A Flattened Hierarchical Scheduler for Real-Time Virtual Machines

Michael S. Drescher

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering

Binoy Ravindran
Antonio Barbalace
Haibo Zeng

May 18, 2015
Blacksburg, Virginia

Keywords: Real-Time, Virtualization, Scheduling, KVM, Linux
Copyright 2015, Michael S. Drescher
A Flattened Hierarchical Scheduler for Real-Time Virtual Machines

Michael S. Drescher

(ABSTRACT)

The recent trend of migrating legacy computer systems to a virtualized, cloud-based environment has expanded to real-time systems. Unfortunately, modern hypervisors have no mechanism in place to guarantee the real-time performance of applications running on virtual machines. Past solutions to this problem rely on either spatial or temporal resource partitioning, both of which under-utilize the processing capacity of the host system. Paravirtualized solutions in which the guest communicates its real-time needs have been proposed, but they cannot support legacy operating systems. This thesis demonstrates the shortcomings of resource partitioning using temporally-isolated servers, presents an alternative solution to the scheduling problem called the KairosVM Flattening Scheduling Algorithm, and provides an implementation of the algorithm based on Linux and KVM. The algorithm is analyzed theoretically and an exact schedulability test for the algorithm is derived. Simulations show that the algorithm can schedule more than 90% of all randomly generated tasksets with a utilization less than 0.95. In comparison to the state-of-the-art server based approach, the KairosVM Flattening Scheduling Algorithm is able to schedule more than 20 times more tasksets with utilization of 0.95. Experimental results demonstrate that the Linux-based implementation is able to match the deadline satisfaction ratio of a state-of-the-art server-based approach when the taskset is schedulable using the state-of-the-art approach. When tasksets are unschedulable, the implementation is able to increase the deadline satisfaction ratio of Vanilla KVM by up to 400%. Furthermore, unlike paravirtualized solutions, the implementation supports legacy systems through the use of introspection.
Acknowledgments

I’d like to thank the following people, without whom this thesis would not be possible:

Dr. Binoy Ravindran, who provided me with guidance in my research and through weekly meetings ensured I continued to make progress.

Dr. Antonio Barbalace, whose Linux kernel expertise proved invaluable during my time in the lab.

Dr. Haibo Zeng, who graciously took time out of his busy schedule to participate in my thesis committee.

Dr. Vincent Legout, who helped me compose the ideas in this thesis, write them down in a manner which can be understood by other humans, and evaluate my ideas analytically and experimentally.

The members of the System Software Research Group, whose friendship and skills helped turn the lab into an inviting and productive environment for research.

My fiancée, Nicolette Thompson, who unconditionally supported me throughout my thesis work, even when my research took most of my attention from her.

Finally, my parents Linda and Gregory Drescher, without whom, my collegiate education would never have been possible.
# Contents

1 Introduction ................................................. 1  
   1.1 Limitations of Past Work .......................... 2  
   1.2 Thesis Contributions .............................. 3  
   1.3 Scope of Thesis .................................. 3  
   1.4 Thesis Outline .................................. 4  

2 Background and Related Work ......................... 6  
   2.1 Background ...................................... 6  
      2.1.1 Real-Time .................................. 6  
      2.1.2 Virtualization .............................. 9  
   2.2 Related Work .................................. 11  
      2.2.1 Hierarchical Scheduling .................... 11  
      2.2.2 Real-Time Virtualization ................. 13  

3 Models and Assumptions ................................ 15  
   3.1 Task Model ...................................... 15  
   3.2 Guest Model ................................... 16  
   3.3 Hardware Model .................................. 17  
   3.4 Scheduling Model ................................ 17  

4 Dynamic CSF ............................................. 19  
   4.1 Overview of Static CSF ........................... 19
5 Flattening Scheduling

5.1 Advantages of Flattening Scheduling

5.2 EDF Guests

5.2.1 Algorithm Overview

5.2.2 Schedulability Analysis

5.3 Supporting Fixed-Priority Guests

5.3.1 Algorithm Overview

5.3.2 Schedulability Analysis

5.4 Schedulability Test for the KairosVM Flattening Algorithm

5.4.1 Interference from Tasks Scheduled under EDF

5.4.2 Interference from Higher Priority Tasks

5.4.3 Interference from Fixed-Priority Tasks in other Virtual Machines

5.4.4 Calculating the Worst-Case Response Time of a Task

5.5 Comparison with Server-based Approach

5.5.1 Examples

5.6 Multiprocessor Scheduling

5.6.1 Overview of Partitioning Algorithm

5.6.2 Schedulability Analysis

6 Implementation

6.1 Implementation of Scheduling Algorithms in the Linux Kernel

6.1.1 Scheduling Entities

6.1.2 Adding Support for Non-Real-Time Tasks on Guests

6.1.3 The “sched_yield_task” Function

6.1.4 Adding Support for RM Guests

6.2 Expansion of the Introspection Engine
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.1</td>
<td>The /proc Filesystem</td>
<td>53</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Task-Accounting using Thread-IDs</td>
<td>57</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Porting to AMD</td>
<td>58</td>
</tr>
<tr>
<td>6.3</td>
<td>Multiprocessor Scheduling</td>
<td>58</td>
</tr>
<tr>
<td>7</td>
<td>Experimental Results</td>
<td>59</td>
</tr>
<tr>
<td>7.1</td>
<td>Virtualization and Scheduling Overheads</td>
<td>59</td>
</tr>
<tr>
<td>7.2</td>
<td>Comparison with Vanilla KVM</td>
<td>62</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions</td>
<td>69</td>
</tr>
<tr>
<td>8.1</td>
<td>Contributions</td>
<td>69</td>
</tr>
<tr>
<td>9</td>
<td>Future Work</td>
<td>71</td>
</tr>
<tr>
<td>9.1</td>
<td>Expansion of Scenarios Evaluated</td>
<td>71</td>
</tr>
<tr>
<td>9.2</td>
<td>Extension of Guest OS Support</td>
<td>72</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Multiprocessor Guests</td>
<td>72</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Access to Shared Resources by Guests</td>
<td>72</td>
</tr>
<tr>
<td>9.3</td>
<td>Full Flattening with Paravirtualization</td>
<td>73</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Optimal Multiprocessor Scheduling</td>
<td>73</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 Example Real-Time Application Written for ChronOS Linux ............ 8
2.2 Flowchart of Kairos Introspection Engine, from [1] .................. 11
3.1 KairosVM Hardware and Scheduling Model ............................. 17
4.2 Supply-Bound Function for a Periodic Server \((k = 3)\), from [2] .... 21
4.3 Example Taskset Execution with CSF ................................. 23
4.4 Example Taskset Execution with d-CSF ............................... 24
5.1 Utilization bound of a Single EDF VM Scheduled under CSF .......... 29
5.2 Flattening Scheduling Algorithm with Fixed-Priority Guests ....... 30
5.3 Utilization bound of a Single RM VM Scheduled under CSF .......... 31
5.4 Graphical Representation of \(D'_i\) when \(i = 4\) ........................ 35
5.5 Graphical Representation of \(A\) and \(\Psi_x\) ............................. 38
5.6 Schedulability Comparison between KairosVM Flattening and CSF .... 39
5.7 Schedulability Comparison with Varied Guest Schedulers ............ 40
5.8 Example Taskset Execution with Flattening Scheduling ............. 41
5.9 Hierarchical Scheduling Example ...................................... 42
6.1 Deadline Scheduling Entities in Vanilla Linux ....................... 46
6.2 Deadline Scheduling Entities in KairosVM ............................ 47
6.3 Fixed-Priority Support in KairosVM ................................. 51
6.4 Organization of Data Structures Exported to \(/proc/kairos\) .......... 57
7.1 Overhead Characterization using Deadline Miss Load (Utilization = 0.95) . . 61
7.2 Overall System DSR with Random Synthetic Tasksets . . . . . . . . . . . . . 64
7.3 Overall System DSR with Instances of x264 . . . . . . . . . . . . . . . . . . 65
7.4 DSR of Individual VMs Running Instances of x264 . . . . . . . . . . . . . . 66
7.5 Overall System DSR for SD-VBS Applications . . . . . . . . . . . . . . . . 67
7.6 DSR of Individual VMs running SD-VBS Benchmarks . . . . . . . . . . . . . 68
# List of Tables

3.1 Model Notation Reference .................................................. 16

5.1 Summary of Response-time Analysis Notation .......................... 32

5.2 Set Notation ........................................................................ 36

6.1 KairosVM Kernel Modules .................................................. 45

6.2 Kairos /proc Files .............................................................. 54

7.1 Minimum Average Period of Tasks with No Deadline Misses ........ 62
Chapter 1

Introduction

There exist several applications today which benefit from guaranteed execution time at regular intervals. Such applications are known as real-time applications, and require an operating system capable of delivering the requested execution time in order to maintain an acceptable level of performance. One example of a real-time system is a multimedia decoding application, which at 60 frames-per-second requires enough execution time to decode and output a single frame of audio and/or video 60 times per second at regular intervals. If the requested execution time is not provided to the multimedia application in a timely manner, late-arriving frames will cause a noticeable jitter in playback of the media.

Another trend today is the move towards cloud-based computing on virtual machines. The movement of legacy systems to a virtualized cloud environment provides many advantages such as the ability to easily backup running systems, maintain isolation between two virtualized systems on the same physical machine, and migrate virtualized systems across physical servers in the case of hardware failures. Perhaps the biggest advantage of cloud computing is that it takes advantage of the full computing capacity of the physical servers when hardware upgrades are performed by consolidating many virtual machines into a single physical machine. Without virtualization, each hardware upgrade for a legacy system will lead to wasted clock-cycles and CPU cores. An alternative to virtualization is to rewrite real-time applications so that they take advantage of newer hardware at each hardware upgrade. However, the writing and testing of the rewritten application can come at a large development cost. Additionally, source code is not always available for real-time applications, which eliminates code rewriting as a solution.

While a lot of best-effort computing such as data mining and machine learning has migrated to the cloud, most real-time systems still run directly on hardware platforms. This is due to the lack of real-time performance guarantees in traditional virtualization software. This work focuses on providing real-time guarantees to virtualized systems such that legacy real-time systems such as multimedia decoding applications can be migrated to the cloud in an effort to fully utilize emerging multi-core hardware.
1.1 Limitations of Past Work

Some previous efforts to virtualize real-time systems rely on hardware partitioning to ensure the virtualized guest receives the execution time it requests from the physical CPU. While this serves to guarantee the real-time performance of the guest, it does not make efficient use of the CPU, particularly in the case where the guest does not request much execution time.

A better solution than hardware partitioning of processors is to divide processor time into chunks which are isolated temporally. In this approach, also known as a periodic server-based approach, guests are seen as servers which take turns executing on the processor. Offline analysis of the real-time tasks on each guest can provide the maximum proportion of CPU time required by each server in order to guarantee real-time performance. While this approach better utilizes the processing resources than a hardware partitioning approach, it still suffers from wasted CPU cycles. Additionally, it incurs increased runtime overhead due to the amount of context switches between virtual machines.

Improvement to the temporally-isolated server model can be obtained if guests are allowed to communicate with the host scheduler. Such a model is known as paravirtualization, in which communication between guests and the host is obtained via hypercalls. The drawback of a paravirtualization model is that guests must be modified in order to make the hypercalls. In legacy systems, source code is not always available for modification, and thus paravirtualization is not an appropriate solution when targeting legacy systems.

Introspection is an alternative method to extract information from guest operating systems at runtime. There is an abundance of prior research in the area of introspection into virtual machines; however, this research is nearly entirely focused on maintaining the security of both the virtual machines and the host operating system. There also exists prior work which focuses on using introspection to gather real-time task information from guest operating systems at runtime, but this work does not use the task information to schedule real-time guests.

Scheduling real-time virtual machines is inherently a hierarchical scheduling problem. There must be at least two levels of schedulers, since each guest operating system has its own scheduler which cannot be modified. While most prior work in hierarchical scheduling of virtual machines focuses on the periodic server-based model, there does exist prior work in the realm of hierarchical scheduling without a focus on virtualization. Rather, this prior work focuses on scheduling tasks within a single system using a finite number of priorities. Within each priority level, a dynamic-priority algorithm is used to schedule tasks in an effort to improve overall schedulability. While this work is similar to the algorithm presented in this thesis, it would only apply to a guest model in which virtual machines are assigned priorities relative to one another. In this way, the top-level scheduler would be a fixed-priority scheduler. This thesis examines a guest model in which no priority between guests is assumed; thus, a novel hierarchical scheduler is necessary.
1.2 Thesis Contributions

This goal of this thesis is to provide a solution for the scheduling of real-time virtual machines which does not rely on temporally-isolated servers. The major contributions presented in this work include the following:

- An algorithm which can schedule real-time virtual machines without the use of servers. The algorithm supports guests which use either the earliest-deadline-first (EDF) scheduler, or a fixed-priority scheduler. The algorithm is extended to multiprocessor systems using the first-fit partitioning method. Development of the algorithm is motivated by examining the shortcomings of the state-of-the-art server-based approach to scheduling real-time virtual machines, and addressing these shortcomings by removing the server abstraction.

- An exact schedulability test for the algorithm. The schedulability test is used to conduct simulations which demonstrate the algorithm’s ability to schedule a higher proportion of random tasksets than state-of-the-art server-based approaches. The simulations also demonstrate that at high utilizations, the serverless algorithm’s advantage over the state-of-the-art approach is augmented; this means that the effective utilization bound for the algorithm is higher than the state-of-the-art approach to real-time virtual machine scheduling.

- An implementation of the algorithm based upon Linux and KVM. The implementation supports real-time guests which use either the EDF or RM schedulers running in a legacy real-time operating system. This implementation is the first open-source implementation of a real-time hierarchical scheduler for virtual machines in Linux which does not use servers. Additionally, this is the first serverless implementation to use introspection rather than paravirtualization; that is, this is the first implementation which supports legacy real-time guests without using servers.

- The infrastructure necessary to evaluate the Linux-based implementation using legacy real-time guests. This infrastructure is used to provide evaluations of the implementation’s overheads as well as its real-time performance in comparison to vanilla Linux and a state-of-the-art server-based approach. Evaluations demonstrate that the serverless implementation is able to obtain a higher deadline-satisfaction-ratio than the state-of-the-art alternative when utilization is increased to near-overload and overload levels for multiple real-world real-time applications.

1.3 Scope of Thesis

This work focuses on the consolidation of multiple legacy real-time systems into a single host. Clearly, the breadth of existing legacy real-time systems combined with the complexity of
both the Linux scheduler and virtualization software create a vast problem space. This section describes the steps taken and the assumptions made to reduce the problem space to a more manageable size.

The first major limitation of this thesis comes in the form of the task model considered. It is assumed that tasks never block while waiting for either I/O or access to critical sections protected by a lock. Additionally, the tasks analyzed by this thesis are all assumed to be periodic tasks with defined worst-case execution times. Chapter 3 describes the task model in more detail. For this thesis, inter-dependent tasks, aperiodic tasks, and sporadic tasks are all out-of-scope.

A second limitation of this thesis comes in the form of the guest and hardware models, also discussed in Chapter 3. While the implementation presented in this thesis can be expanded to other guest operating systems, only ChronOS is supported by this work. Additionally, this thesis restricts its hardware evaluations to x86-based processors running x86-based guests, again due to limitations of the implementation. It is important to note that this thesis attempts to isolate the performance of the host scheduler in its evaluations, and that examination of the effects of cache, the hypervisor, or the guest scheduler on performance is out-of-scope.

The third and final limitation of this thesis lies in the evaluations and simulations. For practical reasons, evaluations are only done with up to 6 virtual machines on a single host. Similarly, the benchmarks evaluated are limited to those which support or have been ported to ChronOS Linux. Only one benchmark is evaluated at a time; a system with a mixture of benchmarks running simultaneously is also out-of-scope. Since the task model examined provides for an infinite number of ways to divide tasks among virtual machines, the simulations in Chapter 5 assume a taskset with 6 tasks divided among 3 VMs, two of which are scheduled using RM with the other scheduled using EDF.

The number of potential tasksets and combinations of guest operating systems is infinite; this thesis attempts to limit itself to a set of evaluations which is small enough to be feasible while general enough to provide useful results.

### 1.4 Thesis Outline

The remainder of this thesis is organized as follows:

- Chapter 2 provides background knowledge in the area of real-time virtualization along with a survey of related work.
- Chapter 3 introduces the models and notations used within the thesis to analyze real-time scheduling algorithms.
• Chapter 4 describes a run-time optimization to the server-based scheduling model which leverages introspection.

• Chapter 5 presents the KairosVM Flattening Scheduling Algorithm along with a schedulability test for the algorithm.

• Chapter 6 discusses the implementation of the scheduling algorithm in the Linux kernel.

• Chapter 7 offers results from experimental evaluations of the implementation.

