback);
33     setitimer(ITIMER_REAL, &interval, NULL);
34     ...
35 }
```

LISTING 4.8: Periodic task implemented with `setitimer`.

This code sets up a POSIX interval timer to trigger a signal every millisecond. This signal will call the registered callback function. However, since the documentation does not recommend executing significant amounts of code in the context of a signal handler, this callback only triggers a semaphore that will trigger the execution of another thread, which is waiting for the semaphore.

This implementation looks suspicious compared to the recommendations in the literature, but it is in motion to be used in the new system. Therefore, comparing this mechanism against the recommended *nanosleep* and other options is valuable.

When comparing the interval timer to an implementation using *nanosleep*, available in Listing 4.9, the former comes across as having much more overhead.

```
1 static const struct timespec period
2 {
3     .tv_nsec = 1U * NANOS_PER_MILLI
4 };
5
6 void* tick_thread_func(void* arg)
7 {
8     ...
9
10    struct timespec wakeupTime = {0};
11    clock_gettime(CLOCK_MONOTONIC, &now);
12
13    ...
14
15    while(true)
16    {
17        ...
18
19        timespecIncrement(&wakeupTime, period);
20        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &wakeupTime, nullptr);
21    }
22
23    return NULL;
24 }
```

LISTING 4.9: Periodic task implemented with *nanosleep*.

This example shows a single thread with an eternal loop that uses *clock_nanosleep* to execute periodically, every millisecond. The thread never asks for the time after initialization but instead increments the absolute wake-up time with the period. This ensures the period does not drift over time due to scheduling latency.

Another way to implement periodic execution is to use deadline scheduling, described in section 2.2.2, and seen in Listing 4.10. In this example, the thread is registered to be scheduled under *SCHED_DEADLINE* and must only call *sched_yield()* to forfeit the CPU until the next period.

```
1 void* tick_thread_func(void* arg)
2 {
3     sched_attr.size = sizeof(struct sched_attr);
4     sched_attr.sched_flags = 0U;
5     sched_attr.sched_priority = 0U;
6     sched_attr.sched_period = 1 * 1000 * 1000;
7     sched_attr.sched_deadline = 200 * 1000;
8     sched_attr.sched_runtime = 200 * 1000;
9     sched_attr.sched_policy = SCHED_DEADLINE;
10
11     sched_setattr(gettid(), &sched_attr, 0);
12
13     while (true)
14     {
15         ...
16
17         sched_yield();
18     }
19
20     return NULL;
21 }
```

LISTING 4.10: Periodic task implemented with deadline scheduling.

A final implementation of periodic execution tested in this experiment is a condition variable with a timeout, as seen in Listing 4.11. It is possible to configure a POSIX condition variable to use a monotonic clock, similar to what the literature recommends to use with `clock_nanosleep`. Although this might indicate that the results will be similar to that of `clock_nanosleep` it is interesting to see how they measure up against one another.

```
1 static const struct timespec timeout
2 {
3     .tv_nsec = 1U * NANOS_PER_MILLI
4 };
5
6 void* tick_thread_func(void* arg)
7 {
8     ...
9
10    pthread_condattr_setclock(&cond_attr, CLOCK_MONOTONIC);
11    pthread_cond_init(&cond, &cond_attr);
12
13    ...
14
15    while(true)
16    {
17        ...
18
19        struct timespec wakeupTime = getTimespecNow();
20        timespecIncrement(&wakeupTime, timeout);
21
22        pthread_mutex_lock(&mutex);
23        pthread_cond_timedwait(&cond, &mutex, &wakeupTime);
24        pthread_mutex_unlock(&mutex);
25    }
26
27    return NULL;
28 }
```

LISTING 4.11: Periodic task implemented using condition variable with timeout.

The benefit of this mechanism is that it is a drop-in replacement for the existing solution. According to the literature, it should result in lower latencies, and since it makes the infrastructure required for the interval timer redundant, it should also use significantly less CPU time.

4.5.3 Shared Memory Interaction

The primary real-time constraint on the software system is that the FPGA will read and write to the shared memory area at a fixed rate. This test consists of a thread that reads from the shared memory while the system's key performance indicators are monitored and compared across different scheduling policies, preemption models, and execution schemes. The execution schemes refer to whether the task reads a fixed amount of data each period or uses the aperiodic concept described in section 2.3.3, where it will only sleep when the shared memory buffer is empty or below a certain threshold.

The software used in this test will be a subset of the SUT described in section 4.4.6, and can be seen in Figure 4.4

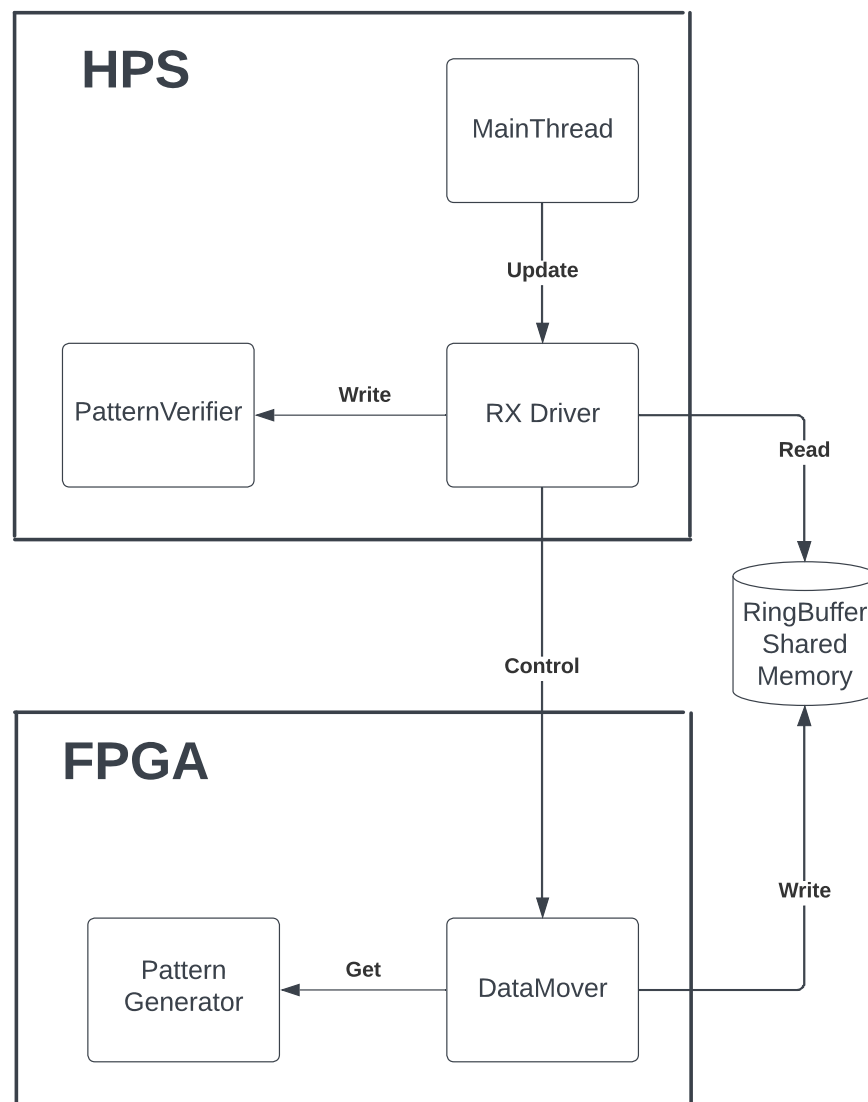


FIGURE 4.4: Main routine of shared memory test.

In this case, the pattern verifier is placed in user space and will execute in the same thread as the rest of the software. This added overhead will mimic the time required to search through each package for a start pattern in combination with an added 50 μ s busy wait for each package read from the shared buffer.

The scheme is tested using packet sizes of 4 KiB and 512 B, respectively. The only system calls used in this test are those required to sleep and busy wait, so the only factors expected to affect the achievable performance are the system's scheduling latency and the main loop's execution time. A simplified excerpt of the test can be seen in Listing 4.12.

```
1 bool RxDriver::update()
2 {
3     if (!ringBuffer.isEmpty())
4     {
5         const auto packet = ringBuffer.readPacket();
6         patternVerifier.put(packet);
7         busyWaitUs(50U);
8
9         return true;
10    }
11
12    return false;
13 }
14
15 void SharedMemoryTest::thread_fn()
16 {
17     while(true)
18     {
19         const bool packetRead = rxDriver.update();
20
21         if(periodic)
22         {
23             timespecIncrement(&nextPeriod, period);
24             clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextPeriod, ..);
25         }
26         else
27         {
28             if (!packetRead)
29             {
30                 auto wakeupTime = getTimespecNow();
31                 timespecIncrement(&wakeupTime, period);
32                 clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &wakeupTime, ..);
33             }
34         }
35     }
36 }
```

LISTING 4.12: Shared memory test.

4.5.4 Generic Netlink Interface

As mentioned, the target system's user space application will communicate with an LKM using Generic Netlink sockets. This test will, in isolation, ask a kernel module for data and, upon receiving it, directly write it to another kernel module. Through this test, it is possible to measure the potential reduced throughput of these particular system calls in kernels with a higher degree of preemptibility.

The subset of the SUT used in this test can be seen in Figure 4.5.

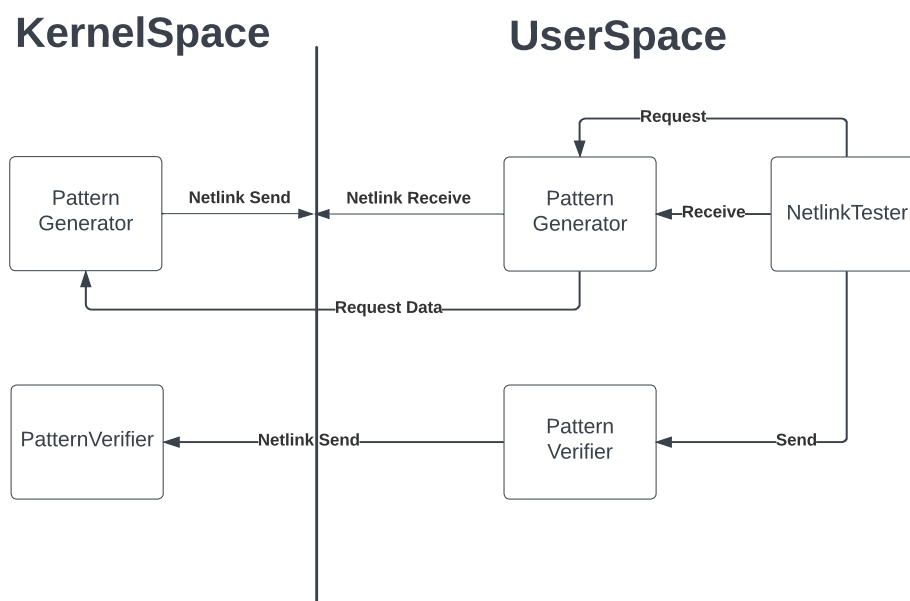


FIGURE 4.5: Generic Netlink throughput test.

In this case, the kernel module is modified not to require an external component to write data to its queue but to immediately send newly generated data to the client on each request.

A simplified version of the test can be seen in Listing 4.13.

```

1 void GenericNetlinkTester::update()
2 {
3     requestTimer.start();
4     generator.request_message();
5     requestTimer.stop();
6
7     readTimer.start();
8     const auto packet = generator.read();
9     readTimer.stop();
10
11    writeTimer.start();
12    verifier.verify(newPacket);
13    writeTimer.stop();
14 }

```

LISTING 4.13: Main routine of Generic Netlink throughput test.

Requesting data from the driver is achieved by sending an empty netlink message, in this case, abstracted by a *Generator*. After sending a request, the client will block on a call to `nl_recvmsgs_default` until a response is received. A thread in the driver (as described in section 4.4.6) will detect this event and continue to write a package of 1500 bytes to the client application. After receiving the message, the client will write it to

another kernel module, abstracted by a *Verifier* class. The duration of these operations is measured, and the software stores average and maximum values.

As discussed, *libnl* requires dynamic allocation of messages through *nlmsg_alloc*. The memory is locked in RAM, and the heap is manually pre-faulted before execution to limit the effect of page-faulting on the results.

4.5.5 Kernel Thread Event Handling

The planned implementation of the kernel thread that provides data to the pipeline for transmission may not be optimal. The existing driver does not block between requests for data from the application but polls for new events. This means that it can potentially burn considerable CPU time. For sleeping, the target system utilizes *usleep_range* between polls, as illustrated in Listing 4.14.

```
1 void kthread_worker()
2 {
3     while(!exit)
4     {
5         while(true)
6         {
7             if(new_event() || exit)
8             {
9                 break;
10            }
11
12            usleep_range(10, 100);
13        }
14
15        //Handle event or exit
16        ...
17    }
18 }
```

LISTING 4.14: Polling kernel worker.

Although this is the recommended mechanism for sleeping for the given timeframe (as discussed in section 3.6.4), it might be more efficient in this case to make the kernel module event-driven by utilizing a *wait_queue* as illustrated in Listing 4.15.

```
1 void kthread_worker ()
2 {
3     while (!exit)
4     {
5         if (!new_event())
6         {
7             wait_event_interruptible(wait_queue_etx, new_event() || exit);
8         }
9
10        //Handle event or exit
11        ...
12    }
13 }
```

LISTING 4.15: Event-driven kernel worker.

These two mechanisms are tested with events occurring at different frequencies. Depending on the frequency of the events, the polling mechanism might be more efficient than the event-driven mechanism, and given a high enough rate of events, the CPU load between the two mechanisms could even out.

The following event frequencies are considered:

- No events
- 1 Hz
- 10 Hz
- 1 kHz
- 10 kHz

4.5.6 Full Test

The full test involves running the full pipelines described in section 4.4.6. In the reception pipeline, the FPGA sets the bitrate externally, and the software is required to keep up. This means that all issues will manifest as errors. The transmission pipeline is different, as it must request data from the kernel module at a sufficient rate to keep the buffer from underrunning. In other words, the software is responsible for keeping the bitrate up.

The full tests either transmit or receive data indefinitely. The reception pipeline mimics a real scenario where the system continuously searches for a preamble in the incoming data. The transmission pipeline, however, has no real-world counterpart to this scenario as the system will write in bursts before switching back to a reception mode. Because of this, the test is not as realistic as one would desire and will most likely demand more of the OS than the actual application.

The package size for the full test will be 1504 bytes. This is because it is approximately the size of an ethernet frame, which is the actual payload the system will handle towards the Netlink interface. This packet size is used across the pipeline for simplicity,

meaning the shared memory interaction will also use a packet size of 1504 bytes. The reason for choosing 1504 over 1500 is because the implementation of the packet generator demands that the packet size be a multiple of 8.

Special Considerations for Transmission Test

This test involves an almost limitless amount of parameter combinations. As in the other tests, the main parameters are the scheduling policy, preemption model, and execution scheme, but for the transmission test, the performance and stability depend on several additional factors:

- **Condition for sleeping:** The aperiodic scheme will only sleep when there is nothing to do, and the definition of *nothing* decides how often the task will sleep. If it sleeps too often, the buffer will underflow, but if it sleeps too seldom, the execution time will grow, leading to real-time throttling if the task has a real-time scheduling policy. This condition is straightforward for the reception pipeline, as it will sleep when the buffer is empty. Concerning the transmission pipeline, many more viable definitions exist for this condition, which can significantly impact the performance differently.
- **Condition for requesting data:** The transmission pipeline will request data from the kernel module when it has the capacity to do so. How this is defined will determine how often it will request more data and how busy the task will be. Since the eternal transmission loop performed in this test does not represent the actual system, requesting data too frequently and keeping the pipeline overly busy will mean that the aperiodic task might get too long execution times. The same scenario for a periodic task can make it unable to maintain its period, essentially turning it into an aperiodic task.
- **Scheduling policy and priority of kernel thread:** Upon receiving a request from the application, a task in the kernel module will provide it with a package. If this kernel schedules this task under CFS, it might not be responsive enough to handle the request in time. Although this can cause this test to fail, the actual system will not necessarily be affected by this, as it would instead just stop transmitting.
- **Separate thread for Generic Netlink Interface:** The reception pipeline will perform a synchronous write to the kernel module through Generic Netlink, meaning that doing so from the main thread will most likely cause limited latencies. In contrast, the transmission pipeline will need to request a package and then block until receiving it, which can cause massive latencies. This can be solved by having a separate thread for receiving data from the kernel module. However, this also brings forth new potential issues and design choices to make.

Naturally, only a subset of the possible combinations can be tested, and the results are not trivial to analyze in the context of determining what can benefit the actual system.

4.5.7 Production Environment Test

Whether or not `PREEMPT_RT` is required or optimal for the target system also depends on the background workload of the target system. The existing SW platform

has several *daemons* and periodic background tasks for housekeeping and IO across the system. The nature of these tasks can significantly affect the responsiveness of the system if they are heavy in long-duration system calls. Therefore, it is natural to perform some of the tests on the existing platform to see if any potential issues arise.

As the scope of this test is to uncover any mechanisms that impact the real-time capabilities or are significantly affected by changing the preemption model, the tests will be limited to the Cyclictest configuration described in section 4.3.2 and an analysis of the load and CPU utilization on an otherwise idle system.

4.6 Summary

This chapter has described and justified the methods used in this thesis. It has outlined the platform built for this project and described several generic tests to evaluate its real-time capabilities and performance in several configurations. Further, the target system was analyzed in the context of findings in the literature, and its real-time constraints were derived from the functional requirements described in Chapter 2. From this analysis, a SUT to mimic the functionality of the target system was presented, and various tests were designed to evaluate the impact of different mechanisms on the system's ability to meet its requirements.

Chapter 5

Test Results and Recommendations

This chapter presents the results of the tests executed on the platform and target system. Throughout this chapter, figures provide the foundation for presenting the results. However, tables providing the raw data can be found in Appendix C, along with any additional figures deemed redundant for the main text.

Unsurprisingly, compiler optimization settings had a significant impact on the results. When running custom test software compiled with `-O0`, the results were considerably worse than with `-O3`. Therefore, all results reported in the thesis are produced with software compiled with `-O3`.

Another notable finding during these tests is that the distinction regarding CPU load bound in user space vs. kernel space can not necessarily be trusted. Different executions of the same SW and configuration gave different results and were seemingly more off on periodic tasks. This was determined to be due to the nature of tick-based CPU time accounting, which is the default setting on ticked kernels. The results were more consistent and aligned with expectations when this was deactivated. However, it also introduced a small overhead and was not used while conducting the experiments.

5.1 Platform Baseline Tests

This section presents the results of the tests described in section 4.3. In cases where the results were unexpected or unclear, additional tests were designed and executed. These are described in the respective sections when applicable.

5.1.1 High-Resolution Timers Verification

The output of `/proc/timerlist` can be seen in Listing C.1 in C.

It shows that all timers have a 1 ns resolution. The `event_handler` parameter of the respective CPUs local timer is set to `hrtimer_interrupt`, which is correct according to [110]. Additionally, `hres_active` is set to 1, indicating the feature is enabled.

5.1.2 Cyclicttest

The results of executing the Cyclicttest configurations described in section 4.3.2 on an otherwise idle system are available in Figure 5.1.

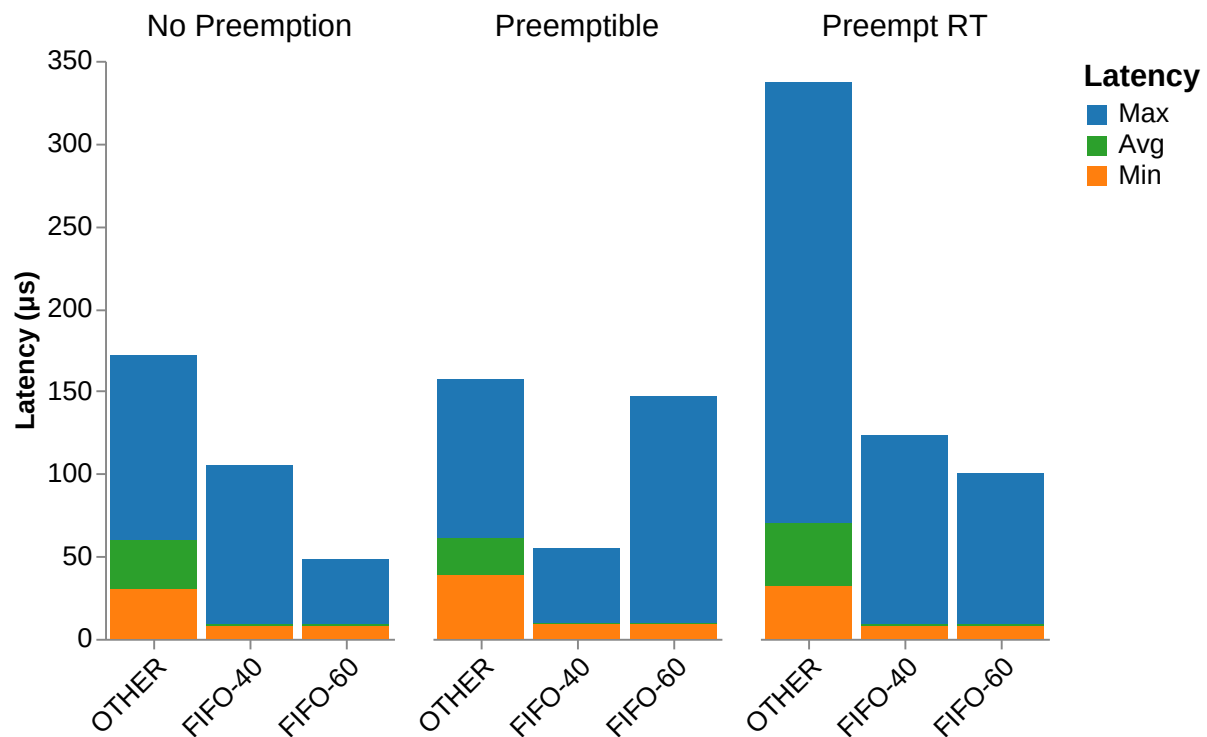


FIGURE 5.1: Cyclictest results, no load.

Using `SCHED_FIFO` with a priority of 60 on a non-preemptible kernel outperforms the other parameters in this setup, with a max latency of about 50 μs . Unexpectedly, the fully preemptible kernel performs worse than the non-preemptible kernel, with a max latency of about 100 μs using `SCHED_FIFO`. The results of the fully preemptible kernel are comparable to what was found by [3] and OSADL results at the time of writing [83], as discussed in section 3.5.1.

Figure 5.2 shows the results of the same tests executed under a 25% load induced by `Stress-NG`. This figure highlights the immense significance the scheduling policy of a task has on its latency compared to the preemption model of the kernel. Puzzlingly, the fully preemptible kernel performs slightly better under `SCHED_OTHER` than the non-preemptible kernel, starkly contrasting the results found on the idle system. It also directly contradicts the expectations that the fully preemptible kernel would perform worse when scheduled under `CFS`, as the overhead in kernel space should be higher.

As expected, on the fully-preemptible kernel, the task has a slightly lower latency when prioritized above 50. However, this is not the case for the preemptible kernel. Part of the explanation for this can be that the preemptible kernel does not have forced threading of its interrupts, but that should leave the tests executing with 40 and 60 with similar results. The remaining difference can be explained as an outlier, as the test only ran once.

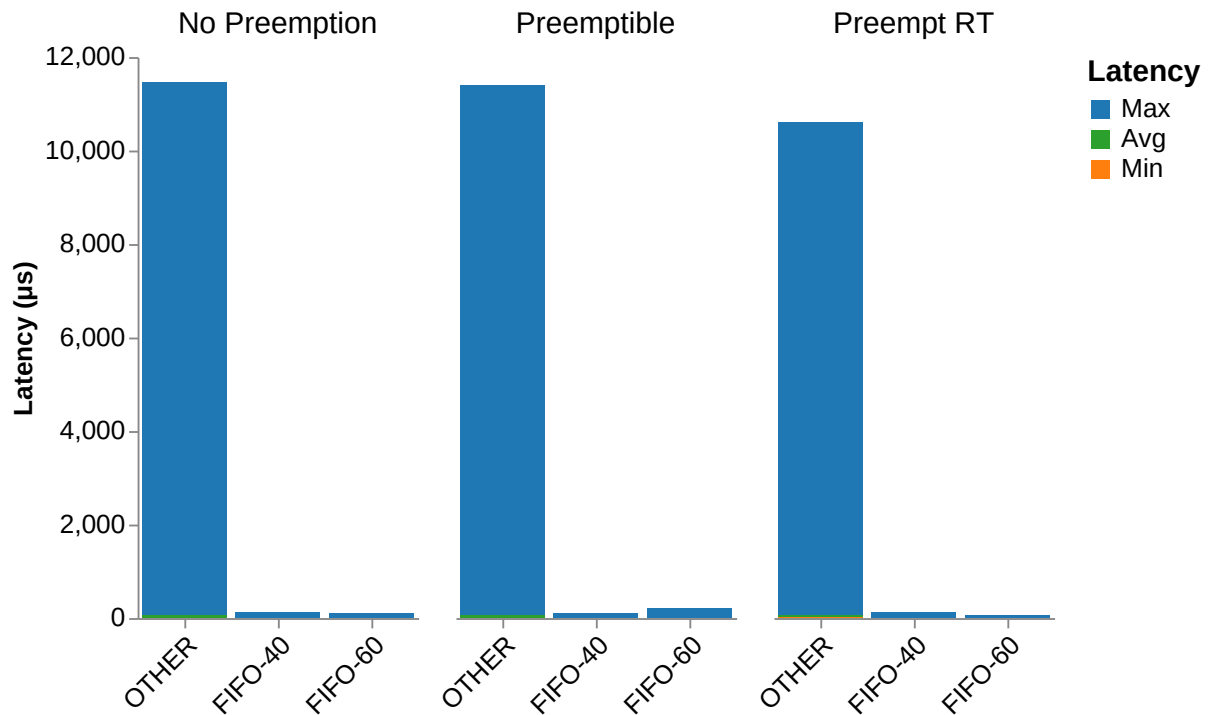


FIGURE 5.2: Cyclictest results, Stress-NG 25 %.

Because the CFS scheduled tasks muddle the graphic representation of the results of the tests executed with load, they are not that useful for comparison. Nevertheless, the remainder of these results can be seen in Figure C.1, in Appendix C.

All tests running under SCHED_FIFO with a priority of 60 are illustrated in Figure 5.3, grouped by external stressors on the system.

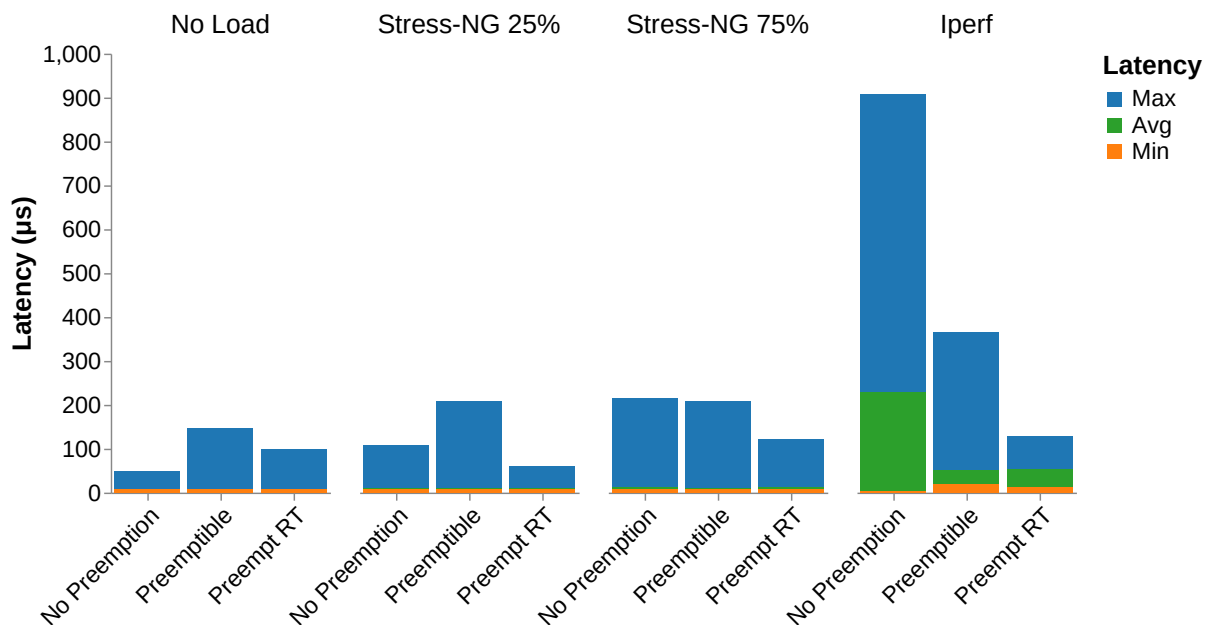


FIGURE 5.3: Cyclictest with SCHED_FIFO.

When stressing the system with Stress-NG, the fully preemptible kernel performs slightly better than the non-preemptible kernel. This finding is unsurprising because this configuration of Stress-NG operates primarily in user space. When stressing the system using Iperf, which operates mainly in kernel space, the fully preemptible kernel performs significantly better than the non-preemptible kernel, both on average and in terms of max latency. Although the preemptible kernel performs on average equally well as the fully preemptible kernel, its max latency is significantly higher.

Even though the difference between the preemption models might look high in some of these cases, they all performed with less than a millisecond of latency in all scenarios. It is also worth noting that while indicating the system's capabilities, the tests are limited in duration. Nevertheless, the results are very similar to those found by Adam et al. [3] and OSADL [83], as discussed in section 3.5.1.

This test was also performed on the kernel used in company production environments. The reason for this was to uncover any significant discrepancies compared to the kernel version built for the thesis, both concerning the version of the kernel and its configuration. This kernel is based on version 5.19 and, on the initial run, showed spikes of latency of up to 40 milliseconds, as seen in Listing 5.1.

```
1 # ./cyclic.sh fifo99 Stress-NG 25
2 T: 0 ( 171) P:99 I:400 C:5000000 Min: 10 Act: 12 Avg: 13 Max: 39771
3 T: 1 ( 172) P:99 I:900 C:2222542 Min: 11 Act: 12 Avg: 14 Max: 93
```

LISTING 5.1: Cyclicttest on the company's production kernel.

After carefully examining the kernel configuration, it became clear that *kmemleak* was enabled, which can cause notable latencies and overhead according to [111]. Listing 5.2 shows the results of Cyclicttest on version 5.19 with *kmemleak* disabled. It shows that the latencies were slightly higher than with version 6.6 of the kernel but still significantly lower than with *kmemleak* enabled.

```
1 # ./cyclic.sh fifo99 Stress-NG 25
2 T: 0 ( 145) P:99 I:400 C:5000000 Min: 11 Act: 17 Avg: 14 Max: 158
3 T: 1 ( 146) P:99 I:900 C:2222217 Min: 11 Act: 13 Avg: 14 Max: 73
```

LISTING 5.2: Cyclicttest on the company's production kernel, without *kmemleak*

Kernel 5.19 was also tested under different preemption models with *kmemleak* enabled, as seen in Figure 5.4. As expected, `PREEMPT_RT` was able to mitigate the latency brought on by *kmemleak*; however, somewhat surprisingly, so was the preemptible kernel, as seen in Figure 5.4. This indicates that while *kmemleak* spends significant time in kernel space, the critical sections are not long enough to cause significant latencies on the preemptible kernel.

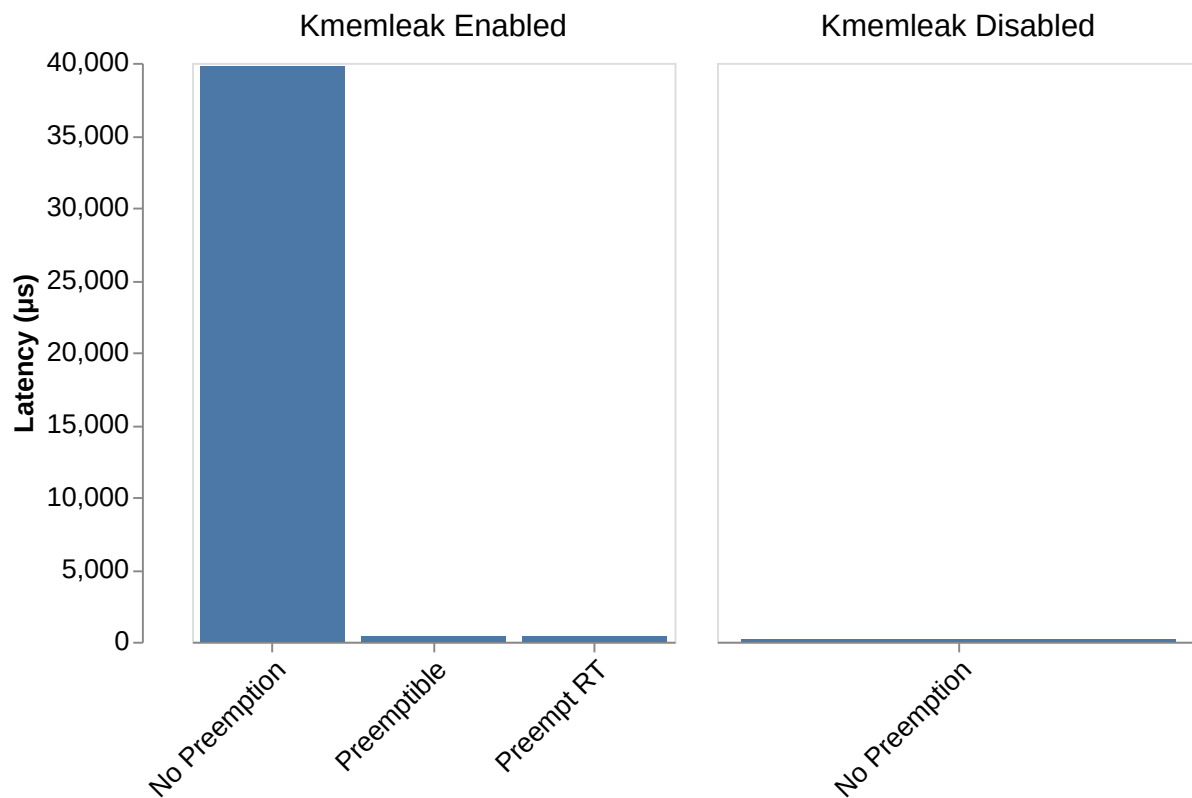


FIGURE 5.4: Cyclictest on the company's production kernel.

Still, the results show that, while producing significant improvements to the latency, using a preemptible or fully-preemptible kernel is not as effective as disabling *kmemleak* on a non-preemptible kernel.

Overall, these results are comparable to those found on the 6.6 kernel, meaning that the Cyclictest results and subsequent latency results should be relevant for the kernel version used in production environments at TechipFMC.

5.1.3 Throughput Test

The throughput test results can be seen in Figure 5.5, which shows the total run time of both tests across the different preemption models.

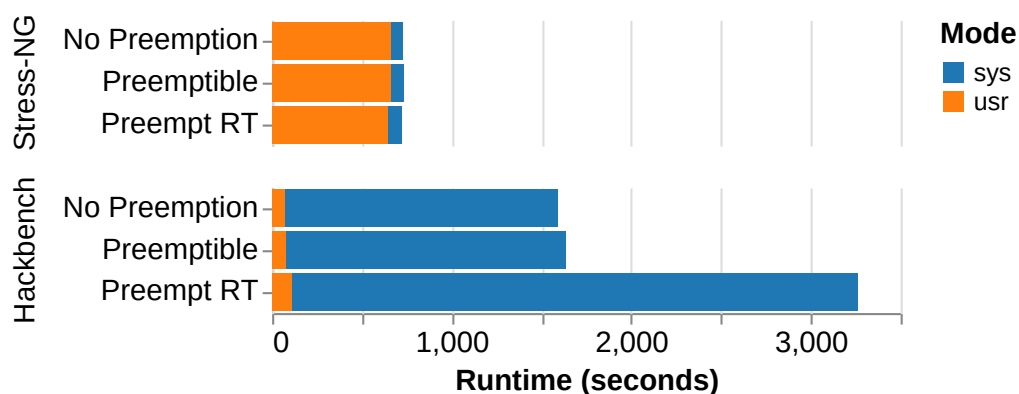


FIGURE 5.5: Throughput comparison across different preemption models.

Using Stress-NG, the difference in overall throughput is miniscule. The preemptible kernel spends the same amount of time in user space as the non-preemptible kernel but spends slightly longer in kernel space, as was expected based on the literature review. The fully preemptible kernel spends slightly longer in kernel space than the preemptible kernel but less than the other preemption models in user space. Although surprising, as these tests schedule their tasks under CFS, this could be explained by a slightly different system base load.

The Hackbench invoked throughput test shows a dramatic throughput reduction on the fully preemptible kernel compared to the others. In contrast to the Stress-NG test, Hackbench spends most of its time in kernel space, which could explain the difference in throughput. The preemptible and non-preemptible kernels spend about the same duration in user space. Still, the preemptible kernel spends slightly longer in kernel space, slightly reducing its throughput.

These results highlight the importance of analyzing the nature of the workload before selecting a preemption model. As Madden points out in [2], the fully preemptible kernel best suits CPU-intensive workloads with few system calls. The lack of throughput degradation in the CPU-intensive Stress-NG test and the vast reduction in throughput in the system-call-heavy Hackbench test confirm this claim.

One explanation for the throughput reduction found using Hackbench could be the overhead of the fully preemptible kernel. Since there was close to no degradation in throughput found using the preemptible kernel, this indicates that the overhead of the preemptible RT-mutexes is vastly more significant than the overhead of simply being preemptible. `PREEMPT_RT`, in addition to being fully preemptible, also enables forced threading of all interrupts, as discussed in section 3.3.3. This could be another element that led to these results.

To further investigate the high discrepancies in throughput found using `PREEMPT_RT`, the test was executed again using *pipes* as the IPC mechanism. Figure 5.5 illustrates how they compared against the default *Unix sockets*.

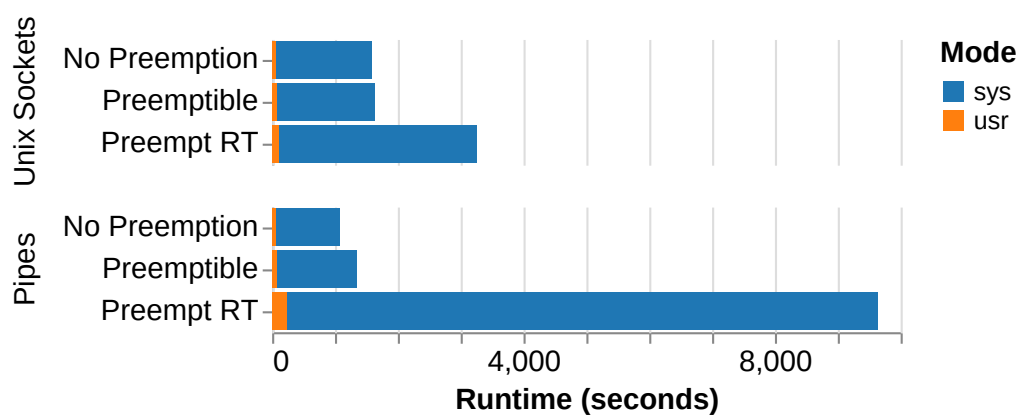


FIGURE 5.6: Hackbench throughput comparison across preemption models using Unix sockets and pipes.

Surprisingly, the difference in throughput was even more pronounced when using pipes as the IPC mechanism. This was true for the preemptible kernel but especially

the fully preemptible kernel. The reasons for this are probably the same as those found in the Unix socket test. However, it highlights that different system calls may give drastically different results concerning throughput degradation and that the applicable system calls for a system should be considered during system design.

When rerunning the test with real-time scheduling policies, the stressors were subject to real-time throttling, but this should not have a significant impact on the results, as it is only throttled for 5 ms every 1 s (and should be the same for all preemption models). Using a high priority for the Hackbench test led to a kernel panic on the fully preemptible kernel, as the `rcu_preempt` thread experienced starvation. Reducing the priority to 1 and the number of messages from 50 000 to 5000 led to a system stable enough for the test to complete. The results can be seen in Figure 5.7.

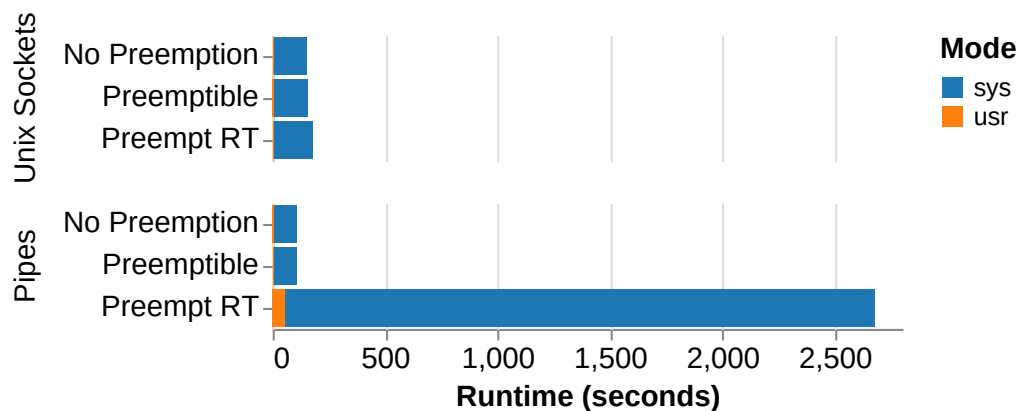


FIGURE 5.7: Hackbench throughput comparison across preemption models using Unix sockets and pipes, scheduled under SCHED_FIFO.

Pipes show an even more pronounced throughput degradation when scheduled under SCHED_FIFO on the fully preemptible kernel. In contrast, real-time scheduling policies positively impacted the Unix sockets. Although the reason for this remains unclear, it eliminated the difference across the preemption models found in the previous experiment.

Due to the surprising nature of these results, devising additional tests to gain more insight into the throughput degradation of PREEMPT_RT became desirable.

Firstly, Iperf was used to set up a TCP connection on localhost. As seen in Listing 5.3, the configuration was set to transmit a fixed amount of data, where the total duration would provide a measure of throughput.

