Evaluating Guest Isolation on a Hypervised System

Master's thesis in Computer science and engineering

Agnes Asp
Alfred Karlsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Evaluating Guest Isolation on a Hypervised System

Agnes Asp
Alfred Karlsson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Evaluating Guest Isolation on a Hypervised System
Agnes Asp
Alfred Karlsson

© Agnes Asp & Alfred Karlsson 2025.

Supervisor: Magnus Almgren, Department of Computer Science and Engineering
Advisors: Caio Carvalho & Nikolaos Korkakais, Volvo Cars
Examiner: Magnus Almgren, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Evaluating Guest Isolation on a Hypervised System

Agnes Asp
Alfred Karlsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As mixed-critical systems become more prevalent in automotive systems, virtualization has emerged as a promising solution to reduce system complexity and improve cost-efficiency. This thesis investigates the ability of hypervisors to maintain temporal isolation between virtual machines (VMs) under conditions that simulate disturbances. Two general-purpose hypervisors, Xen and QEMU/KVM, are evaluated on an ARM-based Raspberry Pi 4B using ZephyrOS as a Real-Time Operating System (RTOS) in both measurer and stressor roles.

A test framework was developed to benchmark low-level latency operations and application-level performance using adapted MiBench workloads (Qsort and Basicmath), and long-term scheduling behavior through thread metrics. Performance metrics were collected under various configurations, including stressed and unstressed scenarios across different CPU core assignments.

The results show that while both hypervisors provide a baseline level of temporal isolation, their behaviors diverge under stress. QEMU/KVM generally demonstrates better raw performance and responsiveness, whereas Xen offers more predictable behavior in specific scheduling configurations. These findings underscore the trade-offs involved in selecting a hypervisor for real-time automotive applications and contribute to a broader understanding of how virtualization affects temporal determinism in embedded systems.

Keywords: Hypervisor, Temporal Isolation, ARM, Virtual Machine, Real-Time Operating System, Virtualization, Mixed-Criticality, Xen, QEMU/KVM, ZephyrOS

Acknowledgements

We want extend our gratitude for the opportunity to do this master thesis in collaboration with Volvo Cars. We want to thank our industrial supervisors Nikolaos Korkakais and Caio Carvalho for all their help and encouragement through the whole process of the thesis. We also want thank our academic supervisor and examiner Magnus Almgren for his support and guidance throughout this project.

Agnes Asp & Alfred Karlsson, Gothenburg, 2025-10-12

Contents

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**
1.1 Background 1
1.2 Aim 2
1.3 Related Work 2
1.4 Limitations 4
1.5 Disposition 4

2 Theory **5**
2.1 Operating Systems 5
2.2 Virtualization 5
2.3 Hardware support for virtualization 7
2.3.1 x86 architecture 8
2.3.2 ARM architecture 8
2.4 Isolation 10
2.4.1 Temporal Isolation 10
2.4.2 Spatial Isolation 10
2.5 Hypervisors 13
2.5.1 Xen 13
2.5.2 QEMU/KVM 14
2.6 Real Time Operating Systems 15

3 Methodology **17**
3.1 The Design of Experiment 17
3.2 System on Chip 18
3.3 Implementation 18
3.3.1 ZephyrOS 19
3.3.2 Xen 19
3.3.3 QEMU/KVM 20
3.3.4 Stressor 20
3.3.5 Measuring VM 20
3.4 Test Framework Areas 21
3.4.1 Latency measurments 21
3.4.2 MiBench 22

3.4.3 Thread metrics . . . 22

4 Results **25**
4.1 Latency Measurments . . . 25
4.2 MiBench . . . 28
4.2.1 Qsort . . . 28
4.2.2 Basicmath . . . 31
4.3 Thread metrics . . . 34
4.3.1 Stress Start Order and CPU Affinity . . . 34
4.3.2 30-minute test . . . 38

5 Discussion **43**
5.1 Latency Measurements . . . 43
5.2 MiBench . . . 44
5.2.1 Qsort . . . 45
5.2.2 Basicmath . . . 46
5.3 Thread Metrics . . . 47
5.3.1 Start Order of the Stress VM . . . 47
5.3.2 30 minute test . . . 47
5.4 Threats to Validity . . . 49
5.5 Evaluation of the Test Framework . . . 50
5.6 Evaluation of Research Questions . . . 51
5.6.1 Research Question 1 . . . 51
5.6.2 Research Question 2 . . . 52
5.7 Ethical Considerations . . . 52
5.8 Future work . . . 53

6 Conclusion **54**

Bibliography **55**

A U-Boot script for Xen **I**

B All Figures of Latency Measurements **II**

List of Figures

2.1 Visualization of how a fully virtualized guest communicates with hypervisor and hardware . . . 6
2.2 Visualization of how a para-virtualized guest communicates with hypervisor and hardware. The privileged instruction triggers the hardware trap that relays the instruction to the hypervisor before executing. . . . 7
2.3 Exception levels in ARM architechture . . . 9
2.4 Stage 2 translation used in ARM architecture to virtualise memory addresses 10
2.5 Visualization of broken temporal isolation. In this case, the execution of guest 1 impacts the time it takes for guest 0 to execute its program. . . . 11
2.6 Visualization of working spatial isolation where the hypervisor ensures that the memory is mapped to the different guests without interference. . . . 11
2.7 Visualization of type 1 and 2 hypervisors . . . 13
2.8 Overview of the Xen hypervisor architecture. Dom0 manages DomUs and provides access to hardware resources, but DomUs can also interact directly with the hypervisor via hypercalls, reducing the need for all drivers to be routed through Dom0. . . . 14
2.9 KVM hypervisor . . . 14
2.10 EDF-sheduling algorithm with pre-emption example. T2 is pre-empted at 15 ms in favour of T3 which has an earlier deadline. T2 is completed afterwards. . . . 16

3.1 Overview of the test setup . . . 18
3.2 Illustration of core and cache hierarchy on the Broadcom BCM2711 chip. 19
3.3 A list of latency measurement items sourced from the Zephyr Project RTOS benchmarks [24] . . . 22

4.1 Distribution of the number of cycles it takes to wait for any event to occur, with a context switch. . . . 26
4.2 Distribution of the number of cycles it takes to wait for all events, no context switch happening. . . . 26
4.3 Distribution of the number of cycles it takes to set events with a context switch. . . . 27
4.4 Distribution of the number of cycles it takes to perform a context switch when yielding a thread. . . . 27
4.5 Qsort: Distribution of qsort execution time (in CPU cycles) across six system configurations using Xen and QEMU/KVM hypervisors, measured under various stress conditions. . . . 29

4.6 Qsort: An enhanced look at Xen's performance running qsort with a disturbed guest on a different VM and a guest running solo. 29
4.7 Qsort: An enhanced look at QEMU/KVM's performance running qsort with a disturbed guest on a different VM and a guest running solo. . . . 30
4.8 Qsort: Line plot of the first 500 iterations of QEMU/KVM with a disturbed guest and a solo guest, showing the performance ramp-up for the solo guest. 30
4.9 Qsort: Distribution of qsort execution time (in CPU cycles) between iterations 500 and 3000, comparing a disturbed guest and a solo guest after the ramp-up period of the QEMU/KVM solo configuration. 31
4.10 Basicmath: Distribution of basicmath execution time (in CPU cycles) across six system configurations using Xen and QEMU/KVM hypervisors, measured under various stress conditions. 32
4.11 Basicmath: An enhanced look at Xen's performance running Basicmath with a disturbed guest on a different VM and a guest running solo. . . . 32
4.12 Basicmath: An enhanced look at QEMU/KVM's performance running basicmath with a disturbed guest on a different VM and a guest running solo. 33
4.13 Basicmath: Line plot of the first 500 iterations of QEMU/KVM with a disturbed guest and a solo guest, showing the performance ramp-up for the solo guest. 33
4.14 Basicmath: Distribution of basicmath execution time (in CPU cycles) between iterations 500 and 3000, comparing a disturbed guest and a solo guest after the ramp-up period of the QEMU/KVM solo configuration. . 34
4.15 On QEMU/KVM, scheduling over 10 minutes with different start orders of the stress and the test. Different CPU affinity was also tested with the test running on either the same or different CPUs. 35
4.16 On Xen, scheduling over 10 minutes with different start orders of the stress and the test. Different CPU affinity was also tested, with the test running on either the same or different CPUs. 36
4.17 Magnified version of Figure 4.16 to see the difference in performance at the start of the test on Xen hypervisor. 37
4.18 Run over 30 minutes with thread metrics test on Xen where stress VM is started at 10 minutes and stopped at 20 minutes 39
4.19 Enhanced variant of Figure 4.18 to see the dip when the stress VM is turned on. 39
4.20 Distribution of the thread metric count on the Xen hypervisor, a higher number of operations per time unit is favorable. 40
4.21 Run over 30 minutes with thread metrics test on QEMU/KVM where stress VM is started at 10 minutes and stopped at 20 minutes 40
4.22 Focused version of Figure 4.21 to see the change of the stress better. . . . 41
4.23 Focused version of Figure 4.22 to see the values at the time the stress is introduced in more details. 41
4.24 Distribution of the thread metric count on the QEMU/KVM hypervisor, a higher number of operations per time unit is favorable. 42
4.25 Distribution of the thread metric count on the QEMU/KVM hypervisor, outlier excluded to be able to enlarge the graph 42

5.1 Lineplot of qsort performance in cycles on Xen with a disturbing guest on a different core 46

B.1 Distribution of cycles across different configurations for the Latency Measurement *Allocate to add data to LIFO (no context switch)* III
B.2 Distribution of cycles across different configurations for the Latency Measurement *Allocate to add data to LIFO (with context switch)* III
B.3 Distribution of cycles across different configurations for the Latency Measurement *Average time for heap free* IV
B.4 Distribution of cycles across different configurations for the Latency Measurement *Average time for heap malloc* IV
B.5 Distribution of cycles across different configurations for the Latency Measurement *Context switch via k_yield* V
B.6 Distribution of cycles across different configurations for the Latency Measurement *Create thread* V
B.7 Distribution of cycles across different configurations for the Latency Measurement *Free when getting data from FIFO (no context switch)* VI
B.8 Distribution of cycles across different configurations for the Latency Measurement *Free when getting data from FIFO (with context switch)* VI
B.9 Distribution of cycles across different configurations for the Latency Measurement *Free when getting data from LIFO (no context switch)* VII
B.10 Distribution of cycles across different configurations for the Latency Measurement *Free when getting data from LIFO (with context switch)* VII
B.11 Distribution of cycles across different configurations for the Latency Measurement *Get data from FIFO (no context switch)* VIII
B.12 Distribution of cycles across different configurations for the Latency Measurement *Get data from FIFO (with context switch)* VIII
B.13 Distribution of cycles across different configurations for the Latency Measurement *Get data from kernel stack (no context switch)* IX
B.14 Distribution of cycles across different configurations for the Latency Measurement *Get data from kernel stack (with context switch)* IX

List of Tables

3.1 Example configuration table with thread metric and stress start times, and assigned core. The last row is when the thread metric is executed without any stress 23

5.1 Outliers of Latency Measurements and the respective median values of the benchmark 44
5.2 Comparative Performance Statistics of Virtualization Setups vs. Baselines when running qsort, WCET 44
5.3 Comparative Performance Statistics of Virtualization Setups vs. Baselines for basicmath, WCET 47
5.4 Comparative Performance Statistics of Stress Order 47
5.5 Comparative Performance Statistics of 30-minute test 48

1
Introduction

The use of mixed-critical systems is becoming increasingly relevant in the automotive industry due to the need for cost efficiency proved system performance, and environmental sustainability. A mixed-criticality system can be described as having both real-time and general-purpose systems sharing the same hardware. By combining multiple systems on shared hardware, manufacturers can reduce both the system complexity and production requirements of hardware components. An important challenge in such an approach is ensuring that one system's performance does not negatively impact others on the same hardware, especially when dealing with time-sensitive tasks.

1.1 Background

The trend in the automotive industry currently is to move away from a distributed and complex network of electronic control units (ECUs) that all have one task and communicate using networking inside the car. The issue with having individual ECUs for each task is that it makes the system very complex and hard to manage. Instead, the new vision for the architecture is to have fewer centralized units that handle many tasks to reduce the complexity of the system [1].

One solution to this challenge is virtualization, which enables multiple systems to run independently on shared hardware by partitioning computing resources such as CPU, Network, and Input/Output (I/O). Virtualization allows systems to be either isolated with their own operating environments or to run tailored software directly on the virtualization method. Hypervisors, also called Virtual Machine Monitors (VMM) is one technology behind solving virtualization and is a software layer responsible for distribution of resources between the virtual systems. Hypervisors are generally categorized into two types: Type-1 hypervisors, which run directly on the hardware and manage virtual machines (VMs), and Type-2 hypervisors, which run on top of a host operating system.

When dealing with time-critical tasks, any disturbance between systems sharing resources could disrupt their performance and lead to missed deadlines or failures. While certain tasks may tolerate some disturbance, others, especially safety-critical tasks, are more sensitive. In a mixed-critical system, each component can tolerate a different level of disturbance and still meet its timing and functional requirements. Therefore, it is essential to evaluate each system individually to understand the effects locally and within the right context.

1.2 Aim

The problem at hand consists of getting a better understanding if the virtualization techniques are safe and efficient enough to be able to be used in an automotive context. This is a complex tasks since there are many different aspects to safety. Hence the scope includes finding a suitable set and then testing some of these, to try to evaluate the trade-offs under a set of different conditions.

For this thesis, the objective is to evaluate hypervisor performance with regard to guest isolation, which is one aspect of safety. Specifically, the goal is to determine if, and to what extent, a guest operating system is impacted when multiple systems run concurrently on the same hardware. The evaluation will primarily focus on temporal isolation.

The central objective is to compare hypervisors based on how effectively they isolate virtual machines from one another in the context of real-time tasks. The selected hypervisors will be evaluated using a defined set of Key Performance Indicators (KPIs) to ensure a fair and consistent comparison.

Temporal isolation will be the key criterion for assessing performance disturbances, in other words, measuring delays caused by insufficient VM isolation. To test this, a Real-Time Operating System (RTOS) will be used to run both a test VM and a disturbance VM. Two general-purpose hypervisors, which can be adapted for use in real-time embedded systems, will be evaluated.

To make it more comprehensible, the aim is broken down into the following research questions:

RQ 1: What needs to be part of a suitable taxonomy of KPIs, and which indicative KPIs can be used to evaluate guest isolation in embedded/automotive contexts?

RQ 2: Do embedded hypervisors actually differ in temporal isolation, and how effectively can a framework expose those differences and each hypervisors performance strengths?

1.3 Related Work

In Cinque et al. [2] they study how different parameters can affect the temporal isolation of the Xen hypervisor used in a real-time domain with a focus on railway systems. They study how CPU affinity, different scheduling algorithms, and parameters of these algorithms affect the timing of the system. This is measured in execution time. They also measure the system under stress. Their findings suggest that temporal isolation can break under the Xen hypervisor, but this can be controlled by carefully choosing the parameters of Xen. Their study evaluates the Xen hypervisor by itself against multiple metrics such as CPU, Cache, Device, I/O, Interrupt, File system activity, and network performance. They leveraged the Linux Real Time patch PREEMPT_RT to use Linux-based guests. Furthermore, the study tried to mimic real-world Automatic Train Control (ATC) operations as measurements and see their worst-case execution times to evaluate temporal isolation. This thesis aims to evaluate towards operations common in an automotive embedded context and evaluate if temporal isolation can break in low level kernel operations.

In another study by Cinque et al. [3], they survey the state of the art when it comes

to virtualization and mixed criticality systems. They describe how general-purpose hypervisors, such as KVM and Xen, can be adapted to handle the constraints of real-time systems. They claim that these types of hypervisors are a good choice for mixed criticality applications with the right adaptations, but do not perform their own evaluations of isolation.

Martins and Pinto [4] evaluates static partitioning hypervisors (SPH) such as Xen, Jailhouse, Bao, and seL4 using the MiBench Automotive and Industrial Control Suite. The MiBench suite is designed to emulate embedded applications, such as airbag controllers and sensor systems, which are representative of workloads found in safety-critical and real-time environments. The evaluation focuses on key metrics including performance, interrupt latency, inter-VM communication, boot time, and code size. This thesis aims to further evaluate, other than interrupt latency, kernel-level operations as a metric for hypervisors and see if any significant performance degradation can be seen.

The findings of Martins and Pint [4] indicate that, due to the lack of support for directly delivering interrupts to guests on ARM platforms, all SPHs experience increased interrupt latency. The study reveals that this latency can increase tenfold when an interfering guest is present. However, the issue can be effectively mitigated through the use of cache-coloring. Another finding is that, because two of the evaluated hypervisors, Jailhouse and Bao, implement an interrupt handling technique called direct injection, they provide near-native interrupt latency. Another key finding is that the major bottleneck of VM boot time is caused by the bootloader and not the hypervisors.

In Shen et al. [5] a new hypervisor tailored towards mixed-critical systems is presented. A Linux-based benchmarking tool called cyclictest is used to evaluate its performance in an embedded system setting. This study presents a metric called "jitter" to describe the discrepancy of execution time for different iterations of cyclictest. It uses Linux guests to evaluate embedded performance and only uses cyclictest to display how well the hypervisor performs.

Modica et al. [6] investigate spatial and temporal isolation techniques in the XVISOR hypervisor, targeting ARM multi-core platforms. Their work introduces two key mechanisms: cache coloring for Last-Level Cache (LLC) partitioning and a memory bandwidth reservation scheme integrated with the hypervisor's scheduler to manage DRAM controller contention.

Their evaluation was performed on a Raspberry Pi 2 (ARM Cortex-A7), using two Linux guest domains. The first, a victim domain, ran Isol-Bench to measure memory access times. The second, an interfering domain, generated contention through continuous memory access patterns or bandwidth-saturating workloads. They assessed the effectiveness of their isolation mechanisms by comparing benchmark results between an unmodified XVISOR and their enhanced version. Results showed that cache coloring stabilized execution times under cache pressure, and bandwidth reservation effectively mitigated DRAM contention. Since both Isol-Bench and cyclictest are Linux-based benchmarking suites they cannot be used on a Real Time Operating System (RTOS) as measuring guests.

When it comes to where the state of the art is, this is surveyed by Gou et al. [1]. Here, the trends in automotive computing are surveyed, and the needs regarding virtualization are discussed. When supporting the change from a distributed network of computers with individual tasks to a more centralized system, reliable virtualization methods are needed.

Paravirtualized systems are one part that Guo et al. list as a research area that shows promise. In a paravirtualized context, the operating systems need to be modified in order to be able to interact with the hypervisor, which adds extra work but could in turn reduce overhead and increase efficiency [1].

1.4 Limitations

While this thesis provides insights into the isolation of hypervisors, several limitations must be acknowledged to contextualize the findings properly. All tests performed will be on the same hardware, a Raspberry Pi 4B, which means only one version of ARM architecture and no other type of architecture.

The Advanced RISC Machine (ARM) architecture is more commonly used in this context, and the combination of hypervisors on ARM is the focus of this thesis. More research has been done on x86 architectures regarding hypervisors than on ARM, so there is a research gap to be filled here [7]. Hence, this thesis will focus on the ARM architecture.

Seeing if time-critical tasks are impacted by virtualization and disturbances from other guest VMs is crucial to be able to study the safety of the system. Thus, for the guests, an RTOS is of interest since missed deadlines are a serious consequence that could happen because of disturbance. In an automotive context, important tasks are often real-time tasks and can have detrimental consequences for safety if the deadlines are not met. The RTOS ZephyrOS will be used for the testing guest and for the disturbing guest.

1.5 Disposition

This section describes the disposition of the thesis, starting with the current Chapter 1, introduction. Chapter 2 contains the theory that explains all necessary concepts for the thesis. Chapter 3 describes the method and chapter 4 presents the results of the thesis. In chapter 5 the results are discussed and the conclusions are presented.

s

2

Theory

This chapter lays the theoretical groundwork necessary to understand the evaluation of guest isolation in hypervised systems. We begin by exploring the fundamental concept of virtualization and the specific hardware support that enables it, focusing on both x86 and ARM architectures. Subsequently, we delve into the crucial concept of isolation, differentiating between temporal and spatial isolation, which are key to ensuring system integrity and performance in mixed-criticality environments. Following this, we examine hypervisors, the core technology facilitating virtualization, with a closer look at Xen and KVM, the specific hypervisors evaluated in this work. Finally, the chapter discusses RTOS, specifically ZephyrOS, used within the guest environments, and the U-Boot bootloader, providing context for the experimental setup, which will be detailed later.

2.1 Operating Systems

The main purpose of the operating system (OS) is to be able to manage hardware and software. The OS is the interface that the application uses to access the hardware and the OS is also responsible for file management, I/O services, providing a user interface and many other things.

The key functions of the operating system can be divided into *system security*, *data management*, and *hardware interaction* [8]. System security relates mostly to access control, such as who can access what files, but also what parts of memory can be accessed by which processes, and other parts that relate to access control.

Data management is related to security since it relates to file permissions, but also extends to file management systems, directories, and so on. The last part of hardware interaction is also what is at the top of most people's minds when thinking of OSs. This function regards how the CPU, memory, and other physical resources are distributed to all processes. The OS acts as a mediator between the hardware and the software [8].

2.2 Virtualization

Virtualization is the concept of creating virtual representatives of networks, storage, CPUs, and systems. This thesis will primarily use the term virtualization to describe dividing a system into smaller parts such that there are multiple systems running on the same hardware. These systems that share the hardware are usually called *Virtual Machines* or *guests*. The primary goal of virtualization is to utilize the hardware better. For example, when running a single system on some hardware,e it is reasonable to assume

that said system does not use all computational resources available to it. In such a case, it is possible to optimize the hardware usage by running a second system simultaneously.

The concept of virtualization is an old idea that was first introduced in practice by IBM in the 1960s [9]. Today, virtualization is commonly utilized in the form of containerization, cloud computing, and optimization of computation and storage. There exist multiple technologies that make use of virtualization today, e.g, hypervisors, containerization, and microkernels.

In general, there are two types of virtualization: *Full virtualization* or *Paravirtualization*. The difference between them lies in how the guests perceive the hardware. In a fully virtualized system, the component is unaware of the virtualization and thinks it is speaking directly to the hardware. When a guest uses a privileged instruction in a fully virtualized system, a hardware trap catches the instruction and sends it to the hypervisor, as illustrated in Figure 2.1. The hypervisor has a trap handler that processes the instruction and then emulates the instruction for the guest. This way, no modification is needed to the guest, and the virtualization layer is in charge of handling all types of instructions.

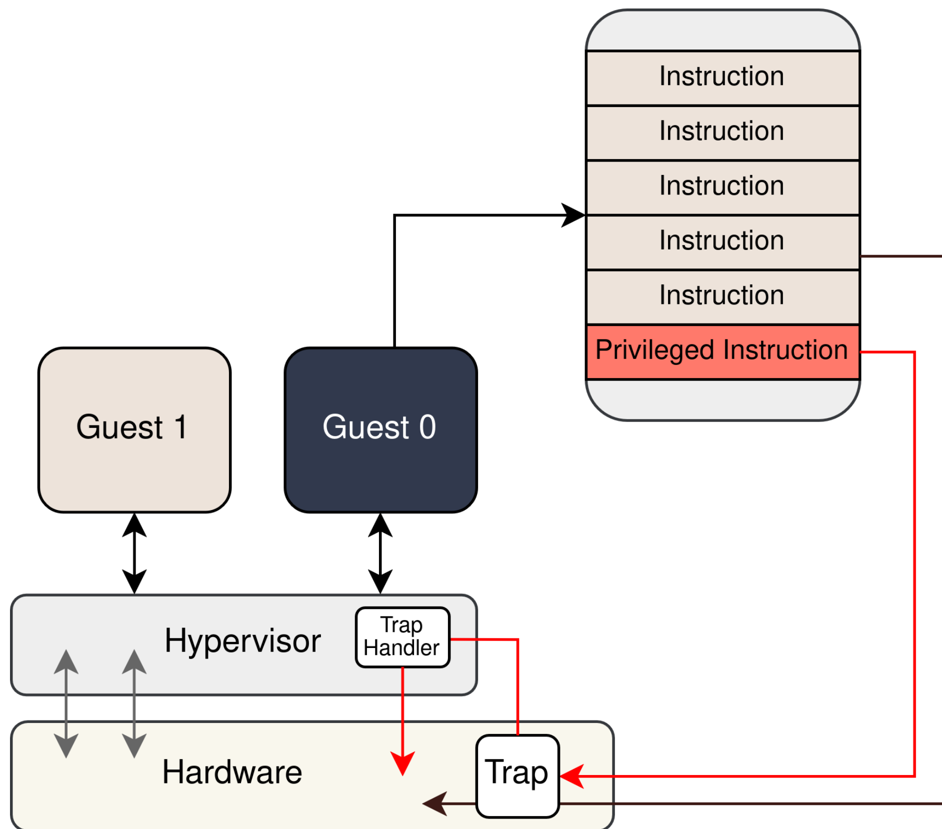

Figure 2.1: Visualization of how a fully virtualized guest communicates with hypervisor and hardware

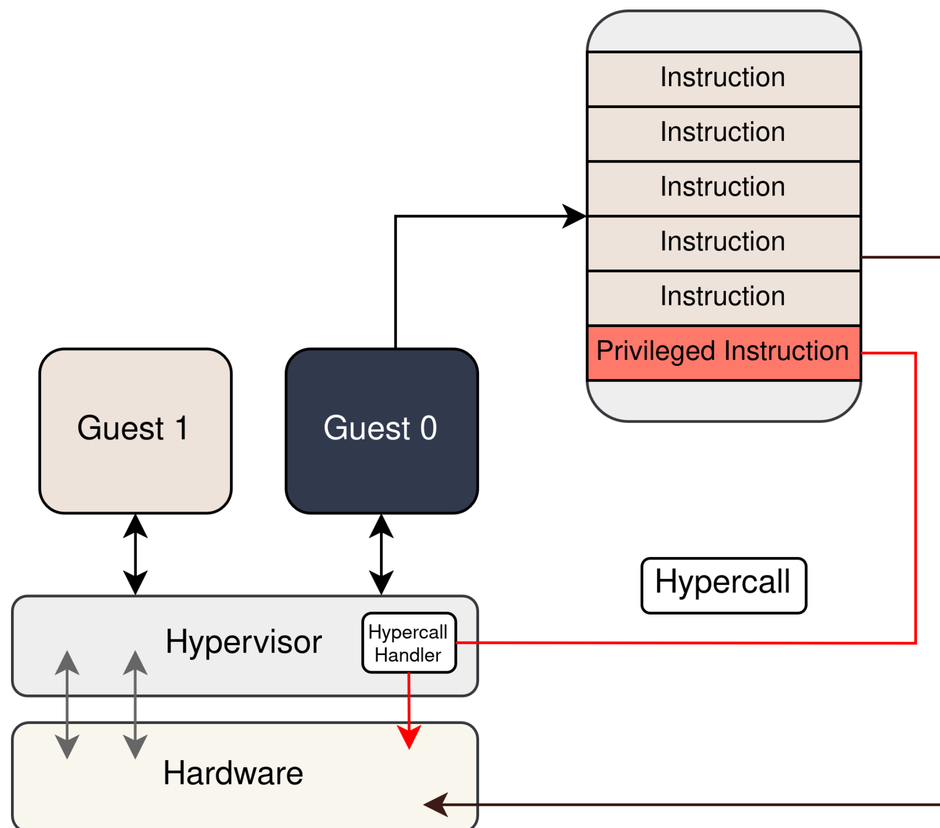

Figure 2.2: Visualization of how a para-virtualized guest communicates with hypervisor and hardware. The privileged instruction triggers the hardware trap that relays the instruction to the hypervisor before executing.