• Chapter 8 presents the conclusions drawn from this work.

• Chapter 9 provides a list of potential research topics which build upon the work presented in this thesis.
Chapter 2

Background and Related Work

This chapter provides background knowledge on some topics which are integral to the work presented in this thesis. An overview of existing works in hierarchical real-time scheduling as well as real-time virtualization is also presented.

2.1 Background

In order to fully understand the contributions of this thesis, a base knowledge of the concept of real-time systems is necessary. This background information is presented in Section 2.1.1 along with information about the host real-time scheduler and guest real-time scheduler used in this work.

Section 2.1.2 provides background knowledge of the virtualization technology used in this thesis. Additionally, an explanation of the introspection mechanism and what it provides the host is presented here.

2.1.1 Real-Time

Real-time computer systems consist of systems which run applications that have strict timing requirements. In real-time theory, the result of a computation is only useful to the application if it is obtained before a specific deadline. For example, a networked server may be expected to deliver a response to a request within a certain number of milliseconds before the client assumes the server is down. If the server waits too long to respond, the client may request a backup server instead, rendering the initial server’s late response useless.

Real-time systems and applications can be further subdivided into two classes: hard real-time and soft real-time. Hard real-time applications cannot afford to miss any deadlines and
thus run on systems with deterministic processing times and overheads. An example of a hard real-time system and application would be an automotive controls system, in which case a deadline miss could result in failure to deploy an airbag on time during an impact, possibly leading to human injury or death. Soft real-time applications can afford to miss a certain proportion of their deadlines without a significant degradation of performance. An example of a soft real-time application would be a multimedia decoder, which can afford to miss a frame or two without significantly degrading the quality of the user experience. Soft real-time applications can be run on operating systems with less-deterministic overheads such as Linux.

**SCHED DEADLINE**

One such example of a system targeting soft real-time applications is SCHED DEADLINE [3], an Earliest Deadline First (EDF) real-time scheduling policy in the Linux kernel. SCHED DEADLINE was merged into the mainline Linux kernel in version 3.14.0. It is only able to support soft real-time applications because the overheads of the Linux kernel and scheduler are non-deterministic. Hard real-time systems require overheads which are strictly bounded.

SCHED DEADLINE also implements a policy called Constant Bandwidth Server (CBS) [4], which prevents tasks from overrunning their allotted execution time, thus guaranteeing the temporal isolation of tasks in the system. The CBS implementation in SCHED DEADLINE was revisited in 2014 by Abeni et al. in [5], to add support for the “self-suspending task model”, for which the original SCHED DEADLINE implementation does not provide temporal isolation.

SCHED DEADLINE is implemented as a scheduling class within the Linux scheduler. Each real-time task has its period, deadline, and runtime stored in a data structure. The data structure also contains values for the absolute deadline and remaining runtime for the current release of a task. SCHED DEADLINE stores the absolute deadline of every task in the system in a red-black tree ordered by deadline from earliest to latest. Thus, the earliest deadline in the system is always the leftmost node in the tree.

When a task has exhausted its runtime for a given release, SCHED DEADLINE’s CBS implementation takes effect. The task is throttled, which means it is removed from the current CPU’s runqueue, and put to sleep. The task’s absolute deadline is also removed from the red-black tree at this time. At the time of throttling, a timer is started which will wake the task up at the next release. When the task finally awakens for its next release, it is put back onto the runqueue and the absolute deadline is added back into the red-black tree.

The CBS algorithm provides temporal isolation between tasks, which means a deadline miss in one task will never cause deadline misses in other tasks. In SCHED DEADLINE, if a given task attempts to exceed its runtime, or misses its deadline, the throttling mechanism
described above takes effect. In this way, unlike in the classical version of EDF, when a single task misses a deadline, the execution of other tasks is not ever delayed.

ChronOS Linux

```c
int main() {
    struct timespec period, deadline, tmp;
    unsigned long prio = 90;
    period.tv_sec = 0;
    period.tv_nsec = 10000000;
    for (int i = 0; i < NUM_JOBS; i++) {
        gettimeofday(&tmp);
        deadline = timespec_add(&tmp, &period);

        /* Begin real-time segment; Pass in real-time parameters to scheduler */
        begin_rtseg_selfbasic(prio, &deadline, &period);

        /* ... Real-Time Code */

        /* End real-time segment; Let scheduler know it is free to schedule other tasks */
        end_rtseg_self(prio);

        gettimeofday(&tmp);
        tmp = timespec_sub(&deadline, &tmp);
        if (timespec_above_zero(&tmp))
            usleep(tmp.tv_nsec/1000);
    }
    return 0;
}
```

Figure 2.1: Example Real-Time Application Written for ChronOS Linux

ChronOS Linux [6] is a real-time Linux-based operating system which implements a number of traditional real-time scheduling algorithms as well as best-effort policies.

Unlike SCHED_DEADLINE, the ChronOS scheduler does not implement its own additional real-time scheduling class in Linux. Rather, ChronOS extends the real-time scheduling class which implements a fixed-priority scheduler. The ChronOS scheduling queue resides at a fixed priority level $n$ within the `rt_sched_class`. Different scheduling algorithms and
policies are implemented via loadable kernel modules, using a pluggable interface into the kernel scheduler.

Real-time applications written for ChronOS Linux are required to mark the beginning and end of the code representing each real-time task with calls to `begin_rtseg()` and `end_rtseg()` respectively. The `begin_rtseg()` system call requires the relative period (represented as a `struct timespec`), absolute deadline (represented as a timestamp with `struct timespec`, and priority (represented as an integer) to be passed in as arguments. Additionally, the optional argument of worst-case execution time in microseconds can also be passed as an argument (also represented as an integer). The `end_rtseg()` system call lets the ChronOS scheduler know when a task release has completed its execution, so that a different real-time task can then be scheduled. An example ChronOS real-time application’s code is presented in Figure 2.1.

### 2.1.2 Virtualization

Virtualization is the concept of executing an entire operating system (guest) as an application on top of another operating system (host) which runs on bare hardware. This host operating system is known as a hypervisor, and is in charge of scheduling the guest operating systems. Hypervisors are not necessarily fully-featured operating systems; they need only provide the minimal functionality necessary to run guests. There are two common methods of virtualization in industry: full virtualization and paravirtualization.

In full virtualization, the guest operating systems do not have privileged access to hardware. This means that interactions between the guests and the hardware on the system which require privilege processor access must be mediated by the hypervisor. Additionally, guest operating systems in a fully virtualized setup are unaware that they are running in a virtual machine. Typically this is accomplished by emulating all of the hardware devices that the guest expects to have access to.

In paravirtualization, the guest is aware that it is running under a hypervisor, and communicates with the hypervisor through a set of function calls known as hypercalls. Legacy systems which used to run on bare hardware but are now running as virtual machines cannot utilize paravirtualization, as it requires modifications to the guest operating system kernel.

The Kernel-based Virtual Machine, or KVM [7], is a loadable kernel module in Linux which mediates access between virtual machines and the host hardware. KVM is typically paired with QEMU [8] to provide emulation of hardware devices, and KVM/QEMU combined provide a fully-virtualized environment for guests.

KVM leverages virtualization hardware support through the `kvm_intel` and `kvm_amd` modules. The `kvm_intel` module utilizes the Intel-VT hardware extensions in Intel x86-based processors to provide the full processing power of the CPU to the guest operating system with minimal overheads. This is accomplished via calls to the x86 instructions `vmenter`
and `vmexit`. These instructions provide isolation between the host system and the guest by redirecting hardware exceptions such as divide-by-zero, page-faults, and segmentation faults to KVM. In this way, a single faulty guest operating system cannot bring down the entire host. Furthermore, the hardware extensions provide isolation between guests such that one compromised guest cannot compromise the other guests in the system. The `kvm_amd` module leverages AMD-V hardware support, which provides similar services as Intel-VT.

**Introspection**

While hypercalls provide a method to export real-time parameters from guest tasks to the host, they are not possible in a fully-virtualized setup. The Kairos Introspection Engine [1] is an alternative method of exposing real-time parameters from guests to the host operating system. It is implemented as a modification to KVM/QEMU and takes advantage of the fact that the Intel-VT and AMD-V hardware extensions return control to the host whenever an exception occurs in the guest.

The Kairos Introspection Engine works by writing the undefined x86 ud0 instruction at strategic points in the guest operating system address space. The insertion is done using a modified version of QEMU. If a ud0 instruction is inserted at the addresses of the `begin_rtseg` and `end_rtseg` ChronOS kernel functions on a ChronOS guest, the Intel-VT or AMD-V hardware exceptions will give control of the processor to the host with the execution of the virtual machine paused at the start of these system calls. At this point, the host extracts the real-time arguments to the system calls from the virtual guest registers.

When the guest address space is overwritten, the Kairos Introspection Engine saves the original instruction at this point so that the guest can still execute the instruction after the ud0 instruction is trapped. This is important to ensure that the guests still operate as if they were unmodified. Also, it is important to note that it is entirely possible that the hardware extensions trap illegal instructions that were not placed there by Kairos, but instead by either malicious or buggy guest software. It is important that the modified version of KVM can distinguish the difference between trapped exceptions caused by Kairos and other exceptions. This is done by storing the addresses that were overwritten, and later checking the current instruction pointer of the guest against these stored addresses.

Figure 2.2 from [1], demonstrates the flow of execution on a CPU running a guest under the Kairos Introspection Engine. At the start, the guest executes its applications as normal until it encounters an illegal or undefined instruction. The virtualization extensions trap the instruction and turn over control to the hypervisor, at which point the Kairos Introspection Engine checks to see if the address of the illegal instruction is registered (if yes, this means the ud0 instruction was placed there by Kairos). If the address is registered, the real-time parameters are extracted and the original instructions at that address before the overwriting are emulated. If the address is not registered, the hypervisor acts as normal and attempts to emulate the illegal instruction before returning control to the guest.
2.2 Related Work

2.2.1 Hierarchical Scheduling

Scheduling virtual machines inherently implies a hierarchy of schedulers, and as such several hierarchical scheduling literature is relevant to the work presented in this thesis. The concept of hierarchical scheduling using servers was first presented in [9] and [10]. Servers provide a method of representing an application or virtual machine with its own real-time tasks and scheduler as a single entity to be scheduled by the host operating system. [11] presents a schedulability analysis of servers scheduled under fixed-priority or EDF as the top-level scheduler.

[9] introduced the deferrable server algorithm, which accommodates high priority aperiodic
tasks by providing a server capacity which is depleted upon arrival of aperiodic tasks and replenished at the start of each period. Strosnider et al. also introduced the polling server in [9], which operates similarly to a typical periodic server in that it has a budget which executes any pending aperiodic tasks at the start of its period until either the budget is depleted or there are no more pending aperiodic tasks.

Lipari et al. extend and improve upon hierarchical scheduling with servers by integrating the Constant Bandwidth Server framework in [12], examining how to partition resources among servers in [13], providing a method to compute server parameters in [14], and how to extend hierarchical scheduling to multiprocessor systems in [15].

The concept of “flattening scheduling” was first introduced in [16], which observed that removing the concept of servers can improve schedulability of a system. However, no hierarchical algorithms were presented in this work which allow servers to be discarded. This thesis expands on the ideas presented by Lackorzynski et al. in [16] by providing an algorithm, theoretical analysis, and implementation of flattening scheduling.

There also exists analytical work for the hierarchical scheduling of tasks without real-time servers, such as an analysis of EDF within priorities presented by Harbour and Palencia in [17]. While this work is similar to the work presented in this thesis, the scheduler presented here differs from the one in this thesis and does not have a focus on virtualization. In [18] and [19], hierarchical scheduling is used to integrate real-time tasks with multimedia tasks.

Unlike much of this related work, this thesis differs in that an implementation with experimental evaluations is provided.

**Compositional Scheduling Framework**

In [20] and [2], Shin and Lee introduced the Compositional Scheduling Framework (CSF). CSF provides a method to compute the budget of a periodic server given the set of guest tasks, the guest scheduler, and the period of the server on the host. CSF is useful for hierarchical scheduling because it is fully composable, meaning that a server can schedule servers which schedules its own servers, and so on. Additionally, CSF supports many scheduling policies and is easy to implement. In theory, if the server period is small, the taskset of the server will have the same schedulability as if the taskset was running directly on the host.

This thesis uses CSF as a competitor because it has been adopted by competing real-time virtualization solutions such as RT-Xen [21]. Additionally, because CSF is composable and supports multiple schedulers within the server, CSF is one of a handful of hierarchical scheduling algorithms which can handle tasksets generated by the task model used in this thesis.
2.2.2 Real-Time Virtualization

Many hypervisors capable of running virtual machines exist today. This thesis restricts itself to the analysis of open-source hypervisors such as Xen and KVM rather than proprietary software, because the research community does not have evaluations of these proprietary systems.

In [22], a system which handles resource-sharing among virtual machines is examined. In this system, hypercalls are used to expose the critical sections within guest tasks to the host operating system. Using this information, the hypervisor is able to reduce bounds on blocking times, increasing the schedulability of the system. Unlike the work presented in this thesis, [22] uses servers to schedule virtual machines.

Xen-based Solutions

In [23], Lee et al. introduce a method to support soft real-time tasks in the Xen hypervisor by reducing scheduling latency and managing shared caches among VMs.

Real-Time Xen [21] was introduced in 2011 by Xi et al. as a method to schedule real-time virtual machines in Xen without paravirtualization. RT-Xen uses the concept of servers to schedule virtual machines, which in this work are limited to a single virtual CPU pinned to a single physical CPU core, similar to the model used by this thesis. In 2012, Lee et al. added CSF to Real-Time Xen [24]. With CSF, RT-Xen is able to guarantee that deadlines are met when the taskset is schedulable.

Multicore scheduling was introduced in RT-Xen in 2014 by Xi et al. [25]. This work evaluates the use of both global and partitioned EDF (gEDF and pEDF respectively) schedulers as the top-level scheduler for the servers. They found that overheads are moderate for both, and that contrary to the results of theoretical analysis, experiments prove that global edf performed better than partitioned edf. This work also includes support for multicore guests using pEDF or gEDF as the guest scheduler. They claim that the best multicore setup includes pEDF as the scheduler on guests, and gEDF as the scheduler on hosts, claiming that the benefits of gEDF outweigh the cache penalty incurred when migrating a virtual machine.

Linux/KVM-based Solutions

Kiska et al. examine the use of Linux and KVM as a real-time hypervisor in [26]. They assign real-time priorities to QEMU threads using the SCHED_FIFO fixed-priority scheduling class in an effort to better satisfy the real-time requirements of guests. They examine the latencies of I/O from the guest due to virtualization with and without prioritization of QEMU threads, and conclude that Linux has potential as a real-time hypervisor since the latencies are
manageable. Additionally, they discuss a mechanism for paravirtualization to ensure that shared-resources do not induce priority-inversion between real-time guests and host tasks.

In [27], Cucinotta et al. examine the use of KVM as a hypervisor when the guest is running IRMOS, an operating system designed to run real-time multimedia applications. IRMOS partitions its real-time tasks using cgroups in Linux, while Cucinotta et al. modified KVM to partition host resources among real-time guest tasks in an effort to reduce network latencies. The authors evaluate their implementation using measurements of network response times from the guest. This work differs from this thesis in that it is limited to a specific use case and focuses on reducing network latencies in the guest rather than modifying the host scheduler to maintain real-time guarantees.

Other Solutions

Lackorzynski et al. use a real-time hypervisor known as Fiasco in [16]. In the paper, guests are scheduled using servers; however the authors note that servers present a problem when trying to maintain hard real-time guarantees, and thus introduce their “flattening scheduling” idea. This work suggests using hypercalls to flatten guest and host schedulers, but does not go into detail about how this could be implemented and how the host could use this information to schedule guests. Additionally, the authors did not evaluate the feasibility of a flattening solution in the context of real-time guarantees.
Chapter 3

Models and Assumptions

In real-time theory, there are many lenses through which to view a computational system which has time constraints. To simplify both the analysis and the implementation, this thesis restricts real-time systems to a task and guest model. These models are presented in detail in the following chapter.

3.1 Task Model

This work considers the task model defined by Liu and Layland [28]. In this model, a real-time task $\tau_i$ is characterized by its period $T_i$, its deadline $D_i$, and its worst-case execution time $C_i$. For the benchmarks evaluated in this work, every task’s deadline $D_i$ is equal to its period $T_i$. Each task in this model releases a single job once per period. A job is denoted as $\tau_i^j$, where $j$ represents the $j^{th}$ release of task $\tau_i$. It is assumed that no job will ever exceed its worst case execution time $C_i$. However, the implementation provides safeguards against this possibility.