```

1 iperf3 --server > /dev/null 2>&1 &
2 time iperf3 --client localhost \
3     --interval 0 \
4     --blockcount 500000 \
5     --omit 10

```

LISTING 5.3: Iperf configuration for throughput test.

As Iperf primarily executes in kernel mode, the results were similar to those of the Hackbench test but with less pronounced differences, as seen in Figure 5.8. Again, this shows that the actual throughput reduction depends on the system calls used by the workload.

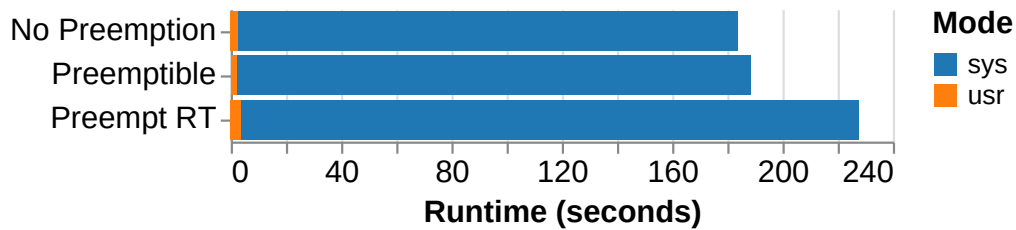


FIGURE 5.8: Throughput comparison of Iperf across different preemption models.

p7zip (a 7-zip fork) is a popular file-archiving utility that additionally provides a system benchmarking tool, which can be invoked as seen in Listing 5.4.

```
1 time 7zr b -mm=*
```

LISTING 5.4: 7z benchmark.

With this configuration, p7zip performs a so-called *complex* benchmark, which involves many different operations. The throughput measure is the benchmark duration and the reported millions of instructions per second (MIPS). The results can be seen in Figure 5.9 and Figure 5.10.

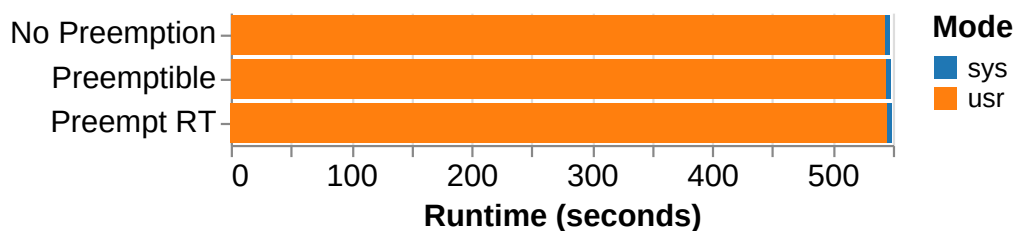


FIGURE 5.9: Throughput comparison of 7z across different preemption models.

As the operations performed by p7Zip are executed almost exclusively in user space, the different preemption models again perform about the same. The throughput decreases as the kernel becomes increasingly preemptible, but the difference is insignificant.

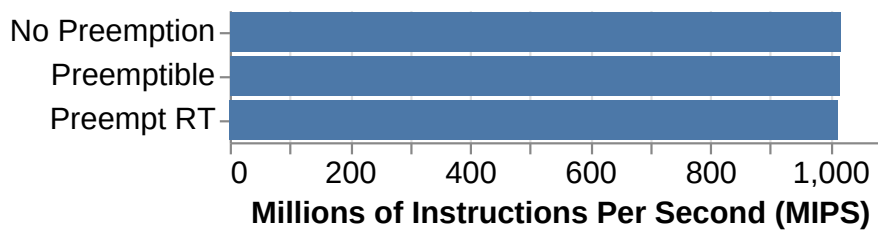


FIGURE 5.10: Throughput comparison of 7z across different preemption models measured in MIPS.

5.1.4 Preemption Test

Figure 5.11 shows the results of the preemption test executed under CFS.

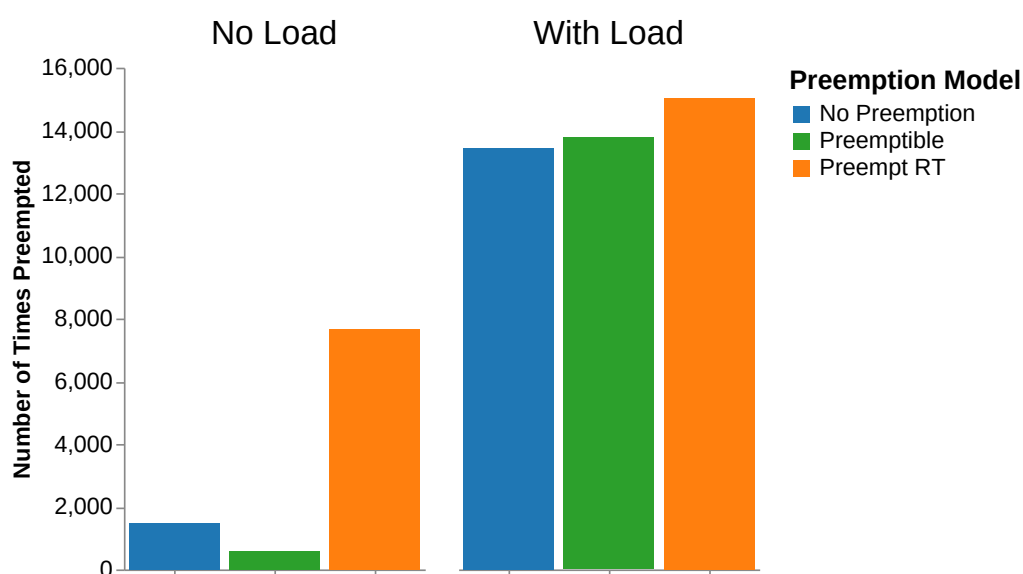


FIGURE 5.11: Preemption count of CPU-intensive task under SCHED_OTHER.

These results were as expected. With only the minimal background load on the system, the fully preemptible kernel had a much higher preemption count than the other preemption models, most likely due to threaded interrupts. The non-preemptible kernel and preemptible kernel have interrupts executing in hard interrupt context, meaning that although interrupted, it is not counted as a preemption. In contrast, the fully preemptible kernel has all interrupts threaded, meaning that they are counted as preemptions by *rusage* and somewhat muddle up our results.

When applying load to the system, the number of preemptions increases significantly, as is to be expected. With the introduction of more load, the timeslice of the CPU-intensive task reduces, and when exhausted, the load-inducing task preempts it. There is a somewhat higher preemption count on the fully preemptible kernel, which again can be explained by the threaded interrupts.

The results of the same test repeated with SCHED_FIFO can be seen in Figure 5.12.

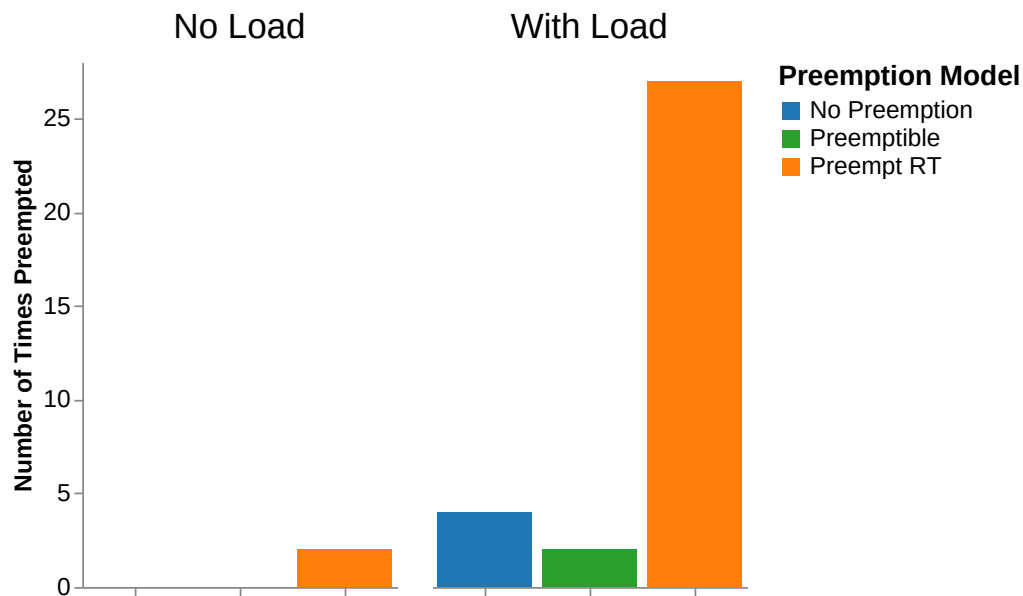


FIGURE 5.12: Preemption count of CPU-intensive task under SCHED_FIFO.

In this scenario, one would expect close to no preemptions, as the task is running at a higher priority than the load-inducing task and any threaded interrupts. However, there is still a higher preemption count when the system with the fully preemptible kernel is under load.

To investigate this, an *eBPF*¹ program was used to trace the beneficiary of these preemptions, as seen in Listing 5.5.

```

1 SEC("tp_btf/sched_switch")
2 int BPF_PROG(sched_switch, bool preempt, struct task_struct *prev, struct
  task_struct *next)
3 {
4     if (bpf_strncmp(prev->comm, 16, "Preemption_Test") == 0)
5     {
6         if (preempt)
7         {
8             bpf_printk("Preempted by: %d - %s\n", next->pid, next->comm);
9         }
10    }
11
12    return 0;
13 }

```

LISTING 5.5: eBPF program to detect preempting task.

This program was injected and hooked to the *sched_switch* trace point. This trace point is found in the kernels *__schedule* function, meaning that it is called every time the scheduler is about to perform a context switch [27, kernel/sched/core.c, Line 6695].

¹Extended Berkeley Packet Filter (eBPF) is a mechanism that enables the injection of code into the kernel at run-time. It has many use cases, one of them being tracing and debugging [112].

The eBPF injected code checks if the task preempted is our CPU-intensive task, and if so, it prints the PID and name of the task preempting it.

After running this program, it became apparent that the *migration task* was responsible for the preemptions in every case. This makes sense as it is the only task capable of doing so with its priority of 99. This was not the case when running the test on the other preemption models, so the natural assumption is that since all interrupts are threaded, the migration task is triggered to ensure that they get run time on the other CPU.

To better understand this, another eBPF program was devised, available in Listing 5.6, which is injected to the *sched_migrate_task* tracepoint, invoked on every task migration [27, kernel/sched/core.c, Line 3386]. Paired with the injected code in Listing 5.5, this facilitated seeing which task was subject to migration when the migration task would preempt the CPU-intensive task.

```
1 SEC("tp_btf/sched_migrate_task")
2 int BPF_PROG(sched_migrate_task, struct task_struct* p, int new_cpu)
3 {
4     char curr_comm[16];
5     bpf_get_current_comm(curr_comm, 16);
6
7     if ((bpf_strncmp(curr_comm, 16, "migration/0") == 0) ||
8         (bpf_strncmp(curr_comm, 16, "migration/1") == 0))
9     {
10        bpf_printk("%s, migrating %s to CPU %d\n", curr_comm, p->comm,
11                new_cpu);
12    }
13    return 0;
14 }
```

LISTING 5.6: eBPF program to detect task migration.

The results of this test can be seen in Figure 5.13, which shows all migrations detected throughout the run time of the CPU-intensive task. The CPU-intensive task only experienced preemptions on the fully preemptible kernel, and in all cases, it was preempted by the migration task. This happened a total of 20 times. In all of these cases, the migration task proceeded to move the CPU-intensive task to the other CPU. Iperf was migrated many times on all kernels but significantly more on the non-preemptible kernel.

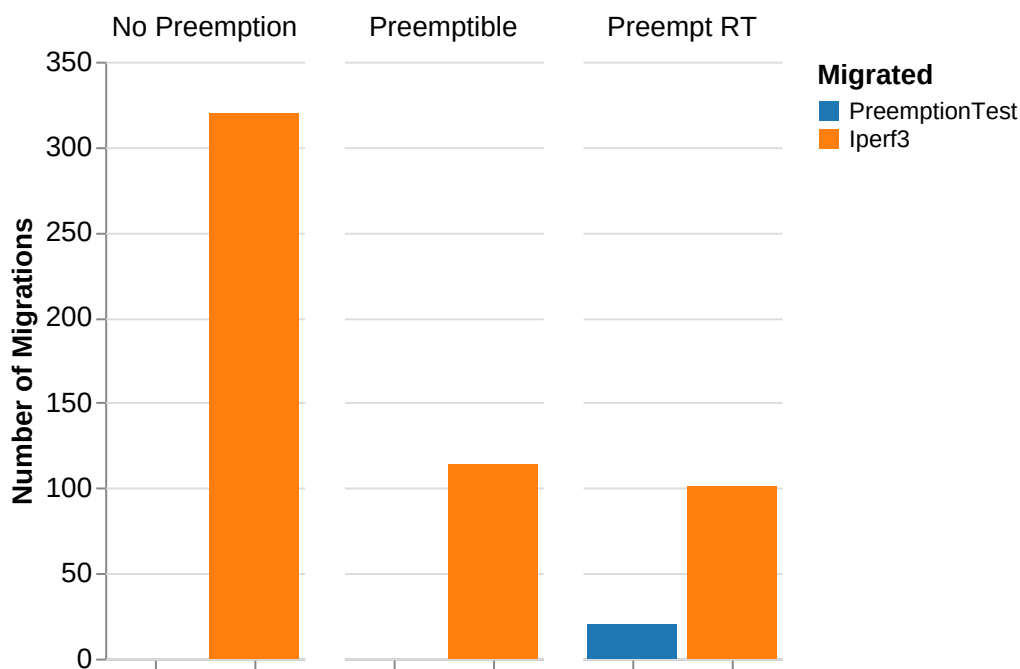


FIGURE 5.13: CPU migrations detected using eBPF.

The reasons for this are unclear. One theory could be that migrations to the other CPU are more prevalent since Iperf heavily uses system calls and cannot be preempted on the non-preemptible kernel. Another test was created to see the number of times the CPU-intensive task would preempt Iperf on all three kernels. These results are illustrated in Figure 5.14.

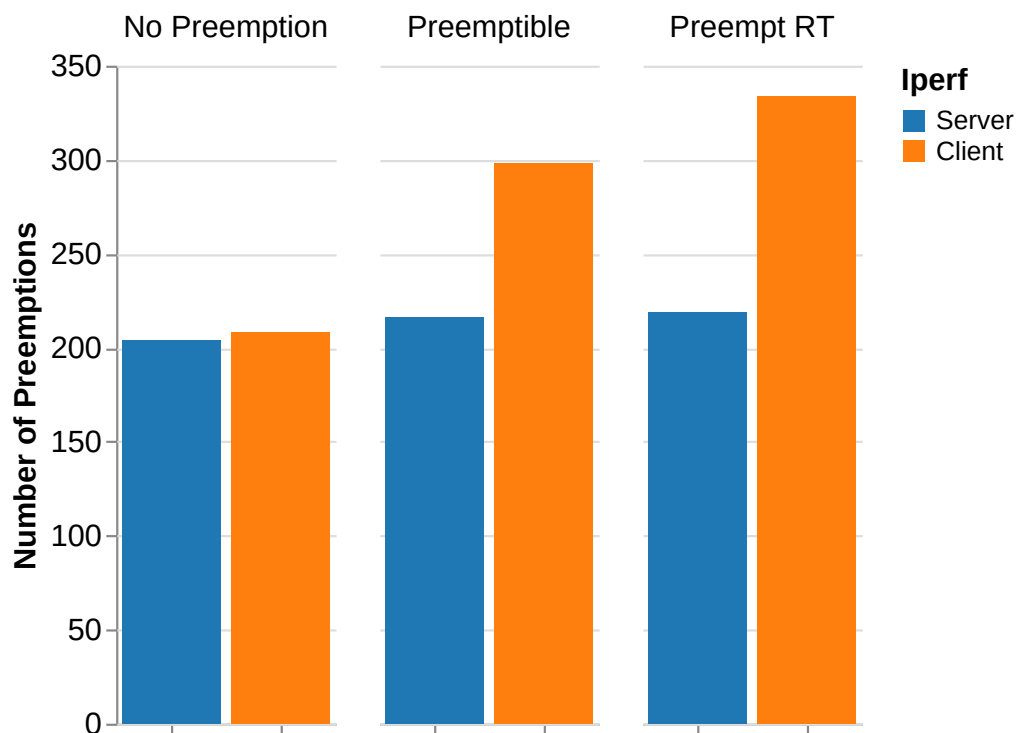


FIGURE 5.14: CPU-intensive task preempting Iperf.

There does seem to be a correlation between the number of migrations and the number of times the CPU-intensive task preempts Iperf. The number of preemptions does not account for the entire discrepancy but appears to have had a significant impact.

Interrupt Latency

Another fact worth noting is that while running under real-time scheduling policies on a fully preemptible kernel, the warning seen in Listing 5.7 was encountered at regular intervals.

```
1 [ 3023.647003] dw_mmc ff704000.flash: Unexpected interrupt latency
```

LISTING 5.7: Interrupt latency caused by CPU-intensive SCHED_FIFO task.

Our CPU-intensive task is running at a higher priority than the threaded interrupts, causing a high latency in interrupt handling. Neither the task nor the interrupt was tied to any particular CPU, meaning that ideally, the kernel should have moved the task or handled the interrupt on another CPU to avoid this. For unknown reasons, this did not occur. However, the warning did not reappear after setting the task's CPU affinity or moving the interrupt to another CPU.

On subsequent test executions, this problem was not always present, meaning developers should be mindful of situations like this in production environments.

Kernel Preemption Test

A test was also conducted out of purely academic interest to verify that the preemptible kernels are, in fact, preemptible. This also provides insight into how often a high-priority task directly benefits from the preemption model.

Cyclictest was again chosen for this, and it ran with Iperf as a stressor. As the fully preemptible kernel has threaded IRQs, Cyclictest ran at a priority of 40 so that it would not preempt the interrupts, making the test as similar as possible across the different preemption models. Iperf was scheduled under SCHED_OTHER, and the test ran for 500 000 loops, or just over 3 minutes, long enough for any differences to become apparent.

As seen in Listing 5.8, the eBPF program was modified to detect when Cyclictest preempted a task executing in kernel mode. The eBPF-injected code checks if the next task is Cyclictest and whether it is about to preempt the previous task. If it is, and the previous task was also executing in kernel mode, it logs it.

