Inter-Core Interference Mitigation in a Mixed Criticality System

Michael Glenn Hinton
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd
Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation
Hinton, Michael Glenn, "Inter-Core Interference Mitigation in a Mixed Criticality System" (2020). Theses and Dissertations. 8648.
https://scholarsarchive.byu.edu/etd/8648

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
Inter-Core Interference Mitigation in a Mixed Criticality System

Michael Glenn Hinton

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

James Archibald, Chair
Jeffrey Goeders
Phil Lundrigan

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2020 Michael Glenn Hinton
All Rights Reserved
ABSTRACT

Inter-Core Interference Mitigation in a Mixed Criticality System

Michael Glenn Hinton
Department of Electrical and Computer Engineering, BYU
Master of Science

In this thesis, we evaluate how well isolation can be achieved between two virtual machines within a mixed criticality system on a multi-core processor. We achieve this isolation with Jailhouse, an open-source, minimalist hypervisor. We then enhance Jailhouse with core throttling, a technique we use to minimize inter-core interference between VMs. Then, we run workloads with and without core throttling to determine the effect throttling has on interference between a non-real-time VM and a real-time VM. We find that Jailhouse provides excellent isolation between VMs even without throttling, and that core throttling suppresses the remaining inter-core interference to a large extent.

Keywords: hypervisor, Jailhouse, virtual machine, mixed criticality system, real-time operating system, core throttling, inter-core interference, interference mitigation
ACKNOWLEDGMENTS

I wish to give my thanks to: my father, Glenn Hinton, for reviewing my thesis, discussing ideas, and pushing me to finish; Professor Archibald, for his encouragement and extensive feedback on my thesis, as well as his guidance throughout my entire graduate career; Tyler Dawson and Royce Hinton, for reviewing and proofreading my thesis; my graduate classmates and professors; my graduate committee; Brigham Young University and the Department of Electrical and Computer Engineering; my family; and above all, God.
# TABLE OF CONTENTS

Title Page .................. i

Abstract ................. ii

Acknowledgements .......... iii

Table of Contents ........ vi

List of Tables ........... vii

List of Figures ........... viii

Chapter 1 Introduction ........ 1
  1.1 Motivation .......................... 1
  1.2 Contribution .................. 1
  1.3 Architectural Overview ........ 2
  1.4 Thesis Outline ................ 3

Chapter 2 Background .......... 4
  2.1 Computer Processors ........ 4
  2.2 CPU Caches .................. 4
  2.3 Locality .................. 5
  2.4 Operating Systems ........ 6
  2.5 Real-Time Operating Systems .. 6
  2.6 Mixed Criticality Systems .. 7
  2.7 Virtual Machines and Hypervisors .. 7
  2.8 The Jailhouse Hypervisor .. 8
  2.9 Metrics .................. 11
    2.9.1 Performance Metrics .... 11
    2.9.2 Predictability Metrics .. 11
  2.10 Cache Interference Mitigation .. 12
  2.11 Intel C-states and P-states .. 13
  2.12 Hardware P-states .......... 13
  2.13 Time Stamp Counter .......... 13
  2.14 Intel VMX .................. 14
  2.15 Related Work .............. 15
    2.15.1 Work related to Jailhouse .. 15
    2.15.2 Other related work .... 16
    2.15.3 Contributions ........ 16

Chapter 3 Methods and Implementation .......... 17
  3.1 Overview .................. 17
B.5 How Jailhouse Uses the Preemption Timer .......................... 56
LIST OF TABLES

4.1 Input Response - Average Speedup - All Combinations . . . . . . . . . . . . . . . . . 31
LIST OF FIGURES

1.1 A Basic Mixed Criticality System with a Hypervisor .......................... 2
2.1 Typical Hypervisor Diagram .................................................. 8
2.2 Jailhouse Hypervisor Diagram .............................................. 9
2.3 Jailhouse Startup Sequence .................................................. 10
3.1 Experimental Setup with Jailhouse, Workloads, and Core Throttling .......... 17
3.2 Memory Layout and Hardware Configuration ................................. 21
4.1 Workload Comparison, No Interference Workload, No Throttling .............. 25
4.2 Input Response - Speedup - SHA3 Inmate Workload .......................... 27
4.3 Input Response - Speedup - CSB Inmate Workload .......................... 28
4.4 Input Response - Speedup - RA Inmate Workload ............................ 28
4.5 Input Response - Average Speedup - SHA3 Inmate Workload .................. 30
4.6 Input Response - Average Speedup - CSB Inmate Workload .................. 30
4.7 Input Response - Average Speedup - RA Inmate Workload .................... 31
4.8 Input Response - Throughput - SHA3 Inmate Workload ........................ 32
4.9 Input Response - Throughput - CSB Inmate Workload ........................ 33
4.10 Input Response - Throughput - RA Inmate Workload ........................ 33
4.11 Input Response - MOET Difference - SHA3 Inmate Workload .................. 35
4.12 Input Response - MOET Difference - CSB Inmate Workload .................. 35
4.13 Input Response - MOET Difference - RA Inmate Workload .................... 36
4.14 Input Response - Jitter Difference - SHA3 Inmate Workload .................. 36
4.15 Input Response - Jitter Difference - CSB Inmate Workload .................... 37
4.16 Input Response - Jitter Difference - RA Inmate Workload .................... 37
4.17 Throttle Response - Speedup - SHA3 Inmate Workload ........................ 39
4.18 Throttle Response - Speedup - CSB Inmate Workload ........................ 39
4.19 Throttle Response - Speedup - RA Inmate Workload ........................ 40
4.20 Throttle Response - Throughput - SHA3 Inmate Workload .................... 40
4.21 Throttle Response - Throughput - CSB Inmate Workload .................... 41
4.22 Throttle Response - Throughput - RA Inmate Workload .................... 41
B.1 State machine for the preemption timer mechanism added to Jailhouse ........ 58
CHAPTER 1. INTRODUCTION

1.1 Motivation

Real-time operating systems (RTOSs) are increasingly being deployed side by side with conventional OSs. An RTOS can provide real-time responsiveness and reliability for critical tasks, while a conventional OS can provide access to a vast software ecosystem. However, this is often accomplished by using two separate systems with separate computer hardware, including processors, motherboards, storage devices, and power supplies. The cost of deploying two systems instead of one can be prohibitive, especially in fields like the Internet of Things (IoT).

In contrast, a mixed criticality system is the best of both worlds: it combines an RTOS and a conventional OS into a single system. This can be done by having the OSs run on the same processor in separate VMs under a hypervisor.

When OSs run on the same processor, hardware resources are more likely to be shared, leading to unwanted interference due to coupling. A good way to reduce this interference is to minimize resource sharing between the OSs as much as possible. Thus, partitioning hypervisors like Jailhouse exist which only allow their child VMs to run on separate cores and to access separate hardware devices.

While this greatly reduces the potential for interference, it does not eliminate it. There are still some inherently shared inter-core hardware resources (most notably CPU caches) that can lead to non-negligible interference in certain cases.

1.2 Contribution

In this thesis, we develop a technique to mitigate this leftover interference. We enhance the Jailhouse hypervisor with a core throttling mechanism that can suppress the execution of one of the child VMs, reducing the interference felt by the other running VMs. With Jailhouse, we then...
implement a system representing a mixed criticality system and show that this throttling technique suppresses inter-core interference to a large extent.

1.3 Architectural Overview

Figure 1.1 depicts a basic diagram of a mixed criticality system with a hypervisor running two virtual machines – one real time, one not. Each VM runs on processor cores assigned to it by the hypervisor. The hypervisor enforces this division by preventing the VMs from using cores they are not assigned. In addition to cores, there are other hardware resources that can be made exclusive to a single VM, including external IO devices as well as portions of main memory and CPU caches.

![Figure 1.1: A Basic Mixed Criticality System with a Hypervisor](image-url)
1.4 Thesis Outline

This thesis is outlined as follows: in Chapter 1 (this chapter), we introduce the aims and motivation of this thesis and outline the work done; in Chapter 2, we discuss background topics necessary to understand this thesis, as well as prior related work; in Chapter 3, we discuss our experimental framework; in Chapter 4, we discuss the experiments and experimental results; and in Chapter 5, we draw conclusions from those results and identify future work.
CHAPTER 2. BACKGROUND

This chapter is an overview of topics, concepts, and terminology necessary to understand this thesis.

2.1 Computer Processors

For the purposes of this thesis, a processor or central processing unit (CPU) will refer to a computer chip comprising one or more cores. A core is the main unit of the CPU capable of executing a sequence of instructions. Modern processors typically have multiple cores.

