Real-Time Hierarchical Scheduling of Virtualized Systems

Kevin P. Burns

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

September 25, 2014
Blacksburg, Virginia

Keywords: Real-Time, Virtualization, Scheduling, Introspection, KVM, Event Trapping, Syscall Tracing, Linux

Real-Time Hierarchical Scheduling of Virtualized Systems

Kevin P. Burns

(ABSTRACT)

In industry there has been a large focus on system integration and server consolidation, even for real-time systems, leading to an interest in virtualization. However, many modern hypervisors do not inherently support the strict timing guarantees of real-time applications. There are several challenges that arise when trying to virtualize a real-time application. One key challenge is to maintain the guest's real-time guarantees. In a typical virtualized environment there is a hierarchy of schedulers. Past solutions solve this issue by strict resource reservation models. These reservations are pessimistic as they accommodate the worst case execution time of each real-time task. We model real-time tasks using probabilistic execution times instead of worst case execution times which are difficult to calculate and are not representative of the actual execution times. In this thesis, we present a probabilistic hierarchical framework to schedule real-time virtual machines. Our framework reduces the number CPUs reserved for each guest by up to 45%, while only decreasing the deadline satisfaction by 2.7%. In addition, we introduce an introspection mechanism capable of gathering real-time characteristics from the guest systems and present them to the host scheduler. Evaluations show that our mechanism incurs up to 21x less overhead than that of bleeding edge introspection techniques when tracing real-time events.

Acknowledgements

I am incredibly thankful to everyone who helped me throughout this endeavor. I would like to specifically thank the following people for their support:

Dr. Binoy Ravindran, for his tireless guidance and support and for granting me the opportunity and environment to work on my something I am passionate about.

The members of my committee, Dr. Cameron Patterson and Robert Broadwater.

The students and faculty involved in the KairosVM project, Dr. Vincent Legout, Dr. Antonio Barbalace, and Matthew Chittum.

The members of the SSRG lab, especially Robert Lyerly, Aaron Lindsay, and Ben Shelton for camaraderie and friendship.

I would like to recognize the open source software and all the teams involved with the tools I used in the production of this thesis.

Finally, I would like to recognize the constant support from my family, my parents John and Carol Burns and my siblings James and Megan Burns.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There exists a plethora of previously deployed real-time systems in the world today. Where, real-time, refers to systems with time sensitive constraints. These constraints are defined by several parameters assigned to each task running on the system. A task's deadline is used to schedule its order of execution. The deployment and/or redeployment of a real-time system can be a tedious and demanding undertaking. Specifically, the validation and real-time analysis stages. Another scenario is the sources of the code base could be unavailable and therefore unmodifiable.

This results in a copious number of aging real-time software systems running on obsolete hardware platforms. Specifically, real-time systems running on or designed for a uniprocessor platform. With the industry shift from uniprocessor to multiprocessor systems, also came an additional level of software complexity. Transitioning existing code to support a multiprocessor platform requires deep modification. For legacy real-time software systems, this can be difficult to achieve. This has lead to the adoption of virtualization techniques for such scenarios. Where virtualization provides a fully emulated hardware environment, called a virtual machine, which can be tuned for each guest's requirements. Specifically, each legacy software system is executed on top of one virtual machine. Thus, we have many virtual machines running on a subset of physical processors.

When introducing real-time systems into this virtualized environment, there are issues that arise that need to be addressed. First, the guarantees (hard, soft, or best-effort) provided by the real-time stack need be conserved. In addition, traditional real-time scheduling algorithms are not meant to deal with a hierarchy of schedulers. Therefore, real-time deadline aware hierarchical schedulers must be developed and employed. Existing hierarchical scheduling solutions do not focus on reducing resource allocations to allow for more virtualized guests. As hard real-time systems do not allow for deadline misses, there is little room for resource consolidation. We instead focus our research on soft real-time systems following a stochastic model.

1.1 Limitations of Past Work

Traditionally, real-time systems have not been targets for virtualization. However, recent trends in both commercial and academic settings have provided solutions to the problems that arise when these systems are introduced into a virtualized environment. In the commercial scene many proprietary options use hardware partitioning practices to ensure real-time guarantees. While providing isolation and ensuring real-time guarantees, this method does not take advantage of CPU slack time, specifically in soft real-time systems, therefore does not support full utilization of the systems resources. When virtualized guests share system resources (i.e., CPUs), a partitioning scheme must be devised to prevent or ease contention. Conventionally, this problem has been solved via strict resource partitioning schedules derived by using the worst case execution times of all the guest's tasks. When scheduling soft real-time guests it is not always necessary to maintain these strict worst case boundaries.

Conventionally, more dynamic solutions to the hierarchy of schedulers problem depend on virtualization platforms that support communication between the hypervisor and its guests, or paravirtualization. However, this is not congruent with supporting the virtualization of legacy systems. As these paravirtualization techniques require modification of the guest operating system. A current trend of research in the security domain, is the concept of introspection. Introspection is the term coined for the action of extracting system information from a guest while the guest is executing. The current direction of this research is aimed at malware detection. The introspection engines required to detect malicious software often exceeds the needs of real-time introspection. They also tend to accrue a significant amount of overhead, as the malware detection is concerned in the entire domain of the system, where we are only concerned with the real-time domain.

1.2 Thesis Contributions

This thesis focusses on improving the current state of real-time scheduling algorithms for virtual machines. Our major contributions include:

- We created a framework for statically allocating bandwidth for virtual machines with the purpose of reducing resource consumption. In addition, we propose the extension to the multiprocessor domain by the use of a partitioning scheme on the hypervisor.
- We implemented an infrastructure supporting the full evaluation of our proposed schedulers. We then make use of this infrastructure to offer extensive evaluations of our proposed framework.
- We implemented a lightweight introspection engine capable of informing the host scheduler about real-time characteristics without degrading real-time performance.

1.3 Scope of Thesis

The main focus of this work is to provide the theory and implementation for a static probabilistic hierarchical scheduling policy, extend this scheduler to the multiple processor domain, and lastly to prove the feasibility of our introspection engine implemented in KairosVM. This will lay the groundwork for future KairosVM implementations.

Given the depth of complexity of the field, the scope of this thesis comes with limitations. Specifically, the concepts, experiments, and algorithms presented by this work try to be as general and overarching as is possible. However, there are assumptions and limitations that intentionally need to be considered to limit the scope.

The first limitation is that of our task simulations. For our real-time simulations we chose to model CPU intensive workloads. That is, our tasks do not depend on I/O of any kind.

Second, is the overall limitations of our task model, discussed further in Chapter 3. Our model disregards dependencies between tasks. That is, the tasks are independent and can execute without interfering the execution times of the other tasks. We also limit the scope of our model to exclusively periodic tasks. That is, we do not include an aperiodic or sporadic task model in any of out designs or experiments. As this covers a very large percentage of existing deployed real-time systems.

Third, our evaluations, following the stochastic task model, is limited to normal distributions. This also includes the real-world distributions collected using the libMAD benchmarks, discussed further in Chapters 4 and 5. That is when modelling stochastic scenarios, we choose to distribute the execution times normally using a ratio of the worst case execution time as the mean and a sixth of the worst case execution time as the distributions standard deviation.

Though we believe our work can be extended to other scheduling algorithms. Thus the fourth limitation is that our evaluations were done using the Earliest Deadline First (EDF) scheduler on the guest system. EDF is a dynamic scheduler that can guarantee complete schedulability of our guest task sets. This can be easily extended to static models, like Rate Monotonic based schedulers. Our current implementation of our static stochastic based virtual machine scheduler is based on an EDF Constant Bandwidth Server, therefore granting consistency. The blending of different host and guest scheduling algorithm combinations is left as future work.

As the number of combinations and permutations of parameters for experimental evaluations of such a solution are infinite in nature, we try to identify those most significant.

1.4 Thesis Outline

The remainder of this thesis is organized in the following manner:

- Chapter 2 reviews the related work in the area of real-time virtualization along with providing necessary background information.
- Chapter 3 discusses the real-time models and assumptions made during the development of the theories and implementations described in this thesis.
- Chapter 4 lays out the uniprocessor static scheduling algorithm based on the stochastic model for real-time virtual machines.
- Chapter 5 describes in detail the extension of our uniprocessor algorithm and implementation to the multiple processor domain.
- Chapter 6 describes in detail the design and implementation of our light-weight introspection engine.
- Chapter 7 outlines the conclusions of this thesis.
- Chapter 8 offers future work to extend the research presented in this area.

Chapter 2

Related Work

In this chapter, we provide some background of various aspects involved in this thesis. We then present existing solutions for hierarchical scheduling of virtual machines and virtual machine introspection.

2.1 Background

This section provides background knowledge for various implementation aspects of our presented solutions. Starting with the operating system used on each guest virtual machine. The other aspects are related to Linux. These are SCHED_DEADLINE, *cgroups*, and KVM. KVM is Linux's kernel virtual machine that exploits the underlying hardware's virtualization extensions, which we used with our hypervisor during our experiments. SCHED_DEADLINE is a scheduling policy and a task partitioner, that we use to schedule our virtual machines in Linux.

2.1.1 ChronOS

ChronOS Linux [1] is a scheduling framework which aims to provide a Linux kernel testbed for real-time scheduling and resource management research on multi-core platforms. It currently implements traditional real-time scheduling algorithms as well as best-effort policies.

ChronOS Linux was originally developed to support the real-time aperiodic task model, for the use with best-effort scheduling policies. However, it still supports the real-time periodic task model, which we will use for this thesis.

The scheduler in Linux is implemented as an ordered list of scheduling classes, as illustrated in Figure 2.1. The classes represent tiers of priorities. That is, every time the scheduler makes

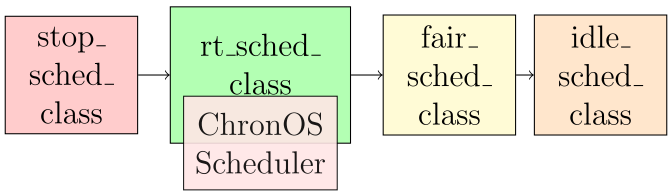

Figure 2.1: Linux's scheduling classes ordered by priority

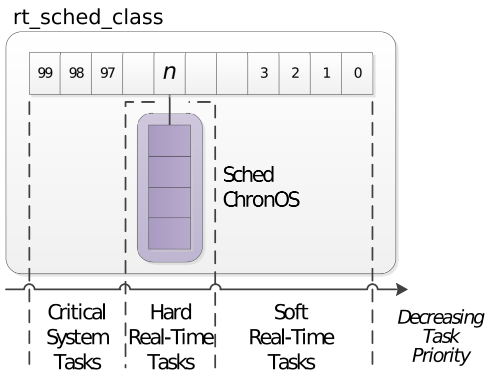

Figure 2.2: Real-time scheduling classes in ChronOS Linux. ChronOS' scheduled tasks are meant to be at `SCHED_FIFO` priority n

a scheduling decision the class list is scanned. The scheduler stops at the first class that has a task that is ready to be executed. The list is ordered by class priority where the highest priority class is (`stop_sched_class`) and the lowest priority class is (`idle_sched_class`). These two classes are not designed to be used from user space applications but are reserved for kernel space threads. The other two classes are designed to schedule kernel and user space threads. The `fair_sched_class` implements a time-sharing scheduling policy (derived from the `nice` functionality in UNIX). The `rt_sched_class` implements the POSIX `SCHED_FIFO` and `SCHED_RR`, real-time scheduling policies by means of a multi-level priority queue indexed by a bitmap. There are 100 priorities. In kernel space, 0 represents the highest real-time priority, conversely 99 is the least. This is reversed for user space priorities. The Linux kernel maintains a ready queue per scheduling class for each processor in support of multiprocessor architectures.

The ChronOS scheduler does not implement an additional Linux scheduling class, like in Calandrino et. al.'s Litmus^{RT} [2], but creates an extension to `rt_sched_class`. A ChronOS scheduling queue resides at a fixed priority level n in the aforementioned class extension, see Figure 2.2. For the sake of completeness it is important to note that with this architecture real-time tasks can be categorized as *soft real-time*, *hard real-time*, and *system critical*. ChronOS real-time guarantees are dependent upon the real-time guarantees provided by the parent `rt_sched_class` and the *PREEMPT_RT* patches. In addition ChronOS introduces a pluggable interface for real-time scheduling policies, which takes full advantage of Linux's kernel module infrastructure.

```
long begin_rtseg(int tid, int prio, int max_util, struct
   timespec* deadline, struct timespec* period, unsigned long
   exec_time);
long end_rtseg(int tid, int prio);
long add_abort_handler(int tid, int max_util, struct timespec *
   deadline, unsigned long exec_time);
long set_scheduler(int scheduler, int prio, unsigned long cpus)
   ;
```

Figure 2.3: Chronos application programming interface (without mutexes)

In ChronOS, a real-time task begins as a user-space thread scheduled at priority n within the `SCHED_FIFO` policy. ChronOS semantic requires to mark each job with library calls, `begin_rtseg()` and `end_rtseg()`; to mark the jobs start and end respectively. The concise application programming interface (API) without mutexes is depicted in Figure 2.3.

The function `add_abort_handler()` allows the user to be notified if a job exceeds its deadline and thus requires abortion; `set_scheduler()` dictates ChronOS' scheduling policy to use. From a system software point of view all of the APIs are system calls but the first three multiplexed to the syscall number `__NR_do_rt_seg` while the latter extends `__NR_set_scheduler`.