```

1 #define ARM_cpsr      (uregs[16])
2 #define user_mode(regs) (((regs)->ARM_cpsr & 0xf) == 0)
3
4 SEC("tp_btf/sched_switch")
5 int BPF_PROG(sched_switch, bool preempt, struct task_struct *prev, struct
6   task_struct *next)
7 {
8   if (bpf_strncmp(next->comm, 16, "cyclictst") == 0)
9   {
10    if (preempt)
11    {
12     struct pt_regs *prev_regs = (struct pt_regs *) bpf_task_pt_regs(prev)
13     ;
14
15     if (!user_mode(prev_regs))
16     {
17      bpf_printk("Preempting: %d - %s\n", prev->pid, prev->comm);
18     }
19    }
20 }
21 return 0;

```

LISTING 5.8: eBPF program to detect when preempting task executing in kernel mode.

The results in latency reported by Cyclictst in this test were similar to those in section 5.1.2. Regarding kernel preemptions, the hypothesis for this test was that the non-preemptible kernel would not detect any such preemptions. However, this was not the case, as seen in Figure 5.15, which shows the number of preemptions of tasks common to all test runs.

The non-preemptible kernel detected 37 preemptions of tasks in kernel mode. Although the preemptible kernel experienced a higher number of preemptions, a total of 50, the fact that the non-preemptible had any was unexpected. Any driver can push work to global work queues, meaning the nature of the preempted work is unknown. Further, this means that the preempted work could contain a *cond_resched()* call, making it preemptible, regardless of the preemption model. Another explanation could be that the recorded kernel preemptions on the non-preemptible kernel are false positives, either because of a misunderstanding of the ARM CPU mode macros or the implications of the trace point itself.

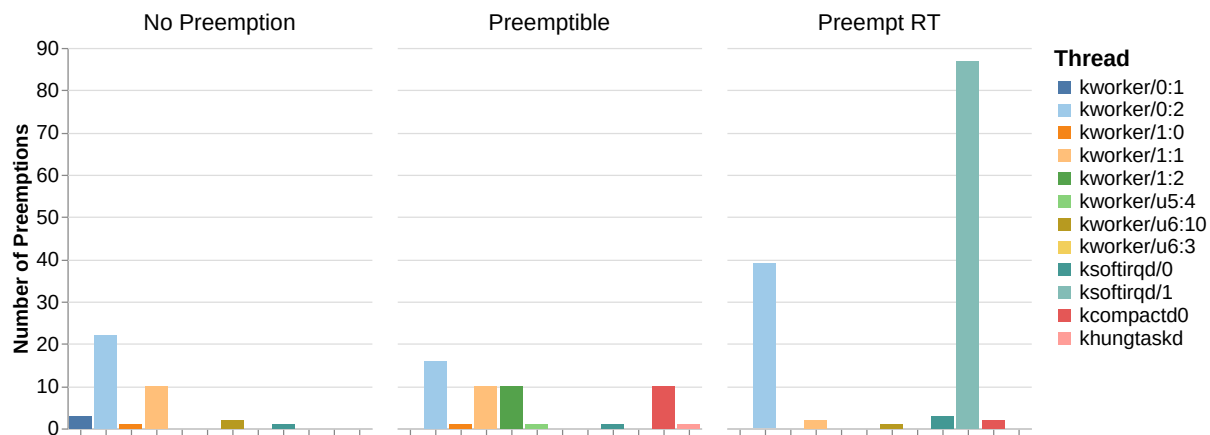


FIGURE 5.15: Preemptions of kernel tasks by Cyclicttest.

Based on the results, one can conclude that the significant difference between the latency of the non-preemptible kernel and the preemptible kernel is most likely due to its ability to preempt the work done by either the *kworker/1:2* or the *kcompact* daemon, as those make up the majority of the discrepancies.

The fully preemptible kernel had 134 preemptions, significantly higher than the other preemption models. The *ksoftirq* daemon was responsible for most of this discrepancy, which makes sense as `PREEMPT_RT` pushes all soft interrupts to the daemon [2]. Given that soft interrupts on the other preemption models are only pushed to the daemon when they occur too frequently, the likelihood of preempting the soft interrupt daemon is much higher on the fully preemptible kernel due to its increased workload.

The fully preemptible kernel has several kernel threads that do not exist in the other configurations. Figure 5.16 shows the total kernel preemption detected on the fully preemptible kernel. The staggering difference from the other preemption models illustrates how much extra overhead the fully preemptible kernel has and how throughput can degrade to the degree seen in the throughput test.

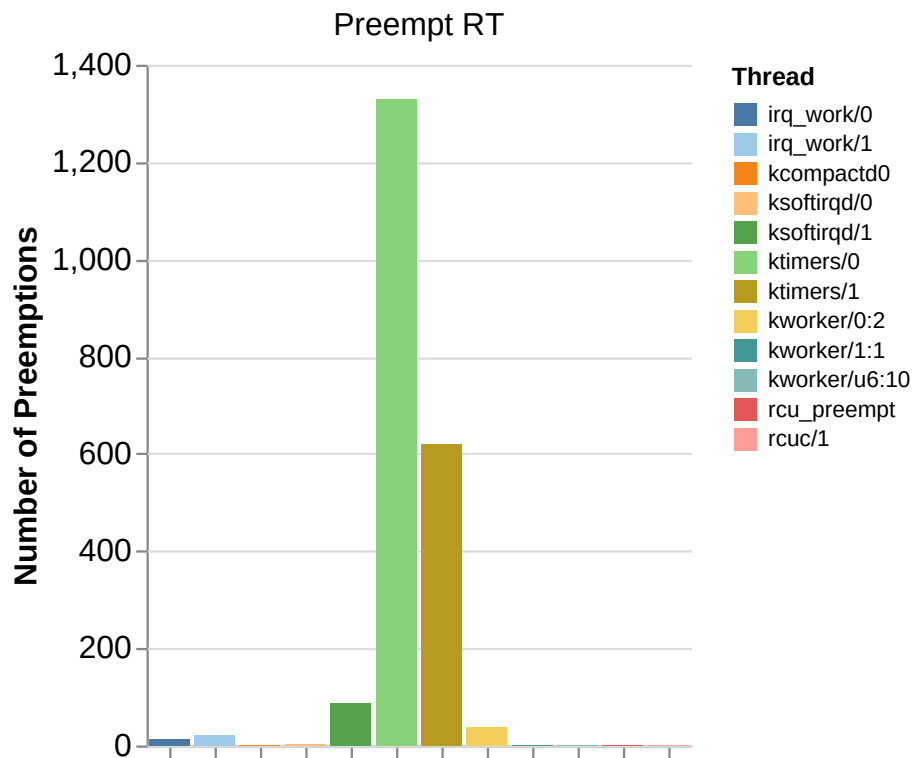


FIGURE 5.16: Preemptions of kernel tasks by Cyclicttest using PREEMPT_RT.

The test on the fully preemptible kernel also ran with the task at a priority of 60 to see how often it would preempt threaded interrupts. The task preempted the serial line, ethernet, and MMC interrupt threads 105 times in total.

5.1.5 Memory Lock Test

The first goal of the memory lock test is to determine what steps are required to avoid page faults at run-time. The result was that locking memory during initialization was sufficient to avoid page faults. Manual pre-faulting was not necessary for any of the types of memory. In addition, whether the memory was locked by the thread using the memory or if the memory was locked before thread initialization did not make a difference.

A second goal of this test was to see the impact of run-time page faults on latency, which is available in Figure 5.17.

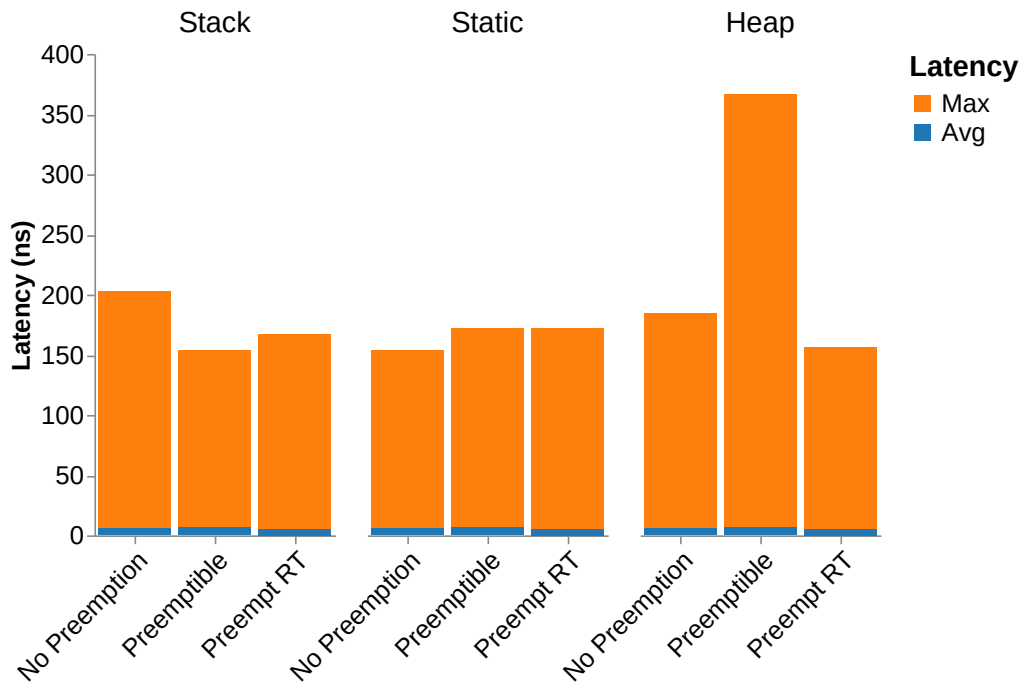


FIGURE 5.17: Latencies of soft page faults.

The results show that the latencies a soft page fault produces, on average, are minuscule. The max latencies are also relatively low. Although the preemptible kernel had a latency spike above the others when faulting on heap memory, all results are in the order of nanoseconds.

This indicates that avoiding soft page faults is not critical unless an application has extreme latency constraints or is accessing large amounts of memory consecutively.

5.1.6 Discussion

The platform baseline tests show that the mainline Linux kernel can provide low-latency performance in many cases simply by applying real-time scheduling policies. The results show that the nature of the system's real-time and background workload determines the need for and consequences of using a fully preemptible kernel.

An example of this was seen in the Cyclicttest results, where the latency impact of Iperf was much less significant on the non-preemptible kernel than the impact of kmemleak. However, while the preemptible kernel could mitigate most of the effects of kmemleak, almost to the extent of the fully preemptible kernel, it could not reduce the latencies brought on by Iperf to the same degree.

The throughput tests also showed that the fully preemptible kernel is not always the best choice, as it can lead to significant throughput degradation in system-call-heavy workloads. It is unclear which system calls are most affected by this, and further experiments should be conducted if applicable.

The preemption test showed that tasks on a fully preemptible kernel are preempted more often than on other preemption models, which was expected due to threaded

interrupts. The difference becomes more pronounced when using real-time scheduling policies.

Although not surprising, utilizing eBPF to verify that the fully preemptible kernel is preemptible in kernel mode was also an intriguing activity. This test also showed examples of a non-preemptible kernel preempted by tasks executing in user mode, which was unexpected.

Lastly, the results showed that soft page faults are not a significant source of latency, but avoiding them through locking memory can be beneficial in some cases.

5.2 Target System Tests

This section presents and discusses the results found when executing the tests designed to evaluate the need and benefits of using mechanisms to increase the real-time capabilities of the target system.

5.2.1 Dynamic Memory Allocation

The execution time of dynamic memory allocation through calls to `nlmsg_alloc()` can be seen in Figure 5.18. The subfigures show the results of the tests executed under different loads. Within each load scenario, the average execution times are identical. The maximum execution times are more challenging to interpret. Locking and pre-faulting the heap memory does not significantly impact the execution times. There are massive spikes in some test scenarios, which are inconsistent across the different load scenarios. One would assume the sample size is sufficient to produce consistent results, but this may not be true.

The kernel's preemption model also exhibits no pattern in the results, making it difficult to conclude how the preemption model impacts the execution times.

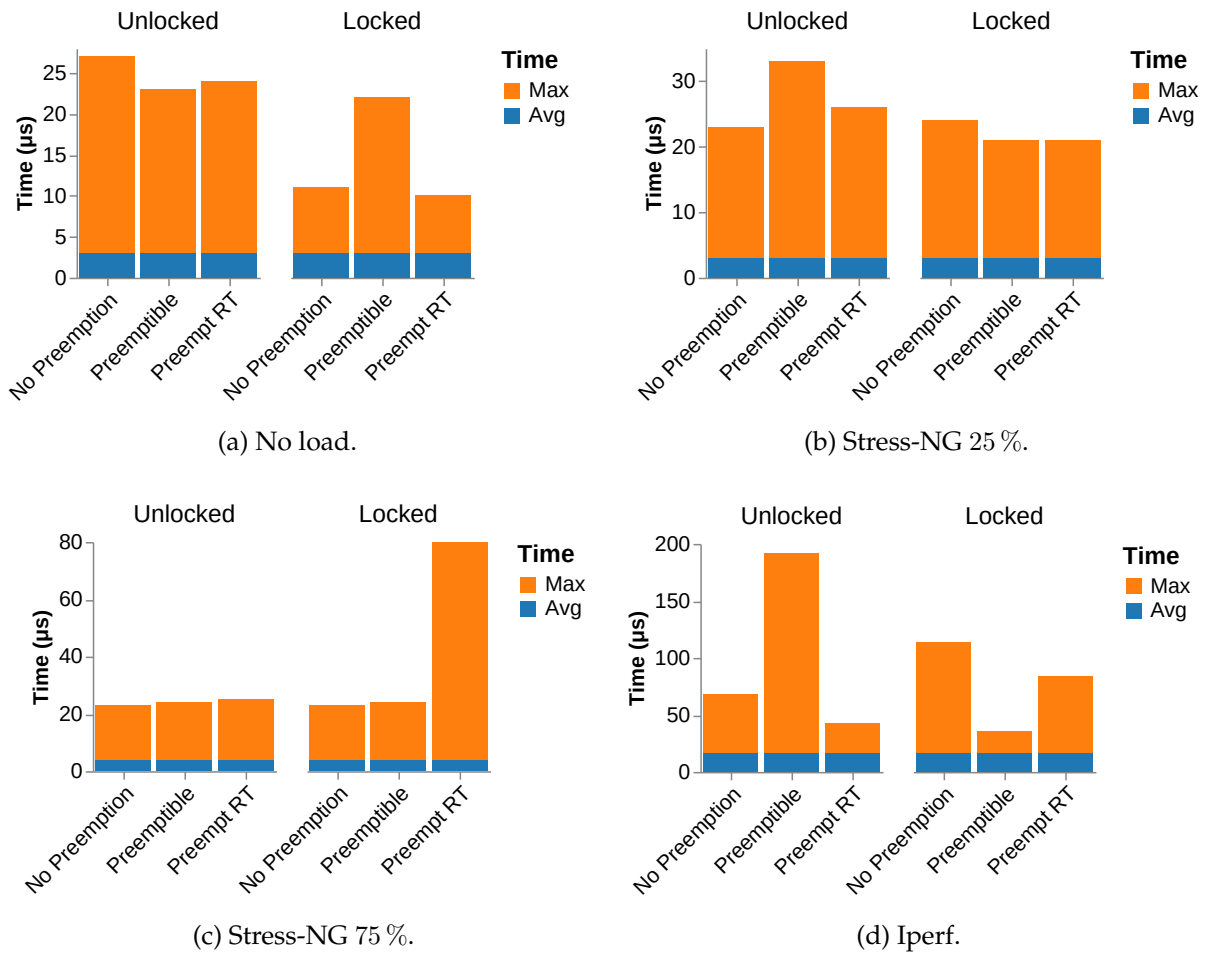


FIGURE 5.18: Dynamic memory allocation results.

The execution time results for when touching and freeing the dynamically allocated memory (using `nlmsg_free`) are available in Appendix C, Figures C.2, and C.3. These results were similarly inconclusive, with no easily identifiable pattern. The only distinctive result across all three operations is that the fully preemptible kernel experienced a latency spike when under 75 % load. The reasons for this remain unclear.

To gain a different perspective, Figure 5.19 looks past the respective test configurations and only shows the worst measured case for each operation.

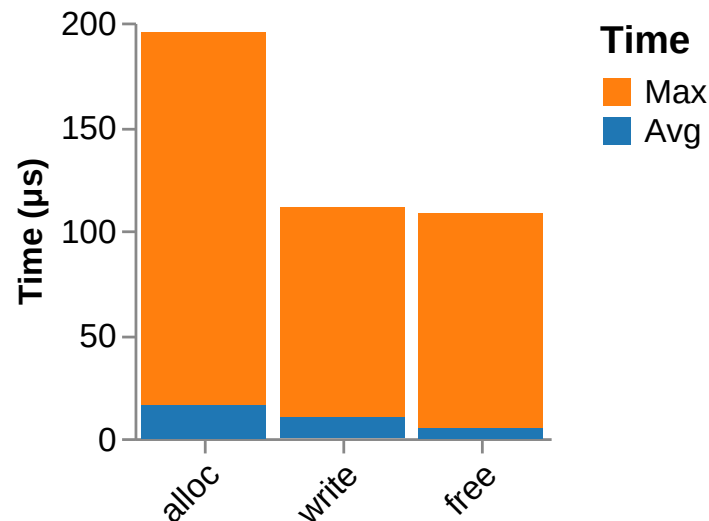


FIGURE 5.19: Worst case latencies per dynamic memory operation.

Here, it becomes clear that the average execution times of these operations are minor and should not negatively impact the target system. Spikes of up to 200 μs were seen during allocation when stressed under Iperf, which could be a concern. During touching and freeing memory, spikes of around 100 μs were also observed.

Although unlikely, if a task with a period of 1 ms were to experience these spikes simultaneously, that would eat up almost 50 % of the time budget. This aligns with accepted knowledge and recommendations from the literature that dynamic memory allocation should be avoided in real-time systems, regardless of locking and pre-faulting.

5.2.2 Periodic Execution Test

The results of the periodic execution test are illustrated in Figure 5.20, which illustrates the average latency experienced across all mechanisms, with a 25 % load induced by Stress-NG. Two schemes stand out negatively: the *setitimer* tests, executed with SCHED_OTHER on either the main thread or the signal. Still, the average latency is very low for all mechanisms. The highest being around 12 μs .

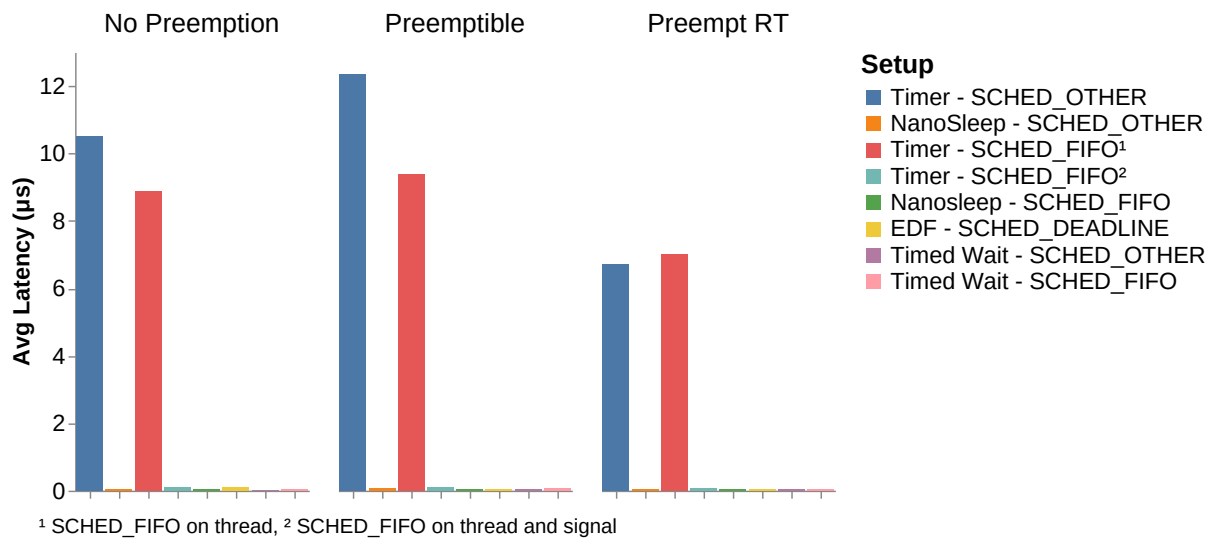


FIGURE 5.20: Avg latency of periodic execution test, at 25 % load.

Similarly to the findings when using *Cyclictest*, *setitimer* scheduled with `SCHED_OTHER` perform better under `PREEMPT_RT`. This result is somewhat puzzling because a `SCHED_OTHER` scheduled task benefits nothing from running with `PREEMPT_RT`. Instead, it should perform worse due to the reduced throughput. This observation is more a curiosity than a helpful finding.

The maximum latencies of the tests, illustrated in Figure 5.21, tell a different story. All mechanisms using `SCHED_OTHER` perform significantly worse, as is to be expected. It is somewhat interesting that for *setitimer*, it is not sufficient to set a real-time scheduling policy on the thread. According to [113], signals route to a random thread within the process that registered it unless configured otherwise. This explains why it is also necessary to schedule the main thread, which initially registered the signal handler, under a real-time policy to achieve low latencies.

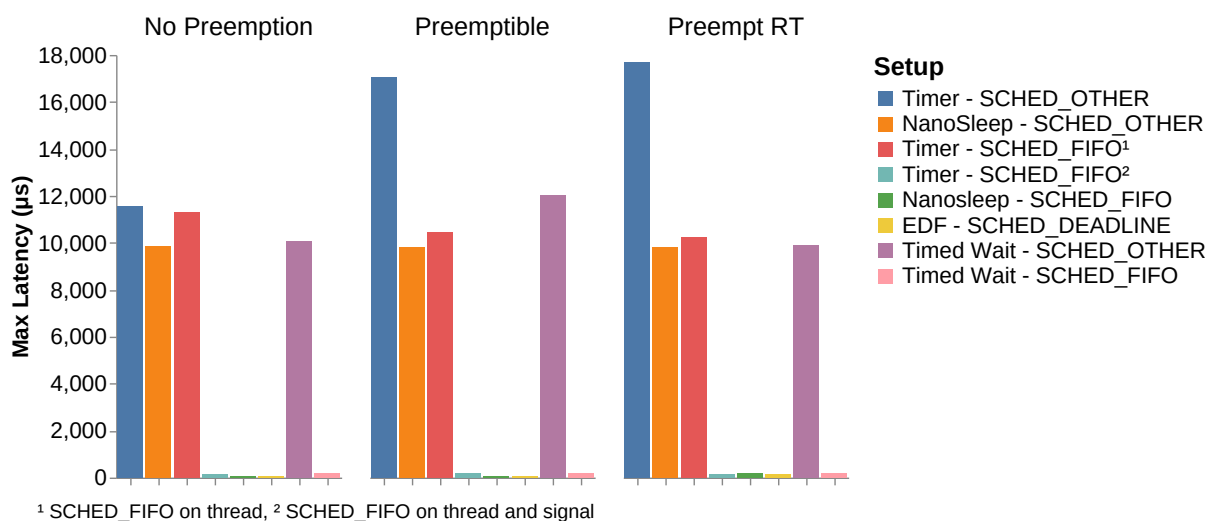


FIGURE 5.21: Max latency of periodic execution test, at 25 % load.

The reason for the low average latencies, compared to the sometimes very high maximum latencies, is the implementation of the latency calculations. If only testing `nanosleep`, it would be natural to calculate the latency as the time between when the task explicitly asks to be awoken and when it awakes. If the task misses entire periods, it is natural to skip them so that the latencies will not propagate to the measurement in the next period. However, when using a mechanism like `setitimer`, the thread has no control over when the signal will trigger. Additionally, there can be a high latency between the signal handler triggering the semaphore and the main thread waking up. This means that inside the main thread, it is possible to experience negative latencies even though the signal will never trigger early [114]. To ensure consistent results across all mechanisms, the latency is calculated as the time between each entry to the periodic thread. This means all mechanisms can experience negative latencies, leading to low average latencies.

This is illustrated in Figure 5.22, which shows the latency distribution for `setitimer` with `SCHED_OTHER` between $[-2500\mu\text{s}, 2500\mu\text{s}]$.

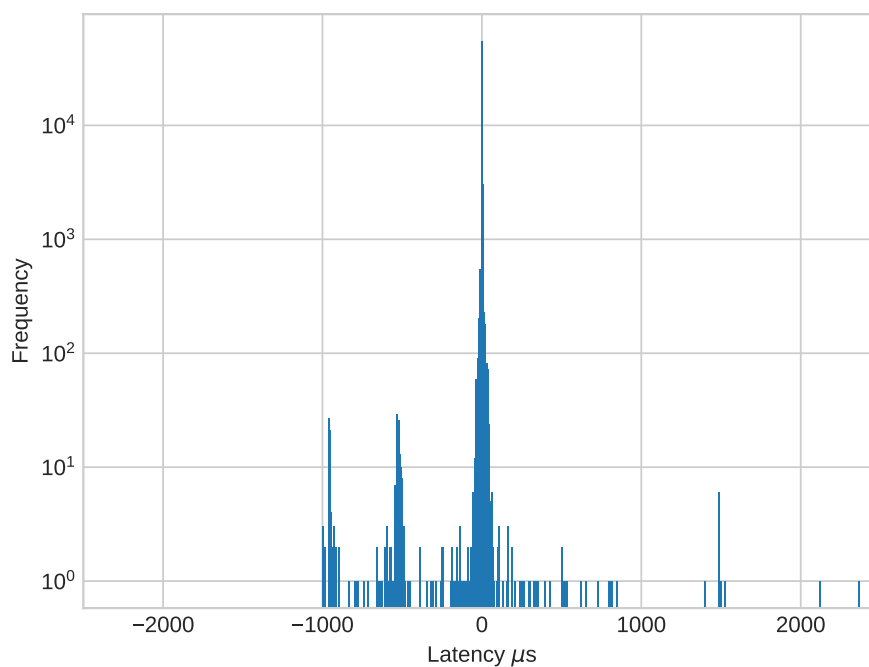


FIGURE 5.22: Latency distribution of `setitimer` with `SCHED_OTHER` and 25% load.

The distribution is similar for all mechanisms when scheduled under `SCHED_OTHER`. This can be seen in Figure C.4, in Appendix C. Figure 5.23 shows the latency distribution of mechanisms under `SCHED_FIFO`.

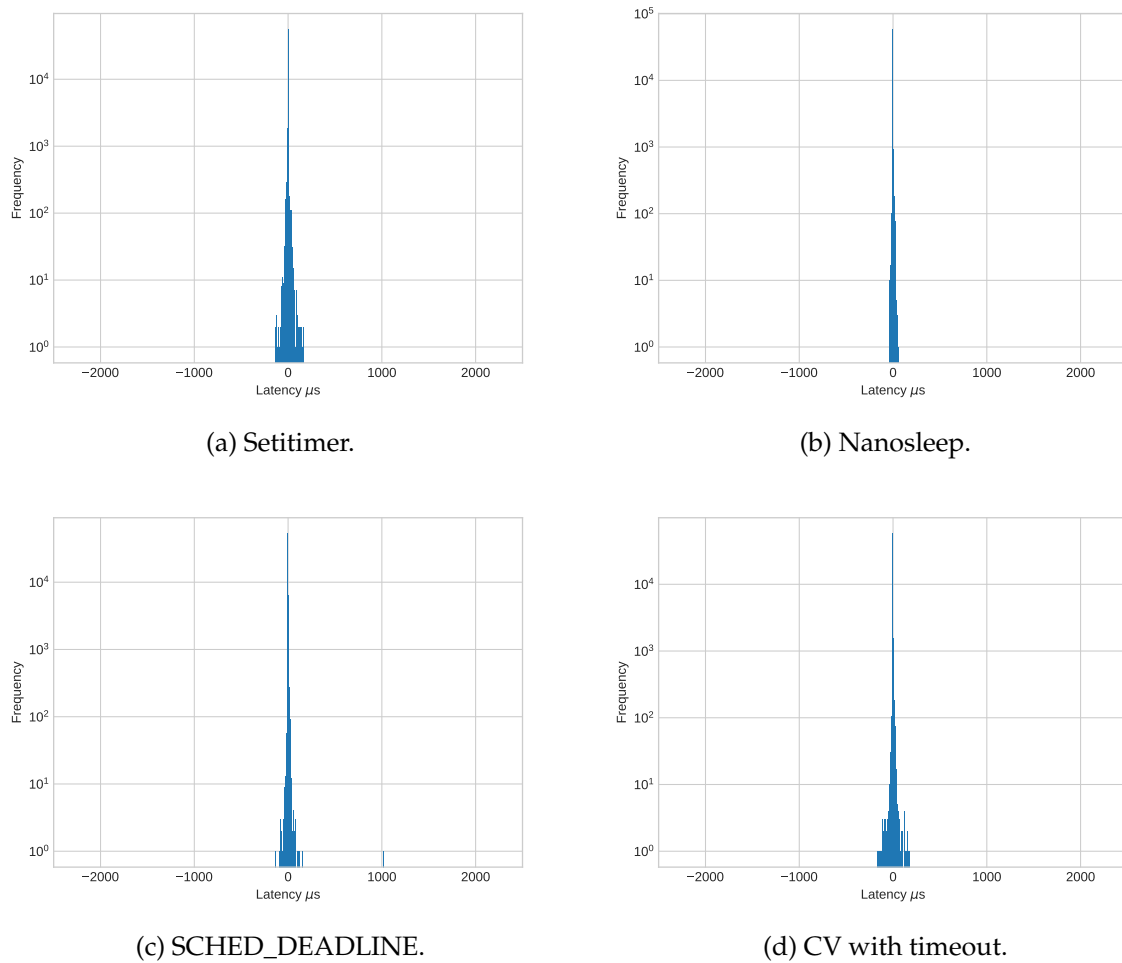


FIGURE 5.23: Latency distribution of periodic mechanisms with SCHED_FIFO and 25 % load.

These results show that based on latency alone, the nanosleep mechanism scheduled under SCHED_FIFO is the best choice, in accordance with the literature's recommendations. They also show that the existing *setitimer* solution can be improved dramatically by using real-time scheduling policies.

Figure 5.24 shows the average CPU load of the different mechanisms.

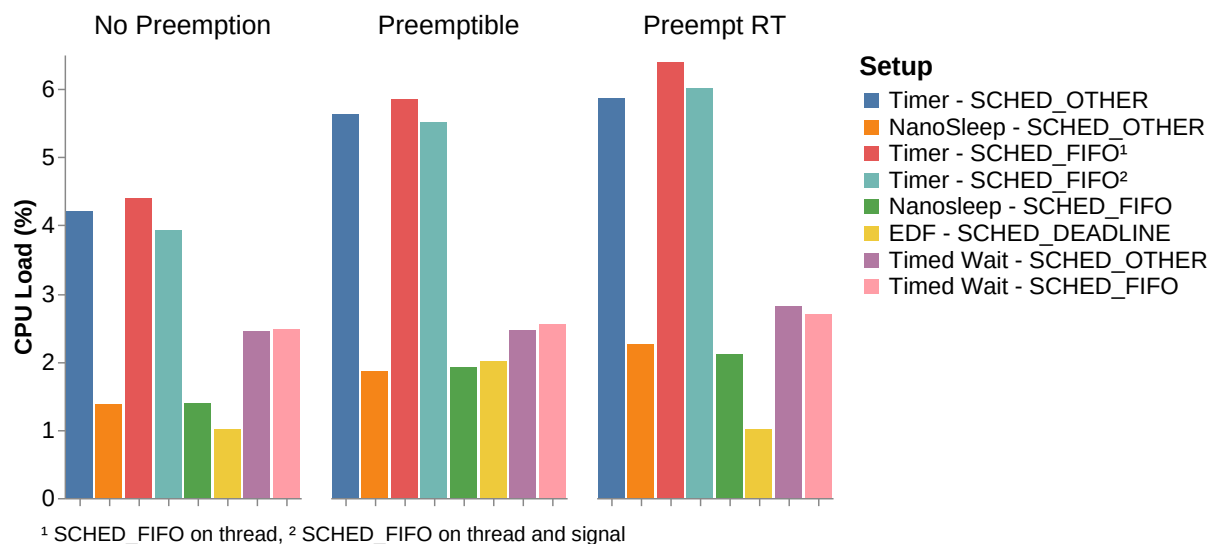


FIGURE 5.24: Average CPU load of periodic execution mechanisms.

Unsurprisingly, the extra overhead of *setitimer* leads to a higher CPU load. The condition variables with a timeout improve the CPU load by about 50%, while the nanosleep improves it further. SCHED_DEADLINE required the lowest overhead, which aligns with the fact that the kernel contains all the logic required to drive it periodically, not the application.

Based on this, one can conclude that condition variables with a timeout, nanosleep, and deadline scheduling are the best mechanisms for periodic execution. If the application must support waking on events while blocking, a condition variable with timeout should be selected. If not, nanosleep should be preferred over SCHED_DEADLINE, as it is easier to set up, provides more flexibility, and has a slightly lower latency.

The test was also executed with 75% load, and they were also done on a fully preemptible kernel. This gave similar results, and is illustrated in Appendix C, Figures C.5, C.6, C.7 C.8, C.9 and C.10.

5.2.3 Shared Memory Test

The initial shared memory test read 4 KiB chunks from the circular buffer on every update. The periodic scheme executes once every millisecond, meaning that the thread sleeps for a millisecond minus the execution time every period. In contrast, the aperiodic scheme keeps executing while there is still data to read before sleeping for a whole millisecond.

Based on the average CPU load measured during the test in Figure 5.25, it is clear that the scheme did not significantly impact the CPU load. This indicates that the aperiodic task also yields the CPU sufficiently.

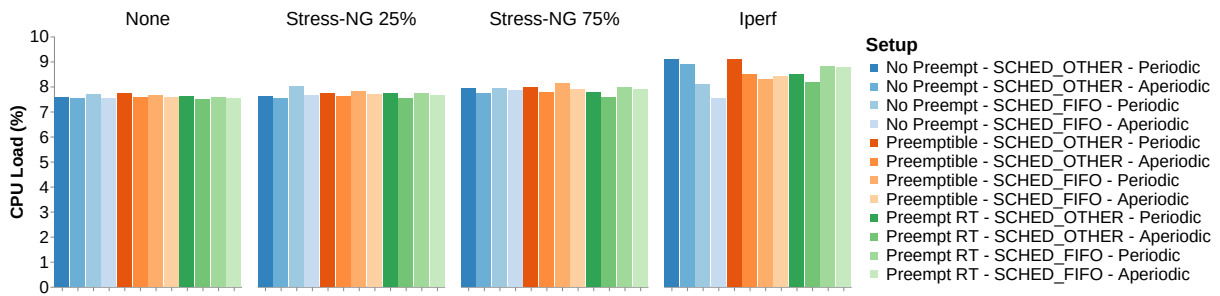


FIGURE 5.25: Shared memory test, average CPU load.

A concern with using SCHED_OTHER with the aperiodic scheme was that the task would be heavily preempted involuntarily after continuously exhausting its timeslice. Figure 5.26 shows this concern was unwarranted.

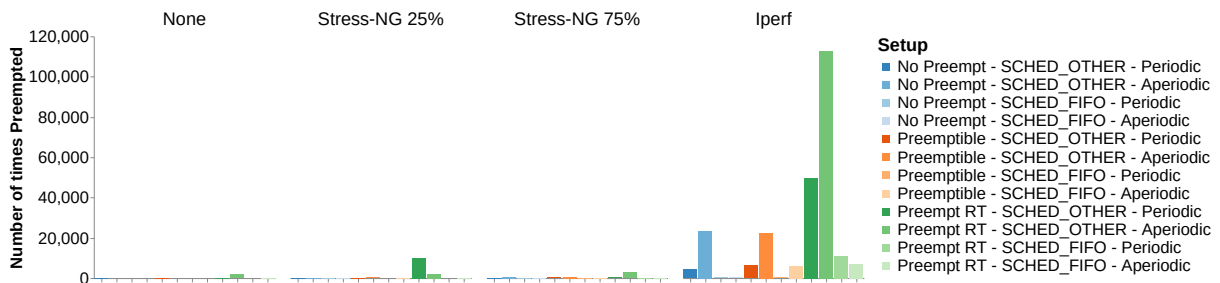


FIGURE 5.26: Shared memory test, preemption count.

The only scenario that showed a significant number of preemptions was when Iperf induced stress on a fully preemptible kernel. This pattern was observed during the preemption test with the CPU-intensive task, indicating that the high number of preemptions on SCHED_OTHER is due to threaded interrupts and that preemptions occurring when using real-time policies are due to the migration task. Additionally, under SCHED_OTHER, the aperiodic scheme was preempted more than twice the amount of the periodic scheme.

The primary measure of success for this test is whether or not the reader can keep up with the rate of the producer. Throughout the test, the software kept track of the maximum reached buffer size, and the results can be seen in Figure 5.27.

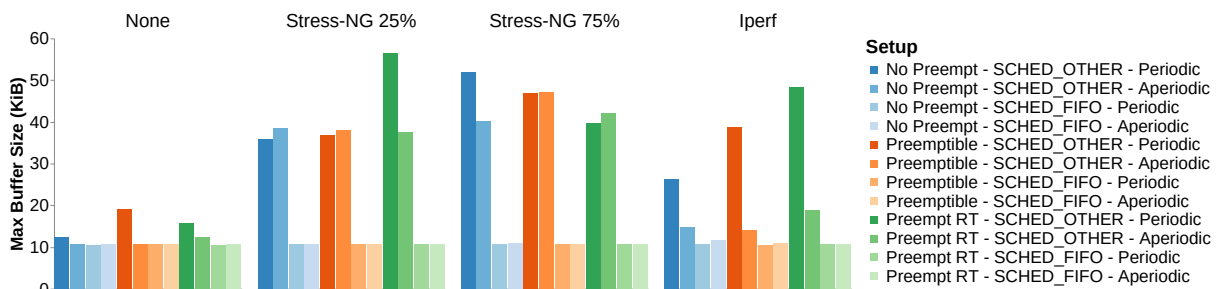


FIGURE 5.27: Shared memory test, max buffer size

A buffer overflow occurs if the size reaches 128 KiB, but all setups could keep up with the producer by a wide margin. The worst measured size was just below 60 KiB. Still, the benefit of using a real-time scheduling policy is clear, as the buffer size is consistent across all stressors and preemption models. As expected, the aperiodic scheme performed better than the periodic scheme under SCHED_OTHER.

The data recorded during the tests indicate that while everything measured is connected somehow, the latency is the most significant factor in determining how high the buffer size will get. Figure 5.28 shows the correlation between the maximum buffer size and the wake-up latency of the reader thread for all test configurations.

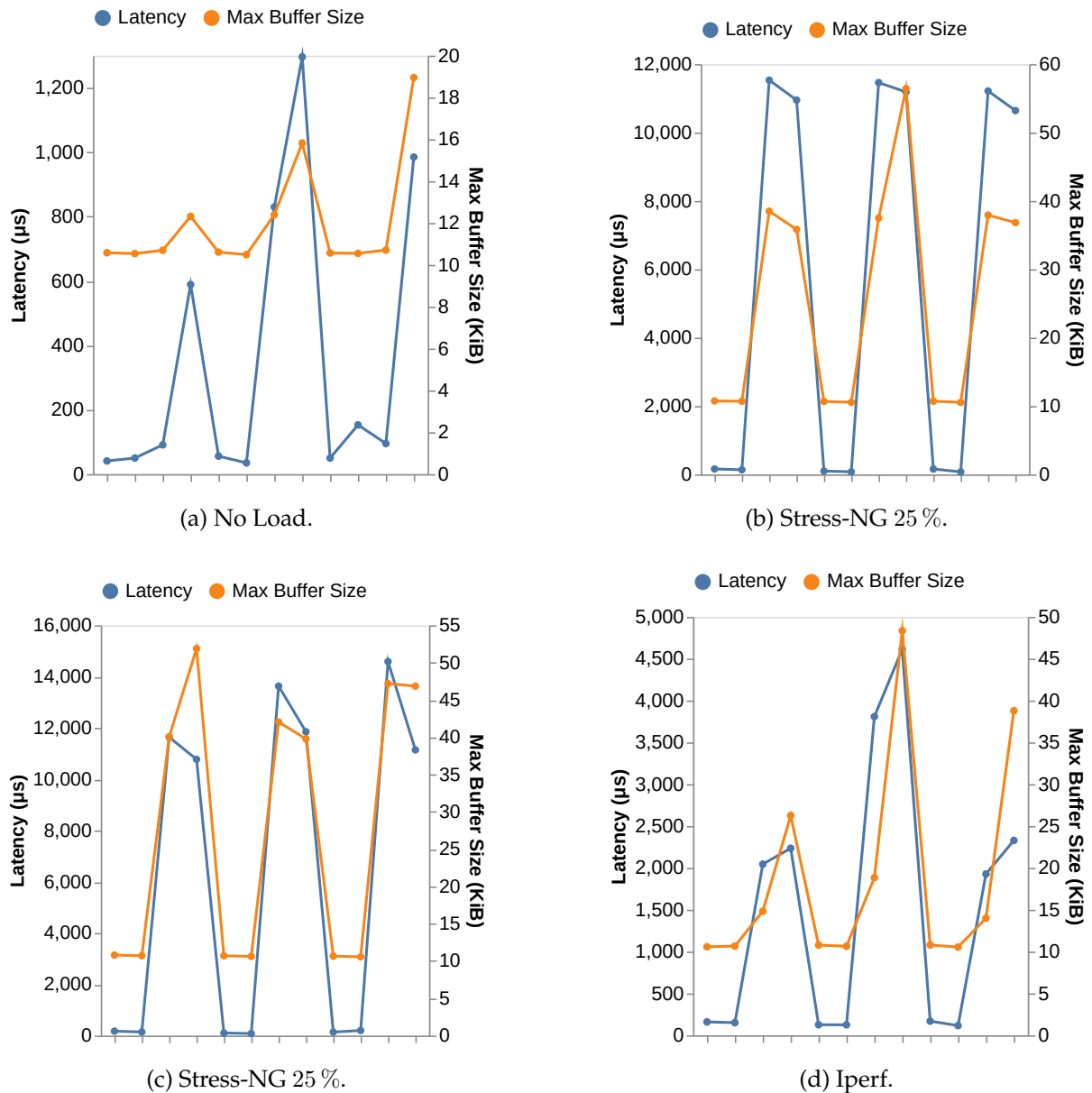


FIGURE 5.28: Correlation between latency and max buffer size for each shared memory test configuration.

512 Byte Packets

To increase the burden of the application, the simulated overhead per packet remained the same while testing with 512-byte packages. This led to an 8x increase in overhead compared to the previous setup. The periodic task would not be able to keep up with the producer using this package size if it were only to read one package per period, as it would take

$$\frac{128 \text{ KiB} - 4 \text{ KiB}}{512 \text{ B}} = 248 \text{ cycles} \quad (5.1)$$

to clear a buffer.

As discussed in section 4.4.5, the buffer fill rate is every 50 μs . Instead of increasing the rate of the periodic task, the task was modified to clear the buffer on each period.

Figure 5.29 shows that the CPU load was, also in this configuration, stable across all scenarios but had increased to around 20% on average. This means the decreased package size requires twice the CPU time to handle.

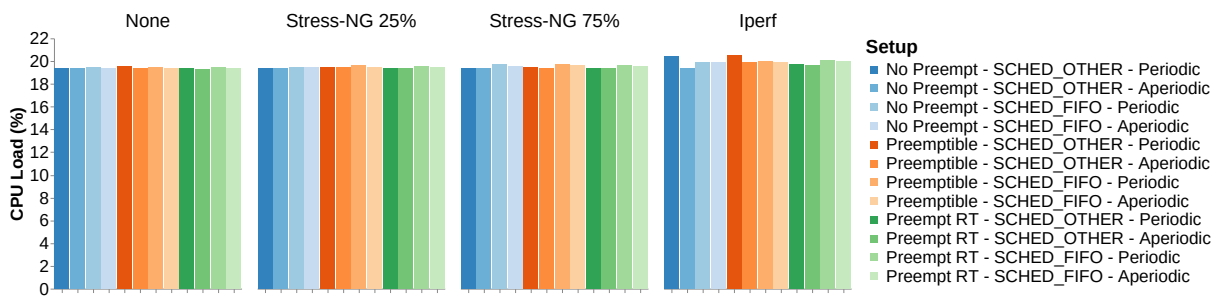


FIGURE 5.29: Shared memory test, 512-byte packets, average CPU Load.

The number of involuntary preemptions remained low for most tests but increased dramatically for the non-preemptible kernel when stressed with Iperf. Interestingly, this did not affect the execution time of the task, which in the worst case was lower than when stressed with Stress-NG at 75% load.

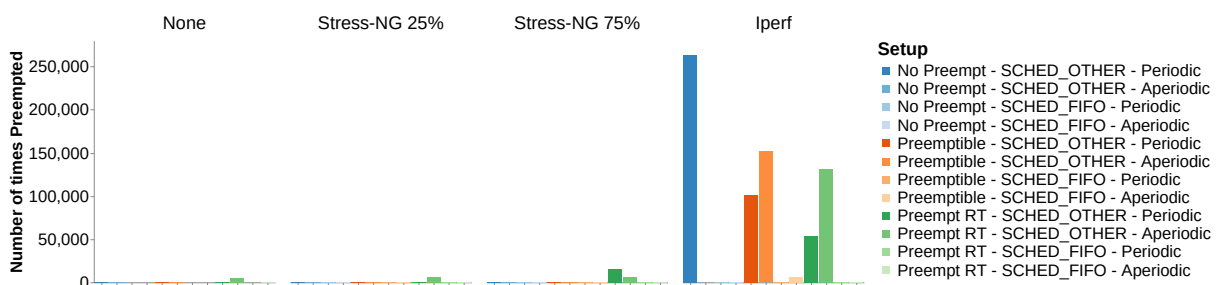


FIGURE 5.30: Shared memory test, 512-byte packets, preemption count.

Also, when looking at the maximum buffer size recorded during the tests in Figure 5.31, high preemption counts did not impact the results significantly. Other than

that, the max buffer sizes were consistent with the 4 KiB tests. However, the well-performing configurations were able to keep the buffer size lower, at around 6 KiB to 7 KiB. This makes sense for the periodic task, as it now clears the buffer on each period, but the aperiodic scheme also performed better. Of course, they both use more CPU time, and the smaller packages mean that the producer reads the buffer more often, and the reader is more likely to keep up with the producer.

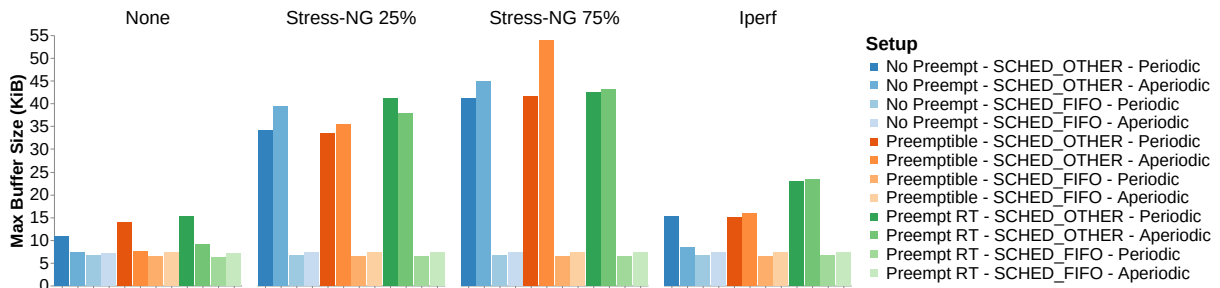


FIGURE 5.31: Shared memory test, 512-byte packets, max buffer size.

The execution times of the tasks under SCHED_OTHER went entirely off the charts in some setups, as expected. However, if focusing only on the results of the SCHED_FIFO tasks, the execution times are all lower than 1 ms. This is illustrated in Figure 5.32, which shows the execution times when stressed with Stress-NG at 75 % load.

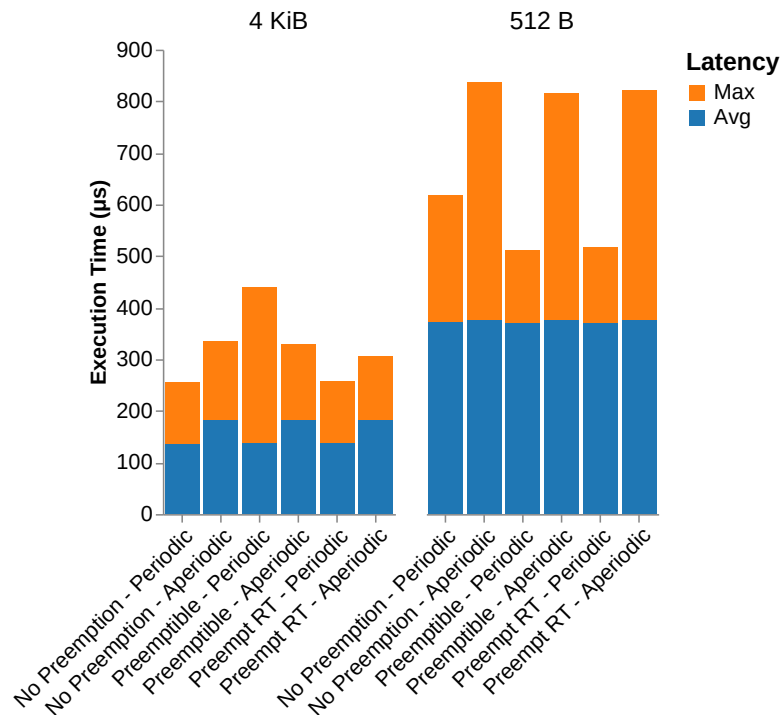


FIGURE 5.32: Shared memory test, execution times compared under SCHED_FIFO with Stress-NG 75 %.

In every comparison between the periodic and aperiodic schemes except one, the peak execution time of the aperiodic scheme was higher, as expected. However, on average, they behaved similarly.

The latencies recorded during the SCHED_FIFO-based tests with Iperf as a stressor can be seen in Figure 5.33.

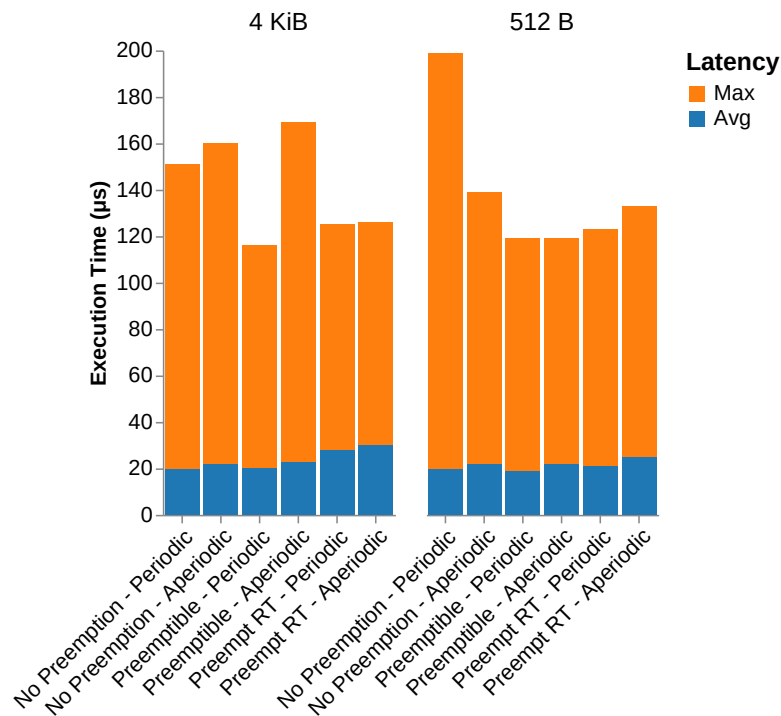


FIGURE 5.33: Shared memory test, latencies compared under SCHED_FIFO and Iperf load.

As all SCHED_FIFO-based tests could keep up with the producer, it is unsurprising that all the latencies are low. What is more surprising is that even though the execution time is longer, the latency spikes experienced when running Cyclictest on a non-preemptible kernel with Iperf did not manifest in this test. The explanation is most likely that Cyclictest had a period of 400 µs, while the shared memory test had a period of 1 ms. Still, the other latencies found in this test are comparable with the Cyclictest results.

This test only covered *reading* from the shared memory area. Brief tests of writing to the buffer were also conducted, but the results were similar to the read tests and are not included in this thesis. The Full Test covers Transmission mechanisms well, as discussed in section 5.2.7.

5.2.4 Generic Netlink Test

This test was performed with the client application scheduled under SCHED_OTHER and SCHED_FIFO. However, since the task was heavily preempted under SCHED_OTHER, it only makes sense to compare the results found under SCHED_FIFO policies.

The test results can be seen in Figure 5.34, which illustrates the execution time of requesting, reading, and writing data separately. Figure 5.34(d) shows accumulated execution time for all operations.

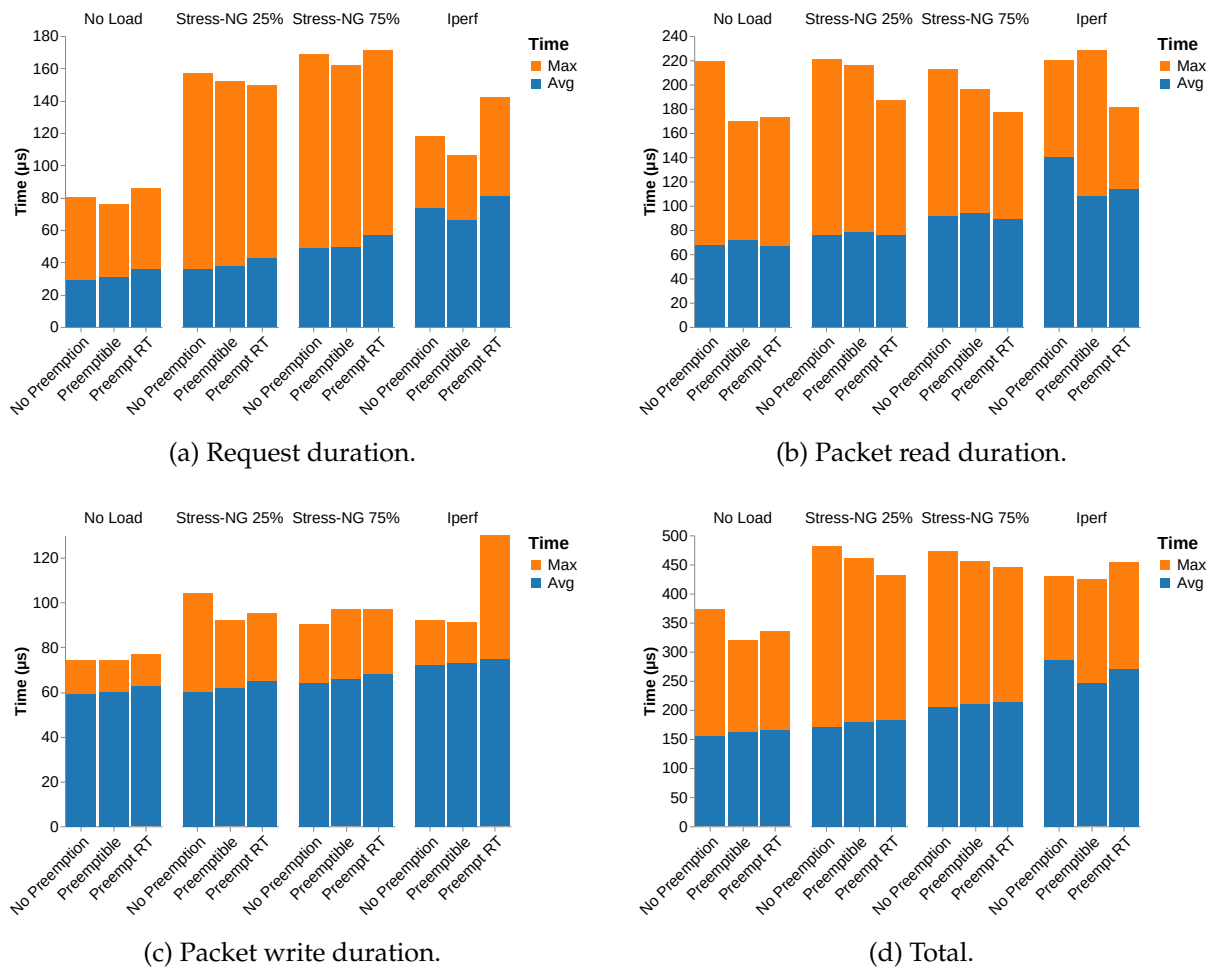


FIGURE 5.34: Execution time of kernel thread communication over Generic Netlink.

Under no load and load induced by Stress-NG (i.e., in user space), the average execution time increases slightly as the kernel becomes more preemptible. Simultaneously, the maximum execution time decreases. However, the differences are in the order of 10s of microseconds, meaning that the choice of preemption model is not critical.

The results of the tests with Iperf as a stressor stand out. Here, the fully preemptible kernel performs worse during a transmission but better during a reception. This leads to the total average execution time being lower on the preemptible kernels than on the non-preemptible kernel when stressed by Iperf. Since Iperf executes in kernel space for the most part, this aligns with earlier findings in this section.

These results show that the choice of the preemption model is not critical for the performance of the Generic Netlink communication and should not introduce any significant issues for the target system.

When comparing these numbers with the dynamic memory allocation test results, one can assume that a big part of the execution times is due to the dynamic memory allocation that occurs as part of these operations. Some efforts were made to avoid or limit dynamic memory allocation when using *Libnl*, but no workable solution was discovered.

Although the baseline throughput test showed that the preemption model could drastically reduce throughput in system-call-heavy scenarios, this test did not exhibit this behavior. Based on this, one can conclude that the mechanisms used in Generic Netlink communication towards the kernel modules are not susceptible to the potential degradations introduced by a more preemptible kernel.

5.2.5 Kernel Thread Event Handling

Figure 5.35 compares the CPU load of an event-driven kernel worker and a polling worker. As expected, they show that the event-driven kernel worker has a lower impact on the CPU load. The improvement was predictable in an idle system, as the kworker will indefinitely go to a blocking state when using an event-driven scheme. However, the degree of the improvement seen at event rates up to 1 kHz was higher than expected, but it makes sense given the rapid poll rate of [10, 100] ms.

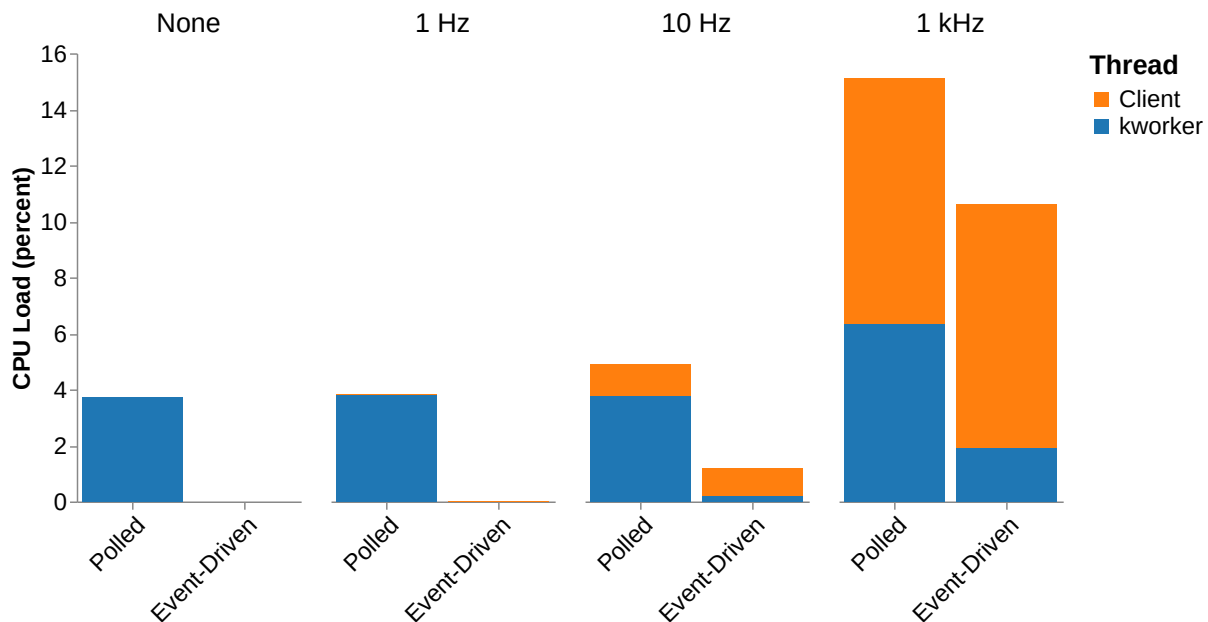


FIGURE 5.35: Polled vs. event-Driven kernel worker, at different input event frequencies.

An event rate of 10 kHz was also tested and showed the polling scheme to perform better. This can be seen in Appendix C, Tables C.23 and C.24. This shows that as the event rate increases to a point close enough to the polling rate, the thread no longer performs redundant work, as it will always find a new event to handle on each poll. At 10 kHz, the reduced overhead of the polling scheme increases the efficiency.

However, this rate of events is unrealistic for the target system. It is expected to receive events at a rate significantly lower than 1 kHz, which means that the event-driven scheme is the best choice for the target system. One could argue that reducing the polling rate could have the same effect, but that would result in a less responsive worker and increase the latency between asking for data and receiving it.

5.2.6 Full Test RX

The first thing to note about the full reception pipeline test results is that the CPU load was virtually the same for all permutations, hovering just around 23%. However, the periodic scheme consistently used about 0.5% to 1% less CPU than the aperiodic scheme.

As the selected package size for these tests was 1504 B, the periodic scheme was modified to read three packages on each update to allow it to keep up with the producer without modifying the period. The average execution time of the thread between blocking is illustrated in Figure 5.36.

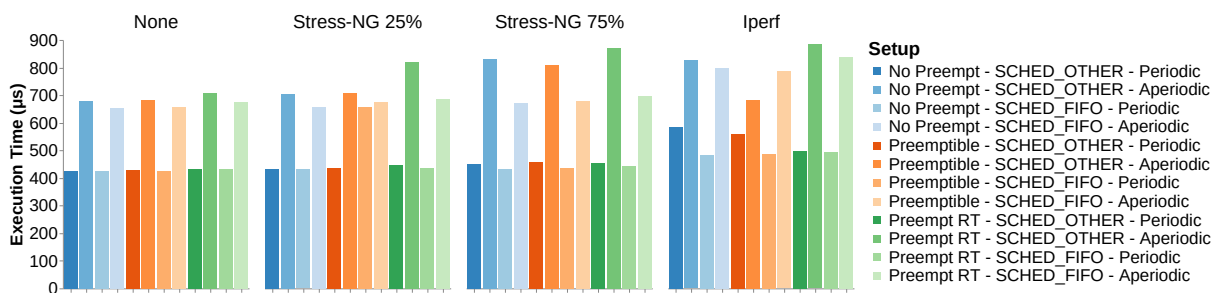


FIGURE 5.36: Average execution time of full RX pipeline.

The stressors and preemption model hardly affected the average execution time, but the periodic scheme had a significantly lower execution time than the aperiodic scheme. However, their average CPU time is almost identical since the aperiodic scheme blocks for a whole millisecond.

The maximum execution time measured during the tests can be seen in Figure 5.37. When inducing load on the system and the kernel schedules the application under SCHED_OTHER, the maximum execution time of the aperiodic scheme increases dramatically. The rate at which the task experiences involuntary preemptions grows accordingly, further underlining the non-deterministic nature of this approach.

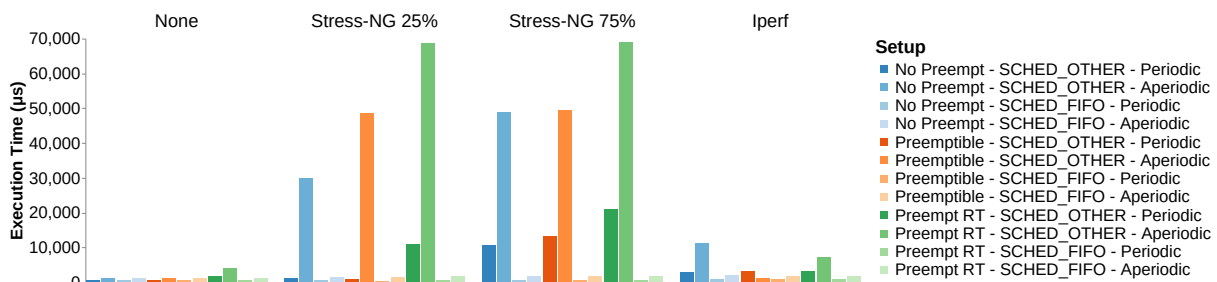


FIGURE 5.37: Maximum execution time of full RX pipeline.

To more clearly examine the difference in maximum execution times between the tests using real-time scheduling priorities, Figure 5.38 shows these results in isolation.

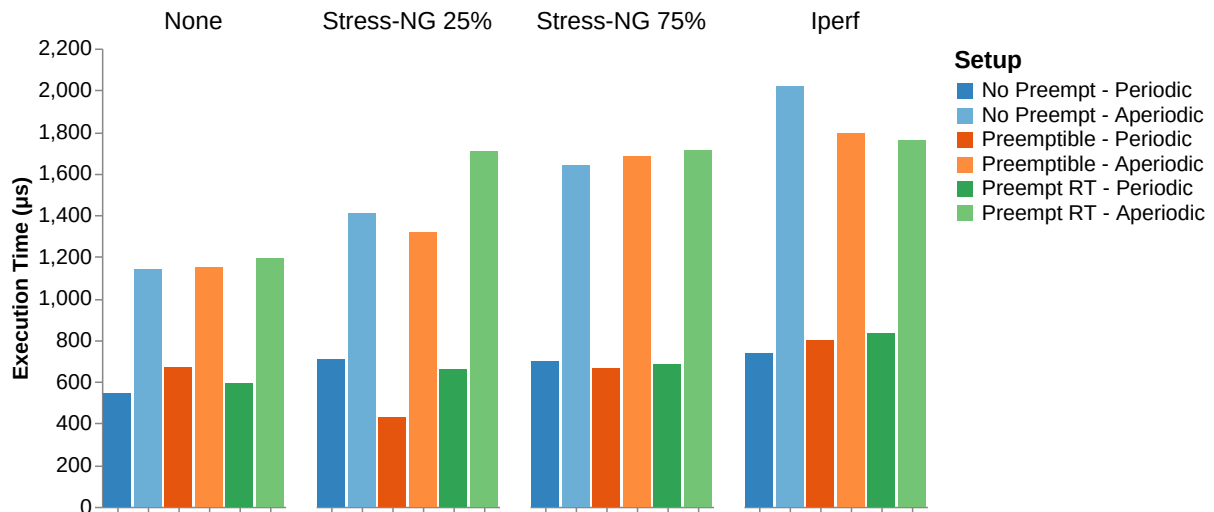


FIGURE 5.38: Maximum execution time of full RX pipeline under SCHED_FIFO.

These results show that while the aperiodic scheme does not have extreme spikes in execution times when scheduled under SCHED_FIFO, the periodic scheme is much more stable. The periodic scheme is not as susceptible to an external load and, in all cases, manages to keep the maximum execution time below the period of 1 ms.

As for the shared memory test, the most crucial measure is again the maximum buffer size reached during the test. This is illustrated in Figure 5.39.

Again, all setups could keep up with the producer, and none of them was close to the 128 KiB limit, even under load. However, the positive impact of using a real-time scheduling policy remains clear.

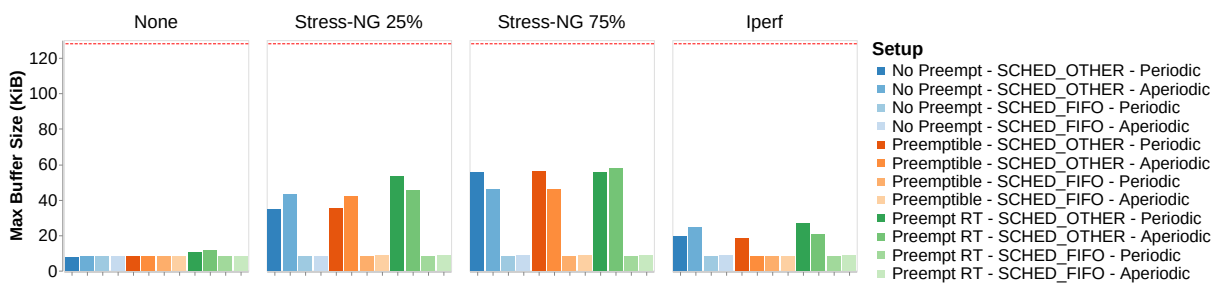


FIGURE 5.39: Maximum buffer size of full RX pipeline.

In order to make the differences between the system configurations more apparent, an additional test was conducted, which pushed the system further. An additional 200-microsecond busy wait was added after each read to simulate more work the application does during decoding. The periodic scheme was modified to read only two packages on each update to allow it to maintain its period.

This modification naturally impacted the CPU load, now around 40% on average for all configurations. This means that the main thread is using 80% of a single core. As this is just below the threshold for real-time throttling, this is at the limit of what should be considered acceptable for a thread scheduled under real-time policies.

The maximum buffer size reached during this test can be seen in Figure 5.40, with no load, Stress-NG at 25 % load, and Iperf, respectively.

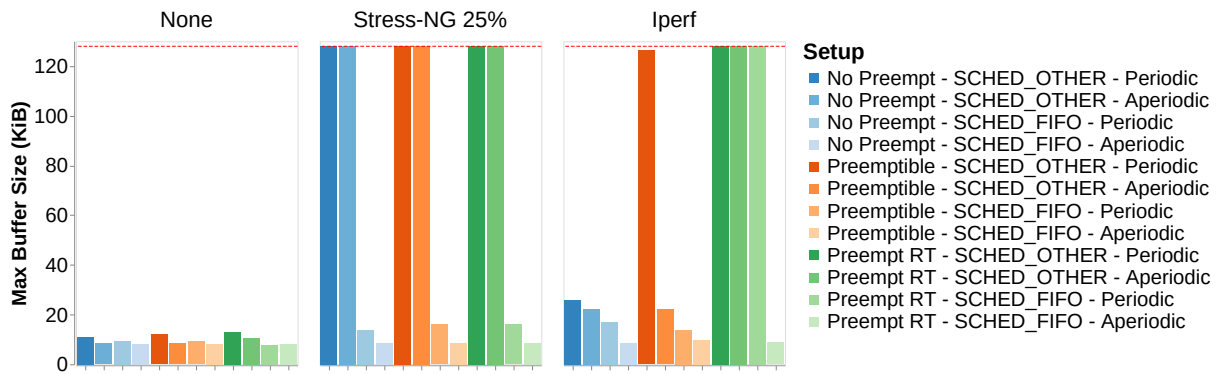


FIGURE 5.40: Maximum buffer size of full RX pipeline, increased overhead.

As with the other configurations, when no load is applied, all tests perform equally well. However, Stress-NG induced load causes the tests scheduled under CFS to fail. The aperiodic scheme performs slightly better than the periodic scheme under SCHED_FIFO, potentially explained by the fact that the periodic scheme now reads only two packages on each update. When Iperf induced the load, the non-preemptible kernel could keep up with the FPGA producer even under SCHED_OTHER. This was not the case for the other preemption models, which could be due to their increased overhead. The fact that this setup experienced the highest recorded latency of all tests, as seen in Figure 5.41, could indicate that the increased overhead of system calls used by Iperf is significant on preemptible kernels.