On the other hand, in a para-virtualized system, the guest is aware of the virtualization. When a guest in a para-virtualized system is trying to access a privileged instruction, it sends the instruction directly to the hypervisor with a so-called *hypercall*. A hypercall is the hypervisor equivalent of an OS system call; they are needed to handle privileged instructions in a safe way. Non-privileged instructions still directly access the hardware as can be seen in Figure 2.2. This also means that the guests have to be modified to use para-virtualization and will not work ”out-of-the-box”.

2.3 Hardware support for virtualization

All processors are not able to perform virtualization in an efficient way. Without hardware support virtualization requires emulation which takes a lot of resources usually seen as a significant slow down of the execution.

In the 70’s, the Popek/Goldberg theorems were created that describe the requirements of equivalence, safety, and performance for virtualization [7]. Equivalence means that the virtualized system should seem identical to the underlying processor. Safety means that the VM must be isolated from the hardware and other VMs, and the Virtual Machine Monitor (VMM) has to be in control of the hardware. The performance part states that there should be a minimal decrease in performance so that the VM is efficient.

Their theorems have long been the standard for hardware support for virtualization, even

though exceptions to the rule exist. The first theorem is also the main one.

> ”For any conventional third-generation computer, a VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.” [7]

This is the foundation for having different privilege levels, or modes, which differ between architectures. In the following sections 2.3.1 and 2.3.2, this is explained further.

2.3.1 x86 architecture

x86 is the most common architecture used today for workstations and servers. The architecture is of CISC type, which is an acronym for Complex Instruction Set Computer. This enables easier support for higher-level programming languages since the instructions support more programming constructs compared to less complex instruction sets, for example, loops can be included.

Intel VT-x is the architecture for virtualization implemented by Intel for the x86 architecture that was introduced in 2005. The main idea for supporting virtualization is by introducing *root* and *non-root* modes [7]. The hypervisor and host OS are run in root mode while the virtual machine is run in non-root mode. These modes are completely separate from other modes of execution, such as protected mode and protection levels. This is the solution chosen to support the first Popek/Goldberg equivalence.

The root mode and non-root mode have their own address spaces, and the translation lookaside buffer (TLB) switches address space automatically when the mode changes. The first implementations of VT-x showed that the virtualization is not efficient if the memory management unit (MMU) is not also virtualized [7]. This was solved in later versions by extended page tables, since this removes the need for software to do shadow paging. Extended page tables mean having two sets of page tables, where one is the classical hardware-defined page table. This is managed by the guest OS. The hypervisor manages the other. This enables the TLB to work almost as before, except for when handling misses. Such misses can, in the worst case, take a lot more time since the worst case time is quadratic [7].

2.3.2 ARM architecture

Advanced RISC Machines (ARM) is a family of RISC architectures used for CPUs. RISC stands for Reduced Instruction Set Computers and was designed to make every instruction fast to execute, preferably within one data memory cycle, to be able to pipeline instructions more efficiently. This is in comparison to the x86 architecture, which has more instructions, but in turn can require more cycles per instruction. ARM usage has increased in popularity in recent years, especially within IoT (Internet of Things) because of its adaptability, performance, and energy efficiency [10].

ARM has different privilege levels, also called exception levels, as shown in Figure 2.3. It is the layer 2 that makes the machine able to perform virtualization and this is called the hypervisor mode. The exception levels exist to prohibit programs from accessing things they shouldnt be allowed to. Most processing happens in EL0/1, and the exception level is only allowed to change in certain cases. This is done to limit the privilege of parts of

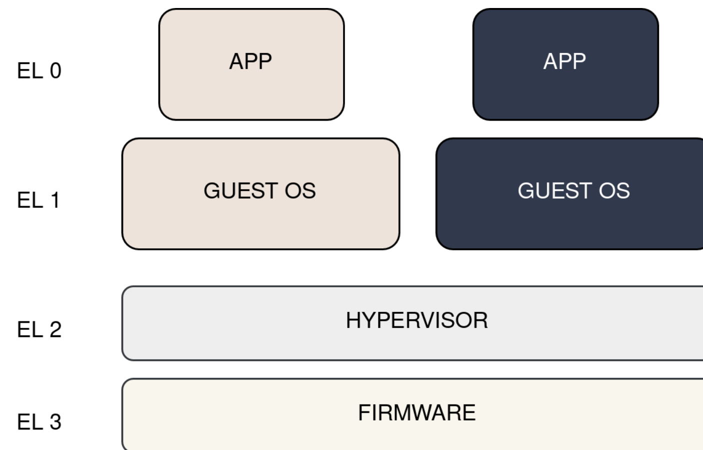

Figure 2.3: Exception levels in ARM architechture

the system where it is not needed. The only time a change in exception level is allowed are during the circumstances listed below.

- Handling an exception
- Returning from an exception
- Processor reset
- During Debug state
- Exiting from Debug state

Using exception levels to handle virtualization differs from the root mode of x86 in a major way because EL 2 is strictly more privileged than EL 1, which is the kernel mode. Root mode exists outside other levels of privilege and these lines are not as defined. This implies that the ARM way is easier to implement since the number of features is less, but also that the x86 may have more functionality because of its more complex states [7].

One of the key features in providing virtualization on ARM is the stage 2 translation. This allows the hypervisor to control the memory mapping, which is important for isolation and sandboxing [11]. The stage 2 translation refers to the memory translation that happens between the virtual address space and the physical address space. In between there is the intermediate physical address space, which is what the operating system thinks is the physical address space, see Figure 2.4. This is similar to how memory virtualization is handled in x86.

One important feature that ARM has implemented to help the hypervisor is the ability to trap operations. Some instructions that a VM tries to execute might break isolation or, in other ways, give the VM more privileges than it should have. The hypervisor has to be able to trap these kinds of instructions to handle the low-level processor controls and emulate them towards the VM without affecting other VMs [11]. This is needed for full virtualization.

The trap triggers an interrupt that can change the exception level and handle the instruction. To enable this functionality, the most common tool is to use ARM's *Generic Interrupt Controller* (GIC). This can handle both virtual and physical interrupt signals.

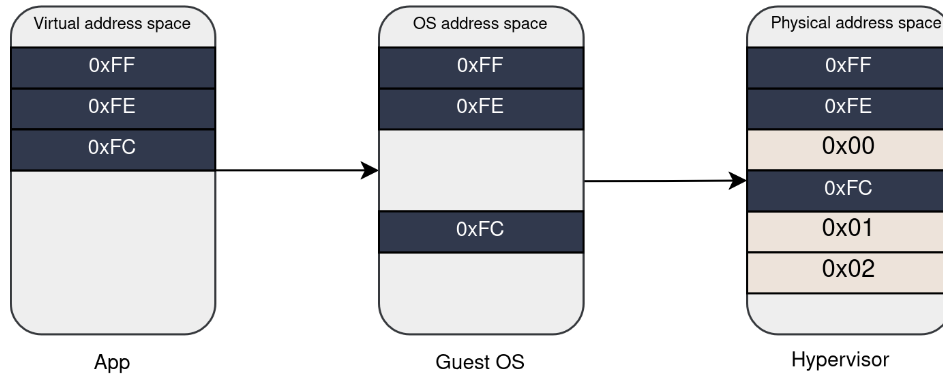

Figure 2.4: Stage 2 translation used in ARM architecture to virtualise memory addresses

The hypervisor can set up the interface between the VMs and the GIC, and thus each interrupt does not have to pass by the hypervisor. This reduces the overhead of interrupts.

2.4 Isolation

The fundamental goal of isolation is to keep different software components from interfering with each other's operations. Within virtualization, this principle is paramount. It means that each VM should operate as if it has exclusive access to the underlying hardware it has been allocated, irrespective of other VMs concurrently running on the same physical machine. The hypervisor is responsible for defining and enforcing strong separation polices and boundaries between these VMs [12].

Isolation is important for a number of different reasons. As mentioned, performance is a key factor; isolation prevents the resource demands of one VM from negatively impacting the responsiveness and throughput of others. Equally important are safety and security. Isolation ensures that errors or security breaches within one VM are contained and do not compromise the integrity or confidentiality of other VMs. The two primary types of isolation are typically separated as temporal and spatial isolation [13].

2.4.1 Temporal Isolation

Temporal isolation regards how well a hypervisor allocates and schedules shared resources between guests. Temporal isolation can also be referred to as performance isolation. The aim is to protect processes from being delayed by the interference of other processes, to always provide reliable performance. In a virtualized environment, resources such as CPU time, network bandwidth, and disk I/O are shared. If temporal isolation is not sufficiently achieved, one VM could use up resources in such a way that it delays other VMs, as displayed in Figure 2.5.

2.4.2 Spatial Isolation

Spatial isolation is sometimes also called memory isolation and refers to code and data only being accessible from the correct application, as shown in Figure 2.6. In a virtualized

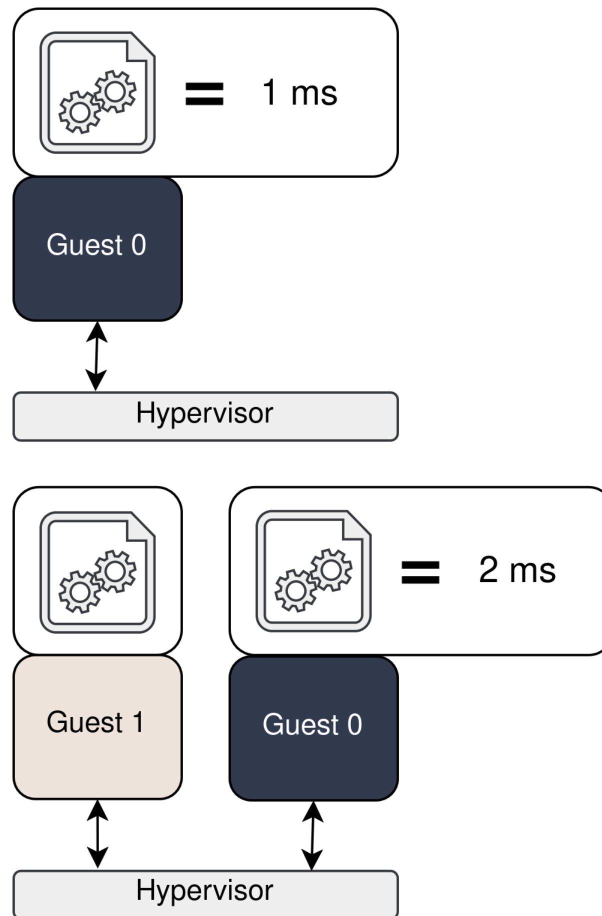

Figure 2.5: Visualization of broken temporal isolation. In this case, the execution of guest 1 impacts the time it takes for guest 0 to execute its program.

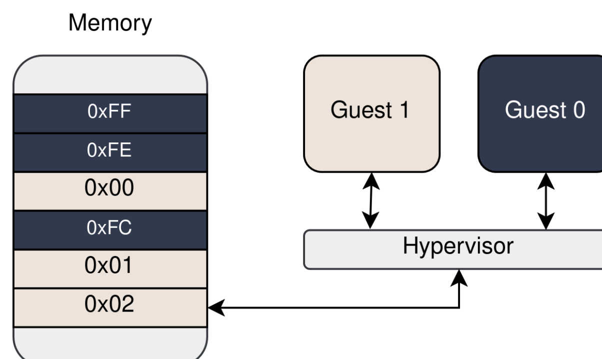

Figure 2.6: Visualization of working spatial isolation where the hypervisor ensures that the memory is mapped to the different guests without interference.

context, this ensures that the virtual memory addresses are mapped such that they never interfere with each other in the physical memory. Furthermore, a guest should not be able to gather information about other guests' memory allocation and memory addresses.

2.5 Hypervisors

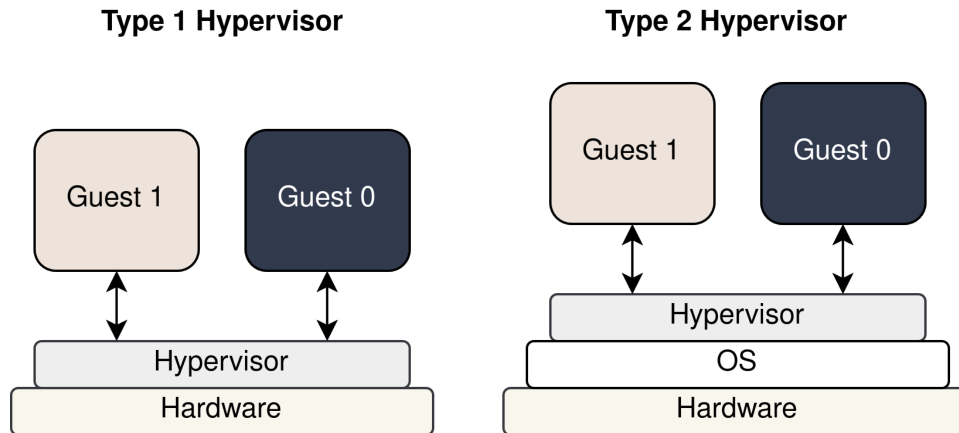

Figure 2.7: Visualization of type 1 and 2 hypervisors

Hypervisors or VMMs are software that handle virtualization. There are two types of hypervisors, and they handle virtualization in different ways. This can be viewed conceptually in Figure 2.7. *Type 1* is also called a bare metal hypervisor and is the first layer on the hardware that several operating systems can be hosted on. On the other hand, the *type 2* hypervisor sits on top of a host operating system, and all communication with the hardware has to go through the host operating system.

Type 1 hypervisors can be very close in performance to running on bare metal, as shown by Abeni et al. [14]. This makes them useful in the embedded use case where the system is more sensitive to large overheads, especially in real-time or mixed-criticality systems.

2.5.1 Xen

Xen [15] is an open-source type 1 hypervisor that can support both hardware virtualized instructions and paravirtualization. In Xen, virtual machines are called *domains*. The design is to have one VM that controls the others. The control domain is called dom0, and the other VMs are called domUs. Dom0 is usually a Linux kernel and is more privileged. It is mainly used to manage the domUs, see illustration of this in Figure 2.8.

Since Xen is a general-purpose hypervisor, it supports many different types of use cases, everything from server virtualization to embedded real-time systems. Originally, Xen was written for x86, and many differences were made when porting it to ARM. The most significant change is that the ARM version is much smaller and requires no emulation which makes it faster and safer [16]. Another difference to the version on x86 is that the VMs can no longer be of either paravirtualized or fully virtualized type, now all guests are a mix of both thus making the codebase less complex. This is explained by the Xen on ARM whitepaper [16] that the I/O interfaces are paravirtualized to avoid emulating hardware, while hardware virtualization extensions are used as much as possible in a fully virtualized way to avoid changing the kernel as much when adapting it to the hypervisor.

Xen can be used with real-time guests and supports the scheduling algorithm RTDS, a version of the Earliest Deadline First (EDF) algorithm. The algorithm works by scheduling a task on a virtual CPU (vCPU) with the earliest deadline to a physical CPU (pCPU)

in a global EDF fashion [17].

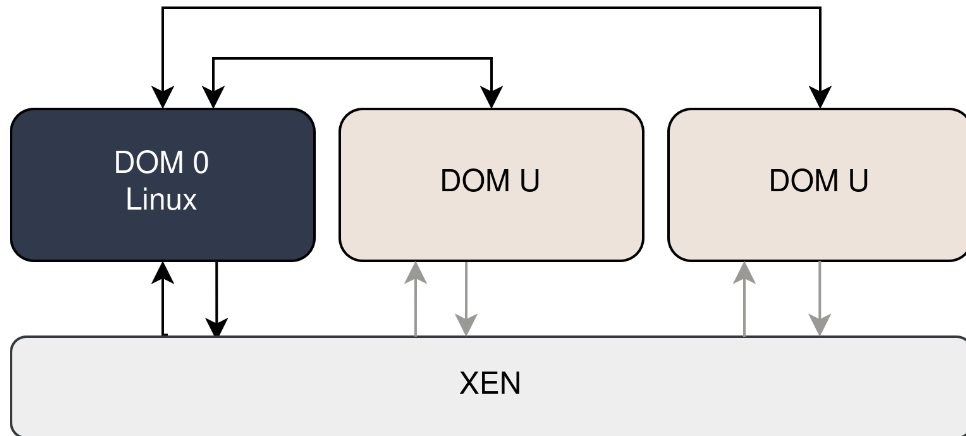

Figure 2.8: Overview of the Xen hypervisor architecture. Dom0 manages DomUs and provides access to hardware resources, but DomUs can also interact directly with the hypervisor via hypercalls, reducing the need for all drivers to be routed through Dom0.

2.5.2 QEMU/KVM

KVM is short for Kernel-based Virtual Machine and it is an open-source hypervisor that is part of the Linux kernel, as shown in Figure 2.9. KVM is a type 1 hypervisor in the sense that it makes the Linux kernel act as a hypervisor. There are several different interfaces that exist to interact with KVM. KVM itself adds next to nothing in performance delays compared to bare metal but the interfaces to interact with KVM can have performance degradations.

QEMU is one example of a tool that acts as a type 2 hypervisor and is using KVM underneath. There are other interfaces to interact with KVM such as libvirt and virt-manager. KVM needs QEMU to perform I/O emulation. On its own QEMU is a fully functioning emulator but together with KVM it accelerates QEMU to work as a hypervisor.

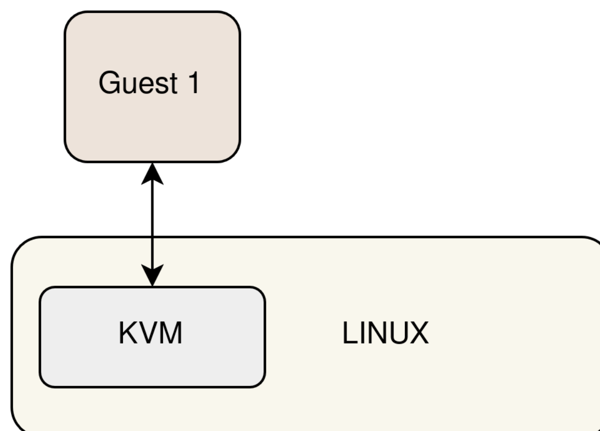

Figure 2.9: KVM hypervisor

KVM is designed with the perspective to be on top of hardware, with virtualization to be as efficient as possible. It only handles CPU and MMU virtualization and multiplexing,

leaving more complex tasks to QEMU and the Linux kernel. KVM is written as a part of the Linux kernel to minimize overhead and avoid redundancies [7].

KVM was originally created for the x86 architecture. The porting of KVM to ARM was made around 2014 by Dall and Neith [18]. Since ARM exists in many different variants, a bare metal KVM implementation that is compatible with all can be very complex; therefore, KVM was not rewritten to a higher degree but kept as part of the Linux kernel since the kernel already is compatible with almost all systems [7].

Split-mode virtualization was chosen for KVM partly because it takes advantage of the ARM hardware virtualization extensions. KVM uses the hypervisor execution level to handle sensitive instructions and hardware interrupts. To avoid having to run the entire kernel in EL2, which would have been difficult to implement, the hypervisor is split into two parts. Only the routines that need to be run in EL2 is run there, called the *lowvisor*, while the rest is run in EL1 and is called the *highvisor*. This is the foundation of KVM on ARM.

2.6 Real Time Operating Systems

Real-time operating Systems are used when the system depends on tasks being executed on time. In contrast to a General Purpose Operating System (GPOS) where things are more made for multi-tasking and user interaction, a RTOS is usually specified to a specific job and is useful in a niche embedded environment.

RTOS can be of two kinds, either a hard or soft RTOS. Hard RTOS are used for critical applications where timings absolutely cannot be missed, where one example is medical devices such as pacemakers. The soft RTOS still have deadlines that it needs to be met but the constraints are a bit more relaxed because the tasks are not as crucial, for example in multimedia systems.

Scheduling is what makes the operating system work in real time, pre-emptive scheduling is used to ensure timing constraints. This means that tasks might be interrupted and continued later to allow other tasks to execute.

One of the most common scheduling algorithms is called *Earliest Deadline First* (EDF) which is a dynamic priority scheduling algorithm. The algorithm chooses the task to run with the deadline that is first about to expire. This can be changed at any point and a task will be pre-empted, put on pause, to give way for a new task with an earlier deadline that has arrived. This is illustrated in Figure 2.10.

An example of an RTOS is Zephyr OS[19], which is an open-source RTOS hosted by the Linux Foundation with a strong emphasis on security and connectivity. It supports multiple hardware and is scalable, which makes it possible for developers to tailor it to their specific needs [19]. The ZephyrOS kernel is a small-footprint kernel, which means that it uses a smaller amount of memory and is targeted towards embedded systems or other resource-constrained systems.

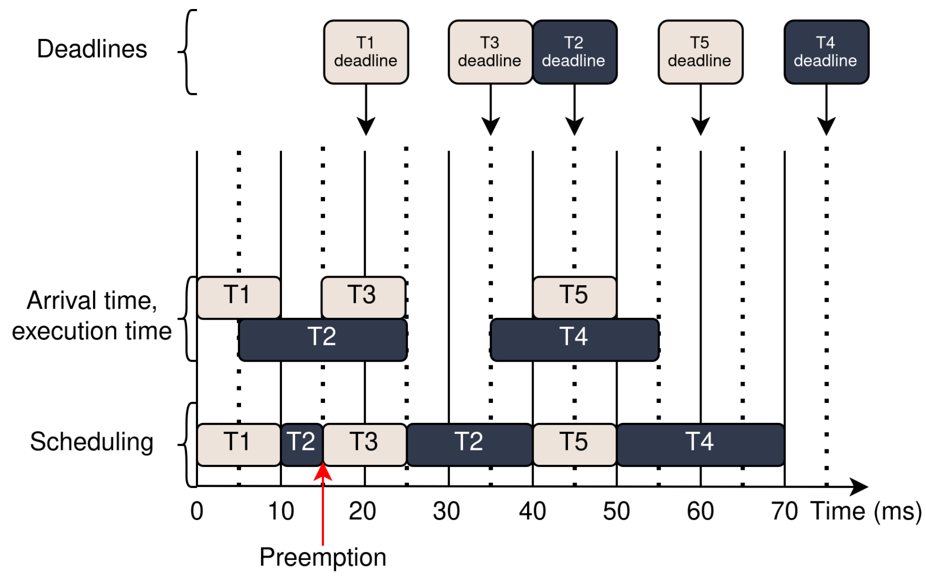

Figure 2.10: EDF-sheduling algorithm with pre-emption example. T2 is pre-empted at 15 ms in favour of T3 which has an earlier deadline. T2 is completed afterwards.

3 Methodology

This chapter presents how the testing framework is set up to compare the performance of hypervisors. Figure 3.1 shows the idea of the test setup. The varying factor is the hypervisor, while all other parameters are as similar as possible. Since the aim is to measure the performance of the hypervisor regarding isolation, the two guests are set up in a way that one is the measuring guest performing a test and the other is a stressor. This way, disturbances can be measured and compared between hypervisors; the details are explained in section 3.1.

Different tests are created using the ZephyrOS test framework in these areas. A stressor is also developed as its own zephyr image. Both images are run simultaneously on a hypervisor, which in turn runs on the Raspberry Pi. The testbed is described in section 3.2 and the implementation of hypervisors and the stressor is described in section 3.3.

The two hypervisors that are evaluated with the test framework are Xen and QEMU/KVM. The framework is aimed at testing latencies on low level operations, the scheduling of threads, and finally similar measurements as in *MiBench* which is a benchmark suite designed for automotive [20]. The different tests are explained in detail in section 3.4.

3.1 The Design of Experiment

The foundation of the experiments is to have different hypervisors run on top of ARM hardware and using test frameworks to measure their performance, see Figure 3.1. The hypervisor runs two VMs, with one being the measuring guest and the other one, called the stressor or the disturbing guest, running intensive tasks with the purpose of measuring if this has an impact on the first VM.

To isolate the evaluation of the hypervisors, the VMs run as identical an image as possible for each test area across the two different hypervisors. Despite this, the images have to be built for different targets because of the different hypervisors. For each test area, the guest is being run as the only guest on the hypervisor, as well as being co-run with another guest that is running as a stressor. For certain test areas the stressor is adapted to tailor towards stressing the same area as the test is evaluating. For each test configuration, the measuring guests is also run on its own, "solo". The solo evaluation is being run on a single core, and when running together with the stressor, the guest and stressor are being run on independent cores. For some test setups, the implications of running both VMs on the same core are also investigated.

For a fair comparison, the system is rebooted in between tests to ensure that it is in as close to an identical state as possible at the start of each test run. All tests are repeated

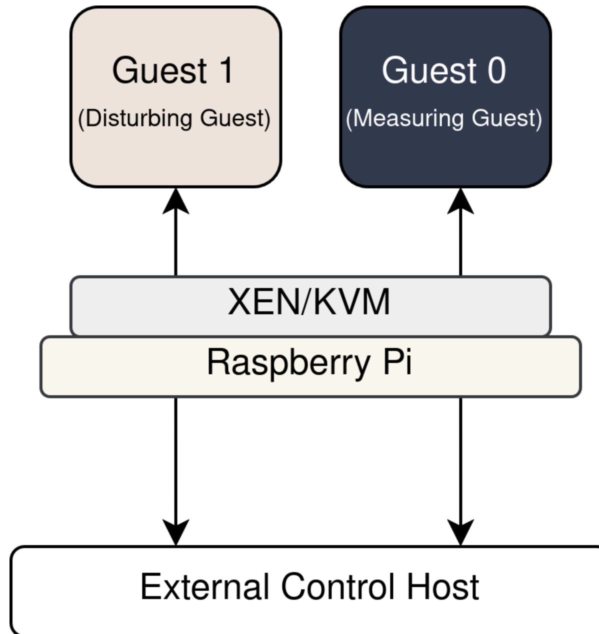

Figure 3.1: Overview of the test setup

multiple times for validity, although there are variations in the number of times depending on test setup; this is explained further in section 3.4.