2.1.2 KVM

The Kernel-based Virtual Machine (KVM) relies on the host OS to schedule and manage the virtualized domains. KVM is the driver module that manages unprivileged access to privileged virtualization features, specifically the extensions offered by the underlying processor (Intel VT or AMD-V). For the use of hosting real-time virtual machines the target component for optimization is the underlying host operating system. The typical approach in literature is to use a host with real-time modifications. In essence, using an RTOS as a hypervisor. KVM is inherently designed for full virtualization purposes and is normally paired with QEMU for full emulation.

2.1.3 SCHED_DEADLINE

A new scheduling policy was introduced in the mainline release of Linux kernel version 3.14.0. This policy is called SCHED_DEADLINE, first introduced by Faggioli et. al. in [3]. This new policy is an implementation of the well-known Earliest Deadline First (EDF) real-time scheduler, with the addition of a mechanism called Constant Bandwidth Server (CBS). The CBS feature prevents the behavior of tasks influencing the performance of other tasks, thus guaranteeing temporal isolation.

This scheduling policy moves away from the fixed priority paradigm, currently implemented in Linux. That is, instead of tasks being assigned static priorities, they are know assigned a deadline, runtime, and a period. When scheduling virtual machines in this way, an explicit runtime needs to be enforced. That is, the executing VM process needs to be throttled if it begins exceeding it's runtime. This is done via the CBS mechanism.

2.1.4 Real-Time Group Scheduling

An alternative to scheduling virtual machine processes with mechanisms native to the Linux kernel, is by using *cgroups*. This concept was explored by Checconi et. al. in [4]. *Cgroups* are process containers, used to partition groups of tasks into hierarchical groups. This paired with Linux's process *throttling* mechanism can mimic the effects of a CBS implementation. The *throttling* interface presented by Linux let's the user to assign a runtime (time before throttling is active) and a period to a specific process group. Thus, similarly to a bandwidth server implementation, this provides the process groups isolation and prevents interference from other processes. That is, this implements the well-known resource reservation paradigm. This implementation differs from the above mentioned SCHED_DEADLINE as it makes use of SCHED_FIFO or SCHED_RR for the real-time scheduling of real-time process groups.

2.2 Related Work

2.2.1 Hierarchical Real-time Scheduling Algorithms

The scheduling of real-time virtual machines is similar to the notion of hierarchical scheduling. Hierarchical scheduling was first introduced in Strosnider et. al. [5] and Deng et. al. [6]. Traditional hierarchical scheduling refers to the scheduling of real-time applications and not virtual machines, but the overall objective is the same. As each application has many internal tasks to schedule and execute. Servers are typically implemented for hierarchical scheduling, in order to schedule the applications. Hierarchical scheduling theory has been extended and improved by Lipari et. al. in [7, 8, 9, 10].

2.2.2 Real-Time Virtualization

There are many existing virtual machine managers in use today. However, this paper is restricted to the analysis of open source solutions such as Xen, KVM, and several micro-kernel based designs. There are proprietary hypervisors on the market (e.g. Wind River's [11], NI's [12], TenAsys's [13], RTS's [14] real-time hypervisors). These hypervisors use strict hardware isolation and partitioning methods. However, the research community does not have evaluations of such systems.

Xen-based Solutions

The Xen hypervisor is a popular platform for real-time hierarchal scheduling research. This is due to the paravirtualization support, inherent to Xen. Paravirtualization (PV) is a virtualization technique that creates a communication layer integrated into the guest operating systems, called hypercalls, which allow lightweight communication between the host and guest. Therefore, the guest operating systems require modification. This PV technique does not require hardware virtualization extensions and thus can run on any underlying hardware platform.

Lee et. al. in [15] extends Xen's default credit-based scheduler by adding the *laxity* real-time parameter. Xen's credit-based scheduler is a proportional fair share CPU scheduler. The objective for this extension was to support soft real-time systems with I/O dependencies. One observation was that many soft real-time tasks do not require CPU cycles directly after the reception of a packet. As the default credit-based scheduler automatically boosts the priority of the VM due to I/O activity, this was wasteful. Therefore, the addition of *laxity* offers the interface to relax the boost in priority due to I/O reception.

Xen was equipped with a Simple Earliest Deadline First (SEDF) scheduler by default, since then the scheduler has been depreciated. SEDF allows the assignment of a runtime, period, and deadline to each virtual machine, or domain. One major issue with SEDF is that it had no method of distinguishing between real-time and none real-time VMs, or domains. This issue was addressed by the authors in [16].

Without a paravirtualization option, the hierarchy schedulers of real-time tasks has been addressed by the introduction of real-time servers. Servers are a thoroughly studied theoretical solution for this problem. There are few well established platforms in order to compare the various server implementations. In the above mentioned Xen-based solutions the researchers simply modify existing schedulers inside of Xen, without drastically altering the original code base. RT-Xen boasts to be the first hierarchical real-time scheduling framework [17]. It was designed to be a testing platform for researchers to implement their theoretical schedulers in Xen. For evaluation purposes, the authors implemented versions of Deferrable, Periodic, Polling, and Sporadic VM servers into their prototype platform. These are not novel approaches, the authors instead wanted to provide a comprehensive evaluation of each scheduler on one platform. The modifications of the existing Xen platform included some xm (Xen monitoring utility) extensions along with the scheduler implementations themselves. Along with the code, the implementation came with assumptions. The biggest amongst these are that each VM is assigned to a single VCPU, that is pinned to a single physical core. The most significant of the evaluations provided was the deadline miss ratio for each of the schedulers. This is significant as an empirical evaluation of these schedulers has never been done before.

RT-Xen has been extended to realize compositional scheduling of virtual machines [18]. RT-Xen was modified to support compositional scheduling frameworks (CSF). On top of the modification, two novel compositional schedulers were introduced. A method to share components amongst schedulers and even swap schedulers online was implemented as well. The authors are trying to mitigate, the scheduling overhead incurred and the loss of timing guarantees incurred when a hierarchy of schedulers is introduced. The compositional periodic schedulers introduced were the Purely Time-driven Periodic Server (PTPS), the Work-conserving Periodic Server (WCPS), and lastly the Capacity Reclaiming Periodic Server (CRPS). PTPS implements a time-driven budget replenishment technique in RT-Xen, while WCPS is similar to PTPS it differs when dealing with idle domains. WCPS will demote domains that are currently scheduled and idle. Lastly, CRPS is like WCPS except it makes the effort of using any idle time of the currently running domain to farm out to other domains. For the evaluations the authors focussed on the metric of response times latency and jitter, without taking into account the effect on the task's deadline satisfaction.

Linux/KVM-based Solutions

The use of Linux's KVM to schedule real-time virtualized guests have been explored by Kiszka et. al. in [19]. This work showed that Linux has promise as a real-time hypervisor. In that, using a real-time fixed-priority scheduling policy (SCHED_FIFO) on both the guest and host can lead to reasonably low overhead for some use cases. However, there are still problems to be solved, namely if there are tasks executing on the native Linux host, there may be priority inversions.

IRMOS is a real-time operating system that targets soft real-time guarantees necessary for many multimedia applications. The authors of [20] illustrate how the infrastructure presented by IRMOS, with the use of KVM, can be used to schedule real-time virtual machines. The existing reservation based scheduler the real-time OS, IRMOS, provides operates by partitioning the real-time tasks (in this case real-time virtual machines) via Linuxs *cgroups*. *Cgroups* reliably partitions resource access of process groups. The authors present the idea of spare reservations, these are essentially smaller containers used for the shorter time sen- sitive tasks, while the main containers are used amongst the computationally heavy tasks. This is triggered by a modified network driver on the host OS. The spare reservations are in a pool and are shared amongst the VMs. This technique vastly improves the responsiveness of the real-time network traffic. However, the real-time performance was not evaluated extensively. The authors showed the ping response time performance with and without the addition of their spare reservations. This work is limited to a very specific use case, however it definitely shows promise for existing real-time Linux variants being used as real-time hypervisors.

Other Solutions

Other hierarchical scheduling designs have been implemented in other virtualization platforms. Xen and KVM are very popular virtualization solutions, however there are other hypervisors that offer unique characteristics that Xen and KVM lack. One such solution is the L4/Fiasco microkernel.

Fiasco has been used as a real-time hypervisor to observe the need of a ”flattened hierarchical scheduler” [21]. Conventionally, Fiasco supports scheduling guest VMs as black boxes. However, the authors show that this poses a distinct problem with hard real-time applications. The Fiasco microkernel offers an interface that allows user level schedulers to change parameters in the kernel level scheduler policy. The authors use paravirtualization techniques (vmcalls) to ”flatten” the guest and host schedulers. One key aspect to Fiasco is its extremely low latency Inter-Process Communication (IPC) mechanims, which their vmcalls take full advantage of. The flattening is done by allowing the guest real-time operating system to choose the underlying priority used on the host (Fiasco) via the interface it provides. That is, the guest has task by task control over the VCPU process’ priority. The results of this technique indicates an overhead incurred by the vmcalls that are used to do the priority switch. The overhead was measured for a simple benchmark accessing the host’s TSC. They target mixed-criticality systems but they do not go into much detail on how the tasks are scheduled on the host. Their objective however is to use the knowledge of which task is active on guest to update guest bandwidth accordingly and increase the overall number of deadline hits. This work is limited in that it has not been fully evaluated for real-time guarantees.

2.2.3 Introspection

Several solutions for virtualizing real-time systems assume paravirtualization support. The solutions presented in this thesis aim to target legacy systems without the need for any modifications of the guest. Therefore, we cannot rely on paravirtualization techniques and instead require to gather information about the activities on the guest by way of virtual machine introspection, shortened to introspection.

There exists several introspection engine implementations, supporting various combinations of operating systems and hypervisors. These existing solutions focus on security and target the monitoring of guest’s execution patterns for malicious and suspect activities. These solutions include Nitro [22] and others [23, 24, 25, 26].

Nitro is a virtual machine introspection framework designed to monitor system calls made in the guests and detect malicious activities. Nitro is Intel specific and is currently limited to only trapping system calls and no other system events. As Nitro traps all system calls made by the guest operating system and relies on filtering techniques, it is not suitable for real-time systems because of the latency associated with catching system call made in the guests.

Real-time introspection solutions have been proposed by Cucinotta et al. [27, 28]. However, these solutions, like Nitro, trace every system call and thus generate large amounts of overhead.

Chapter 3

Models and Assumptions

3.1 Task Model

We consider the periodic task model laid out by Liu and Layland [29]. Each task τ_i has a period and deadline denoted as T_i and D_i respectively. For the design and implementations seen in this thesis, we assume the deadline of a task is equal to its period. One execution of a task can be described as a job and will be denoted as τ_i^j, where j is the job's iteration being described. We denote the set of all tasks assigned to a specific guest g_i, also referred to as a taskset, as Γ_i. The hyperperiod of a taskset Γ_i is characterized by the least common multiple (LCM) of Ti for all $\tau_i \in \Gamma_i$.

Table 3.1: Model Notation Reference

Name	Notation
Task	τ_i
Job	τ_i^j
Taskset, Guest	Γ_i, g_i
Period	T_i
Deadline	D_i
Worst Case Execution Time	C_i
Chebychev-based Execution Time	c_i
Distribution of Execution Times	Y_i
Variance	$Var()$
Expected Value	$E()$
Probability	ρ_i
Task's Utilization	U_i

For the probabilistic based static scheduler, presented in detail in Chapter 4 and extended in 5, we use the probabilistic model defined by Yuan et al. [30] and then later used by Wu et al. [31][32]. This model adds two additional characteristics to each task. That is, each task now has an expected execution time $E(Y_i)$ and the variance $Var(Y_i)$ of execution times, both derived from a distribution of execution times, Y_i. Though we do not represent synthetic tasks strictly by their Worst Case Execution Time (WCET), C_i, we do not let synthetic tasks exceed their alloted WCET. Thus, the original calculated WCET is respected.

The utilization U_i of each task is defined as $\frac{C_i}{T_i}$. Furthermore, the overall utilization of a guest g_i is defined as the sum of all utilizations in Γ_i. We make the assumption that each guest's utilization is less than one, as each guest is alloted a single core.

3.2 Guest Model

A guest g_i is defined as a virtual machine running on top of a hypervisor without privileged access to the processor. Each guest has a set of real-time tasks, that are scheduled by the guest's scheduler and are uninfluenced by the hypervisor's scheduler. For the evaluations performed as part of this thesis, the operating system running on each guest is ChronOS Linux. The guest scheduler for our evaluations is EDF, we leave RM and other schedulers for future work.

Our focus is to support legacy, non-paravirtualized, systems. That is each guest acts as if it is running on bare metal. Therefore, the environment presented to each guest is fully virtualized.

3.3 Scheduling Model

We use a hierarchical scheduling model, with one host and multiple guests. Where the host is responsible for the scheduling of the guests and each guest is responsible for scheduling its real-time tasks. Each guest is limited to an EDF scheduler, while the host uses a constant bandwidth server based scheduler. As each guest is only associated with one emulated CPU, or virtual CPU (VCPU), we treat each VCPU as a server. Similar to the approached used by RT-Xen [17].

Each guest is allocated exactly one processor. Therefore each guest operating system only has access to one VCPU. We use partitioned scheduling, so these VCPUs are assigned to one physical processor and cannot migrate. Our scheduling policy does not consider overheads incurred by preemption of the the VCPU processes. Therefore they will be incorporated in our experimental results. When assigning a physical CPU to a virtual CPU, a best-fit heuristic is used.