The execution of a sequence of instructions is called a thread of execution or simply a thread. If the CPU supports and enables simultaneous multithreading, then two or more threads can run on the same core. Hyper-Threading is Intel’s implementation of simultaneous multithreading.

When two threads run on the same core via simultaneous multithreading, they share the same intra-core resources. Because of this, sibling threads on the same core have a greater potential to interfere with each other than would two threads on separate cores.

2.2 CPU Caches

A CPU cache is a small amount of fast, on-chip memory. Modern processors have multi-level caches in which the cache is subdivided into successively larger portions. Though cache size is measured in bytes, the smallest indivisible unit of data stored by a cache is a block of bytes called a cache line.

On modern Intel processors, the multi-level cache has three levels: the L1 cache, which is the fastest and smallest cache, on the order of a few dozen kilobytes; the L2 cache, which is on the order of a few hundred kilobytes; and the L3 cache, which is on the order of a few megabytes. Unlike the L1 and L2 caches, the L3 cache is shared among the other cores of a multi-core processor. In this case, the L3 cache is the last-level cache (LLC), since there are no other
caches between it and DRAM. The processor registers, while not technically part of the multi-level cache, form yet another de facto cache smaller than the L1 cache.

The L1 cache itself is subdivided into two sub-caches: the L1 data cache (L1 d-cache) and the L1 instruction cache (L1 i-cache). As the names suggest, the i-cache stores program instructions and the d-cache stores program data. The L2 and L3 caches do not distinguish between program data and program instructions.

*Cache thrashing* is when sets of data repeatedly evict each other from the cache. This can occur when the cache is too small to hold the entire working data set of the program at once, or when there is plenty of available space in the cache, but the data sets happen to map to the same cache sets.

The L3 cache on Intel systems is inclusive, which means that a copy of everything in the L1 and L2 caches is also stored in the L3 cache [1]. However, this can lead to inter-core interference, even for programs with a working data set that fits within the core-exclusive L1 and L2 caches. If there is a “noisy neighbor” core that uses the entire shared L3 cache, L1 and L2 cache lines from other cores will be evicted, invalidating the original copies. Thus, subsequent data access by other cores will be slower.

### 2.3 Locality

*Locality* (derived from the word “local”) is the extent to which programs access close data instead of far data. Closeness could be in terms of physical location or in terms of time, leading to two general types of locality: *spatial locality* and *temporal locality*. Spatial locality says that nearby pieces of data will often be accessed close together in time. Temporal locality (temporal as in time, not material things) says that a single piece of data is likely to be accessed again in the near future [2] [3].

Programs with good locality are usually faster than programs with bad locality, since they do not spend as much time accessing data in proportion to working on that data. A program that accesses a single piece of data at a random location and then only uses it once exhibits extremely poor spatial and temporal locality. Thus, random-access workloads (workloads in which the working data set is accessed at random locations) are usually very slow compared to workloads with more predictable memory access patterns.
The following real-world example can help explain locality. A person borrowing a book from a library is analogous to a processor retrieving data from DRAM. It takes time and effort for the person to travel to the library to get a book, and the library limits how many books can be borrowed at one time. Similarly, from the processor’s perspective, it takes a long time to retrieve data from DRAM, and the CPU cache size limits how much data can be retrieved and worked on at one time.

When a person borrows a large book, but then only reads a single page from that book, that is an example of poor spatial locality. This is because none of the other pages are being used. However, it is possible that the person only needs that single page, and refers to it frequently. In this case, good temporal locality is exhibited despite poor spatial locality. If the person only references the page once while borrowing the book, however, then that would be an example of poor spatial and temporal locality.

Repeatedly borrowing books only to read a single page from them once before returning them is analogous to a random-access workload. The person who does this wastes a lot of time and energy getting books from the library per page read. This “book thrashing” corresponds to the cache thrashing that occurs in random-access workloads.

2.4 Operating Systems

Conceptually, an operating system (OS) is a program that abstracts the computer hardware and allows for the execution of other programs. The kernel is the heart of an OS and runs at the highest privilege level in the processing architecture.

Common OSs include Microsoft Windows, Apple macOS, and Linux for desktop machines, and Google Android (a derivative of Linux) and Apple iOS for mobile devices. These OSs are what we will call conventional OSs. Conventional OSs are usually designed to directly interact with users via a graphical interface and run a wide variety of programs.

2.5 Real-Time Operating Systems

A real-time operating system (RTOS) is an OS that enables guaranteed response times to meet deadlines, where if the deadline is missed, system failure occurs. This is a key aspect that
differentiates an RTOS from a conventional OS. So while job throughput is the main metric for conventional OSs, latency and response-time jitter (i.e., how predictably an RTOS responds) are the main metrics for evaluating RTOSs [4]. Another common difference is that while conventional OSs are meant to be general purpose, RTOSs are usually employed to run a specific task.

2.6 Mixed Criticality Systems

A mixed criticality system is a system that has components of more than one criticality level, where the criticality level denotes the level of assurance against failure [5]. Many mixed criticality systems incorporate a real-time subsystem to govern mission-critical tasks and a non-real time subsystem to govern other non-mission critical tasks. This can be implemented by running an RTOS alongside a conventional OS on the same processor.

Examples of mixed criticality systems include many automobiles, aircraft (including UAVs), and industrial machines. In these systems, a mixed criticality architecture helps insulate critical tasks like navigation from non-critical tasks like entertainment, improving reliability and safety.

2.7 Virtual Machines and Hypervisors

A virtual machine (VM) generally means a computer system that runs within another computer system rather than directly on computer hardware. VMs fall under two categories: process virtual machines and system virtual machines. A process virtual machine is software within an OS that runs programs that do not compile down to the same instruction set architecture as the underlying processor. Examples of process virtual machines include language interpreters like a Java VM and a JavaScript engine. A system virtual machine, on the other hand, is an OS instance that runs on an OS-like software called a hypervisor [6]. The hypervisor mediates between the system virtual machine and the actual hardware, unbeknownst to the system virtual machine. For the remainder of this thesis, we use the term VM to refer to a system virtual machine.

Hypervisors, also known as virtual machine monitors (VMMs), run on physical hardware and virtualize hardware resources for their child VMs. Though they also abstract the hardware like OSs, they differ in that they run OSs instead of programs. This hardware virtualization is accomplished by either emulating non-existent hardware or by arbitrating access to existing hardware.
Hypervisors coordinate the sharing of resources among multiple child VMs. In addition, most hypervisors can grant exclusive access to certain resources. This is called *partitioning*, and it can be static or dynamic. *Static partitioning* means that the hardware is partitioned beforehand and cannot change during runtime. *Dynamic partitioning* allows the hypervisor to adjust which VM gets what resource on the fly at runtime, but at the cost of added complexity to the hypervisor.

**Figure 2.1: Typical Hypervisor Diagram**

Hypervisors can also allow child VMs to share the exact same resource *at the same time*. This is called *oversubscription* or *overcommitment*, and the hypervisor achieves this by time-slicing the resource to each VM that accesses it. Figure 2.1 shows a simple diagram of a typical hypervisor with four physical cores being oversubscribed across six virtual cores.

### 2.8 The Jailhouse Hypervisor

Jailhouse is an open-source, Linux-based static-partitioning hypervisor. Compared to other hypervisors like Xen, KVM, and VMware ESX, Jailhouse takes a minimalist approach: it does not
overcommit resources, does no scheduling of resources, and only virtualizes essential resources that cannot be partitioned in hardware. The simplicity of Jailhouse is an intentional design goal that enables it to be on the order of ten thousand lines of code. This is attractive for mixed criticality systems that require certifiably high reliability, including automotive, industrial, and aeronautical systems. Besides having limited features, Jailhouse achieves a small code footprint by reusing Linux as much as possible for things like hardware driver support, runtime management, and bootup [7].

In Jailhouse terminology, child VM allocations are called cells, and the initial Linux cell is the root cell. All hardware resources are initially allocated to the root cell until subsequent cells start and claim resources from the root cell. These non-root cells are called inmates. Figure 2.2 shows a simple diagram of a system with Jailhouse as the hypervisor with a single inmate.

For Jailhouse to start, it requires the host Linux OS to reserve a contiguous section of memory for Jailhouse and its child VMs. Linux then starts up Jailhouse through a Jailhouse kernel.
module that loads the hypervisor code into the reserved memory section. Once loaded, Jailhouse runs and inserts itself between the cores and Linux. (Interestingly, this virus-like startup technique first appeared in malware.) From that point on, Jailhouse is running on bare metal, and Linux itself becomes a child VM under Jailhouse. This process is shown in steps one and two of Figure 2.3.