3.2 System on Chip

The hardware requirements to be able to perform the evaluation are that the processor is of ARM type with support for virtualization. For this, the chosen board is Raspberry Pi 4B with the following specifications.

- **Processor:** Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
- **Memory:** 8GB LPDDR4-3200 SDRAM
- **Networking:** Gigabit Ethernet
- **USB Ports:** 2 USB 3.0 ports and 2 USB 2.0 ports
- **Storage:** Micro-SD card slot for operating system and data storage

Each core of the Broadcom BCM2711 includes a 32 KB L1 data cache and a 48 KB L1 instruction cache. In addition, the cores share a 1 MB L2 cache. As this L2 cache is a shared resource, it was taken into account when designing certain experiments. The cache hierarchy is illustrated in Figure 3.2.

3.3 Implementation

The hypervisors require different implementations to get them up and running; this is explained further in the respective sections 3.3.2 and 3.3.3. To be able to later start a

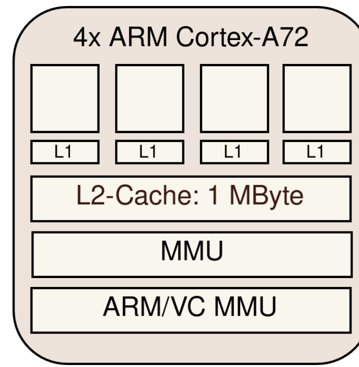

Figure 3.2: Illustration of core and cache hierarchy on the Broadcom BCM2711 chip.

VM, the process is similar across both hypervisors. ZephyrOS is built to run a test from a host and then upload it to the file system to be able to start a VM.

3.3.1 ZephyrOS

ZephyrOS is the RTOS chosen for both the testing and disturbing VMs. To build ZephyrOS images, the official build tool `west` provided by the Zephyr Project [19] is used. The Zephyr Project includes predefined board configurations for both Xen and QEMU-based ARM targets, named `xenvm` and `qemu-kvm-aarch64`, respectively.

No modifications are required for the `xenvm` target used with Xen. However, the `qemu-kvm-aarch64` target is configured by default to use GIC version 3, which is incompatible with the Raspberry Pi 4B hardware, as it supports only GIC version 2. To address this incompatibility, a custom device tree overlay was created. This overlay overrides the default GIC configuration and enforces the use of GIC version 2 to ensure compatibility with the Raspberry Pi 4B platform.

The `west` tool provided by the Zephyr Project is a python-based tool to simplify building ZephyrOS images for a large selection of hardware boards. The `west` tool makes cross-compiling and niche builds as for hypervisors simpler. Furthermore, Zephyr has several pre-implemented benchmarks which can be customized by modifying their source code and easily rebuilt with west.

3.3.2 Xen

Xen was compiled natively from source on an ARM-based cloud server. This was necessary due to failed builds when cross-compiling the Xen-tooling software. To boot Xen a kernel is needed and the latest Linux kernel version 6.1.18 was compiled natively as well. A rootfs for dom0 was generated using the tool debootstrap [21]. Xen-tooling was installed on the rootfs using the generated install script to compile Xen. This is needed to get access to the Xen management tool CLI xl. A Raspberry Pi 4B device tree blob (dtb) was used to enable booting for the given hardware. For booting, U-boot was chosen for its modularity and ease of reproducibility; the full boot script can be seen in appendix A.1. The U-boot binary used was extracted from an image with a pre-built binary for Raspberry Pi [22].

```c
void stress_test(void *arg1, void *arg2, void *arg3) {
    while (1) {
        /* Do nothing but keep CPU busy */
    }
}

K_THREAD_DEFINE(stress_thread, STACK_SIZE, stress_test,
               NULL, NULL, NULL,
               STRESS_PRIORITY, 0, 0);
```

Listing 3.1: Stress test thread definition in C

```c
#define BUFFER_SIZE (L2_CACHE_SIZE + (L2_CACHE_SIZE / 2)) // 1.5 MB
static char buffer[BUFFER_SIZE];
int main(void) {
    while (1) {
        memset(buffer, 'A', BUFFER_SIZE);
    }
}
```

Listing 3.2: Stress test thread definition with cache impact in C

3.3.3 QEMU/KVM

For the KVM/ARM hypervisor, we use QEMU, installed on a minimal Ubuntu host OS targeting the Raspberry Pi 4B board. The QEMU qemu-system-aarch64 emulator is employed, leveraging KVM for hardware acceleration of the virtual machines. To pin CPUs to VMs, we utilize the taskset(1) tool [23], ensuring each VM ran on its own dedicated core by pinning its corresponding QEMU process.

3.3.4 Stressor

There are two versions of the stressing guest, one only affecting the CPU usage and one also affecting the cache. To perform as high of a CPU load as possible from the stressing guest a thread with the highest priority is created which loops infinitely while doing nothing. This means the guest always has a 100% CPU load when executing, where the details can be seen in Listing 3.1.

The stressing VM is also affecting the cache, as shown in the contents of Listing 3.2. It fills up the cache in an attempt to create cache misses for the testing VM. This stress VM is used for the qsort test, where the cache performance is evaluated; otherwise, the first stress VM is used.

3.3.5 Measuring VM

The measuring VM is responsible for quantifying the performance characteristics of the guest, providing data from various tests to analyze the temporal isolation properties of the hypervisor. To achieve this, the measuring guest runs a tailored instance of ZephyrOS executing specific test cases. The different test variants are further detailed in Section 3.4.

The measuring VM is deployed as its own guest, either co-located with the disturbing guest or isolated on a separate core. According to our industry partners, the latter configuration, using separate cores, is of greater interest, as it reflects a common real-world approach to reducing shared resources for critical systems.

3.4 Test Framework Areas

In this section the different tests performed are described. There are three different tests aimed to complement each other. The areas are latency measurments to evaluate performance on operating system level, MiBench tests to evaluate application level and see impact on cache contention, and thread metrics to evaluate scheduling over a longer time period.

3.4.1 Latency measurments

To evaluate temporal performance, this thesis utilizes the latency benchmark suite provided by the Zephyr Project RTOS. These benchmarks are implemented under the official Zephyr repository **zephyr_latency_measure**.

The latency measurements focus on a range of low-level kernel operations that are critical for real-time systems. Since these operations form the foundation of task scheduling and synchronization in an RTOS, analyzing their execution times helps determine whether a disturbing guest introduces unacceptable delays with respect to automotive RTOS deadlines.

Each benchmarked operation typically completes within a very small number of processor cycles, ranging from approximately 2 to 200 cycles, depending on the specific test. The goal is to detect latency anomalies introduced by a disturbing guest.

Figure 3.3 provides an overview of the measured operations included in the benchmark. These operations span context switches, thread life cycle management, interrupt handling, and common synchronization primitives.

- Context switch time between preemptive threads using k_yield
- Context switch time between cooperative threads using k_yield
- Time to switch from ISR (Interrupt Service Routine) back to interrupted thread
- Time from ISR to executing a different thread (rescheduled)
- Time to signal a semaphore then test that semaphore
- Time to signal a semaphore then test that semaphore with a context switch
- Times to lock a mutex then unlock that mutex
- Time it takes to create a new thread (without starting it)
- Time it takes to start a newly created thread
- Time it takes to suspend a thread
- Time it takes to resume a suspended thread
- Time it takes to abort a thread
- Time it takes to add data to a fifo/LIFO
- Time it takes to retrieve data from a fifo/LIFO
- Time it takes to wait on a fifo/lifo (and context switch)
- Time it takes to wake and switch to a thread waiting on a fifo/LIFO

- Time it takes to send and receive events
- Time it takes to wait for events (and context switch)
- Time it takes to wake and switch to a thread waiting for events
- Time it takes to push and pop to/from a k_stack
- Measure average time to alloc memory from heap then free that memory

Figure 3.3: A list of latency measurement items sourced from the Zephyr Project RTOS benchmarks [24]

These measurements allow for a detailed understanding of how virtualization affects real-time responsiveness, which is particularly relevant in safety-critical domains such as automotive systems.

3.4.2 MiBench

MiBench is a benchmark suite specifically designed for evaluating the performance of embedded systems [20]. It provides a diverse set of workloads grouped into six categories: Automotive and Industrial Control, Network, Consumer, Office, Security, and Telecommunications. Each category contains programs representing real-world applications typically found in embedded environments.

For our evaluation, we focused on the Automotive and Industrial Control subset, which includes four benchmarks: basicmath, bitcount, qsort, and susan. Out of these, only basicmath and qsort were successfully ported to ZephyrOS and used to evaluate the performance of the selected hypervisors.

Basicmath is a compute-bound benchmark that performs a series of simple mathematical computations such as computing square roots, solving cubic equations, and performing integer-based arithmetic operations. This benchmark required only minor modifications to be integrated as a ZephyrOS guest application, making it an ideal candidate for performance evaluation under the chosen hypervisors.

Qsort is a sorting benchmark that tests memory access and algorithmic efficiency by sorting large arrays of integers. It uses an input data set of 1171 KB, which exceeds the available size of both the L1 and L2 data caches on the Broadcom BCM2711 (see Figure 3.2). This makes Qsort particularly useful for evaluating memory management and caching behavior under constrained embedded conditions. Its cache-intensive nature allows us to observe how hypervisors handle larger working sets and cache thrashing scenarios in practice.

By using both basicmath and qsort, we aim to capture both compute- and memory-bound performance aspects in our hypervisor benchmarking. The selection aligns with the goals of MiBench, which is to provide representative and relevant workloads for embedded systems benchmarking [20].

3.4.3 Thread metrics

In the Zephyr benchmarks, there is also a benchmark for thread metrics [25]. There are several different tests for assessing RTOS performance. The setup of the test is simple; it counts and prints the number of times an RTOS event is executed in a set time span. Five seconds were chosen for this experiment.

The test used is for pre-emptive scheduling. This consists of 5 threads that take turns incrementing a counter and handing over to the next thread, as can be seen in the code block Listing 3.3. This is an excerpt from the thread metrics pre-emptive scheduling test that shows what each thread is executing [25]. The counters are compared to ensure that all threads are prioritized fairly, and the total sum of all counters is printed after the check is passed; otherwise, the test throws an error. The sum printed represents the total number of times a context switch has happened within the latest 5-second period.

Configs	**Thread_metric start**	**Stress start**	**Core**
test first same core	first	second	same
test first diff core	first	second	different
stress first same core	second	first	same
stress first diff core	second	first	different
solo	first	–	–

Table 3.1: Example configuration table with thread metric and stress start times, and assigned core. The last row is when the thread metric is executed without any stress

Two types of tests are performed with the pre-emptive scheduling thread metrics benchmarks. Firstly, a test is set up that runs for over 10 minutes and is done with five different configurations. The different configurations start the test and the disturbing guests in different orders, as well as running the guests on the same or on different cores. Furthermore, the test is performed three times for each configuration on both QEMU/KVM and Xen. The different configurations can be seen in Table 3.1.

The second type of test expands on the last case, where the stress is started during a test run and carried out over 30 minutes. The first 10 minutes of the test are running without disturbance. At the 10-minute mark, the stress VM is started on a different core and runs in parallel for 10 minutes, and for the last 10 minutes, the stress is turned off so that the test is run solo again. The 10-minute intervals were chosen to give the test time to stabilize and allow more time for rare disturbances to happen. The 30 minutes in total constitute one run; this is repeated 10 times for each hypervisor. For Xen, it is possible to use a script to start and stop the guests, but for QEMU/KVM, this has to be done manually, which impacts the exactness of the timings.

```c
    void tm_preemptive_thread_1_entry(void *p1, void *p2, void *p3) {
        (void)p1;
        (void)p2;
        (void)p3;

        while (1) {

                /* Resume thread 2. */
                tm_thread_resume(2);

                /*
                 * We won't get back here until threads 2, 3, and 4 all execute
                     and self-suspend.
                 */

                /* Increment this thread's counter. */
                tm_preemptive_thread_1_counter++;
                /* Suspend self! */
                tm_thread_suspend(1);
        }
}
```

Listing 3.3: Pre-emptive scheduling test for threads. This shows the operations done by each thread, where they all hand over to other threads, increment a counter and suspends their own thread to measure the scheduling. The metrics printed is the counter for every 5s interval.

4

Results

In this chapter, the results of the various test areas are presented and analyzed. These include the Zephyr Projects latency benchmarks [24], the automotive subset of MiBench, specifically the Basicmath and qsort tests, and Zephyrs thread metrics. Each test area is evaluated under different stress configurations across the two hypervisors. Overall, the results indicate that QEMU/KVM consistently outperforms Xen in terms of having the shortest execution time. However, when it comes to performance isolation between guests, the picture is less clear: each hypervisor demonstrates strengths in different scenarios. The following sections provide detailed insights into each test area: latency measurements in section 4.1, MiBench benchmarks in section 4.2, and thread metrics in section 4.3.

4.1 Latency Measurments

In figures 4.1 through 4.4, the most significant results of the latency measurement benchmarks [24] are presented. In Appendix B plots for all test from the latency measurement suite are shown, since not all 16 tests are of equal interest to the thesis. The figures describe the distribution of cycles that the latency measurements have taken over a number of iterations with different stress configurations. For both Xen and QEMU/KVM, one test was done with the measuring guest pinned to a single CPU with no other guest inducing stress to the hypervisor. Furthermore, two tests were done on a disturbed, stressed guest. One was pinned to a different core than the measuring guest, and another was pinned to the same core as the measuring guest. For all instances, the disturbing guest was started before the measuring guest to ensure stress during all times of evaluation.

QEMU/KVM has a shorter execution time in almost all cases and also seems to be more consistent with a smaller spread of execution time for each iteration. This higher performance is consistent across other test areas as well, which is explained further in sections 4.2 and 4.3. Most tests provide a statistically insignificant difference in mean execution time under disturbance compared to running undisturbed on a core for both Xen and QEMU/KVM. Examples of such behavior can be seen in appendix B. On some tests, there are outliers with a significantly slower execution time compared to the average execution time, in particular the tests shown in Figures 4.1 through 4.4.

The outliers presented in the figures show an increase from the average values in the range of 10s of cycles to rare occurrences where this takes up to 542,000 cycles. This magnitude of reduced performance is only seen on the Xen hypervisor and only on singular occasions. Figures 4.1, 4.2 and 4.3 handle events between threads with Figs 4.1 and 4.3 also including the time for a context switch. Figure 4.4 measures the time to context switch when yielding using *k_yield* thread operation.

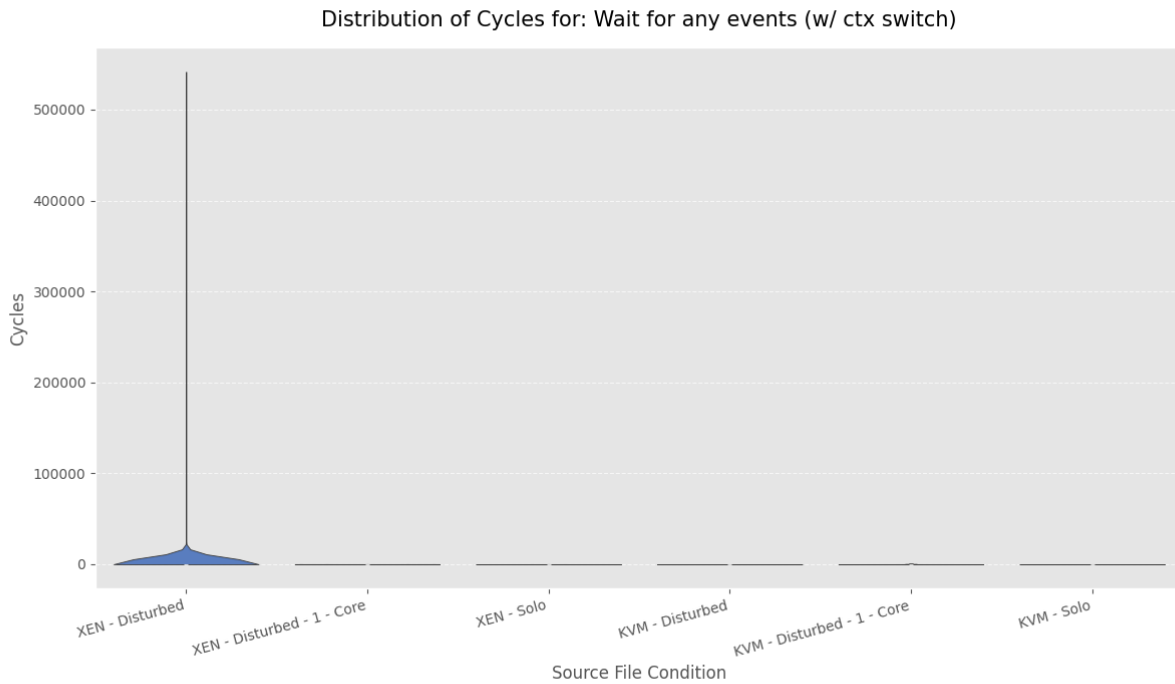

Figure 4.1: Distribution of the number of cycles it takes to wait for any event to occur, with a context switch.

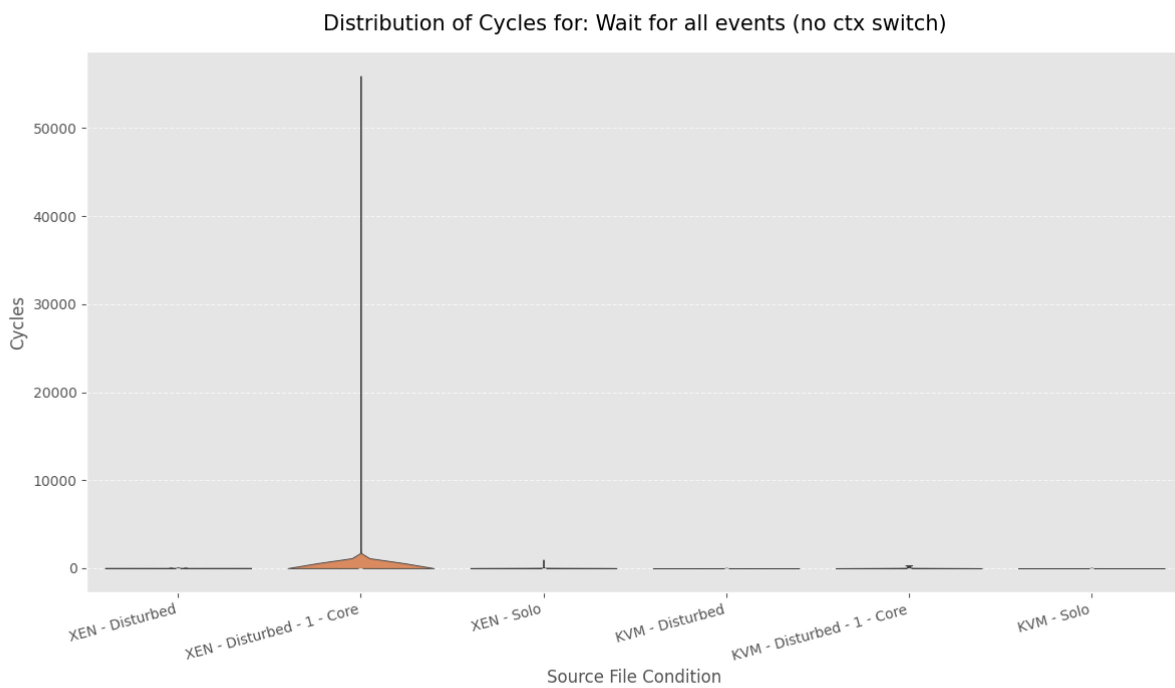

Figure 4.2: Distribution of the number of cycles it takes to wait for all events, no context switch happening.

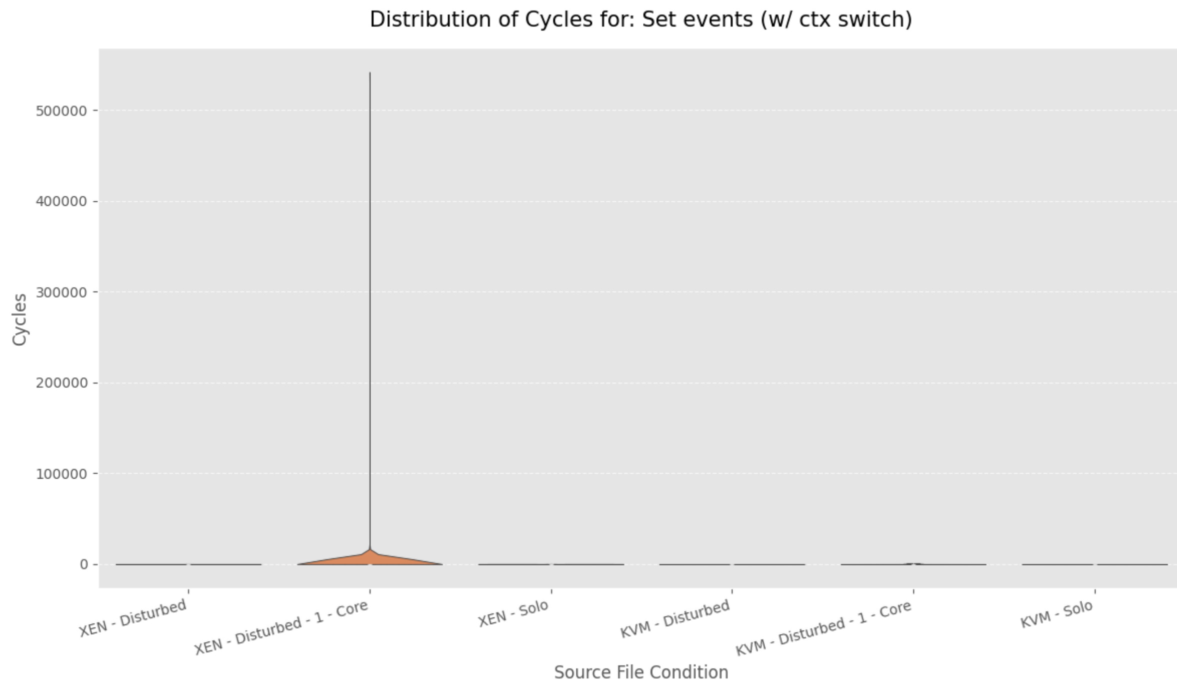

Figure 4.3: Distribution of the number of cycles it takes to set events with a context switch.

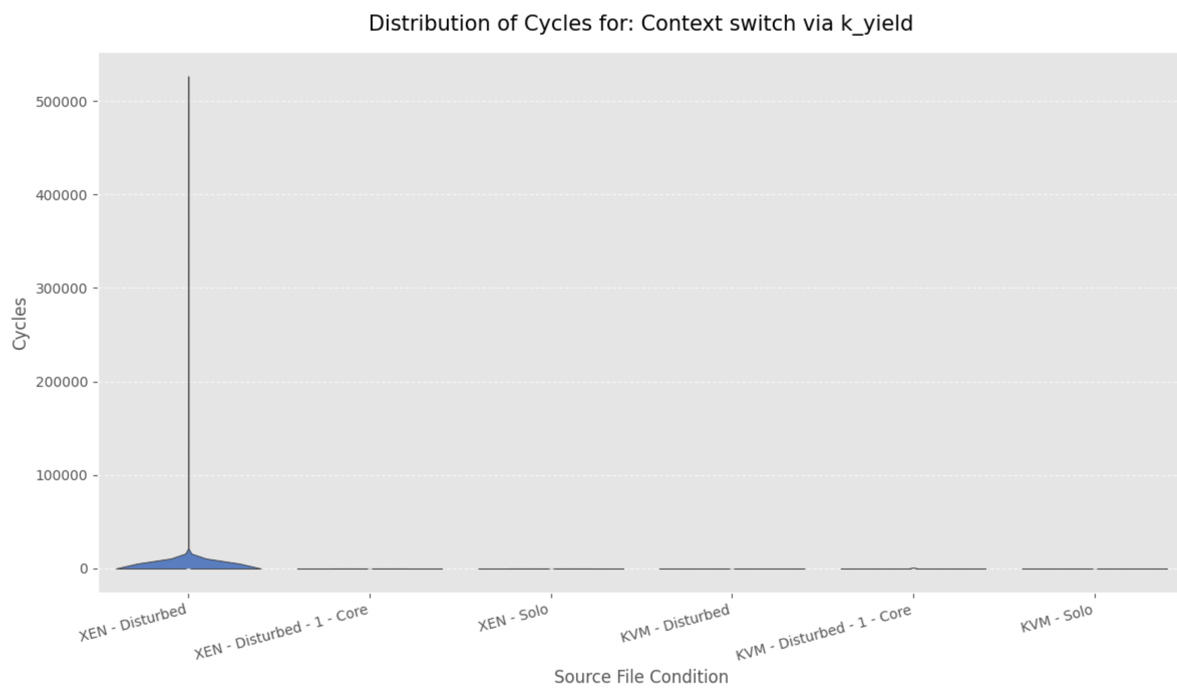

Figure 4.4: Distribution of the number of cycles it takes to perform a context switch when yielding a thread.

QEMU/KVM, on the other hand, has some other noteworthy, but minor, disturbances that happen especially when the stressor is running on the same CPU as the test. This is on the scale of a factor 10 to in some cases a 100. There are also outliers happening to a smaller degree, even when the test is running without disturbance. This is seen mostly on Xen hypervisor, and in general, the solo version is equally unstable as the disturbed versions for these tests. Only in Figure B.2 is there an outlier only on the solo version of a test.

4.2 MiBench

For the MiBench tests, specifically qsort and basicmath, each was executed on both Xen and QEMU/KVM hypervisors under three different stress configurations. The first configuration, serving as a baseline, involved running the test in isolation on a single core pinned to the test VM. The other two configurations introduced interference from an additional "disturbing" guest: one where the disturbing guest was pinned to a different core, and another where both the measuring and disturbing guests were pinned to the same core. In both interference scenarios, the disturbing guest was started before the measuring guest to ensure a consistent load throughout the test duration. Execution time distributions (in cycles) for multiple runs across the configurations are visualized using violin plots, showing the mean, spread, and outliers.