3.4 Hardware Model

We consider a homogeneous multi-core architecture for our host system, with m processor cores. For the scope of this thesis we will refer to a processor core simply as a processor. For the purposes of our scheduling algorithms, all cores are treated as independent but identical entities. We do not take into consideration the effects of the cache on performance.

Chapter 4

Uniprocessor Probabilistic Hierarchical Scheduling

In this chapter we present our solution to schedule virtual machines (VM), and in consequence their residing real-time tasks, onto a uniprocessor. All computations for this schedule are done offline, thus the scheduler is static.

The goal of this scheduler is to give the user the option to reduce the CPU time reserved for each VM, without dramatically degrading real-time performance. To do this, we base our scheduler on the Compositional Scheduling Framework (CSF), developed by Shin et. al. [33]. CSF takes in the set of tasks for each VM and produces a bandwidth allocation. CSF will be described in more detail in section 4.2 of this chapter. Our implementation also derives from the algorithm proposed by Wu et. al. in [31]. Wu's algorithm calculates a new execution time cap for a task, using the task's execution time distribution's mean and variance as inputs. This algorithm will be discussed in more detail in section 4.1. Once CSF has calculated bandwidth allocations, we make use of a constant bandwidth server to schedule the VMs using the Earliest Deadline First (EDF) scheduler.

4.1 Task Level Scheduling

Real-time tasks do not always terminate close to their worst case, in many cases tasks terminate far from this bound. Due to modern processor pipelines becoming more and more unpredictable, along with operating system jitter, Worst Case Execution Time (WCET) calculations are a non-trivial process. Therefore, there is unutilized CPU time. In soft real-time systems, if we have access to the characteristics of a task's distribution of execution times, a more reasonable C_i bound can be calculated. Where C_i is a portion of the previously allocated WCET, that guarantees a deadline won't be missed given a user defined probability. We do this by following the work presented by Wu et. al. [31].

The first step of the scheduling process is to compute the alternative execution time bound. Let C_i denote this bound for task τ_i, given ρ_i denotes the probability that every job of the task will terminate before C_i is reached. C_i can be calculated via the following equation, which is based on Chebychev's inequality[31]:

$$C_i = E(Y_i) + \sqrt{\frac{\rho \times Var(Y_i)}{1-\rho}} \tag{4.1}$$

Where, $E(Y_i)$ represents the expected execution time in τ_i's distribution of execution times denoted by Y_i. Let, $Var(Y_i)$ denotes the variance of Y_i. The guarantee this equation provides is that for any job j_i of task τ_i, its execution will not exceed the computed C_i with a probability no less than ρ. As these allocations can potentially produce a C_i that exceeds τ_i's WCET, we reclassify the above equation as:

$$C_i = min\left(WCET, E(Y_i) + \sqrt{\frac{\rho \times Var(Y_i)}{1-\rho}}\right) \tag{4.2}$$

The scheduler that resides in the guest VM then schedules its tasks using EDF. As each guests is allocated one virtual CPU, the utilization cap is 1.0. We consider guests with much lower utilization caps as a guest with a global utilization becomes uninteresting. As the scheduler could only support the one guest. We consider the guest's total utilization produced by using our calculated C_i's, rather than using each task's WCET. Therefore, the WCET can still produce a global utilization that exceeds 1.0. When performing our evaluations we assume that all tasks on all guests use the same value for ρ. This way, performance fluctuations were easier to measure. Using unique values of ρ on each guest or task would be interesting future work.

4.2 Virtual Machine Scheduling

When a hierarchy of schedulers exists, like in our case of guest and host, there are different ways to accommodate the needs of each tier in the hierarchy. We chose to do this by implementing each VM as a server. Each server is allocated a runtime and period by the host scheduler. More specifically, the server is a Constant Bandwidth Server (CBS)[34][35]. CBS guarantees that the ratio of $\frac{runtime}{period}$ (i.e. the bandwidth) of each server only contributes that same ratio to the overall utilization of the system. For example, if we issue a runtime of 3us and a period of 10us to a server then its contribution to the overall utilization cannot be greater than 0.3. In order to schedule a given VM as a server, we first need to generate a runtime and period that consider the needs of the tasks residing in the guest. Where, the runtime of a server is the amount CPU time executed by the server for every server period. We accomplish this via CSF.

Let's denote the server's allocated period as Π and its runtime as Θ, so that on every Π the server is executed for Θ. The CSF algorithm takes in the set of tasks run on the server. The algorithm outputs the runtime, which can be used to defer the server's bandwidth. Given a fixed period, the calculated budget must honor the following inequalities:

$$\forall 0 < t \leq H_\Gamma \quad \text{and} \quad dbf_{EDF}(\Gamma, t) \leq sbf_R(t) \tag{4.3}$$

Where, H_Γ is the hyperperiod of the guest's taskset Γ. H_Γ is calculated as the least common multiple of p_i for all $T_i \in \Gamma$. The Demand Bound Function *dbf* is dependant on the guest's scheduler, in this case we use EDF and can thus by computed by:

$$dbf_{EDF}(\Gamma, t) = \sum_{\tau_i \in \Gamma} \lfloor \frac{t}{T_i} \rfloor \times C_i \tag{4.4}$$

The Supply Bound Function *sbf* calculates the minimum possible resource supplies during time t and can be described by:

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

Equation (4.3) is used to compute the runtime each server. The execution time for each task is given by Equation (4.2), rather then using the WCET. The fixed period is chosen arbitrarily, this is chosen based on practical characteristics of KVM and thus bounded by implementation constraints. The host uses these calculated runtimes and periods to schedule the VMs using Linux's SCHED_DEADLINE.

4.3 Schedulability

The theory behind CSF guarantees all deadlines of tasks inside a VM are met. Our model is different than a pure CSF model, as we consider probabilistic distributions of execution times.

Due to these distributions, we must consider two cases. The first case is when a job's execution time is less than C_i. In this case, no deadline will be missed as this is covered by the CSF guarantee. The second case is when the execution time of a job exceeds C_i. This case can cause a deadline miss. However, our model, derived from Wu et. al. [31], gives the probability of such an occurrence.

4.4 Complexity

Though the complexity of our approach needs to be feasible, there is no need for our solution to be bounded. This is due to the computations being calculated offline. The complexity of our entire solution is similar to that of CSF and Wu's solution.

The complexity of CSF is dependant on the hyperperiod of the taskset. The complexity of calculating the runtime for each server is as follows. Equation(4.3) is bounded by, $\forall 0 < t \leq H_\Gamma$, and thus depends on the value of H_Γ. For each t, the computations require computing the DBF and SBF. The complexity of DBF is $O(n)$, where n is the number of tasks. The complexity of SBF is $O(1)$. Another level of complexity is added by the range of runtimes given. In that, each runtime is tested until the schedulability test succeeds.

4.5 Experimental Evaluations

To evaluate our probabilistic hierarchical algorithm (PHA) static scheduler, we performed several experiments. The first set of experiments conducted, we used a real-time synthetic benchmark. These simulations were strictly CPU load tests, that make use of tasksets that emulate real-world light workloads, generated based on the Baker model [36]. The details of these experiments can be found in Section 4.5.2 of this chapter. We then furthered our experimentation by using a real-world real-time scenario. That is, we extended the open source MPEG Audio Decoder library, libmad, by adding real-time parameters. This allowed us to evaluate our solution using authentically generated execution time distributions. We used a sample of 10K execution times to derive distribution characteristics used in our PHA calculations. We then evaluated the impact our solution had on the real-time properties of our modified libmad implementaiton.

4.5.1 Evaluation Environment

We run the tests on an Intel Xeon E5520 processor with eight cores at 2.27GHz, and 16GB of RAM. The host is running Ubuntu Server 10.04 with Linux kernel 3.15.0 paired with KVM/QEMU version 0.12.3 as the hypervisor. On each guest, we used Ubuntu Server 10.04 with Linux kernel 3.0.24, patched with our ChronOS 3.0 patches.

The benchmarks and tests were run on VMs, each with a single VCPU. Each VM's VCPU was that is pinned to the same isolated CPU. We mad use of the `isolcpus` kernel command line argument to dedicate specific cores to the KVM VCPU processes.

For our experiments, we discovered two notions of time represented in virtual machines: *virtual time*, measured via `clock_gettime()`, and *physical time*, obtained by reading the CPU's Time Stamp Counter (TSC) register. We tweaked KVM's paramters to disable dynamically changing of the *TSC OFFSET* portrayed in the VCPUs, in order to maintain reliable physical time measurements. Furthermore, because the VCPU is pinned to a single CPU, we are mitigating any possible inconsistencies due to subtracting TSC reads from different CPUs. Compared to virtual time, the physical time has an absolute connotation, while the former is relative, because the TSC does not pause when the virtual machine is idle. Thus we chose to convert all the time measurements in our benchmarks to use *physical time*.

4.5.2 Synthetic Benchmark

For our real-time benchmark, called *sched_test_app*, we generated synthetic tasksets based on the Baker model [36], but extended to fit our probabilistic model. Given this model, a light workload is characterized by the execution time generation bounds, of 1% and 10% of the tasks's period. This generation is done in a random uniform manner. We chose to use uniformly distributed work loads, as they are represent real-world conditions and grant our *sched_test_app* a wide variety of input parameters. Specifically, the Baker model was designed to stress EDF schedulers.

Therefore, we chose to evaluate the EDF scheduler provided by ChronOS. We constructed a series of tests evaluate our PHA static scheduler. However, *sched_test_app* was originally designed to consider task's worst case execution time (WCET). That is, each synthetic job would execute its worst case every time. Therefore, we modified *sched_test_app* to take in distribution characteristics. That is, instead of each job executing one exact same amount of time, our modifications allow the execution times to follow a normal distribution. To construct these distributions we implement the Box-Muller technique[37], which takes in a distributions' expected value and variance, then outputs a point inside the desired distribution. Since we are using normal distributions, a distributions mean and expected value are the same.

To vary these distributions we decided to generate sets of tasks that have means to a percentage of the specific task's WCET. For example, if we have a task τ_i with a WCET of 100us, then we can set our distribution's mean to be $0.30 \times WCET$. We chose to generate tasks that use 30%, 50%, and 70% of their WCETs. With these three distributions, we expect those generated with means equal to 30% of their wcet to have light CPU utilizations and distributions with means 70% their wcet to represent higher CPU loads.

Now that we have distributions, we can generate runtime allocations for each of our VMs. Remember, our VMs are treated as servers in our PHA scheduler, on the host. Therefore, we need to generate a runtime and period to each VM. We do this by the Compositional Scheduling Framework (CSF) technique [33], described in Section 4.2 of this chapter. Before we input our tasksets into their framework, we first apply the Chebychev based equation developed by Wu et. al. [31]. This outputs a new execution time bound, c_i, that can be tweaked by an input probability, ρ_i. More details on this process can be found in section 4.1 of this chapter.

We varied ρ from 10% to 90% and plot this on the x-axis. For the y-axis we measure the Deadline Satisfaction Ratio (DSR) given by our real-time synthetic application *sched_test_app*. DSR is the ratio given by $\frac{\text{(total tasks)}-\text{(deadlines missed)}}{\text{(total tasks)}}$. These tests were run on three VMs, the task sets were generated with a utilization cap of .3. Where a task's utilization is described by $\frac{c_i}{T_i}$, here the execution time is denoted as c_i and the period is T_i. The taskset's utilization is described by $\forall \tau \in \Gamma$ That is, the bandwidth allocations for each VM cannot exceed the available resources. Figure 4.1, shows the results of the evaluations performed with a mean of 30% the WCET, for each allocation output by CSF. Likewise, Figures 4.2 and 4.3, show the results from varying the means to 50% and 70% the task's WCET.