A set of tasks is denoted by $\Gamma$ and referred to as a taskset. In this model, a taskset is divided into non-overlapping subsets $g_k$, where $g_k$ contains the tasks which belong to guest operating system $k$. The hyperperiod of a taskset $\Gamma$ is equal to the least common multiple (LCM) of $T_i$ for all $\tau_i \in \Gamma$.

A tasks utilization $U_i$ is defined as $C_i/T_i$. The utilization of a guest $g_k$ is defined as $\sum_{\forall \tau_i \in g_k} U_i$. The overall utilization $U$ of a taskset $\Gamma$ is defined as $\sum_{\forall \tau_i \in \Gamma} U_i$. A summary of the notation used in this theses is presented in Table 3.1.
Table 3.1: Model Notation Reference

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>$\tau_i$</td>
</tr>
<tr>
<td>Job</td>
<td>$\tau_i^j$</td>
</tr>
<tr>
<td>Taskset</td>
<td>$\Gamma$</td>
</tr>
<tr>
<td>Period</td>
<td>$T_i$</td>
</tr>
<tr>
<td>Deadline</td>
<td>$D_i$</td>
</tr>
<tr>
<td>Worst Case Execution Time</td>
<td>$C_i$</td>
</tr>
<tr>
<td>Task Utilization</td>
<td>$U_i$</td>
</tr>
<tr>
<td>Overall Utilization</td>
<td>$U$</td>
</tr>
<tr>
<td>Guest Taskset</td>
<td>$V_k$</td>
</tr>
<tr>
<td>Guest which contains $\tau_i$</td>
<td>$V_{\tau_i}$</td>
</tr>
<tr>
<td>Guest Server</td>
<td>$g_k$</td>
</tr>
<tr>
<td>Set of EDF Guests</td>
<td>$V_{EDF}$</td>
</tr>
<tr>
<td>Set of RM Guests</td>
<td>$V_{RM}$</td>
</tr>
<tr>
<td>Number of Processors</td>
<td>$m$</td>
</tr>
<tr>
<td>Number of Tasks</td>
<td>$n$</td>
</tr>
</tbody>
</table>

3.2 Guest Model

This work aims to schedule a collection of real-time guests with taskset $\Gamma$, divided into individual tasksets $g_k$. Each guest is defined as a virtual machine running on top of a host operating system known as a hypervisor. The guests in this model are fully virtualized; this means they are executing under the assumption that they are running on real hardware, and thus make no attempts to communicate with the hypervisor. Furthermore, the guests operate without privileged hardware access, just as any user-space process would in Linux.

In this work, each guest runs the ChronOS Linux real-time operating system [6], though the implementation can be ported to support other guest operating systems. Each guest is assumed to be using either EDF or RM as its scheduler. All of the solutions presented in this work also support a mixture of EDF and RM guests. The guest schedulers are uninfluenced by both the hypervisor and other guests. Therefore, an EDF guest will always schedule the active job $\tau_i^j \in g_k$ which has the earliest absolute deadline. Similarly, an RM guest $k$ will always schedule the highest-priority active job which belongs to $g_k$.

It is assumed that each guest runs on a single virtual CPU (vCPU). Both theoretical and practical support for multi-core guests is left to future work.
3.3 Hardware Model

All theoretical and experimental results presented in this work assume a homogeneous multicore x86-based architecture which the host operating system runs on. The implementation presented in this work relies on the Intel-VT or the AMD-V virtualization hardware extensions, described in detail in Section 2.1.2. It is assumed that the host system consists of $m$ identical, independent cores. This work does not consider the effects of cache on performance.

In uniprocessor evaluations, the host only is able to use a single physical CPU to schedule multiple guests, which each have a single vCPU. In multiprocessor evaluations, partitioned scheduling is used, such that vCPUs are unable to migrate to other physical CPUs after they begin executing.

Figure 3.1 demonstrates the layers between the taskset and the physical CPUs. In uniprocessor evaluations, each vCPU in Figure 3.1 would be pinned to a single physical CPU underneath, while multiprocessor setups divide the 3 vCPUs in the figure between the two physical CPUs.

3.4 Scheduling Model

This work examines a hierarchical scheduling model in which a single host schedules multiple guests. The role of the hypervisor is to choose which guest to schedule at any given point in time. Because the guest operating systems are unaware of the hypervisor and other guests, at any given point in time each guest operating system will have a preferred job $\tau^j_i$ which the hypervisor cannot alter. Essentially the hypervisor must determine which preferred job $\tau^j_i$ to schedule at each point in time.

No priorities are assumed between different guest operating systems. Supporting systems with priority requirements between different virtual machines is left to future work.
Figure 3.1 shows the hierarchy of schedulers. Each individual guest must schedule its own taskset with its own scheduler, while the hypervisor must decide which vCPU to execute on the physical CPU at any given time.

The host operating system for both the implementation presented in this paper as well as the competitors presented uses a constant-bandwidth-server (CBS) implementation. This means that tasks are preempted if they attempt to overrun their specified worst case execution times $C_i$. The model used by this work and the work of the competitors does not assume any preemption overhead or scheduling overhead. Contrary to most studies in this field, this thesis includes experimental evaluations. The experimental evaluations incorporate the overheads of preemption and scheduling.
Chapter 4

Dynamic CSF

This chapter outlines an optimization to the compositional scheduling framework (CSF) which is made possible by introspection into the guest operating systems. The optimized version of CSF presented in this chapter is referred to as Dynamic CSF, or d-CSF, while the unoptimized version is referred to as either Static CSF, or simply CSF.

4.1 Overview of Static CSF

In order to better explain the concept behind Dynamic CSF, this paper presents some of the original Compositional Scheduling Framework (CSF) theory for the periodic task model, from [2]. CSF aims to provide a hierarchical method of scheduling tasks such that the hierarchy is composable; that is, it is possible to create a server with CSF whose tasks are actually servers whose budgets were computed using CSF. The CSF theory supports a periodic task model. Servers are able to contain tasks scheduled using either EDF or RM. Figure 4.1 [2] shows an example scenario where CSF is used to schedule 3 servers, each scheduled individually with either EDF or RM. For example, the top server, guest $g_0$, contains two individual tasks scheduled by EDF, which themselves are servers denoted by $g_1$ and $g_2$. Guest $g_2$ contains 2 tasks with periods of 40 and 25, and worst-case execution times of 5 and 4 respectively. If guest $g_2$ gets 4.4 units of execution time every 10 units of time, CSF guarantees that there will be no deadline misses in $g_2$. The same is true for guest $g_1$, and since the framework is compositional, for $g_0$ as well.

CSF provides the theory necessary to ensure that each periodic server supplies enough execution time at a high enough rate to ensure the individual tasks within each server do not miss their deadlines. All equations presented in this section are adapted from [2] to better match the notation of this thesis.

The theory behind CSF relies on the concepts of the supply-bound function (sbf) and the
The demand-bound function (dbf). The sbf represents a lower bound on the capacity of a processor during the interval between time 0 and time $t$. The dbf represents an upper bound on the demand for processing resources during this same interval, given a real-time taskset $\Gamma$. So long as the lower bound on processing supply is at least as large as the upper bound on demand, the workload is schedulable. This schedulability condition is given in Equation 4.1:

$$\text{dbf}(\Gamma, t) \leq \text{sbf}(t) \quad \forall t > 0 \quad (4.1)$$

If Equation 4.1 is satisfied for the hyperperiod of the workload $\Gamma$, then it is also true for every value of $t > 0$, since the execution will repeat beyond the hyperperiod.

The supply bound function for a periodic server with period $\Pi$ and budget $\Theta$ is given below:

$$\text{sbf}(t) = \begin{cases} 
    t - (k + 1)(\Pi - \Theta) & \text{if } t \in [(k + 1)\Pi - 2\Theta, (k + 1)\Pi - \Theta], \\
    (k - 1)\Theta & \text{otherwise}
\end{cases} \quad (4.2)$$

$$k = \max\left(\left\lceil \frac{t - (\Pi - \Theta)}{\Pi} \right\rceil, 1\right)$$

Figure 4.2 [2] illustrates the meaning of the supply-bound function for a period server. The value $k$ represents the minimum number of server executions which overlap with the interval.
of length $t$. In Equation 4.2, the top case is for a server execution which only partially overlaps with the interval (for example, the last execution of the server in Figure 4.2). The bottom case accounts for every other execution of the server within the interval. Since the equation represents a lower bound on the supply, the interval is assumed to begin immediately after a server execution which was released at the very beginning of its period, while each subsequent execution is pushed to the very end of each server period. This is illustrated in Figure 4.2.

The demand bound function for EDF, $\text{dbf}_{\text{EDF}}(\Gamma, t)$, was proposed by Baruah et al. [29]:

$$\text{dbf}_{\text{EDF}}(\Gamma, t) = \sum_{\tau_i \in \Gamma} \left\lfloor \frac{t}{T_i} \right\rfloor \cdot C_i$$

(4.3)

Thus, a server containing tasks scheduled using EDF is schedulable if:

$$\text{dbf}_{\text{EDF}}(\Gamma, t) \leq \text{sbf}(t) \quad 0 < t \leq LCM_\Gamma$$

(4.4)

Where $LCM_\Gamma$ represents the hyperperiod of $\Gamma$, calculated by finding the least-common-multiple of all the periods, $T_i$, in the taskset.

Similarly, the demand bound function for task $\tau_i$ scheduled under RM, $\text{dbf}_{\text{RM}}(\Gamma, t, i)$, was proposed by Lehoczky et al. [30]:

$$\text{dbf}_{\text{RM}}(\Gamma, t, i) = C_i + \sum_{\tau_k \in \text{HP}_\Gamma(k)} \left\lfloor \frac{t}{T_k} \right\rfloor \cdot C_k$$

$$\text{HP}_\Gamma(k) : \{\tau_i \in \Gamma \mid P_i \geq P_k, i \neq k\}$$

(4.5)
Using 4.5, the worst-case response time of task $\tau_i$ can be computed as the first time at which the supply of processing capacity is at least as large as the demand for processing capacity from $\tau_i$:

$$R_i = \min(t| dbf_{RM}(\Gamma, t, i) \leq sbf(t)) \quad (4.6)$$

Thus, the taskset is schedulable if for every task, the worst-case response time is less than the period of the task:

$$\forall \tau_i \in \Gamma \; \exists t_i \in [0, T_i] \; | \; dbf_{RM}(\Gamma, t, i) \leq sbf(t) \quad (4.7)$$

To compute a budget $\Theta$ for a virtual machine scheduled under CSF, the taskset $\Gamma$, the guest scheduler, and the server period $\Pi$ must all be given. Starting with the budget equal to 0, either Equation 4.4 or Equation 4.7 is evaluated, depending on whether the guest scheduler is EDF or RM. The budget is iteratively increased until either the appropriate equation is satisfied, or the server budget exceeds the server period (in the case the taskset is unschedulable). Note that Equations 4.4 and 4.7 require iteration over time from 0 to the taskset hyperperiod, or the task period respectively.

### 4.2 Idle Processor Time in CSF

As outlined in Section 4.1, CSF provides a method of computing the budget necessary to guarantee no deadline misses given a server period and a guest taskset and scheduler. However, even when all tasks execute for their WCET, there still exists the possibility that a server is running and idle, while another server with real-time tasks is blocked from executing.

Consider an example with two virtual machines, each running EDF as their guest schedulers. In this example, the host is also scheduling servers using EDF. Ties are broken arbitrarily; in this example when two tasks or servers have identical deadlines, the task or server with the lower index will execute first (i.e. $\tau_1$ will execute before $\tau_2$ if deadlines are equal). To illustrate the point that server-based approaches can lead to idle CPU time, the server budgets in the following example are chosen such that all deadlines are satisfied, rather than calculated using CSF. This also has the benefit of keeping all relevant example parameters, including server budgets, to whole numbers.

Figure 4.3 shows an example execution where CSF twice delays the execution of a real-time task because an idle server is executing its budget. The first instance of this behavior is at time 8, where guest $g_1$ executes even though task $\tau_4$'s first job still hasn’t terminated. A second instance occurs at time 12, where guest $g_1$ remains idle for its entire budget while task $\tau_4$ continues to be blocked. Even though these idle times are guaranteed not to cause deadline
misses under CSF, they represent wasted computational resources which could potentially be used to either accommodate more real-time tasks, more real-time guests, or both.

Additionally, as presented in [1], jobs do not always execute for their entire worst case execution time. In the case where a job terminates earlier than specified by the WCET, the remaining budget allocated to the job’s server can potentially be claimed by other jobs. While reclaiming this idle time won’t increase the utilization bound for CSF, it can increase experimental deadline-satisfaction-ratios when using a probabilistic WCET in soft real-time systems. For example, consider the case where task $\tau_2$ only executes for half it’s WCET during its first release. Then there will be 2 more instances of wasted execution time at time 4 and at time 6, when guest $g_1$ is executing, but idle, while guest $g_2$ still has a release of $\tau_4$ to execute. If $\tau_4$’s WCET is probabilistic, it can exceed its WCET with some probability. If this happens, the idle time used up by guest $g_1$ could eventually lead to a deadline miss for $\tau_4$, which would have been prevented if the idle time was instead used to execute $\tau_4$.

Note that the hyperperiod for the taskset in Figure 4.3 is 20. Beyond time 20, the execution pattern shown repeats indefinitely.

### 4.3 Reclaiming Unused Budget

The idea behind Dynamic CSF is to take advantage of idle time in CSF without any cost to deadline satisfaction ratio. This is accomplished via introspection. Since introspection
Figure 4.4: Example Taskset Execution with d-CSF

<table>
<thead>
<tr>
<th>Server Period</th>
<th>Server Budget</th>
<th>Task</th>
<th>Period = Deadline</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest $g_1$</td>
<td>4</td>
<td>3</td>
<td>$\tau_1$</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\tau_2$</td>
<td>20</td>
</tr>
<tr>
<td>Guest $g_2$</td>
<td>4</td>
<td>1</td>
<td>$\tau_3$</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\tau_4$</td>
<td>20</td>
</tr>
</tbody>
</table>

This observation leads to the two criteria which must be met in order to reduce a server’s budget to 0:

1. There are no tasks currently executing inside the server
2. No task releases are scheduled to occur inside the server before it will exhaust its remaining budget

Figure 4.4 shows how the example taskset from Figure 4.3 would be scheduled under Dynamic CSF. Note that the idle at time 8 is still present because task $\tau_1$ executes before the budget is fully depleted. However, beyond time 12, the two criteria above are met several times. The most significant of these times is at time 12, where $g_1$ yields its entire budget to $g_2$ so that task $\tau_4$ can execute more quickly. Note that the final release of $\tau_1$ occurs at time 15,
even though $\tau_1$ executes at time 16. This is because the budget for $g_1$ is not replenished until time 16.

Also of significance in Figure 4.4 is that the top timeline, which represents the schedule as seen by the host, is no longer fully loaded as it was in Figure 4.3. The idle times on the top timeline can be used by the host to schedule other processes, or even another real-time server if the added server has a larger period. As was the case in Figure 4.3, the scheduling pattern repeats every hyperperiod.

In this example, each task executes for its entire WCET. [1] demonstrates that often this is a pessimistic assumption. In the case of Dynamic CSF, since job terminations are introspected, the amount of idle time freed when jobs terminate before their WCET becomes even greater. However, Static CSF is unable to free any of this idle time. Therefore, in scenarios where job execution times follow a probabilistic distribution, Dynamic CSF provides even greater benefits over Static CSF.
Chapter 5

Flattening Scheduling

Chapter 4 established that CSF-based approaches are prone to leaving the CPU idle when there are real-time jobs to run. While the Dynamic CSF algorithm was able to reclaim some of this lost execution time the example in Figure 4.4 showed that in some cases, the server is unable to yield its budget when it is idle, because a task release is scheduled to occur within the remaining budget.

This chapter presents a solution which never leaves the processor idle when any guest on the system has real-time jobs ready to execute. The solution works by removing the concept of servers and flattening the virtualization layer such that the host can schedule guest tasks directly. The initial idea for flattening scheduling was presented in [16] without a scheduling algorithm or an implementation. This work also coined the term “flattening”.

The flattening scheduling approach presented by this paper uses EDF as the top-level scheduler in the hierarchy. This is because EDF has been shown to be an optimal algorithm for scheduling tasks on a uniprocessor system. Additionally, using EDF as the host scheduler eliminates the need to assign priorities to each guest, but rather schedule them with no assumption of priority, as specified in the scheduling model, in Section 3.4.