4.2.1 Qsort

Figures 4.5, through 4.9 present the results for the qsort test, which measures the time required to sort a fixed dataset using the standard qsort algorithm. Each run executes multiple iterations, reporting execution time for each. All six stress configurations are visualized using different colors.

Overall, QEMU/KVM demonstrates better performance, both in absolute terms and across comparable configurations. In both Xen and QEMU/KVM, a significant degradation in performance is observed when the measuring and disturbing guests are pinned to the same core. However, this degradation is more pronounced in Xen than in QEMU/KVM, as seen in Figure 4.5.

In Figure 4.6, a slight but noticeable performance drop is also seen when the disturbing guest is on a separate core in Xen, as compared to the baseline. The baseline on the other hand has an outlier with the worst performance of both configurations. For QEMU/KVM, an interesting behavior is observed: the configuration with the disturbing guest on a different core consistently outperforms the baseline (solo) configuration as seen in Figure 4.7. This anomaly may be explained by a ramp-up time required when the solo QEMU/KVM guest starts, as illustrated in Figure 4.8. Even after accounting for this ramp-up period, as shown in Figure 4.9, the disturbed guest still performs marginally better. This can also be seen in the results of the thread metrics measurements in section 4.3.

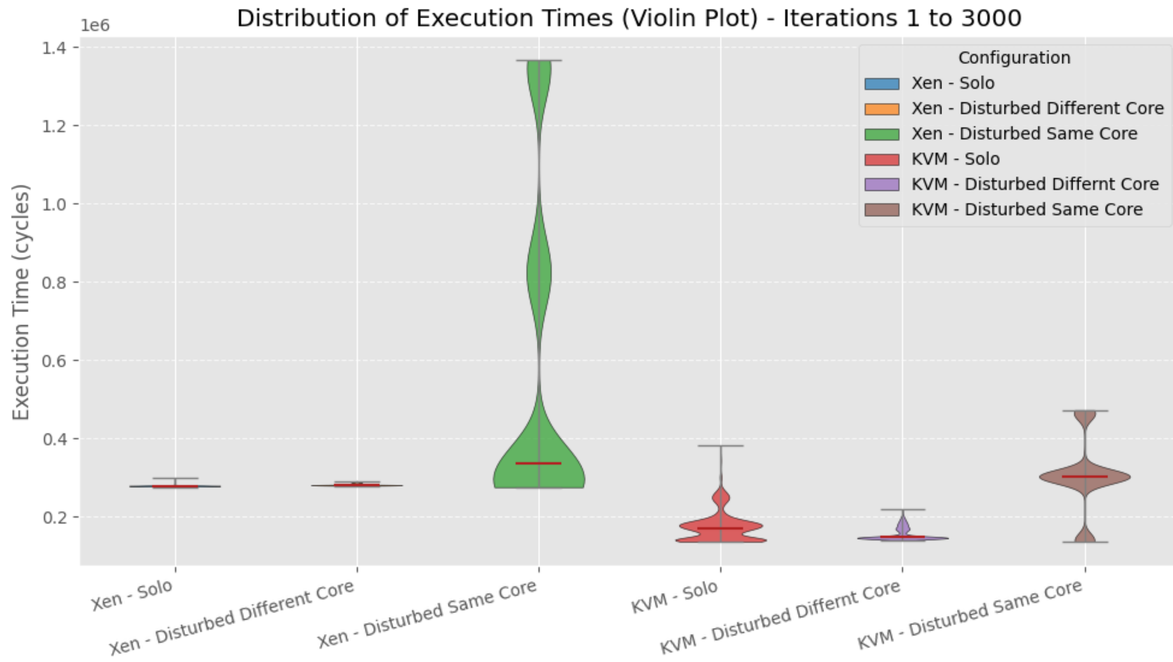

Figure 4.5: Qsort: Distribution of qsort execution time (in CPU cycles) across six system configurations using Xen and QEMU/KVM hypervisors, measured under various stress conditions.

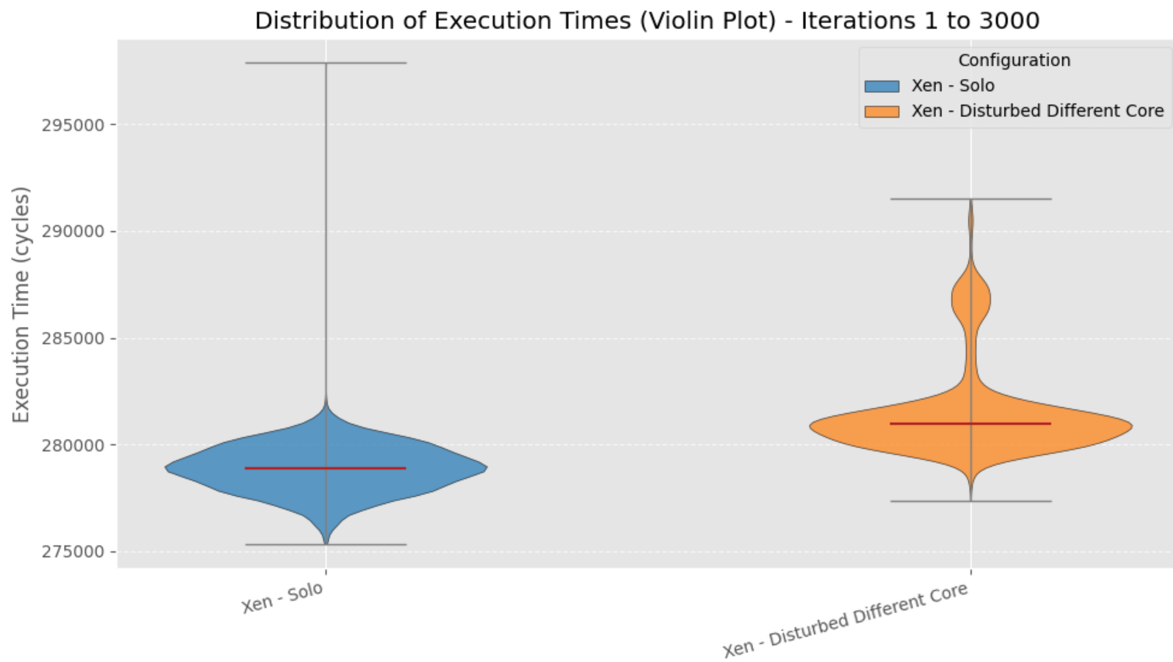

Figure 4.6: Qsort: An enhanced look at Xen's performance running qsort with a disturbed guest on a different VM and a guest running solo.

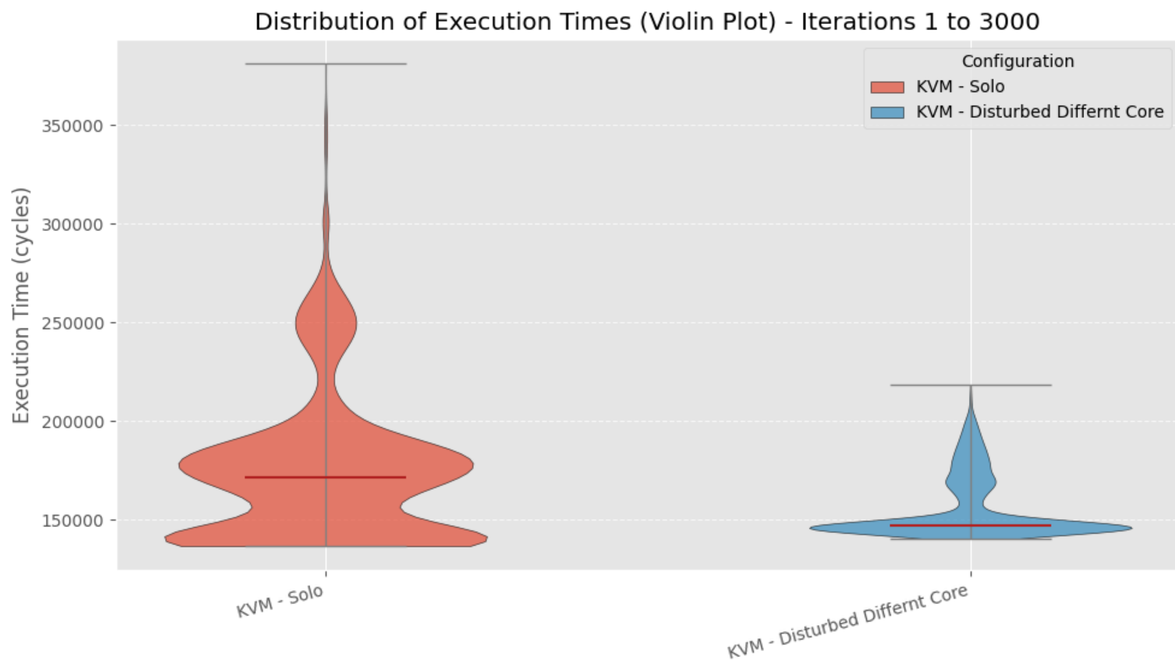

Figure 4.7: Qsort: An enhanced look at QEMU/KVM's performance running qsort with a disturbed guest on a different VM and a guest running solo.

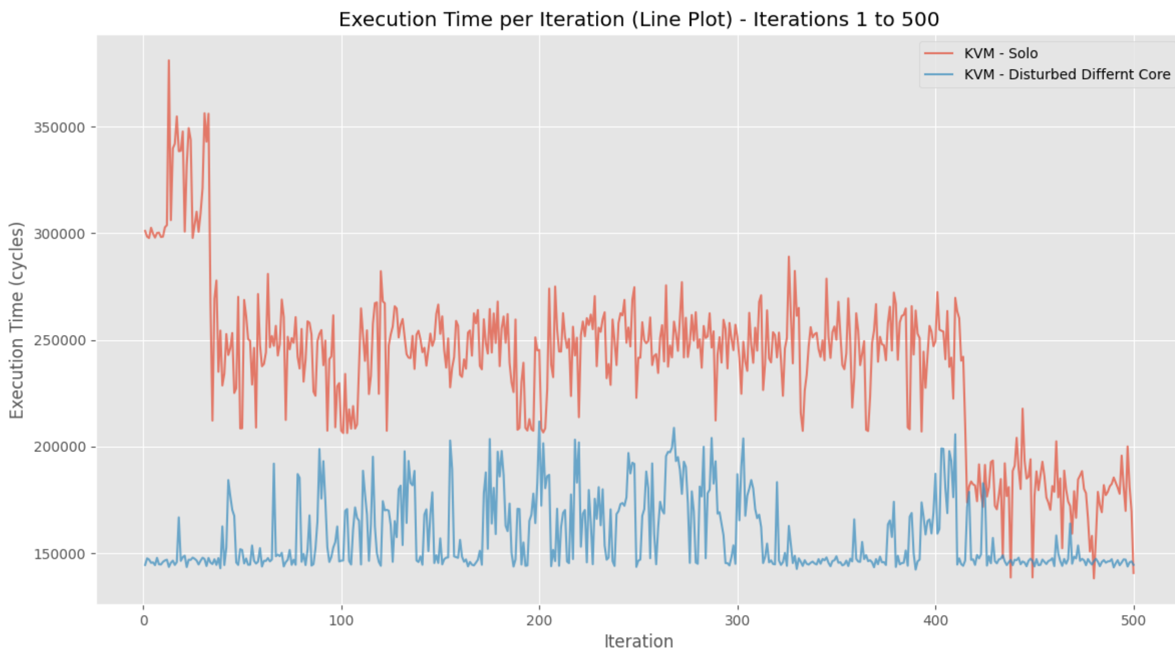

Figure 4.8: Qsort: Line plot of the first 500 iterations of QEMU/KVM with a disturbed guest and a solo guest, showing the performance ramp-up for the solo guest.

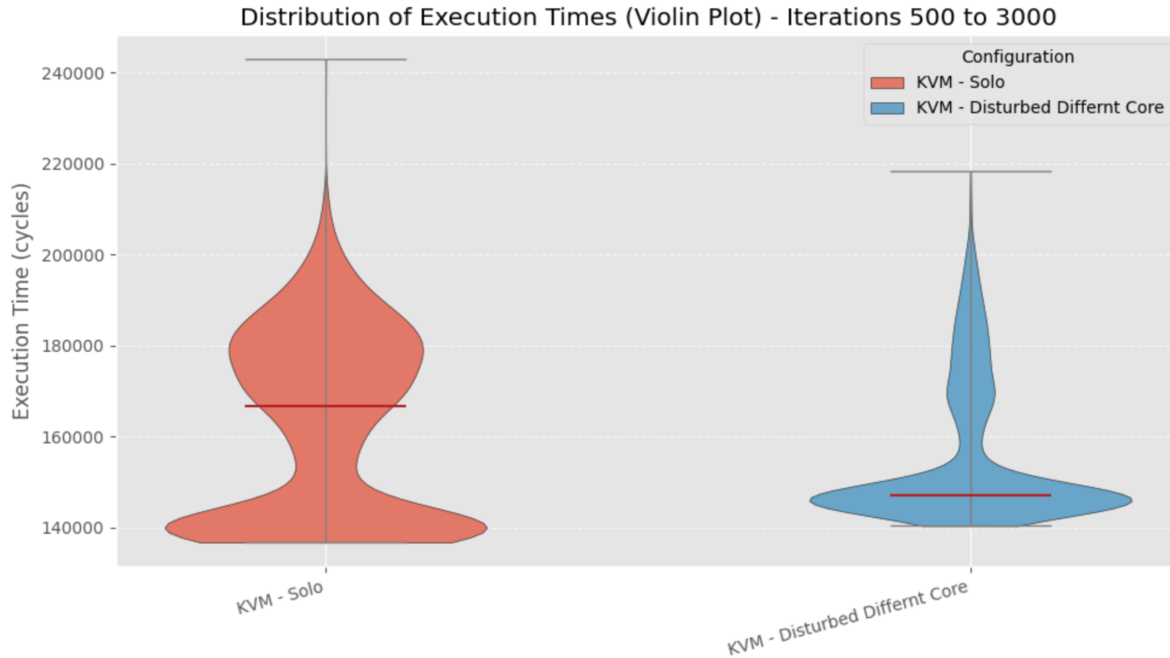

Figure 4.9: Qsort: Distribution of qsort execution time (in CPU cycles) between iterations 500 and 3000, comparing a disturbed guest and a solo guest after the ramp-up period of the QEMU/KVM solo configuration.

4.2.2 Basicmath

Figures 4.10 through 4.14 illustrate the results of the basicmath test, which performs mathematical operations often lacking direct hardware support [20]. As with Qsort in section 4.2.1, a notable drop in performance is seen when the measuring and disturbing guests are pinned to the same core, in both Xen and QEMU/KVM, as seen in Figure 4.10 where the result of all configurations are shown. When the guests are pinned to different cores on Xen, the mean execution time is nearly identical to the solo configuration, although more distant outliers appear in the disturbed configuration. This is shown in Figure 4.11.

For QEMU/KVM, the disturbed guest once again appears to outperform the solo configuration in Figure 4.12. This is likely due to a ramp-up period that affects the solo run, which can be observed in Figure 4.13. After excluding the ramp-up period in Figure 4.14, performance differences between the two configurations narrow considerably, though the disturbed guest still exhibits slightly better performance in both mean and variance.

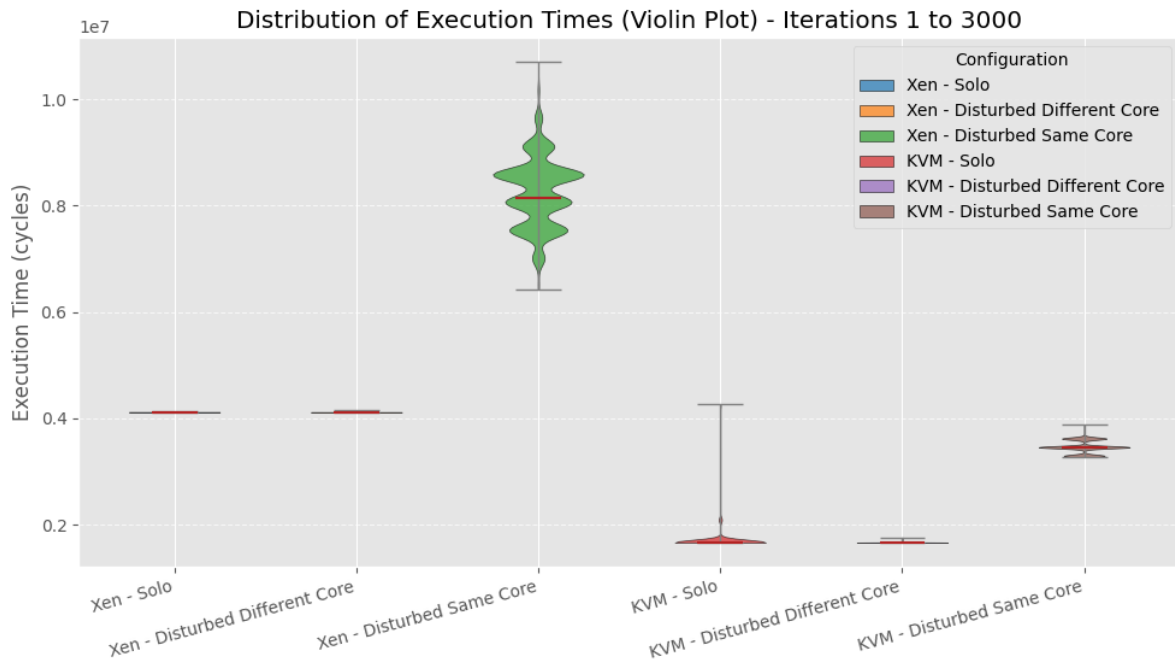

Figure 4.10: Basicmath: Distribution of basicmath execution time (in CPU cycles) across six system configurations using Xen and QEMU/KVM hypervisors, measured under various stress conditions.

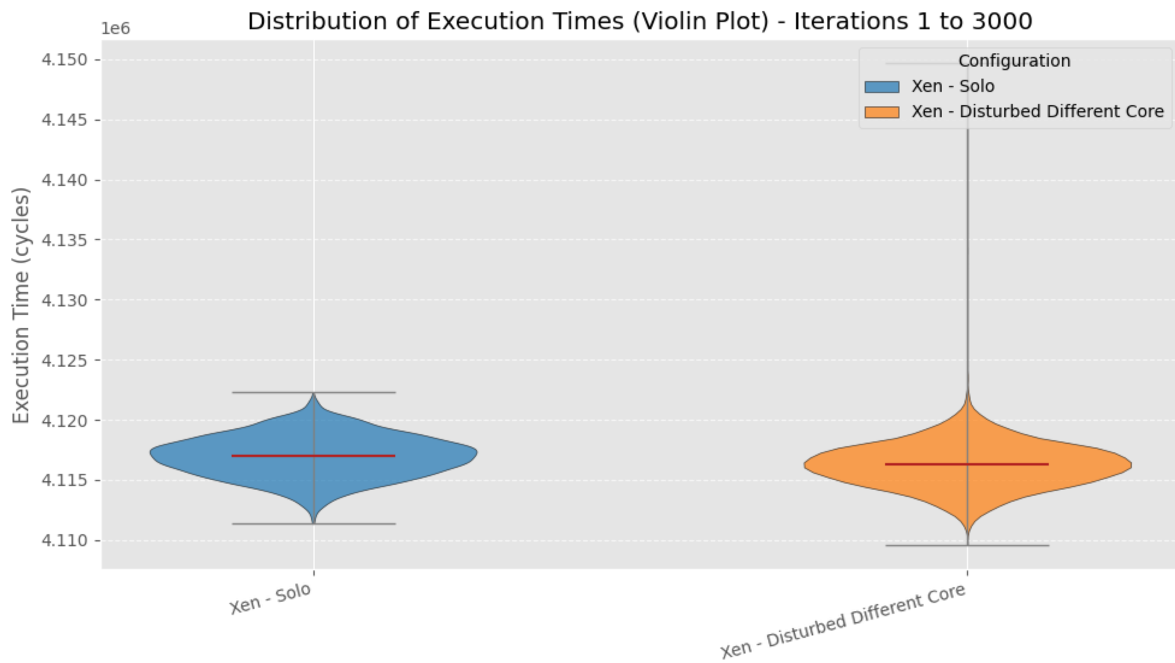

Figure 4.11: Basicmath: An enhanced look at Xen's performance running Basicmath with a disturbed guest on a different VM and a guest running solo.

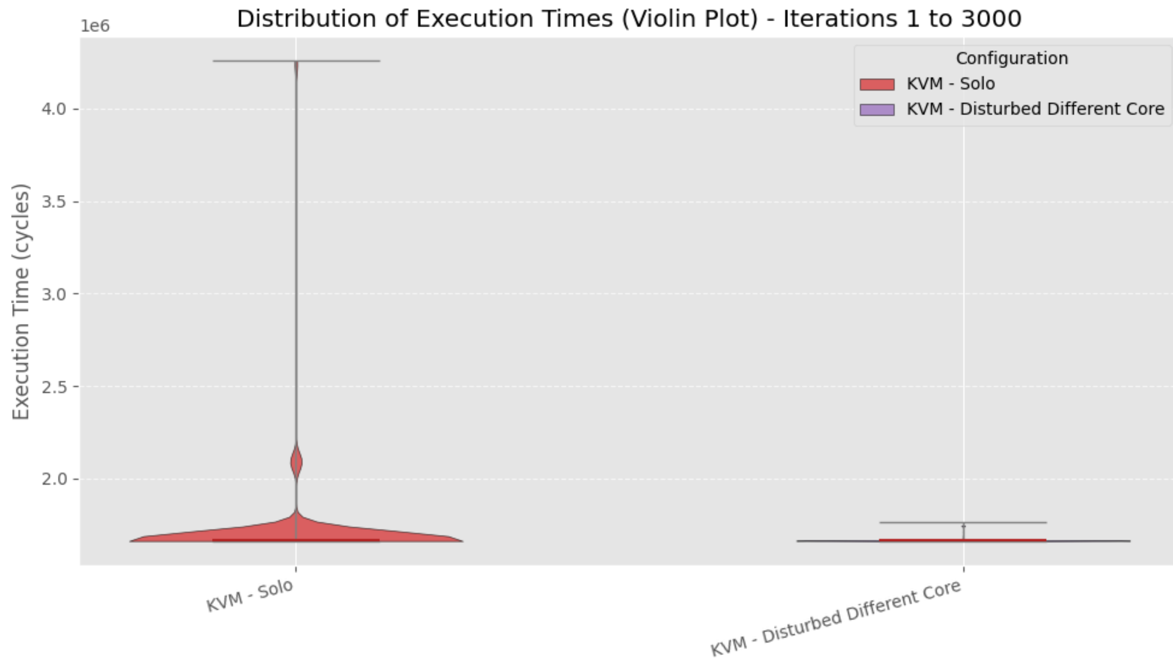

Figure 4.12: Basicmath: An enhanced look at QEMU/KVM's performance running basicmath with a disturbed guest on a different VM and a guest running solo.

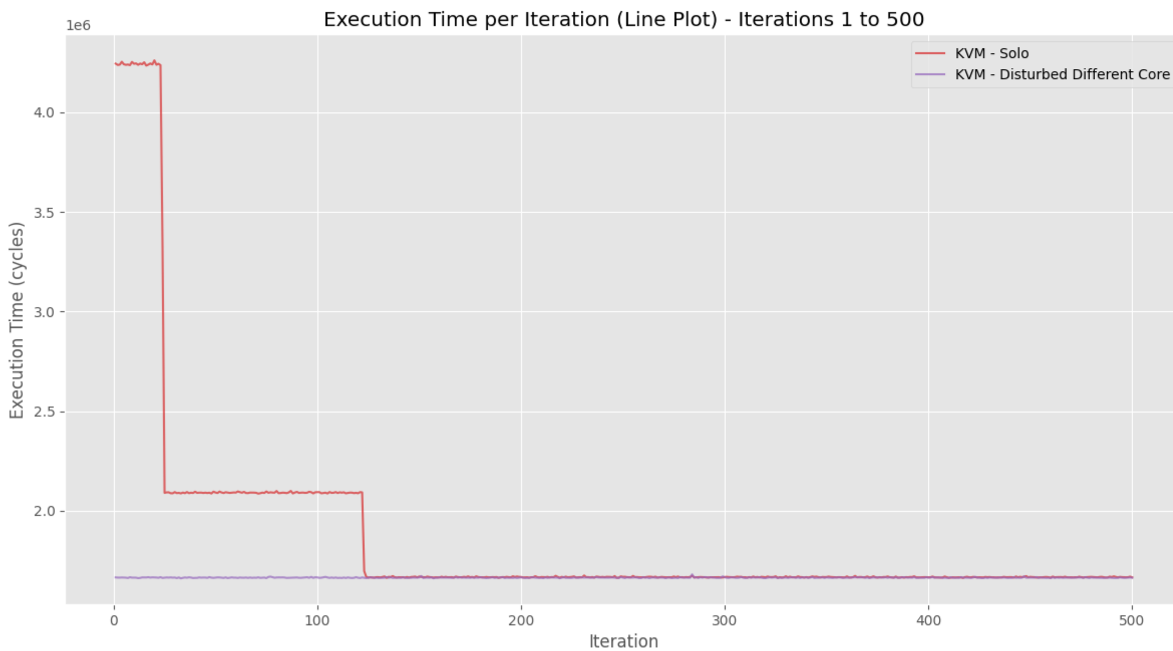

Figure 4.13: Basicmath: Line plot of the first 500 iterations of QEMU/KVM with a disturbed guest and a solo guest, showing the performance ramp-up for the solo guest.