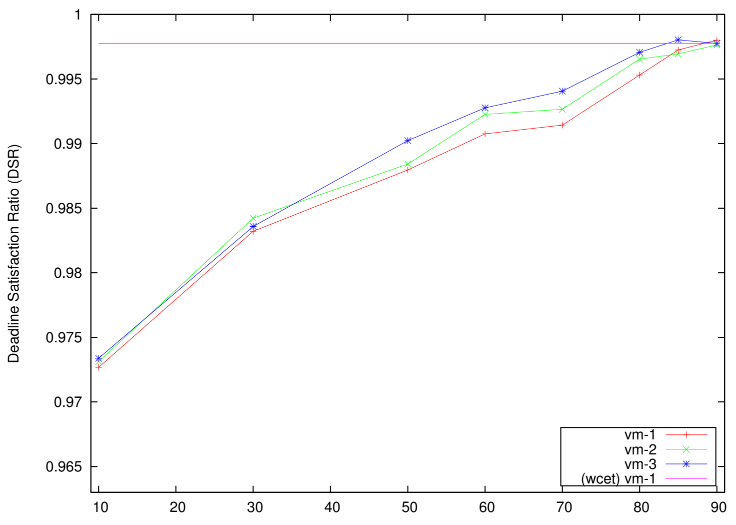

Figure 4.1: PHA for uniprocessor with expected value of 30% the WCET

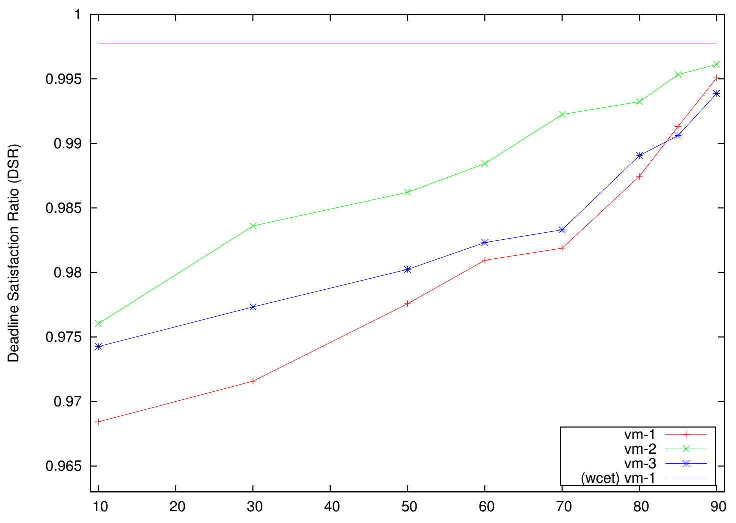

Figure 4.2: PHA for uniprocessor with expected value of 50% the WCET

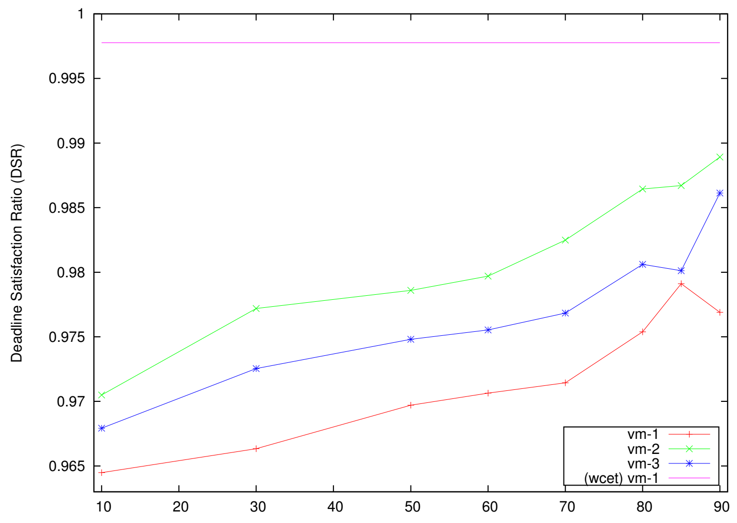

Figure 4.3: PHA for uniprocessor with expected value of 70% the WCET

The three figures all share a similar trend. That is, as the probability ρ_i increases, so does the DSR. This outcome is due to these probabilities' influence on the bandwidth allocation. As ρ parameter we tweak to reduce the deadline satisfaction guarantee for each task, then we expect to see this trend on a per task level. As we input the same ρ for each task in the taskset, the trend of the entire taskset should mirror that of the individual task. If we compare each figure to the other, we notice as the expected value of the distributions increase, the overall DSR decreases. This is especially obvious when $\rho = 90\%$. This is due to the overall utilization of each task being closer to the worst case.

4.5.3 Practical Scenario Evaluations

For the full stack evaluations, we chose to implement a real-time version of the open source MPEG Audio Decoder, or libmad. The MP3 codec lacks specific guidelines for decoding delays, therefore many decoders lack deterministic execution times, this coupled with system jitter and overheads, the distribution normalizes. We discovered the use of libmad in [38] and noticed desirable distribution trend. Our solution is amenable to any distribution of execution times, we chose MP3 decoding times as they best demonstrate the characteristics of our scheduler. Figure 4.4 show five distributions we generated using the libmad library. These distributions were generated using an isolated environment. That is, we gave the libmad processes real-time priority and affined them to a single processor.

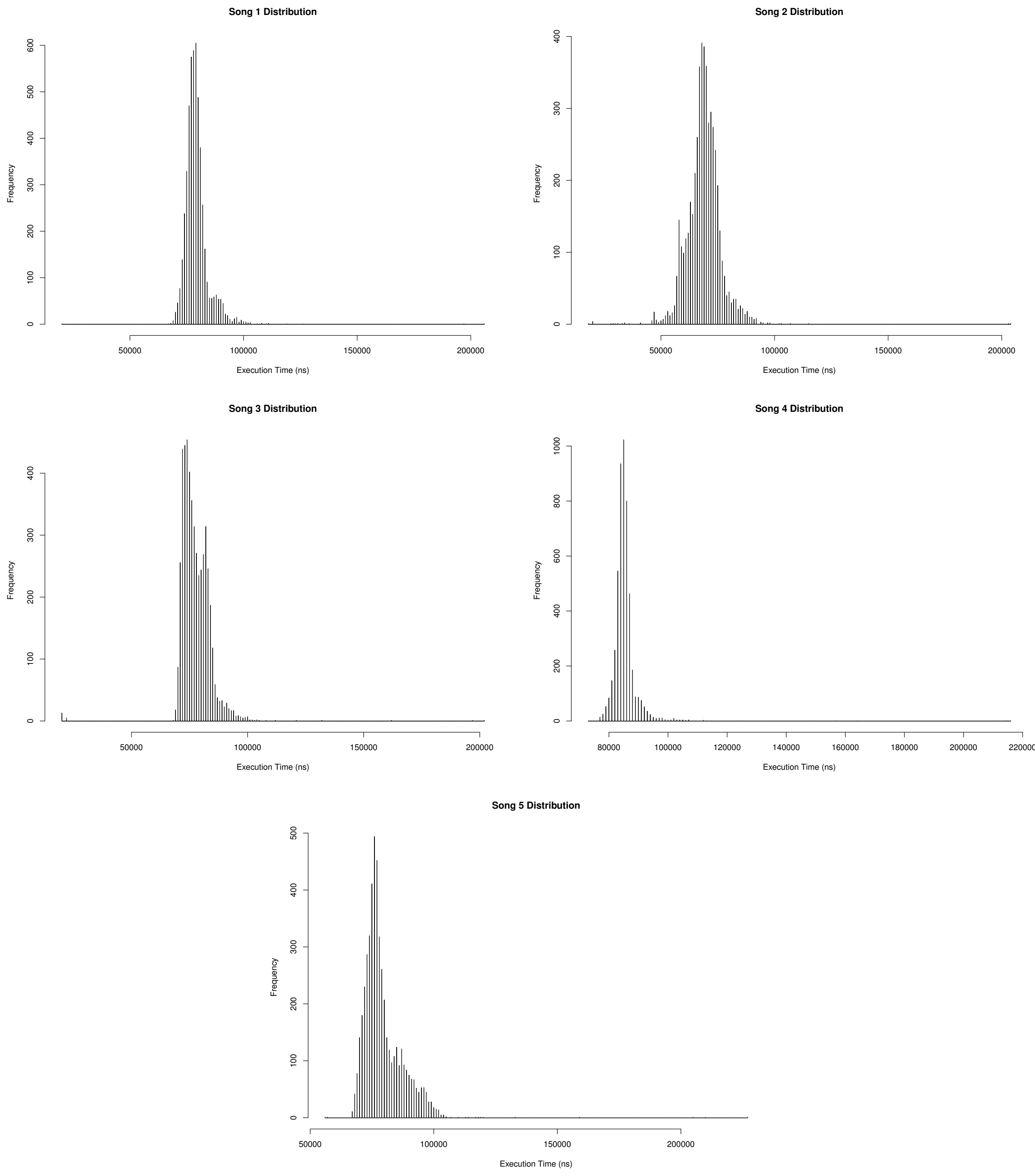

Figure 4.4: MAD execution time distributions for five different songs

Now that the distributions have been generated, let's step through the entire process of scheduling a VM using our PHA static scheduler. Recall, the motivation for our work is to identify and make use of the excess time between a task's absolute worst case execution time and our calculated c_i. In soft real-time environments some execution times may not be deterministic, like MP3 decoding, and can be described by a normal distribution curve. When this is the case, the absolute worst case execution time can be way off the distribution. This can be seen in Figure 4.5, the dashed red line shows the maximum value of the distribution curve. As can be seen, this execution time does not characterize the standard behavior of the curve. Therefore, overallocation is a distinct possibility. We decided to add a more practical estimation of WCET, this is represented by the light red dashed line. Which is located on the 99th Percentile of the distribution.

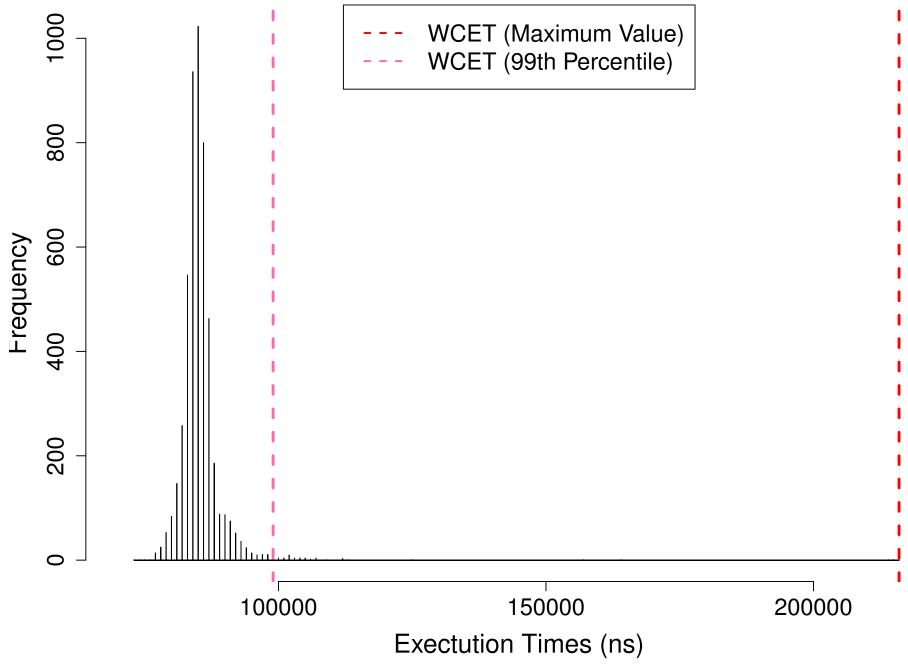

Figure 4.5: MAD distribution of song4 with execution bounds

With this new WCET, we zoom in on Figure 4.5 and create Figure 4.6. Each green line represents a new value of c_i corresponding to different values of the input ρ between the range of 10-100% probability of satisfaction.

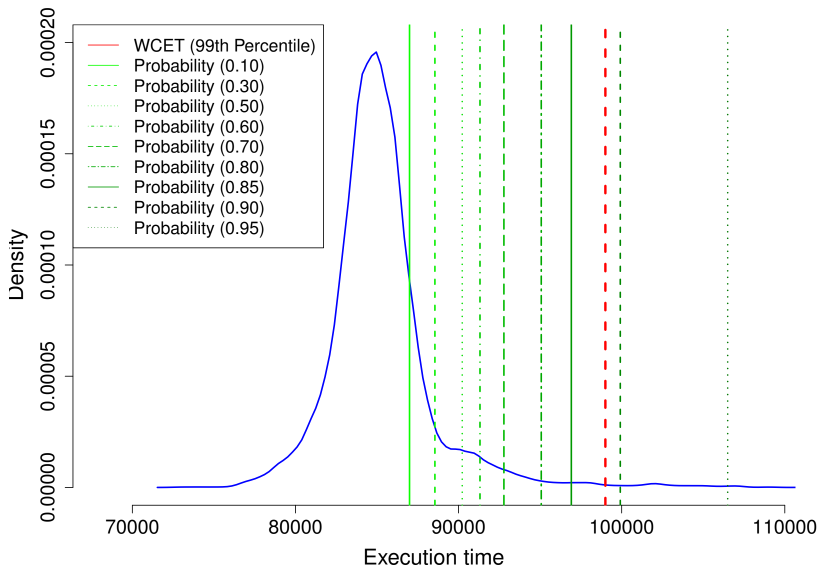

Figure 4.6: MAD distribution of song4, a closer look

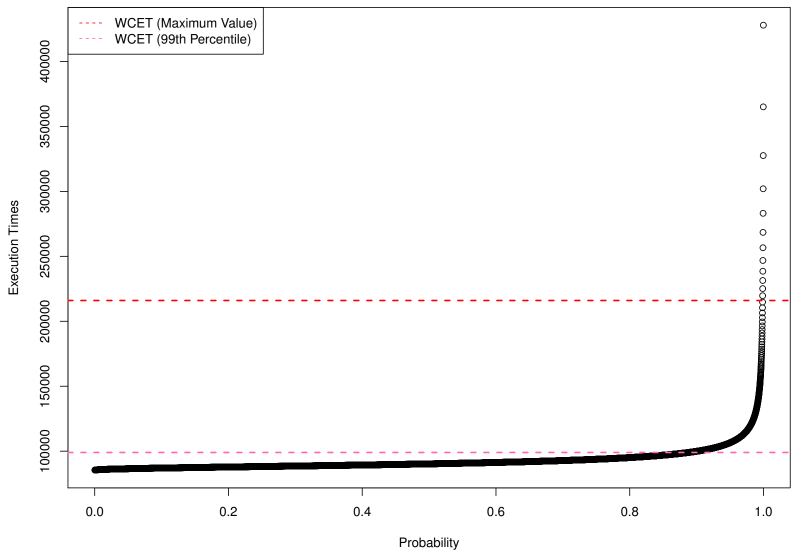

Figure 4.7: Characteristics of Wu's Chebychev equation

Figure 4.8 shows a comparison of bandwidth allocations by the CSF calculator mentioned previously. For each distribution seen in Figure 4.4, we created a task description file for. The description file contains the parameters for Chebychev and CSF. We treated the set of all tasks as our taskset, which acted as the input into the CSF calculator. The figure below, Figure 4.8, shows how the various inputs for ρ affect the overall bandwidth allocation for each VM and how they relate the worst case execution time(s).