![Figure 2.3: Jailhouse Startup Sequence](image)

Jailhouse is strictly a static-partitioning hypervisor, so all resources must be divvied up beforehand via cell configuration files. These configuration files are first loaded into the hypervisor to create an inmate. Then, the hypervisor gives the new inmate the designated resources from the root cell and starts the inmate. This process of creating an inmate from the root cell is shown in steps three and four of Figure 2.3. Oversubscription and emulation of hardware resources are not supported, so every resource can belong to only one cell at any given time.
2.9 Metrics

There are two classes of metrics we use in this thesis: performance metrics and predictability metrics. Conventional OSs care more about performance metrics. They can afford to gain performance at the expense of non-determinism, since they typically do not have any real-time deadlines they need to meet. RTOSs and real-time applications, on the other hand, care more about predictability, since system failure will result if even a single deadline is missed. So real-time systems are better described using predictability metrics. We use both classes of metrics to evaluate the effectiveness of core throttling.

2.9.1 Performance Metrics

The main performance metrics we use in our tests are speedup and throughput.

**Speedup** is defined as how much faster a task will run on one system compared to another system. Usually, speedup describes how an enhancement affects a system’s performance. The equation for speedup is

\[ S = \frac{\text{old}}{\text{new}} \]

where \( S \) is speedup, \text{old} is the run time of the unenhanced system, and \text{new} is the run time of the enhanced system [3].

**Throughput** is the measure of how many things can be processed per unit of time. In our tests, we calculate throughput as the number of bytes of input divided by how long the workload took to process that input.

In our measurements and discussion, we use “MiB” (mebibyte) instead of “MB” (megabyte) to unambiguously refer to \( 2^{20} \) bytes. The same rule applies to other byte units. Though we recognize that the convention is to refer to MB as \( 2^{20} \) bytes unless otherwise noted, we do this to be as clear as possible.

2.9.2 Predictability Metrics

The key metrics for real-time systems are latency and response-time jitter [4]. For latency, the most important property is the worst-case execution time (WCET), which is the longest exe-
cution time possible for a task. However, when the state space of a task is large, it is difficult to determine the WCET. So we use the maximal observed execution time (MOET) instead, recognizing that it will be an underestimate of the true WCET [8]. The MOET is simply the largest execution time observed for a set of test runs with a given configuration.

*Jitter* is the maximum variation of a time quantity, and *response-time jitter* is the jitter of the response time of a system [4] [9]. Jitter is also hard to calculate perfectly without running exhaustive tests on the entire state space, so our jitter values are estimates. We calculate the response-time jitter as the range of latency values, where the range is the MOET minus the minimal observed execution time for a set of test runs with a given configuration.

### 2.10 Cache Interference Mitigation

Since the L3 cache is shared and inclusive, programs on sibling cores can interfere with each other. There are a few techniques that have been proposed to mitigate this interference.

*Cache partitioning* restricts cache line allocation for tasks or cores to a subset of the cache. This can be done via platform-specific hardware mechanisms (for systems that support it) or through software-level approaches [10].

*Cache coloring* is a special case of cache partitioning usually enforced on the physical memory allocator of the OS or hypervisor. Cache coloring assigns a number called a *color* to a page of memory. A page of one color cannot evict the cache lines of a page of a different color [10].

*Cache locking* is a hardware mechanism that “locks” a cache line in place (i.e., prevents it from being evicted) [10]. Intel Cache Allocation Technology is one example of this [11] [1]. Another example is the explicit [12] and implicit [13] cache lockdown of ARM processors. Cache locking is one approach to simplifying WCET estimation and enhancing predictability [14].

*Core throttling* is where the execution of one core is suppressed to reduce the primarily cache-based interference felt by the other cores. Generally, if the number of executed instructions on a core per unit of time is reduced, then any cache thrashing will occur at a slower rate, reducing interference with other cores.

We investigate core throttling in Chapter 4, and do so in part because there has been a lack of research on this mitigation technique. As for comparing core throttling to other mitigation techniques, we leave that for future work. However, we will mention that core throttling can
potentially reduce interference from shared resources besides just caches, which sets it apart from other mitigation techniques.

2.11 Intel C-states and P-states

*C-states* are the low-power idle states for cores in Intel processors. Too frequent C-state transitions will lead to energy inefficiency (meaning that code runs at a higher average power than it could if superfluous C-state transitions were avoided) [15].

In contrast to C-states, *P-states* are the power states of Intel processors *during execution*. Each P-state is a discrete voltage and frequency combination. P-states are Intel’s implementation of *dynamic voltage-frequency scaling* (DVFS), which modern processors use to reduce power and energy usage during periods of low activity [3].

2.12 Hardware P-states

The OS normally coordinates with the processor when setting the current P-state. However, recent Intel chips have introduced a feature called *hardware P-states* in which the processor hardware automatically changes P-states without OS intervention.

With hardware P-states enabled, we found that the processor would set the same P-state for each core according to the average usage across *all* cores, rather than setting individualized P-states for each core. This caused unanticipated behavior where an isolated workload on one core would actually run slower when all the other cores were idle. Because of this coupling, we disabled hardware P-states in our test system so that P-state changes would be exclusively under OS control.

2.13 Time Stamp Counter

The mechanism used for timing the workloads in this thesis is the *Time Stamp Counter* (TSC) found on Intel x86 processors starting with the Pentium processor [11]. The TSC is a hardware counter provided by the processor for time keeping.

Originally, the TSC recorded the number of processor instruction cycles since system reset as a way to measure time. However, once processors began dynamically changing frequencies and
sleep states were introduced, a TSC cycle could no longer represent a fixed amount of time. The solution in newer processors is an enhancement called the invariant TSC. The invariant TSC runs at a constant rate, regardless of changes in P-states or C-states [11]. Thus, the TSC frequency never changes and is a dependable source of time keeping.

To find the actual real time elapsed, we used the following equation:

\[ t = \frac{tsc\_cycles}{tsc\_freq} \]

where \( t \) is time in seconds, \( tsc\_cycles \) is the number of TSC cycles during the time period recorded, and \( tsc\_freq \) is the TSC frequency in cycles per second. For our system, the TSC frequency is 3.7 GHz, which is also the base processor frequency.

2.14 Intel VMX

Jailhouse uses Intel’s virtual-machine extensions (VMX) extensively. VMX is a set of additions to the x86-64 ISA that adds hardware-assisted virtualization support by extending the x86-64 instruction set. Most modern Intel processors support VMX. Jailhouse also supports AMD’s analogous AMD Virtualization (AMD-V) technology, but that was irrelevant as we used Intel processors exclusively.

VMX provides many useful features to hypervisors, including native support for VM entries and exits. A VM exit is when an event happens that causes the hypervisor to take control from the guest VM, and a VM entry is when the hypervisor relinquishes control back to the guest VM. When a hypervisor uses VMX, it can configure which events will force a VM exit in the guest VM. For example, accessing an IO device commonly forces a VM exit.

VMX specifies two run-time execution modes: host mode and guest mode. Guest mode simply means that execution is happening in the child or “guest” VM as normal, and no VM exit has occurred. Host mode means that a VM exit occurred and the hypervisor is now in control, with full unvirtualized access to hardware resources. Host mode and guest mode are analogous to kernel space and user space in Linux, respectively, and a VM exit is analogous to an interrupt or system call that forces Linux to change from user space to kernel space.
Another useful feature VMX provides is the preemption timer. The preemption timer operates while the processor is not idle and counts down at a rate proportional to the TSC. Once it counts down to zero, it triggers a VM exit. This is similar to how an OS implements time-slicing, where processes are periodically put on hold to let others take their turn to execute. The preemption timer is key in the implementation of our core throttling interference mitigation technique, as it gives us a deterministic opportunity to stop a VM from executing further instructions [16].

2.15 Related Work

2.15.1 Work related to Jailhouse

Ramsauer et al. (which include the primary developers of Jailhouse) introduce Jailhouse and highlight its unique benefits [17]. Ramsauer et al. also demonstrate a mixed criticality system with Jailhouse where only 800 ns of additional latency per GPIO IRQ dispatch is attributable to the hypervisor on average compared to a bare-metal application, with a max of 2.88 \( \mu \)s. They further show that memory mapping all IO to the same page causes contention issues that reduce performance [18].

Kloda et al. discuss techniques to guarantee determinism in multi-core mixed criticality systems with a shared LLC. They implement cache coloring in Jailhouse, discuss various side effects of cache coloring, and introduce a novel technique to control cache content in spite of a random replacement policy [10].

Danielsson et al. determine how to measure the level of isolation of a processor’s cores, cache, and memory bus on a system with Jailhouse and show that Jailhouse exhibits good isolation regarding cores, but not with cache or memory bus interference [19].

Yang et al. implement a mixed criticality system with Jailhouse and a POSIX-compatible RTOS that demonstrates superior real-time performance compared to solutions like a PREEMPT-RT-patched Linux or Xenomai 3 [20].