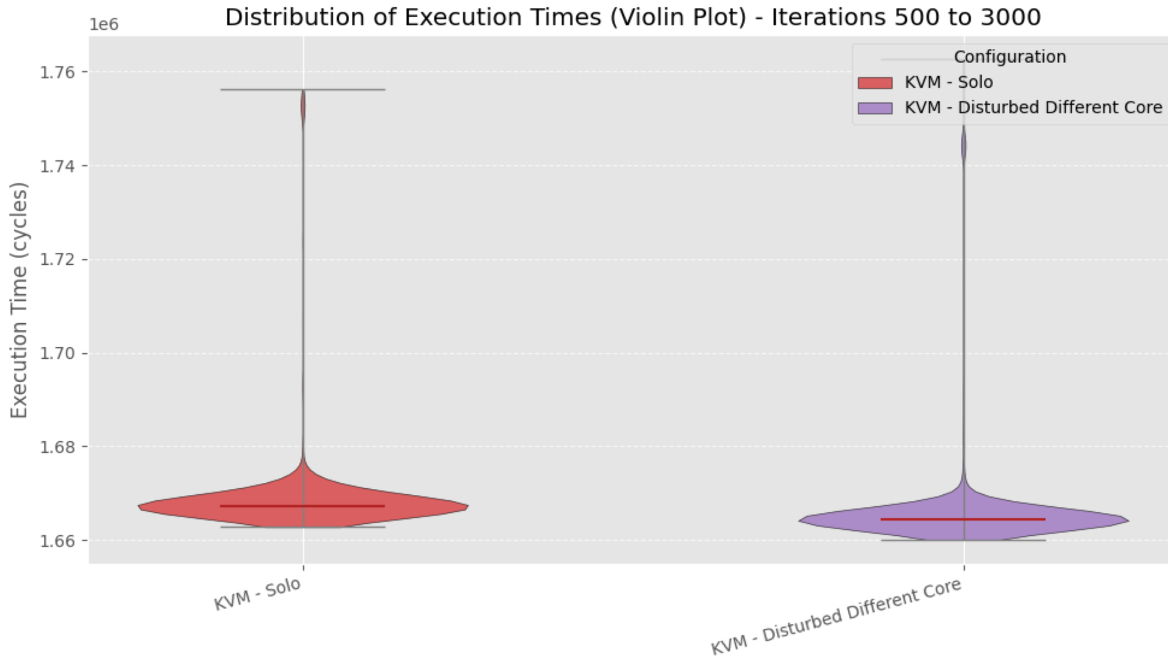

Figure 4.14: Basicmath: Distribution of basicmath execution time (in CPU cycles) between iterations 500 and 3000, comparing a disturbed guest and a solo guest after the ramp-up period of the QEMU/KVM solo configuration.

4.3 Thread metrics

In this section, the result of the scheduling test called thread metrics is presented. All test runs have in common that they all succeed, and there are no reports of threads being scheduled more times than others. There are three different cases that are investigated. The first two are the impact of starting the stress before or after the thread metrics test and the CPU affinity of the VMs, i.e., having the VMs run on their own core or sharing a core. These are presented in Section 4.3.1.

The third area that is presented is looking at the impact of the stress on the performance over a longer time period. This is investigated to see if there is a difference in performance during and after running the stress VM. The results of this is presented in section 4.3.2.

4.3.1 Stress Start Order and CPU Affinity

Figure 4.15 shows the result of experimenting with stress order and CPU affinity on the QEMU/KVM hypervisor. The first part to notice is that the performance drops to about half when executing the stress and the test on the same CPU, compared to each VM having its own dedicated CPU. This can also be seen on the Xen hypervisor in Figure 4.16.

Another easily noticeable characteristics is the start-up period required for QEMU/KVM where the performance is on a significantly lower level during the first 5 seconds. This is eliminated when the stress is started before the test. In all cases where the test VM is started first on QEMU/KVM the first measurements are lower, while the performance is more consistent when the stress is started before the test.

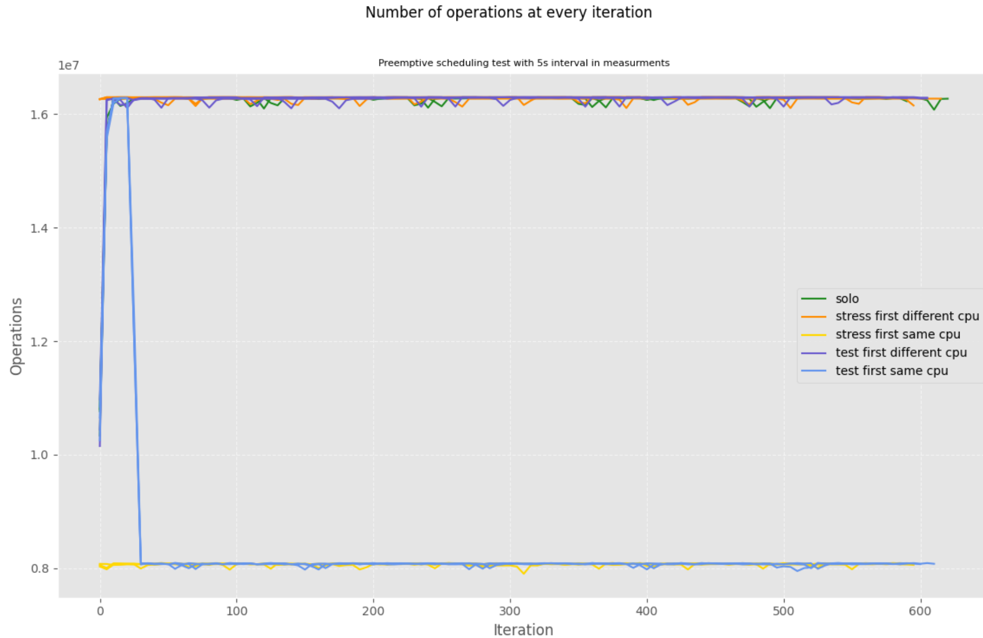

Figure 4.15: On QEMU/KVM, scheduling over 10 minutes with different start orders of the stress and the test. Different CPU affinity was also tested with the test running on either the same or different CPUs.

The ramp-up period of QEMU/KVM does not noticeably interfere with the results of CPU affinity. When the stress is started after 30 seconds, the performance drops when running on the same CPU, as seen in Figure 4.15, while the performance stays about the same when running on different CPUs. The ramp-up period seems to be connected to the start order only.

For Xen hypervisor the results are similar regarding CPU affinity, the performance in this case also drop to about half. A noticeable difference is that the number of times the test is scheduled is much lower. For QEMU/KVM the higher level is around 16 300 000 but for Xen this number is around 6 590 000, which is less than half.

On the other hand, the ramp-up period that was seen in QEMU/KVM does not exist on Xen. There is a slight increase in the performance during the first 5 seconds that can be seen when looking closely at the first iterations in Figure 4.17, but this is only in the order of around 7,000 context switches at the most.

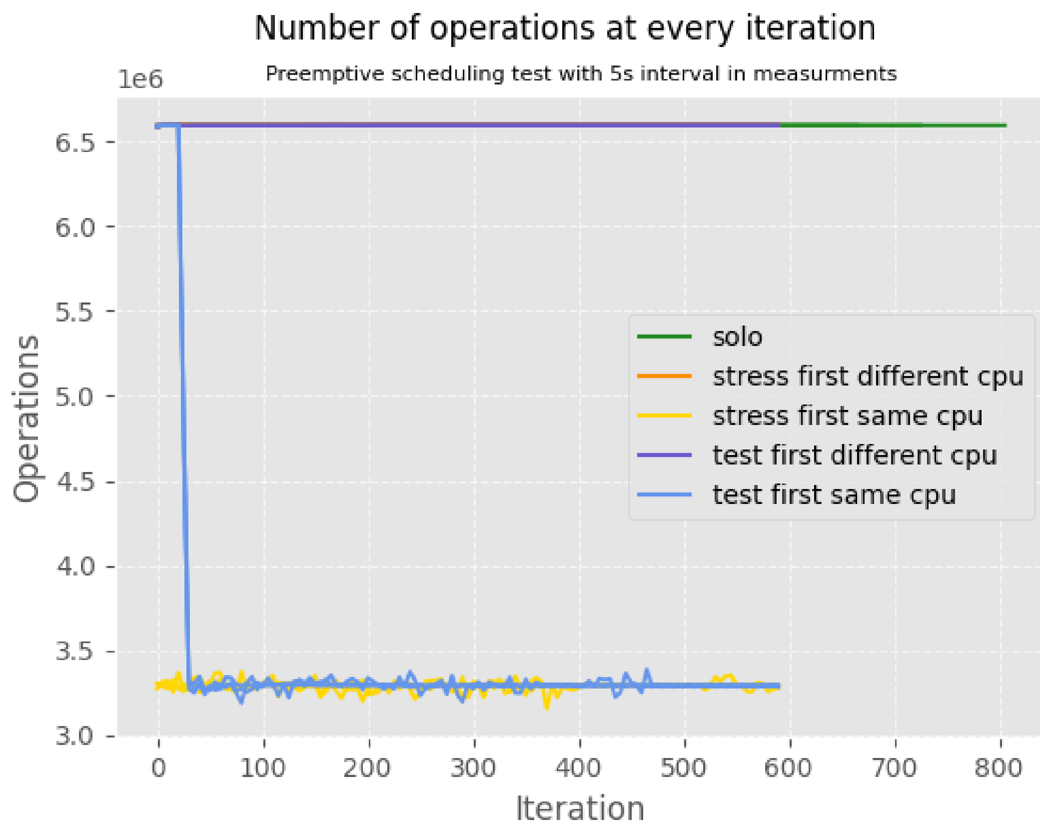

Figure 4.16: On Xen, scheduling over 10 minutes with different start orders of the stress and the test. Different CPU affinity was also tested, with the test running on either the same or different CPUs.

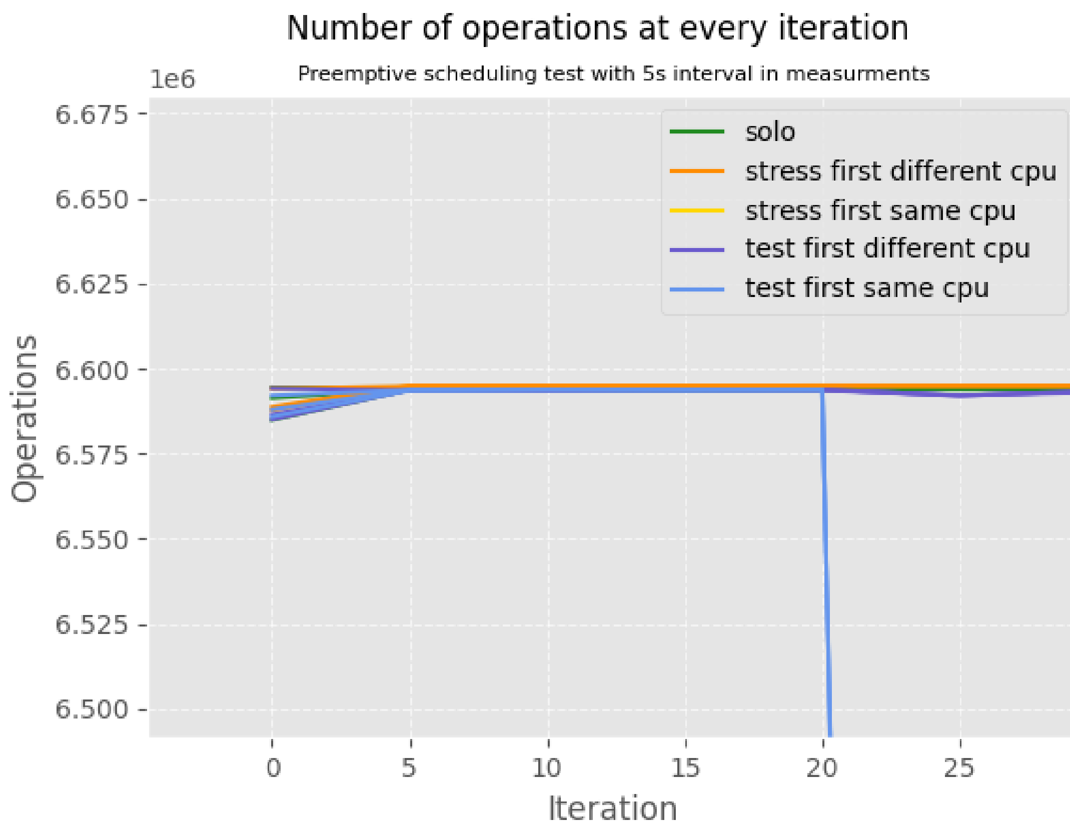

Figure 4.17: Magnified version of Figure 4.16 to see the difference in performance at the start of the test on Xen hypervisor.

4.3.2 30-minute test

In Figure 4.18, the output of the 30-minute test on the Xen hypervisor is shown. The dashed lines are at the intervals where the stress VM is turned on and off. Between each run, the system is rebooted to ensure similar conditions are achieved.

Figure 4.18 shows that there are two stable levels where the tests converge, one higher when running solo and one lower when running the stress VM in parallel. There is also a lower, less consistent level where most tests are running for some amount of time. This level has more variations and is somewhat the same whether having a stress VM or not. The first iteration is always lower when the system starts up, and all runs have a dip in performance when the stress is turned on at about 600s. This can be seen in more detail in Figure 4.19, where this portion of the graph has been enhanced.

To get another view of the data, violin charts are also created in Figure 4.20 to showcase the distribution of the number of times the threads are scheduled in the 5s interval. In these, the first iteration, that is the startup, is removed as well as the transitional periods when the stress is turned on and off; this was done to get a better view of the system in a stable state. These charts show that the test is scheduled less during stress and also that the scheduling is more consistent after compared to before the stress is running. This test is only performed on different CPUs.

Regarding the 30-minute test on the QEMU/KVM hypervisor, the results are shown in Figure 4.21 and in enhanced versions in Figure 4.22 and Figure 4.23. These indicate that the performance is slightly enhanced when the stress is added. It can also be seen that there is one major outlier that does not represent the average run. There seems to be more variation in the results during stress compared to before, even though the scheduling times in general are higher; this is seen most clearly in Figure 4.23.

These results can also be seen in the violin charts in Figure 4.24, where the outlier is apparent. The graph was magnified to exclude the outlier and see the other data more clearly in Figure 4.25. Here it can be seen in more detail that the performance is slightly enhanced, but the variations are larger when the stress is running. The outlier when running before stress can be explained by the start-up period.

When comparing Xen and QEMU/KVM, two main observations can be made. The first is that QEMU/KVM has time to execute the scheduling more times than Xen, about 2.5 times more. The other is that the variations seem less in the general case with QEMU/KVM, but this is also where an outlier has been observed. The implications of this will be discussed in section 5.3.

In the violin chart in Figure 4.24 of the QEMU/KVM run, this outlier can be seen clearly. It can also be seen that, as with Xen, the results become slightly more stable when the stress has stopped compared to before it ran.

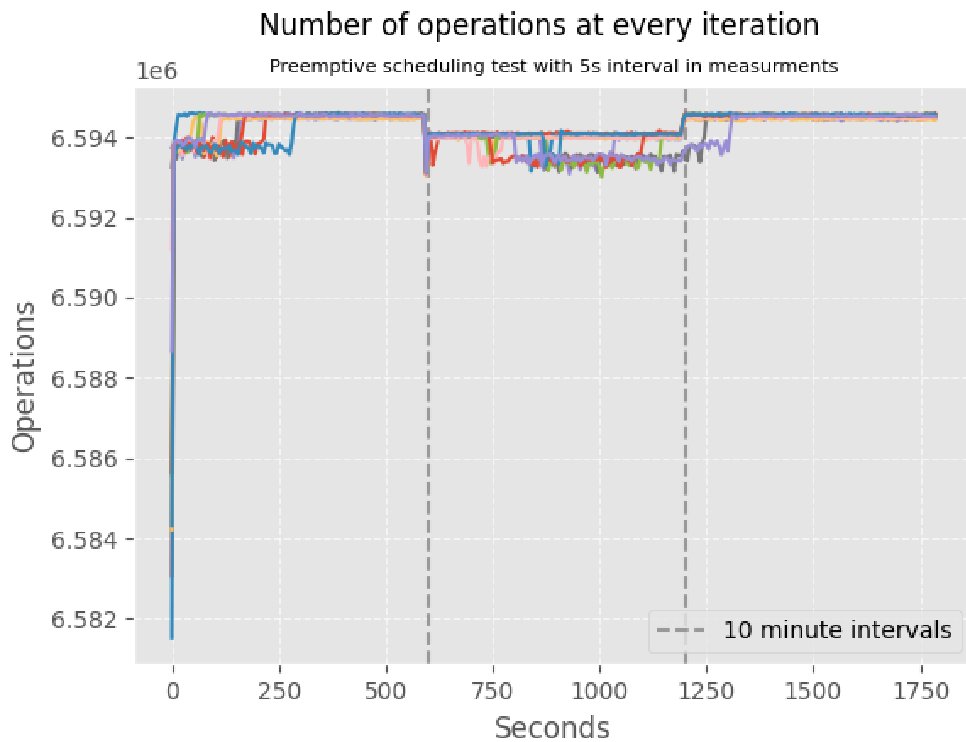

Figure 4.18: Run over 30 minutes with thread metrics test on Xen where stress VM is started at 10 minutes and stopped at 20 minutes

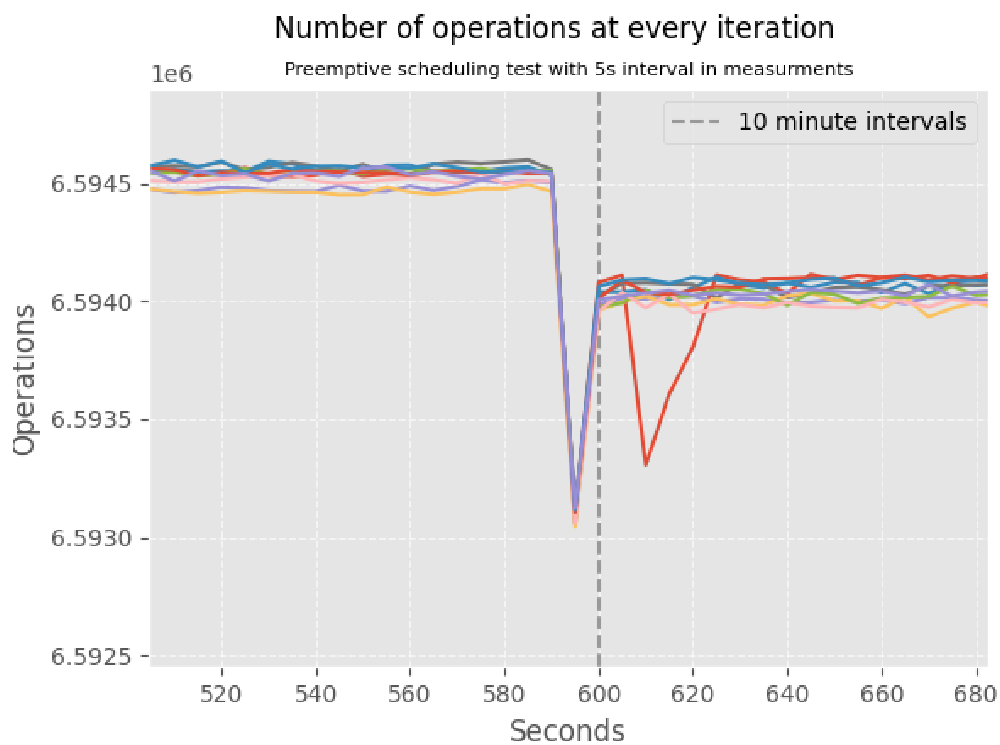

Figure 4.19: Enhanced variant of Figure 4.18 to see the dip when the stress VM is turned on.

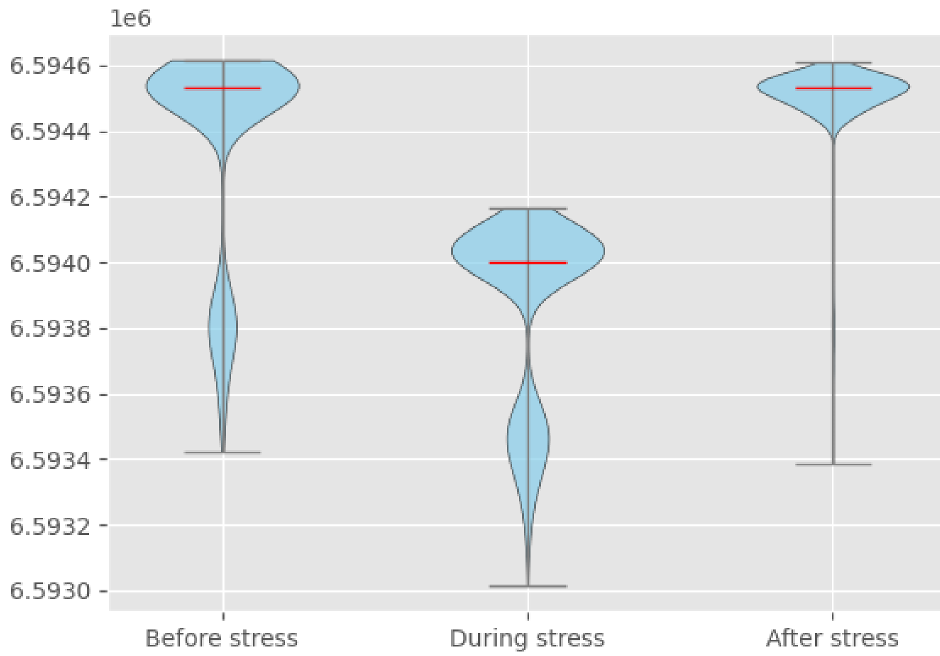

Figure 4.20: Distribution of the thread metric count on the Xen hypervisor, a higher number of operations per time unit is favorable.

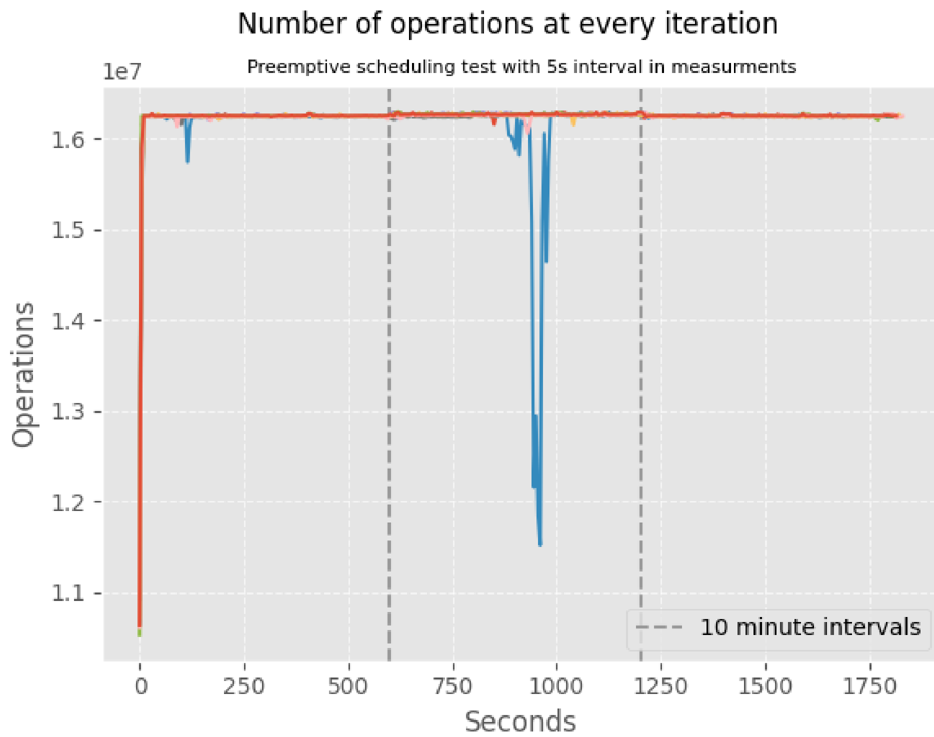

Figure 4.21: Run over 30 minutes with thread metrics test on QEMU/KVM where stress VM is started at 10 minutes and stopped at 20 minutes

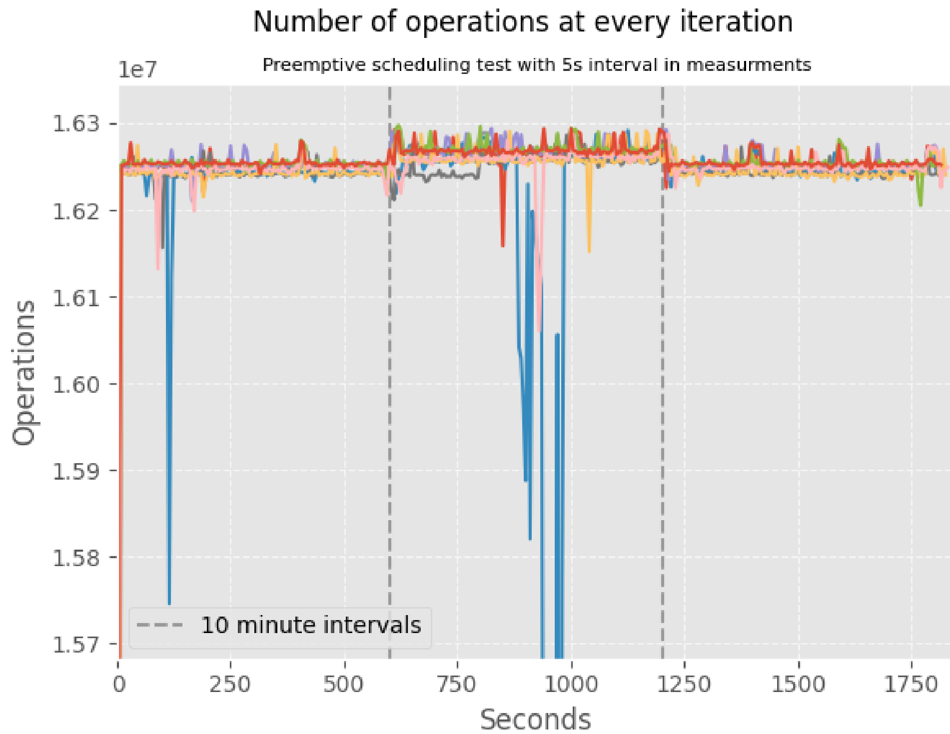

Figure 4.22: Focused version of Figure 4.21 to see the change of the stress better.