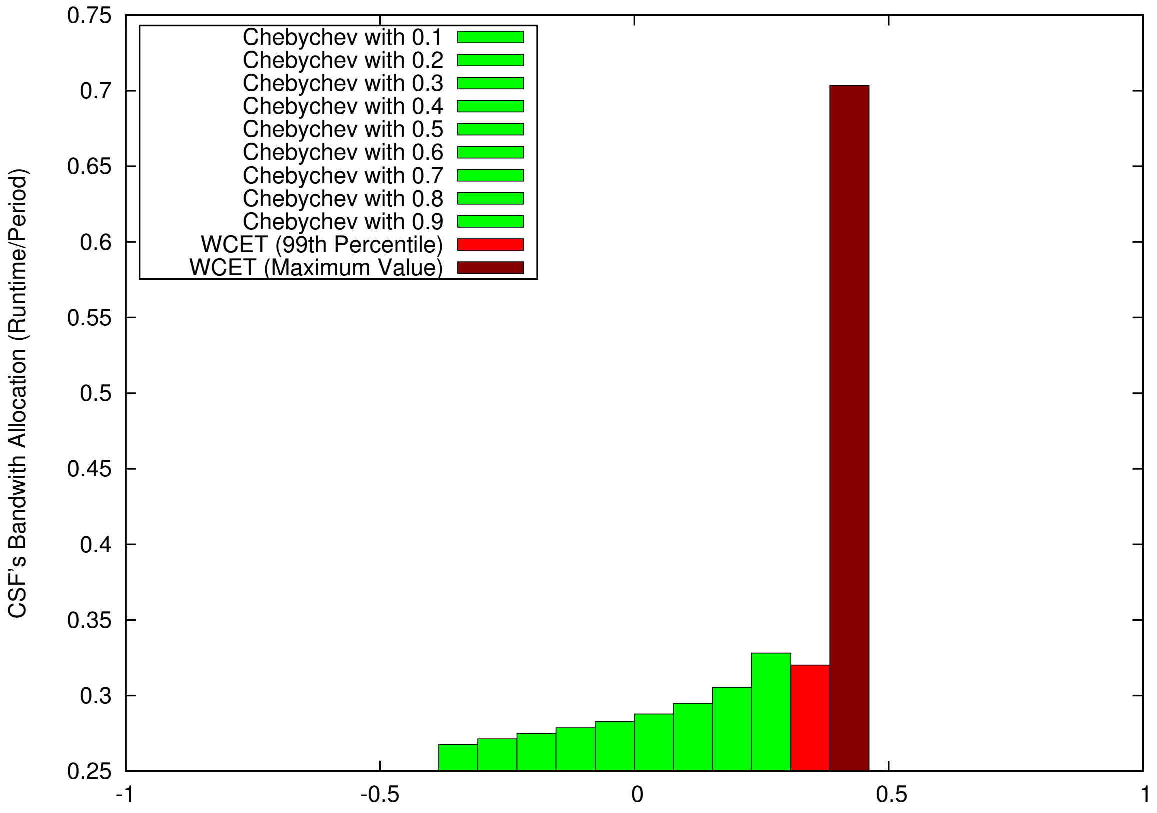

Figure 4.8: Comparison of CSF bandwidth allocations

As demonstrated first in Figure 4.5, then later in Figure 4.8, the worst case allocation selected for this example distribution is over exaggerated for soft real-time scenarios. The comparison of allocations also illustrates how many VMs could be assigned to a single processor. That is, imagine the case of wanting to run three VMs on a single processor. If we took the worst case bandwidth allocation, which utilizes greater than 70% of one processor. Therefore for the one processor case, this would not work as it requires 210% of the CPU. If we instead use the 99th percentile of the distribution as the WCET, our distribution is about 33% of the processor. Which would indeed fit on one CPU, but we lose our 100% guarantee offered by the actual worst case. Therefore, our interface allows the user to select the exact probability to fit their needs. That is, we give the user the power to vary the probability until all of their VMs fit on one CPU, while knowing what probability of deadline satisfaction is lost.

Lastly, we implemented real-time deadline satisfaction calculations into the full stack. That is, we measured the Deadline Satisfaction Ratio (DSR) for our libmad implementation when decoding the same five songs on each VM simultaneously. This is very similar to our results found in Section 4.5.2. Our libmad results can be found in Figure 4.9.

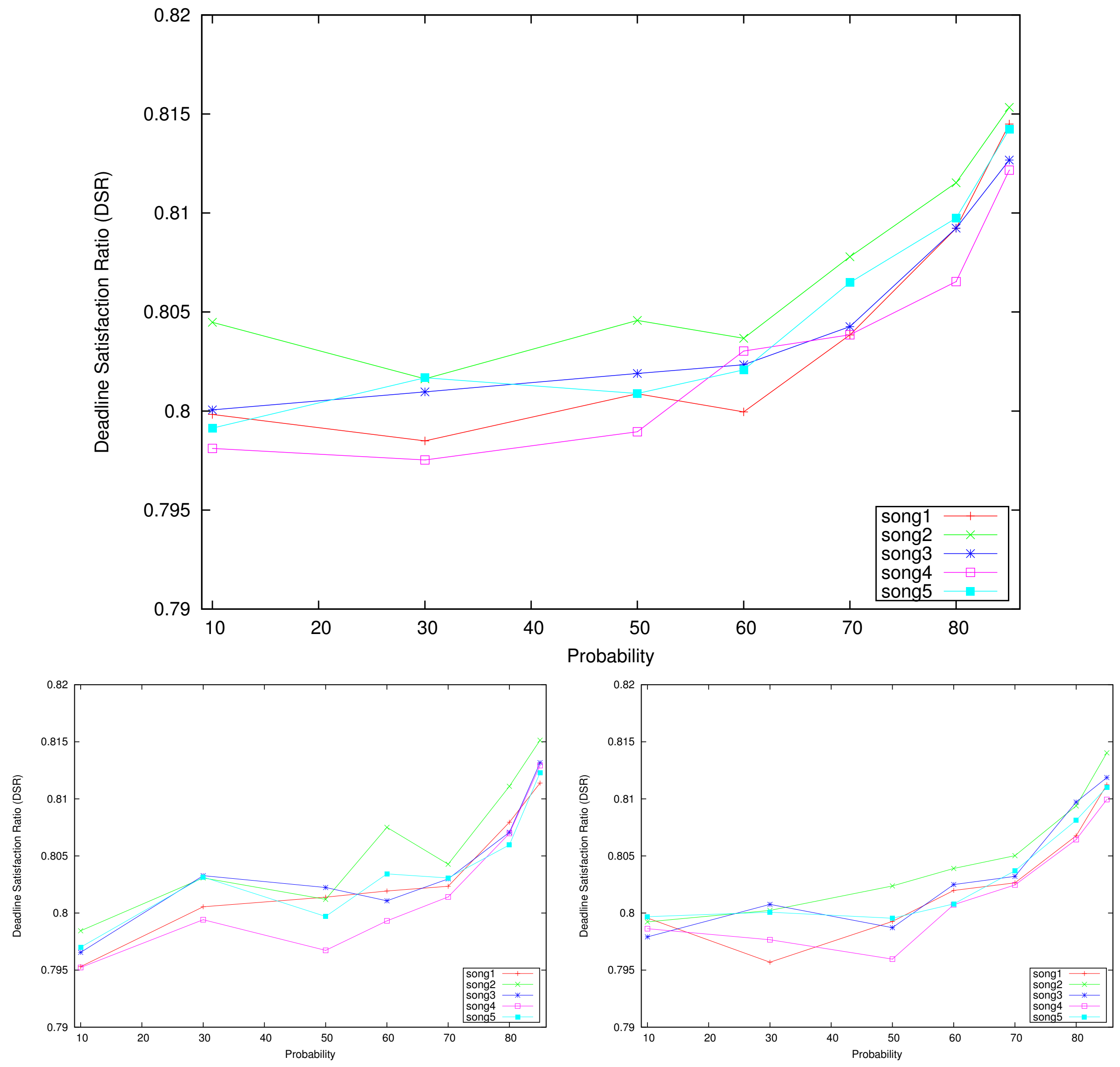

Figure 4.9: DSR using real-time Madplay for three virtual machines

These results follow similar trends to the synthetic benchmarking results, in Section 4.5.2. In that, as the probability ρ increases, the DSR increases. This again, is due to the relationship of ρ to the bandwidth allocated to each VM. We also see that each song shares similar deadline satisfaction ratios. This is due to nature of the MP3 decoder, although the delay for decoding is not defined, the distribution should not vary depending on the song being decoded.

Chapter 5

Multiprocessor Probabilistic Hierarchical Scheduling

Chip manufacturers hit a thermal barrier when creating processors capable of higher clock speeds, ultimately leading to an unsustainable increase energy costs. Thus, there was an industry shift from the uniprocessor design to multi-core architectures. As multi-core architectures become more ubiquitous, we see them being used in low power industrial systems, as well as large servers. With core counts up to 64 for one machine. With this transition, we can modify our design to utilize multi-core architectures to support a scalable number of virtual machines.

In Chapter 4, we introduced a uniprocessor static scheduler based on our Probabilistic Hierarchical Algorithm (PHA). This solution expands upon the Compositional Scheduling Framework (CSF) [33] and Wu's work in probabilistic scheduling in [31]. In this Chapter, we extend this single-core PHA static scheduler to the scope of multiple processors. We do this by means of partitioning method to statically distribute the VMs to each processor. That is, we distribute the VMs using their bandwidth allocations and the best-fit bin packing heuristic. For our solution, we assume VMs do not share memory in any way. After the processor has been assigned a set of VMs, they are scheduled following the uniprocessor solution depicted in Chapter 4.

5.1 Approach

One of the main goals of this thesis is to provide a method of reducing the amount of CPU time a real-time virtual machine is allocated. That is, we would like to allow the number of VMs to fit on the smallest amount of processors necessary. Where the necessity can be addressed by adjusting the probability ρ of each task on the VM. Leaving it to the user to determine what the level of guarantees are necessary for the system.

The complexity of the multiprocessor PHA scheduler is similar to the uniprocessor version. With the additional complexity added by the partitioning stage. As we are scheduling the VMs based on their fixed bandwidth allocations given a set of CPUs, this problem becomes synonymous with the well-known bin packing problem. The complexity of such a problem has been categorized as a combinatorial NP-hard problem. Due to the well known complexity of bin packing, efficient approximations have been developed to substitute the optimal solution. One of these heuristics is known as the best-fit decreasing strategy. We chose the strategy for our partitioning solution. This heuristic has been proven to add no more than $\frac{11}{9}$ Optimal solution $+ 1$ bins and adds O(N log N) complexity to the overall scheduling solution. The best-fit algorithm places the puts the current VM in the on the CPU that creates a "tight" fit. That is, the chosen CPU is one that would have the smallest empty space left.

5.2 Experimental Evaluations

We first present results of combining our modified CSF allocation calculator with the best-fit heuristic. We then extend the experiments performed for the uniprocessor PHA design to the realm of multi-core. The evaluation environment is similar to the one used in Chapter 4. With the addition of another processor, to evaluate our partitioning scheme. These evaluations can be split into two categories, synthetic benchmarking and practical scenario evaluations. The synthetic benchmarking is done by using our before mentions *sched_test_app* benchmark with synthetic real-time workloads. The practical scenario evaluations make use of a real-time audio decoder library we implemented to produce real-world distributions on-the-fly.

5.2.1 Number of Processors

We present a static analysis of the scalability and effectiveness of our partitioning solution in Figures 5.1 and 5.2. In these figures, we use the Baker model tasksets generated for use with our *sched_test_app*. We used the light workload utilization levels, discussed in the evaluation section of Chapter 4. We generated 100 tasksets for each step on the x-axis. That is, each VM point added to the plot is the average of 100 iterations. We start with one VM and perform our bandwidth allocation calculations based while varying the probability. Each green line represents a different probability indicated in the legend. We try to simulate different distributions when evaluating our uniprocessor PHA allocation calculator. Our solution takes in an expected value and a variance of a distribution of execution times, as inputs. For Figure 5.1 we used an expected value of $\frac{3}{10}$WCET and a variance of $(\frac{1}{6}\text{WCET})^2$. The red line represents this WCET value, it should be noted that these distribution characteristics are pessimistic representations of real-world distributions. Specifically, the distribution characteristics used for Figure 5.2, the expected value here was calculated as $\frac{7}{10}$WCET with the same variation as the previous figure. As discussed in Chapter 4, our solution does not allow for allocations larger than that of the worst case. We show these calculations for to broaden the scope of evaluations.