The remainder of the chapter is organized into sections based on adding algorithmic support for different scheduling scenarios. First, the advantages of flattening scheduling over CSF are outlined in Section 5.1. Next, an algorithm is presented which can schedule earliest deadline first guests on a uniprocessor system in Section 5.2. This is followed by Section 5.3, which describes the necessary changes to the algorithm to support guests with fixed-priority schedulers, or a mixture of EDF and fixed-priority guests. Finally, Section 5.6 describes the partitioning approach used by this thesis to schedule guests on multiprocessor hosts.
5.1 Advantages of Flattening Scheduling

Removing the concept of servers and breaking down the divide between the host and guest schedulers comes with many advantages over CSF, including:

- The potential to support any guest scheduler, not just EDF and RM.
- The ability to better adapt to less-predictable guest tasksets, through the ability to gather task information on-the-fly. For example, aperiodic tasks, which only appear once and can appear at any time.
- The accommodation of tasks which do not specify a worst-case execution time. Since the top-level scheduler uses EDF, the only parameter each task is required to report is absolute deadline.
- The lack of any offline calculations to compute a server budget. This also allows guests and tasks to be added and removed on-the-fly at runtime without the recalculation of budgets.
- The reduction of idle CPU time, which directly leads to increased schedulability and performance.
- A reduced number of preemptions between virtual machines at run-time.

5.2 EDF Guests

This work first examines the case where each guest is scheduled using EDF. Since the host also is using EDF, flattening the hierarchy of schedulers is simpler than in the heterogeneous case.

5.2.1 Algorithm Overview

The idea behind flattening scheduling is to remove the idea of virtual machines altogether, and instead simply schedule the entire taskset $\Gamma$ using a single scheduling algorithm.

This work chooses EDF as that single scheduling algorithm, because in the uniprocessor case, EDF is an optimal scheduling algorithm. Additionally, when the guest operating systems are also scheduling using EDF, it is possible to schedule the job with the earliest deadline at any given time. Proof of this fact is given in Theorem 5.1.

**Theorem 5.1.** The active job $\tau^* \in \Gamma$ with the earliest deadline is also the job with the earliest deadline in its virtual machine.
Proof. Assume $\tau^* \in g \subseteq \Gamma$. Let $d^*$ be $\tau^*$'s absolute deadline, and $d_i$ be the absolute deadline of job $\tau_i$. It is known that $d^* \leq d_i$, $\forall \tau_i \in \Gamma$. Since $g \subseteq \Gamma$, we know that every active job $\tau_i \in g$ is also in $\Gamma$. Therefore, $d^* \leq d_i$, $\forall \tau_i \in g$. Thus, $\tau^*$ also has the earliest deadline in its virtual machine, $g$. □

The flattening algorithm for EDF guests consists of simply choosing the job with the earliest deadline and scheduling that job’s VM. Since it is also the earliest deadline job in the VM, the host can be assured that the VM is running the job it chose to schedule.

5.2.2 Schedulability Analysis

In essence, Theorem 5.1 states that EDF-within-EDF hierarchical scheduling is the same as running EDF on the entire taskset $\Gamma$. Because of this, the schedulability analysis reduces to the analysis of EDF, which on uniprocessors is an optimal algorithm with a utilization bound of 1 [28]. Without considering overheads, the flattening algorithm with EDF guests is also an optimal uniprocessor algorithm with a total taskset utilization bound of 1.

Comparison with CSF

Figure 5.1 shows the utilization bound for a single EDF virtual machine scheduled under CSF [2]. The parameter $k$ increases as the server period decreases relative to the minimum period of a task in the server. In essence, $k$ represents a trade-off between theoretical schedulability and the number of preemptions which will occur at runtime. As $k$ is increased, schedulability increases, but so does the number of preemptions.

When $k$ is large for every virtual machine, the utilization bound of CSF is close to 1 regardless of the number of virtual machines. However, if even a single virtual machine has a small $k$ value, the utilization bound can drop significantly. For example, consider a scenario with 2 virtual machines, each with tasksets such that the value of $k$ is 1. If half of the processor time is allocated to each virtual machine (i.e. each server utilization is 0.5), the utilization bound for each VM is just above 0.2, yielding an overall system utilization bound of just above 0.4.

By comparison, flattening yields a utilization bound of 1 regardless of guest tasksets. While it is possible to increase the utilization bound for CSF in the example given above by increasing $k$, this is a trade-off because it increases the frequency of preemptions between servers. Flattening provides an ideal utilization bound for EDF guests, without the need to increase preemptions.
5.3 Supporting Fixed-Priority Guests

Although EDF is an optimal scheduler for uniprocessors, a majority of legacy real-time systems are scheduled using fixed priorities. Therefore, adding algorithmic support for fixed-priority guests vastly increases the number of legacy systems which can be scheduled by the algorithm. Unlike EDF, fixed-priorities introduce heterogeneity into the scheduling hierarchy. This means that a simple, well-studied scheduling algorithm with a known utilization bound does not already exist for this scenario.

5.3.1 Algorithm Overview

Figure 5.2 presents the algorithm devised to support fixed-priority guests under an EDF top-level scheduler. The goal of the algorithm is to minimize deadline misses among all the virtual machines, and as a result no priority relationship is assumed between the virtual machines. Note that a system with multiple guests all using fixed-priority schedulers such as RM does not necessarily produce the same schedule as running RM on all of the tasks would; this is because the top-level scheduler is still EDF. In fact, a system in which every guest runs the RM scheduler with one task per guest will actually result in a pure EDF schedule under this algorithm.
begin
  Select the active job $\tau^j$ with the earliest absolute deadline;
  Let $g$ be the guest which contains $\tau^j$;
  if $g$ is scheduled using EDF then
    Schedule $\tau^j$
  else
    Schedule the highest priority active job on $g$
end

Figure 5.2: Flattening Scheduling Algorithm with Fixed-Priority Guests

5.3.2 Schedulability Analysis

Unlike the pure EDF case, the fixed-priority guests cause the flattening schedule to be suboptimal for the taskset $\Gamma$. For the purposes of this utilization bound, it is assumed that all fixed-priority guests are scheduled with the rate-monotonic scheduler, since it is an optimal fixed-priority scheduler for the task model used in this thesis. The worst-case schedule for the flattening algorithm occurs when all of the tasks are in a single virtual machine, scheduled using RM. In this case, the algorithm actually reduces to RM, and has the utilization bound given in Equation 5.1 [28].

$$U \leq n(2^{1/n} - 1)$$ (5.1)

While this equation represents the utilization bound for the KairosVM flattening algorithm, it is a poor schedulability test for the algorithm. In cases where one guest is scheduled with EDF, or even in cases where there are multiple guests all scheduled with RM, the actual utilization bound is higher than the bound for RM on the taskset.

Comparison with CSF

Figure 5.3 shows the utilization bound for a single RM virtual machine scheduled under CSF [2]. The parameter $k$ is the same as before, and again offers a trade-off between preemptions and schedulability.

Also similar to the EDF case, flattening scheduling maintains the ideal utilization bound. Instead of 1, however, this time that bound is given by equation 5.1. In the same scenario as before, if 2 virtual machines each scheduled under RM have $k = 1$, and the processor is divided equally among the two servers, CSF has an overall utilization bound of around 0.4, as compared to 0.69 for flattening. Again, $k$ can be increased in order to approach a bound of 0.69 for CSF, but this comes at the cost of more preemptions at runtime because of a
reduced server period. With both EDF guests, and RM guests, flattening provides the ideal utilization bound without increasing the number of preemptions.

### 5.4 Schedulability Test for the KairosVM Flattening Algorithm

In an effort to demonstrate the KairosVM Flattening Algorithm’s superiority over both RM and server-based approaches to hierarchical scheduling, a schedulability test for the algorithm was developed based on response-time analysis [28]. The theory presented below is built upon the response-time analysis techniques developed by Spuri [31] for EDF scheduling, and the notation used below is based upon the notation used in [17], a response-time analysis of EDF within fixed-priorities. First, the contribution of tasks scheduled using EDF to a task’s worst-case response time is examined. This is followed by the contribution of tasks in the same virtual machine with a higher priority to the task’s response time. Finally, the contribution of tasks scheduled in other fixed-priority virtual machines to a task’s worst-case response time is examined. Table 5.1 provides a summary of the notation used in the following sections.
Table 5.1: Summary of Response-time Analysis Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_a$</td>
<td>Task currently under response-time analysis</td>
</tr>
<tr>
<td>$t$</td>
<td>Length of the busy period.</td>
</tr>
<tr>
<td>$p$</td>
<td>Index of task release (first release at $p = 0$).</td>
</tr>
<tr>
<td>$T_i$</td>
<td>Period of task $\tau_i$. Note that the deadline $D_i = T_i$.</td>
</tr>
<tr>
<td>$D$</td>
<td>Deadline of the task under analysis, $\tau_a$.</td>
</tr>
<tr>
<td>$C_i$</td>
<td>Worst-case execution time of task $\tau_i$.</td>
</tr>
<tr>
<td>$W_i$</td>
<td>Worst-case contribution of task $\tau_i$ to the busy period.</td>
</tr>
<tr>
<td>$P_i$</td>
<td>Priority of task $\tau_i$.</td>
</tr>
<tr>
<td>$V_{\tau_i}$</td>
<td>The virtual machine which contains task $\tau_i$.</td>
</tr>
<tr>
<td>$D'_i$</td>
<td>The closest deadline in $V_{\tau_i}$ to $D$ such that $D'_i \leq D$ and the deadline belongs to a task with a priority less than or equal to $P_i$.</td>
</tr>
<tr>
<td>$\Psi$</td>
<td>The set of all release times for $\tau_a$ which can possibly lead to the worst-case response time.</td>
</tr>
<tr>
<td>$\Psi^*$</td>
<td>The subset of all the release times, $\Psi$, which must be checked for a given activation $p$.</td>
</tr>
<tr>
<td>$\Psi_x$</td>
<td>A single release time within the set $\Psi^*$.</td>
</tr>
<tr>
<td>$L$</td>
<td>Length of the longest possible busy period for the taskset.</td>
</tr>
<tr>
<td>$A$</td>
<td>Offset from the start of the busy period at which the first release of $\tau_a$ occurs.</td>
</tr>
<tr>
<td>$w^A_a(p)$</td>
<td>Completion time of activation $p$ of task $\tau_a$, when the first release occurs with offset $A$.</td>
</tr>
<tr>
<td>$D^A_a(p)$</td>
<td>Deadline of activation $p$ of task $\tau_a$, when the first release occurs with offset $A$.</td>
</tr>
<tr>
<td>$R^A_a(p)$</td>
<td>Worst-case response time for release $p$ of task $\tau_a$ when the first activation occurs at offset $A$.</td>
</tr>
<tr>
<td>$R_a$</td>
<td>Worst-case response time for task $\tau_a$.</td>
</tr>
</tbody>
</table>

### 5.4.1 Interference from Tasks Scheduled under EDF

The following analysis uses the notion of a “busy period”, in which the processor is never idle. The busy period is assumed to begin at time 0, and to continue until a processor idle period is reached. The busy period is assumed to have length $t$, and maximum length $L$.

**Theorem 5.2** (Spuri). The worst-case response time of a task $\tau_a$ is found during a busy period in which all other tasks are released simultaneously at the beginning of the busy period.

**Proof.** Let $t_0$ be the instance at which an interfering task $\tau_i$ is released for the first time during the busy period. Let $D$ denote the deadline of the analyzed task, $\tau_a$, relative to the beginning of the busy period. Suppose $t_0$ does not coincide with the beginning of the busy
period. Then moving $t_0$ back towards the busy period can only increase the response time for $\tau_a$, because it will have the effect of either adding new releases into the busy period for the interfering task $\tau_i$, or moving the deadline of activations of $\tau_i$ backward such that they are earlier than $D$, thus causing the activations to interfere with $\tau_a$. Since moving $t_0$ backwards can only increase the response time of $\tau_a$, it follows that the maximum response time for $\tau_a$ occurs when $t_0$ coincides with the start of the busy period for all interfering tasks.

Theorem 5.2 provides the conditions necessary to calculate the worst-case contribution of an interfering task $\tau_i$ to the response time of the analyzed task $\tau_a$. Let the busy period be of length $t$. Then, only the activations of $\tau_i$ that fall in the interval $[0, t)$ contribute to the worst-case response time of $\tau_a$. However, only activations with a deadline earlier than $D$ should be considered, since these tasks are scheduled using EDF.

Let each activation be identified by the sequence number $p$. Then the sequence number of interest identifies the last activation which contributes to the worst-case response time of $\tau_a$. The number of activations of task $\tau_i$ in the busy period is given by

$$p_t = \left\lceil \frac{t}{T_i} \right\rceil \quad (5.2)$$

Similarly, since we assume that deadlines are equal to periods, the number of activations of task $\tau_i$ with deadlines before $D$ is given by

$$p_D = \left\lfloor \frac{D}{T_i} \right\rfloor \quad (5.3)$$

By combining Equations 5.2 and 5.3, the worst-case contribution of an EDF task $\tau_i$ to the busy period is:

$$W_{i}^{EDF}(t, D) = \min\left(\left\lceil \frac{t}{T_i} \right\rceil, \left\lfloor \frac{D}{T_i} \right\rfloor\right) \cdot C_i \quad (5.4)$$

Equations 5.2, 5.3, and 5.4 are adapted from [17]. Next, the worst-case contribution of a fixed-priority task from the same virtual machine is examined.

### 5.4.2 Interference from Higher Priority Tasks

Naturally, if the task under analysis, $\tau_a$ is in an EDF VM (i.e. $V_{\tau_a} \in V_{EDF}$), the contribution of other tasks $\tau_i \in V_{\tau_a}$ can be calculated using the equation for $W_{i}^{EDF}(t, D)$. However, if the task $\tau_a$ is in an RM VM (i.e. $V_{\tau_a} \in V_{RM}$), the worst case contribution of other tasks has nothing to do with deadlines and everything to do with priority.
Theorem 5.3. The worst-case contribution of a task $\tau_i \neq \tau_a$, $\tau_i \in V_{\tau_a} \in V_{RM}$ to the busy period is equal to the number of releases of $\tau_i$ in the busy period multiplied by $C_i$ if $P_i \geq P_a$, and equal to 0 if $P_i < P_a$.

Proof. Because guest schedulers are honored, it follows that every release of $\tau_i$ will preempt $\tau_a$ for its full execution time $C_i$ if $P_i \geq P_a$. Similarly, no task with a lower priority than $P_a$ will preempt $\tau_a$’s execution, thus lower priority tasks do not contribute to the busy period.

Given the results of Theorem 5.3, the contribution of tasks in $V_{\tau_a} \in V_{RM}$ with a higher priority than $P_a$ can be calculated using the number of releases during the busy period, which is given by Equation 5.2. This leads to the following equation for the worst-case contribution of a task in the same VM with a higher priority, also adapted from [17]:

$$W_{Hi}(t) = \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \quad (5.5)$$

5.4.3 Interference from Fixed-Priority Tasks in other Virtual Machines

The final set of tasks for which worst-case contribution must be computed is the set of tasks which reside in a fixed-priority virtual machine which is not the same machine in which $\tau_a$ resides. In other terms, the set of tasks $\tau_i \in V_{RM} \setminus V_{\tau_a}$.

Unlike tasks in the same fixed-priority VM as $\tau_a$, there is no priority relationship between these tasks and $\tau_a$. However, it is also not as simple as the EDF case, because $\tau_i$ has the capability to contribute to the busy period even if its relative deadline $d_i$ falls after $D$ when another task in $V_{\tau_i}$ has a deadline earlier than $D$.

To obtain the number of releases of $\tau_i$ which contribute to $\tau_a$, the term $D'(D)$ is defined as follows:

$$D'(D) = \max(\forall \tau_j \in V_{\tau_i}, P_j \leq P_i)\left\lfloor \frac{D}{T_j} \right\rfloor T_j \quad (5.6)$$

Figure 5.4 represents what the value of $D'$ means graphically. Let $d_j$ represent the relative deadline of $\tau_j \in V_{\tau_i}$, where a lower value of $j$ represents a lower priority. Assume $\tau_i$’s priority is 4. Finally, let $D$ represent the absolute deadline of the task under analysis, $\tau_a$.

$D'$ represents the latest lower-priority deadline in $V_{\tau_i}$ that still falls before $D$. In Figure 5.4a, note that the last release of $\tau_i$, which in this case occurs at the first $d_4$, has its deadline well beyond the deadline $D$. Thus, if scheduled using EDF, this release would not contribute to the busy period. However, since $V_{\tau_i}$ is scheduled using a fixed-priority algorithm, this last
release can still contribute to the busy period if a lower priority task in $V_{\tau_i}$ is active and has an earlier deadline than $D$. In this case, $\tau_i$ would preempt the lower priority task and execute, even though its deadline falls after $D$. This is the case in Figure 5.4a, where the lower-priority task $\tau_2$ has its deadline before $D$ but after the last release of $\tau_i$. Even though $d_5$ falls closer to $D$ than $d_2$, since $\tau_5$ has a higher priority than $\tau_i$, $\tau_i$ will not preempt $\tau_5$ and thus $d_5$ is not considered in the calculation of $D'_i$.