Baryshnikov ports an x86 RTOS to Jailhouse and explains the architecture of Jailhouse [21]. Ramos implements an x86 Jailhouses system in QEMU and explores the IVSHMEM mechanism [22]. Corbet mentions throttling as a possible technique to reduce inter-VM interference in a LWN.net article discussing Linux, Jailhouse, and mixed criticality systems [23].
Toumassian et al. compare the Jailhouse and Xen hypervisors and find that Jailhouse has less overhead [24].

Li et al. present ACRN, a hypervisor tailored to IoT use cases that serves as an alternative to Jailhouse. Their direct comparisons show that ACRN has slightly better latency jitter than Jailhouse, though ACRN is currently limited to x86, while Jailhouse supports both x86 and ARM [25].

2.15.2 Other related work

West et al. present the Quest V separation kernel, a multikernel that achieves resource partitioning and performance isolation for subsystem components. Quest V is not a hypervisor, however [26].

Manco et al. discuss using lightweight VMs as containers to achieve better security and performance [27].

Agache et al. present Firecracker, Amazon’s recently open-sourced hypervisor combining the security of a hypervisor with the low overhead of containers for server systems [28].

Gracioli et al. discuss cache management schemes to avoid inter-core interference in real-time systems [29].

Esposito et al. discuss consolidating separate real time and non-real time systems onto the same multi-core processor and discuss applications to triple modular redundancy (TMR) [30].

Intel Corporation describes how its Cache Allocation Technology (CAT) could be used to improve the performance of a real-time application in a whitepaper [1].

2.15.3 Contributions

To our knowledge, there has been little research evaluating cache-based inter-core interference on a Jailhouse system, and no other works have investigated using a throttling technique to mitigate this interference.
CHAPTER 3. METHODS AND IMPLEMENTATION

3.1 Overview

In this thesis, we implement a mixed criticality system on an Intel x86-64 processor with the Jailhouse hypervisor. We then modify Jailhouse so it can throttle the execution of the root cell. Finally, we run various tests to see how effective that throttling is in shielding the inmate (real-time VM/RTOS) from interference from the root cell (non-real time VM/Linux). Figure 3.1 shows a simplified diagram of our experimental setup with workloads and core throttling.

Figure 3.1: Experimental Setup with Jailhouse, Workloads, and Core Throttling
In our tests, we run workloads on the root cell and inmate concurrently and measure the change in inmate workload execution time when we throttle the root cell. With that, we determine the speedup due to throttling. We also use execution time and input size to calculate the change in throughput due to throttling. In addition, we calculate the MOET difference and response-time jitter difference between unthrottled and throttled to see if throttling improves predictability. The inmate in our setup only has one core, and each inmate workload is single threaded.

3.2 Workloads

We categorize the workloads used into two different types: inmate workloads and interference workloads.

3.2.1 Inmate Workload

We define the *inmate workload* as the workload running on the inmate, representing a real-time task. The inmate workload operates on input either received from the root cell or locally generated by the inmate itself. It then sends the output to the root cell. When we record the execution time of the inmate workload, we are careful to exclude any overhead not directly associated with the workload function itself, like obtaining input or returning output. We do this by starting and stopping the TSC-based timer as close in the code as possible to the call of the function containing the inmate workload code.

3.2.2 Interference Workload

While the inmate is running its workload, something else entirely could be running on the root cell. We define the *interference workload* as the primary application running on the root cell while the inmate is executing its workload. The purpose of the interference workload is to cause a measurable impact on the inmate workload for our tests. This allows us to see how much core throttling protects the inmate from interference coming from the root cell.
3.2.3 Workload Types

We have created three workloads – SHA3, CSB, and RA – for use as both inmate workloads and interference workloads. An additional fourth workload – Handbrake – is used as an interference workload only.

The first workload we call **SHA3**. It calculates the Secure Hash Algorithm 3 (SHA-3) cryptographic hash of an input. The code we used is an unoptimized, open-source implementation of SHA-3 intended for academic rather than production use, available at https://github.com/mjosaarinen/tiny_sha3.

The second workload we call **Count Set Bits**, or **CSB**. It counts the total number of ones (asserted bits) of an input using a lookup table that returns a pre-calculated number for each byte. The number of ones in a piece of binary data is also known as the Hamming weight.

The third workload, named **Random Access** or **RA** for short, approximates a random-access workload, but only when the input itself is random. This is because it derives its random memory access pattern from the input. Unless otherwise specified, we always provide random input to RA.

The fourth workload encodes a video via **Handbrake**, an open source video transcoder. Video encoding is CPU intensive, spans all cores, and works on a large data set. We chose a large .m2ts (Blu-ray) movie file as the video to ensure that the encoding doesn’t prematurely finish during our tests. The same video file is transcoded with the same encoding preset each time.

To reduce variations due to compiler options, we make sure that the inmate workload and interference workload wrapper programs use the exact same compiled object files for the workload code.

Handbrake is designed to run on all available cores. In contrast, the SHA3, CSB, and RA workloads are all single threaded. So when we use these latter workloads as the interference workload, we run one workload instance for each core in the root cell, plus an extra instance for good measure. This ensures that all cores are busy during execution.

We originally chose to use the SHA3 workload exclusively, as it exemplifies a real-world application. However, since the SHA3 algorithm was designed to be cache friendly, we needed other workloads with poorer locality to induce more interference. So we created the CSB workload as a plausible real-world application with less cache-friendly characteristics than SHA3. We then
created the RA workload as a kind of worst-case scenario to induce as much interference as we could. The Handbrake interference workload was also chosen as a real-world application that turned out to be more cache friendly than expected, though not quite as much as SHA3.

We expect that the majority of real-world workloads will look more like SHA3, CSB, and Handbrake, with temporary random-access spikes more in line with RA. The sustained random memory access pattern exhibited by RA is most likely unrealistic.

### 3.3 Throttling

The basic idea behind core throttling is that if we can artificially restrict how often a core can execute instructions, then it will naturally reduce contention for any shared resources it accesses. Thus, the inmate should run faster on a wide variety of workloads experiencing a wide variety of interference vectors. Some inherently shared resources between the root cell and inmate include last-level caches and the memory bus [19].

In order to investigate whether throttling the root cell could help improve the inmate’s performance, we developed and built a novel throttling capability into Jailhouse. The high-level implementation of the throttling mechanism is as follows: Jailhouse takes control every millisecond and checks if throttling is requested. If so, a delay is inserted before returning execution back to the VM. This delay is created by executing the x86-64 `PAUSE` instruction a configurable number of times. We term this number “spin loop iterations.” For further explanation on how we implemented throttling in Jailhouse, see Appendix B.2.

### 3.4 Inter-VM Communication

In order for the root cell and inmate to communicate and pass data to each other, we use a shared memory channel called IVSHMEM (short for Inter-VM SHared MEMory). This channel lives in Jailhouse’s dedicated memory region and is reserved for use by both the root cell and the inmate cell. In the root cell, Linux can memory map this shared memory channel to easily send data to the inmate through simple userspace programs. On the inmate side, we memory map the shared memory channel using the Jailhouse-provided API. Jailhouse implements IVSHMEM as an emulated PCI hardware device that both VMs can access.
While Jailhouse provides a communication channel, it does not provide any synchronization or coordination of that channel. We developed a simple handshake protocol to coordinate communication between the root and the inmate. This allows the channel to be shared properly and avoids collisions when data are sent and received. For more technical details on the implementation, see Appendix B.4.

3.5 Jailhouse Memory Layout

![Memory Layout and Hardware Configuration](image)

Figure 3.2: Memory Layout and Hardware Configuration

A simplified memory layout and hardware configuration is shown in Figure 3.2. The figure depicts two VMs: the root cell (Linux VM) and the inmate (RTOS VM). The VMs communicate to each other through the shared memory channel (IVSHMEM). Shared memory simply means that both VMs can write to a segment of their own guest physical memory that will map to the same segment of host physical memory. Since the guest physical address space for each VM is
completely separate from every other VM, this shared memory channel requires special support in
the hypervisor in order to work correctly.

When the hypervisor itself needs to communicate with the root cell or the inmate, however,
there is another memory channel used as a special Jailhouse communication region. When the
root is in host mode, it can communicate to the inmate through this channel. This communication
region is maintained by Jailhouse in its hypervisor memory region and is exposed to the VMs at a
certain guest physical address. We use this channel to communicate the throttle commands from
the inmate to the hypervisor in order to initiate throttling in the root cell.
CHAPTER 4. EXPERIMENTS AND RESULTS

We will now discuss the experiments conducted to show how well isolation is achieved between the root cell and inmate and the extent to which core throttling of the root cell improves the performance and predictability of the inmate.