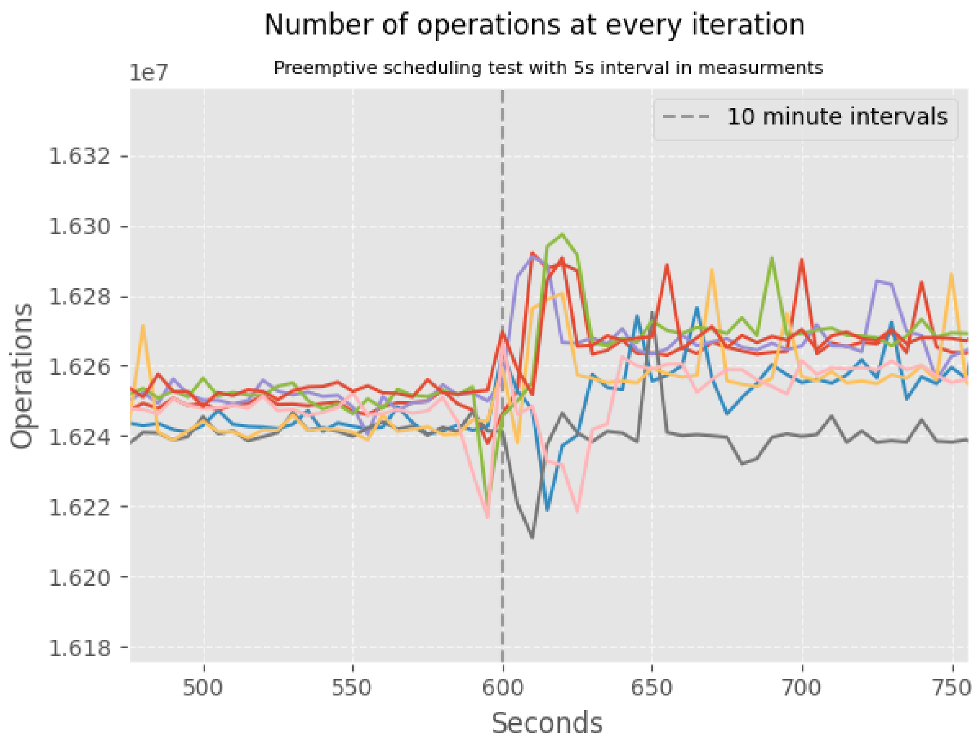

Figure 4.23: Focused version of Figure 4.22 to see the values at the time the stress is introduced in more details.

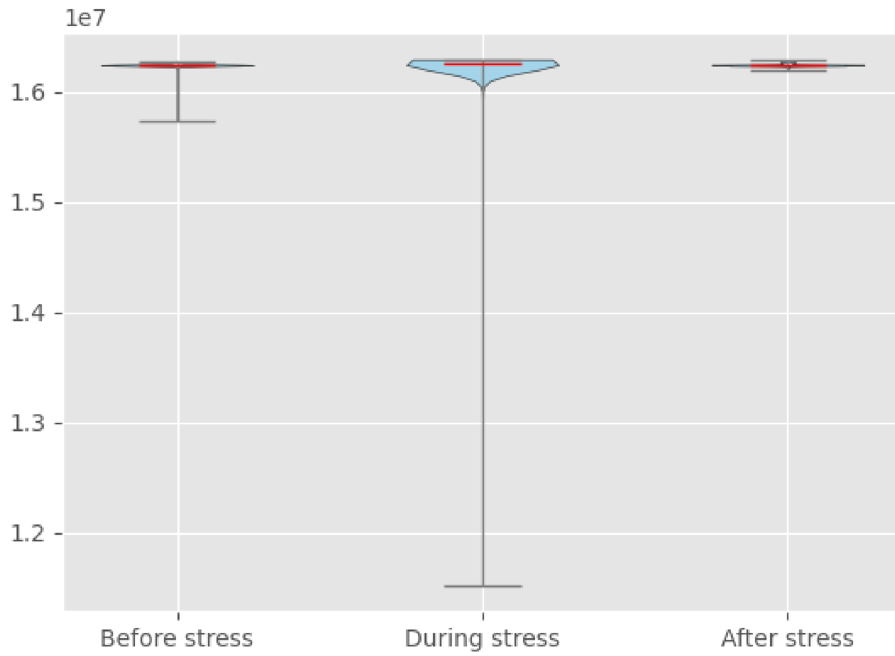

Figure 4.24: Distribution of the thread metric count on the QEMU/KVM hypervisor, a higher number of operations per time unit is favorable.

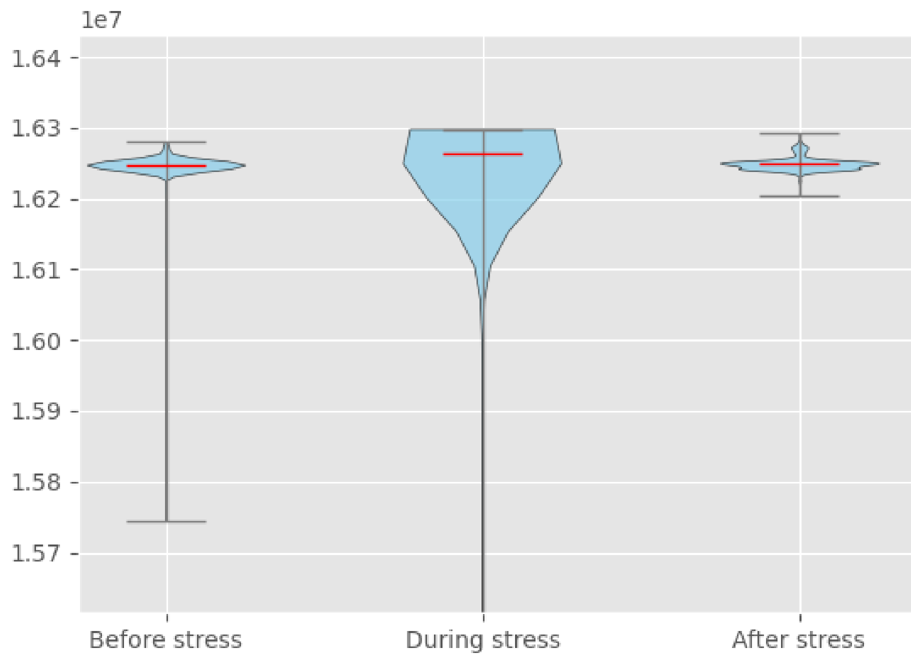

Figure 4.25: Distribution of the thread metric count on the QEMU/KVM hypervisor, outlier excluded to be able to enlarge the graph

,

5 Discussion

In this chapter, the results from the previous chapter are discussed and evaluated. Firstly, the test areas are discussed independently in sections 5.1 through 5.3. Later, the test areas are put in relation to each other when evaluating the areas themselves in section 5.5, which is relevant to the research questions that are discussed in section 5.6. Possible sources of error and future work are also analyzed in sections 5.4 and 5.8 to be able to see the contributions of the work of this thesis.

As a general observation for the all results a decline in performance is shown when VMs are executed on the same CPU core. This is expected behavior since the VMs have to share the processing power and get scheduled for an equal amount of time. Reduced performance has to be taken into account if multiple systems are running on the same CPU. Thus, having systems on the same CPU is more suitable for tasks that are not critical. Critical tasks are safer to run on their own CPUs. There are cases where lighter, non-critical tasks would be able to share a CPU for increased system efficiency. This has to be designed on a case by case level.

5.1 Latency Measurements

The first topic to highlight from the latency measurements is that, in general, the results indicate that a stressed VM on a different core does not significantly impact the operations. In most cases where variation can be seen the delays are a few cycles, which accounts to nanoseconds and is tolerable to most systems. The results also indicate that in most cases the hypervisor is able to handle the low level operations without any noticeable impact of the stress VM.

More worrisome from a safety perspective are the outliers that were presented in the result of the latency measurements in section 4.1 and can be seen here in Table 5.1. The outliers add over 500,000 cycles to three of the tasks where threads are waiting for each other, which accounts for about 10 ms. One task was delayed with over 50,000 cycles, which is also an outlier, much larger than all other benchmarks, and results in a delay of about 1 ms. For the context switch benchmark, the delay happens multiple times, which is even more concerning.

According to our industry partners, 10 ms is an acceptable deadline for tasks and delays in industry real-time systems. Thus, delays longer than 10 ms in particular could be enough to cause tasks to miss deadlines, which would impair the safety of the system if this were to happen to critical tasks. Why this happens is more difficult to state from our experiences and would need further research. Since the outliers happen rarely, it is

Table 5.1: Outliers of Latency Measurements and the respective median values of the benchmark

Benchmark	**Setup**	**Median Value** (cycles)	**Outlier** (cycles)	**Outlier** (ns)
Wait for any event (w/ ctx switch)	Xen w/ stress, different core	52	541 429	10 026 462
Wait for all events (no ctx switch)	Xen w/ stress, same core	22	55 927	1 035 685
Set events (w/ ctx switch)	Xen w/ stress, same core	82	542 083	10 038 574
Ctx switch via k_yield	Xen w/ stress, different core	43	526 558	9 751 074

Table 5.2: Comparative Performance Statistics of Virtualization Setups vs. Baselines when running qsort, WCET

Configuration	**Baseline**	**Mean Incr.** (%)	**Median Incr.** (%)	**WCET (Config)** (cycles)	**WCET (Baseline)** (cycles)
Xen - Disturbed Different Core Xen - Disturbed Same Core	Xen - Solo	0.95 101.48	0.74 20.25	291 504 1 366 470	297 885
QEMU/KVM - Disturbed Different Core QEMU/KVM - Disturbed Same Core	QEMU/KVM - Solo	-10.98 73.69	-14.28 76.35	218 381 472476	381 080
QEMU/KVM - Disturbed Different Core QEMU/KVM - Disturbed Same Core	QEMU/KVM - Solo - After Rampup	-4.21 86.75	-11.74 81.66	218 381 471 376	242 946

difficult to find the root cause, whether this is a fault at the hardware, hypervisor, or test level. The circumstances indicate that it could be a hypervisor problem since this only happens on the Xen hypervisor when there is a disturbance, but the testing is not extensive enough to draw this conclusion with confidence.

Additionally, the results of the test are somewhat varying on the smaller scale. This is especially seen when running both VMs on the same CPU core, which is in line with the results of the other tests. The delays for this case is still very small and would most likely not have an impact on a system if not happening repeatedly.

5.2 MiBench

Both qsort and basicmath from the MiBench suite exhibit similar performance trends across Xen and QEMU/KVM. While the primary focus of these experiments is temporal isolation, it is worth noting that QEMU/KVM consistently outperforms Xen in all configurations when it comes to pure execution time. However, this comes at the cost of greater execution time variability compared to Xen.

Performance degradation is particularly evident when a co-located guest shares the same CPU core, as shown in Figures 4.5 and 4.10. This outcome is expected, as the hypervisor must fairly schedule both guests. When a single iteration of Qsort exceeds its allocated time slice, its execution is negatively impacted by the competing guest.

In contrast, the impact of interference from a guest on a different CPU core is less predictable. Interestingly, QEMU/KVM occasionally shows improved performance under such cross-affinity stress conditions, likely due to differences in how QEMU/KVM manages CPU caches and resource contention.

A more detailed analysis of these behaviors is presented in subsections 5.2.1 and 5.2.2 for qsort and basicmath, respectively.

5.2.1 Qsort

Both hypervisors exhibit notable performance degradation when a disturbing guest runs on the same core as the measuring guest, as illustrated in Figure 4.5. This degradation is expected due to scheduling contention. Qsort processes a 1MB data array per iteration, and as detailed in Section 3.2, the L1 data cache per core is 32KB, while the shared L2 cache is 1MB.

In this context, cache contention – particularly in the L1 cache – likely contributes significantly to the observed performance losses. When two guests share a core, the disturbing guest frequently accesses large data blocks, causing excessive L1 cache evictions and resulting in frequent cache misses for the measuring guest.

Figure 4.6 shows that even when the disturbing guest is pinned to a different core under Xen, a slight performance degradation still occurs. Interestingly, the largest execution time outlier appears in the solo guest scenario. However, as illustrated in Figure 5.1, the solo guest generally exhibits more consistent performance, with fewer and less frequent outliers. For the disturbed guest, the outliers remain within 3.41% of the mean execution time, indicating relatively limited impact.

In this cross-core setup, the working set still exceeds the L1 cache capacity, necessitating frequent L2 cache access. The disturbing guest, which operates on data that surpasses both L1 and L2 capacities, may evict data from the shared L2 cache, leading to additional cache misses for the measuring guest. However, the performance penalty is notably smaller than in the same-core case, which aligns with the expectation that L2 cache contention is less severe than direct L1 interference. Table 5.2 confirms this, showing that the increase in mean and median execution times under Xen with a disturbing guest on a separate core remains within 1%.

Under QEMU/KVM, however, the behavior diverges. As shown in Table 5.2, the guest running qsort performs worse when executing alone than when a disturbing guest is present on another core. This counter-intuitive result may partly stem from a ramp-up period observed at the start of execution, as shown in Figure 4.8. During this phase, performance gradually improves as the system stabilizes.

After the ramp-up, the Worst Case Execution Time (WCET) improves significantly, from 381,080 cycles during ramp-up to 242,946 cycles. Notably, when excluding the ramp-up period, the disturbed guest still performs better than the solo guest in terms of mean, median, and worst-case execution times.

The underlying cause of this behavior remains unclear. It may be attributed not to KVM itself but to the way QEMU interacts with KVM, potentially due to nuances in how guest initialization or CPU resource management is handled.

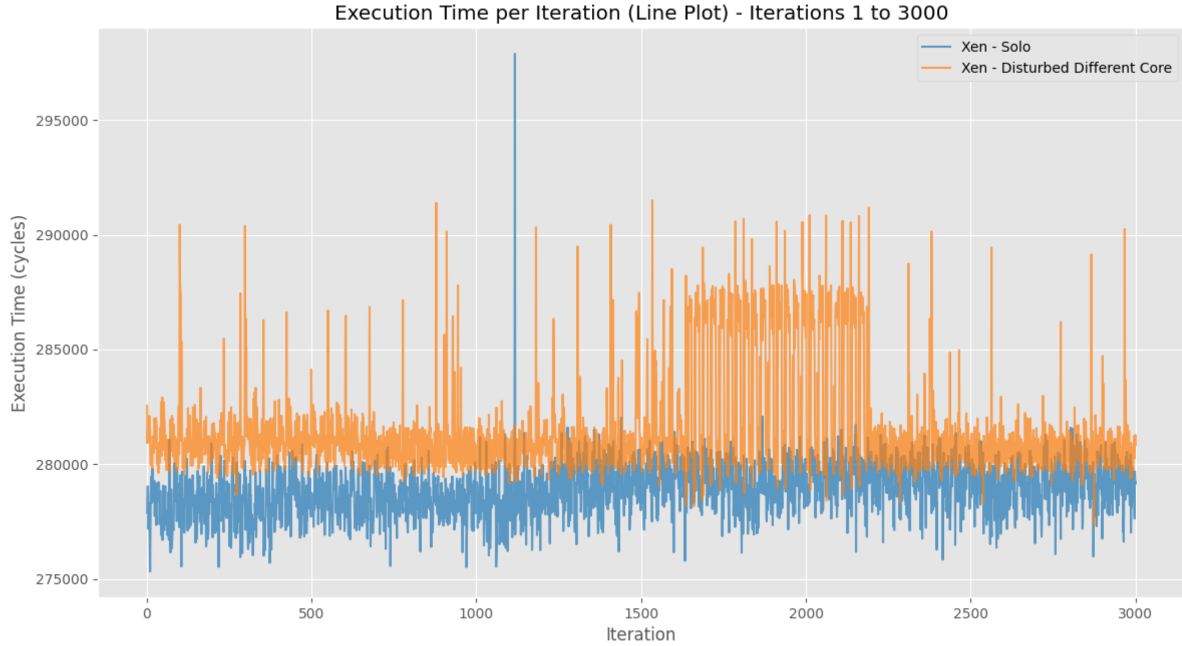

Figure 5.1: Lineplot of qsort performance in cycles on Xen with a disturbing guest on a different core

5.2.2 Basicmath

A similar performance pattern is observed for the basicmath test when the measuring and disturbing guests run on the same core. For both Xen and QEMU/KVM, the number of CPU cycles increases by approximately an order of magnitude of 2 across the mean, median, and worst-case execution times, as shown in Table 5.3. Unlike Qsort, basicmath does not rely on large datasets, suggesting that the performance degradation is primarily due to scheduling interference rather than cache contention.

Figure 4.11 and Table 5.3 further illustrate that, under Xen, the performance of the measuring guest with a disturbing guest on a different core is nearly identical to its solo execution. The WCET differs by only 0.66%, indicating strong temporal isolation in this scenario.

As discussed in Section 5.2.1, QEMU/KVM exhibits an interesting behavior where the disturbed guest consistently performs better than the solo run. This phenomenon is also observed with basicmath, albeit to a lesser degree. The disturbed guest achieves approximately 2.18% better average performance, and its WCET after ramp-up is lower than when running without interference.

Overall, both hypervisors demonstrate stronger temporal isolation in the basicmath benchmark compared to Qsort. This may be attributed to basicmath's reduced dependence on shared hardware resources, such as caches. Additionally, since the basicmath test runs longer, fixed-time disturbances may have a proportionally smaller impact, reducing their effect on percentage-based performance metrics.

Table 5.3: Comparative Performance Statistics of Virtualization Setups vs. Baselines for basicmath, WCET

Configuration	Baseline	Mean Incr. (%)	Median Incr. (%)	WCET (Config) (cycles)	WCET (Baseline) (cycles)
Xen - Disturbed Different Core	Xen - Solo	-0.02	-0.02	4 149 680	4 122 264
Xen - Disturbed Same Core		100.52	97.75	10 711 064	
QEMU/KVM - Disturbed Different Core	QEMU/KVM - Solo	-2.18	-0.17	1 762 702	4 261 137
QEMU/KVM - Disturbed Same Core		103.37	107.37	3 883 260	
QEMU/KVM - Disturbed Different Core	QEMU/KVM - Solo - After Rampup	-0.18	-0.17	1 762 702	1 756 200
QEMU/KVM - Disturbed Same Core		107.47	107.37	3 883 260	

Table 5.4: Comparative Performance Statistics of Stress Order

Hypervisor	First Iteration	Median Value	Difference / Median
Xen - solo	6590252	6594518	0.06%
Xen - stress first	6590244	6595009	0.07%
Xen - test first	6588574	6594043	0.08%
QEMU/KVM - solo	10636442	16281263	34.67%
QEMU/KVM - stress first	16262067	16296038	0.21%
QEMU/KVM - test first	10522553	16295769	35.43%

5.3 Thread Metrics

The results are somewhat inconclusive when the thread metrics VM is running without interference on the hypervisor. This implies that analysis and conclusions of the results have to be made carefully. Overall, changes in performance can be observed during stress, even though the differences are minor. The changes in performance could indicate a break in temporal isolation; more details will be described for each test in the following sections 5.3.1 and 5.3.2.

5.3.1 Start Order of the Stress VM

Looking at the results of the stress start order, one important aspect is the ramp-up period when starting the test as the first VM on QEMU/KVM. The statistics are presented in Table 5.4. This does not happen when starting the stress VM before the test VM, which was also observed in the MiBench tests in section 4.2. The ramp-up period was much more significant on QEMU/KVM than on Xen, and the implications of this could be that QEMU/KVM requires more resources for the hypervisor to handle the start of VMs. This, in turn, would lead to the thread metrics test being scheduled less. In the Xen case, where the Dom0 has its own dedicated CPUs, this still happens, but to a much smaller extent which could not be seen in the MiBench tests, only on the thread metrics tests. On Xen, there is no particular difference in the start order of the VMs. Xen seems to handle the start-up of the first VM more efficiently than QEMU/KVM.

5.3.2 30 minute test

When looking at the results of the 30-minute test on Xen in Figure 4.18, where the stress is started and stopped with 10-minute intervals, several observations are made. The first

Table 5.5: Comparative Performance Statistics of 30-minute test

Hypervisor	Before Stress median value	During Stress median value	After Stress median value	Difference (%)	Outliers (%)
Xen	6594535	6594001	6594531	-0.01	-0.022 (Dip during transition)
QEMU/KVM	16248085	16263584	16248360	0.1	-29 (Worst Case Outlier)

is that there are two distinct top levels of each section, where the number of operations is relatively consistent. When the stress is applied, the maximum number of operations is distinctly lower than when the test is running alone on the system. The change is about a 0.01% decrease in the number of operations, which could be an insignificant amount in a practical situation. The maximum values when running during stress are around 6,594,001 compared to when running alone, when the values are around 6,594,580.

Regarding QEMU/KVM the 30 minute test shows a slight increase in performance when the stress is turned on in Figure 4.22. This slight increase and more stable performance was also seen in the MiBench-tests in section 5.2.1. The difference is of about 15,000 iterations, which is an increase of 0.1% compared to before the stress VM. This is a larger difference than Xen by a factor of 10, but the values are also less consistent which makes them less reliable. A summary of median values and the differences during stress are summarized in Table 5.5.

It can also be observed that all runs on Xen in Figure 4.18 have a dip in performance when the stress VM is starting. This dip is consistently down to 6,593,100, which accounts for a decrease of 0.022% compared to the stable value when running alone. This dip indicates that the adjustment for the hypervisor when running one VM compared to both requires some extra work and this is what could be reflected by this dip. On a larger scale, the change is still very small, and it is unknown whether it would be able to affect the safety of a system. Transitional periods, as when the stress VM is started or stopped, are of particular interest since the system could be less predictable when changes are made to it. On QEMU/KVM, the transitional phase does not have any noticeable dips or peaks in performance. After the transition to having a stress VM running in parallel, the scheduling seems slightly less consistent than before, but the results are not consistent enough to draw any conclusions but could inspire further research.

When looking at the performance in the context of isolation it can be seen that there is an observable impact when the stress VM is started. This indicates that the temporal isolation between the VMs is not perfect since the test is scheduled differently during stress even though both VMs are having their own dedicated CPU.

On Xen, in addition to the stable levels, there are also a lot of less consistent values during the runs; the variations are seen in figures 4.20. This adds to the insecurity of the result since most of the execution during stress is inside the range of running before the stress VM. All of the runs spend a significant amount of time having this lower, more varied performance. The cause of this varied performance is hard to determine, but it can be seen in Figure 4.25 that some form of performance variation is true both before and during the stress, but not after. Also, as can be seen in Figure 4.20, there is a wider range of values during stress for QEMU/KVM, but still, the variation is small in the common case.

Figure 4.24 shows the distribution of the thread metrics for QEMU/KVM, what is most noticeable is the significant outlier that happened during one run under stress. This is only seen once, hence not many conclusions can be drawn from it. In general, it can be said that outliers like this, where the performance is more severely impacted, could be dangerous, but since they don't happen in the general case, they are difficult to predict. A performance hit like the one shown for QEMU/KVM might be able to make a system miss deadlines and can be replicated during stress. This can be a sign of broken temporal isolation.

Rare, big hits, to performance could be considered more dangerous than predictable small performance degradations. A decrease of 0.01% would not impact most systems, especially not if it is expected and can be accounted for. Unexpected performance dips, on the other hand, could be detrimental, causing systems to fail. Although this could be very dangerous, it is impossible to tell from the experiments in this thesis with what frequency this would happen, or if this one run could be due to some testing error. Because of this, further research would be needed to see if this is a recurring problem or if there are other reasons the outlier happened.

Additionally, it is important to highlight that the scheduling never failed and no deadlines in the test itself were missed, even though the performance was impacted. Part of the test was to check that all threads are run the same number of times, otherwise the test will stop and the result will be a failure. This never happened on any of the runs, from this we can know that the scheduler continues to be fair during the test regardless of the stress. This reinforces the trust in the hypervisor as being reliable.

5.4 Threats to Validity

One factor that could impact the stability of the thread metrics test is the time it takes to print the results each iteration. In the original test, the interval was set to 30s so that the printing time is insignificant [25]. The choice was made to have 5s intervals instead to be able to collect more data points, with the risk that the printing time could have more of an impact. The printing time should be the same each iteration, but this is not guaranteed.

Another factor that might affect the results of the thread metrics test is the exactness of carrying out the test. For Xen the start and stops of the VMs can be scripted, making the timings consistent between runs. This was not a possibility in the same way with the QEMU/KVM setup, but here the VMs had to be started manually. This leads to some variations in timings which makes the results a little less consistent.

Because MiBench originally was developed for a unix based system some changes had to be done to port it to ZephyrOS. MiBench qsort uses input from a file for the data it's supposed to sort. This data had to be converted to a header file and loaded statically into image of the zephyr guest. This port might affect the validity of the results since it is not executed with the native MiBench implementation.

Although both QEMU/KVM and Xen were evaluated using Ubuntu-based operating systems, serving as the host OS for QEMU/KVM and Dom0 for Xen, they did not rely on identical disk images or Linux kernel versions. As a result, the evaluation of the hypervisors is not entirely isolated from the influence of differing system configurations.

The system was rebooted whenever possible between tests, but it is a possible source of error that the system might not have been a clean slate at the start of a test. There can be instructions saved in cache and other memory left behind that could affect the execution of a new test, both for better or worse, which could cause variance in the result not connected to the test.

One source of error that could affect the performance of the systems is the temperature of the Raspberry Pi. Because of the high load the temperature of the chip increased significantly which gives succeeding runs different conditions than its predecessors. It is unknown how much this affects the results of this thesis since there is no noticeable decline with increased number of tests.

5.5 Evaluation of the Test Framework

The different test categories, latency measurements of Section 5.1, MiBench benchmarks of Section 5.2, and thread-level metrics of Section 5.3 were selected to evaluate distinct aspects of hypervisor behavior. The goal was to assess temporal isolation across multiple layers of system operation, with the hope that analyzing these different scales would reveal varied effects and potential causes of interference. For instance, the outliers observed in the latency measurements Section 4.1 could stem from minor delays introduced by larger operations running concurrently, something that may not be apparent independently. On the other hand, how well a hypervisor handles cache conflicts might not be noticeable if only looking at low-level kernel operations. This is in reference to RQ1 to develop the first step of a set of evaluation metrics for hypervisors in a safety context.

The latency benchmark measures the time it takes for Zephyr's kernel operations for the hypervisor. These operations are crucial and happen many times during the execution of a program. On this level, even small impacts could cause delays since they would happen multiple times. This is also an aspect of performance that is not as commonly measured, compared to the other test areas that have been researched more [4] [14].

The MiBench tests measure how well a system performs tasks that are common in real-time systems. The two successfully ported ones used for this thesis were basicmath and qsort. Bitcount and Susan are also a part of the standard MiBench automotive suite, although these were not successfully ported, but could provide further insight into hypervisor performance in a real-time system setting. Basicmath performs math operations that seldom have direct hardware support, and qsort sorts a large data array with the quicksort algorithm, since sorted data is often needed in real-time systems [20]. Furthermore, the disturbing guest accesses memory when running qsort and seems to slightly affect Xen during stress which might be because of cache misses. To ensure that cache misses are the cause, one improvement would be to use different forms of stressing guests to isolate if worse execution time only happens with a guest designed to disturb memory. Even though these tests are directed towards real-time systems, they do not take deadlines into consideration.