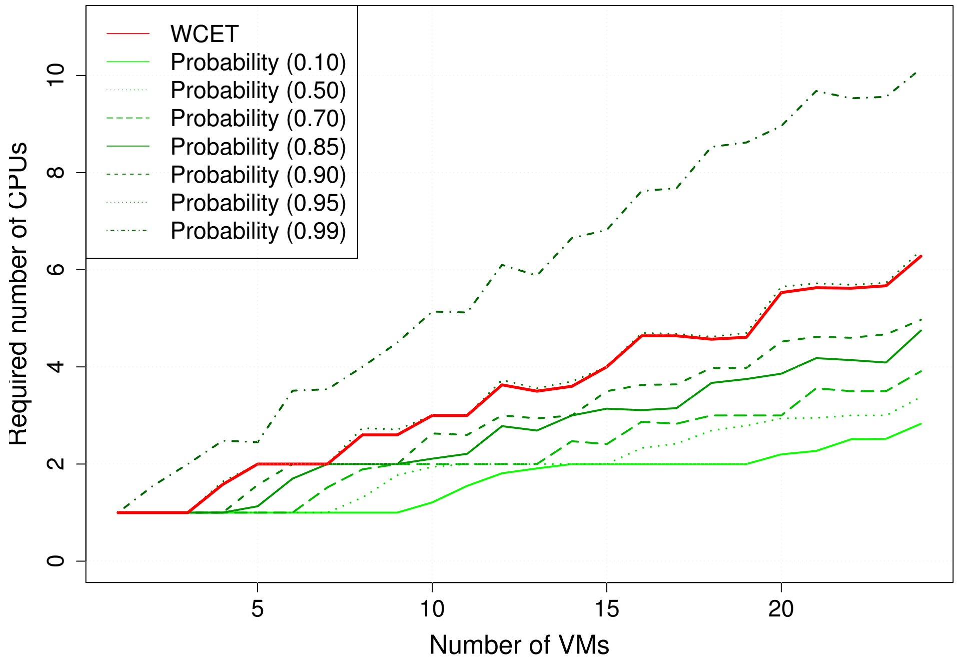

Figure 5.1: CPU allocations of distribution with expected value of 30% WCET

We see in Figure 5.1, that our solution takes less CPUs as number of VMs increase for probabilities less than 0.95. Which seems to line up with the worst case allocations. It is not unlikely for our solution to allocate more space than the worst case and here is why. With the variation and expected value input to our solution, the output is guaranteed for the probability ρ. It can only estimate the worst case, based on the variation. Thus, the worst case here is very optimistic as the expected value is so close to it. However, we can conclude from these plots that our solution will be able to scale as the number of virtual machines increases.

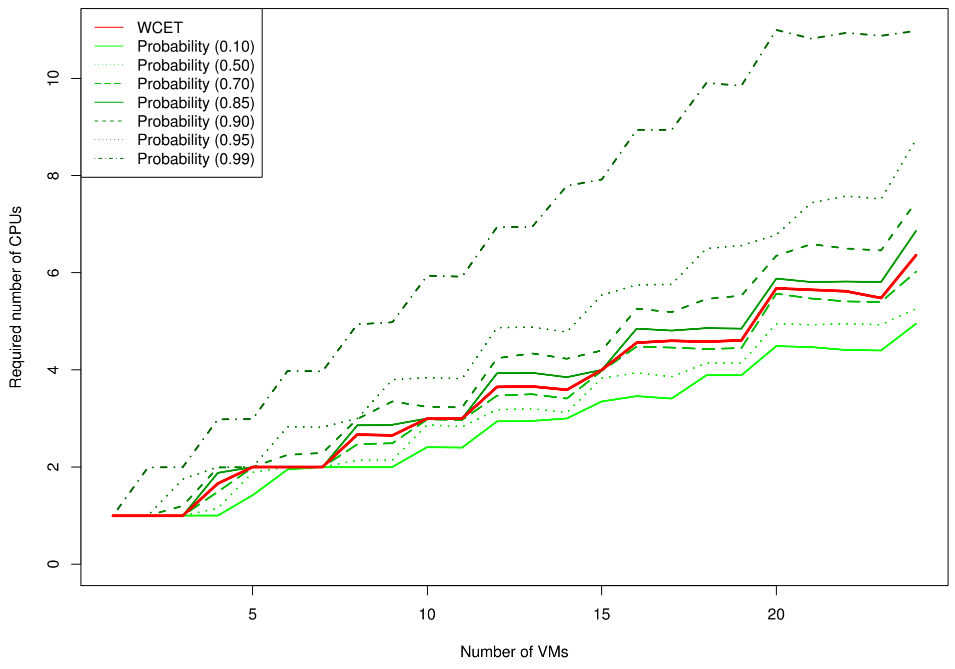

Figure 5.2: CPU allocations of distribution with expected value of 70% WCET

We show in Figure 5.2 the effects of a distribution with a higher demand. That is, the expected value is closer to the worst case bounded by WCET. As our solution has a variance of $(\frac{1}{6}\text{WCET})^2$, the guarantees require more CPU time to meet deadlines.

5.2.2 Synthetic Benchmark

We altered the single-core scheduler test configurations to accommodate our multi-core scheduler. We generated tasksets in the same manner, except for modifying the taskset's total utilization cap. For this test we created six VMs and randomly generated the utilization cap (between 0.2 and 0.3) such that the total utilization of all six VMs would be greater than 1.0. This better illustrates our solution's goal of fitting more VMs on a smaller set of CPUs. As having static utilization levels would represent the same allocations for every iteration. This is done off-line using our partitioning scheme based on the best-fit bin packing heuristic. The distribution characteristics for each task in the tasksets were calculated in the same manner, that is we simulated normal distributions with expected values equal to 30%, 50%, and 70% their WCET. We ran each test for twenty iterations and plotted the average Deadline Satisfaction Ratio (DSR) of these iterations.

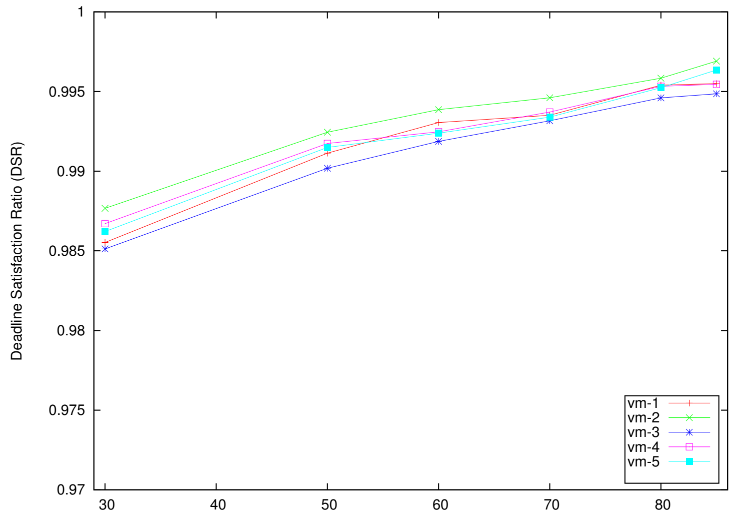

Figure 5.3: PHA for multi-processor with expected value of 30% the WCET

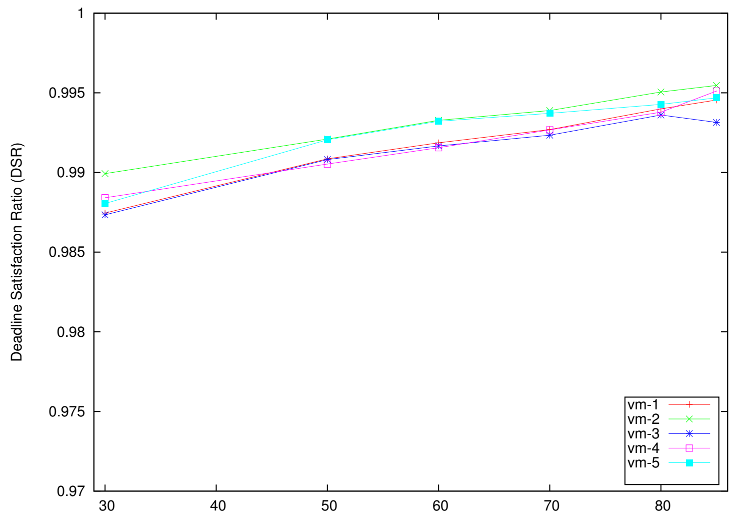

Figure 5.4: PHA for multi-processor with expected value of 50% the WCET

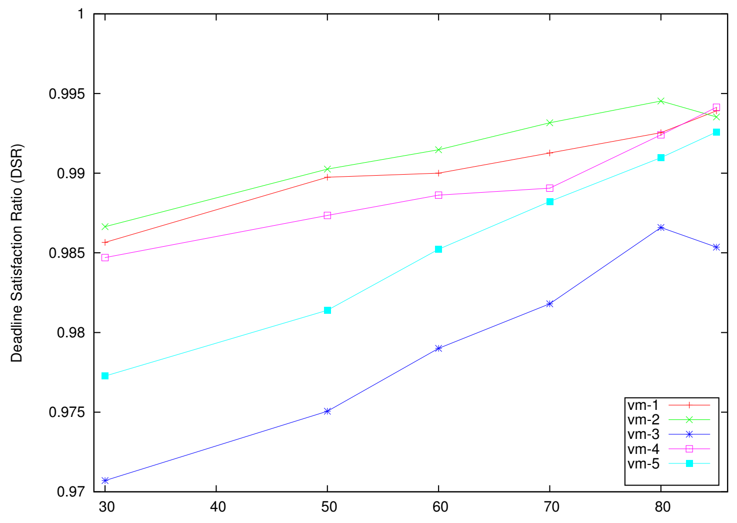

Figure 5.5: PHA for multi-processor with expected value of 70% the WCET

These results demonstrate trends similar to those found in Section 4.5.2 of Chapter 4. That is as as the value of ρ increases, so does the overall DSR of the taskset. As well as showing that as the demand of the distribution increases the overall DSR decreases. Thus, making our PHA static solution scalable to multiple processors. As these distributions are synthetic and designed to show a broad and generic scope of distribution characteristics, we take a look at how our multi-core extension of the PHA scheduler performs with an audio decoding application that generates a normal distributions more representative of real-world scenarios.

5.2.3 Practical Scenario Evaluations

We use five instances of the open source MPEG Audio Decoder library, libmad, to show how our multi-core extensions to our PHA static scheduler scale to handle more practical realistic distributions. The process of allocating a bandwidth to a VM based on the distributions, laid out by the five songs used as inputs to libmad, is identical to that described in detail in Section 4.5.3 of Chapter 4. Though, we modified the evaluation setup slightly to accommodate our multi-core scenario.

We extended the evaluations to include six virtual machines. Each running five instances of libmad, decoding a different song for each instance. We estimate the distribution characteristics used for our PHA solution based on a distribution of execution times generated while decoding 5000 frames. We modified libmad to fir our periodic model by adding a period for each frame decoded. This allows us to create utilizations of approximately 0.30. Figures 5.6 and 5.7, show the results of these tests from the perspective of each VM. This shows that temporal isolation is respected amongst VMs.