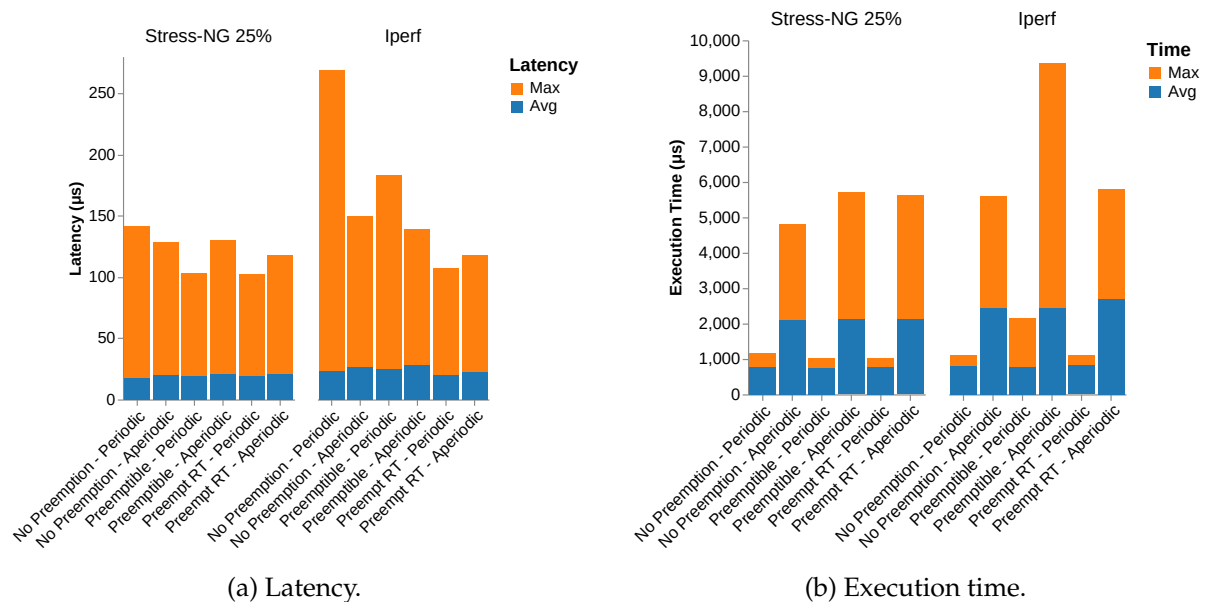


FIGURE 5.41: Temporal data for full RX pipeline with increased overhead.

Interestingly, the periodic scheme on a fully preemptible kernel was the only setup under SCHED_FIFO that could not keep up with the producer. This temporal data shows that the periodic scheme under SCHED_FIFO on a fully preemptible kernel is

among the setups with the lowest latencies and execution times. Although the fully preemptible kernel has more overhead, according to the temporal data, the task never missed its period to the degree that can explain a buffer overrun as experienced in the test.

However, the periodic scheme now strictly reads up to two packages of 1504 B on each update. This means that it will clear the buffer in

$$\frac{128 \text{ KiB} - 4 \text{ KiB}}{1504 \text{ B} \times 2} = 41.3 \text{ cycles} \quad (5.2)$$

As the period is 1 ms, and the buffer fills every 50 milliseconds, the margins are slimmer than in previous setups. This means that consecutive high latencies could cause the overruns recorded during the test.

Again, this test shows that the real-time scheduling policies have more impact than the preemption model. It is also clear that when pushing the system to its limits, the periodic scheme requires more tuning to keep up with the producer.

5.2.7 Full Test TX

As described in section 4.5.6, testing all possible variations on the transmission pipeline is impossible. Based on the results of the preceding tests, a subset was selected for testing.

Setup 1

The first test setup was done without a separate thread for the generic netlink communication, and the main thread blocked while waiting for the response from the kernel thread. Initial tests showed that the kernel thread scheduled under CFS was not responsive enough. All tests were, therefore, done with the kernel thread scheduled under SCHED_FIFO. The application would ask the kernel module for more data if the TX driver queue contained less than ten packets, and the aperiodic scheme would sleep when all queues were empty.

The CPU load measured during the tests showed a massive discrepancy between the periodic and aperiodic schemes, as seen in Figure 5.42.

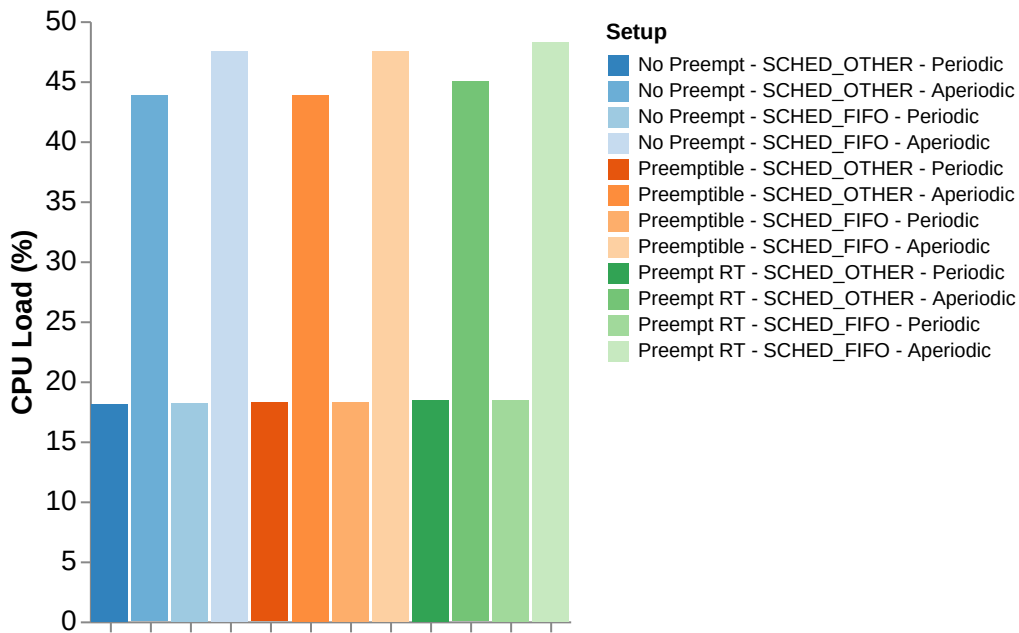


FIGURE 5.42: CPU load of full TX pipeline, setup 1, under Stress-NG 25 %

For the transmission tests, the lowest recorded buffer sizes towards the FPGA are the most crucial measure of success. Figure 5.43 shows the results when running with and without load.

Without load, the periodic scheme performed well across both scheduling policies. Interestingly, the aperiodic scheme performed well under SCHED_OTHER but could not keep up with the consumer under SCHED_FIFO. As predicted in section 4.5.6, the aperiodic scheme in this system configuration is CPU-hungry and causes the kernel to throttle the task when scheduled under SCHED_FIFO. Under SCHED_OTHER, the task does not need to sleep purposefully, as the kernel schedules it out when its timeslice is exhausted. This also explains the high CPU load of the aperiodic scheme compared with the periodic scheme.

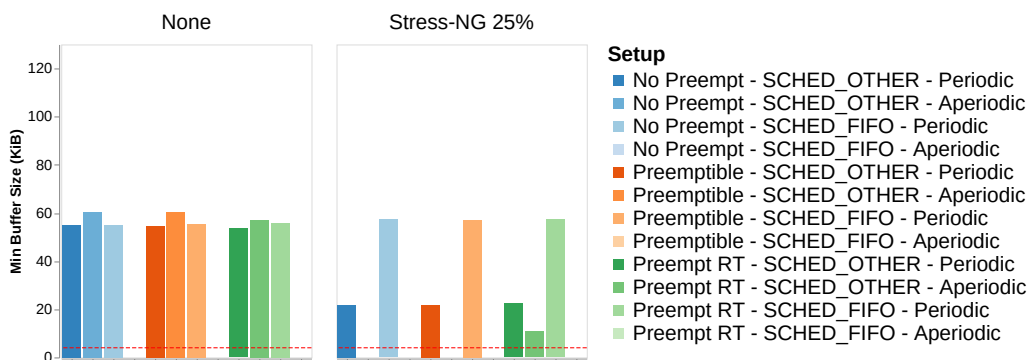


FIGURE 5.43: Minimum buffer size in TX pipeline, setup 1

When running with 25 % load, the periodic scheme could keep up under both scheduling policies but performed significantly better under SCHED_FIFO. The aperiodic scheme seems to have performed equally poorly under both scheduling policies, but the number of overruns shows a more nuanced picture. Under SCHED_OTHER, the buffer

underran 25 times, while under SCHED_FIFO, it underran 1190 times due to real-time throttling. Interestingly, the aperiodic task did not experience underruns when stressed under SCHED_OTHER on the fully preemptible kernel.

The periodic task performed well in the remaining stress scenarios when scheduled under SCHED_FIFO.

Setup 2

The second test setup ran with a separate thread for the Generic Netlink communication. The main thread transmits requests directly to the kernel thread, and another listens for responses. The condition requesting new packages was modified only to be done when all queues were empty², while the condition for sleeping remained the same.

The test was executed with the kernel thread scheduled under SCHED_OTHER and SCHED_FIFO.

The average CPU load measured when running the full transmission pipeline across the different configurations can be seen in Figure 5.44.

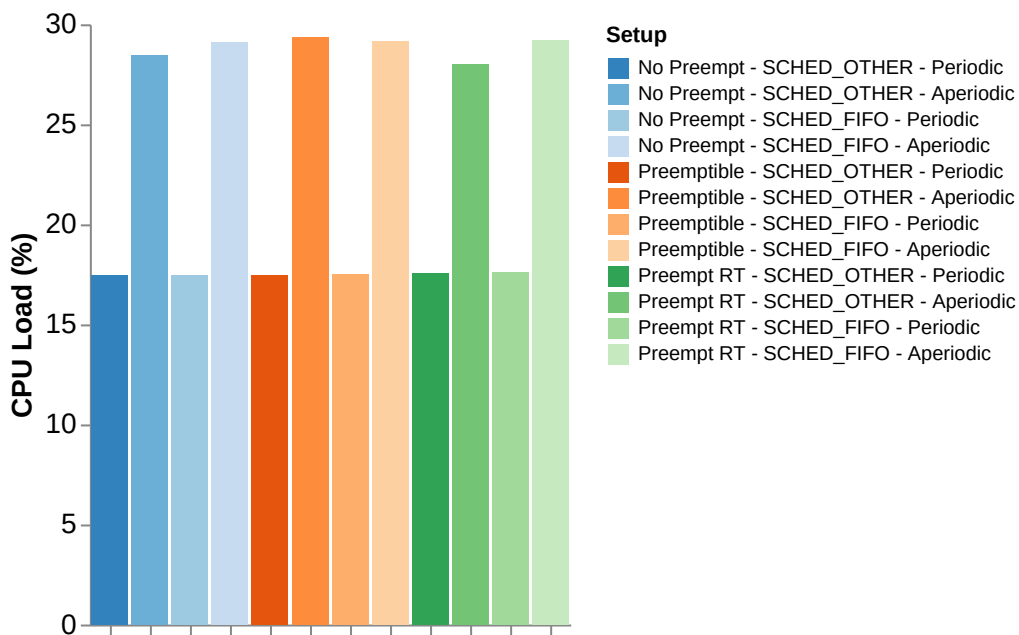


FIGURE 5.44: Average CPU load of full TX pipeline, setup 2 under Stress-NG
75 %

The difference in CPU load between the periodic and aperiodic approaches is noticeable, with the aperiodic scheme using about 11 % more CPU on average. However, the

²A brief test was executed with a separate thread for the generic netlink communication while keeping the other parameters the same as in the first setup. This setup was abandoned as the throttling issues grew even worse when the aperiodic task did not have to block on the receive call in the main thread anymore.

difference is much smaller than in the first setup, indicating that the change in condition for requesting new data impacted the CPU load positively, as the task will yield more frequently.

The minimum buffer size for this setup when scheduling the kernel thread under SCHED_OTHER can be seen in Figure 5.45. All tests performed equally well without load, but with 25% background load, the SCHED_OTHER tests fail. The periodic scheme seems to outperform the aperiodic scheme, which is somewhat surprising given that the aperiodic scheme spends more time on the CPU. The issues with throttling seen in the first setup have altogether disappeared.

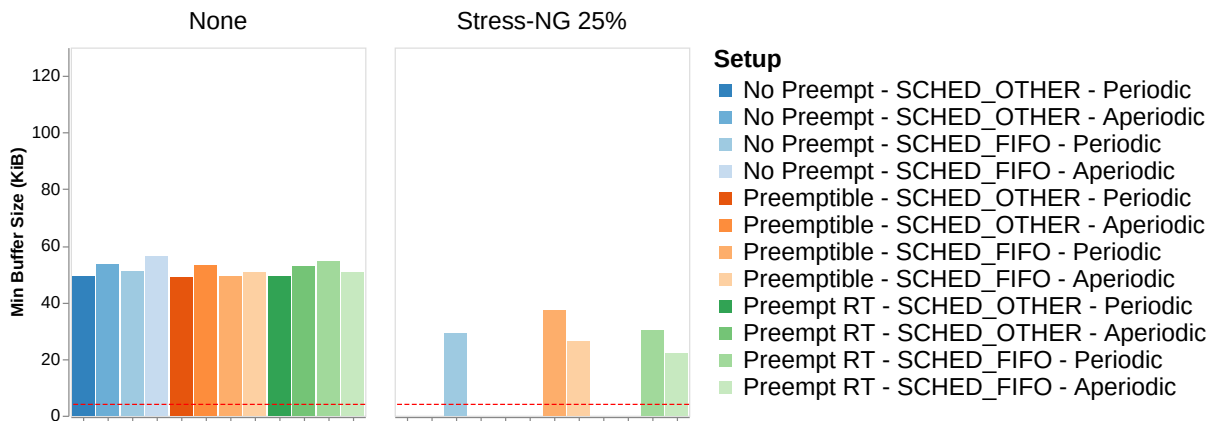


FIGURE 5.45: Minimum buffer size of full TX pipeline, setup 2 with kernel thread under SCHED_OTHER.

When scheduling the kernel thread under SCHED_FIFO, the results improved as the kernel module became more responsive to the main thread. SCHED_OTHER was still inferior to SCHED_FIFO, but all configurations could keep the buffer filled except for the aperiodic scheme on the non-preemptible kernel.

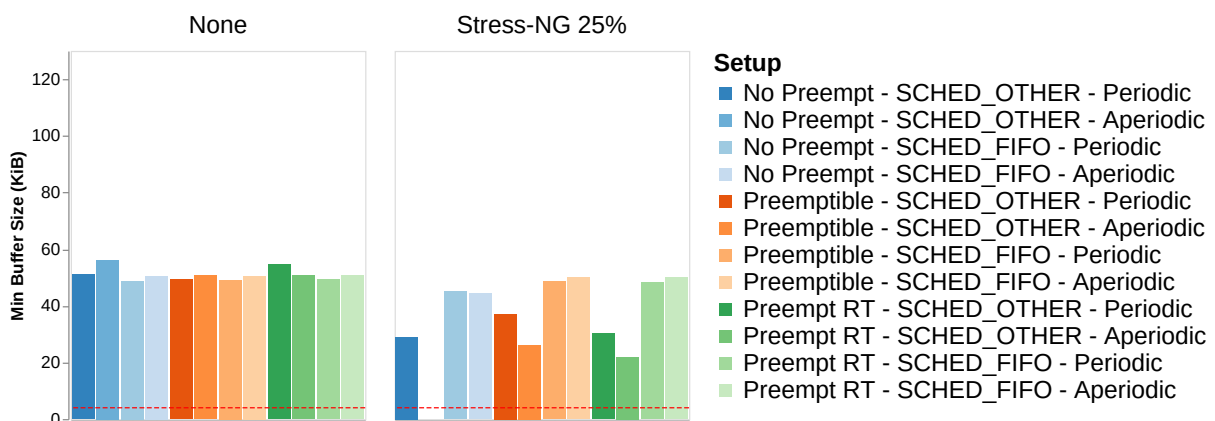


FIGURE 5.46: Minimum buffer size of full TX pipeline, setup 2 with kernel thread under SCHED_FIFO.

Setup 3

The third and final setup kept a separate thread for the Generic Netlink communication. The condition for requesting new packages was modified back to the condition in the first setup, but with an additional guard to ensure the application only sends one request at a time. Additionally, the kernel exclusively scheduled the kernel worker under SCHED_FIFO.

Figure 5.47 shows that the average CPU load measured during the tests was almost identical to the second setup.

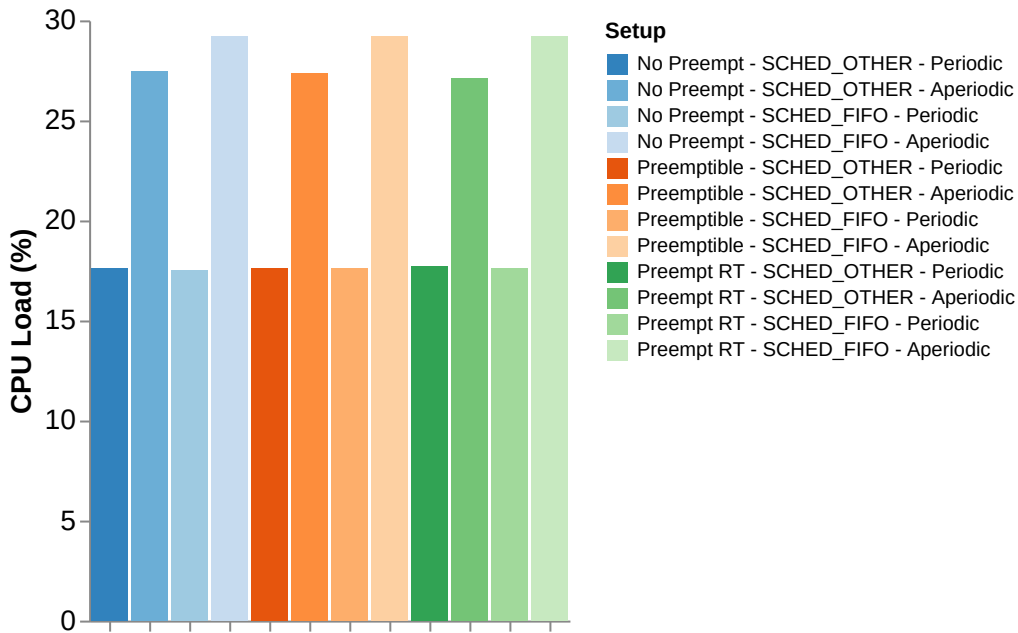


FIGURE 5.47: Average CPU load of full TX pipeline, setup 3

Figure 5.48 shows the minimum buffer sizes recorded during the tests. This shows that the SCHED_FIFO-based tests performed better, while the SCHED_OTHER tests performed worse when compared to the previous setups.

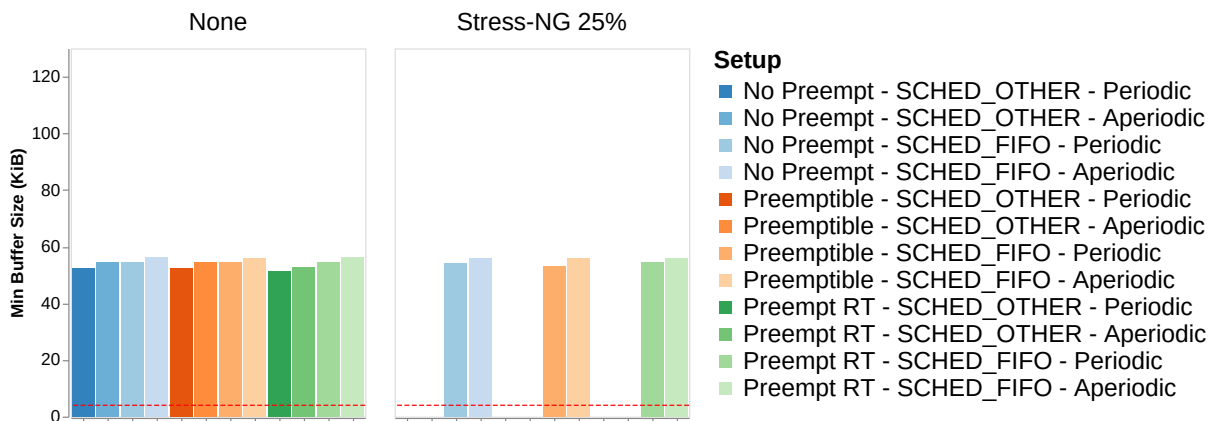


FIGURE 5.48: Minimum buffer size of full TX pipeline, setup 3 with kernel thread under SCHED_FIFO.

Comparison

Figure 5.49 compares the minimum buffer sizes recorded during the tests of the three setups.

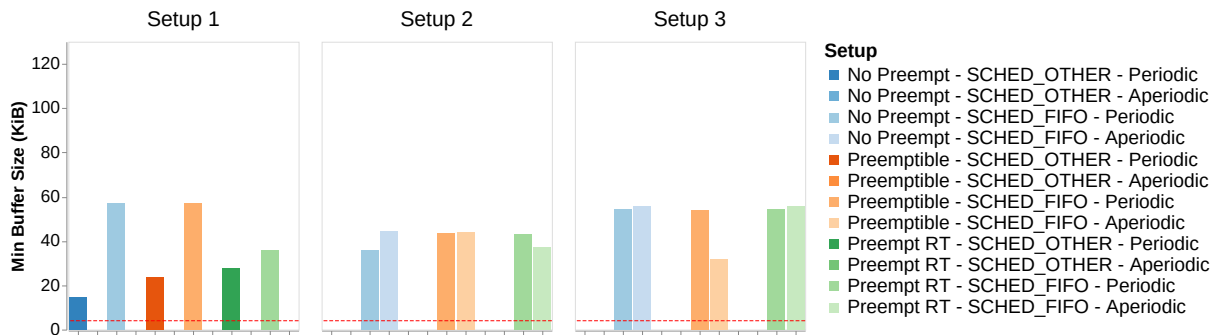


FIGURE 5.49: Minimum buffer size of Full TX pipeline compared across setups with 75 % load.

Here, it is more apparent that the first setup lends itself to an aperiodic scheme without real-time scheduling policies. This shows that the current platform and target system might be able to meet the requirements without using any of the techniques discussed in this thesis. The downside is that it uses a lot of CPU time and might not be as reliable as the other setups.

The second and third setups are suitable for both a periodic and an aperiodic scheme, with Setup 3 performing slightly better than Setup 2. In these setups, the periodic scheme performs slightly worse than the aperiodic scheme, but the difference is minuscule compared to the significant difference in CPU efficiency.

These conclusions are also supported by the number of recorded buffer underruns as seen in Figure 5.50. The aperiodic scheme in the first setup is victim to throttling under SCHED_FIFO, while in the second and third setups, the aperiodic scheme can no longer keep up when using SCHED_OTHER.

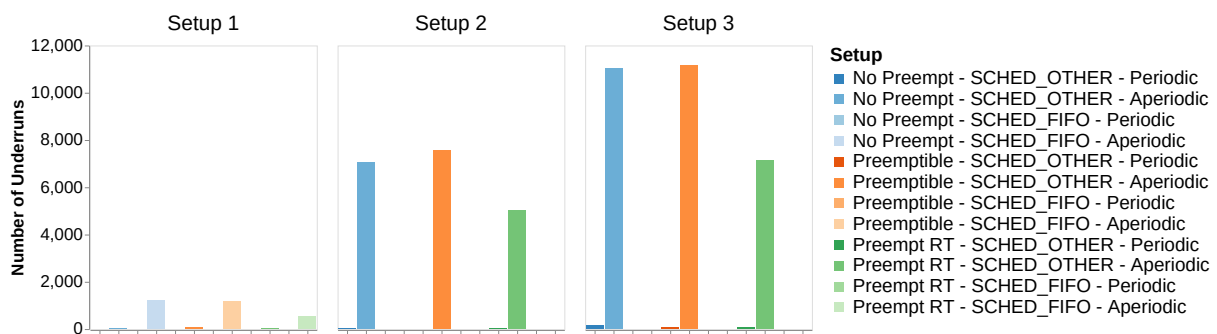


FIGURE 5.50: Number of underruns on Full TX pipeline compared across setups with 75 % load.

Interestingly, a periodic task firing at the rate required to keep up with the consumer is more CPU-effective than an aperiodic best-effort task, even when they perform the same amount of work.

5.2.8 Production Environment Test

The production environment test was conducted to evaluate the performance of the company's production Linux distribution. As found in section 5.1.2, `kmemleak` significantly impacted the system's latency and was disabled for this test. No artificial load was applied, meaning that the only load on the system was the typical background workload running on the system.

The latency recorded on the standard non-preemptible kernel can be seen in Figure 5.51, where CPU 0 had a task with a period of 400 μs and CPU 1 had a task with a period of 900 μs . Although the average latency was low, there were frequent spikes of about 500 microseconds. Additionally, isolated spikes of up to 2775 μs were recorded. As the results show no spikes between 750 μs and 2400 μs , it is likely that a single source of latency causes the highest spikes.

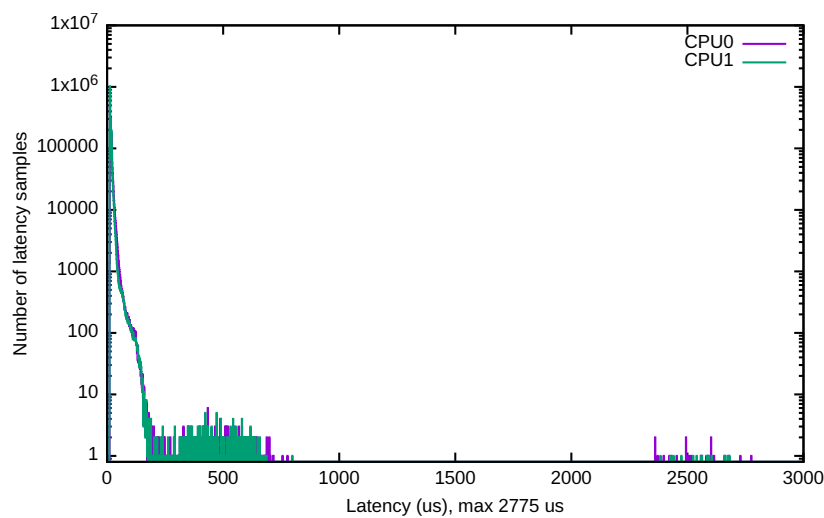
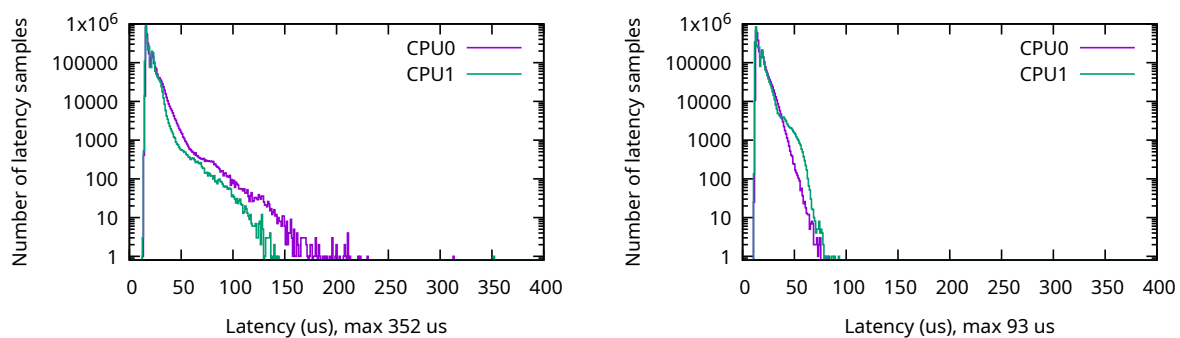


FIGURE 5.51: Latency distribution of the company's production Linux system with no forced preemption.

The preemptible kernels significantly improved the system's latency, as seen in Figure 5.52. A maximum latency of 352 μs was recorded on the preemptible kernel, and the fully preemptible kernel had a maximum latency of 93 μs .



(a) Preemptible kernel.

(b) Fully preemptible kernel.

FIGURE 5.52: Latency distribution of the company's production Linux system with preemptive kernels.

The effects of the preemption model on the system's average load can be seen in Figure 5.53 and were extracted by reading `/proc/loadavg`. The difference between the non-preemptible and preemptible kernels is slight and can be deemed a normal variation. However, the load average of the fully preemptible kernel is significantly higher than the other two.

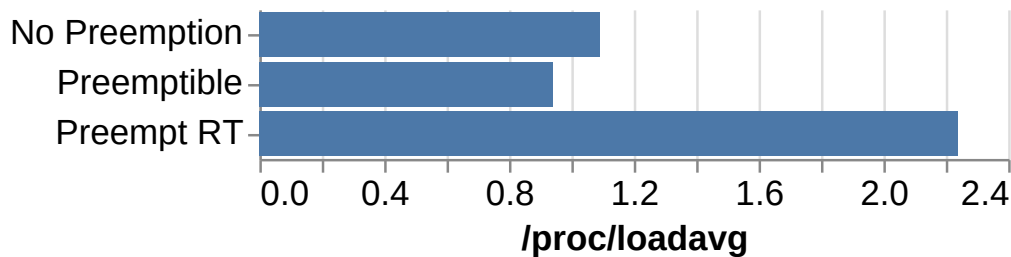


FIGURE 5.53: CPU load average of background workload on the company's production Linux distribution.

Since the load average on the fully preemptible kernel is greater than the number of CPU cores, one could argue that the system is overloaded. However, the load average in Linux is not only based on the number of runnable tasks but includes tasks in uninterruptible sleep, waiting for IO [27, `kernel/sched/loadavg.c`, Line 16]. In order to gain more insight, the CPU utilization of the system was measured with `mpstat` and is illustrated in Figure 5.54. This shows that the CPU is idle for about 90% of the time for all three preemption models. Based on this, one can guess that the high load average on the fully preemptible kernel is due to tasks in uninterruptible sleep, waiting for IO, and not due to a large number of runnable tasks. `Mpstat` reported that over the course of an hour, the fully preemptible kernel spent less time waiting for IO than the non-preemptible kernel. No documentation could be found to explain this. However, these findings could indicate that the fully preemptible kernel will immediately swap out a task waiting for IO. In contrast, the non-preemptible kernel may allow the task to spin on the CPU until the IO is complete. This would explain the fully preemptible kernel's higher load, as numerous tasks in uninterruptible sleep would always be present, waiting for IO. It is hard to say whether the high load average recorded is a risk for the target system. Still, it should be evaluated further if a fully preemptible kernel is considered.

The difference in time spent servicing soft interrupts between the preemption models could be explained by the fact that the fully preemptible kernel will handle all soft interrupts in the `ksoftirqd` daemon. In contrast, the other kernels will handle many of them in hard interrupt context. The `mpstat` documentation does not state how it calculates the soft interrupt column, making it difficult to confirm this.

Still, based strictly on the CPU's idle time, the fully preemptible kernel significantly impacts the system's CPU load negatively.

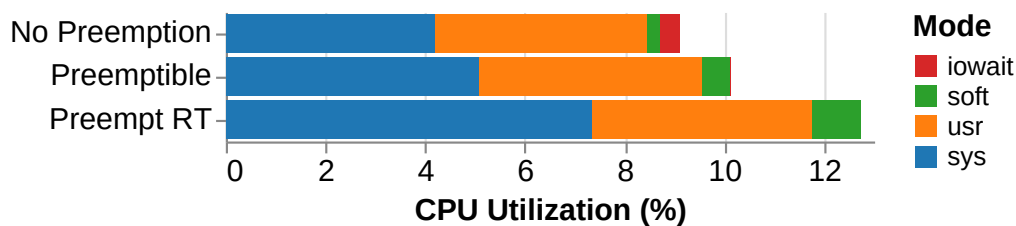


FIGURE 5.54: CPU utilization of background workload on the company’s production Linux distribution.

Based on these results, it is clear that the background workload on the company’s Linux distribution used in production environments does not cause critical latencies as long as `kmemleak` is disabled. A preemptible or fully preemptible kernel significantly reduces the latencies, but the fully preemptible kernel also significantly increases CPU load. Regarding the load average, the source code itself states that *“Its a silly number but people think its important”* [27, `kernel/sched/loadavg.c`, Line 6], meaning that it should not necessarily be taken as a critical measure of system performance.

5.3 Recommendations for the Target System

As stated, the main goal of this thesis is to contribute to the success of porting the legacy target system to a new hardware and software platform. This section discusses recommendations and measures that can be taken to ensure that the system will perform as desired on the new platform based on the findings of this thesis.

5.3.1 *If it Ain’t Broke, Don’t Fix it?*

Firstly, the results have shown that it is plausible for the system to work well on the new platform without any modifications to the application or the configuration of the software platform. Unfortunately, it is impossible to predict this with certainty, as the time complexity of the modulation algorithms and the background load of the target platform are both unknown.

However, the results show that while the system might work well, it can still be vulnerable to latencies induced by an external load. Edge cases in background tasks can cause the system to degrade in performance intermittently, or changes in a later release of the BSP can cause the system to fail. This means that taking measures to ensure the system’s reliability could be a good idea, even if the system works as intended.

5.3.2 Real-Time Scheduling Policies

The results have shown that the single most substantial factor in improving a system’s real-time performance is the application of real-time scheduling policies. This should be the first step in any scenario.

Upon enabling this, the system should be monitored to verify that real-time throttling does not occur. If it does, the application software should be tuned to yield the CPU to

a sufficient degree. A periodic scheme can be considered, which is discussed further in section 5.3.6.

A priority level below 50 should be tested first and potentially increased above 50 if required. This might cause latency or even starvation of critical interrupts. If kernel logs reveal this to be the case, one should consider increasing the priorities of relevant threads accordingly.

5.3.3 Memory

The dynamic memory allocation within the target application cannot be limited further due to its dependency on *libnl*. The results show that locking memory to RAM to avoid page faults can be beneficial, but the improvements are minor.

Still, the system should be monitored, and memory locking should be considered if page faults occur frequently.

In any regard, over-allocation of memory should be disabled to avoid out-of-memory situations.

5.3.4 kmemleak

Kmemleak was found to be enabled on the kernel used in production environments in the company's systems. The results found that this can periodically cause latencies of up to 40 ms.

This is at the threshold of what the target system can potentially withstand and could lead to intermittent needs for retransmissions. If the current preemption model is kept, kmemleak should be disabled. If other preemption models are chosen, the results show that kmemleak does not significantly impact the system's scheduling latency. However, as the official documentation does not recommend using it in production systems, disabling it should still be considered.

5.3.5 Preemption Model

A definite recommendation regarding the preemption model of the platform is challenging to make, as the results show that the nature of the background load is the determining factor.

If background tasks frequently perform system calls, a kernel without forced preemption might still allow the target application to meet its requirements. It will only become an issue if the background tasks perform system calls with long durations, leading to large spikes in latency for the target application.

If this is found to be the case, the preemption model should be changed to *Preemptible Kernel (Low-Latency Desktop)*. This allows a real-time scheduled task to preempt tasks in kernel mode as long as they are not in a critical section. The results found that the throughput reductions induced by this modification are negligible in the scenarios tested.

Concerning `PREEMPT_RT`, the analysis of the target system shows that its real-time constraints are not stringent enough to warrant the need for a fully preemptible kernel. On the contrary, the results show that the throughput reduction brought on by `PREEMPT_RT` can sometimes be extreme and could negatively impact the target system, which is also dependent on performance.

Based on the results of the production environment test, the non-preemptible kernel or preemptible kernel should have sufficient real-time capabilities for the target system if the actual workload of the final system is similar to the background workload of the test.

5.3.6 Modifications to the Legacy Application

The previous sections have identified several recommended modifications to the application software. Firstly, the mechanism for sleeping should be changed. The existing infrastructure using an interval timer should be removed entirely and replaced with `nanosleep` or a condition variable with a timeout if waking up on an event is still required. If the application is not scheduled under a real-time scheduling policy, this will not improve the scheduling latency of the thread. However, the results have shown that this will reduce the CPU load on the system, making it beneficial in any case. This is especially true when scheduled under CFS, as it will increase the timeslice of remaining tasks.

The kernel thread responsible for handling data requests from the application should be modified to use an event-driven scheme instead of polling for events. This will reduce the CPU load of the system, and the results show that this will not negatively impact the system's latency. Further, if the blocking read call to the kernel module is not separated into a separate thread, the kernel thread should be modified to use a real-time scheduling policy. The results show that this will improve the responsiveness of the kernel module.

Whether handling packages to and from the kernel module should be moved to separate threads is an open question. It can improve the determinism of the main thread but will also make for a slightly more complicated system. Additionally, it might reduce the system's throughput, depending on the priority of the threads. In some setups, the aperiodic thread had a very high CPU load, in some cases over 50%. This means the thread spent one hundred percent of a single core's capacity, which does not produce a reliable system. Separating the handling of the Generic Netlink interface towards the kernel module from the main thread will reduce the CPU load of the main thread. If the time complexity of the modulation algorithms proves to be too demanding, this could be a potential solution.

The same goes for whether a periodic scheme should be preferred. As shown throughout this thesis, reasoning about the reliability of a system based on an aperiodic, best-effort task is difficult. It can occasionally improve the throughput but also makes it more challenging to analyze the temporal characteristics of the system. In some setups, the aperiodic scheme resulted in real-time throttling when paired with a real-time scheduling policy, which led to it performing worse than when scheduled under CFS. In the full tests, the aperiodic scheme had a higher CPU load than the periodic scheme, even when they performed practically the same in keeping up with the rate of

the FPGA. Although this can be tuned by tweaking the conditions for when the main thread should sleep, it still highlights the unpredictability of this approach.

The fact is that the full transmission test, unfortunately, was not able to mimic the actual system appropriately, meaning that the results are not directly applicable to the target system. Although these results provide an indication of what to expect, performing benchmarks and experiments with the actual system is required to uncover the optimal solution.

5.4 Summary

This chapter has described and discussed the results of the tests conducted in this thesis. Further, it has given several recommendations for the target system based on these results, both concerning the software platform and the application software. The main finding is that the porting effort of the target system is feasible. It was found that utilizing real-time scheduling policies has a much greater impact on the system's latency than the preemption model. The best course of action for the target system is to apply real-time scheduling policies to the application software and monitor the system for real-time throttling. If high latencies are still experienced, modifying the preemption model to *Preemptible Kernel (Low-Latency Desktop)* should be sufficient.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The goal of this thesis is to enable the porting of a legacy system with real-time constraints to a Linux platform on an Altera Cyclone V. To achieve this, the thesis has investigated, analyzed, and tested state-of-the-art techniques for improving the real-time capabilities of Linux. A review of the literature was conducted with a focus on the PREEMPT_RT patch, which is the most widely used technique for reducing the scheduling latency of Linux and increasing its determinism. Various best practices and recommendations were identified, which made up the foundation for the methodology of the thesis.

Several generic tests were developed and conducted to evaluate the impact of using different preemption models and scheduling policies. These experiments showed that while the PREEMPT_RT patch can improve the real-time capabilities of Linux, it brings with it a reduction in throughput that can be significant, depending on the workload. It also introduces risk factors that can negatively impact the system, such as the possibility of system crashes due to starving interrupts on a poorly configured system. These findings align with the current assertions in the literature. Additionally, the results show that using the preemptible low-latency kernel, available in the mainline tree, significantly improves the scheduling latencies of the system without showing the same reduction in throughput.

The legacy system was analyzed, and parts of the time-critical functionality were implemented and tested in isolation. The results show that the system's temporal requirements are well within what is possible to meet with Linux and that the porting effort is feasible. The target system might function well on a mainline non-preemptible kernel without taking any additional measures. However, applying real-time scheduling policies and a preemptible low-latency kernel will improve the system's reliability and stability. The results showed that the most significant variable impacting the scheduling latency of a task is the choice of scheduling policy. Whether a preemptible kernel is required depends more on the background workload of the system than the nature of the target application itself. However, based on the results and the assumptions made, the requirements of the system are not strict enough to justify using the PREEMPT_RT patchset.

Based on the results, the thesis has also proposed several modifications to the target application to reduce the CPU load and increase the system's predictability.

6.2 Future Work

With regard to optimizing both Linux as a platform and the target system specifically, numerous configurations and mechanisms could be further investigated.

The thesis only had the temporal scope to evaluate a subset of these, barely scratching the surface of what is possible.

Below are some suggestions for future work:

- **Using Real Target System:** As this thesis was conducted as a pre-study of a planned porting effort, the actual system was not available for experimentation. Therefore, the results were produced with software mimicking the target system, which was based on several assumptions. Most notable is the fact that the modulation algorithms' time complexity and the main thread's average execution time are unknown. Additionally, the background load of the actual system is unknown, which significantly impacts the system's real-time capabilities. The assumptions were sometimes sufficient, but some questions cannot be answered without testing the actual system. This includes whether an aperiodic or periodic execution scheme should be preferred and whether the application should be further partitioned into multiple tasks. Testing the recommended modifications to the actual target platform and application will provide more information and be a natural next step after this thesis.
- **Stressors:** Several stressors were used to evaluate the latency and throughput of the system under various conditions. The results varied greatly across some of these mechanisms, for instance, between using pipes and sockets in the Hackbench throughput test. Delving into the reasons for this and investigating which operations are more prone to throughput reductions and a cause of high latencies could be an interesting area for future work.
- **Priority Boosting:** To increase the round trip time of packets passing through the target system, different schemes of priority boosting through the use of priority inheritance could be investigated. This could potentially be used to temporarily boost specific threads depending on the current state of the system or if data is piling up in specific queues.
- **Latency NICE:** *Latency NICE* is a recent interesting concept that allows a task scheduled under CFS to split its timeslice into smaller slices and lower its response time. For the target system, this could potentially be a good alternative to using real-time scheduling policies. This is especially true if the target application is found not to yield the CPU sufficiently under `SCHED_FIFO` and is subject to real-time throttling.
- **CPU Isolation:** Because of the limited number of CPU cores on the Altera Cyclone V, CPU isolation was not investigated in this thesis. Additionally, the requirements for the target system did not warrant using this. However, are the requirements for the target system to change, or should the background workload create latencies the system cannot handle, this could be an interesting area to investigate.

Appendix A

BSP Configuration

A.1 Crosstool-NG Configuration