4.1 Experimental Setup

4.1.1 Hardware

Experiments were done on a desktop computer system with a 6-core Intel® Core™ i7-8700K processor with 15.5 GiB of 2666 MHz DDR4 DRAM. An NVIDIA GeForce GTX 1650 GPU was also present on the system. While the graphical desktop environment took advantage of the GPU, none of the inmate workloads or interference workloads offload any computation to the GPU.

Notable processor features include: an inclusive L3 cache; Hyper-threading; VMX (which Jailhouse and our throttling infrastructure rely on heavily); Hardware P-States (HWP); and an invariant TSC. All of these are described in Chapter 2.

We made sure to turn off Hyper-threading, so that there was only one thread per core. Internal core resource contention and other hardware-level coupling between sibling logical threads could affect the performance of the single-threaded inmate, so turning this off helped simplify the analysis.

We also disabled HWP for the system. We found that HWP was unaware of the core partitions made by Jailhouse, since HWP is a black-box processor mechanism where the algorithm is not controllable at the OS or hypervisor level. When enabled, it introduced extreme frequency coupling between the root cell’s cores and the inmate’s core. This made the inmate counter-intuitively run slower when the root cell was idle. We suspect that when HWP sees five of the six cores as
idle, it assumes that the whole processor is mostly idle. Thus, it scales down the frequency of all cores to match the mostly-idle state of the processor as a whole.

4.1.2 Software

For the root cell’s Linux OS, we used Kubuntu 19.10, a Linux distribution based on Ubuntu 19.10 with KDE Plasma as the graphical desktop environment. We installed Handbrake to act as one of the interference workloads and CoreFreq to monitor the processor frequency and query processor features.

For the hypervisor, we used a fork of Jailhouse based on version 0.11 with additional changes (see https://github.com/hintron/jailhouse). The main difference between vanilla Jailhouse and our version is that we added the ability for the hypervisor to throttle the root cell at the request of the inmate. For details on how to get our modified version of Jailhouse running, see Appendix A. For details on the modifications made to the hypervisor to add throttling, our workloads, and our test setup, see Appendix B.

4.1.3 Workloads Overview

The inmate workloads were SHA3, Count Set Bits (CSB), and Random Access (RA). The interference workloads were Handbrake, SHA3, CSB, RA, and None. These workloads are described in Section 3.2.

Before discussing the throttling tests, we first look at how each workload performed without interference to establish baseline performance. Figure 4.1 compares the run times of the SHA3, CSB, and RA workloads under various nominal conditions. All runs use a 20 MiB input. Each value is an average of values from ten successive runs, all without throttling enabled.

The inputs for “Inmate Local Uniform” (blue) and “Inmate Local Random” (teal) were both generated locally in the inmate and were not received from the root cell over the IVSHMEM shared memory channel. The former consisted of a single repeated byte value, while the latter was a pseudo-random input. The difference between the uniform and random inputs only manifested in RA, since the access pattern for uniform inputs was much more cache friendly.
“Inmate Local Random” (teal) and “Inmate IVSHMEM” (green) both used random inputs, but the first used inputs residing in memory only accessible to the inmate, while the latter used inputs sent to it from the root cell over the IVSHMEM shared memory channel. There was not much of a difference in execution time other than the RA workload actually being a bit faster with IVSHMEM. This shows that the performance penalty for passing inputs over IVSHMEM is minor, at least without an interference workload.

“Linux” (orange) and “Linux Jailhouse” (purple) show the execution time of each workload in Linux (i.e., not in the inmate). The first is running the workload in vanilla Linux, while the second is running the workload in Linux under Jailhouse. Under Jailhouse, the workload is slower. This is because even a low-overhead hypervisor like Jailhouse has some run-time overhead due to the few required VM exits.

This also shows that for the most part, the inmate workloads (blue, teal, and green) run faster than even vanilla Linux (orange). This is expected, because the inmate does not have OS-
related overhead. Among other things, Linux likely disrupts the workload every few milliseconds to check up on other processes, causing context switching which slows down the workload process.

4.2 Experiments Overview

Our throttle experiments fall under two categories: input response and throttle response. For the input response, we ran tests across a range of input sizes with different workloads using the same throttle parameters. For the throttle response, we ran tests across a range of throttle durations with different workloads using the same input. (These are akin to a frequency response graph, where gain is graphed across a range of frequencies.)

For both the input response and throttle response, we measured the execution time of the inmate workload for unthrottled and throttled cases at each configuration. From these data we derived the speedup from unthrottled to throttled to show the relative performance change. We also derived throughput for both unthrottled and throttled workloads to highlight the absolute performance change. For predictability metrics, we derived the maximal observed execution time (MOET) as an estimate of the worst-case execution time (WCET) and graphed the differences between the unthrottled and throttled cases. We then derived response-time jitter and graphed the differences between the unthrottled and throttled cases. Additionally, for the input response, we showed the average speedup over all input sizes to highlight which workloads caused the most interference on average.

4.3 Input Response Results

Our input response experiments tested throttling for each combination of interference workload and inmate workload, resulting in 15 combinations. Table 4.1 lists each of these combinations, along with average speedup measurements discussed later in this section.

For each combination, the inmate workloads ran with random inputs ranging in size from 1 MiB to 39 MiB in 1 MiB steps, with 10 experiment runs at each step. Each run was given a unique random data input at the current input size. The first five runs were with the root cell unthrottled, while the second five runs were with the root cell throttled. Each set of five runs was averaged to produce a single execution time value.
When the interference workload was either SHA3, CSB, or RA, six instances of the workload ran independently in parallel across the five cores available to the root cell, ensuring no idle cores. A 1 GiB random input file was given to each thread to ensure that the working data set could not fit in the L3 cache. However, there was always enough DRAM available to prevent the root cell from running out of main memory. Once the last of the six workload threads completed, they were all restarted again with the same input file. This was repeated for the duration of the experiment. The interference workloads were always given 30 seconds to ramp up before each experiment started, which was plenty of time to ensure maximum interference.

4.3.1 Speedup

The graphs in Figure 4.2, Figure 4.3, and Figure 4.4 show the speedup in execution time from the unthrottled runs to the throttled runs at each input size for each workload. These graphs show that inmate workloads running concurrently with an RA interference workload in the root
Figure 4.3: Input Response - Speedup - CSB Inmate Workload

Figure 4.4: Input Response - Speedup - RA Inmate Workload
cell experienced the largest speedup due to throttling. Since the RA interference workload caused the most interference, throttling it brought about the biggest relative improvement in performance.

The SHA3 inmate workload was hardly affected by the interference workloads, as can be seen in Figure 4.2. Even the RA interference workload did not induce much interference in SHA3, as throttling it produced a marginal speedup of only about 1.01 on average. The marginal improvements from throttling indicates the limited interference experienced in the first place.

The CSB inmate workload in Figure 4.3 is a little more interesting. Throttling the RA interference workload leads to a more significant speedup than it did in the SHA3 inmate workload, with an average speedup of about 1.06.

Finally, the RA inmate workload in Figure 4.4 shows the most disturbance from interference workloads. The speedup of the RA inmate workload with an RA interference workload is initially very high at around 3 or 4, then drops to about 2. Once the input size exceeded 12 MiB, there was likely cache thrashing in the inmate since the input could not all fit in the 12 MiB L3 cache at the same time. So although throttling still produced a speedup of about 2, there was not as much potential for speedup as when the input was less than 12 MiB and could fit completely in the L3 cache.

### 4.3.2 Average Speedup

The graphs in Figure 4.5, Figure 4.6, and Figure 4.7 show the inmate workload speedup from unthrottled to throttled, averaged over all input sizes for each interference workload. On average, the RA interference workload was significantly more disruptive than the other interference workloads for all inmate workloads. The SHA3 interference workload had very little impact on all inmate workloads. The “None” workload is simply a control where we had no interference workload running but still throttled the root cell anyway.