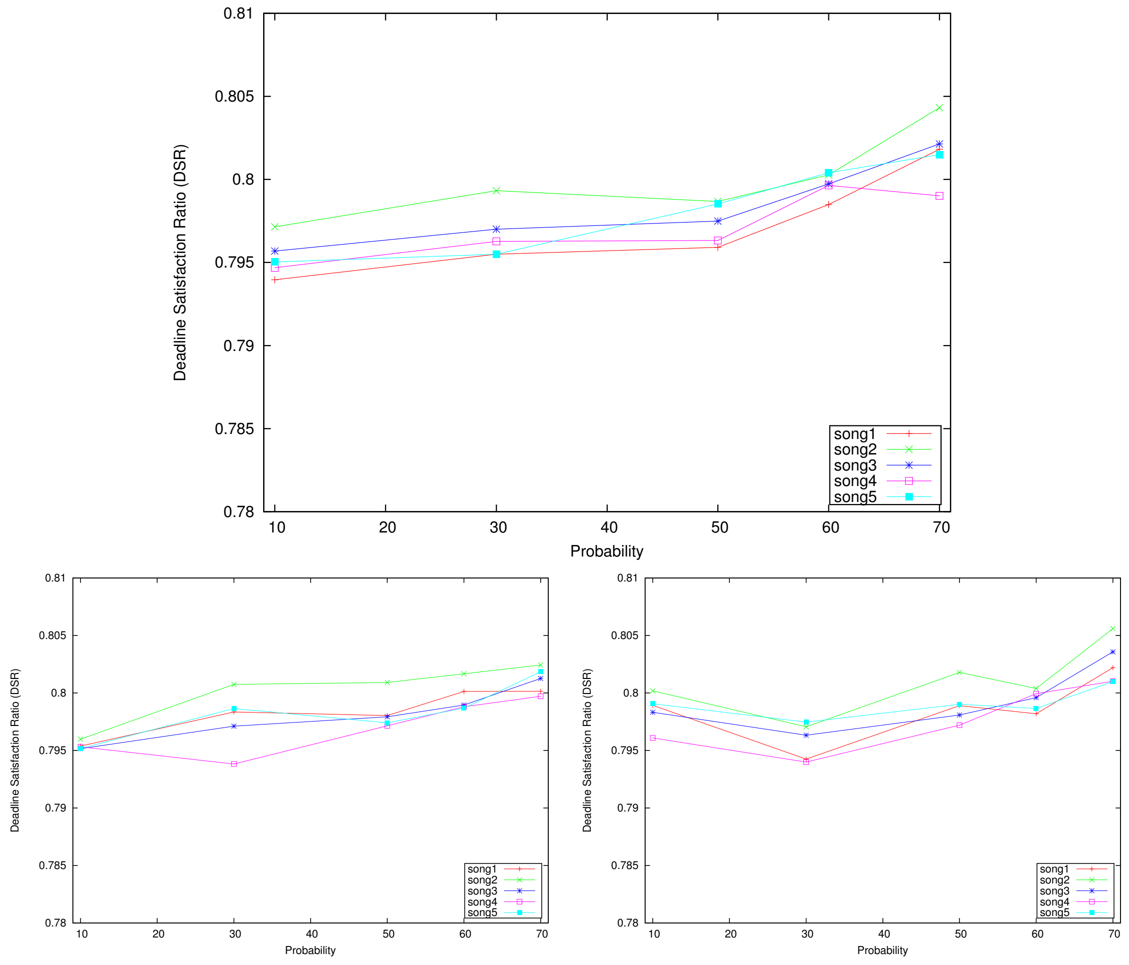

Figure 5.6: PHA for multi-processor with practical workload for VM's 1,2,3

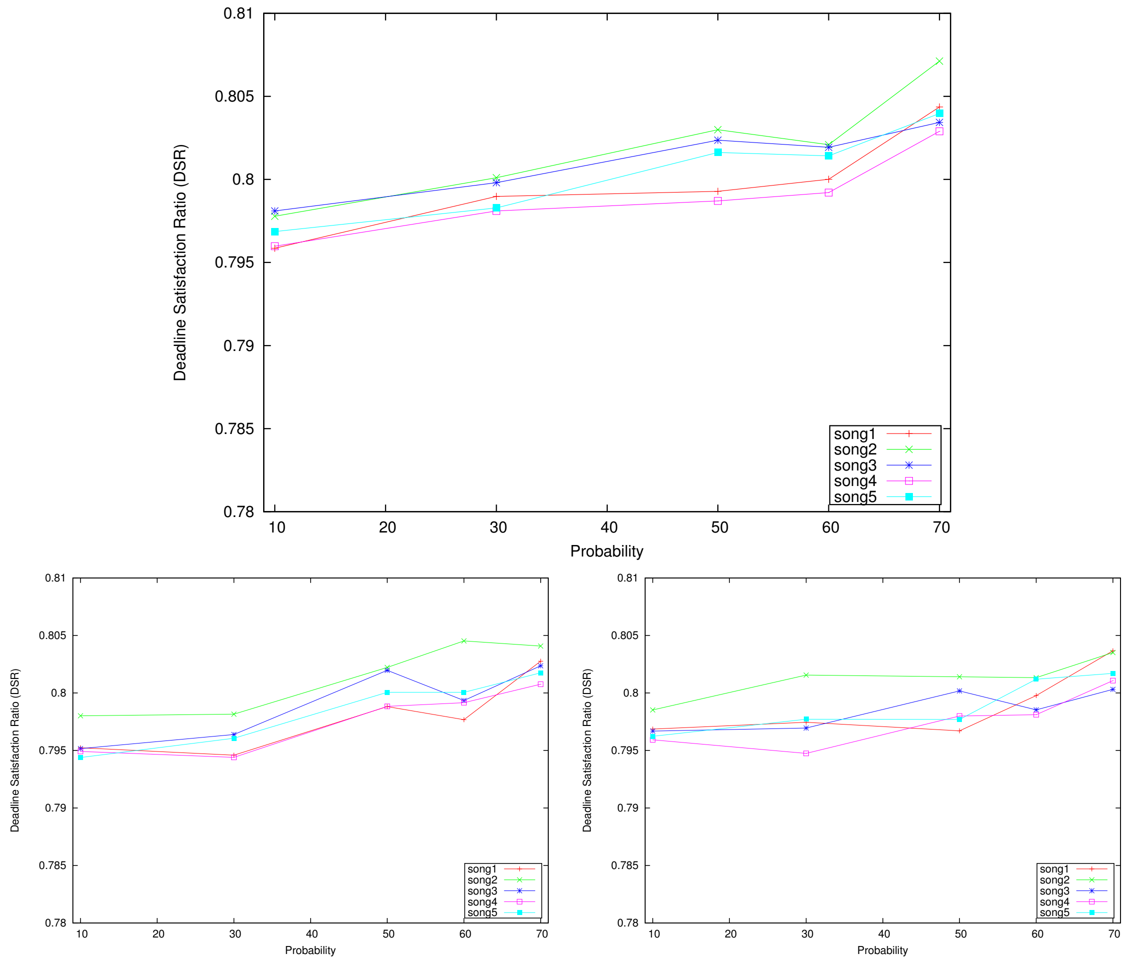

Figure 5.7: PHA for multi-processor with practical workload for VM's 4,5,6

Each virtual machine shows the trends produced by our synthetic benchmark. That is, as the probability ρ increases, the DSR of each song also increases. We also notice that extending our PHA solutions to the multi-core domain does not diminish the levels of deadline satisfaction. However, notice that we had to trim the maximum value of ρ from 85%, like used for the uniprocessor libmad evaluations, to 70%. This was an implementation restriction caused by our use of Linux's *cpuset* infrastructure to pin each virtual machine process to a processor. This, in combination with SCHED_DEADLINE does not allow for utilizations higher than 90% a given processor.

Chapter 6

Real-time Introspection

Some real-time tasks can have execution times that are not close to the allocated Worst Case Execution Time (WCET). That is, if we take a distribution of a task's execution times, the variance can be much greater than zero. Therefore, a static solution can be limited, as the output bandwidth of the scheduler will be fixed for each VM. Thus, we present a light-weight real-time introspection engine, as part of our KairosVM project, to bridge the hierarchy of completely none communicating schedulers. This contribution is material of a paper [39], that has been accepted and will be presented at the VtRES 2014 workshop.

Introspection is the concept of making the hypervisor aware to what is executing on a guest virtual machine. We choose to do this for a limited scope. The end goal of our introspection implementation is to increase the deadline satisfaction by adding a dynamic aspect to our solutions presented in Chapters 4 and 5. In this chapter we present an engine to make this possible. This mechanism needs be light-weight and must not dramatically influence the real-time guarantees on the guest. This introspection engine acts as the basis for our proposed KairosVM hypervisor, first presented in [39].

Event trapping has been used in introspection implementations like in Nitro [40]. However, the Nitro system traps all the system calls made on the guest. Nitro was developed for use in security and seeks malicious patterns on each guest. Nitro's interface provides the functionality to perform real-time system tracing. However, due to the design of Nitro, there is a massive amount of overhead. Thus, we introduce our KairosVM introspection engine to reduce these overheads to more practical levels.

6.1 Approach

Our introspection engine, residing in KairosVM, is only tacking the activity of real-time tasks running on each guest operating system. Therefore, we chose to trap real-time oriented events on the guest to present them to the hypervisor's scheduler.

6.2 Implementation

We implemented the introspection approach, described in the former section, by injecting undefined illegal instructions, presented by x86 assembly, inside the guest binary. We implement a prototype of our design in KVM version 0.12.3 running on Linux kernel version 3.4.28 and later ported to version 3.15.0-rc7.

To be able to trap events generated by guest systems, we first must perform some initialization. The guest operating system must be online. Even if the scheduling interface of the guest's real-time operating is known, the entry point is still unknown. Various techniques exist to find these points: symbol listing, disassembling, exploiting debug symbols can be performed offline on the guest's binaries. For our prototype we choose to follow the suggestions by [41, 42] and make use of the Linux kernel's *System.map*. The *System.map* contains the symbol list of the kernel and is usually installed to the `/boot/` directory in Linux and can also be accessed via the `/proc/kallsyms/` kernel interface.

For our prototype we use a Linux variant, ChronOS, as the guest operating system. Therefore, we choose to target system calls specifically. Assuming the system call numbers are known to us, via our pluggable interface, we can derive them using the `sys_call_table` symbol from *System.map*. The `sys_call_table` can be accessed in various other ways, but they are more complicated.

Once the address of the entry point is known, it is stored along with the first m bytes. The m bytes are stored in a hashmap, for use with KVM's emulation capabilities later. This entry point address is used to inject the undefined instruction used in the event trapping process.

Our event trapping mechanism injects an illegal instruction, specifically undefined instruction, *ud0*, at the addresses where each real-time scheduling function is located. The *ud0* instruction was specifically selected, as the *ud1* and *ud2* instructions presented by x86 assembly, are much more commonly used by the Linux kernel. For example, the *ud2* instruction is used by the kernel's `BUG()` macro, this would therefore generate a large amount of false positives. The undefined instructions are undocumented in Intel's x86 specifications, however AMD's documentation can be found here [43]. By default, Intel's VTx virtualization extensions trap undefined instructions via KVM, this is in order to emulate instructions that are undefined by a guest. Our implementation makes use of this, by inserting the *ud0* instruction, a real-time specific event will trigger the pre-existing traps provided by the processor's virtualization extensions. Thus a context switch is made into the hypervisor.

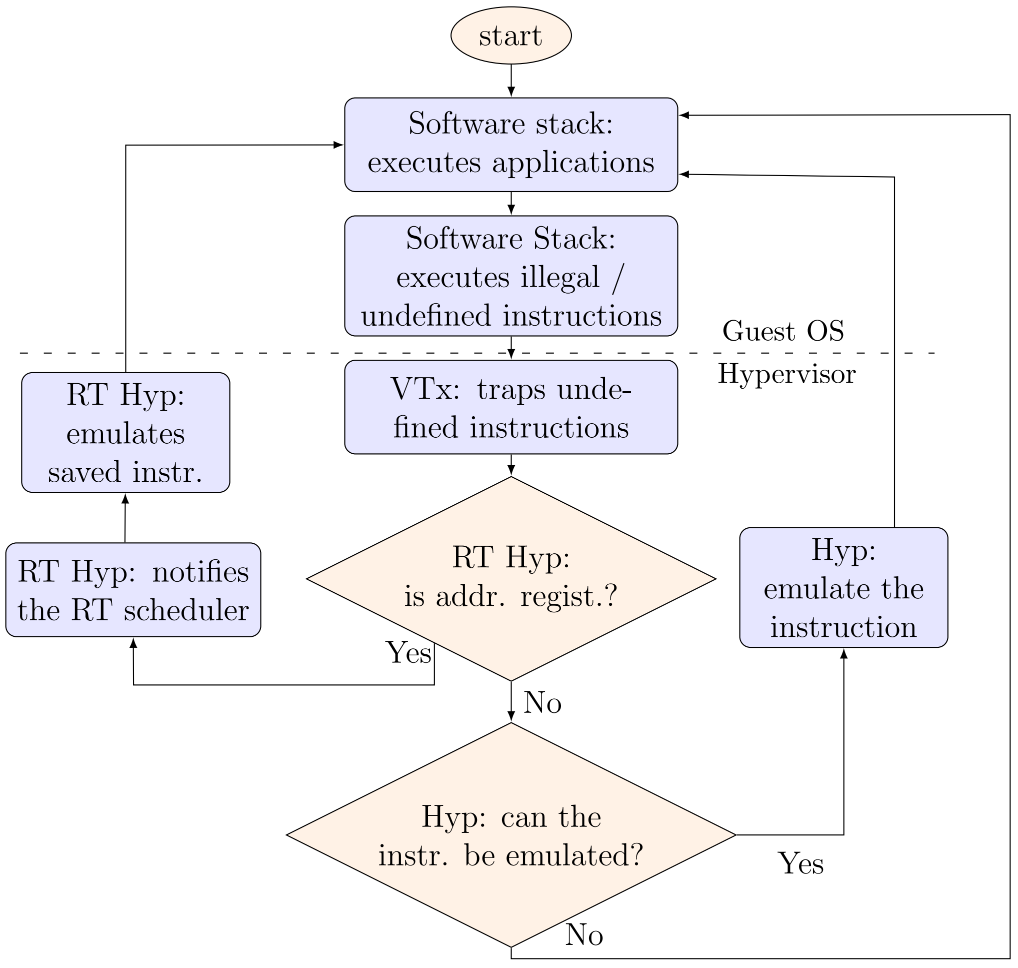
start
Software stack: executes applications
Software Stack: executes illegal / undefined instructions
Guest OS
Hypervisor
VTx: traps undefined instructions
RT Hyp: is addr. regist.?
Yes
RT Hyp: notifies the RT scheduler
RT Hyp: emulates saved instr.
No
Hyp: can the instr. be emulated?
Yes
Hyp: emulate the instruction
No

Figure 6.1: Introspection implementation flow chart

Before the execution of the guest virtual machines, the system specific addresses are inserted into a hashmap by the initialization phase. Our implementation references this map when a trap is triggered, recall the host is now in kernel mode, if it detects an undefined instruction that it did not inject, the hypervisor will continue its original execution path. If the undefined instruction was injected by our implementation, this is where we send the guest task's real-time parameters is passed to the host's real-time scheduler. Once the real-time information is passed along, we emulated the real-time system call on the guest. This is done by use of `x86_emulate_instruction()`, provided by KVM. Once emulation is complete our introspection engine returns control to the virtual machine.

Due to our introspection engine implementation using system addresses to trap events, we are not dependant on any one real-time operating system for guests. That is, our implementation allows us to present the user with an interface to change the real-time events trapped to those required by their real-time guest operating system. Therefore, our only restriction for guest operating system is they need a separate address space for kernel and user space. More specifically, guest operating systems that implement one common address space for all processes, like VxWorks, are not supported. For those operating systems that are supported, we present a plugin interface, as illustrated in Figure 6.2.