Figure 5.4b shows the case where the closest deadline in $V_{\tau_i}$ to $D$ belongs to $\tau_i$ itself (the period of $\tau_i$ has been divided by 2). In this case, the interference from $\tau_i$ on $\tau_a$ is the same as if $\tau_i$ was scheduled using EDF, since it gets no deadline advantage from lower priority tasks on its VM.

**Theorem 5.4.** Every release of a task $\tau_i \in V_{RM}, V_{\tau_i} \neq V_{\tau_a}$ which occurs strictly before $D'_i$ contributes to the busy period for $\tau_a$.

**Proof.** According to the flattening scheduling algorithm, $\tau_i \in V_{\tau_i} \in V_{RM}$ is scheduled when any task on $V_i$ has the earliest overall deadline, and $\tau_i$ is the highest priority active task in $V_{\tau_i}$. Therefore, for an activation of $\tau_i$ to preempt $\tau_a$, it is required that $V_{\tau_i}$ contains a task $\tau_j$ with priority $P_j \leq P_i$ and deadline $d_j \leq D$. Note that it is possible that $\tau_j \equiv \tau_i$. Since $D'_i$ is equal to the latest absolute deadline that falls no later than $D \forall \tau_j$, releases of $\tau_i$ that fall
before $D'_i$ will contribute to the busy period, while releases that fall no earlier than $D'_i$ will not preempt because there will be no lower priority task with a deadline between $D'_i$ and $D$. Therefore, it follows that every release of $\tau_i$ that falls strictly before $D'_i$ contributes to the busy period.

With the result of Theorem 5.4, it is possible to define the worst-case contribution of a task $\tau_i$ which belongs to different fixed-priority guest from $\tau_a$:

$$W_{RM}^i(t, D) = \min\left(\left\lceil\frac{t}{T_i}\right\rceil, \left\lceil\frac{D'_i(D)}{T_i}\right\rceil\right) \cdot C_i$$  \hspace{1cm} (5.7)

Note that when $D'_i(D)$ coincides with $D_i$, Equation 5.7 reduces to Equation 5.4. This is because tasks in other VMs provide the same interference as EDF tasks except in special cases such as the one shown in Figure 5.4a.

### 5.4.4 Calculating the Worst-Case Response Time of a Task

To lighten the notation in this section, this thesis defines the following sets in Table 5.2:

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{EDF}$</td>
<td>$\bigcup V_i, \forall i</td>
<td>V_i \in V_{EDF}$</td>
</tr>
<tr>
<td>$S_{HI}$</td>
<td>$\bigcup {\tau_i}, \forall \tau_i \in V_{\tau_a}, P_i \geq P_a$</td>
<td>The set of all tasks in the same guest as $\tau_a$ with a higher priority than $\tau_a$</td>
</tr>
<tr>
<td>$S_{RM}$</td>
<td>$\bigcup V_i, \forall i</td>
<td>V_i \in V_{RM}, V_i \neq V_{\tau_a}$</td>
</tr>
</tbody>
</table>

With the worst-case contribution of fixed-priority tasks in other VMs accounted for, the following analysis is adapted directly from Harbour [17]. As demonstrated by Spuri [31], the release time of the task under analysis $\tau_a$ is not necessarily the beginning of the busy period. It is possible that the first release of $\tau_a$ which leads to the worst-case response time occurs at an offset from the start of the busy period, such that a future release shares its deadline with another task in the system. Thus, response-time analysis for $\tau_a$ must examine release times which coincide with other task deadlines, $pT_i$. The set of all of these release times which must be examined, $\Psi$, is given below:

$$\Psi = \bigcup \{p \cdot T_i\}$$

$$\forall p = 1 \ldots \left\lceil\frac{L}{T_i}\right\rceil, \forall i \in S_{RM} \cup S_{EDF} \cup S_{HI}$$  \hspace{1cm} (5.8)
In Equation 5.8, \( L \) refers to the longest busy period, which is calculated recursively using the following equation:

\[
L = \sum_{\forall i \in S_{RM} \cup S_{EDF} \cup S_{HI}} \left\lfloor \frac{L}{T_i} \right\rfloor \cdot C_i \tag{5.9}
\]

The critical release for \( \tau_a \) is found by subtracting \( T_a \) from each value in the set \( \Psi \). Since \( \tau_a \) may have multiple activations within the busy period, every activation must be analyzed for worst-case response time. If the first activation of \( \tau_a \) within the busy period occurs at time \( A \), and the busy period begins at time 0, the completion time of activation \( p \) of \( \tau_a \) can be calculated by the following equation:

\[
w_a^A(p) = p \cdot C_a + \sum_{\forall \tau_i \in S_{HI}} W_i^{HI}(w_a^A(p)) + \sum_{\forall \tau_i \in S_{RM}} W_i^{RM}(w_a^A(p), D_a^A(p)) + \sum_{\forall \tau_i \in S_{EDF}} W_i^{EDF}(w_a^A(p), D_a^A(p)) \tag{5.10}
\]

Equation 5.10 combines the worst-case contributions from Equations 5.5, 5.7, and 5.4, as well as the contribution from \( \tau_a \) itself. The term \( D_a^A(p) \) refers to the deadline of activation \( p \) when the first activation occurs at \( A \):

\[
D_a^A(p) = A + p \cdot T_a \tag{5.11}
\]

Finally, the worst-case response time for task \( \tau_a \) is calculated by subtracting the activation time of iteration release \( p \) from the completion time:

\[
R_a^A(p) = w_a^A(p) - A - (p - 1)T_a \tag{5.12}
\]

It is only necessary to check values of \( A \) which fall between 0 and \( T_a \), since the first release must fall within this range. Thus, it is only necessary to check the values of \( \Psi \) in the subset:

\[
\Psi^* = \{ \Psi_x \in \Psi | p \cdot T_a \leq \Psi_x < (p + 1)T_a \} \tag{5.13}
\]

Thus, we define the set of \( A \) values to check as:

\[
A(\Psi_x) = \Psi_x - (p \cdot T_a) \tag{5.14}
\]

Figure 5.5 demonstrates the graphical meaning of the different deadlines to analyze for task \( \tau_a \), which this thesis denotes \( \Psi_x \). The top timeline begins at the start of the busy period.
Figure 5.5: Graphical Representation of $A$ and $\Psi_x$

(Time 0) and ends with the $p^{th}$ release of $\tau_a$. The value of $A$ represents an offset from the start of the busy period. The first release of task $\tau_a$ occurs at $A$, and subsequent releases occur at $A + pT_a$ for integer values of $p$. Clearly, the value of $A$ must fall between 0 and $T_a$, since $A$ is defined as the first release of $\tau_a$ in the busy period and if $A > T_a$, there would be another release of $\tau_a$ which falls between 0 and $T_a$ and thus is the first release.

The second timeline in Figure 5.5 demonstrates how the values of $A$ to iterate over are calculated. Each $\Psi_x$ on this timeline represents an absolute deadline of an interfering task in the system (or $\tau_a$ itself) that falls between $pT_a$ and $(p + 1)T_a$. Each of these values of $\Psi_x$ is offset from $pT_a$ by the same offset of the first release, $A$. Only values of $A$ which correspond with $\Psi_x$ need to be checked for the $p^{th}$ release of $\tau_a$ because having $\tau_a$'s deadline coincide with the deadline of an interfering task provides a (local) maximum of interference on $\tau_a$.

Finally, the absolute worst-case response time for the task $\tau_a$, denoted $R_a$, can be calculated:

$$R_a = \max(R^A_a(p))$$

$$\forall p = 1 \ldots \lceil \frac{L}{T_a} \rceil, \forall A(\Psi_x) | \Psi_x \in \Psi^*$$

Thus, a taskset is schedulable by the KairosVM Flattening Algorithm if the worst-case response time for every task is less than the relative deadline for the task. In other terms, the taskset is schedulable if:

$$R_i \leq D_i, \forall i$$
5.5 Comparison with Server-based Approach

Figure 5.6 shows the percentage of schedulable tasksets for both CSF and the KairosVM Flattening Scheduling Algorithm over a range of taskset utilizations. In the figure, CSF-50 denotes CSF with a server period of 50, while CSF-100 denotes a server period of 100, and so on. Each taskset contains 6 individual tasks, divided randomly among 3 VMs. Two of the VMs are scheduled using RM, while the third is scheduled using EDF. The utilization of each task is computed randomly to be between 0.01 and 0.99 using a uniform distribution. This is done via the UUniFast algorithm from Bini and Buttazzo [32]. The period of each task is chosen randomly as a multiple of 100 between 100 and 1000. The global utilization of the taskset is set between 0.1 and 0.95. 200 tasksets were generated for each utilization.

Figure 5.7 uses a similar setup as Figure 5.6, except it uses 15 randomly generated tasks spread across 6 VMs, and 500 random tasksets were generated for each data point. Additionally, each subfigure provides results for a different ratio of EDF and RM guests. As expected, increasing the proportion of guests scheduled using EDF improves schedulability for each variation of CSF. KairosVM performs even better than it did in Figure 5.6 be-
cause increasing the number of virtual machines actually transforms the taskset closer to a deadline-scheduled one, since the top-level scheduler for KairosVM is EDF. For CSF, the opposite is true; adding more virtual machines significantly decreases schedulability because each VM is an additional server, and every server must reserve more CPU time than its taskset utilization in order to guarantee all deadlines are met.

Clearly, the KairosVM Flattening Scheduling Algorithm is able to schedule the largest proportion of tasksets, regardless of utilization. As expected, CSF performs better with a smaller server period; however, this a smaller period comes with the trade-off of a higher number of virtual machine preemptions. Even though preemption overhead is not considered in this simulation, it is a concern at runtime.
5.5.1 Examples

Even with every guest using EDF, CSF is unable to provide a utilization bound of 1 unless the server period is reduced to an infinitesimally small length. Unfortunately, reducing the period of a server increases the number of virtual machine preemptions, which in practice do not have zero cost.

Even with a relatively large period, CSF incurs more preemptions than an EDF schedule. Consider Figure 5.8, which schedules the same taskset as Figures 4.3 and 4.4 in Chapter 4. The flattening schedule only produces 6 preemptions between $g_1$ and $g_2$, while the CSF schedules produce 9 preemptions each (the Dynamic CSF schedule must preempt $g_2$ at time 12 in order to determine $g_1$ is idle, then preempt $g_1$ in order to reschedule $g_2$).

Beyond preemptions, Figure 5.8 also demonstrates that flattening scheduling ensures that the processor is never idle when there are real-time jobs ready to execute. While Dynamic CSF improved processor usage over Static CSF, Figure 5.8 shows that flattening scheduling best utilizes CPU resources.

Figure 5.9 demonstrates the schedules generated for an example taskset scheduled by global EDF, global RM, and the KairosVM flattening algorithm. In this example, the two global algorithms schedule without the concept of guests with their own schedulers, while the flattening algorithm must honor guest schedulers. Also of note is that this taskset is unschedulable using CSF.

Note that in terms of schedulability, the global EDF schedule is optimal, while the flattening schedule is less optimal than EDF but more optimal than RM. This relationship manifests
itself in a deadline miss for $\tau_3$ in the RM schedule. The EDF schedule is able to avoid the deadline miss by delaying the executions of $\tau_1$ and $\tau_2$ until $\tau_3$ terminates its first job. Similarly, the flattening schedule is able to delay the execution of $\tau_1$ in order to ensure $\tau_3$ meets its deadline. However, because the flattening schedule must respect guest 2’s priority ordering, it is unable to delay execution of $\tau_2$.

### 5.6 Multiprocessor Scheduling

In order to benefit from the uniprocessor theory in Section 5.4, this thesis uses partitioning to extend the KairosVM Flattening Scheduling Algorithm to multiprocessor host systems with $m$ processors.

#### 5.6.1 Overview of Partitioning Algorithm

The first-fit heuristic is used to partition virtual machines among different processors. First-fit is chosen because it is a heuristic with a low online complexity, and provides similar worst-case and average-case performance as other algorithms such as best-fit [33].

Given a virtual machine, the algorithm works by assigning that virtual machine to the first processor whose taskset does not become unschedulable upon adding the new virtual machine. The schedulability test from Section 5.4 is used to verify schedulability on a
processor. If a virtual machine is unable to fit on any of the \( m \) processors, the taskset is deemed unschedulable. Otherwise, the taskset is schedulable.

Even if a taskset is deemed unschedulable, it is possible for the KairosVM Flattening Scheduling Algorithm to attempt to schedule the taskset. In this case, the multiprocessor partitioning algorithm switches to a worst-fit heuristic, where the processor with the lowest utilization is chosen for each VM.

### 5.6.2 Schedulability Analysis

When all guests are scheduled using EDF, the partitioning mapping reduces to the well-studied bin-packing problem. It has been shown that the first-fit algorithm described above has the worst-case utilization bound given in Equation 5.17 below [34].

\[
\frac{m + 1}{2} \quad (5.17)
\]

However, this worst-case bound is pessimistic. In practice, the achievable utilization on an \( m \) processor system increases as the number of tasks and the number of processors increase, as demonstrated by Lopez et al. in [34].
Chapter 6

Implementation

This chapter describes the methods and data structures used to implement the ideas and algorithms laid forth in Chapter 5. In addition, the solutions to engineering problems, and the implementations of engineering optimizations are described. All modifications to Linux, as well as all Linux kernel modules created for this thesis were based on Linux kernel version 3.18.0.

6.1 Implementation of Scheduling Algorithms in the Linux Kernel

The implementation of the algorithms in Chapters 4 and 5 is split into separate Linux kernel modules. The csf- modules provide introspection into the virtual machines, but do not use the introspection to schedule. Rather, the introspection is only used to report statistics about the real-time tasks in the virtual machines. The dcsf- modules implement Dynamic CSF, described in Chapter 4, while the kairosvm- modules implement the KairosVM Flattening Algorithm from Chapter 5. Each of the modules depends on a modified version of the kvm kernel module, and replaces the kvm-intel or kvm-amd kernel module included in vanilla Linux. A full list of the modules and their descriptions is provided in Table 6.1.

Several modifications to Linux were necessary to maintain the real-time guarantees of the guest operating systems without using servers. Chief among these modifications is breaking the assumption that each task_struct can only be representative of a single real-time task with a single period, deadline, and execution time. This modification is described in detail in Section 6.1.1.

The Linux scheduler is also one of the most complicated portions of the Linux kernel, so naturally several engineering mechanisms were necessary to ensure that the new modified kernel is stable and usable. One such mechanism is ensuring that non-real-time tasks on
guest operating systems also receive some execution time in the event that no real-time task is executing. While it might not seem like an important thing to implement, the non-real-time tasks are actually critical to the proper operation of the virtual machine to allow for basic applications such as `ssh` to work, giving the user a way to administrate the VM. This implementation is described in Section 6.1.2.

One of the benefits that introspection provides over the server approach is that the host can detect when jobs terminate. A discussion of these benefits and how they are achieved is provided in Section 6.1.3. Finally, a discussion of the modifications necessary to support fixed-priority guests in the same system as EDF guests is provided in Section 6.1.4.

### 6.1.1 Scheduling Entities

The Linux scheduler provides four scheduling classes for tasks running on the system. By default, tasks fall under the *fair* scheduling class, which schedules processes and threads on the system in a fair manner. The work of this thesis focuses on another scheduling class in Linux called SCHED_DEADLINE [3], which has been part of the Linux kernel since version 3.14. SCHED_DEADLINE provides an earliest-deadline-first real-time scheduler for the Linux kernel. Each scheduling class implements its scheduling algorithm using a concept known as scheduling entities.

Every `task_struct` contains a single scheduling entity for each scheduling class. As shown in Figure 6.1, this entity is statically allocated inside the `task_struct`. The entity provides information to the Linux scheduler about the task which is relevant to the particular scheduling class. For example, a task’s deadline entity provides the relative period, relative deadline, execution time, absolute deadline, and remaining budget to the Linux scheduler. If the task’s scheduling class is set to SCHED_DEADLINE, the Linux scheduler then uses the `task_struct`’s deadline entity information to schedule the task.
Figure 6.1: Deadline Scheduling Entities in Vanilla Linux

SCHED_DEADLINE uses an ordered red-black tree to determine the scheduling entity with the earliest absolute deadline at any given time. A representation of this tree is given in Figure 6.1. There is one red-black tree per runqueue, and one runqueue per physical CPU. The left-most node on the tree represents the entity with the earliest deadline, while the right-most node represents the entity with the latest deadline. When a scheduling event occurs and the current running task is marked to be rescheduled, the pick_next_task_dl function selects the left-most node in the red-black tree and returns the task_struct which contains that node to the scheduler for scheduling, while removing the previous task from the runqueue.