Similar to the average speedup graphs already seen, Table 4.1 shows the average speedup over all input sizes for each interference workload and inmate workload combination, in tabular form. What stands out in this table is that the combination of an RA interference workload and an RA inmate workload produced the greatest speedup results with throttling. Having an RA inmate workload run alongside a Handbrake or CSB interference workload also showed significant
Figure 4.5: Input Response - Average Speedup - SHA3 Inmate Workload

Figure 4.6: Input Response - Average Speedup - CSB Inmate Workload
Figure 4.7: Input Response - Average Speedup - RA Inmate Workload

<table>
<thead>
<tr>
<th>Inmate Workload</th>
<th>Interference Workload</th>
<th>Average Speedup Over All Input Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA3</td>
<td>Handbrake</td>
<td>1.001</td>
</tr>
<tr>
<td>SHA3</td>
<td>SHA3</td>
<td>1.000</td>
</tr>
<tr>
<td>SHA3</td>
<td>CSB</td>
<td>1.002</td>
</tr>
<tr>
<td>SHA3</td>
<td>RA</td>
<td>1.010</td>
</tr>
<tr>
<td>SHA3</td>
<td>None</td>
<td>1.000</td>
</tr>
<tr>
<td>CSB</td>
<td>Handbrake</td>
<td>1.003</td>
</tr>
<tr>
<td>CSB</td>
<td>SHA3</td>
<td>1.001</td>
</tr>
<tr>
<td>CSB</td>
<td>CSB</td>
<td>1.004</td>
</tr>
<tr>
<td>CSB</td>
<td>RA</td>
<td>1.056</td>
</tr>
<tr>
<td>CSB</td>
<td>None</td>
<td>1.000</td>
</tr>
<tr>
<td>RA</td>
<td>Handbrake</td>
<td>1.306</td>
</tr>
<tr>
<td>RA</td>
<td>SHA3</td>
<td>1.018</td>
</tr>
<tr>
<td>RA</td>
<td>CSB</td>
<td>1.450</td>
</tr>
<tr>
<td>RA</td>
<td>RA</td>
<td>2.354</td>
</tr>
<tr>
<td>RA</td>
<td>None</td>
<td>0.997</td>
</tr>
</tbody>
</table>

Table 4.1: Input Response - Average Speedup - All Combinations
speedup results. The SHA3 interference workload did not interfere much with non-RA inmate workloads.

### 4.3.3 Throughput

Figure 4.8, Figure 4.9, and Figure 4.10 show throughput graphs for the SHA3, CSB, and RA inmate workloads for both unthrottled and throttled experiments. Each line represents the throughput of the inmate workload with an interference workload running on the root cell with either throttling applied or not applied (unthrottled). Solid lines represent the throttled case and dashed lines represent the unthrottled case. The spikes and dips shown in these graphs directly correlate to the speedup graphs earlier in Section 4.3.1, since they both are derived from the same data.

![Throughput Graph](image)

**Figure 4.8: Input Response - Throughput - SHA3 Inmate Workload**

These throughput graphs show that the throttled cases match the “None - Unthrottled” case closely, which is the case where no throttling is applied and the root cell is idle (i.e., no interference...
Figure 4.9: Input Response - Throughput - CSB Inmate Workload

Figure 4.10: Input Response - Throughput - RA Inmate Workload
workload). The throughput graphs further show that the RA interference workload was by far the most disruptive to each inmate workload. When the RA interference workload was left unthrottled, throughput dropped drastically in comparison to the other interference workloads. The second- and third-most disruptive interference workloads were CSB and Handbrake, respectively. SHA3 was not very disruptive at all.

For all interference workloads, the inmate workload throughput remained stable across the entire input range for SHA3 and CSB. However, for RA in Figure 4.10, the throughput dropped significantly as the input increased in size. As the L3 cache is 12 MiB, it was expected that a random access workload would cause issues near the 12 MiB input size due to cache thrashing. Since the RA interference workload ran with one more thread than available cores, there was likely some OS context switching during execution. This means that with a 2 MiB input, there would be 12 MiB of L3 cache being used by those six threads. This is likely why there is an immediate precipitous drop in throughput for the “RA + Unthrottled” green dashed line starting at about 2 MiB.

Both the speedup and throughput graphs show that the maximum inter-VM interference occurred when both the inmate workload and the interference workload were RA. This makes sense because a truly random memory access pattern would cause the most cache thrashing due to low spatial and temporal locality.

4.3.4 Maximal Observed Execution Time

Figure 4.11, Figure 4.12, and Figure 4.13 show the difference in MOET between unthrottled and throttled SHA3, CSB, and RA inmate workloads, respectively. A positive difference means that the MOET is greater for the unthrottled case than for the throttled case. These data are derived from the same data used for the prior speedup and throughput graphs, meaning the MOET is the maximum of a set of five duration measurements each for throttled and unthrottled.

These graphs show that throttling reduces the MOET in most cases for the SHA3, CSB, and RA inmate workloads. The RA and CSB workloads consistently show the largest differences in MOET between unthrottled and throttled.
Figure 4.11: Input Response - MOET Difference - SHA3 Inmate Workload

Figure 4.12: Input Response - MOET Difference - CSB Inmate Workload
Figure 4.13: Input Response - MOET Difference - RA Inmate Workload

Figure 4.14: Input Response - Jitter Difference - SHA3 Inmate Workload
Figure 4.15: Input Response - Jitter Difference - CSB Inmate Workload

Figure 4.16: Input Response - Jitter Difference - RA Inmate Workload
4.3.5 Response-Time Jitter

Similar to the MOET graphs, Figure 4.14, Figure 4.15, and Figure 4.16 show the difference in response-time jitter for each of the SHA3, CSB, and RA inmate workloads running alongside various interference workloads. Response-time jitter is a measure of variability in workload execution and in part describes how predictable a workload is. Like MOET, these data are also derived from the same data used for the speedup and throughput graphs.

These graphs show that the RA and CSB workloads have the largest variation in execution times. Interestingly, the RA inmate workload with the CSB interference workload in Figure 4.16 shows consistently more jitter than even an RA interference workload, which is a slight departure from other results in this section.

4.4 Throttle Response Results

Our next set of experiments (the “throttle response”) show how a workload responds to shorter or longer throttling durations.

Figure 4.17, Figure 4.18, and Figure 4.19 show the speedup graphs of the throttle response of each inmate workload over a range of spin loop iterations with throttling applied every millisecond. The inmate workload is repeatedly given the same 20 MiB random input passed over IVSHMEM, and the interference workload is also RA. These speedup graphs have a logarithmic horizontal axis.

Figure 4.20, Figure 4.21, and Figure 4.22 show the corresponding throughput graphs of the throttle response of the inmate workloads. They are derived from the same data as the speedup throttle response graphs. However, the horizontal axis is linear rather than logarithmic to better highlight the diminishing returns as the number of spin loop iterations increases. Though the number of spin loop iterations should be irrelevant to the unthrottled series, it acts as a control to show the throughput gains.

We note that the default spin loop iterations value used for throttling in prior input response graphs in Section 4.3 was 50,000. This value likely achieves the vast majority of potential throughput and speedup gains, as shown by Figures 4.17 through 4.22.
Figure 4.17: Throttle Response - Speedup - SHA3 Inmate Workload

Figure 4.18: Throttle Response - Speedup - CSB Inmate Workload
Figure 4.19: Throttle Response - Speedup - RA Inmate Workload

Figure 4.20: Throttle Response - Throughput - SHA3 Inmate Workload
Figure 4.21: Throttle Response - Throughput - CSB Inmate Workload

Figure 4.22: Throttle Response - Throughput - RA Inmate Workload
4.5 Observations

The input response speedup and throughput graphs (Figures 4.2 - 4.10) show that nearly every combination of inmate workload and interference workload at every input size exhibited an inmate speedup when the root cell was throttled. This included real-world workloads like SHA3 and Handbrake as well as more contrived workloads like CSB and RA. Even so, these graphs show that the speedup due to throttling was relatively insignificant in most cases not involving an RA inmate or interference workload. This can be clearly seen in table 4.1.

Similarly, we can infer that the inter-VM interference was highest when both workloads exhibited a random memory access pattern. This is because a random memory access pattern has extremely poor locality, leading to interference due to frequent cache line evictions. For the most part, the throughput graphs also show that when throttling was applied, inmate performance was restored to match what it would have been in the absence of an interference workload.

In addition, these graphs show that throttling did not have significant overhead for various combinations of interference and inmate workloads, even when performance gains were insignificant. This makes it relatively safe to use throttling with disparate workloads, as there is no major performance penalty for using throttling when it is unnecessary.

The MOET and jitter graphs (Figures 4.11 - 4.16) show that throttling not only can improve performance, but can improve predictability as well. Both MOET and jitter were improved in many cases, indicating that throttling can have a stabilizing effect on predictability in the face of interference.

The throttle response speedup and throughput graphs (Figures 4.17 - 4.22) show that the default throttle settings used for the prior input response graphs were sufficiently well configured, as most of the speedup for each inmate workload occurred with less than the default of 50,000 spin loop iterations. The throughput graphs also show diminishing returns – the performance gains were logarithmic as the number of spin loop iterations increased.
CHAPTER 5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

We have shown that for real-world workloads like SHA3 and Handbrake, Jailhouse provides high levels of isolation out of the box. We have also shown that there is significant potential for interference when the inmate and the root cell are both running workloads with random memory access patterns. In the configuration with the highest interference, we have shown that core throttling the root cell effectively mitigates interference so that the inmate runs 2-5x faster than it would have without throttling. The performance overhead of throttling in the inmate was undetectable, even in cases where throttling the root cell was superfluous.