The thread metrics test evaluates scheduling over a longer time. Here the details in every operation is not visible as in the latency measures but rather how much time the VMs gets to run and how the hypervisor is able to handle the scheduling of multiple VMs. This was mostly investigated when running on different CPUs, since this is the most common

case when running safety-critical systems, although the case of sharing a CPU was also tested.

These three together test different levels of performance of the hypervisors, which we consider to give an overview of the hypervisor as a whole in the context of understanding the impact of a disturbing VM on a system. Specifically, the framework allows us to see the impact on kernel functions' latencies, floating point operations, cache congestion, and scheduling over longer periods of time. Together, this makes it possible to see to what degree the impact of a disturbing VM is noticeable to the testing VM, in other words, seeing if there exists poor temporal isolation.

In the bigger picture, the aim is to see if hypervisors can be used in safety-critical and mixed-criticality environments, such as the automotive industry. This requires that the test areas can ensure that there are no aspects missing that could cause a critical failure which would have dire consequences. The test areas evaluated here are a start, but not an extensive test bench that would guarantee the safety of a system.

5.6 Evaluation of Research Questions

This section revisits the research questions outlined in the introduction and discusses how the findings of this thesis contribute to answering them. Each research question is addressed individually, drawing on the results obtained from the experimental evaluations and analysis. The goal is to assess to what extent the research objectives have been met and to reflect on the implications and limitations of the findings.

5.6.1 Research Question 1

RQ1: What needs to be part a suitable taxonomy of KPIs, and which indicative KPIs can be used to evaluate guest isolation in embedded/automotive contexts?

To answer **RQ1**, this thesis proposes a starting point for a taxonomy of KPIs specifically suited for evaluating guest isolation in embedded and automotive systems running on a hypervisor. The taxonomy is designed to capture performance characteristics across multiple layers of the system, reflecting both time-critical requirements and general task execution constraints typical in these domains. The proposed KPI categories are as follows:

An effective KPI taxonomy could address:

- Temporal Performance: Ensuring that critical operations at the kernel level do not experience delays that could compromise real-time constraints.
- Application-Level Behavior: Measuring performance during domain-specific, common tasks to ensure acceptable degradation levels that do not affect overall system functionality.
- Memory Intense Tasks: To what degree is the temporal isolation impacted of misses in a shared last level cache caused by another guest.
- Long-Term Stability: Observing system behavior over extended periods to verify that temporal isolation is maintained and that no performance drift or cumulative

degradation occurs. There is also a possibility to catch rare errors that would go unnoticed during shorter tests.

In this thesis, these aspects were evaluated using benchmarking tools, some of which were specifically tailored for automotive embedded systems. To ensure meaningful and reliable results, it is recommended that these evaluations be conducted using the actual guest operating system intended for deployment. This minimizes variability introduced by using other OSes, which may not represent the behavior of the target system. Similarly, testing should be performed on the same hardware platform planned for production to accurately capture any hardware-specific behaviors or failures that could affect guest isolation.

Furthermore, due to limited time, the evaluation of I/O performance in hypervisor-based systems was not conducted in this thesis, but it represents an important extension of this work. I/O subsystems are often shared among guests, and as such, they can become a critical contention point affecting isolation guarantees. Including I/O performance in the KPI taxonomy would provide a more holistic understanding of guest isolation, particularly in systems where real-time responsiveness and determinism depend heavily on timely access to I/O resources.

5.6.2 Research Question 2

RQ 2: Do embedded hypervisors actually differ in temporal isolation, and how effectively can a framework expose those differences and each hypervisors performance strengths?

The test framework displays that there is an observable impact of the disturbing VM across all tests, which would indicate that the temporal isolation is imperfect. This is seen on both QEMU/KVM and Xen. There is a difference in how the impact is presented in the hypervisors. Xen has a slight degradation in performance on a higher level and has large delays occasionally on operating system-level tasks on the lower level. QEMU/KVM, on the other hand, has one outlier on the long-term stability test, and otherwise, the impact on temporal isolation takes the form of higher variability during stress. This is not always consistent, and more research would be preferred to find a root cause and a more definitive answer.

All in all, QEMU/KVM excels in shorter execution time across all tests, while Xen excels in consistency. When running the test and the stress on the same CPU core, the performance drops to about half for both hypervisors, but again, Xen shows more stability. This has exceptions to the rule, and no hypervisor can be said to be undoubtedly better than the other.

5.7 Ethical Considerations

The ethical implications of this thesis primarily relate to two areas: sustainability and safety.

From a sustainability perspective, virtualization technologies such as hypervisors have the potential to improve the resource efficiency of future embedded products. By enabling multiple guests to share the same hardware platform, the overall number of required

computing nodes can be reduced. This leads to less demand for dedicated chips and associated electronic components, which in turn reduces material consumption, energy usage, and electronic waste. In large-scale manufacturing domains, such as the automotive industry, such optimization can contribute meaningfully to more sustainable production and operation.

From a safety and reliability perspective, the results of this thesis can help support the validation of hypervisors in safety-critical applications. In the automotive domain, where system failures can have fatal consequences, ensuring that critical functionality remains unaffected by non-critical tasks is of utmost importance. The test framework and evaluation methods proposed here aim to identify potential weaknesses in temporal isolation that could lead to unsafe behavior. Equally important, however, is that the validation methodology itself be accurate and reliable false positives or misleading results could undermine trust in the safety assurance process. Therefore, any testing and evaluation of safety-critical systems must be performed with rigor, transparency, and repeatability to ensure that conclusions drawn can be trusted.

5.8 Future work

There are several areas that could be considered the next step for future work. One of them is extending the testing to be able to benchmark the hypervisor against real time deadlines. In this thesis common deadlines are compared to the delays seen but this could be extended to test where this could become measurable and comparable between hypervisor in a clearer way. This would also be able to benchmark if the delays and degradations experienced in the tests of the thesis are negligible or not.

Other functions of the hypervisor, such as handling I/O, would also need to be an addition to the test bench for it to be exhaustive. With this most important aspects of a hypervisor in an embedded system could be benchmarked to make sure it is suitable for the needs of specific safety-critical systems.

One aspect that was part of the results of this thesis that would be interesting to continue investigating is the start-up phase of hypervisors and their impact on performance. The root cause for this was not identified during this thesis, but several possible causes were proposed. It would be interesting to see if similar ramp-up phases happen to other hypervisors and if there are ways to mitigate this.

A test framework like this could be used to test many more hypervisors than presented in this thesis. There are plenty more options that also could be good choices for mixed-criticality embedded systems and more research is needed to see if they would be satisfactory in keeping VMs isolated from each other.

6 Conclusion

The aim of this thesis is to investigate temporal isolation of hypervised systems on ARM architecture and what could be included in a test framework for this purpose. The tests were chosen to evaluate different layers of operations, from kernel operations to applications with memory intense tasks to scheduling tests over long periods of time.

The findings of the tests performed show that the impact of having a stressing VM present when running has a noticeable impact on the performance. All tests gave some difference in performance, although the change was minor in most cases. The largest persistent performance degradation observed was of 1%.

The findings we found most interesting were the outliers that seem to be caused from the interference of the stress VM. There exists occasional outliers in all tests that impact the system much more than 1%. These are worrying form a safety point of view since they can cause missed deadlines and are difficult to predict. On a low level this is seen mostly on Xen and on a higher level this is seen more on QEMU/KVM.

Lastly, the ramp-up period of QEMU/KVM is a finding that differs from expected behavior. The reason behind this is not clear from the performed tests, but what can be said is that it only affects QEMU/KVM and not Xen. The implication is that Xen seem to be more consistent on a larger scale and have more predictable behavior.

To summarize, virtualization can be a valuable tool in mixed criticality systems. This thesis provides a framework with different layers that complement each other when assessing temporal isolation of the system. The result show that impacts to temporal isolation can be observed to a small degree, but the implications have to be evaluated on a case by case level. The areas presented does not prove without reasonable doubt that the assessment is exhaustive and further research and development is needed to extend the testing to other areas.

Bibliography

[1] Z. Guo, K. Koufos, M. Dianati, and R. Woodman, "State-of-the-art virtualisation technologies for the centralised automotive E/E architecture," *Frontiers in Future Transportation*, vol. 6, p. 1 519 390, 2025. DOI: `10.3389/ffutr.2025.1519390`.

[2] M. Cinque, L. De Simone, and D. Ottaviano, "Temporal isolation assessment in virtualized safety-critical mixed-criticality systems: A case study on Xen hypervisor," *The Journal of Systems and Software*, vol. 216, p. 112 147, 2024. DOI: `10.1016/j.jss.2024.112147`.

[3] M. Cinque, D. Cotroneo, L. De Simone, and S. Rosiello, "Virtualizing mixed-criticality systems: A survey on industrial trends and issues," *Future Generation Computer Systems*, vol. 129, pp. 315–330, 2022, ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2021.12.002`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0167739X21004787`.

[4] J. Martins and S. Pinto, "Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems," Mar. 2023. DOI: `10.48550/arXiv.2303.11186`.

[5] Y. Shen, L. Wang, Y. Liang, S. Li, and B. Jiang, "Shyper: An embedded hypervisor applying hierarchical resource isolation strategies for mixed-criticality systems," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 1287–1292. DOI: `10.23919/DATE54114.2022.9774664`.

[6] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, 2018, pp. 1651–1657. DOI: `10.1109/ICIT.2018.8352429`.

[7] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and Software Support for Virtualization* (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers, 2017. DOI: `10.2200/S00754ED1V01Y201701CAC038`.

[8] J. Sheehan, *10 key functions of an operating system explained*, Accessed: July 15, 2025, Feb. 2025. [Online]. Available: `https://synchronet.net/operating-system-key-function/#Overview_of_Core_Responsibilities`.

[9] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, 1974. DOI: `10.1109/MC.1974.6323581`.

[10] A. Abudaqa, T. Al-Kharoubi, M. Mudawar, and A. Kobilica, "Simulation of ARM and x86 microprocessors using in-order and out-of-order CPU models with Gem5 simulator," May 2018, pp. 317–322. DOI: `10.1109/ICEEE2.2018.8391354`.

[11] ARM Limited, *Armv8-A virtualization*, Doc ID 102142, 2019.

[12] D. Revelle, “Scientific Collaboration Needs Better Software Practices,” *;login: The USENIX Magazine*, vol. 41, no. 3, pp. 24–34, Fall 2016. [Online]. Available: https://www.usenix.org/system/files/login/articles/105498-Revelle.pdf.

[13] D. Firesmith, *Virtualization via Virtual Machines*, Carnegie Mellon University, Software Engineering Institute’s Insights (blog), Accessed: 2025-Apr-17, Sep. 2017. [Online]. Available: https://insights.sei.cmu.edu/blog/virtualization-via-virtual-machines/.

[14] L. Abeni and D. Faggioli, “Using Xen and KVM as real-time hypervisors,” *Journal of Systems Architecture*, vol. 106, p. 101 709, 2020. DOI: https://doi.org/10.1016/j.sysarc.2020.101709.

[15] Xen Project, Accessed: 2025-05-27. [Online]. Available: https://xenproject.org/.

[16] *Xen ARM with Virtualization Extensions whitepaper*, https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper, [Accessed 11-03-2025], 2018.

[17] S. Xi, M. Xu, C. Lu, L. T. Phan, C. Gill, O. Sokolsky, and I. Lee, “Real-Time Multi-Core Virtual Machine Scheduling in Xen,” in *Proceedings of the 12th International Conference on Embedded Software*, ser. EMSOFT ’14, ACM, 2014, pp. 1–10, ISBN: 978-1-4503-3052-7. DOI: 10.1145/2656045.2656061.

[18] C. Dall and J. Nieh, “KVM/ARM: The design and implementation of the linux ARM hypervisor,” *SIGPLAN Not.*, vol. 49, no. 4, pp. 333–348, Feb. 2014, ISSN: 0362-1340. DOI: 10.1145/2644865.2541946. [Online]. Available: https://doi.org/10.1145/2644865.2541946.

[19] Zephyr Project Contributors, *About the Zephyr Project*, Accessed: April 18, 2025, 2025. [Online]. Available: https://zephyrproject.org/learn-about/.

[20] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[21] The Debian Project, *Debootstrap - Debian Wiki*, Debian Wiki, Last updated 2025-01-21. Accessed on 2025-04-15, Jan. 2025. [Online]. Available: https://wiki.debian.org/Debootstrap.

[22] openSUSE Wiki Contributors, *HCL:Raspberry Pi4*, openSUSE Project. [Online]. Available: https://en.opensuse.org/HCL:Raspberry_Pi4 (visited on 04/15/2025).

[23] Linux man-pages project, *taskset(1) - Linux man page*, Online; Linux man-pages, Accessed: 2025-05-21. [Online]. Available: https://man7.org/linux/man-pages/man1/taskset.1.html.

[24] Zephyr Project Contributors, *Zephyr RTOS Latency Benchmarks*, https://github.com/zephyrproject-rtos/zephyr/tree/main/tests/benchmarks/latency_measure, Accessed: May 2025, 2024.

[25] W. E. Lamie, *tm_preemptive_scheduling_test*, version 6.1.7, Defines the preemptive scheduling test, Microsoft Corporation, Oct. 15, 2021. [Online]. Available: https://github.com/zephyrproject-rtos/zephyr/tree/main/tests/benchmarks/thread_metric.

A

U-Boot script for Xen

The following script was used to boot the Xen hypervisor with a Linux-based Dom0 on a Raspberry Pi 4 using U-Boot. It manually loads the necessary binaries into memory and configures the Flattened Device Tree (FDT) for Xen multiboot compatibility.

```
# Load Xen hypervisor, Linux kernel, and Device Tree Blob (DTB) from MMC
load mmc 0:1 0x39000000 xen
load mmc 0:1 0x37000000 Image.gz
load mmc 0:1 0x34000000 bcm2711-rpi-4-b.dtb

# Set up the Device Tree
fdt addr 0x34000000
fdt resize
fdt set /chosen \#address-cells <1>
fdt set /chosen \#size-cells <1>

# Define the kernel module for Xen multiboot
fdt mknod /chosen module@0
fdt set /chosen/module@0 compatible "xen,linux-zimage" "xen,multiboot-module"
fdt set /chosen/module@0 reg <0x37000000 12481918>

# Set kernel boot arguments
fdt resize
fdt set /chosen/module@0 bootargs "rw root=/dev/mmcblk0p2 rdinit=usr/sbin/init rootwait earlyprintk=serial,ttyAMA0 console=hvc0 earlycon=xenboot dom0_mem=2G"

# Set Xen hypervisor boot arguments
fdt set /chosen xen,xen-bootargs "console=dtuart dtuart=serial0 sync_console dom0_mem=3G"

# Boot using the Xen hypervisor
booti 0x39000000 - 0x34000000
```

Listing A.1: U-Boot script for booting Xen on Raspberry Pi 4

B

All Figures of Latency Measurements

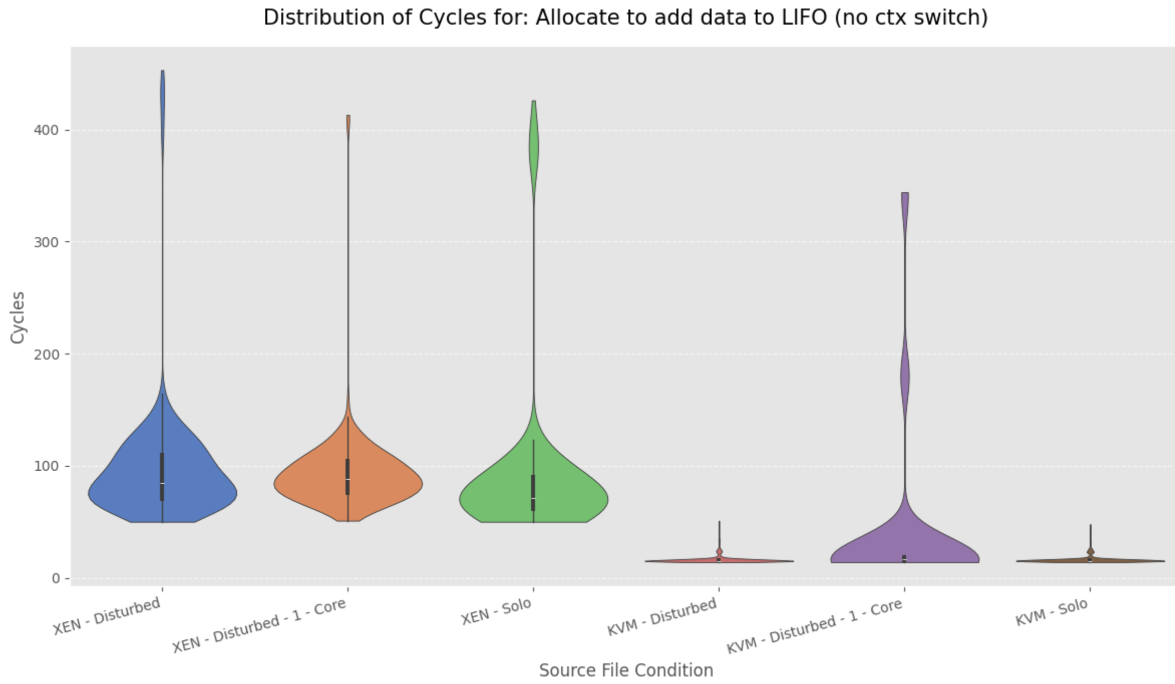

Figure B.1: Distribution of cycles across different configurations for the Latency Measurement *Allocate to add data to LIFO (no context switch)*

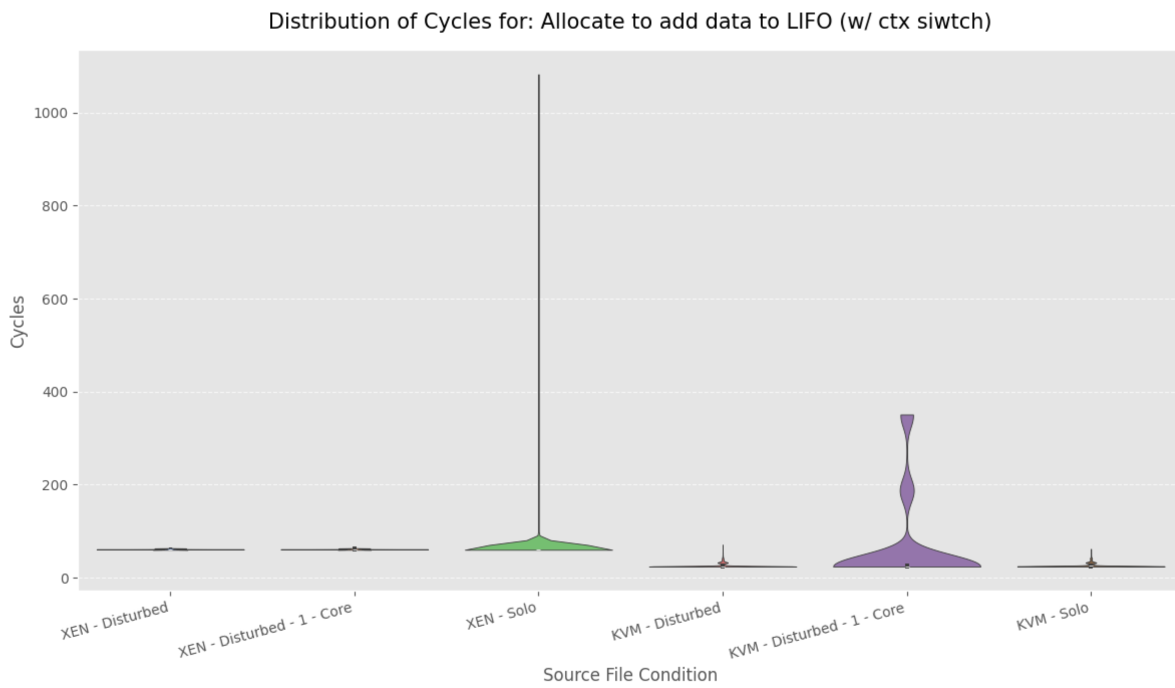

Figure B.2: Distribution of cycles across different configurations for the Latency Measurement *Allocate to add data to LIFO (with context switch)*

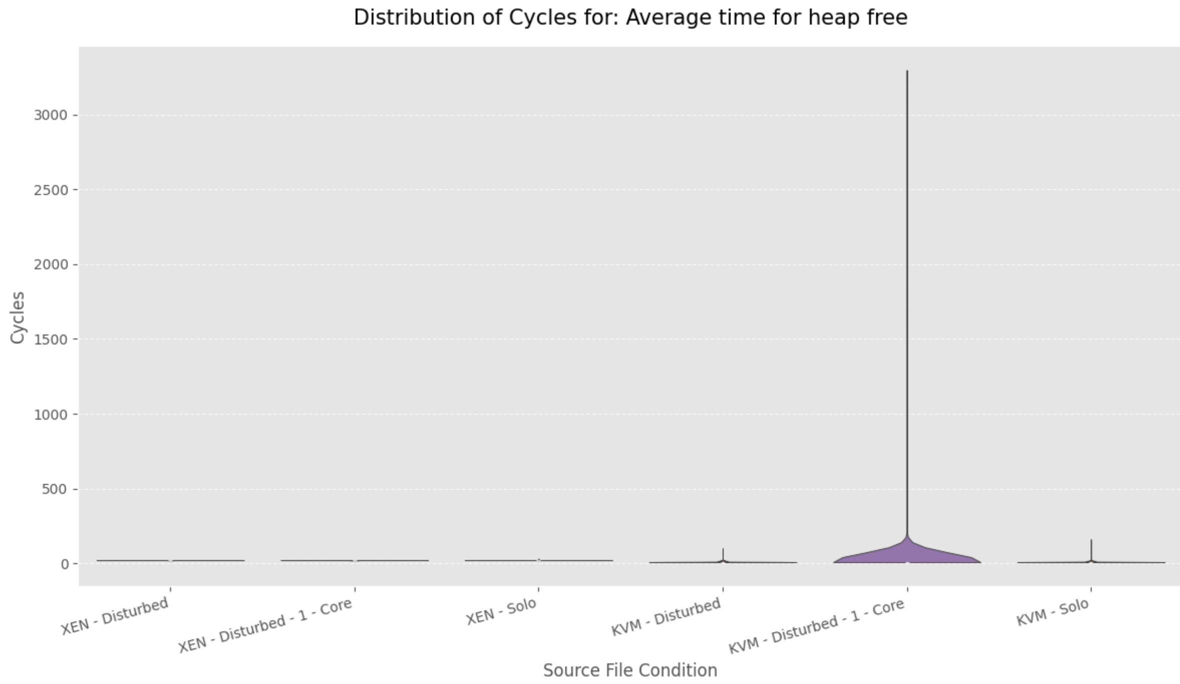

Figure B.3: Distribution of cycles across different configurations for the Latency Measurement *Average time for heap free*

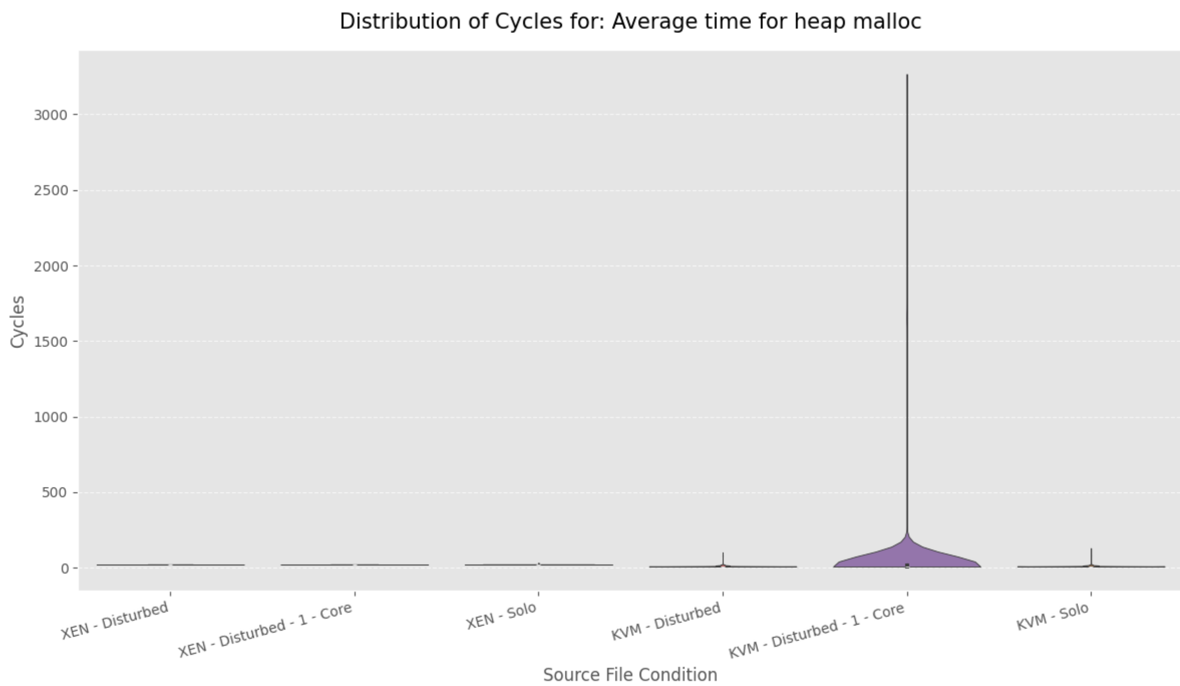

Figure B.4: Distribution of cycles across different configurations for the Latency Measurement *Average time for heap malloc*