Assumptions made by SCHED_DEADLINE and Linux

The SCHED_DEADLINE implementation in Linux makes a few assumptions which are broken in this project. First and foremost is the assumption that each task in Linux has at most one period, deadline, and execution time associated with it. In Linux, each vCPU is seen as a single thread or task_struct. However, a single virtual machine can potentially contain many different real-time tasks, each with their own period, deadline, and execution time. In other words, the vCPU task_struct has multiple periods, deadlines, and execution times associated with it. Clearly, this is a violation of the assumption that a single task only has one period, deadline, and execution time.

A second assumption made by the SCHED_DEADLINE implementation is that when a
scheduling entity has used all of its available budget, the task can be safely removed from the runqueue. However, if a task has multiple periods, deadlines, and budgets, this assumption is violated. Just because one entity associated with the task has exhausted its budget doesn’t mean all of the budgets for this task have been exhausted. Furthermore, a vCPU task is a special case in that it also has non-real-time components to process at any given time, and therefore should only be removed from the runqueue in the event that other equal-or-higher priority processes need the processor for execution time.

Memory Management

The first major hurdle to modify the scheduler to support real-time vCPU tasks was the way that memory is managed for deadline scheduling entities. In the vanilla Linux kernel, a single deadline entity is allocated for each task directly within the task_struct. However, static allocations will not work when the number of associated deadline entities is not known until runtime. Therefore, for this project, the statically allocated deadline entity was converted into a dynamically allocated linked list.

The biggest fallout with this conversion came in the form of memory management, as an entity now had to be dynamically created every time a task entered the SCHED_DEADLINE class. To minimize the changes to non-vCPU tasks in the kernel, it was decided that each task, regardless of scheduling class, should have an initial deadline entity which behaves similarly to the statically allocated one. This entity is allocated when the task is initially forked, and deallocated when the task exits. Additionally, this entity acts as the head of the linked list.
The modified layout of scheduling entities in memory is shown in Figure 6.2. As in Figure 6.1, each orange box represents a scheduling entity with all of the associated real-time task information. The main change in the organization for Kairos is that each task struct now contains a list of scheduling entities. The entities themselves are still scheduled by the red-black tree in the same manner. The dl pointer in the task struct now points to the entity that would be executed if the task struct were to run. In the case of EDF guests, this is the entity with the earliest deadline.

Additional entities are added with a new kernel function called sched_add_dl_entity, and removed by a kernel function called sched_rm_dl_entity. These functions key the entities by the PID of the associated real-time task on the guest. The initial entity always has a key of 0. A call to sched_add_dl_entity automatically converts the task into the SCHED_DEADLINE scheduling class. When a task exits or is killed, the entire list of entities except for the head is freed from memory.

**Changes to SCHED_DEADLINE Task Accounting**

Naturally, altering the task struct structure to contain a list of entities instead of just one was just a minor step towards scheduling real-time guests. Since SCHED_DEADLINE previously assumed each task only had one scheduling entity, it was able to deduct from that single entity’s budget every time the task ran. Now that there are multiple entities, it is important to deduct from the budget of the correct entity whenever a task runs.

This is accomplished by setting a pointer in the task struct to point to the scheduling entity associated with the task that is currently executing in the guest. In the case of earliest-deadline-first guests, this task is equal to the task with the earliest absolute deadline. Therefore, deducting from the correct budget is as simple as setting the current pointer to the left-most entity in the red-black tree.

In vanilla Linux, deadline entities are replenished at each job release. This is accomplished by a timer in the kernel which is started whenever an entity’s budget goes to 0. The timer is set to go off at the next release point for the task, and the task is put to sleep until the timer callback function wakes it up and puts it back onto the runqueue. In Kairos, it is important that the entity is removed from the red-black tree when its budget is exceeded, but the task cannot be put to sleep because the vCPU may have other entities with remaining budget. Furthermore, a more accurate task release time is available in the form of the begin_rt_seg system call.

Upon a begin_rt_seg system call, a flag (dl_begin) is set in the associated scheduling entity which tells the scheduler that the system call has occurred. Additionally, SCHED_DEADLINE uses a flag called dl_throttled for each scheduling entity which remains set until the timer callback function is called. In the Kairos implementation, a call to enqueue_task_dl only enqueues entities which are not throttled. Additionally, a scheduling entity is not enqueued
until a begin<rtseg> event is trapped for the associated guest task. This ensures that the budget of the correct entity is depleted at any given time, which is essential to the proper execution of the flattening scheduling algorithm.

Introspection also provides the added benefit of knowing when a guest real-time job terminates via the end<rtseg> system call. Upon this introspection event, it is safe to reduce the currently executing task’s budget to 0, remove it from the red-black tree, and start the replenishment timer. The reduction of the budget to 0 is accomplished by the sched_yield_task function described in Section 6.1.3.

6.1.2 Adding Support for Non-Real-Time Tasks on Guests

Initially, it isn’t obvious why it is necessary to provide execution time to non-real-time tasks. After all, the real-time metrics of interest such as deadline misses, dsr, and number of jobs are exactly the same whether the VM receives execution time for these processes and threads or not. However, as mentioned in Section 6.1.1, the vCPU process is put to sleep when its real-time budget is exhausted. Furthermore, it is impossible to access the VM in any way while the vCPU process is sleeping, which means it is impossible to launch additional instances of real-time applications on the guest. Clearly, this is an engineering problem which needed to be solved in order to run experiments.

Implementation of “Fair” Entities

An idea of how to solve the above problem was to designate the original deadline entity as a “fair” entity, meaning it represents the tasks on the guest which belong to the fair scheduling class. Since the “fair” entity is actually a deadline entity in the host, it is scheduled like the other entities, using the SCHED_DEADLINE scheduling class. This means that the earliest deadline is scheduled first. However, the “fair” tasks on the guest do not have any deadlines; they should simply be scheduled at a lower priority than the real-time tasks.

To accomplish this, the budget of the “fair” entity should be set to an arbitrary value greater than zero, so that the budget never runs out. Additionally, the “fair” entity should have a later absolute deadline than any actual deadline entity, so that it does not get scheduled unless there is no pending real-time job to execute. Finally, a method of forcing the “fair” entities to share execution time with each other needed to be devised, so that every VM on a single physical CPU is able to be accessed at any time.

The design goals in the preceding paragraph were achieved by marking the initial deadline entity as “fair” using a Boolean flag named is_fair, whenever a call to sched_add_dl_entity is made. Additionally, the budget for the entity is set to 100 million nanoseconds, and the absolute deadline is set to S64_MAX, the maximum value for the 64-bit counter used to store the deadline in a scheduling entity. The is_fair flag is used to ensure that the budget of
the “fair” entity is never reduced at any point in the kernel.

In order to force “fair” entities to share available execution time with each other, the absolute deadline is set to \( S64\text{MAX} - 1 \) whenever the entity is not currently executing, and it is set to \( S64\text{MAX} \) whenever it is chosen to be the currently running entity. This causes the non-running entity to have an earlier “deadline” than the currently executing “fair” entity, but a later deadline than any actual real-time task. This way, if only two “fair” entities exist, they alternate between each other at each scheduling quantum.

### 6.1.3 The “sched\_yield\_task” Function

In order to implement the idea presented in Chapter 4, it was necessary to create a function which forced a task struct to yield processor time to other processes in the system until the next task release. As it turns out, Linux already includes a system call named sched\_yield which for deadline tasks reduces the remaining budget to zero and kicks off the replenishment timer. The only issue with sched\_yield is that it must be called from the task struct itself, which in this case is a QEMU vCPU thread.

The dynamic CSF idea from Chapter 4 requires the yield call to take place from the end\_rt\_seg event handling context. Additionally, the yield has to take place on the correct vCPU task struct. For this purpose, the sched\_yield\_task function was created. This function operates in a similar manner to sched\_yield, but it takes the task struct as an argument and does not make any function calls which are inappropriate from the interrupt handling context.

The sched\_yield\_task function is called for dynamic CSF whenever a vCPU has no currently executing tasks (i.e. every most recent begin\_rt\_seg for the vCPU has a matching end\_rt\_seg), and the next release of every task (i.e. the most recent begin\_rt\_seg plus the task period) falls at a point after the vCPU budget runs out. When both of these cases are true, it means the vCPU task has no actual real-time work to do during the remainder of the budget for this release, and it should yield execution to other real-time tasks either on the host or on some other guest. These two cases are evaluated on every end\_rt\_seg system call introspection event.

While sched\_yield\_task was originally created for the purpose of implementing dynamic CSF, it provides an opportunity for optimization beyond this algorithm. Because of introspection, the flattening algorithm is able to switch to an entity with remaining budget as soon as the currently executing job encounters a end\_rt\_seg system call. Unlike in static CSF, this means that a vCPU process which has no real-time tasks to run will never execute on the processor when another vCPU has budget remaining.

Besides increased use of the physical CPU, the sched\_yield\_task optimization also removes the requirement for guest tasks to specify their execution times, which is something neither CSF solution can do. If an execution time is unspecified, it can simply be set equal to
the deadline. If the guest task finishes early, the remaining budget will be depleted by `sched_yield_task` when `end_rt_seg` is encountered. In the CSF solution, an unspecified execution time can’t simply be set to the deadline, as this would lead to a utilization greater than 1 in the case where deadline is equal to period and there is at least one more task on the system. The flattening solution is a completely dynamic solution, so it is able to give a best-effort schedule in the case where utilization is greater than 1.

### 6.1.4 Adding Support for RM Guests

Since SCHED_DEADLINE is designed as an earliest deadline first scheduler, it was inherently easier to support EDF guests, since the global earliest deadline is guaranteed to be the local earliest deadline for at least one VM. However, in the case of fixed-priority guests, the task which a VM will execute is not necessarily the task on that VM with the earliest deadline. In cases like these, it is important to set the currently executing entity for the VM to the correct entity in order to deplete the budget associated with the task which actually executes on the guest.

The implementation approach for guests scheduled with the rate-monotonic (RM) or deadline-monotonic (DM) scheduling algorithms is similar to the approach taken by SCHED_DEADLINE for EDF tasks. The scheduling entities for a guest which has been marked with the `dm` flag are added to an ordered linked-list whose head pointer is stored in the `task_struct` data structure. This linked-list is analogous to the red-black tree used by SCHED_DEADLINE, but rather than being ordered by absolute deadline, the list is ordered by relative deadline.
in increasing order, such that the first element in the list has the smallest relative deadline. In other words, the first element of the list has the highest priority when scheduling using the DM algorithm.

Whenever the \texttt{is dm} flag is set for a \texttt{task struct}, all of the process’s scheduling entities which have budget remaining are added to both the per-CPU red-black tree for SCHED_DEADLINE, and also the DM linked-list for the \texttt{task struct}. Then, when \texttt{pick next task dl} is called, the left-most node in the red-black tree is selected as usual. However, if this node belongs to a \texttt{task struct} with the \texttt{is dm} flag set, the first entity in the \texttt{dm list} is set as the currently executing entity.

This solution ensures that the guest containing the task with the earliest deadline is the guest that executes, as specified by the algorithm in Section 5.3. However, if the guest is not EDF, this implementation also ensures that the task which the guest will actually execute is also the task whose entity’s budget is depleted.

Figure 6.3 shows how the system was modified to support DM guests. Note that even though the top \texttt{task struct}’s second \texttt{sched dl entity} is the leftmost node in the red-black tree, the \texttt{dl} pointer is pointing to the first node in the list. This is because the top \texttt{task struct} has its \texttt{is dm} flag set to true, and the first node happens to be the first element of the \texttt{dm list}, which is ordered by priority. In the example shown in Figure 6.3, the top \texttt{task struct} would be selected for execution since it owns the leftmost node in the red-black tree, and the top \texttt{task struct} would execute the first entity in its list, since that entity has the highest priority. Note that in the bottom \texttt{task struct}, \texttt{is dm} is set to false, and thus the \texttt{dl} pointer is pointing to the leftmost entity which belongs to the \texttt{task struct}. Furthermore, the \texttt{dm list} is empty for this \texttt{task struct} because the guest is not marked as a deadline-monotonic guest.

### 6.2 Expansion of the Introspection Engine

The majority of the contributions of this thesis are work based on the Kairos Introspection Engine [1]. However, in order to implement the Dynamic CSF and Flattening scheduling algorithms, the introspection engine had to be augmented. Initially, the introspection mechanism relied on hard-coded addresses, did not store any task information, and was not usable by the scheduler or any other part of the kernel. This section describes the modifications made to the introspection engine which allowed it to support many different guests at once, to inform the host scheduler of what tasks on the guests are doing, and to act as the primary means of measuring and reporting task information, deadline misses, and timestamps for real-time processes running on the guest operating systems.
6.2.1 The /proc Filesystem

One of the main issues with the initial introspection engine was that there was no way to configure it. As mentioned above, it used hard-coded addresses, which means it was only compatible with the specific guest virtual machine that the hard-coded addresses referred to. Additionally, the modified version of QEMU used by the introspection engine required the user to pass in the virtual machine PID as an argument to kairos_init.

The main design goal for the modifications to the introspection engine was to set up the system such that the user can perform some configuration to specify information about each VM, and then overwrite the guest address space by calling kairos_init with no arguments. Since the majority of the introspection code lives inside the Linux Kernel as a module, it was decided that the most convenient and efficient method to perform this user configuration was to use the /proc filesystem.

A secondary design goal was to relocate the kairos-specific code from the /arch/kvm subdirectory to its own subdirectory. The introspection code is now all located in the /arch/kairos subdirectory in the Linux kernel tree. When modules are built for the kernel, in addition to the modified kvm module and the unmodified kvm-intel and kvm-amd modules, the csf-intel, csf-amd, dcsf-intel, dcsf-amd, kairosvm-intel, and kairosvm-amd modules are all built. These modules correspond with the different algorithms evaluated in this thesis. The csf- modules only use introspection to measure real-time statistics, and rely on servers to maintain real-time guarantees on the guests. The dcsf- modules implement the dynamic CSF optimization specified in Chapter 5. The kairosvm- modules implement the flattening scheduling algorithm from Chapter 5. This organization scheme allows for ease-of-use when evaluating KairosVM.

All of the Kairos-related files in the /proc filesystem are in the /proc/kairos directory. This directory is created whenever the kvm module is loaded, and destroyed whenever the module is unloaded. Table 6.2 presents the location and description of all /proc files generated by Kairos.

Interaction with User-Space

To initiate introspection on a guest operating system, one of the Kairos modules must be loaded into the kernel. Then, before launching real-time applications on the guest, the user must first configure the guest using the read/write /proc files in Table 6.2. These files include begin_rt_seg, end_rt_seg, and dm. Once the VM is properly configured, the overwriting of addresses is performed by a QEMU monitor command called kairos_init. The user can then launch real-time applications on the guest, which will automatically be trapped, timestamped, and reported to the /proc filesystem on the host.

The two address files are necessary because each individual guest may have these two
Table 6.2: Kairos /proc Files

<table>
<thead>
<tr>
<th>File Name (in /proc/kairos)</th>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dl_misses</td>
<td>Read Only</td>
<td>Total number of deadline misses for all VMs.</td>
</tr>
<tr>
<td>dsr</td>
<td>Read Only</td>
<td>Deadline satisfaction ratio for the entire system.</td>
</tr>
<tr>
<td>id</td>
<td>Directory</td>
<td>Directory containing files specific to guest id. Guests are indexed from 0 to N in the order that they appear</td>
</tr>
<tr>
<td>id/address</td>
<td>Directory</td>
<td>Directory containing read/write address files for guest system calls to be introspected.</td>
</tr>
<tr>
<td>id/address/begin_rt_seg</td>
<td>Read/Write</td>
<td>Address on guest id where the ChronOS begin_rt_seg syscall resides.</td>
</tr>
<tr>
<td>id/address/end_rt_seg</td>
<td>Read/Write</td>
<td>Address on guest id where the ChronOS end_rt_seg syscall resides.</td>
</tr>
<tr>
<td>id/dl_misses</td>
<td>Read Only</td>
<td>Displays the total number of deadline misses for id</td>
</tr>
<tr>
<td>id/dm</td>
<td>Read/Write</td>
<td>0 if guest scheduler is EDF, 1 if guest scheduler is DM (Default: 0).</td>
</tr>
<tr>
<td>id/dsr</td>
<td>Read Only</td>
<td>Displays the deadline satisfaction ratio for id</td>
</tr>
<tr>
<td>id/num_jobs</td>
<td>Read Only</td>
<td>Displays the total number of jobs encountered on id</td>
</tr>
<tr>
<td>id/tasks</td>
<td>Directory</td>
<td>Contains a directory for each task encountered on id.</td>
</tr>
<tr>
<td>id/tasks/pid</td>
<td>Directory</td>
<td>Contains files related to the task with PID pid on id</td>
</tr>
<tr>
<td>id/tasks/pid/begin_ts</td>
<td>Read Only</td>
<td>Contains every timestamp for the ChronOS begin_rt_seg syscall for the task with PID pid on id in order from most-recent to least-recent.</td>
</tr>
<tr>
<td>id/tasks/pid/deadline</td>
<td>Read Only</td>
<td>Displays the relative deadline for task pid in seconds.</td>
</tr>
<tr>
<td>id/tasks/pid/dl_misses</td>
<td>Read Only</td>
<td>Displays number of times pid missed its deadline.</td>
</tr>
<tr>
<td>id/tasks/pid/dsr</td>
<td>Read Only</td>
<td>Displays deadline satisfaction ratio of task pid.</td>
</tr>
<tr>
<td>id/tasks/pid/end_ts</td>
<td>Read Only</td>
<td>Contains every timestamp for the ChronOS end_rt_seg syscall for the task with PID pid on id in order from most-recent to least-recent.</td>
</tr>
<tr>
<td>id/tasks/pid/exec</td>
<td>Read Only</td>
<td>Displays the worst case execution time reported by task pid in seconds.</td>
</tr>
<tr>
<td>id/tasks/pid/num_jobs</td>
<td>Read Only</td>
<td>Displays number of jobs released by task pid.</td>
</tr>
<tr>
<td>id/tasks/pid/period</td>
<td>Read Only</td>
<td>Displays the period reported by task pid in seconds.</td>
</tr>
<tr>
<td>id/vcpu</td>
<td>Directory</td>
<td>Contains files pertaining to the guest’s virtual CPU information.</td>
</tr>
<tr>
<td>id/vcpu/cpu</td>
<td>Read Only</td>
<td>Displays the PID of the virtual CPU with ID cpu on id.</td>
</tr>
<tr>
<td>num_jobs</td>
<td>Read Only</td>
<td>Total number of jobs released in the entire system.</td>
</tr>
</tbody>
</table>
ChronOS system calls located at different addresses in the kernel. The dynamically-generated /proc files allow the user to ensure introspection can happen properly on any collection of ChronOS guests. Additionally, minor modifications to the introspection code in the kernel would allow for non-ChronOS guests to be introspected as well.