Additionally, we have shown that throttling can reduce the MOET and the observed response-time jitter. From this, we infer that throttling can improve the WCET and worst case response-time jitter, improving predictability.

5.2 Contributions

We have modified the Jailhouse hypervisor to implement core throttling, something that other works to our knowledge have not yet investigated. We have then built a system representing a mixed criticality system on an Intel x86-64 processor with Jailhouse. Finally, we conducted experiments showing that core throttling can be an effective tool in protecting a workload from sibling VM interference.

5.3 Future Work

While this thesis looked at interference due to cache contention, core throttling may also be effective at reducing other sources of inter-core interference, including temperature-related coupling, internal processor resource contention, IO or memory bus contention, network adapter band-
width contention, and IO device contention, among other sources. Low-hanging fruit in this area include looking at memory bus contention as programs fetch data from DRAM, as investigated by Danielsson et al. [19].

The system we built was developed and tested on Intel x86-64 architectures due to availability and ease of implementation. However, the principles and techniques discussed can be adapted to work on systems with other computer architectures, including low-powered ARM devices.

Our tests throttled the inmate in predetermined intervals. However, one application of throttling could be dynamic throttling, where Jailhouse only throttles the root cell when a slowdown puts the inmate in danger of missing a deadline.

Other possible future work includes implementing a mixed criticality system and throttling techniques with other hypervisor platforms, including Xen, KVM, and Firecracker; porting Jailhouse modifications to work not just with Intel’s VMX, but also with AMD’s SVM and with ARM CPUs; enhancing and developing core throttling techniques; investigating the usability of the VM while it is being throttled; and comparing core throttling with other cache interference mitigation techniques like cache coloring, cache locking, and cache partitioning.
REFERENCES


APPENDIX A. CONFIGURING AND RUNNING JAILHOUSE

The following is a discussion of various ad hoc topics when configuring and running Jailhouse. We use our fork of Jailhouse available at https://github.com/hintron/jailhouse on the mgh branch. The current working directory for all commands in this section is assumed to be the source folder of Jailhouse.

A.1 Enabling Jailhouse during Linux Bootup

We are required to reserve a memory region from Linux for Jailhouse to operate, or else Linux could inadvertently trample over Jailhouse. So when we boot up Linux initially, we need to tell it via the kernel command line parameters that some memory is off limits. This can be accomplished by using the `memmap` kernel command line parameter as the Linux kernel is initially loaded by the GRUB bootloader. In `/etc/default/grub` we add

```plaintext
memmap=82M\$0x3a000000
```

to `GRUB_CMDLINE_LINUX_DEFAULT`. This tells Linux to reserve 82 MiB of memory starting at physical address `0x3a000000`.

Jailhouse also requires passing in `intel_iommu=off` to the Linux kernel in order to disable the VT-d IOMMU usage (DMAR).

Setting `kvm_intel.nested=1` allows Jailhouse itself to run nested within Linux KVM (Kernel Virtual Machine), another hypervisor built into the Linux kernel. This is optional, but allows running Jailhouse within QEMU, a machine emulator and virtualizer that takes advantage of KVM under the hood.

Linux can also configure some aspects of the processor through drivers (otherwise known as kernel modules) in Linux. Setting `intel_pstate=no_hwp` tells the `intel_pstate` kernel module to disable the `hardware p-states` (HWP) mechanism of the Intel processor on bootup.

Adding all these to some existing parameters already specified, we get:
A.2 Configuring the Root Cell and Inmate

Before running a cell under Jailhouse, configuration files are required. These files tell Jailhouse which hardware devices and memory regions are accessible from which cells. If a cell tries to access a hardware device or memory region that is not configured, Jailhouse will reject that access.

On x86 machines, Jailhouse comes with a convenient configuration generator tool. This tool automatically detects the hardware and devices on the system and generates a base configuration file. While example configuration files for various ARM and x86 root cells and inmates come with the source code, this autogenerated file is a great starting point.

With the following command, an output configuration file is generated:

```
sudo ./tools/jailhouse config create --console ttyS4 <output_filename>
```

This generated file serves as the basis for the root cell configuration. We made minor adjustments to this autogenerated file as needed, and it exists at `configs/x86/bazooka-root.c`.

The critical adjustment we made is to add a virtual PCI device and memory region for the IVSHMEM shared memory channel to enable inter-cell communication. We based the IVSHMEM PCI device and memory region off of the `configs/x86/x86-qemu.c` example root config for the QEMU Jailhouse example.

```
--console ttyS4
```

is needed to configure Jailhouse to print debug output to a hardware serial port interface while the inmate is running. It needs to be a hardware serial port monitored by an external computer, because if the debug output was simply sent to a file or terminal screen and the host computer crashes, the recent console debug output can get lost and it becomes very difficult to debug the source of the crash.
To create our inmate’s configuration file, we created a copy of `configs/x86/ivshmem-demo.c`, modified it, and saved it to `configs/x86/bazooka-inmate.c`. We changed the inmate name, shifted the inmate’s memory region to the start of the global inmate memory region, expanded the IVSHMEM memory region from 4 KB to 40 MB, and added an additional 36 MB memory region that the inmate can use. We also changed some of the cell flags.

### A.3 Checking for Jailhouse Hardware Prerequisites

The following command allows you to check if your system has the hardware prerequisites needed to run Jailhouse:

```
./tools/jailhouse hardware check /etc/jailhouse/qemu-x86.cell
```

### A.4 Compiling Cell Configuration Files

Jailhouse automatically compiles each cell configuration file under the `./configs` directory into a corresponding `.cell` binary file. Simply calling `make` again will do this.

### A.5 Starting the Root Cell

Using a user space tool provided by Jailhouse, we start the root cell by passing in the root cell configuration binary and running the following command:

```
sudo ./tools/jailhouse enable ./configs/x86/bazooka-root.cell
```

When the `jailhouse enable` command runs, the kernel module places Jailhouse ‘underneath’ the currently running Linux and effectively converts Linux into a child VM. From here on out, Jailhouse is running on bare metal hardware and all processor instructions executed by child VMs pass through it. Thus, Jailhouse is now acting as a full-fledged hypervisor, mediating between the system hardware and Linux.
A.6 Starting the Inmate

The inmate’s configuration is first loaded with the following command:

```
sudo ./tools/jailhouse cell create ./configs/x86/bazooka-inmate.cell
```

However, nothing is running yet in the newly-started cell. The C program that the inmate will run, which is in `./inmates/demos/x86/mgh-demo.c`, is compiled into a binary file, located at `./inmates/demos/x86/mgh-demo.bin`. We tell the inmate to load this file into the cell:

```
sudo ./tools/jailhouse cell load bazooka-inmate
      ./inmates/demos/x86/mgh-demo.bin
```

Now that the program is loaded and ready to run, we tell Jailhouse to start cell execution:

```
sudo ./tools/jailhouse cell start bazooka-inmate
```
APPENDIX B. JAILHOUSE MODIFICATIONS

The following is a discussion regarding debugging the Jailhouse hypervisor, the modifications we made to implement throttling, and how we used IVSHMEM to communicate between the root cell and the inmate.

B.1 Debugging the Hypervisor

Before we could effectively start modifying Jailhouse, we needed to access debug console output in order to debug the hypervisor. The easiest method was to access a virtual console via the `jailhouse console -f` command in the root cell Linux. This was how we collected our experiment output for Chapter 4.

Since we specified the `JAILHOUSE_CELL_VIRTUAL_CONSOLE_ACTIVE` flag in the inmate’s configuration file, the console debug output was written to both a hardware serial console and a buffer in Linux. That Linux buffer can be accessed through the following command:

```
sudo ./tools/jailhouse console
```

Or, the buffer can be dumped via sysfs like so:

```
cat /sys/devices/jailhouse/console
```

Unfortunately, there were drawbacks to this method. If our modifications ever made Jailhouse crash, there was little hope that useful debugging output would be accessible for long in a Linux terminal before the system itself stopped responding. And it was unlikely that the relevant debug information was ever saved to disk before the system crashed. Thus, to enable proper debugging, we needed to set up an external computer to monitor the serial port console in the event that Jailhouse crashed.
Since we were unable to get the built-in serial port that came with the system working, it was necessary to add a PCIe-to-serial port expansion card with a serial port-to-USB adapter. Once we installed the serial port hardware interface, a device file software interface was provided in Linux at `/dev/ttyS4`. Then, using a serial port-to-USB adapter cable and a small single-board computer (Raspberry Pi 2 Model B), we could monitor the serial port output of Jailhouse while the inmate was running.