```
1 CT_CONFIG_VERSION="4"  
2 CT_EXPERIMENTAL=y  
3 CT_LOCAL_TARBALLS_DIR="./local-tarballs"  
4 CT_PREFIX_DIR=" ../x-tools/${CT_TARGET}"  
5 CT_ARCH_ARM=y  
6 CT_ARCH_CPU="cortex-a9"  
7 CT_ARCH_FPU="neon"  
8 CT_ARCH_FLOAT_HW=y  
9 CT_TARGET_VENDOR="cortexa9_neon"  
10 CT_TARGET_ALIAS="arm-cyclone5-linux"  
11 CT_TOOLCHAIN_ENABLE_NLS=y  
12 CT_KERNEL_LINUX=y  
13 CT_LINUX_V_6_1=y  
14 CT_BINUTILS_LINKER_LD_GOLD=y  
15 CT_BINUTILS_GOLD_THREADS=y  
16 CT_BINUTILS_LD_WRAPPER=y  
17 CT_BINUTILS_PLUGINS=y  
18 CT_GLIBC_USE_LIBIDN_ADDON=y  
19 CT_GLIBC_LOCALES=y  
20 CT_GLIBC_KERNEL_VERSION_NONE=y  
21 # CT_CC_GCC_ENABLE_TARGET_OPTSPACE is not set  
22 CT_CC_LANG_CXX=y  
23 CT_DEBUG_GDB=y  
24 # CT_GDB_CROSS_PYTHON is not set  
25 CT_GDB_NATIVE=y  
26 CT_GDB_NATIVE_STATIC=y
```

LISTING A.1: Crosstool-NG configuration.

A.2 Linux Configuration

```
1 CONFIG_SYSVIPC=y  
2 CONFIG_HIGH_RES_TIMERS=y  
3 CONFIG_BPF_SYSCALL=y  
4 CONFIG_IKCONFIG=y
```

```
5 CONFIG_IKCONFIG_PROC=y
6 CONFIG_LOG_BUF_SHIFT=14
7 CONFIG_CGROUPS=y
8 CONFIG_CPUSETS=y
9 CONFIG_NAMESPACES=y
10 CONFIG_BLK_DEV_INITRD=y
11 CONFIG_EXPERT=y
12 CONFIG_PROFILING=y
13 CONFIG_ARCH_INTEL_SOCFPGA=y
14 CONFIG_ARM_THUMBEE=y
15 CONFIG_SMP=y
16 CONFIG_NR_CPUS=2
17 CONFIG_HIGHMEM=y
18 CONFIG_VFP=y
19 CONFIG_NEON=y
20 CONFIG_MODULES=y
21 CONFIG_MODULE_UNLOAD=y
22 CONFIG_CMA=y
23 CONFIG_NET=y
24 CONFIG_PACKET=y
25 CONFIG_UNIX=y
26 CONFIG_NET_KEY=y
27 CONFIG_NET_KEY_MIGRATE=y
28 CONFIG_INET=y
29 CONFIG_IP_MULTICAST=y
30 CONFIG_IP_PNP=y
31 CONFIG_IP_PNP_DHCP=y
32 CONFIG_IP_PNP_BOOTP=y
33 CONFIG_IP_PNP_RARP=y
34 CONFIG_NETWORK_PHY_TIMESTAMPING=y
35 CONFIG_VLAN_8021Q=y
36 CONFIG_VLAN_8021Q_GVRP=y
37 CONFIG_CAN=y
38 CONFIG_PCI=y
39 CONFIG_PCI_MSI=y
40 CONFIG_PCIE_ALTERA=y
41 CONFIG_PCIE_ALTERA_MSI=y
42 CONFIG_DEVTMPFS=y
43 CONFIG_DEVTMPFS_MOUNT=y
44 CONFIG_MTD=y
45 CONFIG_MTD_BLOCK=y
46 CONFIG_MTD_RAW_NAND=y
47 CONFIG_MTD_NAND_DENALI_DT=y
48 CONFIG_MTD_SPI_NOR=y
49 # CONFIG_MTD_SPI_NOR_USE_4K_SECTORS is not set
50 CONFIG_OF_OVERLAY=y
51 CONFIG_BLK_DEV_LOOP=y
52 CONFIG_BLK_DEV_RAM=y
53 CONFIG_BLK_DEV_RAM_COUNT=2
54 CONFIG_BLK_DEV_RAM_SIZE=8192
55 CONFIG_BLK_DEV_NVME=m
56 CONFIG_SRAM=y
57 CONFIG_EEPROM_AT24=y
58 CONFIG_SCSI=y
59 # CONFIG_SCSI_PROC_FS is not set
60 CONFIG_BLK_DEV_SD=y
61 # CONFIG_BLK_DEV_BSG is not set
62 # CONFIG_SCSI_LOWLEVEL is not set
```

```
63 CONFIG_NETDEVICES=y
64 CONFIG_ALTERA_TSE=m
65 CONFIG_E1000E=m
66 CONFIG_IGB=m
67 CONFIG_IXGBE=m
68 CONFIG_STMMAC_ETH=y
69 CONFIG_MARVELL_PHY=y
70 CONFIG_MICREL_PHY=y
71 CONFIG_CAN_C_CAN=y
72 CONFIG_CAN_C_CAN_PLATFORM=y
73 CONFIG_CAN_DEBUG_DEVICES=y
74 CONFIG_INPUT_EVDEV=y
75 CONFIG_INPUT_TOUCHSCREEN=y
76 CONFIG_TOUCHSCREEN_STMPE=y
77 # CONFIG_SERIO_SERPORT is not set
78 CONFIG_SERIO_AMBAKMI=y
79 CONFIG_LEGACY_PTY_COUNT=16
80 CONFIG_SERIAL_8250=y
81 CONFIG_SERIAL_8250_CONSOLE=y
82 CONFIG_SERIAL_8250_NR_UARTS=2
83 CONFIG_SERIAL_8250_RUNTIME_UARTS=2
84 CONFIG_SERIAL_8250_DW=y
85 CONFIG_I2C=y
86 CONFIG_I2C_CHARDEV=y
87 CONFIG_I2C_DESIGNWARE_PLATFORM=y
88 CONFIG_SPI=y
89 CONFIG_SPI_CADENCE_QUADSPI=y
90 CONFIG_SPI_DESIGNWARE=y
91 CONFIG_SPI_DW_MMIO=y
92 CONFIG_SPI_SPIDEV=y
93 CONFIG_GPIOLIB=y
94 CONFIG_GPIO_SYSFS=y
95 CONFIG_GPIO_ALTERA=y
96 CONFIG_GPIO_DWAPB=y
97 CONFIG_GPIO_ALTERA_A10SR=y
98 CONFIG_SENSORS_MAX1619=y
99 CONFIG_PMBUS=y
100 CONFIG_SENSORS_LTC2978=y
101 CONFIG_SENSORS_LTC2978_REGULATOR=y
102 CONFIG_WATCHDOG=y
103 CONFIG_DW_WATCHDOG=y
104 CONFIG_MFD_ALTERA_A10SR=y
105 CONFIG_MFD_ALTERA_SYSMGR=y
106 CONFIG_MFD_STMPE=y
107 CONFIG_REGULATOR=y
108 CONFIG_REGULATOR_FIXED_VOLTAGE=y
109 CONFIG_USB=y
110 CONFIG_USB_STORAGE=y
111 CONFIG_USB_DWC2=y
112 CONFIG_NOP_USB_XCEIV=y
113 CONFIG_USB_GADGET=y
114 CONFIG_MMC=y
115 CONFIG_MMC_DW=y
116 CONFIG_NEW_LEDS=y
117 CONFIG_LEDS_CLASS=y
118 CONFIG_LEDS_GPIO=y
119 CONFIG_LEDS_TRIGGERS=y
120 CONFIG_LEDS_TRIGGER_TIMER=y
```



```
121 CONFIG_LEDS_TRIGGER_CPU=y
122 CONFIG_RTC_CLASS=y
123 CONFIG_RTC_DRV_DS1307=y
124 CONFIG_DMADEVICES=y
125 CONFIG_PL330_DMA=y
126 CONFIG_DMATEST=m
127 CONFIG_IIO=y
128 CONFIG_LTC2497=y
129 CONFIG_FPGA=y
130 CONFIG_FPGA_MGR_SOCFPGA=y
131 CONFIG_FPGA_MGR_SOCFPGA_A10=y
132 CONFIG_FPGA_BRIDGE=y
133 CONFIG_SOCFPGA_FPGA_BRIDGE=y
134 CONFIG_ALTERA_FREEZE_BRIDGE=y
135 CONFIG_FPGA_REGION=y
136 CONFIG_EXT2_FS=y
137 CONFIG_EXT2_FS_XATTR=y
138 CONFIG_EXT2_FS_POSIX_ACL=y
139 CONFIG_EXT3_FS=y
140 CONFIG_AUTOFS_FS=y
141 CONFIG_VFAT_FS=y
142 CONFIG_NTFS_FS=y
143 CONFIG_NTFS_RW=y
144 CONFIG_TMPFS=y
145 CONFIG_JFFS2_FS=y
146 CONFIG_NFS_FS=y
147 CONFIG_NFS_V3_ACL=y
148 CONFIG_NFS_V4=y
149 CONFIG_ROOT_NFS=y
150 CONFIG_NFSD=y
151 CONFIG_NFSD_V3_ACL=y
152 CONFIG_NFSD_V4=y
153 CONFIG_NLS_CODEPAGE_437=y
154 CONFIG_NLS_ISO8859_1=y
155 CONFIG_DMA_CMA=y
156 CONFIG_PRINTK_TIME=y
157 CONFIG_DEBUG_INFO_DWARF_TOOLCHAIN_DEFAULT=y
158 CONFIG_DEBUG_INFO_BTF=y
159 CONFIG_MAGIC_SYSRQ=y
160 CONFIG_DEBUG_FS=y
161 CONFIG_DETECT_HUNG_TASK=y
162 CONFIG_SCHEDSTATS=y
163 CONFIG_FUNCTION_TRACER=y
164 CONFIG_DEBUG_USER=y
```

LISTING A.2: Linux kernel configuration.

A.3 Preemptible Mainline Kernel Configuration

```
1 diff --git a/linux_defconfig b/linux-preempt-dynamic_defconfig
2 index 823ca49..d30fec3 100644
3 --- a/linux_defconfig
4 +++ b/linux-preempt-dynamic_defconfig
```

```
5 @@ -1,6 +1,8 @@
6 +CONFIG_LOCALVERSION="-preempt-dynamic"
7 CONFIG_SYSVIPC=y
8 CONFIG_HIGH_RES_TIMERS=y
9 CONFIG_BPF_SYSCALL=y
10 +CONFIG_PREEMPT=y
11 CONFIG_IKCONFIG=y
12 CONFIG_IKCONFIG_PROC=y
13 CONFIG_LOG_BUF_SHIFT=14
```

LISTING A.3: Preemptible mainline kernel configuration diff

A.4 Linux-stable-rt Configuration

```
1 diff --git a/linux_defconfig b/linuxrt_defconfig
2 index 823ca49..eb0c6c2 100644
3 --- a/linux_defconfig
4 +++ b/linuxrt_defconfig
5 @@ -1,6 +1,7 @@
6 CONFIG_SYSVIPC=y
7 CONFIG_HIGH_RES_TIMERS=y
8 CONFIG_BPF_SYSCALL=y
9 +CONFIG_PREEMPT_RT=y
10 CONFIG_IKCONFIG=y
11 CONFIG_IKCONFIG_PROC=y
12 CONFIG_LOG_BUF_SHIFT=14
13 @@ -117,7 +118,6 @@ CONFIG_LEDS_CLASS=y
14 CONFIG_LEDS_GPIO=y
15 CONFIG_LEDS_TRIGGERS=y
16 CONFIG_LEDS_TRIGGER_TIMER=y
17 -CONFIG_LEDS_TRIGGER_CPU=y
18 CONFIG_RTC_CLASS=y
19 CONFIG_RTC_DRV_DS1307=y
20 CONFIG_DMADEVICES=y
```

LISTING A.4: PREEMPT_RT kernel configuration diff

Appendix B

Stressor Configurations

```

1 # Stress-NG --cpu $(nproc) \
2   --cpu-load 25 &
3
4 # htop
5   0[#####]          25.7%] Tasks: 10, 0 thr, 62 kthr; 1 run..
6   1[#####]          25.0%] Load average: 1.43 1.50 1.21
7 Mem[ |*@           19.5M/998M] Uptime: 1 day, 00:25:22
8 Swp[              0K/0K]
9
10 [Main] [I/O]
11 PID USER      PRI  NI  VIRT   RES   SHR  S  CPU% MEM%   TIME+  Command
12 598 root        20   0 12864   2824  1408  S   25.7  0.3   0:05.26 stress -..
13 599 root        20   0 12864   3080  1536  S   25.0  0.3   0:05.26 stress -..
14 600 root        20   0  2816   2176  2048  R    0.7  0.2   0:00.16 htop
15   1 root        20   0  2900   1536  1536  S    0.0  0.2   0:00.31 init
16 110 root        20   0  2900   1664  1664  S    0.0  0.2   0:00.01 /sbin/s..
17 119 root        20   0  2900   1664  1664  S    0.0  0.2   0:00.01 /sbin/k..
18 130 root        20   0 16024   1884  1280  S    0.0  0.2   0:00.20 /sbin/u..
19 164 root        20   0  6904   3316  2816  S    0.0  0.3   0:00.00 sshd: /..
20 168 root        20   0  2900   1920  1792  S    0.0  0.2   0:00.23 -sh
21 596 root        20   0 12864   2944  2688  S    0.0  0.3   0:00.00 stress -..

```

LISTING B.1: Stressor: Stress-NG 25%.

```

1 # Stress-NG --cpu $(nproc) \
2   --cpu-load 75 &
3
4 # htop
5   0[#####* 77.6%] Tasks: 10, 0 thr, 63 kthr; 2 run..
6   1[##### 71.7%] Load average: 1.61 1.46 1.22
7 Mem[|*@          19.7M/998M] Uptime: 1 day, 00:27:26
8 Swp[              0K/0K]
9
10 [Main] [I/O]
11 PID USER      PRI  NI  VIRT   RES   SHR  S  CPU% MEM%   TIME+  Command
12 604 root        20   0 12864  2952  1536  R   77.0  0.3   0:40.45 stress -..
13 605 root        20   0 12864  3080  1536  R   72.4  0.3   0:40.52 stress -..
14 606 root        20   0  2816  2304  2048  R    0.7  0.2   0:00.39 htop
15   1 root        20   0  2900  1536  1536  S    0.0  0.2   0:00.31 init
16 110 root        20   0  2900  1664  1664  S    0.0  0.2   0:00.01 /sbin/s..
17 119 root        20   0  2900  1664  1664  S    0.0  0.2   0:00.01 /sbin/k..
18 130 root        20   0 16024  1884  1280  S    0.0  0.2   0:00.20 /sbin/u..
19 164 root        20   0  6904  3316  2816  S    0.0  0.3   0:00.00 sshd: /..
20 168 root        20   0  2900  1920  1792  S    0.0  0.2   0:00.24 -sh
21 602 root        20   0 12864  2944  2688  S    0.0  0.3   0:00.01 stress -..

```

LISTING B.2: Stressor: Stress-NG 75 %.

```

1 # iperf3 --server &
2
3 # iperf3 --client localhost \
4   --time 0 &
5
6 # htop
7   0[#####85.5%] Tasks: 9, 0 thr, 64 kthr; 2 runn..
8   1[#####90.1%] Load average: 1.67 0.90 0.96
9 Mem[|*@          19.5M/998M] Uptime: 1 day, 00:22:20
10 Swp[              0K/0K]
11
12 [Main] [I/O]
13 PID USER      PRI  NI  VIRT   RES   SHR  S  CPU% MEM%   TIME+  Command
14 590 root        20   0  6020  2816  2432  R   92.1  0.3   0:45.34 iperf3 ..
15 589 root        20   0  6020  2560  2176  R   85.5  0.3   0:41.54 iperf3 ..
16 591 root        20   0  2816  2304  2048  R    1.3  0.2   0:00.43 htop
17   1 root        20   0  2900  1536  1536  S    0.0  0.2   0:00.31 init
18 110 root        20   0  2900  1664  1664  S    0.0  0.2   0:00.01 /sbin/s..
19 119 root        20   0  2900  1664  1664  S    0.0  0.2   0:00.01 /sbin/k..
20 130 root        20   0 16024  1884  1280  S    0.0  0.2   0:00.20 /sbin/u..
21 164 root        20   0  6904  3316  2816  S    0.0  0.3   0:00.00 sshd: /..
22 168 root        20   0  2900  1920  1792  S    0.0  0.2   0:00.21 -sh

```

LISTING B.3: Stressor: Iperf.