The common method of obtaining the addresses of ChronOS system calls is to retrieve the System.map file from the guest. The System.map file contains the addresses of all of the symbols exported by the guest kernel, including the symbols for the begin_rt_seg and end_rt_seg system calls. Using these addresses, the guest can write to the appropriate /proc files. Upon error, the writing action will return a negative error code. If the application used to write to these proc files does not report errors, the user can detect any error by then reading the file to make sure the address has been written correctly.

The two address files are tied to a KVM data structure known as a kvm struct. In KVM, there is one kvm struct per virtual machine, so in this case it is appropriate to tie the VM-specific addresses to this structure. The actual addresses are overwritten by the user-space portion of the introspection engine, a modified version of QEMU. When the user executes the kairos_init command in the QEMU monitor for a VM, QEMU performs an ioctl on the kvm module to retrieve the addresses from the kvm struct associated with the VM. QEMU then overwrites the addresses and performs another ioctl, passing in the overwritten addresses. After this point, the introspection engine operates as described in [1].

The other read/write file is named dm, and should be set to 1 if the guest is scheduling real-time tasks using the deadline-monotonic algorithm. This /proc file is tied to the task_struct structure in the kernel, and represents a Boolean flag. The flag affects the operation of the Linux scheduler, in that it lets the scheduler know which guest task would currently run if the task_struct were scheduled. Scheduler modifications to support fixed-priority guests are described in detail in Section 6.1.4.

**Reporting Statistics and Timestamps**

The majority of files exported to the /proc filesystem are read-only files which report statistics about the tasks on the guest. The statistics reported are all based on timestamps obtained at the begin_rt_seg and end_rt_seg introspection points. Timestamps are obtained using the getnstimeofday kernel function. The begin_rt_seg system call is introspected on each job release for a task. Similarly, the end_rt_seg call is introspected whenever a job terminates.

The begin_ts and end_ts files list every timestamp collected for a particular task in order from most-recent to least-recent. This is accomplished by looping through a linked-list which contains all of the ordered timestamps. When either of these /proc files is read, a kernel function is called which performs the looping and copies the formatted timestamps into a user-space memory buffer which is what eventually is returned to the user. All of
the information in the other /proc statistics files can be derived from these timestamps. Therefore, the other files simply exist as a more convenient way to view statistics about the guest.

The \texttt{dl_misses} and \texttt{num_jobs} proc files are implemented as separate counters. The \texttt{num_jobs} counters are incremented every time a \texttt{begin_rt_seg} system call is introspected, as this corresponds to a task release. The system-wide counters are incremented every single time, the VM-specific counters are incremented only when the system calls are introspected for that VM, and the task-specific counters are only incremented when the introspection occurs because of that specific guest task. The \texttt{dl_misses} counters are incremented only when the most recent job termination timestamp ($t_e$) is greater than the most recent task release timestamp ($t_b$) plus the relative deadline for the task ($d$), i.e. $t_e > t_b + d$.

Figure 6.4 demonstrates how the data structures tied to each file in the /proc filesystem are organized in memory. The system-wide counters are global in memory, while each VM (represented by the \texttt{kvm} struct) stores its own counters as well as the system call addresses on the guest and a pointer to the list of guest tasks. Each guest task structure holds its task ID, its own counters, the real-time parameters for the task, and pointers to lists of timestamps for both \texttt{begin_rt_seg} and \texttt{end_rt_seg}. Each circle in Figure 6.4 represents a single timestamp, which is stored in a \texttt{timespec} structure.

The \texttt{dsr} file differs from the other files in that it isn't tied to any particular data structure like a counter, flag, or linked-list. Rather, the \texttt{dsr} file executes a function when read, which takes the \texttt{num_jobs} and \texttt{dl_misses} counters from the same directory and performs a fixed-point division with 8 digits of precision. The ratio returned is equal to the number of jobs ($j$) minus the number of deadline misses ($d$) divided by the number of jobs ($j$), i.e. $(j - d)/j$. If the number of jobs is still 0, the function will just report that the deadline satisfaction ratio is 1.

The \texttt{period}, \texttt{deadline}, and \texttt{exec} files are all tied to fields in a data structure called \texttt{kairos_task_list}. As the name indicates, this data structure is a member in a linked list which contains all of the tasks present in a particular guest. The head of the list is stored in the \texttt{kvm} struct. The fields of the data structure are populated when a new task is introspected for the first time. The \texttt{begin_rt_seg} system call passes the task period, deadline, and worst case execution time as arguments, which are captured by the introspection engine, stored in a \texttt{kairos_task_list} data structure, and added to the list. The /proc files provide an easy-to-use interface for the user to view these task parameters.

Finally, the \texttt{vcpu} files are tied to the actual \texttt{kvm_vcpu} data structures, and print the PID of each vCPU \texttt{task_struct} when read. These files are used to make it easier to pin each vCPU thread to a particular physical CPU for evaluations.
6.2.2 Task-Accounting using Thread-IDs

In order to keep track of each task’s timestamps in the Linux kernel, it was necessary to find a way to index each task for each virtual machine. It was easy to index each vCPU, as there is a separate `kvm.vcpu` data structure tied to each vCPU, and upon introspection this data structure is passed to the introspecting functions as an argument. Similarly, each VM itself has a separate `kvm` data structure tied to it, which can be access upon introspection using a pointer in the `kvm.vcpu` data structure. However, no such data structure exists for guest tasks, so upon introspection it is important to have a method to determine which guest task initiated the event.

Additionally, it is important that the method used can uniquely identify tasks so that the timestamps and statistics gathered are stored in the correct `kairos.task_list` data structure. It was decided that the PID of the task on the guest would be the most appropriate identifier to use, since PIDs are guaranteed to be unique in the scope of a single VM. The 3-tuple of period, deadline, and WCET was considered as an identifier, but it would cause issues if two instances of the same task were run on the same guest, since the 3-tuples would not be unique.
Retrieving the PID of the guest tasks required modification to the introspection engine. A field for PID was added to the rt_data structure which is passed as an argument to begin_rt_seg in ChronOS. Additionally, the ChronOS userspace libraries were modified to set this field equal to the value returned by the gettid system call. The gettid system call is used because it guarantees a unique ID, even when called from different threads belonging to the same process.

With the PID of the guest real-time threads now available in the host, it was possible to key each element in the linked-list of tasks with the PID. Now, upon introspection, it is possible to loop through each task and compare with the currently introspected PID to determine which task the current timestamp should be added to. In addition, the PID provides a method to determine when a new task has released a job, as compared to an already known task releasing a subsequent job instance.

6.2.3 Porting to AMD

The introspection engine initially only supported Intel processors because of a dependency on the Intel-VT hardware extensions. The engine was ported to AMD in an effort to support a wider range of hardware. AMD has similar virtualization hardware extensions known as AMD-V, but the software supporting these extensions is laid out differently in Linux.

Intel-VT-specific KVM code is organized into the arch/x86/kvm/vmx.c kernel source file. This is the file which was modified to create the initial introspection engine presented in [1]. AMD-V-specific KVM code is organized into the arch/x86/kvm/svm.c kernel source file. One of the added benefits of the organization design goal for this thesis is that reorganization of Kairos code into the arch/x86/kairos subdirectory reduced the amount of changes which were made to the vmx.c file down to a single Kairos function call. Therefore, to port to AMD, all that needed to be done was to find the proper point in svm.c to insert this same Kairos function call. In vmx.c this point happened to be in the is_invalid_opcode clause of the handle_exception function. In svm.c the corresponding point is at the beginning of the ud_interception function, which is called whenever an undefined instruction exception is raised by the AMD-V hardware extensions.

6.3 Multiprocessor Scheduling

Multiprocessor scheduling is done using partitioning, as described in Section 5.6. Each virtual CPU is assigned to a specific physical CPU using cpuset. This pinning of vCPUs is done before the release of any real-time jobs on any of the virtual machines. The first-fit algorithm is implemented in Python.
Chapter 7
Experimental Results

Unlike most research into real-time scheduling theory ([9], [15], and [17], for example), this thesis also includes an implementation in an existing real-time operating system with experimental evaluations in addition to simulations. Evaluations were performed on two identical Intel servers with 12 GB of RAM and an 8-core Intel Xeon E5520 CPU running at 2.27 GHz. For all but the multicore evaluations, execution was restricted to a single CPU core using cpusets.

An evaluation of the overheads associated with the guest scheduler, the virtualization layer, and the host scheduler is given in Section 7.1. A comparison of the deadline-satisfaction-ratios of Vanilla KVM and KairosVM as utilization is increased is given in Section 7.2.

7.1 Virtualization and Scheduling Overheads

To measure the overheads of both the virtualization infrastructure and the scheduling algorithms on the guest and host, the deadline miss load metric, first introduced in [35], was used. Deadline miss load presents a useful method of characterizing scheduler overheads by running a schedulable taskset and reducing the average period and execution time for the tasks until a deadline is missed, while keeping the utilization of the taskset constant. The end result is a minimum average period for tasksets run on the system; if a taskset is run with an average period below this number, the scheduler will not perform as well as it should according to theory.

In this thesis work, there are three sources of overhead which are of interest:

1. Overhead due to the guest scheduler (EDF) and kernel (ChronOS Linux)
2. Overhead due to the virtualization layer (KVM/QEMU)
3. Overhead due to the host scheduler (KairosVM Flattening Algorithm) and kernel (Linux)

In order to isolate each of these overheads, the deadline miss load was calculated for 3 different experimental setups:

1. Taskset is run on a single processor core
2. Taskset is run on a single virtualized processor core
3. Taskset is run on multiple virtualized processor cores, scheduled onto a single physical processor core

The first setup characterizes the overhead of the EDF algorithm. In this setup, the host operating system is ChronOS with the EDF module loaded (i.e. the guest operating system in all other experiments). By running the same taskset without virtualization, all virtualization overhead is eliminated, leaving only the overhead of the ChronOS EDF scheduler. A graph of deadline satisfaction ratio for different average taskset periods is given for this setup in Figure 7.1.

This setup was run with a taskset utilization of 0.95. The taskset consisted of 12 tasks with an average period which was varied between 4.4 ms and 155 ms. Each point on the graph is an average of 2 trials which ran for 60 seconds each; this is enough time to provide between 4,600 and 632,500 task activations per trial, depending on the average period. For the non-virtualized results in Figure 7.1, only the results up to 5 ms are shown, as the DSR remained constant at 1 for every trial beyond 4.5 ms.

The second setup characterizes the overhead added by the virtualization layer. Since there is only a single virtual machine to schedule, the host simply allocates all of the physical CPU time to the single virtual CPU. The virtual CPU was prevented from migrating across physical CPUs using cpuset. Experiments in this setup made use of the csf-intel module, so that the host scheduler is not involved, and the only overheads present are due to the guest scheduler (ChronOS EDF) and virtualization and introspection using KVM and QEMU.

The virtualization results in Figure 7.1 show the deadline satisfaction ratio as a function of average taskset period. The same tasksets used on the first setup were also used in this setup. The graph only shows data with average period between 153 ms and 155 ms because below 153 ms the deadline satisfaction ratio of the taskset is below 0.5. Beyond 154.4 ms, no deadlines are missed. These results demonstrate that the overhead due to virtualization is much greater than the overhead due to scheduling on the guest. Furthermore, Figure 7.1 demonstrates tasksets with a relatively high utilization and average periods below 154.4 ms should not be virtualized, as overheads will significantly impact performance in this case.

One of the interesting characteristics of the results from this setup is that the DSR oscillates between 1 and a number less than 1 before it ultimately converges to 1. In the non-virtualized
setup, this was not the case. One possible explanation for this behavior is that the virtualization layer transfers control of the physical CPU between the guest and the host at regular intervals; when task periods coincide with a harmonic of these intervals, the DSR is higher because the number of interruptions the guest experiences is constant for each task period. On the other hand, if the task periods do not coincide with a harmonic, during certain task releases the number of preemptions due to the virtualization software could be 1 higher than the other task releases, causing a deadline miss. This hypothesis is educated speculation, however; further examination is left to future work.

Finally, the third setup is used to characterize the additional overhead added by the KairosVM Flattening Algorithm. Similar to the first two setups, the overheads of the guest schedulers (ChronOS EDF) as well as the virtualization software (QEMU/KVM) are still present. However, the KairosVM Flattening Algorithm adds a third source of overhead: the host scheduler. However, since the host scheduler is a hierarchical version of EDF, it is expected that the overhead is similar to the overhead due to the guest scheduler, which is negligible compared to the overhead of virtualization.

This setup was evaluated with the same tasksets as the first two setups; however, in this case the tasksets were divided between 3 guest operating systems (4 tasks on each, for 12 total). Figure 7.1 shows the deadline satisfaction ratio for different average taskset periods. The same range of average periods is shown for this setup as was shown for the virtualization setup.
Table 7.1: Minimum Average Period of Tasks with No Deadline Misses

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Virtualization and Introspection</th>
<th>KairosVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Period (ms)</td>
<td>4.5</td>
<td>154.4</td>
</tr>
</tbody>
</table>

setup. Unlike in the second setup, the DSR remains above 0.9 even for lower average taskset periods. However, the region of interest for deadline miss load is the area where the DSR converges to 1, which in this case is at 154.4 ms, thus, the same region from the second setup is shown.

The improved performance for KairosVM over a single VM setup can be accounted for by the Constant Bandwidth Server (CBS) implementation provided in SCHED_DEADLINE. Since the KairosVM implementation contains a CBS implementation, when tasks miss their deadlines, they are not given any more execution time. Rather, this execution time is given to other guests, who can then make their deadlines. In the single VM setup, no CBS implementation is present, and thus the DSR is much poorer during overload periods. It is important to note that the oscillations present in the virtualization setup are still present in KairosVM; they are just less visible because the DSR only oscillates between 1 and around 0.999 in this setup. The significance of these results is twofold: the overhead due to the host scheduler is so much smaller than the virtualization and introspection overhead that it can be ignored, and the CBS implementation in KairosVM increases the performance in overload conditions beyond what a competitor in overload conditions could possibly match without CBS.

A summary of the deadline miss load values for each setup is given in Table 7.1. These results imply that virtualization of real-time systems is only possible when the overall utilization is below 0.95 and the average period of the taskset is above 154.4 ms. It is suspected that the average period value for the deadline miss load would be lower when utilization is lower; however evaluation of this hypothesis is left to future work.

### 7.2 Comparison with Vanilla KVM

Since CSF (and by extension Dynamic CSF) is designed without a mechanism to handle overload conditions, CSF can only be evaluated with tasksets which are schedulable according to the schedulability test outlined in Section 4.1. Evaluations performed on these tasksets yielded a perfect DSR of 1 for every solution, including Vanilla KVM. While it is expected that both KairosVM and CSF provide a perfect DSR for schedulable tasksets, it is somewhat surprising that Vanilla KVM is also able to satisfy all deadlines. This is most likely due to the fact that the applications evaluated rarely approach their worst-case execution times, which means that the CPU is so underloaded that even Vanilla KVM is able to ensure no deadlines are missed using simple round-robin scheduling.
The evaluations presented in the remainder of this chapter examine the case where taskset utilization is too high to be scheduled using CSF; as a result Vanilla KVM is used as a baseline comparison. Evaluations are performed using the following benchmarks:

- sched_test_app: A synthetic real-time benchmark application which runs a loop workload for each task to burn CPU time.