We used the program `minicom` to both monitor Jailhouse with an external computer and to initialize the serial connection from the host. We found that we had to “prime the pump” with `minicom` from the host side before any console debug output from Jailhouse was properly written to `/dev/ttyS4`.

B.2 Throttling Mechanism

We considered two throttling mechanisms: frequency modulation and spin looping. Both are techniques to throttle individual cores rather than the processor as a whole. We eventually settled on using spin looping, as it was straightforward and effective.

B.2.1 Frequency Modulation

Frequency modulation attempts to throttle the core by reducing the clock frequency, which in turn reduces the rate at which instructions are executed. With fewer instructions being executed, the core will naturally use fewer resources than it would have, ideally reducing resource contention and improving inmate performance. Our initial implementation indicated that this was likely not an effective throttling method, so we abandoned this mechanism early on.

B.2.2 Spin Looping

Spin looping throttles the core by periodically inserting a delay before the core executes further instructions. We implemented this delay as a for loop that executes the x86-64 `PAUSE` instruction a certain number of times (spin loop iterations) while the hypervisor is in host mode. The number of spin loop iterations directly corresponds to the length of the delay.
In order to periodically enter host mode, we use the VMX preemption timer to force a VM exit every millisecond. This happens independently and concurrently on each core. Once the VM exit occurs and the core enters host mode, Jailhouse first checks whether the core belongs to the root cell or the inmate. If the core belongs to the root cell, it then checks to see if the inmate requested throttling. If it did, the core gets throttled. Once the throttling delay elapses, control is returned to the VM via a VM entry (i.e., the core returns to guest mode).

`PAUSE` gives the processor a hint that we are idling instead of doing real work, which allows it to avoid a performance penalty and greatly reduce power consumption [31] [32]. We implemented spin loop throttling by adding the following code to the preemption timer handler:

```c
if (spin_loop_throttle) {
    unsigned long count = 0;
    while (count++ < SPIN_LOOP_ITERATIONS)
        cpu_relax();
}
```

See `hypervisor/arch/x86/vmx.c` → `preemption_timer_handler_mgh()`. `cpu_relax()` is a function provided by Jailhouse that executes the `PAUSE` instruction:

```c
static inline void cpu_relax(void)
{
    asm volatile("rep; nop" : : : "memory");
}
```

See `hypervisor/arch/x86/include/asm.processor.h`.

The "rep; nop" and "pause" assembly constructs both use identical encodings (0xF3 90), so this is just a backwards-compatible way of writing the `PAUSE` instruction in case an older assembler does not recognize the `pause` keyword [33]. Though the `PAUSE` instruction was introduced in the Pentium 4, it is also backwards compatible for all IA-32 processors, where it operates as a `NOP` [31].
The time spent throttling in the spin loop is governed by two parameters: how frequently the preemption timer triggers (`PREEMPTION_TIMEOUT_DEFAULT`, empirically set to achieve a 1 ms period), and how many times the loop executes, or spin loop iterations (`SPIN_LOOP_ITERATIONS_DEFAULT`, currently set to 50,000). We found that the 1 ms preemption timer period combined with a 50,000 spin loop iteration count produced a very slow but usable user experience on a Linux graphical desktop environment.

We settled on a preemption timeout value of about once every millisecond. This seemed like a reasonable starting value, as it was in the ballpark of the 10 ms timeslice values found on many OSs [34] [32].

B.3 Inmate Program Code

The inmate program code was initially based off of `./inmates/demos/x86/apic-demo.c` and then was heavily modified to perform various workloads specifiable via the command line.

B.4 Communication Protocol Over IVSHMEM

While Jailhouse provides the IVSHMEM (Inter-VM SHared MEMory) mechanism for communication between cells, it does not provide any sort of synchronization or protocol on top of it. So we had to develop our own simple handshake protocol, which we describe in this section.

Jailhouse actually implements IVSHMEM as an emulated PCI device, which enables it to have interrupt capabilities in addition to the memory region. However, we don’t take advantage of interrupts and instead use polling to determine change in state, since it was easier to figure out and implement.

Here is how we organized the IVSHMEM shared memory region (i.e., the memory mapping):
where \( N \) (the total size of the IVSHMEM region) is currently set to 40 MiB. This means that the total possible data payload size is nearly 40 MiB (40 MiB minus the first 8 bytes).

Byte 0 is a synchronization byte used to coordinate access of the shared memory channel between the root and the inmate. The following table describes what each value means:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Tells the root that the inmate is not yet up and running.</td>
</tr>
<tr>
<td>1</td>
<td>Tells the root that the inmate is initialized and ready for the first workload from the root.</td>
</tr>
<tr>
<td>2</td>
<td>Tells the inmate that the root has placed input data into the shared memory channel.</td>
</tr>
<tr>
<td>3</td>
<td>Tells the root that the inmate is currently executing the workload.</td>
</tr>
<tr>
<td>4-255</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

After starting the root cell but before starting the inmate, it is necessary to clear the sync byte to 0 in case there is any leftover state from a previous run.

### B.5 How Jailhouse Uses the Preemption Timer

Before we could use the VMX preemption timer, we needed to understand how Jailhouse used the preemption timer in the first place. With this understanding, we were able to modify Jailhouse to use the preemption timer for our own purposes while preserving the original functionality. The following is a discussion of how Jailhouse natively uses the preemption timer. Figure B.1 shows the state diagram of how the preemption timer works alongside our enhancements.

Jailhouse tries really hard to avoid intervening between the child VMs and the underlying processor. This means that most of the time the VM is executing instructions uninhibited on its
cores. However, Jailhouse needs a way to proactively interrupt a VM on one core from another core. This situation occurs when a user in the root cell attempts to turn off an inmate via the Jailhouse command line interface. Normally, the inmate’s cores are completely isolated from whatever is happening on the root cell’s cores. But through the use of a non-maskable interrupt (NMI), the hypervisor instance running on a root cell core is able to interrupt an inmate core, causing a separate hypervisor instance in the inmate to take control.

As the name suggests, NMIs are (in most cases) non-maskable, meaning that they can’t be ignored even by the hypervisor. If the inmate happened to be in the middle of servicing a VM exit, causing the hypervisor to run in VMX host mode, what will happen when an NMI comes along? Since it is non-maskable, the NMI will interrupt even the hypervisor in host mode. So Jailhouse needs to explicitly intercept any and all NMIs, whether they arrive while the VM is running in guest mode or host mode.

If the VM is running in guest mode, then an NMI will trigger a VM exit and the VM exit handler will be able to determine that the VM exit was caused by an NMI.

If the VM is running in host mode (i.e., the hypervisor is running), then the NMI will immediately trigger the native NMI interrupt handler set for the core. No VM exit will occur, since Jailhouse is already running in VMX host mode.

There are two code paths that an NMI can go down. To reduce the number of states, the NMI handler on the host side does not handle the NMI completely; rather, it forwards handling the NMI by forcing a VM exit as soon as possible. It does this by turning on the preemption timer with a countdown value of 0. This will cause a VM exit as soon as the host is done and does a VM entry back into the VM. So once the NMI handler finishes in host mode, the host finishes doing whatever it was doing, and calls VMRESUME. Immediately upon entering guest mode, the preemption timer expires and a VM exit occurs. This time, the VM exit handler sees that the preemption timer was the cause and knows that this is a deferred NMI from host mode.

This process ensures that no matter if the NMI occurs in host mode or guest mode, ultimately it gets handled in the same way. The host doesn’t have to worry about dropping everything it was doing to handle the NMI - it can finish what it is doing and kick the NMI down the road via the preemption timer.
The way that NMIs are used by Jailhouse means that any NMIs that go to the guest will end up being handled by hypervisor; the guest VM itself will never see them. This could break some debugging and profiling utilities running in the guest VM that are expecting to see NMIs (likely including tools like VTune).

When an NMI occurs in guest mode, an NMI-based VM exit is triggered. Notably, the real NMI interrupt handler for the processor is not run by default. The VM exit handler sees that the VM exit was due to an NMI, and then manually triggers the real NMI interrupt handler via the `int` instruction. This indirection makes it so that both host mode and guest mode eventually run the real NMI interrupt handler, thus reusing the same logic for both cases. The real NMI handler then forces a preemption timer VM exit to occur immediately upon the imminent VM entry (i.e., as soon as control is given back to the guest).

In summary, the VMX preemption timer was only natively used by Jailhouse when an NMI arrives in host mode; when in guest mode, an NMI already forced an NMI-based VM exit, so the
preemption timer was not needed. However, we modified this interaction slightly in order to use the preemption timer to check for and initiate throttling in addition to its original usage.