```

1 # hackbench --loops 1000000000 \
2     --groups 1 \
3     --fds 1 \
4     --datasize 1 &
5
6 # htop
7   0[#####87.5%] Tasks: 10, 0 thr, 64 kthr; 2 run..
8   1[#####90.1%] Load average: 2.31 1.93 1.22
9 Mem[|*@          17.4M/998M] Uptime: 1 day, 00:12:24
10 Swp[              0K/0K]
11
12 [Main] [I/O]
13 PID USER      PRI  NI  VIRT   RES   SHR  S  CPU% MEM%   TIME+  Command
14 578 root        20   0  2000   256   256  R  88.8  0.0   0:20.98  hackben..
15 579 root        20   0  2000   256   256  S  88.8  0.0   0:20.75  hackben..
16 580 root        20   0  2816  2176  2048  R   0.7  0.2   0:00.20  htop
17   1 root        20   0  2900  1536  1536  S   0.0  0.2   0:00.31  init
18 110 root        20   0  2900  1664  1664  S   0.0  0.2   0:00.01  /sbin/s..
19 119 root        20   0  2900  1664  1664  S   0.0  0.2   0:00.01  /sbin/k..
20 130 root        20   0 16024  1884  1280  S   0.0  0.2   0:00.20  /sbin/u..
21 164 root        20   0  6904  3316  2816  S   0.0  0.3   0:00.00  sshd: /..
22 168 root        20   0  2900  1920  1792  S   0.0  0.2   0:00.17  -sh
23 577 root        20   0  2000  1408  1408  S   0.0  0.1   0:00.00  hackben..

```

LISTING B.4: Stressor: Hackbench.

Appendix C

Additional Result Data

C.1 Baseline Tests

C.1.1 High-Resolution Timers Verification

```
1 # cat /proc/timer_list
2 Timer List Version: v0.9
3 HRTIMER_MAX_CLOCK_BASES: 8
4 now at 1422130411630 nsecs
5
6 cpu: 0
7 clock 0:
8   .base:      ac6c6ed5
9   .index:     0
10  .resolution: 1 nsecs
11  .get_time:   ktime_get
12  .offset:     0 nsecs
13 active timers:
14 #0: <4206475a>, tick_sched_timer, S:01
15 # expires at 1422140000000-1422140000000 nsecs [in 9588370 to 9588370
16   nsecs]
17 #1: <a6e5a016>, watchdog_timer_fn, S:01
18 # expires at 1424000000000-1424000000000 nsecs [in 1869588370 to
19   1869588370 nsecs]
20 #2: <d621cea0>, sched_clock_poll, S:01
21 # expires at 1438814043825-1438814043825 nsecs [in 16683632195 to
22   16683632195 nsecs]
23 clock 1:
24   .base:      bc28a660
25   .index:     1
26   .resolution: 1 nsecs
27   .get_time:   ktime_get_real
28   .offset:     0 nsecs
29 active timers:
30 clock 2:
31   .base:      6afd47a5
32   .index:     2
33   .resolution: 1 nsecs
34   .get_time:   ktime_get_boottime
35   .offset:     0 nsecs
36 active timers:
37 clock 3:
38   .base:      e3c3f0ba
```

```
36 .index: 3
37 .resolution: 1 nsecs
38 .get_time: ktime_get_clocktai
39 .offset: 0 nsecs
40 active timers:
41 clock 4:
42 .base: 8dd9f45c
43 .index: 4
44 .resolution: 1 nsecs
45 .get_time: ktime_get
46 .offset: 0 nsecs
47 active timers:
48 #0: <9c71ed39>, pm_suspend_timer_fn, S:01
49 # expires at 1423400099080-1424150099080 nsecs [in 1269687450 to
50 2019687450 nsecs]
51 clock 5:
52 .base: 39e636eb
53 .index: 5
54 .resolution: 1 nsecs
55 .get_time: ktime_get_real
56 .offset: 0 nsecs
57 active timers:
58 clock 6:
59 .base: 477a811a
60 .index: 6
61 .resolution: 1 nsecs
62 .get_time: ktime_get_boottime
63 .offset: 0 nsecs
64 active timers:
65 clock 7:
66 .base: 0c7038cd
67 .index: 7
68 .resolution: 1 nsecs
69 .get_time: ktime_get_clocktai
70 .offset: 0 nsecs
71 active timers:
72 .expires_next : 1422140000000 nsecs
73 .hres_active : 1
74 .nr_events : 143639
75 .nr_retries : 2
76 .nr_hangs : 0
77 .max_hang_time : 0
78 .nohz_mode : 0
79 .last_tick : 0 nsecs
80 .tick_stopped : 0
81 .idle_jiffies : 0
82 .idle_calls : 0
83 .idle_sleeps : 0
84 .idle_entrytime : 0 nsecs
85 .idle_waketime : 0 nsecs
86 .idle_exittime : 0 nsecs
87 .idle_sleeptime : 0 nsecs
88 .iowait_sleeptime: 0 nsecs
89 .last_jiffies : 0
90 .next_timer : 0
91 .idle_expires : 0 nsecs
92 jiffies: 112213
```

```
93 cpu: 1
94   clock 0:
95     .base:      5aade46d
96     .index:     0
97     .resolution: 1 nsecs
98     .get_time:  ktime_get
99     .offset:    0 nsecs
100  active timers:
101  #0: <79b94588>, tick_sched_timer, S:01
102  # expires at 1422140000000-1422140000000 nsecs [in 9588370 to 9588370
      nsecs]
103  #1: <8ccb83ac>, sched_rt_period_timer, S:01
104  # expires at 1423010000000-1423010000000 nsecs [in 879588370 to 879588370
      nsecs]
105  #2: <83ae4e14>, watchdog_timer_fn, S:01
106  # expires at 1424010000000-1424010000000 nsecs [in 1879588370 to
      1879588370 nsecs]
107  clock 1:
108     .base:      b35e28b8
109     .index:     1
110     .resolution: 1 nsecs
111     .get_time:  ktime_get_real
112     .offset:    0 nsecs
113  active timers:
114  clock 2:
115     .base:      b62ad788
116     .index:     2
117     .resolution: 1 nsecs
118     .get_time:  ktime_get_boottime
119     .offset:    0 nsecs
120  active timers:
121  clock 3:
122     .base:      18ab1cb4
123     .index:     3
124     .resolution: 1 nsecs
125     .get_time:  ktime_get_clocktai
126     .offset:    0 nsecs
127  active timers:
128  clock 4:
129     .base:      b0b0d2f7
130     .index:     4
131     .resolution: 1 nsecs
132     .get_time:  ktime_get
133     .offset:    0 nsecs
134  active timers:
135  clock 5:
136     .base:      05308a4e
137     .index:     5
138     .resolution: 1 nsecs
139     .get_time:  ktime_get_real
140     .offset:    0 nsecs
141  active timers:
142  clock 6:
143     .base:      742ba17c
144     .index:     6
145     .resolution: 1 nsecs
146     .get_time:  ktime_get_boottime
147     .offset:    0 nsecs
```



```
148 active timers:
149 clock 7:
150   .base:          f4009b5a
151   .index:         7
152   .resolution:    1 nsecs
153   .get_time:      ktime_get_clocktai
154   .offset:        0 nsecs
155 active timers:
156   .expires_next   : 1422140000000 nsecs
157   .hres_active    : 1
158   .nr_events      : 145643
159   .nr_retries     : 1
160   .nr_hangs       : 0
161   .max_hang_time  : 0
162   .nohz_mode      : 0
163   .last_tick      : 0 nsecs
164   .tick_stopped   : 0
165   .idle_jiffies   : 0
166   .idle_calls     : 0
167   .idle_sleeps    : 0
168   .idle_entrytime : 0 nsecs
169   .idle_waketime  : 0 nsecs
170   .idle_exittime  : 0 nsecs
171   .idle_sleeptime : 0 nsecs
172   .iowait_sleeptime: 0 nsecs
173   .last_jiffies   : 0
174   .next_timer     : 0
175   .idle_expires   : 0 nsecs
176 jiffies: 112213
177
178 Tick Device: mode:      1
179 Broadcast device
180 Clock Event Device: timer
181   max_delta_ns:  21474836451
182   min_delta_ns:  50000
183   mult:          429496730
184   shift:         32
185   mode:          1
186   next_event:    9223372036854775807 nsecs
187   set_next_event: apbt_next_event
188   shutdown:      apbt_shutdown
189   periodic:      apbt_set_periodic
190   oneshot:       apbt_set_oneshot
191   oneshot stopped: apbt_shutdown
192   resume:        apbt_resume
193   event_handler: tick_handle_oneshot_broadcast
194
195   retries:       0
196
197 tick_broadcast_mask: 0
198 tick_broadcast_oneshot_mask: 0
199
200 Tick Device: mode:      1
201 Per CPU device: 0
202 Clock Event Device: local_timer
203   max_delta_ns:  18572831531
204   min_delta_ns:  1000
205   mult:          496605594
```