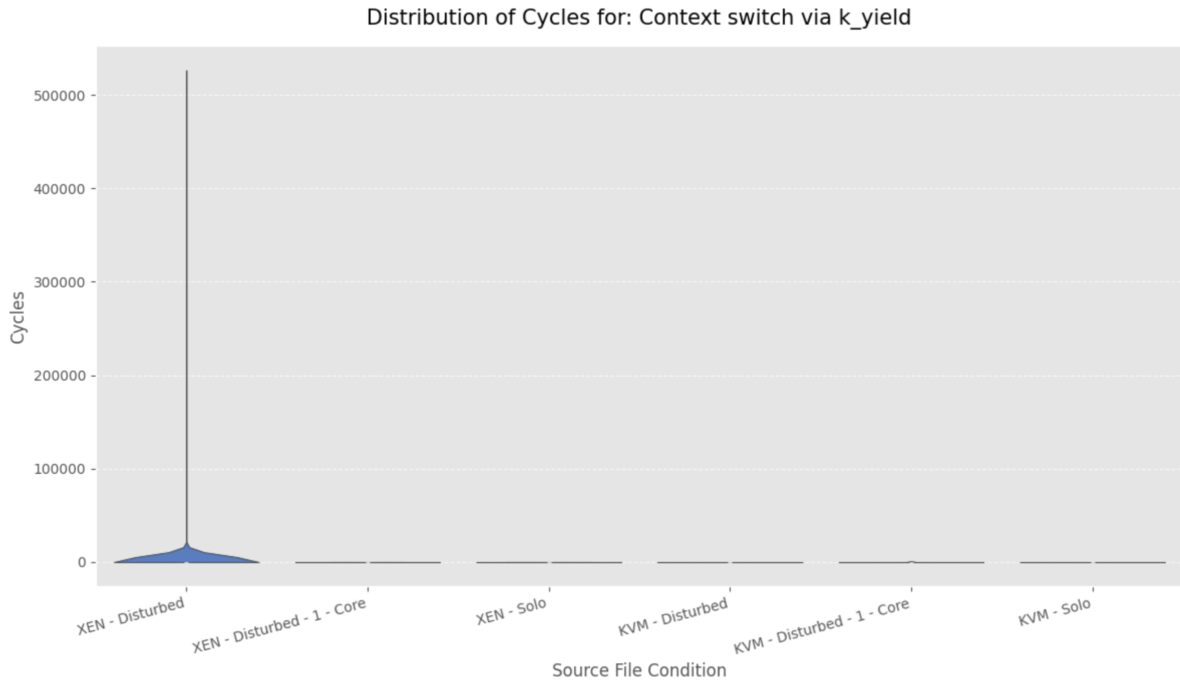

Figure B.5: Distribution of cycles across different configurations for the Latency Measurement *Context switch via k_yield*

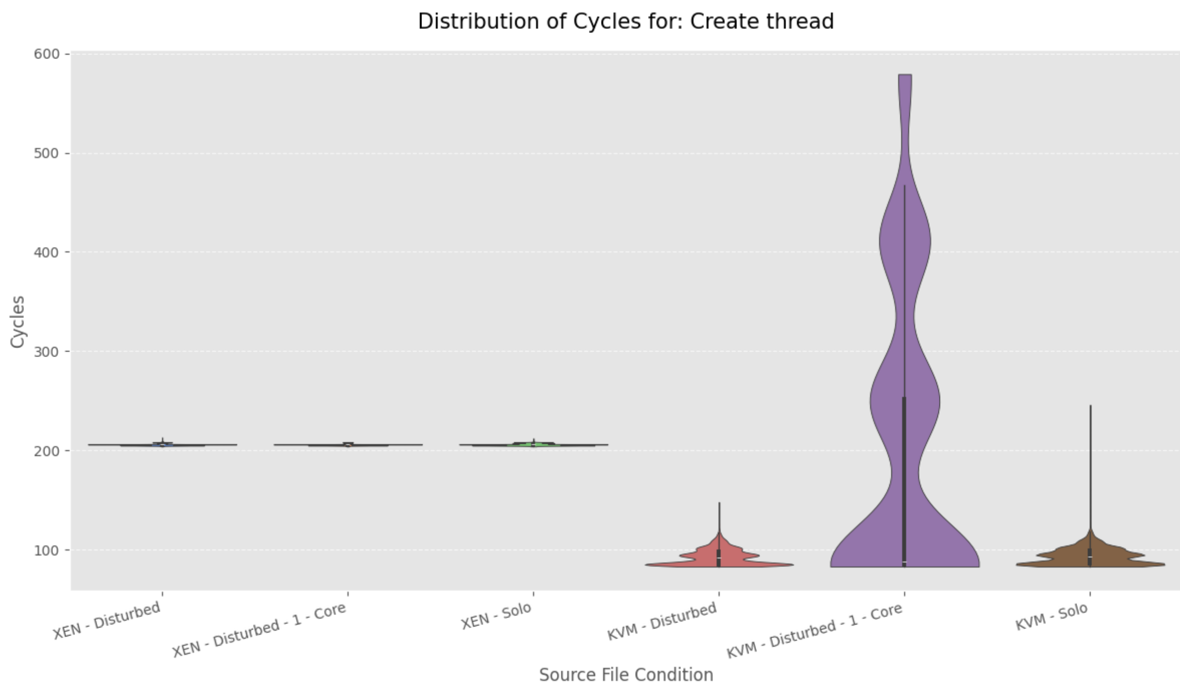

Figure B.6: Distribution of cycles across different configurations for the Latency Measurement *Create thread*

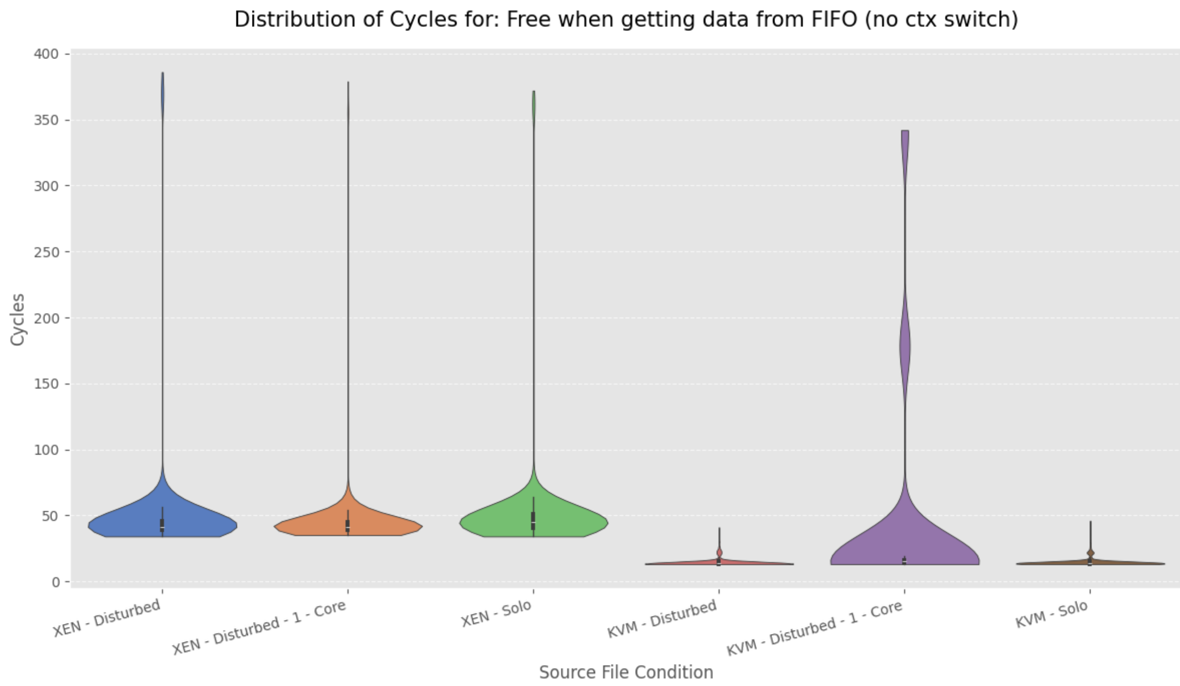

Figure B.7: Distribution of cycles across different configurations for the Latency Measurement *Free when getting data from FIFO (no context switch)*

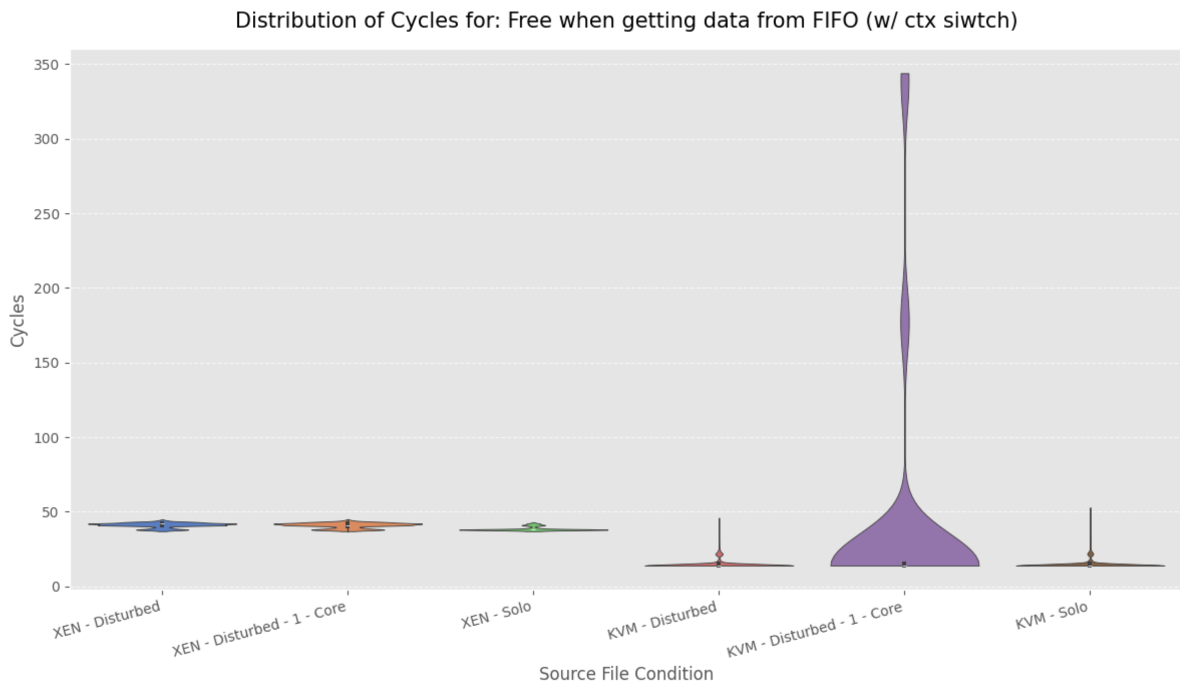

Figure B.8: Distribution of cycles across different configurations for the Latency Measurement *Free when getting data from FIFO (with context switch)*

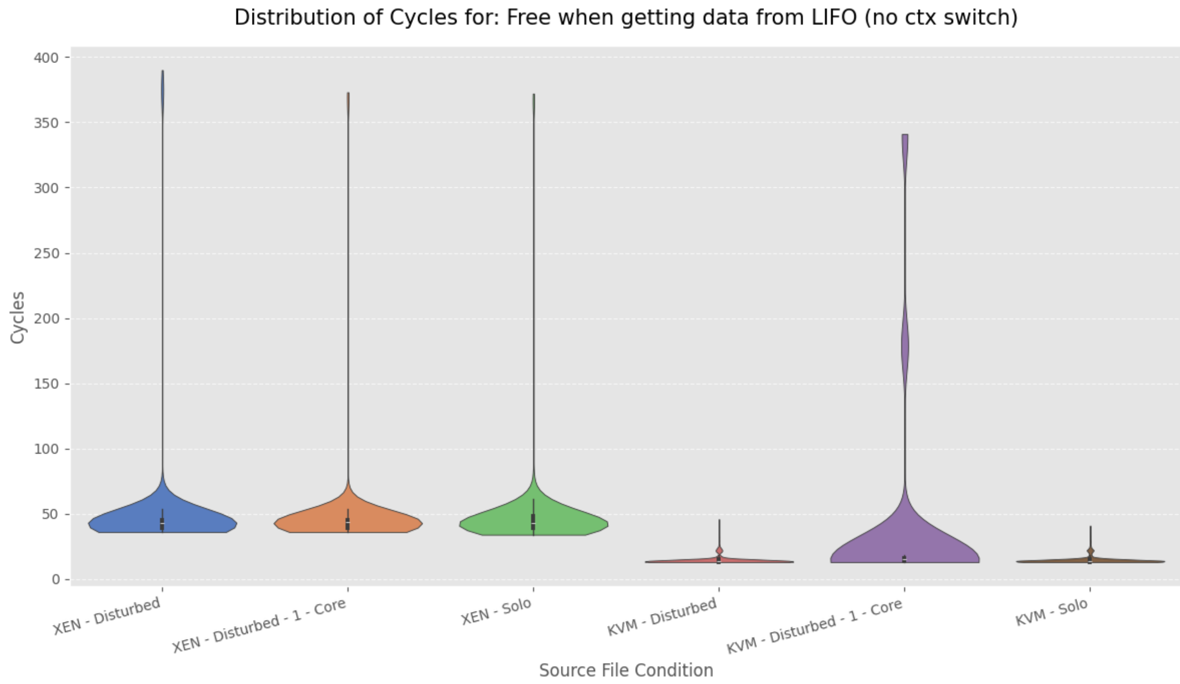

Figure B.9: Distribution of cycles across different configurations for the Latency Measurement *Free when getting data from LIFO (no context switch)*

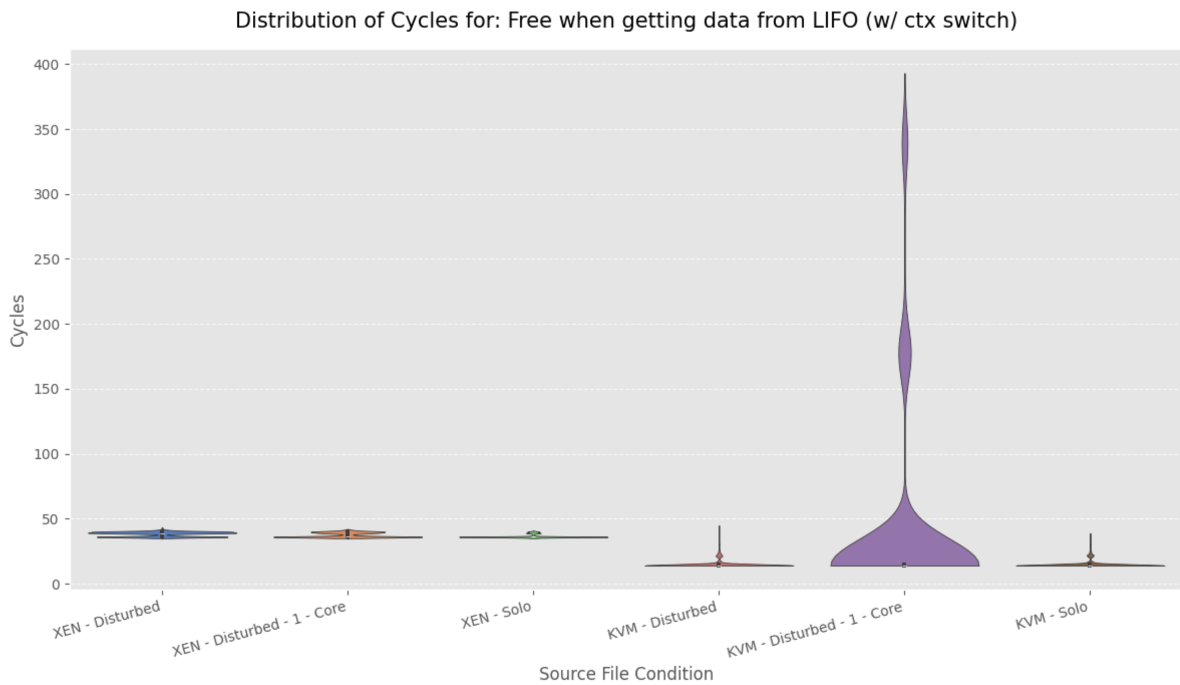

Figure B.10: Distribution of cycles across different configurations for the Latency Measurement *Free when getting data from LIFO (with context switch)*

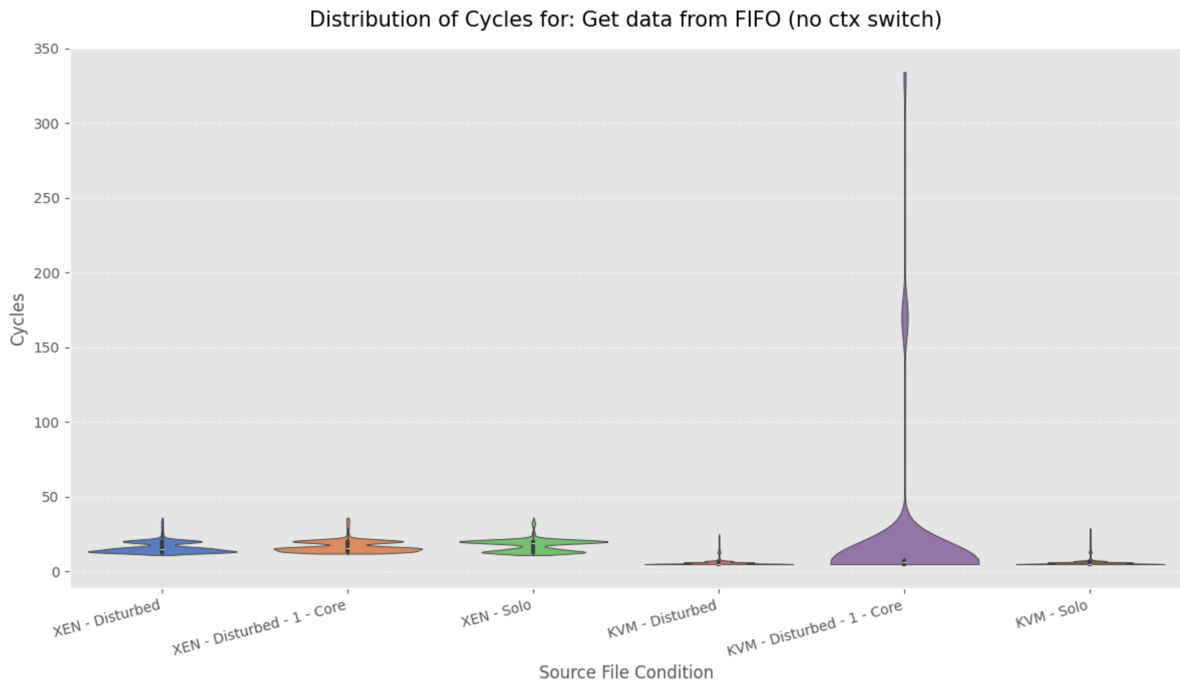

Figure B.11: Distribution of cycles across different configurations for the Latency Measurement *Get data from FIFO (no context switch)*

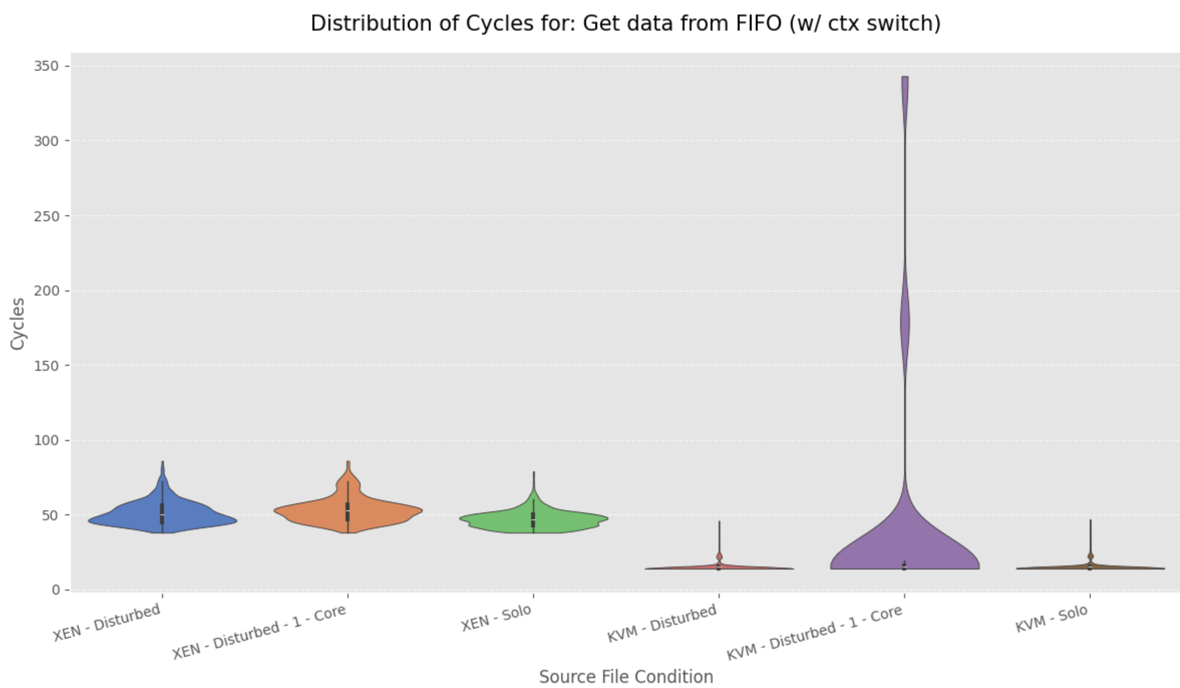

Figure B.12: Distribution of cycles across different configurations for the Latency Measurement *Get data from FIFO (with context switch)*

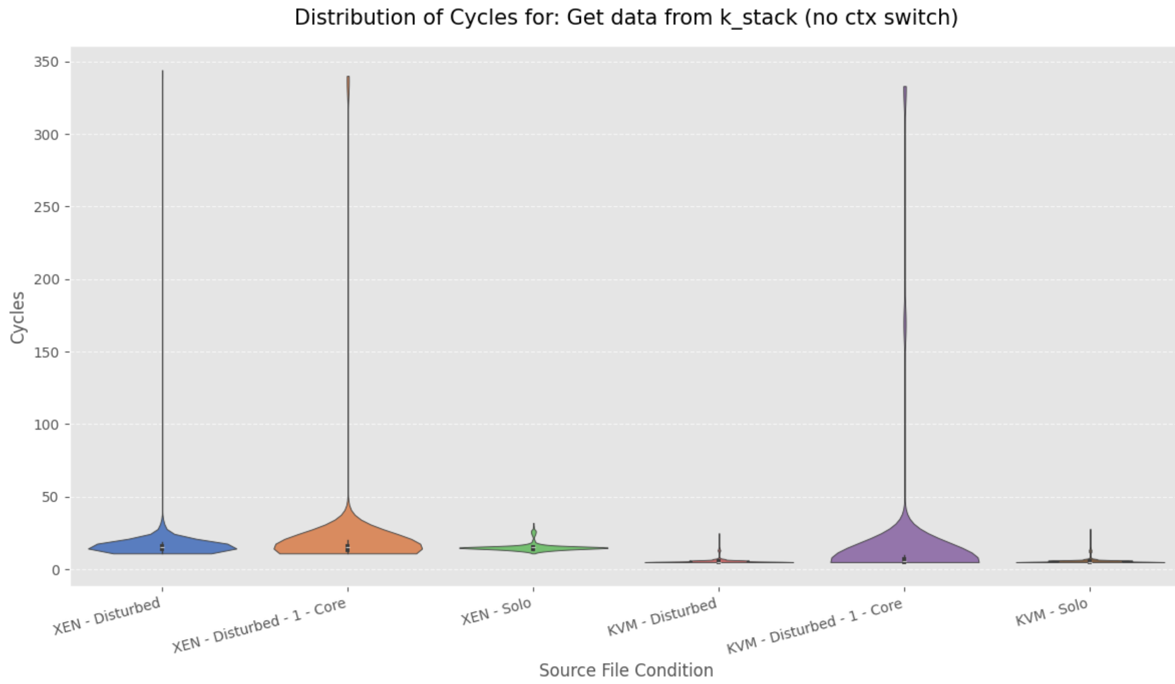

Figure B.13: Distribution of cycles across different configurations for the Latency Measurement *Get data from kernel stack (no context switch)*

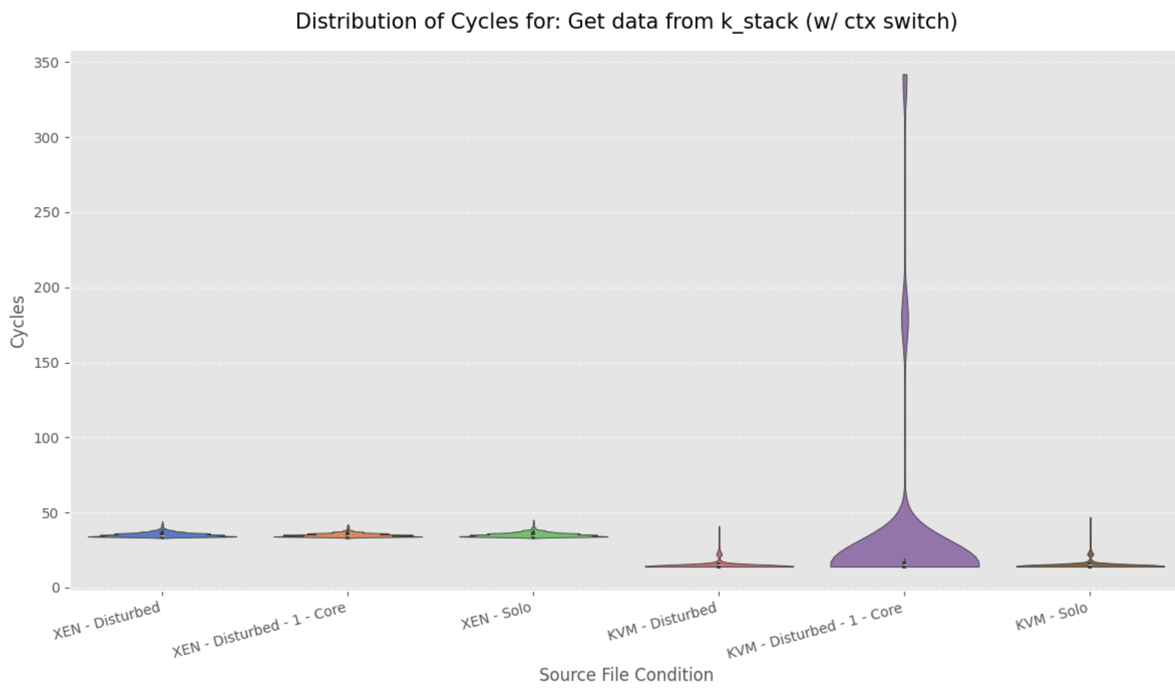

Figure B.14: Distribution of cycles across different configurations for the Latency Measurement *Get data from kernel stack (with context switch)*