- x264: A video compression application which has been modified to execute as a single real-time task in ChronOS Linux [36].

- disparity: A motion-tracking application which uses stereo-vision. This application has been modified to run as a single real-time task in ChronOS Linux. This application is part of the San Diego Vision Benchmark Suite (SD-VBS) [37].

- Multi n-cut: An image segmentation application, modified to execute as a single real-time task in ChronOS Linux. This application is also part of SD-VBS.

Evaluations using the sched_test_app synthetic benchmark were done by generating 10 random tasksets using the Baker model [38] at each utilization on the x-axis, and measuring the deadline satisfaction ratio for each taskset. The same random tasksets are used for the evaluation of both Vanilla KVM and KairosVM. Additionally, the same random tasksets are used for the experiments done with all EDF guests, 1 RM guest, and 2 RM guests. The workload for each task produces an average execution time equal to the utilization reported on the x-axis; this means the actual utilization of the tasksets at each point can possibly exceed the value on the x-axis.

Tests on the x264 and SD-VBS benchmarks are performed by spawning a number of identical instances of each application running on identical input. These instances are spread as evenly as possible among the virtual machines in the system. As the number of instances increases, so does the overall taskset utilization. Once the number of instances increases to the point where the system is overloaded, the DSR begins to drop from 1.

All sched_test_app and x264 experiments were run on an Intel Xeon E5520 processor running at 2.27 GHz with 12 GB of RAM, while SD-VBS experiments were run on an AMD Opteron 6168 processor running at 1.9 GHz with 12 GB of RAM.

**Synthetic Benchmark Results**

Figure 7.2 provides the measured deadline-satisfaction-ratio for random tasksets over a range of utilizations. The error bars represent the minimum and maximum DSR over the course of 10 trials, where each trial represents a random taskset generated using the Baker model [38]. Each task has a period no smaller than 100 ms, and a period no larger than 1 second.
(a) 3 EDF Guests  

(b) 2 EDF Guests and 1 RM Guest

Figure 7.2: Overall System DSR with Random Synthetic Tasksets

In Figure 7.2a, all 3 guests were scheduled using EDF, whereas in Figure 7.2b, one guest was scheduled with RM while the other 2 were scheduled using EDF.

As expected, KairosVM outperforms Vanilla KVM in both cases. In fact, using flattened EDF (Figure 7.2a), KairosVM is able to maintain an average DSR above 0.95 up to a utilization of 0.95, while Vanilla KVM's average DSR drops to 0.75 under that same load. Additionally, KairosVM is able to use its CBS implementation to maintain a DSR near 0.9 even under heavy overload, while Vanilla KVM allows deadline misses to cascade, pushing its average DSR well below 50%.

Figure 7.2b demonstrates the effect of altering the guest scheduler to RM for one of the virtual machines. As expected, the reduced schedulability of RM causes the DSR to fall from 1 faster (at 0.85) than in a purely EDF setup (at 0.9). However, in overload situations, RM is able to satisfy more deadlines than EDF because it provides a higher priority to tasks which release more often, while EDF allows each deadline miss to delay execution of other jobs until the number of deadline misses cascades.

In both cases, KairosVM is able to provide a significantly higher DSR than Vanilla KVM in both high-load and overload situations, as well as a lower variability than Vanilla KVM when it comes to the difference between the best-case DSR and the worst-case DSR (i.e. KairosVM has smaller error bars than Vanilla KVM).

Real-World Benchmark Results

Figure 7.3 demonstrates the performance of the KairosVM Flattening Algorithm compared to the baseline of Vanilla KVM, for the x264 application. For Figure 7.3a, x264 was configured
with a period of 480 ms. For Figure 7.3b, each instance of x264 alternated between 480, 360, and 120 ms for the period. The instances were configured to run on inputs with the proper number of frames to have each instance finish up at around the same time (i.e. the 120 ms instance encoded 3 times as many frames as the 360 ms instance). The worst-case execution time of each job is 40 ms, but the actual execution time varies and is dependent on the input frame. Each 480 ms instance has a utilization of 1/12, so for Figure 7.3a, beyond 12 instances the system is technically overloaded. However, since each job for each instance of x264 rarely executes for 40 ms, it is still possible to attain an acceptable DSR even in overload conditions. The error bars in Figure 7.3 represent the span between the worst DSR and the best DSR over 5 trials, while the data points themselves represent average DSR over 5 trials.

Figure 7.3: Overall System DSR with Instances of x264

Note that KairosVM begins to pull away from Vanilla KVM as soon as the system is fully loaded (12 instances). That is because at high load, each scheduling decision is magnified. Unlike in low load conditions, it is very likely that every VM has an active job release. Choosing the wrong VM to execute can lead to a missed deadline in the other VMs. Since KairosVM uses the guest task information to choose which guest to schedule, it misses less deadlines. Furthermore, the Constant Bandwidth Server implementation used by KairosVM ensures that a task which has already missed its deadline does not continue to execute; this is an optimization for overload conditions which is not present in Vanilla KVM.

Figure 7.3b differs from Figure 7.3a in that the DSR falls from 1 with a smaller number of instances. This is due both to the less-optimal RM scheduler, as well as the difference in tasksets. Since some of the instances in Figure 7.3b have their period cut by 4, they have a utilization that is 4 times higher. This causes the system overload point to occur between 4 and 5 instances, rather than at 12 instances. As was the case with EDF guests, KairosVM
Figure 7.4: DSR of Individual VMs Running Instances of x264

is able to maintain a higher DSR than Vanilla KVM in conditions of both high load, and overload. Unlike the pure EDF case, Vanilla KVM’s DSR does not fall below 0.7, even at high load. This is because RM gives priority to tasks with a lower period, which produce the majority of the jobs. Therefore, even in high overload, RM is able to satisfy the deadlines of its high-priority tasks.

Figure 7.4 provides a per-VM breakdown of the DSR for x264 with a period of 480 ms (Figure 7.3a). As in Figure 7.3, the error bars represent deviation between minimum and maximum DSR over 5 trials. This plot demonstrates the stability of KairosVM during overload situations. Since x264 is a soft real-time application, it is acceptable to have a DSR slightly below 1. Note that in Figure 7.4a, deadline misses are not spread evenly over the 3 VMs. This can be an issue for soft real-time applications. Assume x264 has acceptable performance when the DSR is above 0.9, and unacceptable performance below 0.9. With 12 instances running, Vanilla KVM achieves an overall system DSR of just above 0.9. However, this does not mean that performance is acceptable; rather, only Guest 1 satisfies this performance bound, while Guests 0 and 2 fall below 0.9. In reality this means that 2 out of the 3 VMs are experiencing unacceptable performance. If the deadline misses were split evenly among all VMs, all 3 of them would have individual DSRs above 0.9.

By contrast, Figure 7.4b spreads deadline misses nearly perfectly among the 3 guests. The real-world impact of this is that for soft real-time applications, if the system-wide DSR is high enough for acceptable performance, the DSR within each system will most likely be high enough as well. Figure 7.4b also demonstrates the experimental implications of designing the algorithm based on a model where no priority between guests is assumed. In the absence of priority, each guest should maintain similar performance during overload.

Figure 7.5 displays the DSR of both KairosVM and Vanilla KVM for an increasing number
of instances of SD-VBS applications. A single instance of the disparity benchmark (Figure 7.5a) has a period of 500 ms, and an average execution time of 46 ms when run on the qcif input file. A single instance of the multi n-cut benchmark (Figure 7.5b) has a period of 985 ms and an average execution time of 83 ms when run on the sim_fast input file. The error bars in the plots represent the minimum and maximum DSR over the course of 3 trials.

Similarly to the x264 results, KairosVM provides a higher DSR with a smaller standard deviation. Note that in Figure 7.5a, the DSR for KairosVM is not 1, but close to 1. In both benchmarks, the DSR for Vanilla KVM falls away from 1 when the overall system utilization approaches 1 (full load). KairosVM is able to maintain its high DSR using the CBS implementation; whenever a job is going to miss its deadline, it is abandoned in favor of other jobs that may still make their deadlines. Without the CBS implementation (as is the case with Vanilla KVM), the job with the earliest deadline will always execute until termination, even if the deadline has already been missed. This pushes back the execution of other jobs and can cause a cascade of deadline misses.

Figure 7.6 displays the per-VM breakdown of the results from Figure 7.5. As was the case with x264, KairosVM ensures that deadline misses are more evenly spread among the 3 virtual machines than in the Vanilla KVM case. In Figure 7.6b, KairosVM does not miss many deadlines at all, so the DSR of each VM naturally stays near 1, while Figure 7.6a displays the tendency of Vanilla KVM to provide extremely volatile results when the DSR leaves 1.

The Multi n-cut results, in Figures 7.6c and 7.6d, better demonstrate KairosVM’s ability to provide a consistent DSR among VMs in comparison to Vanilla KVM. At 8 application instances, for example, Vanilla KVM favors Guest 2, whereas KairosVM does not favor any guest in particular (Guest 0’s average is brought down by deadline misses from a single
trial in this instance; the other two trials maintain a DSR close to 1). The conclusion from these results is the same as that from x264; in a soft real-time environment, KairosVM will support a higher utilization (more instances) before the DSR for any single virtual machine falls below the acceptable limit.
Chapter 8

Conclusions

In order to ensure tasks on a virtual machine are able to meet their deadlines, the host scheduler must be aware of the real-time taskset on each guest. Prior research into the virtualization of real-time systems depended on the use of servers to solve the hierarchical scheduling problem. However, servers are sub-optimal in that they introduce a trade-off between runtime overhead and taskset schedulability in the form of the server period length. This thesis provided a method to remove this trade-off without compromising support for fully virtualized systems, through the use of introspection and a flattened hierarchical scheduling algorithm. Through theoretical analysis, this thesis demonstrated the superiority of the flattened model over the server model. Additionally, the evaluations presented in this thesis demonstrated that the overhead of the flattening implementation does not compromise this superiority. As a result of this work, it is possible to consolidate more guests with larger tasksets onto a single system than was previously possible, maximizing the utility of a given system’s hardware.

8.1 Contributions

The major contributions of this thesis are as follows:

1. An algorithm was developed to schedule multiple virtual machines with either fixed-priority or EDF schedulers on a single system without the use of servers. The algorithm schedules the guest which contains the task with the earliest deadline. If that guest is scheduled with fixed-priorities, the algorithm must honor the priority relationship on the guest and schedule the highest-priority job on that guest. Because the algorithm is based on EDF, an optimal uniprocessor algorithm, it is also optimal in the uniprocessor case when all guests are also scheduled with EDF.
2. An exact schedulability test for the algorithm was derived, and then evaluated using simulations. The schedulability test is based on response-time analysis. If all of the tasks in all of the guests have a worst-case response time which is less than the task deadline, the taskset is schedulable. Simulations prove that the flattening algorithm can schedule a higher proportion of randomly generated tasksets than CSF, regardless of the server period chosen. Examples demonstrate that the flattening approach also leads to fewer virtual machine preemptions.

3. An implementation of the algorithm was provided in Linux, which provides real-time guarantees to legacy software stacks. The implementation supports fully virtualized guests running the ChronOS Linux real-time operating system. Unlike CSF or other server-based approaches, no offline computations are necessary to schedule real-time guests with this implementation. Instead, all real-time task parameters are extracted at runtime using the Kairos Introspection Engine. The Linux kernel was then modified to make use of these parameters by scheduling guests using the flattening algorithm.

4. An infrastructure to evaluate the Linux-based implementation with legacy real-time guests was provided. The infrastructure was used to conduct evaluations which show that the implementation matches the deadline-satisfaction-ratio provided using CSF when tasksets are schedulable using CSF. Furthermore, when tasksets are unschedulable with CSF, evaluations demonstrate that the implementation is able to provide a higher deadline-satisfaction-ratio than vanilla Linux, even in overload situations.
Chapter 9

Future Work

While the work presented in this thesis provides one solution to the virtualization of real-time systems, many areas remain which could be explored with further research. By relaxing the constraints set by the scope of this work, such as the task model, guest model, and scheduling model, future implementations of KairosVM can potentially lead to both interesting and practical results.

9.1 Expansion of Scenarios Evaluated

The task and guest model presented in Chapter 3, while restrictive, yield an infinite number of tasksets. Furthermore, these tasksets have a second degree of complexity due to the number of ways tasks can be spread among virtual machines with different schedulers. Simulations in this thesis were based on tasksets generated using the uniform distribution; these simulations could be expanded to other distributions as well. For practical reasons, the experimental evaluations presented in this thesis were only able to focus on tasksets generated by a few real-time applications. One potential avenue for future work is to expand these evaluations to more applications, as well as to a heterogeneous model, where different guests run different applications with different schedulers.

Beyond expanding evaluations within the same task model presented in Chapter 3, another avenue for future research is expanding the task model itself. Possible extensions to the task model would include sporadic tasks, which arrive randomly at a frequency no higher than their period $T$, adding jitter to the response-time analysis from Section 5.3, and relaxing the restriction that deadlines are equal to periods. Any of these three expansions of the task model would alter the schedulability test for the KairosVM flattening algorithm in interesting ways as well as increase the practicality of the implementation by expanding to applications with sporadic tasks or with tasks that have deadlines not equal to periods.


9.2 Extension of Guest OS Support

While expanding the task model is one interesting area for future research, expanding the guest model promises to be both more practical and more interesting. In the scope of this work, the guests all must run ChronOS Linux. Expanding support to other guest operating systems would immediately increase the practicality of the KairosVM hypervisor. However, this expansion would only require a small amount of engineering effort to adapt the introspection engine, making this direction for future work more suited to industry than to research. Each of the following subsections describes a method of expanding the guest model which provides a much greater challenge, suitable to the realm of research.

9.2.1 Multiprocessor Guests

In the scope of this thesis, guests were limited to a single virtual processor. This choice was made for the simplification of the theoretical analysis as well as due to specific issues which arise in the implementation when guests have multiple vCPUs. One direction of future work would be to expand the guest model to allow for up to $m$ vCPUs, where $m$ is the number of processors available on the host, and to allow for multiprocessor schedulers to run on the guest.

From a theoretical perspective the difficulty of the scheduling problem exponentially increases; rather than each guest having a single preferred task to execute, now each guest can have up to $m$. Additionally, each task would now have the ability to migrate from physical CPU to physical CPU, which is much more complex than the partitioned case examined in this thesis. From an implementation perspective, the implementation would have to handle migrations of tasks from vCPU to vCPU within a specific guest. To keep the accounting straight on the host, each migration would have to be introspected such that the host knows which task is tied to each vCPU in order to correctly apply its multiprocessor schedule.

Since hardware trends point towards multicore processing, even at the embedded level, the legacy real-time systems that this thesis aims to virtualize may quickly become multicore. Therefore, an expansion of the work to support multiprocessor guests would not only be interesting from a research standpoint, but also practical.

9.2.2 Access to Shared Resources by Guests

Another potential direction for future research is to examine distributed real-time tasksets which operate among several guests. If one virtual machine must block because another virtual machine is accessing a shared resource, the response-time analysis from Section 5.3 becomes more interesting due to the added term of blocking time, $B_i$. Additionally, many real-time systems use a heartbeat mechanism to report their readiness to a central server.
Ensuring that several virtualized guests within a single system can meet real-time network requirements such as heartbeat responses would be a practical application of this research.

9.3 Full Flattening with Paravirtualization

While this thesis limited its guest model to full virtualization, meaning the guest does not know it is virtualized, an interesting expansion of the guest model would be to allow for paravirtualization. In paravirtualization, the guest knows it is running as a virtual machine, and has the ability to communicate with the host using hypercalls. This communication mechanism could allow the guest scheduler to delegate to the host, giving the host scheduler full control over which task runs, and where it runs. Unlike the flattening solution presented in this thesis, such as system would be fully flattened, as if all of the tasks were run directly on the host.

9.3.1 Optimal Multiprocessor Scheduling

A fully flattened system would allow for the host to use a better multiprocessor real-time scheduling algorithm than partitioned EDF, such as LLREF [39] or RUN [40]. Additionally, the overheads of virtualization and the preemption of virtual machines could lead to more interesting cost models in the context of multiprocessor algorithms, with the opportunity to develop optimizations which reduce the number of virtual machine migrations or preemptions. Additionally, expanding the hardware model to examine the effects of non-uniform memory accesses and cache on the performance of these multiprocessor algorithms could lead to a hardware-aware scheduler which achieves the best experimental performance.
Bibliography