```
206  shift:          31
207  mode:           3
208  next_event:    1422132143100 nsecs
209  set_next_event: twd_set_next_event
210  shutdown:      twd_shutdown
211  periodic:      twd_set_periodic
212  oneshot:       twd_set_oneshot
213  resume:        twd_shutdown
214  event_handler: hrtimer_interrupt
215
216  retries:       1
217  Wakeup Device: <NULL>
218
219  Tick Device: mode: 1
220  Per CPU device: 1
221  Clock Event Device: local_timer
222  max_delta_ns:  18572831531
223  min_delta_ns:  1000
224  mult:          496605594
225  shift:         31
226  mode:          3
227  next_event:    1422140000000 nsecs
228  set_next_event: twd_set_next_event
229  shutdown:      twd_shutdown
230  periodic:      twd_set_periodic
231  oneshot:       twd_set_oneshot
232  resume:        twd_shutdown
233  event_handler: hrtimer_interrupt
234
235  retries:       1
236  Wakeup Device: <NULL>
```

LISTING C.1: Output of /proc/timerlist on 6.6.14-rt21.

C.1.2 Cyclictest

Table C.1: Cyclictest results, no load.

Preemption Model	Kernel	Policy	Nice/Prio	Latency (μ s)		
				Min	Avg	Max
No Preemption	6.6.14	OTHER	0	30	60	142
		OTHER	-20	15	60	145
		FIFO	40	8	9	97
		FIFO	60	8	9	40
Preemptible Kernel (Low Latency Desktop)		OTHER	0	39	61	118
		OTHER	-20	19	62	134
		FIFO	40	9	10	46
		FIFO	60	9	10	138
Fully Preemptible Kernel (Real-Time)	6.6.14-rt21	OTHER	0	32	70	305
		OTHER	-20	36	71	323
		FIFO	40	8	9	115
		FIFO	60	8	9	92

Table C.2: Cyclictest results, Stress-NG 25 %.

Preemption Model	Kernel	Policy	Nice/Prio	Latency (μ s)		
				Min	Avg	Max
No Preemption	6.6.14	OTHER	0	18	67	11 442
		OTHER	-20	16	63	11 021
		FIFO	40	8	11	123
		FIFO	60	8	11	100
Preemptible Kernel (Low Latency Desktop)		OTHER	0	18	68	11 381
		OTHER	-20	18	63	11 023
		FIFO	40	9	12	105
		FIFO	60	9	11	200
Fully Preemptible Kernel (Real-Time)	6.6.14-rt21	OTHER	0	17	77	10 600
		OTHER	-20	18	73	11 358
		FIFO	40	8	11	128
		FIFO	60	8	11	51

Table C.3: Cyclicttest results, Stress-NG 75 %.

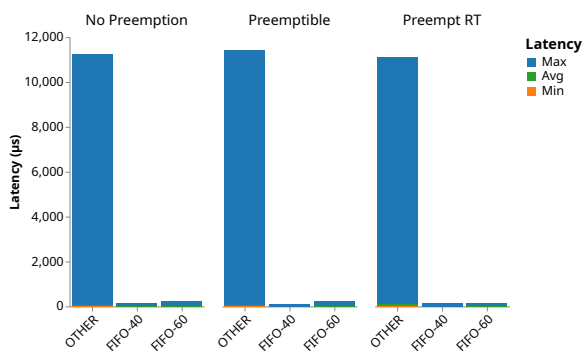
Preemption Model	Kernel	Policy	Nice/ Prio	Latency (μ s)		
				Min	Avg	Max
No Preemption	6.6.14	OTHER	0	18	81	11 217
		OTHER	-20	15	69	10 945
		FIFO	40	9	14	120
		FIFO	60	8	14	208
Preemptible Kernel (Low Latency Desktop)		OTHER	0	18	68	11 381
		OTHER	-20	18	63	11 023
		FIFO	40	9	12	105
		FIFO	60	9	11	200
Fully Preemptible Kernel (Real-Time)	6.6.14-rt21	OTHER	0	17	90	11 073
		OTHER	-20	18	80	10 997
		FIFO	40	8	14	142
		FIFO	60	8	14	114

Table C.4: Cyclicttest results with Iperf load.

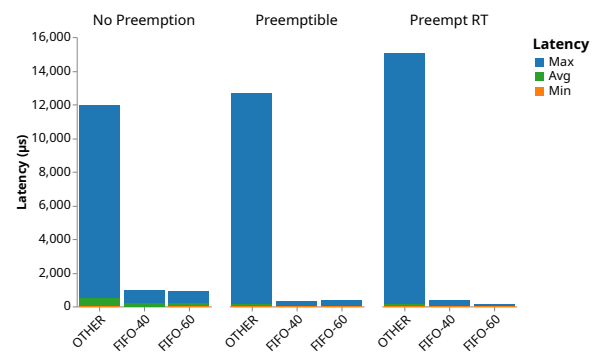
Preemption Model	Kernel	Policy	Nice/ Prio	Latency (μ s)		
				Min	Avg	Max
No Preemption	6.6.14	OTHER	0	23	531	11 966
		OTHER	-20	26	376	11 546
		FIFO	40	7	213	922
		FIFO	60	5	230	902
Preemptible Kernel (Low Latency Desktop)		OTHER	0	35	168	12 613
		OTHER	-20	31	106	12 264
		FIFO	40	19	54	277
		FIFO	60	20	52	345
Fully Preemptible Kernel (Real-Time)	6.6.14-rt21	OTHER	0	27	180	15 017
		OTHER	-20	23	139	12 718
		FIFO	40	16	54	351
		FIFO	60	13	54	116

Table C.5: Cyclictest results, 800 kernel.

Preemption Model	KMemleak	Latency (μs)		
		Min	Avg	Max
No Preemption	Disabled	11	14	158
	Enabled	10	13	39 771
Preemptible Kernel (Low Latency Desktop)	Enabled	10	14	354
Fully Preemptible Kernel (Real-Time)	Enabled	11	14	352



(a) Stress-NG 75 %.



(b) Iperf.

FIGURE C.1: Cyclictest results with Stress-NG 75 % and Iperf.

C.1.3 Throughput Test

Table C.6: Throughput test initial results.

Workload	Kernel	Preemption Mode	Run time (s)		
			usr	sys	total
Stress-NG	6.6.14	No Preemption	664.88	62.79	727.78
		Preemptible Kernel (Low Latency Desktop)	664.53	68.62	733.24
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	644.75	77.53	743.11
Hackbench	6.6.14	No Preemption	74.77	1515.95	1590.72
		Preemptible Kernel (Low Latency Desktop)	79.07	1553.90	1632.97
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	110.90	3146.64	3257.54

Table C.7: Throughput results, Hackbench

Workload	Kernel	Preemption Mode	Run time (s)		
			usr	sys	total
Unix Sockets	6.6.14	No Preemption	74.77	1515.95	1590.72
		Preemptible Kernel (Low Latency Desktop)	79.07	1553.90	1632.97
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	110.9	3146.64	3257.54

Continued on next page

Table C.7: Throughput results, Hackbench (Continued)

Pipes	6.6.14	No Preemption	64.09	1007.35	1071.44
		Preemptible Kernel (Low Latency Desktop)	75.91	1263.89	1339.80
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	238.88	3146.64	3385.52

Table C.8: Throughput results, Hackbench SCHED_FIFO

Workload	Kernel	Preemption Mode	Run time (s)		
			usr	sys	total
Unix Sockets	6.6.14	No Preemption	7.52	145.09	152.61
		Preemptible Kernel (Low Latency Desktop)	7.33	149.01	156.34
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	7.41	170.92	178.33
Pipes	6.6.14	No Preemption	3.37	99.28	102.65
		Preemptible Kernel (Low Latency Desktop)	8.29	99.32	107.61
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	55.13	2623.17	2678.30

Table C.9: Throughput test Iperf and p7zip.

Workload	Kernel	Preemption Mode	Run time (s)		
			usr	sys	total
Iperf	6.6.14	No Preemption	2.57	181.13	183.70
		Preemptible Kernel (Low Latency Desktop)	2.50	186.01	188.51
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	3.67	224.52	228.19
p7zip	6.6.14	No Preemption	543.99	3.41	547.40
		Preemptible Kernel (Low Latency Desktop)	544.61	3.79	548.40
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	545.10	3.76	548.86

C.1.4 Preemption Test

Table C.10: Preemption test results.

Preemption Model	Kernel	Policy	Nice/ Prio	Context Switches		Run- Time (ms)
				Vol	Invol	
No Preemption	6.6.14	OTHER	0	600	1467	600 542
		OTHER	-20	600	1017	600 543
		FIFO	40	600	0	600 513
		FIFO	60	600	0	600 513
Preemptible Kernel (Low Latency Desktop)		OTHER	0	600	582	600 543
		OTHER	-20	600	574	600 543
		FIFO	40	600	0	600 513
		FIFO	60	600	0	600 513
Fully Preemptible Kernel (Real-Time)	6.6.14- rt21	OTHER	0	600	7669	600 552
		OTHER	-20	600	8300	600 513
		FIFO	40	600	2	60 013
		FIFO	60	600	0	600 513

Table C.11: Preemption test results, with load.

Preemption Model	Kernel	Policy	Nice/ Prio	Context Switches		Run- Time (ms)
				Vol	Invol	
No Preemption	6.6.14	OTHER	0	600	13 449	604 359
		OTHER	-20	600	1678	600 923
		FIFO	40	600	4	600 656
		FIFO	60	600	1	600 654
Preemptible Kernel (Low Latency Desktop)		OTHER	0	600	13 775	602 260
		OTHER	-20	600	1480	600 963
		FIFO	40	600	2	600 567
		FIFO	60	600	1	600 570
Fully Preemptible Kernel (Real-Time)	6.6.14- rt21	OTHER	0	600	15 038	602 863
		OTHER	-20	600	7637	600 769
		FIFO	40	600	27	600 571
		FIFO	60	600	24	600 571

Table C.12: Migrations detected during throughput test

Preemption Model	Kernel	Migrations	
		<i>PreemptionTest</i>	<i>Iperf</i>
No Preemption	6.6.14	0	320
Preemptible Kernel (Low Latency Desktop)		0	114
Fully Preemptible Kernel (Real-Time)	6.6.14-rt21	20	101

C.1.5 Memory Lock Test

Table C.13: Page fault test results.

Memory Type	Kernel	Preemption Model	Latency ns	
			Avg	Max
Stack	6.6.14	No Preemption	6.53	203
		Preemptible Kernel (Low Latency Desktop)	7.40	154
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	5.56	167
Static	6.6.14	No Preemption	6.48	154
		Preemptible Kernel (Low Latency Desktop)	7.30	172
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	5.50	172
Heap	6.6.14	No Preemption	6.49	185
		Preemptible Kernel (Low Latency Desktop)	7.40	367
	6.6.14-rt21	Fully Preemptible Kernel (Real-Time)	5.50	156

C.2 Target System Tests

C.2.1 Dynamic Allocation

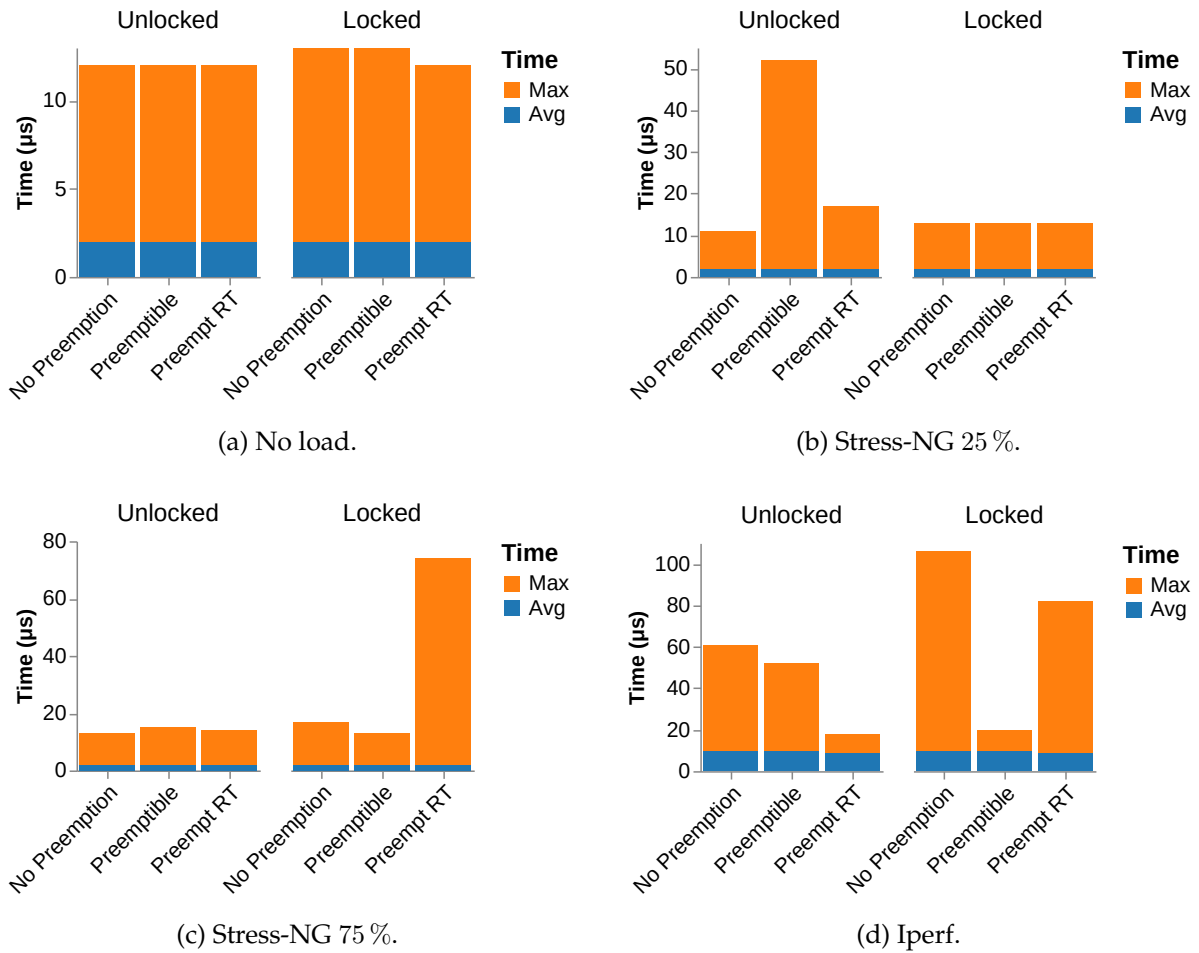


FIGURE C.2: Duration of writing to dynamically allocated memory

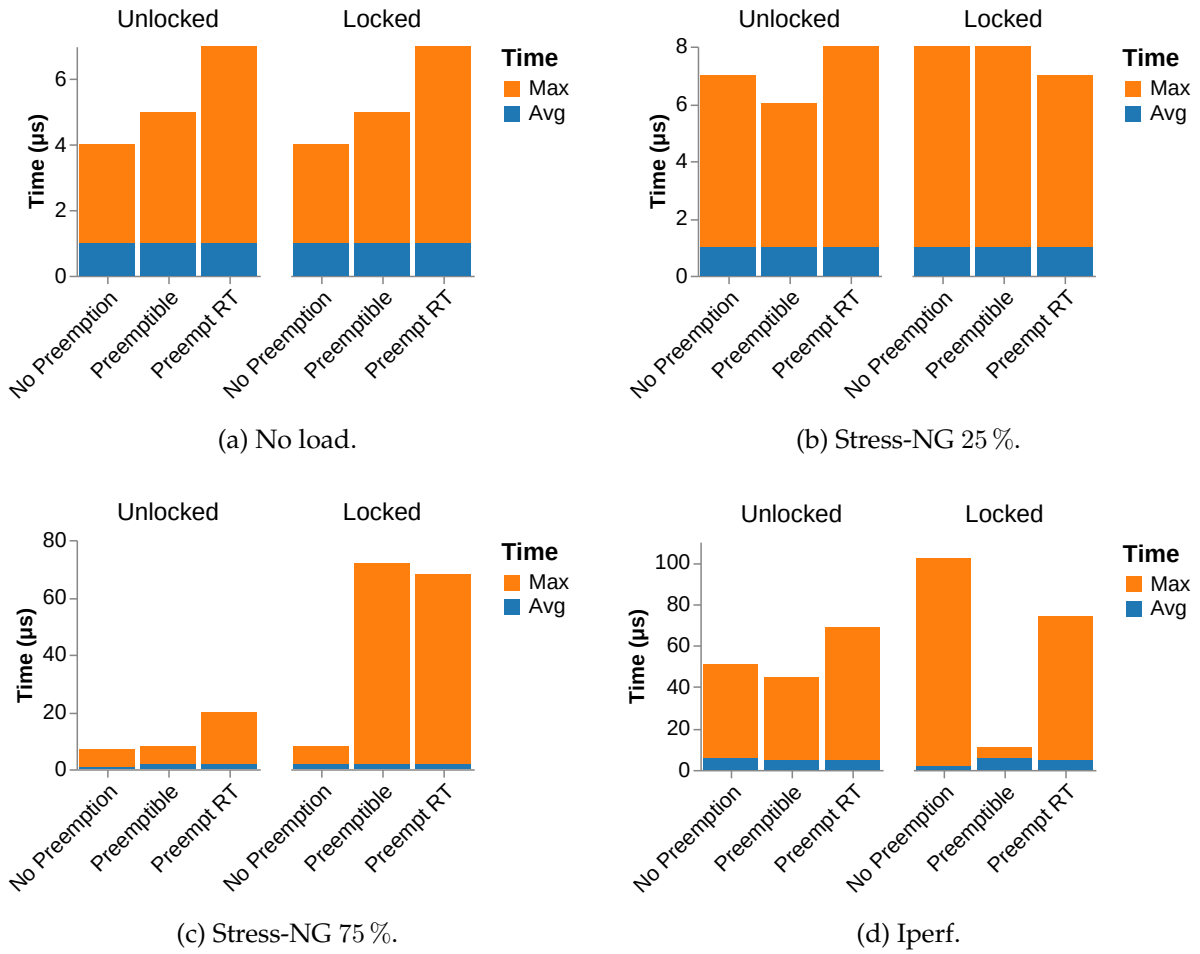


FIGURE C.3: Duration of freeing dynamically allocated memory

C.2.2 Periodic Thread Mechanisms

Table C.14: Periodic execution test results.

Kernel	Ext. Load	Mechanism	CPU Load (%)	Hist. Overflows	Latency (μ s)		
					Min	Avg	Max
6.6.14	25%	Setitimer	4.20	116	-998	10.52	11 577
		Nanosleep	1.37	83	-984	0.04	9875
		Setitimer SCHED_FIFO on Thread	4.39	85	-998	8.88	11 308
		CV with timeout	2.43	82	-972	0.03	10 071
		Setitimer SCHED_FIFO on Thread & Signal	3.92	0	-139	0.11	164
		Nanosleep SCHED_FIFO	1.39	0	-37	0.06	67
		SCHED_DEADLINE	1.00	0	-146	0.06	139
		CV with timeout SCHED_FIFO	2.47	0	-165	0.05	178

Continued on next page

Table C.14: Periodic execution test results. (Continued)

6.6.14	75%	Setitimer	5.00	485	-998	53.85	13 335
		Nanosleep	1.91	207	-984	0.056	13 861
		Setitimer SCHED_FIFO on Thread	4.91	319	-994	30.81	10 713
		CV with timeout	3.14	275	-972	0.06	11 673
		Setitimer SCHED_FIFO on Thread & Signal	4.96	0	-157	0.08	168
		Nanosleep SCHED_FIFO	1.91	0	-142	0.05	130
		SCHED_DEADLINE	2.00	0	-71	0.05	67
		CV with timeout SCHED_FIFO	2.47	0	-165	0.05	178

Continued on next page

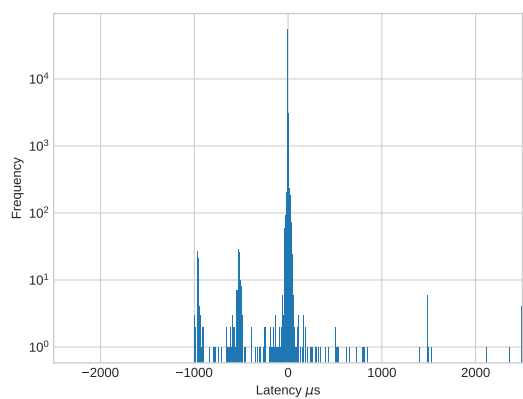
Table C.14: Periodic execution test results. (Continued)

6.6.14-rt21	25 %	Setitimer	5.86	82	-998	6.73	17 716
		Nanosleep	2.25	85	-978	0.04	9800
		Setitimer SCHED_FIFO on Thread	6.38	69	-992	7.01	10 247
		CV with timeout	2.81	93	-973	0.47	9900
		Setitimer SCHED_FIFO on Thread & Signal	6.02	0	-123	0.08	150
		Nanosleep SCHED_FIFO	2.11	0	-179	0.05	175
		SCHED_DEADLINE	1.02	0	-123	0.05	138
		CV with timeout SCHED_FIFO	2.69	0	-171	0.05	202

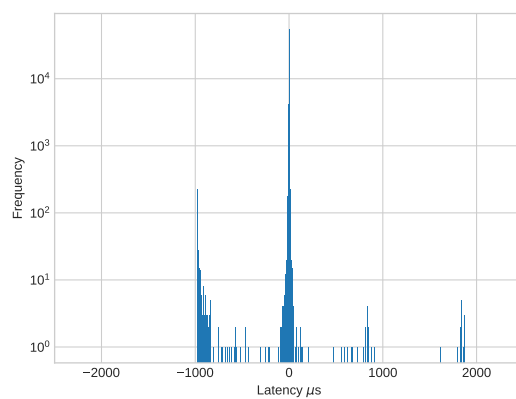
Continued on next page

Table C.14: Periodic execution test results. (Continued)

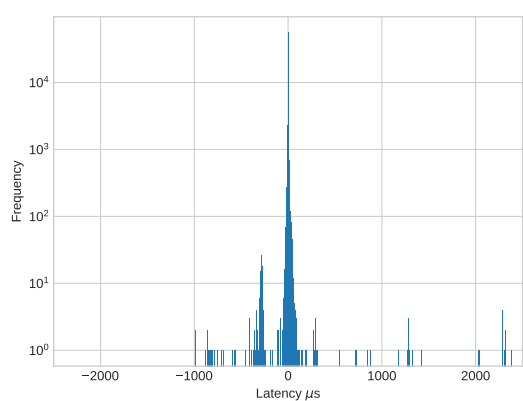
6.6.14-rt21	75 %	Setitimer	6.73	604	-998	58.04	20 529
		Nanosleep	2.79	260	-978	0.04	12 377
		Setitimer SCHED_FIFO on Thread	6.67	345	-901	38.16	11 200
		CV with timeout	3.24	285	-972	0.05	10 861
		Setitimer SCHED_FIFO on Thread & Signal	6.79	0	-172	0.06	174
		Nanosleep SCHED_FIFO	2.58	0	-159	0.04	154
		SCHED_DEADLINE	1.41	0	-88	0.048	99
		CV with timeout SCHED_FIFO	3.18	0	-96	0.46	109



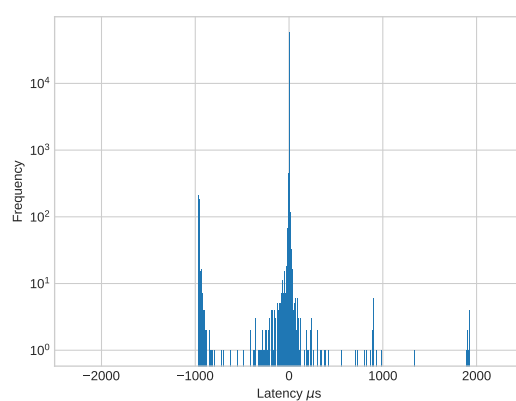
(a) Setitimer.



(b) Nanosleep.

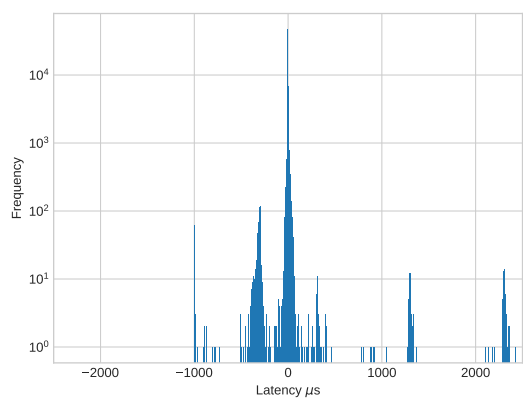


(c) Setitimer SCHED_FIFO on thread.

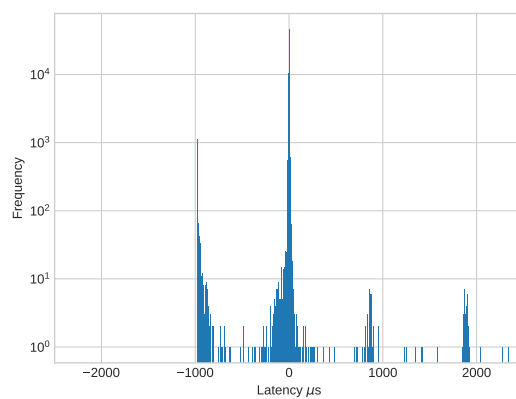


(d) CV with timeout.

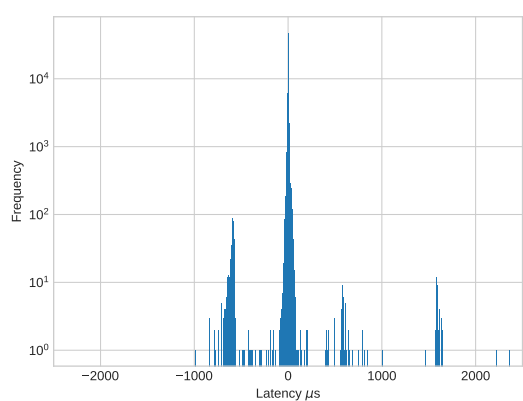
FIGURE C.4: Latency distribution of periodic mechanisms with SCHED_OTHER and 25% load.



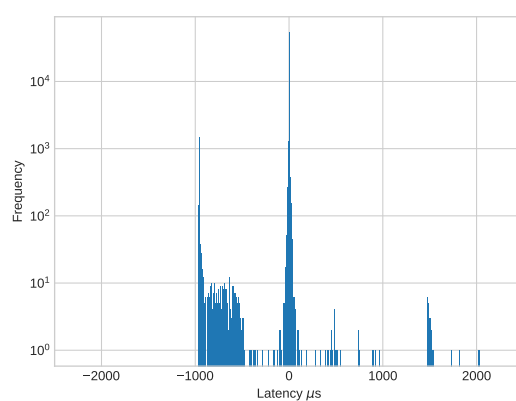
(a) Setitimer.



(b) Nanosleep.

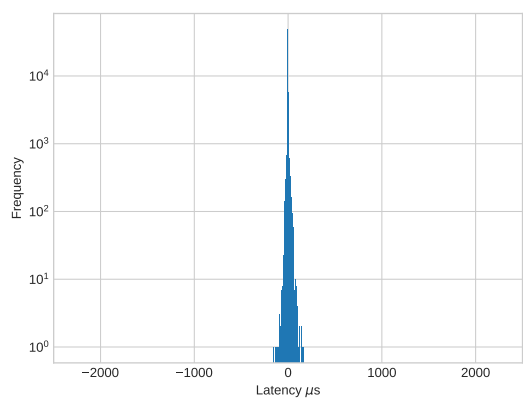


(c) Setitimer SCHED_FIFO on thread.

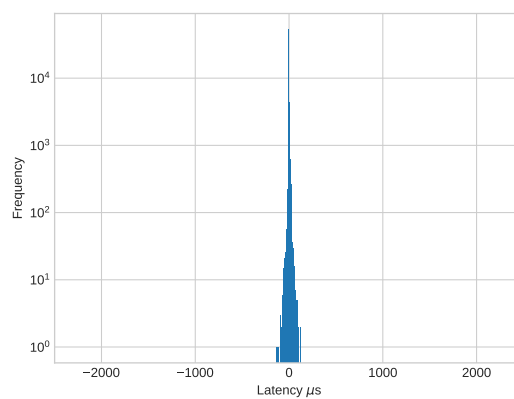


(d) CV with timeout.

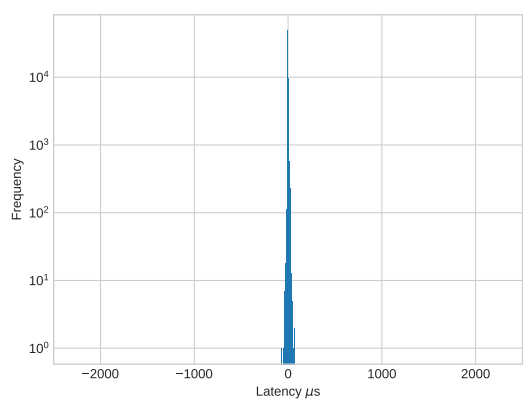
FIGURE C.5: Latency distribution of periodic mechanisms with SCHED_OTHER and 75% load.



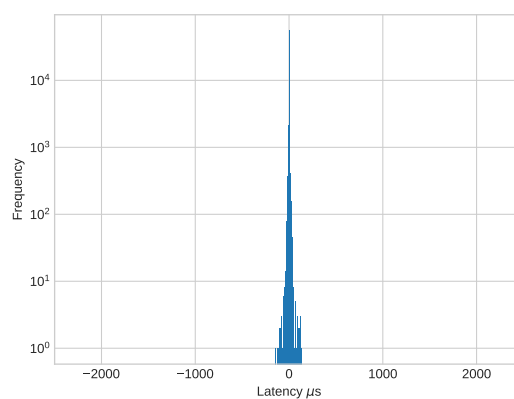
(a) Settimer.



(b) Nanosleep.

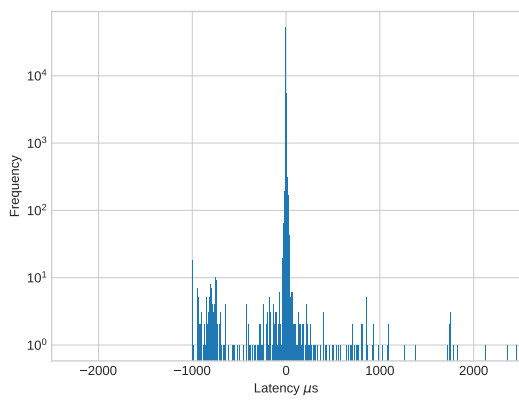


(c) SCHED_DEADLINE.

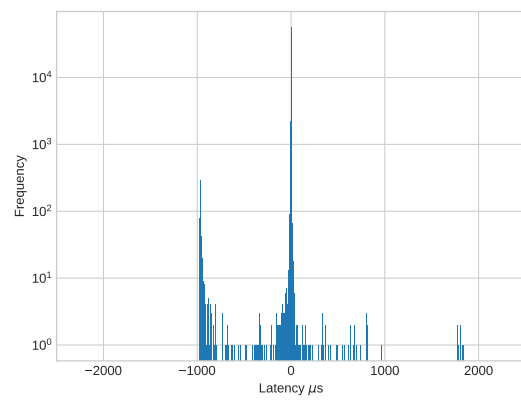


(d) CV with timeout.

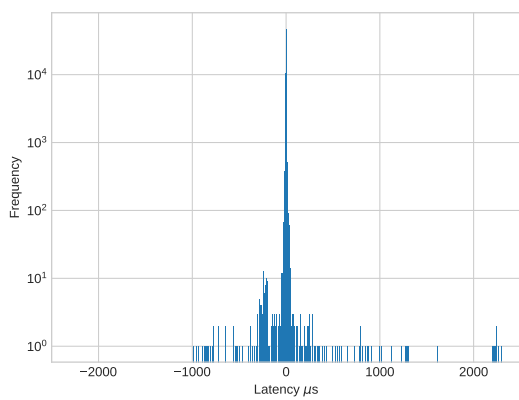
FIGURE C.6: Latency distribution of periodic mechanisms with SCHED_FIFO and 75 % load.



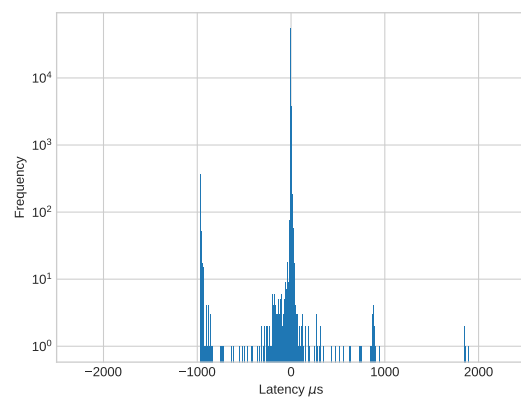
(a) Setitimer.



(b) Nanosleep.

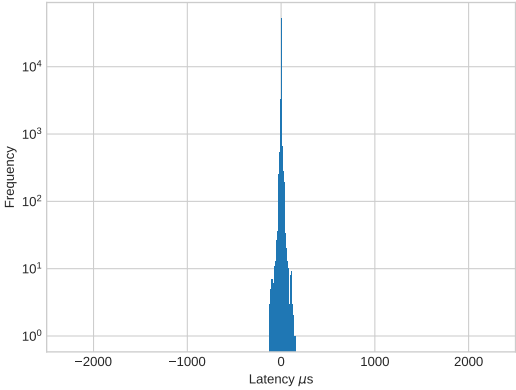


(c) Setitimer SCHED_FIFO on thread.

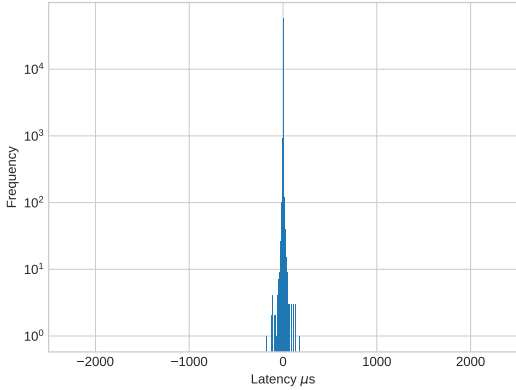


(d) CV with timeout.

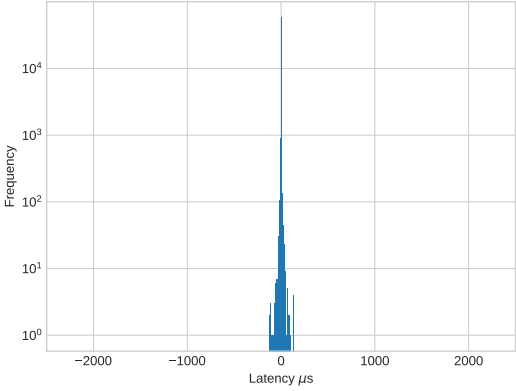
FIGURE C.7: Latency distribution of periodic mechanisms with SCHED_OTHER, 25% load and PREEMPT_RT.



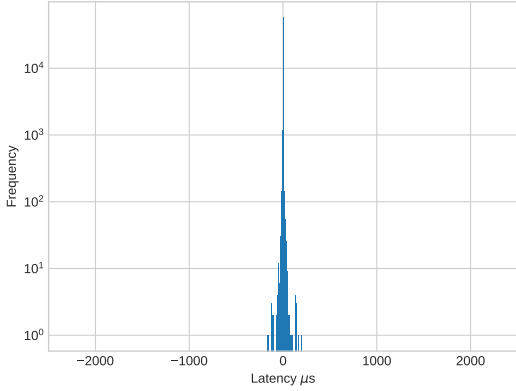
(a) Settimer.



(b) Nanosleep.

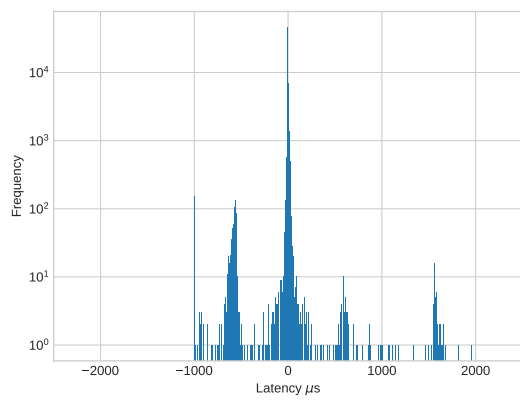


(c) SCHED_DEADLINE.

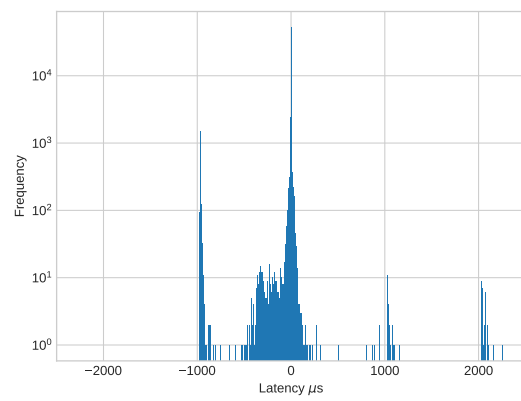


(d) CV with timeout.

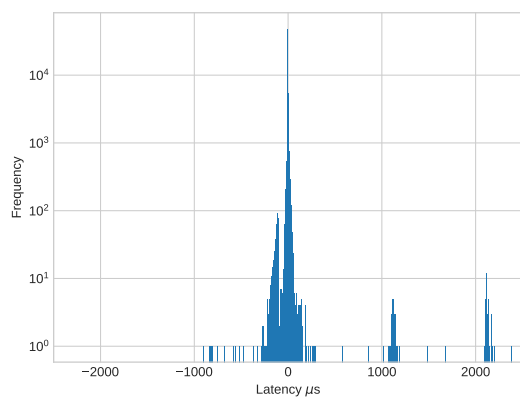
FIGURE C.8: Latency distribution of periodic mechanisms with SCHED_FIFO, 25 % load and PREEMPT_RT.



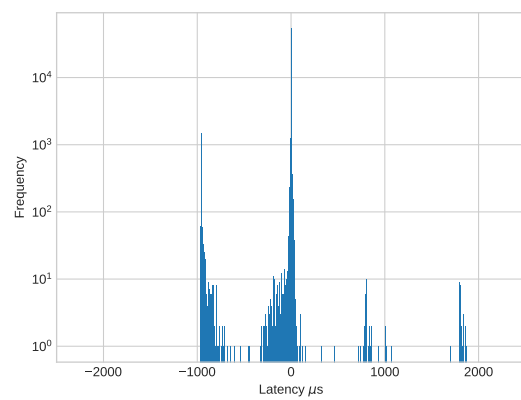
(a) Setitimer.



(b) Nanosleep.

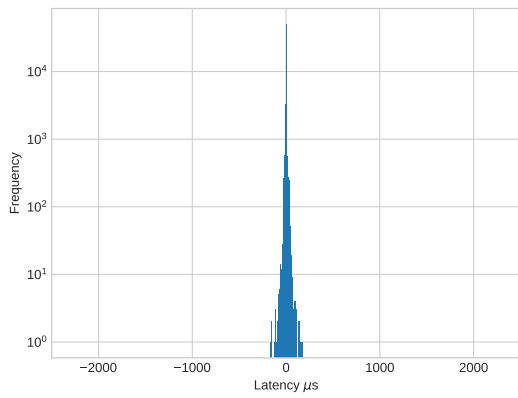


(c) Setitimer SCHED_FIFO on thread.

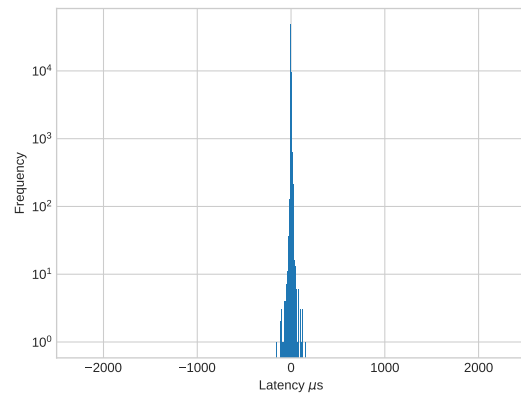


(d) CV with timeout.

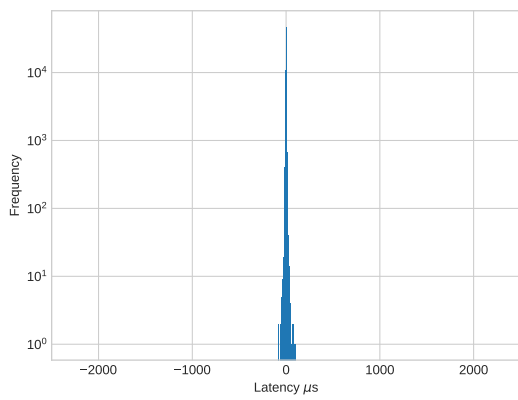
FIGURE C.9: Latency distribution of periodic mechanisms with SCHED_OTHER, 75% load and PREEMPT_RT.



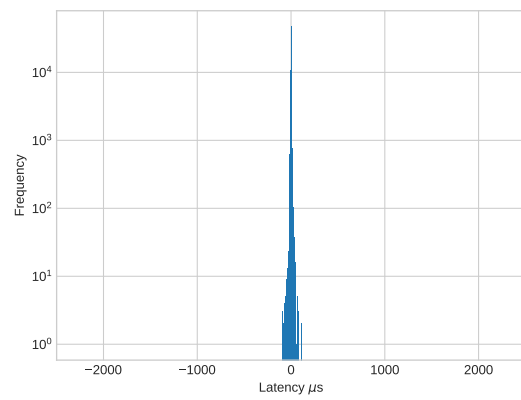
(a) Settimer SCHED_FIFO on thread and signal.



(b) Nanosleep.



(c) SCHED_DEADLINE.



(d) CV with timeout

FIGURE C.10: Latency distribution of periodic mechanisms with SCHED_FIFO, 75 % load and PREEMPT_RT.

C.2.3 Shared Memory Test

Table C.15: Shared memory test results, 4 KiB packets.

Preemption Model	Policy	Schedule	Latency (μs)		Exec. Time (μs)		Invol. Pre empt	Max Buf. Size (KiB)	CPU Load (%)
			Avg	Max	Avg	Max			
No Preemption	OTHER	Periodic	66	589	134	532	1	12.32	7.59
		Aperiodic	66	91	184	257	0	10.70	7.54
	FIFO	Periodic	16	50	135	240	0	10.54	7.69
		Aperiodic	15	41	179	239	0	10.58	7.54
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	67	985	135	348	6	18.95	7.74
		Aperiodic	67	95	184	267	0	10.71	7.58
	FIFO	Periodic	15	153	135	334	0	10.55	7.67
		Aperiodic	16	50	179	242	0	10.57	7.59
Fully Pre-emptible Kernel (Real-Time)	OTHER	Periodic	81	1296	135	389	123	15.82	7.61
		Aperiodic	82	830	187	990	1756	12.40	7.50
	FIFO	Periodic	15	35	134	225	0	10.49	7.59
		Aperiodic	15	56	179	259	1	10.61	7.53

Table C.16: Shared memory test results, 4 KiB packets Stress-NG 25 %.

Preemption Model	Policy	Schedule	Latency (μ s)		Exec. Time (μ s)		Invol. Pre empt	Max Buf. Size (KiB)	CPU Load (%)
			Avg	Max	Avg	Max			
No Pre-emption	OTHER	Periodic	71	10 951	135	1038	10	35.84	7.63
		Aperiodic	73	11 533	185	1837	114	38.50	7.52
	FIFO	Periodic	16	137	136	350	1	10.71	8.04
		Aperiodic	16	162	180	374	3	10.75	7.67
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	74	10 643	167	1039	27	36.83	7.74
		Aperiodic	76	11 217	187	1812	119	37.93	7.61
	FIFO	Periodic	17	76	136	259	0	10.55	7.83
		Aperiodic	17	160	181	362	1	10.72	7.71
Fully Pre-emptible Kernel (Real-Time)	OTHER	Periodic	86	11 188	138	10 309	10 014	56.43	7.73
		Aperiodic	89	11 462	189	11 905	2063	37.50	7.52
	FIFO	Periodic	16	78	135	245	0	10.56	7.74
		Aperiodic	17	100	180	293	4	10.68	7.67

Table C.17: Shared memory test results, 4 KiB packets Stress-NG 75 %.

Preemption Model	Policy	Schedule	Latency (μ s)		Exec. Time (μ s)		Invol. Pre empt	Max Buf. Size (KiB)	CPU Load (%)
			Avg	Max	Avg	Max			
No Pre-emption	OTHER	Periodic	199	10 776	154	1048	79	51.88	7.92
		Aperiodic	231	11 637	240	11 921	621	40.07	7.74
	FIFO	Periodic	18	142	136	255	4	10.71	7.95
		Aperiodic	18	174	182	334	6	10.80	7.84
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	208	11 144	156	9926	224	46.85	7.97
		Aperiodic	232	14 588	240	11 672	651	47.20	7.77
	FIFO	Periodic	19	200	138	439	5	10.56	8.12
		Aperiodic	19	137	182	329	5	10.66	7.91
Fully Pre-emptible Kernel (Real-Time)	OTHER	Periodic	211	11 852	153	9437	413	39.78	7.77
		Aperiodic	235	13 630	238	12 332	2681	42.04	7.57
	FIFO	Periodic	18	82	138	258	4	10.62	7.97
		Aperiodic	20	102	182	306	7	10.70	7.88

Table C.18: Shared memory test results, 4 KiB packets with Iperf.

Preemption Model	Policy	Schedule	Latency (μ s)		Exec. Time (μ s)		Invol. Pre-empt	Max Buf. Size (KiB)	CPU Load (%)
			Avg	Max	Avg	Max			
No Pre-emption	OTHER	Periodic	309	2235	150	1267	4368	28.26	9.09
		Aperiodic	177	2045	219	1748	23 407	14.62	8.89
	FIFO	Periodic	20	151	137	286	373	10.66	8.09
		Aperiodic	22	160	159	353	381	11.58	7.55
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	244	2329	150	2359	6328	38.78	9.07
		Aperiodic	171	1927	219	2796	22 348	13.99	8.50
	FIFO	Periodic	20	116	140	294	226	10.54	8.30
		Aperiodic	23	169	189	408	6002	10.79	8.41
Fully Pre-emptible Kernel (Real-Time)	OTHER	Periodic	253	4614	172	3322	49 716	48.34	8.48
		Aperiodic	368	3808	293	3667	112 715	18.83	8.18
	FIFO	Periodic	28	125	142	295	10 958	10.64	8.79
		Aperiodic	30	126	194	342	7058	10.78	8.76

Table C.19: Shared memory test results, 512 B packets.

Preemption Model	Policy	Schedule	Latency (μ s)		Exec. Time (μ s)		Invol. Pre empt	Max Buf. Size (KiB)	CPU Load (%)
			Avg	Max	Avg	Max			
No Preemption	OTHER	Periodic	65	971	370	1674	1	10.96	19.39
		Aperiodic	66	88	390	757	1	7.29	19.37
	FIFO	Periodic	14	103	372	657	0	6.66	19.43
		Aperiodic	14	39	375	694	0	7.16	19.38
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	66	1106	373	2263	65	13.98	19.57
		Aperiodic	67	201	390	872	4	7.55	19.42
	FIFO	Periodic	15	127	371	640	0	6.39	19.43
		Aperiodic	15	140	375	867	0	7.34	19.40
Fully Pre-emptible Kernel (Real-Time)	OTHER	Periodic	80	725	371	2579	163	15.21	19.36
		Aperiodic	81	811	395	1875	4840	9.03	19.33
	FIFO	Periodic	14	33	372	480	0	6.34	19.44
		Aperiodic	15	57	373	707	6	7.20	19.37

Table C.20: Shared memory test results, 512 B packets with Stress-NG 25 %.

Preemption Model	Policy	Schedule	Latency (μ s)		Exec. Time (μ s)		Invol. Pre empt	Max Buf. Size (KiB)	CPU Load (%)
			Avg	Max	Avg	Max			
No Pre-emption	OTHER	Periodic	74	11 014	375	6737	34	34.19	19.40
		Aperiodic	70	12 656	397	8184	37	39.38	19.40
	FIFO	Periodic	15	137	372	623	68	6.75	19.47
		Aperiodic	16	131	375	842	13	7.38	19.45
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	69	10 576	372	6582	2	33.61	19.50
		Aperiodic	72	11 143	400	7248	31	35.46	19.44
	FIFO	Periodic	16	124	373	675	34	6.51	19.63
		Aperiodic	16	96	375	888	11	7.30	19.47
Fully Pre-emptible Kernel (Real-Time)	OTHER	Periodic	102	13 922	383	16 328	532	41.29	19.36
		Aperiodic	102	12 169	437	16 788	6544	37.97	19.34
	FIFO	Periodic	15	70	371	567	5	6.45	19.52
		Aperiodic	16	96	375	811	16	7.27	19.44

Table C.21: Shared memory test results, 512 B packets with Stress-NG 75 %.

Preemption Model	Policy	Schedule	Latency (μ s)		Exec. Time (μ s)		Invol. Pre empt	Max Buf. Size (KiB)	CPU Load (%)
			Avg	Max	Avg	Max			
No Pre-emption	OTHER	Periodic	183	13 799	437	10 471	255	41.19	19.36
		Aperiodic	278	14 850	767	18 158	754	44.91	19.34
	FIFO	Periodic	17	179	373	618	15	6.76	19.73
		Aperiodic	18	167	377	837	7	7.45	19.56
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	154	14 483	421	16 434	466	41.68	19.47
		Aperiodic	268	18 366	754	27 612	700	54.05	19.40
	FIFO	Periodic	18	95	371	511	9	6.43	19.70
		Aperiodic	19	115	377	816	12	7.36	19.61
Fully Pre-emptible Kernel (Real-Time)	OTHER	Periodic	135	14 458	406	16 488	15 571	42.62	19.34
		Aperiodic	203	14 166	644	18 051	6391	43.26	19.34
	FIFO	Periodic	17	82	371	517	6	6.47	19.64
		Aperiodic	18	96	377	822	32	7.33	19.58

Table C.22: Shared memory test results, 512 B packets with Iperf.

Preemption Model	Policy	Schedule	Latency (μ s)		Exec. Time (μ s)		Invol. Pre empt	Max Buf. Size (KiB)	CPU Load (%)
			Avg	Max	Avg	Max			
No Pre-emption	OTHER	Periodic	108	3634	450	4226	263 461	15.25	20.43
		Aperiodic	66	670	389	961	0	8.53	19.35
	FIFO	Periodic	20	199	375	784	801	6.73	19.95
		Aperiodic	22	139	382	828	524	7.42	19.89
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	85	3520	408	4549	101 567	15.04	20.50
		Aperiodic	121	3564	520	4971	151 719	16.04	19.92
	FIFO	Periodic	19	119	375	675	234	6.59	20.00
		Aperiodic	22	119	384	869	6158	7.38	19.94
Fully Pre-emptible Kernel (Real-Time)	OTHER	Periodic	136	7156	417	12 826	53 636	22.90	19.75
		Aperiodic	252	5987	734	13 306	131 000	23.51	19.66
	FIFO	Periodic	21	123	374	567	77	6.60	20.07
		Aperiodic	25	133	385	818	63	7.37	19.98

C.2.4 Event-Driven vs Polled Kthread

Table C.23: Event-Driven vs polled kernel worker.

Event Frequency	Scheme	CPU Load (%)		
		US Worker	KS Worker	Total
0	Polled	—	3.63	3.63
	Event-Driven	—	0.00	0.00
1 Hz	Polled	0.02	3.72	3.74
	Event-Driven	0.02	0.00	0.02
10 Hz	Polled	1.04	3.79	4.83
	Event-Driven	0.88	0.21	1.09
1 kHz	Polled	8.71	6.13	14.84
	Event-Driven	8.70	1.78	10.48
10 kHz	Polled	28.89	11.56	40.45
	Event-Driven	43.35	6.86	50.21

Table C.24: Event-driven vs polled kernel worker, with Stress-NG 25 %.

Event Frequency	Scheme	CPU Load (%)		
		US Worker	KS Worker	Total
0	Polled	—	3.72	3.72
	Event-Driven	—	0.00	0.00
1 Hz	Polled	0.03	3.81	3.84
	Event-Driven	0.03	0.00	0.03
10 Hz	Polled	1.12	3.79	4.91
	Event-Driven	0.97	0.21	1.18
1 kHz	Polled	8.76	6.35	15.11
	Event-Driven	8.69	1.91	10.60
10 kHz	Polled	28.48	12.11	40.59
	Event-Driven	37.92	8.67	46.59

C.2.5 Generic Netlink Test

Table C.25: Generic Netlink test results with SCHED_FIFO on kernel thread, no load.

Preemption Model	Ask Time (μ s)		Read Time (μ s)		Write Time (μ s)	
	Avg	Max	Avg	Max	Avg	Max
No Preemption	29	80	68	219	59	74
Preemptible Kernel (Low Latency Desktop)	31	76	72	170	60	74
Fully Preemptible Kernel (Real-Time)	36	86	67	173	63	77

Table C.26: Generic Netlink test results with SCHED_FIFO on kernel thread, with Stress-NG 25 %.

Preemption Model	Ask Time (μ s)		Read Time (μ s)		Write Time (μ s)	
	Avg	Max	Avg	Max	Avg	Max
No Preemption	36	157	76	221	60	104
Preemptible Kernel (Low Latency Desktop)	38	152	79	216	62	92
Fully Preemptible Kernel (Real-Time)	43	150	76	187	65	95

Table C.27: Generic Netlink test results with SCHED_FIFO on kernel thread, with Stress-NG 75 %.

Preemption Model	Ask Time (μ s)		Read Time (μ s)		Write Time (μ s)	
	Avg	Max	Avg	Max	Avg	Max
No Preemption	49	169	92	213	64	90
Preemptible Kernel (Low Latency Desktop)	50	162	94	196	66	97

Continued on next page

Table C.27: Generic Netlink test results with SCHED_FIFO on kernel thread, with Stress-NG 75 %. (Continued)

Fully Preemptible Kernel (Real-Time)	57	171	89	177	68	97
--------------------------------------	----	-----	----	-----	----	----

Table C.28: Generic Netlink test results with SCHED_FIFO on kernel thread, with Iperf.

Preemption Model	Ask Time (μ s)		Read Time (μ s)		Write Time (μ s)	
	Avg	Max	Avg	Max	Avg	Max
No Preemption	74	118	140	220	72	92
Preemptible Kernel (Low Latency Desktop)	66	106	108	228	73	91
Fully Preemptible Kernel (Real-Time)	81	142	114	181	75	130

C.2.6 Full Test RX

Table C.29: Full test RX, no load.

Preemption Model	Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Over-runs	Max Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	427	551	22.50	0	7.88
		Aperiodic	678	1148	23.43	0	8.36
	FIFO	Periodic	427	546	22.47	0	7.98
		Aperiodic	653	1142	23.38	0	8.27
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	428	581	22.58	0	7.94
		Aperiodic	684	1183	23.56	0	8.39
	FIFO	Periodic	427	669	22.49	0	7.91
		Aperiodic	656	1149	23.47	0	8.23
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	434	1597	22.70	0	10.41
		Aperiodic	708	4048	23.78	0	11.71
	FIFO	Periodic	434	591	22.82	0	7.92
		Aperiodic	674	1195	23.86	0	8.27

Table C.30: Full test RX, with Stress-NG 25%.

Preemption Model	Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Over-runs	Max Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	434	1996	22.71	0	34.94
		Aperiodic	703	29 882	23.50	0	42.91
	FIFO	Periodic	431	707	22.79	0	8.06
		Aperiodic	658	1411	23.52	0	8.44
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	436	912	22.81	0	35.21
		Aperiodic	707	48 607	23.68	0	42.05
	FIFO	Periodic	658	432	22.82	0	8.00
		Aperiodic	676	1319	23.95	0	8.50

Continued on next page

Table C.30: Full test RX, with Stress-NG 25 %. (Continued)

Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	446	10 791	23.10	0	53.52
		Aperiodic	821	68 779	24.00	0	45.48
	FIFO	Periodic	438	658	23.16	0	8.03
		Aperiodic	686	1709	24.14	0	8.50

Table C.31: Full test RX, with Stress-NG 75 %.

Preemption Model	Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Over-runs	Max Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	452	10 526	22.97	0	55.79
		Aperiodic	831	48 836	23.67	0	46.20
	FIFO	Periodic	434	699	23.05	0	8.12
		Aperiodic	672	1639	23.88	0	8.52
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	459	13 184	23.24	0	56.18
		Aperiodic	810	49 334	23.78	0	45.72
	FIFO	Periodic	438	666	23.30	0	8.00
		Aperiodic	679	1682	24.05	0	8.52
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	455	20 883	23.32	0	55.32
		Aperiodic	870	69 061	24.24	0	57.74
	FIFO	Periodic	444	786	23.56	0	8.10
		Aperiodic	698	1711	24.44	0	8.50

Table C.32: Full test RX, with Iperf.

Preemption Model	Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Over-runs	Max Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	586	2807	25.54	0	19.45
		Aperiodic	826	11 102	25.92	0	24.46
	FIFO	Periodic	482	737	26.16	0	8.10
		Aperiodic	798	2018	26.55	0	8.60

Continued on next page

Table C.32: Full test RX, with Iperf. (Continued)

Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	558	3031	26.10	0	18.48
		Aperiodic	684	1238	23.57	0	8.40
	FIFO	Periodic	486	801	26.49	0	7.95
		Aperiodic	786	1792	26.27	0	8.46
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	498	3165	26.25	0	26.54
		Aperiodic	884	7114	26.80	0	20.66
	FIFO	Periodic	496	835	26.93	0	8.20
		Aperiodic	839	1760	27.26	0	8.59

Table C.33: Full test RX, increased overhead, no load.

Preemption Model	Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Over-runs	Max Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	762	1513	39.17	0	10.97
		Aperiodic	2139	4458	38.18	0	8.35
	FIFO	Periodic	759	982	38.99	0	9.34
		Aperiodic	2070	3560	38.20	0	8.19
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	760	955	39.15	0	11.95
		Aperiodic	2145	4567	38.21	0	8.41
	FIFO	Periodic	759	1033	39.04	0	9.16
		Aperiodic	2073	3610	38.21	0	8.24
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	767	2546	39.22	0	13.07
		Aperiodic	2210	8009	38.32	0	10.55
	FIFO	Periodic	762	923	39.17	0	7.84
		Aperiodic	2111	4463	38.40	0	8.27

Table C.34: Full test RX, increased overhead, with Stress-NG 25 %.

Preemption Model	Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Over-runs	Max Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	758	10 848	38.60	100	128.00
		Aperiodic	2994	424 286	37.89	40	128.00
	FIFO	Periodic	767	1149	39.49	0	13.89
		Aperiodic	2097	4797	38.35	0	8.54
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	764	2346	38.84	144	128.00
		Aperiodic	3245	335 640	37.89	120	128.00
	FIFO	Periodic	764	1031	39.44	0	16.09
		Aperiodic	2141	5709	38.59	0	8.58
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	791	11 002	38.81	232	128.00
		Aperiodic	6840	599 907	37.80	288	128.00
	FIFO	Periodic	770	1003	39.68	0	15.98
		Aperiodic	2144	5640	38.57	0	8.50

Table C.35: Full test RX, increased overhead, with Iperf.

Preemption Model	Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Over-runs	Max Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	795	2822	40.90	0	25.99
		Aperiodic	2498	23 321	39.61	0	22.17
	FIFO	Periodic	792	1094	41.28	0	17.14
		Aperiodic	2430	5588	39.88	0	8.54
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	809	2932	41.06	8	126.33
		Aperiodic	2493	23 353	39.61	0	22.05
	FIFO	Periodic	798	2154	41.62	0	13.84
		Aperiodic	2432	9342	39.89	0	9.82
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	969	4703	21.81	23 424	128.00
		Aperiodic	3631	428 014	40.08	72	128.00
	FIFO	Periodic	841	1113	41.71	8	128.00
		Aperiodic	2700	5798	40.80	0	8.73

C.2.7 Full Test TX

Table C.36: Full test TX, Setup 1, no load.

Preemption Model	App Policy	Schedule	Exec. Time		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	442 μ s	3.01 ms	17.87	0	55.12
		Aperiodic	300 s	600 s	49.39	0	60.55
	FIFO	Periodic	434 μ s	2.97 ms	17.74	0	54.85
		Aperiodic	474 ms	1.29 s	47.50	1198	0.00
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	442 μ s	3.02 ms	17.95	0	54.75
		Aperiodic	300 s	600 s	49.41	0	60.55
	FIFO	Periodic	435 μ s	2.88 ms	17.86	0	55.41
		Aperiodic	474 ms	1.36 s	47.50	1200	0.00
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	465 μ s	3.91 ms	18.14	0	53.32
		Aperiodic	300 s	600 s	48.90	0	57.21
	FIFO	Periodic	439 μ s	3.55 ms	18.06	0	55.79
		Aperiodic	1.20 s	5.98 s	29.22	641	0.00

Table C.37: Full test TX, Setup 1, with Stress-NG 25 %.

Preemption Model	App Policy	Schedule	Exec. Time		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	458 μ s	14.46 ms	18.10	0	21.73
		Aperiodic	841 ms	2.27 s	43.81	21	0.00
	FIFO	Periodic	441 μ s	2.50 ms	18.21	0	57.28
		Aperiodic	472 ms	986 ms	47.50	1190	0.00
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	458 μ s	13.7 ms	18.23	0	21.60
		Aperiodic	848 ms	2.29 s	43.85	28	0.00
	FIFO	Periodic	443 μ s	1.99 ms	18.28	0	57.19
		Aperiodic	473 ms	986 ms	47.50	1191	0.00

Continued on next page

Table C.37: Full test TX, Setup 1, with Stress-NG 25 %. (Continued)

Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	492 μ s	12.9 ms	18.47	0	22.40
		Aperiodic	1.69 s	22.9 s	44.97	0	10.75
	FIFO	Periodic	445 μ s	5.36 ms	18.46	0	57.29
		Aperiodic	1.17 s	7.98 s	48.26	640	0.00

Table C.38: Full test TX, Setup 1, with Stress-NG 75 %.

Preemption Model	App Policy	Schedule	Exec. Time		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	492 μ s	15.82 ms	18.48	0	14.48
		Aperiodic	126 ms	823 ms	33.85	42	0.00
	FIFO	Periodic	452 μ s	2.14 ms	18.85	0	57.09
		Aperiodic	476 ms	1.99 s	47.48	1198	0.00
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	483 μ s	14.5 ms	18.55	0	23.61
		Aperiodic	132 ms	956 ms	34.08	62	0.00
	FIFO	Periodic	453 μ s	6.29 ms	18.88	0	57.23
		Aperiodic	474 ms	1.96 s	47.08	1151	0.00
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	535 μ s	13.6 ms	18.65	0	27.60
		Aperiodic	131 ms	853 ms	34.03	0	61.13
	FIFO	Periodic	451 μ s	11.7 ms	18.88	0	35.84
		Aperiodic	1.46 s	7.98 s	48.43	546	0.00

Table C.39: Full test TX, Setup 1, with Iperf.

Preemption Model	App Policy	Schedule	Exec. Time		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	851 μ s	5.69 ms	19.86	0	48.30
		Aperiodic	292 s	600 s	26.13	0	46.17
	FIFO	Periodic	435 μ s	2.48 ms	17.78	0	57.60
		Aperiodic	474 ms	2.98 s	47.49	0	49.21

Continued on next page

Table C.39: Full test TX, Setup 1, with Iperf. (Continued)

Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	789 μ s	7.23 ms	20.22	0	49.21
		Aperiodic	273 s	573 s	25.77	0	40.00
	FIFO	Periodic	498 μ s	3.81 ms	21.07	0	56.66
		Aperiodic	582 ms	3.97 s	47.32	1040	0.00
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	677 μ s	4.72 ms	20.36	0	50.97
		Aperiodic	32.4 s	105 s	39.74	0	27.09
	FIFO	Periodic	491 μ s	5.33 ms	20.99	0	56.91
		Aperiodic	2.24 s	13 s	48.70	339	0.00

Table C.40: Full test TX, Setup 2, no load.

Preemption Model	kthread Policy	App Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
				Avg	Max			
No Preemption	OTHER	OTHER	Periodic	413	2998	17.18	0	49.28
			Aperiodic	2031	3731	28.14	0	53.48
		FIFO	Periodic	410	2998	17.11	0	50.93
			Aperiodic	2081	3636	29.20	0	56.17
	FIFO	OTHER	Periodic	427	2602	17.29	0	52.68
			Aperiodic	2018	6260	27.90	0	49.99
		FIFO	Periodic	408	2987	17.02	0	48.75
			Aperiodic	2083	7487	29.20	0	50.52
Preemptible Kernel (Low Latency Desktop)	OTHER	OTHER	Periodic	412	3048	17.21	0	48.87
			Aperiodic	2032	3770	28.14	0	53.26
		FIFO	Periodic	410	2950	17.14	0	49.30
			Aperiodic	2080	6382	29.20	0	50.58
	FIFO	OTHER	Periodic	428	2508	17.37	0	52.99
			Aperiodic	2017	7383	27.88	0	50.17
		FIFO	Periodic	410	2982	17.11	0	49.09
			Aperiodic	2082	6455	29.20	0	50.53

Continued on next page

Table C.40: Full test TX, Setup 2, no load. (Continued)

Fully Pre-emptible Kernel (Real-Time)	OTHER	OTHER	Periodic	421	2998	17.30	0	49.38
			Aperiodic	2022	3975	27.42	0	52.73
		FIFO	Periodic	414	1823	17.27	0	54.50
			Aperiodic	2076	6388	29.22	0	50.60
	FIFO	OTHER	Periodic	437	3166	17.41	0	52.12
			Aperiodic	2033	3970	27.51	0	52.55
		FIFO	Periodic	412	3127	17.21	0	49.16
			Aperiodic	2078	7492	29.20	0	50.62

Table C.41: Full test TX, Setup 2, with Stress-NG 25%.

Preemption Model	kthread Policy	App Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
				Avg	Max			
No Preemption	OTHER	OTHER	Periodic	437	27 900	17.47	110	0.00
			Aperiodic	2120	14 635	28.48	205	0.00
		FIFO	Periodic	417	3015	17.48	0	29.01
			Aperiodic	2096	41 807	29.10	43	0.00
	FIFO	OTHER	Periodic	443	33 606	17.52	21	0.00
			Aperiodic	2065	16 218	27.50	361	0.00
		FIFO	Periodic	415	4331	17.40	0	45.18
			Aperiodic	2123	10 095	29.27	0	44.41

Continued on next page

Table C.41: Full test TX, Setup 2, with Stress-NG 25 %. (Continued)

Preemptible Kernel (Low Latency Desktop)	OTHER	OTHER	Periodic	440	15 863	17.47	283	0.00
			Aperiodic	2228	13 178	29.37	312	0.00
		FIFO	Periodic	417	3009	17.53	0	37.09
			Aperiodic	2086	21 202	29.18	0	26.16
	FIFO	OTHER	Periodic	446	14 206	17.58	36	0.00
			Aperiodic	2067	15 093	27.70	282	0.00
		FIFO	Periodic	417	3112	17.51	0	48.52
			Aperiodic	4114	7466	29.25	0	50.03
Fully Pre-emptible Kernel (Real-Time)	OTHER	OTHER	Periodic	462	17 123	17.57	65	0.00
			Aperiodic	2277	13 617	28.03	507	0.00
		FIFO	Periodic	419	4393	17.60	0	30.11
			Aperiodic	2077	7483	29.19	0	21.94
	FIFO	OTHER	Periodic	468	13 199	17.67	22	0.00
			Aperiodic	2172	14 411	27.48	36	0.00
		FIFO	Periodic	419	3108	17.58	0	48.32
			Aperiodic	2079	7452	29.23	0	50.01

Table C.42: Full test TX, Setup 2, with Stress-NG 75 %.

Preemption Model	kthread Policy	App Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Under-runs	Min Buffer Size (KiB)	
				Avg	Max				
No Preemption	OTHER	OTHER	Periodic	469	14 523	17.81	1079	0.00	
			Aperiodic	2160	37 160	20.23	22 759	0.00	
		FIFO	Periodic	423	3035	17.87	97	0.00	
			Aperiodic	2141	41 873	28.86	257	0.00	
		FIFO	OTHER	Periodic	477	22 027	17.85	23	0.00
				Aperiodic	2217	49 381	22.89	7062	0.00
	FIFO		Periodic	421	10 203	17.81	0	35.79	
			Aperiodic	2247	10 075	29.45	0	44.30	
	Preemptible Kernel (Low Latency Desktop)	OTHER	OTHER	Periodic	471	35 384	17.76	882	0.00
				Aperiodic	2394	23 371	21.28	19 067	0.00
			FIFO	Periodic	424	13 066	17.96	0	5.32
				Aperiodic	2106	10 107	29.09	0	31.68
FIFO			OTHER	Periodic	474	28 015	17.94	6	0.00
				Aperiodic	2298	60 779	23.05	7554	0.00
		FIFO	Periodic	423	4533	17.91	0	43.40	
			Aperiodic	2251	10 097	29.45	0	44.25	

Continued on next page

Table C.42: Full test TX, Setup 2, with Stress-NG 75 %. (Continued)

Fully Pre-emptible Kernel (Real-Time)	OTHER	OTHER	Periodic	504	17 791	17.82	756	0.00
			Aperiodic	2502	49 793	19.96	20 686	0.00
		FIFO	Periodic	424	14 909	17.97	2	0.00
			Aperiodic	2076	13 868	29.12	0	29.39
	FIFO	OTHER	Periodic	501	12 915	17.94	37	0.00
			Aperiodic	2210	17 832	23.59	5007	0.00
		FIFO	Periodic	422	4456	17.88	0	43.16
			Aperiodic	2038	13 385	29.27	0	37.09

Table C.43: Full test TX, Setup 2, with Iperf.

Preemption Model	kthread Policy	App Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
				Avg	Max			
No Preemption	OTHER	OTHER	Periodic	701	7025	18.91	0	36.01
			Aperiodic	2264	7265	25.84	0	39.75
		FIFO	Periodic	456	5535	19.72	0	49.39
			Aperiodic	2079	6304	29.15	0	49.73
	FIFO	OTHER	Periodic	526	6572	19.40	0	40.84
			Aperiodic	2072	7259	23.94	0	37.45
		FIFO	Periodic	456	3645	19.69	0	48.45
			Aperiodic	2105	7434	29.29	0	49.52

Continued on next page

Table C.43: Full test TX, Setup 2, with Iperf. (Continued)

Preemptible Kernel (Low Latency Desktop)	OTHER	OTHER	Periodic	507	6087	19.48	0	44.50
			Aperiodic	2452	7744	30.24	0	10.14
		FIFO	Periodic	456	3848	19.71	0	48.76
			Aperiodic	2078	6320	29.15	0	49.91
	FIFO	OTHER	Periodic	502	5768	19.42	0	43.45
			Aperiodic	2025	9408	25.83	2	0.00
		FIFO	Periodic	461	3240	19.90	0	48.56
			Aperiodic	2103	7435	29.28	0	49.78
Fully Pre-emptible Kernel (Real-Time)	OTHER	OTHER	Periodic	671	7247	19.49	0	26.04
			Aperiodic	2465	7040	27.64	961	0.00
		FIFO	Periodic	459	14 840	19.92	2	0.00
			Aperiodic	2074	19 466	29.07	0	21.81
	FIFO	OTHER	Periodic	620	7777	19.64	21	0.00
			Aperiodic	2421	7220	26.44	594	0.00
		FIFO	Periodic	458	4552	19.90	0	42.86
			Aperiodic	2079	9863	29.31	0	43.73

Table C.44: Full test TX, Setup 3, no load.

Preemption Model	App Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	426	1877	17.23	0	52.34
		Aperiodic	2022	3727	27.98	0	54.45
	FIFO	Periodic	410	4028	17.08	0	54.50
		Aperiodic	2082	3621	29.20	0	56.14
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	427	2129	17.39	0	52.50
		Aperiodic	2021	3730	28.00	0	54.43
	FIFO	Periodic	411	1806	17.14	0	54.41
		Aperiodic	2081	3688	29.20	0	56.05

Continued on next page

Table C.44: Full test TX, Setup 3, no load. (Continued)

Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	428	3152	17.34	0	51.20
		Aperiodic	2008	3873	27.29	0	52.76
	FIFO	Periodic	414	3138	17.30	0	54.50
		Aperiodic	2079	3566	29.21	0	56.41

Table C.45: Full test TX, Setup 3, with Stress-NG 25 %.

Preemption Model	App Policy	Schedule	Exec. Time (μ s)		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	433	13 008	17.61	13	0.00
		Aperiodic	2015	13 282	27.48	267	0.00
	FIFO	Periodic	417	1739	17.51	0	54.29
		Aperiodic	2082	3510	29.21	0	55.88
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	433	14 361	17.60	13	0.00
		Aperiodic	2014	13 460	27.37	639	0.00
	FIFO	Periodic	419	1685	17.63	0	54.30
		Aperiodic	2080	3517	29.20	0	55.98
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	454	12 793	17.70	6	0.00
		Aperiodic	2094	13 737	27.14	502	0.00
	FIFO	Periodic	419	3307	17.61	0	54.41
		Aperiodic	2078	3690	29.23	0	56.04

Table C.46: Full test TX, Setup 3, with Stress-NG 75 %.

Preemption Model	App Policy	Schedule	Exec. Time		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	469	14675	17.84	147	0.00
		Aperiodic	2002	15029	21.88	11 043	0.00
	FIFO	Periodic	424	1698	17.93	0	54.31
		Aperiodic	2076	3614	29.21	0	55.72

Continued on next page

Table C.46: Full test TX, Setup 3, with Stress-NG 75 %. (Continued)

Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	476	15867	17.94	52	0.00
		Aperiodic	2000	15006	21.88	11 163	0.00
	FIFO	Periodic	424	1738	17.96	0	54.11
		Aperiodic	2074	10402	29.20	0	31.47
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	501	12926	17.97	70	0.00
		Aperiodic	2076	13986	23.40	7128	0.00
	FIFO	Periodic	428	1764	18.18	0	54.31
		Aperiodic	2074	3557	29.29	0	55.99

Table C.47: Full test TX, Setup 3, with Iperf.

Preemption Model	App Policy	Schedule	Exec. Time		CPU Load (%)	Under-runs	Min Buffer Size (KiB)
			Avg	Max			
No Preemption	OTHER	Periodic	862	5168	18.97	0	44.38
		Aperiodic	2026	6485	25.26	0	43.97
	FIFO	Periodic	458	3359	19.76	0	53.99
		Aperiodic	2077	3586	29.25	0	55.92
Preemptible Kernel (Low Latency Desktop)	OTHER	Periodic	479	5020	19.58	0	45.88
		Aperiodic	2019	7130	24.30	44	0.00
	FIFO	Periodic	457	1741	19.79	0	54.30
		Aperiodic	2074	3552	29.25	0	55.83
Fully Preemptible Kernel (Real-Time)	OTHER	Periodic	567	5120	19.77	23	0.00
		Aperiodic	2046	6093	25.50	0	41.81
	FIFO	Periodic	457	1736	19.88	0	54.34
		Aperiodic	2066	3585	29.26	0	55.66

C.2.8 Production Environment Test

Table C.48: Production environment latency results.

Preemption Model	Policy	Nice/ Prio	Latency (μ s)		
			Min	Avg	Max
No Preemption	OTHER	0	14	82	15 702
	OTHER	-20	13	67	7886
	FIFO	40	7	16	3112
	FIFO	60	11	16	2775
Preemptible Kernel (Low Latency Desktop)	OTHER	0	17	89	11 636
	OTHER	-20	17	72	7152
	FIFO	40	13	19	243
	FIFO	60	13	20	352
Fully Preemptible Kernel (Real-Time)	OTHER	0	20	110	12 777
	OTHER	-20	19	90	11 115
	FIFO	40	11	17	296
	FIFO	60	11	16	93

Table C.49: Production environment load results.

Preemption Model	CPU (%)					loadavg
	usr	sys	softirq	iowait	idle	
No Preemption	4.25	4.19	0.26	0.40	90.90	1.09
Preemptible Kernel (Low Latency Desktop)	4.47	5.07	0.57	0.01	89.89	0.94
Fully Preemptible Kernel (Real-Time)	4.41	7.34	0.98	0.01	87.31	2.24

Appendix D

Miscellaneous

```
1 /linux# git rev-list --count --since="Jan 1 2023" --before="Jan 1 2024" --  
   all --no-merges  
2 134098
```

LISTING D.1: Linux kernel, number of commits in 2023.

Bibliography

- [1] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time Linux kernel: A survey on PREEMPT_RT," *ACM Computing Surveys*, vol. 52, no. 1, 18:1–18:36, Feb. 2019, ISSN: 0360-0300. DOI: [10.1145/3297714](https://doi.org/10.1145/3297714).
- [2] M. M. Madden, "Challenges using Linux as a real-time operating system," in *2019 AIAA SciTech Forum and Exposition*, San Diego, CA, Jan. 2019. DOI: [10.2514/6.2019-0502](https://doi.org/10.2514/6.2019-0502).
- [3] G. K. Adam, N. Petrellis, and L. T. Doulos, "Performance assessment of Linux kernels with PREEMPT_RT on ARM-based embedded devices," *Electronics*, vol. 10, no. 11, p. 1331, Jan. 2021, ISSN: 2079-9292. DOI: [10.3390/electronics10111331](https://doi.org/10.3390/electronics10111331).
- [4] L.-C. Duca, A. Duca, and A.-S. Lup, "Real-time Linux drivers and latency evaluation system for TI OMAP4 mcSPI peripheral," in *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, Jun. 2020, pp. 1–4. DOI: [10.1109/ICECCE49384.2020.9179286](https://doi.org/10.1109/ICECCE49384.2020.9179286).
- [5] R. Delgado and B. W. Choi, "New insights into the real-time performance of a multicore processor," *IEEE Access*, vol. 8, pp. 186 199–186 211, 2020, ISSN: 2169-3536. DOI: [10.1109/ACCESS.2020.3029858](https://doi.org/10.1109/ACCESS.2020.3029858).
- [6] C.-C. J. Huang and C.-F. Yang, "An empirical approach to minimize latency of real-time multiprocessor Linux kernel," in *2020 International Computer Symposium (ICS)*, Dec. 2020, pp. 214–218. DOI: [10.1109/ICS51289.2020.00051](https://doi.org/10.1109/ICS51289.2020.00051).
- [7] Y. Wei, "Research on real-time improvement technology of Linux based on multi-core ARM," in *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, Jun. 2021, pp. 1061–1066. DOI: [10.1109/ICAICA52286.2021.9498165](https://doi.org/10.1109/ICAICA52286.2021.9498165).
- [8] X. Fan, T. Zheng, S. Sun, M. Gidlund, and J. Åkerberg, "Can embedded real-time Linux system effectively support multipath transmission? An experimental study," in *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*, Apr. 2023, pp. 1–8. DOI: [10.1109/WFCS57264.2023.10144118](https://doi.org/10.1109/WFCS57264.2023.10144118).
- [9] L.-C. Duca and A. Duca, "Achieving hard real-time networking on PREEMPT_RT Linux with RTnet," in *2020 International Symposium on Fundamentals of Electrical Engineering (ISFEE)*, Nov. 2020, pp. 1–4. DOI: [10.1109/ISFEE51261.2020.9756165](https://doi.org/10.1109/ISFEE51261.2020.9756165).
- [10] Grammarly, Inc. "Grammarly [Typing Assistant]." [grammarly.com](https://www.grammarly.com/). Accessed: May 17, 2024. [Online]. Available: <https://www.grammarly.com/>
- [11] CS-5000 (1.0) [All relevant source code for this thesis]. (2024). K. Odde. Accessed: May 23, 2024. [Online]. Available: <https://github.com/oddek/CS-5000>
- [12] A. Burns and A. Wellings, *Analysable Real-Time Systems: Programmed in Ada*. CreateSpace Independent Publishing Platform, Oct. 2016, ISBN: 978-1-5302-6550-3.

- [13] D. Abbott, "Chapter 17 - Linux and real-time," in *Linux for Embedded and Real-Time Applications (Fourth Edition)*, D. Abbott, Ed., Newnes, Jan. 2018, pp. 257–270, ISBN: 978-0-12-811277-9. DOI: [10.1016/B978-0-12-811277-9.00017-1](https://doi.org/10.1016/B978-0-12-811277-9.00017-1).
- [14] F. Vasquez and C. Simmonds, *Mastering Embedded Linux Programming: Create fast and reliable embedded solutions with Linux 5.4 and the Yocto Project 3.1 (Dunfell)*. Packt Publishing Ltd, May 2021, ISBN: 978-1-78953-511-2.
- [15] J. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, Oct. 1988, ISSN: 1558-0814. DOI: [10.1109/2.7053](https://doi.org/10.1109/2.7053).
- [16] P. E. McKenney, "'Real time' vs. 'real fast': How to choose?" In *Ottawa Linux Symposium*, Ottawa, Canada, Jul. 2008. [Online]. Available: <https://www.kernel.org/doc/ols/2008/ols2008v2-pages-57-66.pdf>.
- [17] C. Ngolah, Y. Wang, and X. Tan, "Implementing task scheduling and event handling in RTOS+," in *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No.04CH37513)*, vol. 3, May 2004, 1523–1526 Vol.3. DOI: [10.1109/CCECE.2004.1349696](https://doi.org/10.1109/CCECE.2004.1349696).
- [18] X. Wang, X. Chen, X. Yang, and B. Yang, "Requirements patterns for complex embedded systems," in *2022 IEEE 30th International Requirements Engineering Conference Workshops (REW)*, Aug. 2022, pp. 14–17. DOI: [10.1109/REW56159.2022.00011](https://doi.org/10.1109/REW56159.2022.00011).
- [19] AspenCore Media. *Embedded Market Survey 2023*. (2023). Accessed: Nov. 27, 2023. [Online]. Available: <https://www.embedded.com/embedded-survey/>
- [20] R. Stallman. "Linux and the GNU System." gnu.org. Accessed: Apr. 14, 2024. [Online]. Available: <https://www.gnu.org/gnu/linux-and-gnu.en.html>
- [21] The Linux Foundation. *Sched(7) - Linux Man Page*. (2014). Accessed: Dec. 20, 2023. [Online]. Available: <https://man7.org/linux/man-pages/man7/sched.7.html>
- [22] C. Rodriguez, G. Fischer, and S. Smolski, "Chapter 7 - Scheduling and kernel synchronization," in *The Linux Kernel Primer*, Philadelphia, PA: Prentice Hall, 2005, ISBN: 978-0-13-118163-2.
- [23] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, Jan. 2018. DOI: [10.1109/IEEESTD.2018.8277153](https://doi.org/10.1109/IEEESTD.2018.8277153).
- [24] The Linux Foundation. "Linux Standard Base Core Specification, Generic Part LSB Core - Generic 5.0." Accessed: Apr. 15, 2024. [Online]. Available: https://refspecs.linuxfoundation.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic.pdf
- [25] D. Locke, "POSIX and Linux application compatibility design rules," Locke Consulting LLC, Denver, NC, USA, Apr. 2005. Accessed: Apr. 15, 2024. [Online]. Available: https://www.researchgate.net/publication/248644573_POSIX_and_Linux_Application_Compatibility_Design_Rules.
- [26] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the Linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016, ISSN: 1097-024X. DOI: [10.1002/spe.2335](https://doi.org/10.1002/spe.2335).
- [27] Linux (6.6.14) [Operating System]. (2024). The Linux Foundation. Accessed: Apr. 18, 2024. [Online]. Available: <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tag/?h=v6.6.14>

- [28] J. Corbet. "An EEVDF CPU scheduler for Linux." lwn.net. Accessed: Apr. 19, 2024. [Online]. Available: <https://lwn.net/Articles/925371/>
- [29] P. Shah. "[PATCH] Introduce per-task latency_tolerance for scheduler hints." lkml.org. Accessed: Apr. 19, 2024. [Online]. Available: <https://lkml.org/lkml/2019/11/25/151>
- [30] I. Stoica and H. Abdel-Wahab, "Earliest eligible virtual deadline first : A flexible and accurate mechanism for proportional share resource allocation," Nov. 1995. [Online]. Available: <https://api.semanticscholar.org/CorpusID:59824119>.
- [31] M. Barabanov, "A Linux-based real-time operating system," M.S. thesis, New Mexico Institute of Mining and Technology, Socorro, NM, USA, 1997. [Online]. Available: <https://api.semanticscholar.org/CorpusID:18146086>.
- [32] The Linux Foundation. *Preemption Models - The Real Time Linux Collaborative Project Documentation*. (2023). Accessed: Apr. 19, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/preemption_models
- [33] F. Cerqueira and B. B. Brandenburg, "A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS RT," 2013. [Online]. Available: <https://api.semanticscholar.org/CorpusID:14096981>.
- [34] Y. Chen, X. Tang, S. Xu, F. Zhu, Q. Zhou, and T.-H. Weng, "Analyzing execution path non-determinism of the Linux kernel in different scenarios," *Connection Science*, vol. 35, no. 1, p. 2192442, Dec. 2023, ISSN: 0954-0091. DOI: [10.1080/09540091.2023.2192442](https://doi.org/10.1080/09540091.2023.2192442).
- [35] J. Brown, "How fast is fast enough ? Choosing between Xenomai and Linux for real-time applications," Rep Invariant Systems, Inc., Cambridge, MA, USA, 2010. Accessed: Nov. 27, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:15722073>.
- [36] H. Fayyad, M. Timmerman, L. Perneel, F. Guan, and L. Peng, "Real-time capabilities in the standard Linux Kernel: How to enable and use them?" *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 3, no. 1, pp. 131–135, Mar. 2015, ISSN: 2321-8169. [Online]. Available: <https://www.researchgate.net/publication/275018855>.
- [37] C. Emde, "Long-term monitoring of apparent latency in PREEMPT RT Linux real-time systems," OSADL, Schramberg, Germany, 2010. Accessed: Apr. 18, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:36412799>.
- [38] R. F. Molanes, J. J. Rodríguez-Andina, and J. Fariña, "Performance characterization and design guidelines for efficient processor–FPGA communication in Cyclone V FPsocs," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 5, pp. 4368–4377, May 2018, ISSN: 1557-9948. DOI: [10.1109/TIE.2017.2766581](https://doi.org/10.1109/TIE.2017.2766581).
- [39] Intel Altera. *Cyclone V Device Datasheet CV-51002*. (2023). Accessed: Oct. 6, 2005. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/666692?fileName=cv_51002-683801-666692.pdf
- [40] R. Novickis and M. Greitans, *FPGA master based on chip communications architecture for Cyclone V SoC running Linux*. Apr. 2018, p. 408. DOI: [10.1109/CoDIT.2018.8394842](https://doi.org/10.1109/CoDIT.2018.8394842).

- [41] The Linux Foundation. *Netlink - The Linux Kernel Documentation (v6.6)*. (2023). Accessed: Apr. 21, 2024. [Online]. Available: <https://kernel.org/doc/html/v6.6/userspace-api/netlink/intro.html>
- [42] T. Gleixner. "Realtime Linux: academia v. reality." lwn.net. Accessed: Dec. 11, 2023. [Online]. Available: <https://lwn.net/Articles/397422/>
- [43] A. Kofod-Petersen. *How to Do a Structured Literature Review in Computer Science*. (2015). Accessed: Sep. 30, 2023. [Online]. Available: https://research.idi.ntnu.no/aimasters/files/SLR_HowTo2018.pdf
- [44] The Linux Foundation. "Preempt-RT History." [wiki.linuxfoundation.org](https://wiki.linuxfoundation.org/realtime/rtl/blog#preempt-rt_history). Accessed: Dec. 16, 2023. [Online]. Available: https://wiki.linuxfoundation.org/realtime/rtl/blog#preempt-rt_history
- [45] The Linux Foundation. *Lock types and their rules - The Linux Kernel Documentation (v6.6)*. (2023). Accessed: Apr. 22, 2024. [Online]. Available: <https://kernel.org/doc/html/v6.6/locking/locktypes.html>
- [46] P. Zilstra. "[PATCH] Introduce CONFIG_PREEMPT_DYNAMIC." [lore.kernel.org](https://lore.kernel.org/lkml/161278563996.23325.11147110316009301248.tip-bot2@tip-bot2/). Accessed: Apr. 21, 2024. [Online]. Available: <https://lore.kernel.org/lkml/161278563996.23325.11147110316009301248.tip-bot2@tip-bot2/>
- [47] The Linux Foundation. *Time(7) - Linux Man Page*. (2006). Accessed: Apr. 15, 2024. [Online]. Available: <https://man7.org/linux/man-pages/man7/time.7.html>
- [48] T. Gleixner and D. Niehaus, "Hrtimers and beyond: Transforming the Linux time subsystems," in *Ottawa Linux Symposium (OLS)*, Ottawa, Canada, Jul. 2006. [Online]. Available: <https://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf>
- [49] rt-tests (2.5) [Test Suite]. C. Williams and J. Kacur. Accessed: May 19, 2024. [Online]. Available: <https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/tag/?h=v2.5>
- [50] The Linux Foundation. *CyclicTest - The Real Time Linux Collaborative Project Documentation*. (2023). Accessed: Dec. 13, 2023. [Online]. Available: <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start>
- [51] J. Edge. "Moving interrupts to threads." lwn.net. Accessed: Apr. 22, 2024. [Online]. Available: <https://lwn.net/Articles/302043/>
- [52] T. Gleixner. "[PATCH] Add Infrastructure for Threaded Interrupt Handlers." [lore.kernel.org](https://lore.kernel.org/all/20081001223213.078984344@linutronix.de/). Accessed: Apr. 22, 2024. [Online]. Available: <https://lore.kernel.org/all/20081001223213.078984344@linutronix.de/>
- [53] The Linux Foundation. *Threaded Interrupt Handler - The Real Time Linux Collaborative Project Documentation*. (2023). Accessed: Apr. 22, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/threadirq
- [54] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990, ISSN: 1557-9956. DOI: 10.1109/12.57058. (accessed Dec. 7, 2023).
- [55] B. W. Lampson and D. D. Redell, "Experience with processes and monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, Feb. 1980, ISSN: 0001-0782. DOI: 10.1145/358818.358824.

- [56] The Linux Foundation. *PI-futex - The Linux Kernel Documentation (v6.6)*. (2023). Accessed: Apr. 21, 2024. [Online]. Available: <https://kernel.org/doc/html/v6.6/locking/pi-futex.html>
- [57] The Linux Foundation. *Preempt Locking - The Linux Kernel Documentation (v6.6)*. (2023). Accessed: Apr. 21, 2024. [Online]. Available: <https://kernel.org/doc/html/v6.6/locking/preempt-locking.html>
- [58] S. Rostedt and D. V. Hart, "Internals of the RT patch," in *Ottawa Linux Symposium*, Ottawa, Canada, 2007. [Online]. Available: <https://www.kernel.org/doc/ols/2007/ols2007v2-pages-161-172.pdf>.
- [59] The Linux Foundation. *RCU: What is RCU? - The Linux Kernel Documentation (v6.6)*. (2023). Accessed: Dec. 18, 2023. [Online]. Available: <https://kernel.org/doc/html/v6.6/RCU/whatisRCU.html>
- [60] P. McKenney. "The Design of Preemptible Read-Copy-Update." lwn.net. Accessed: Dec. 18, 2023. [Online]. Available: <https://lwn.net/Articles/253651/>
- [61] The Linux Foundation. *RCU - The Real Time Linux Collaborative Project Documentation*. (2023). Accessed: May 20, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/rcu
- [62] Linux (6.6.14-rt21) [Operating System]. (2024). The Linux Foundation. Accessed: Apr. 18, 2024. [Online]. Available: <https://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git/tag/?h=v6.6.14-rt21>
- [63] The Linux Foundation. *Reducing OS jitter due to per-cpu kthreads - The Linux Kernel documentation (v6.6)*. (2023). Accessed: May 20, 2024. [Online]. Available: <https://kernel.org/doc/html/v6.6/admin-guide/kernel-per-CPU-kthreads.html>
- [64] The Linux Foundation. *NO_HZ: Reducing Scheduling-Clock Ticks - The Linux Kernel Documentation (v6.6)*. (2023). Accessed: Apr. 18, 2024. [Online]. Available: https://www.kernel.org/doc/html/v6.6/timers/no_hz.html
- [65] The Linux Foundation. *Dynticks or Tickless kernel or nohz - The Real Time Linux Collaborative Project Documentation*. (2023). Accessed: Apr. 30, 2024. [Online]. Available: <https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/ticklesskernel>
- [66] F. Weisbecker. "CPU Isolation." suse.com. Accessed: Apr. 30, 2024. [Online]. Available: <https://www.suse.com/c/cpu-isolation-introduction-part-1/>
- [67] Linux Plumbers Conference. "QA about PREEMP_RT - Thomas Gleixner." Accessed: Dec. 16, 2023. [Online]. Available: <https://www.youtube.com/watch?v=OldzeGJUvvU>
- [68] J. Corbet. "The real realtime preemption end game." lwn.net. Accessed: Dec. 16, 2023. [Online]. Available: <https://lwn.net/Articles/951337/>
- [69] Red Hat Enterprise Linux. "Getting Started with Red Hat Enterprise Linux for Real Time." redhat.com. Accessed: Dec. 20, 2023. [Online]. Available: <https://access.redhat.com/rhel-real-time-getting-started>
- [70] Canonical. "Real-time Ubuntu." ubuntu.com. Accessed: Dec. 20, 2023. [Online]. Available: <https://ubuntu.com/real-time>

- [71] P. Regnier, G. Lima, and L. Barreto, "Evaluation of interrupt handling timeliness in real-time Linux operating systems," *Operating Systems Review*, vol. 42, pp. 52–63, Oct. 2008. DOI: [10.1145/1453775.1453787](https://doi.org/10.1145/1453775.1453787).
- [72] L. Vignati, S. Zambon, and L. Turchet, "A comparison of real-time Linux-based architectures for embedded musical applications," *Journal of the Audio Engineering Society*, vol. 70, pp. 83–93, Jan. 2021. DOI: [10.17743/jaes.2021.0052](https://doi.org/10.17743/jaes.2021.0052).
- [73] A. Carvalho, C. Machado, and F. Moraes, "Raspberry Pi performance analysis in real-time applications with the RT-Preempt patch," in *2019 Latin American Robotics Symposium (LARS), 2019 Brazilian Symposium on Robotics (SBR) and 2019 Workshop on Robotics in Education (WRE)*, Oct. 2019, pp. 162–167. DOI: [10.1109/LARS-SBR-WRE48964.2019.00036](https://doi.org/10.1109/LARS-SBR-WRE48964.2019.00036).
- [74] G. K. Adam, "Co-design of multicore hardware and multithreaded software for thread performance assessment on an FPGA," *Computers*, vol. 11, no. 5, p. 76, May 2022, ISSN: 2073-431X. DOI: [10.3390/computers11050076](https://doi.org/10.3390/computers11050076).
- [75] J. Altenberg. (2016). Introduction to realtime Linux - Presented at Embedded Linux Conference Europe, Berlin, Germany. Accessed: Oct. 9, 2023. [Online]. Available: https://elinux.org/images/8/8e/Introduction_to_Realtime_Linux.pdf
- [76] Y. Li, Y. Matsubara, H. Takada, K. Suzuki, and H. Murata, "A performance evaluation of embedded multi-core mixed-criticality system based on PREEMPT_RT Linux," *Journal of Information Processing*, vol. 31, no. 0, pp. 78–87, 2023, ISSN: 1882-6652. DOI: [10.2197/ipsjjip.31.78](https://doi.org/10.2197/ipsjjip.31.78). (accessed Sep. 29, 2023).
- [77] R. Webster. "Excessive network latency when using Realtek R8168/R8111 et al NIC." lore.kernel.org. Accessed: Dec. 19, 2023. [Online]. Available: <https://lore.kernel.org/linux-rt-users/CANV1gkc9KvkGNPkEsjXCiV4mUcdrrPcaQ1ueRri9ypjvJbU94g@mail.gmail.com/>
- [78] J. Salisbury. "System Hang With 5.15.79-rt54 Patch Set." lore.kernel.org. Accessed: Dec. 19, 2023. [Online]. Available: <https://lore.kernel.org/linux-rt-users/fe5974c9-3ed0-938a-f43c-4d301f603e92@canonical.com/>
- [79] P. Pisa. "Outstanding latency increase in kernel CAN gateway caught by CAN-latester." lore.kernel.org. Accessed: Dec. 19, 2023. [Online]. Available: <https://lore.kernel.org/linux-rt-users/202310021040.49367.pisa@fel.cvut.cz/>
- [80] G. Medini. "High latency of a system based on 5.19 rt." lore.kernel.org. Accessed: Dec. 19, 2023. [Online]. Available: [https://lore.kernel.org/linux-rt-users/S1JV78\\$805284328FE38E68AC9E7D23F35846D8@eurosoft.it/](https://lore.kernel.org/linux-rt-users/S1JV78$805284328FE38E68AC9E7D23F35846D8@eurosoft.it/)
- [81] M. Franklin. "i2c jitter is worse in PREEMPT_RT kernel than stock Raspberry Pi kernel." lore.kernel.org. Accessed: Dec. 19, 2023. [Online]. Available: <https://lore.kernel.org/linux-rt-users/44fb3d4d-6303-4287-b2ac-7898cc237c47@comfiletech.com/#r>
- [82] Open Source Automation Development Lab. "Hardware - OSADL QA Farm." osadl.org. Accessed: Apr. 18, 2024. [Online]. Available: <https://www.osadl.org/Hardware-overview.qa-farm-hardware.0.html>
- [83] Open Source Automation Development Lab. "ARM Cortex A9 Latency Plot - OSADL QA Farm." osadl.org. Accessed: Apr. 18, 2024. [Online]. Available: <https://www.osadl.org/Long-term-latency-plot-of-system-in-rack.qa-3d-latencyplot-r2s3.0.html?shadow=1>

- [84] T. Beck, F. Boniol, J. Ermont, and L. Maillet, "Impact of environment on the execution of a real-time Linux process on a multicore platform," in *11th european congress on embedded real-time systems (ERTS 2022)*, 2022. [Online]. Available: <https://hal.science/hal-03857305>.
- [85] OSADL. "SIL2LinuxMP." [osadl.org](https://www.osadl.org/SIL2LinuxMP.sil2-linux-project.0.html). Accessed: Dec. 20, 2023. [Online]. Available: <https://www.osadl.org/SIL2LinuxMP.sil2-linux-project.0.html>
- [86] J. Jeong, E. Seo, D. Kim, *et al.*, "Transparent and selective real-time interrupt services for performance improvement," in *Software Technologies for Embedded and Ubiquitous Systems*, R. Obermaisser, Y. Nah, P. Puschner, and F. J. Rammig, Eds., vol. 4761, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 283–292, ISBN: 978-3-540-75663-7. DOI: [10.1007/978-3-540-75664-4_28](https://doi.org/10.1007/978-3-540-75664-4_28).
- [87] The Linux Foundation. *RT throttling - The Real Time Linux Collaborative Project Documentation*. (2023). Accessed: May 1, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/sched_rt_throttling
- [88] Daniel Bristot de Oliveira. "[PATCH] sched/rt: RT_RUNTIME_GREED sched feature." [lore.kernel.org](https://lore.kernel.org/lkml/fa5b1b55d8934c6a0e02e04a7ad6afdf4012c2e0.1478506194.git.bristot@redhat.com/). Accessed: May 1, 2024. [Online]. Available: <https://lore.kernel.org/lkml/fa5b1b55d8934c6a0e02e04a7ad6afdf4012c2e0.1478506194.git.bristot@redhat.com/>
- [89] Red Hat Enterprise Linux. "Tuning Scheduling Policy - Red Hat Documentation." [redhat.com](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/tuning-scheduling-policy_monitoring-and-managing-system-status-and-performance). Accessed: Apr. 18, 2024. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/tuning-scheduling-policy_monitoring-and-managing-system-status-and-performance
- [90] The Linux Foundation. *Capabilities(7) - Linux Man Page*. (2002). Accessed: Apr. 21, 2024. [Online]. Available: <https://man7.org/linux/man-pages/man7/capabilities.7.html>
- [91] The Linux Foundation. *HOWTO build a basic cyclic application - The Real Time Linux Collaborative Project Documentation*. (2017). Accessed: May 1, 2024. [Online]. Available: <https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/cyclic>
- [92] The Free Software Foundation. "Time Types - GNU Libc Documentation (v2.38)." The GNU C Library. Accessed: Dec. 19, 2023. [Online]. Available: https://sourceware.org/glibc/manual/2.38/html_node/Time-Types.html
- [93] GCC (13.2.0) [Compiler]. (2023). The Free Software Foundation. Accessed: Apr. 18, 2024. [Online]. Available: <https://gcc.gnu.org/git/?p=gcc.git;a=commit;h=c891d8dc23e1a46ad9f3e757d09e57b500d40044>
- [94] Boost (1.84.0) [C++ Library]. (2023). The Boost Organization. Accessed: Apr. 18, 2024. [Online]. Available: <https://github.com/boostorg/boost/releases/tag/boost-1.84.0>
- [95] The Linux Foundation. *Timers Howto - The Linux Kernel Documentation (v6.6)*. (2023). Accessed: Apr. 18, 2024. [Online]. Available: <https://kernel.org/doc/html/v6.6/timers/timers-howto.html>

- [96] The Linux Foundation. *Memory for Real-time Applications - The Real Time Linux Collaborative Project Documentation*. (2017). Accessed: Apr. 18, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/memory#memory_locking
- [97] D. Duval, "From fast to predictably fast," in *Ottawa Linux Symposium*, Ottawa, Canada, 2009. [Online]. Available: <https://www.kernel.org/doc/ols/2009/ols2009-pages-79-86.pdf>.
- [98] J. Ogness. "A Checklist for Real-Time Applications in Linux." *linutronix.de*. Accessed: May 1, 2024. [Online]. Available: <https://www.linutronix.de/blog.php>
- [99] The Linux Foundation. *Sysctl VM - The Linux Kernel Documentation (v6.6)*. (2023). Accessed: Apr. 18, 2024. [Online]. Available: <https://kernel.org/doc/html/v6.6/admin-guide/sysctl/vm.html>
- [100] I. Puaut, "Real-time performance of dynamic memory allocation algorithms," in *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, Jun. 2002, pp. 41–49. DOI: 10.1109/EMRTS.2002.1019184.
- [101] J. Cartwright. (2018). What every driver developer should know about RT. Presented at The Embedded Linux Conference North America, Portland, OR. Accessed: Apr. 3, 2024. [Online]. Available: https://static.sched.com/hosted_files/elciotna18/27/What%20every%20device%20driver%20developer%20should%20know%20about%20rt.pdf
- [102] Intel Altera. *Cyclone® V SoC FPGA Development Board Reference Manual 654687*. (2015). Accessed: Dec. 12, 2023. [Online]. Available: <https://www.intel.com/content/www/us/en/content-details/654687/cyclone-v-soc-fpga-development-board-reference-manual.html>
- [103] Crosstool-NG (1.26.0) [Toolchain Generator]. Crosstool-NG Community. Accessed: Apr. 20, 2024. [Online]. Available: <https://github.com/crosstool-ng/crosstool-ng/releases/tag/crosstool-ng-1.26.0>
- [104] Das U-Boot (2023.07) [Bootloader]. DENX Software Engineering. Accessed: Apr. 20, 2024. [Online]. Available: <https://github.com/u-boot/u-boot/releases/tag/v2023.07>
- [105] L. Torvalds. "Linux 6.0-rc1." *lore.kernel.org*. Accessed: May 19, 2024. [Online]. Available: https://lore.kernel.org/lkml/CAHk-=wgRFjPHV-Y_eKP9wQMLFDgG+dEUHiv5wC17OQHsG5z7BA@mail.gmail.com/T/
- [106] Buildroot (2023.08.01) [Linux System Generator]. (2023). Peter Korsgaard. Accessed: Apr. 20, 2024. [Online]. Available: <https://gitlab.com/buildroot.org/buildroot/-/tags/2023.08.1>
- [107] stress-ng (0.15.07) [Load Simulator]. (2023). C. I. King. Accessed: May 4, 2024. [Online]. Available: <https://github.com/ColinIanKing/stress-ng>
- [108] C. I. King. "Issue #363 - Comment." *github.com*. Accessed: Apr. 19, 2024. [Online]. Available: <https://github.com/ColinIanKing/stress-ng/issues/363>
- [109] libnl (3.7.0) [Netlink Library Suite]. (2022). T. Haller. Accessed: May 20, 2024. [Online]. Available: <https://github.com/thom311/libnl>
- [110] Embedded Linux Wiki. "High Resolution Timers." *elinux.org*. Accessed: Apr. 21, 2024. [Online]. Available: https://elinux.org/High_Resolution_Timers#How_to_validate

-
- [111] The Linux Foundation. *Kmemleak - The Linux Kernel Documentation (v.6.6)*. (2023). Accessed: Apr. 19, 2024. [Online]. Available: <https://kernel.org/doc/html/v6.6/dev-tools/kmemleak.html>
- [112] The eBPF Foundation. "What is eBPF? An Introduction and Deep Dive into the eBPF Technology." ebpf.io. Accessed: Apr. 24, 2024. [Online]. Available: <https://ebpf.io/what-is-ebpf/>
- [113] The Linux Foundation. *Signal(7) - Linux Man Page*. (1993). Accessed: May 5, 2024. [Online]. Available: <https://man7.org/linux/man-pages/man7/signal.7.html>
- [114] The Linux Foundation. *Getitimer(2) - Linux Man Page*. (1993). Accessed: Apr. 26, 2024. [Online]. Available: <https://man7.org/linux/man-pages/man2/setitimer.2.html>