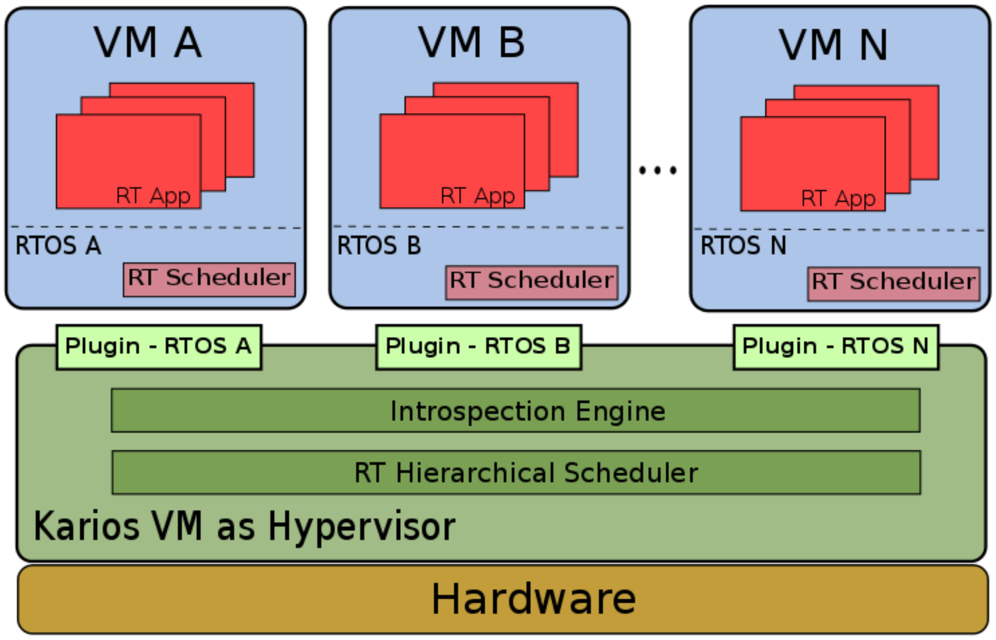

Figure 6.2: KairosVM architecture layout

6.3 Experimental Evaluation

We performed several experiments to evaluate our lightweight real-time introspection engine. These experiments can be separated into two different categories, overhead measurements and real-time synthetic benchmarks. The overhead measurements measure how much time it takes to execute a real-time system call with and without our trapping mechanism enabled, we compare these results with those generated using the Nitro system's mechanism. We do these evaluations in two different scenarios. First, we only run an application that makes real-time system calls with nothing else running on the guest system. Second, we run a none real-time system call intensive application in the background while repeating the first scenario. The second group of evaluations are synthetic real-time application. We make use of our *sched_test_app* benchmark, described in detail in Section 4.5.2 of Chapter 4. We ran this benchmark to determine the effect of our introspection on the real-time performance.

6.3.1 Evaluation Environment

We ran the tests on an eight core Intel Xeon E5520 processor at 2.27GHz and 16GB of RAM. The host operating system is Ubuntu Server 10.04 with Linux kernel version 3.4.82, patched with ChronOS 3.4, and KVM/QEMU version of 0.12.3 as the hypervisor. For the guest operating system we used Ubuntu Server with Linux kernel 3.0.24, patched with ChronOS 3.0. Our comparisons were using vanilla KVM, KVM with our introspection modifications, and Nitro on a set of benchmarks. We backported Nitro from Linux 3.13.0-rc8 to Linux 3.4.82 for fair comparisons.

The tests were run on a single VM with a single VCPU. This VCPU process was pinned to a single processor. We isolated the processor using `isolcpus`. We also heed the advice of Kiszka in [19] and raised the priorities of all QEMU threads.

6.3.2 Overhead Measurements

For the overhead measurements, we created a simple benchmark to calculate the latency of each system call required for for a real-time application, running on our ChronOS patched Linux kernel. This simple test ran for 1000 iterations, where the plotted value was the average latency. We plot these overheads in Figure 6.3. As in our introspection engine implementation the user has the power to select which system calls she/he wants trapped, we select only *begin_rtseg* and *end_rtseg* (Notice, they are indicated by a * in Figure 6.3). In the interface provided by ChronOS, these system calls contain parameters necessary for the real-time host scheduler. Specifically, a task's period, deadline, and execution time. Conversely, the Nitro introspection engine instead captures all of the system calls made on the guest operating system. It then uses filtering techniques to present the user with the requested information.

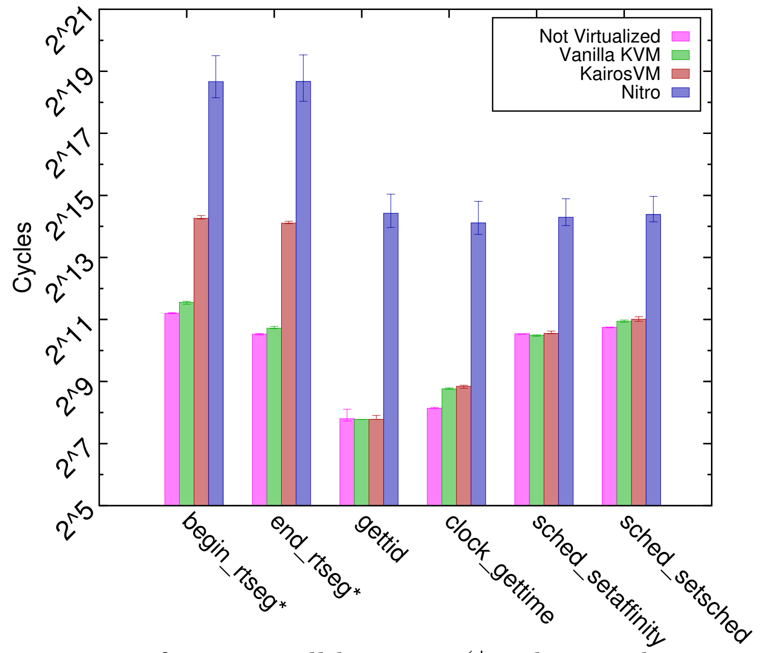

Figure 6.3: Comparison of system call latencies (* indicates the system call being trapped/-traced in KairosVM/Nitro)

The Figure 6.3 shows that our KairosVM introspection engine is a large improvement over the state-of-the-art Nitro implementation, for trapped system calls. The overhead incurred is less than half that of Nitro when compared to the vanilla KVM latency. For the system calls that are necessary for a real-time application, though not necessary for a real-time host scheduler, our implementation overhead is the same as vanilla KVM. However, Nitro's engine has latencies that far exceed our overhead, even when our syscall trapping is engaged. That is, the overheads of *begin_rtseg* and *end_rtseg* for our KairosVM implementation are lower than that of Nitro for system calls that we are not interested in. It is Nitro's filtering techniques that add the additional overhead seen for the two rtseg system calls. For use in real-time systems, it is also important the latencies are consistent and bounded. We show in Figure 6.3 error bars using the standard deviation of the measurements. We see that Nitro is thus unsuitable for real-time environments, as the variance is quite large when compared to the other results.

Therefore, we conclude that our KairosVM introspection engine is suitable for real-time hierarchical solutions. To further our analysis, we evaluate the effect that this latency has on real-time performance of our synthetic real-time benchmark.

6.3.3 Synthetic Benchmark

The goal of this introspection is to provide a mechanism for use in real-time virtual machine schedulers. Thus, we need to provide evaluations to the effect the overhead incurred has on real-time performance of a virtualized real-time operating system. Thus, we provide a set of tests to determine the depth of the effect each mechanism has on our synthetic real-time benchmark, *sched_test_app* which is packaged with our ChronOS distribution.

Our test application takes in descriptions of tasksets based on the Baker model [36]. For these evaluations we used a synthesized medium workload. According to the Baker model, a medium workload is defined by the execution time generation bounds. The bounds of generation for a medium workload are 10% and 40% the task's given period. Thus, the utilization per task, defined as $\frac{C_i}{T_i}$, is between 0.1 and 0.4. All tests compare deadline satisfaction ratios (DSR) when running with our KairosVM engine, the Nitro introspection engine, and with vanilla KVM as a control. These are performed for processor utilization steps of (.25, .50, .75, and 1.0).

We used two scenarios for the tests we ran. We first ran our benchmark without any background tasks. That is, the only system calls called during the tests were by our synthetic real-time application. Our second scenario was rerunning our tests with a system call intensive application, specifically Bonnie++ [44]. Bonnie is a benchmark developed to evaluate hard disk and file system performance. Therefore, the majority of system calls are I/O related. Every point on the following plots are averaged out of twenty iterations. Figure 6.4 shows the results from the first scenario of tests. As total utilization increases, we expect more deadlines to be missed. Therefore, the trend of the "No Introspection" curve is expected. This plot confirms that our introspection is not intrusive to the real-time guarantees of our real-time application. Whereas, the number of deadlines satisfied diminishes for the Nitro case.

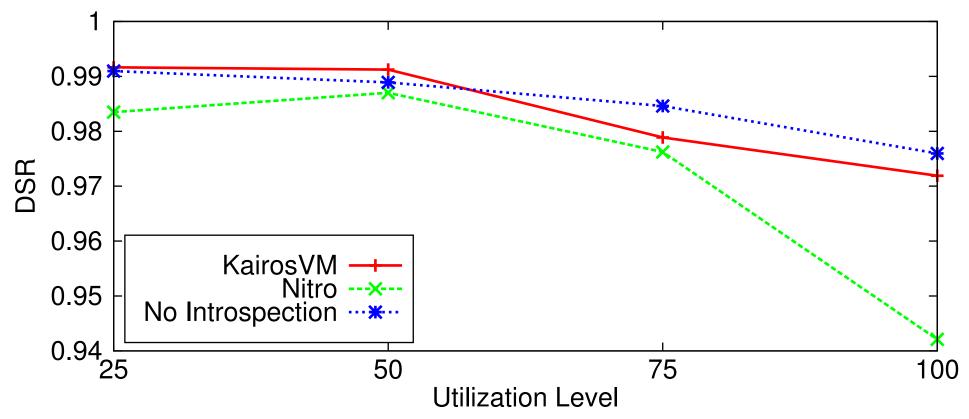

Figure 6.4: DSR of synthetic real-time application

We then introduce the Bonnie application in the background and thus perform the second category of tests, these results can be seen in Figure 6.5. We see similar trends in the DSR characteristics to the first scenario. Notice, that the deadline satisfaction ratios do not drop when the background noise is added, specifically for the Nitro case. As Nitro is implementing using filtering, we expect background system calls to provide excess overhead and thus effecting the overall ratio. However, the guest is a real-time system, and these real-time system calls have priority over the Bonnie lower priority I/O system calls.

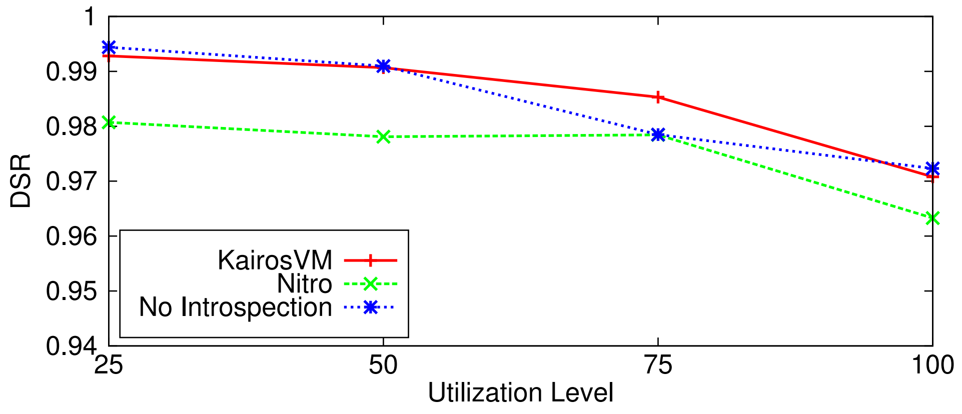

Figure 6.5: DSR of synthetic real-time application and background application

There are real-world scenarios that may require background system calls in a real-time system. For example, a data logging application needs I/O capabilities without real-time priority. To accommodate these types of situations, we include 6.1. This table illustrates the total number of system calls the Bonnie application was able to execute during the real-time evaluations. These numbers were generated using Linux's system call tracer package, *strace*. We ran these evaluations for the two more heavily utilization cases. These results show that due to the time the hypervisor scheduler needed to process the Nitro mechanism, it could not execute the guest virtual machine. Therefore, the guest did not have enough CPU time allocated in order to run as many system calls. The stars on the table indicate which case had the highest number of system calls. As can be seen, our introspection implementation is similar to the vanilla KVM control case.

Table 6.1: Number of syscalls and writes according to the utilization

Util	**# of Syscalls**	**# of Writes**	**Case**
75	4147015	3479204	KairosVM
	3236017	3175538	Nitro
	⋆ 4153681 ⋆	⋆ 3481426 ⋆	No Introspection
100	4065685	⋆ 3452094 ⋆	KairosVM
	1282703	1282300	Nitro
	⋆ 4066879 ⋆	3452492	No Introspection

In conclusion, our KairosVM introspection engine is a first step in building a full real-time virtual machine scheduling system for the Linux/KVM virtualization platforms. We show that our evaluation does not cause a major decrease in real-time performance, unlike the stat-of-the-art alternative.

Chapter 7

Conclusions

Consolidation of multiple real-time systems on the same hardware platform requires real-time aware schedulers for the hypervisor. Past work in this area concentrate on strict hardware partitioning techniques based on worst case execution time of guest real-time tasks. These partitions are typically overly pessimistic when scheduling virtualized soft real-time systems. In this thesis we target the stochastic nature of execution times in soft real-time systems to drastically reduce the amount of CPU allocated to each virtual machine. As the execution times vary depending on the requirements of a specific soft real-time system, we present a framework to provide a flexible solution to allow the virtual machine bandwidth allocations to be fine-tuned for each specific use case. We developed a infrastructure in Linux, to test the effectiveness of our solution when scheduling virtual machines. Evaluations showed that the bandwidth allocations generated via our Probabilistic Hierarchical Algorithm (PHA) drastically reduced the resource consumption for each VM without greatly diminishing deadline satisfaction on each guest. To support the virtualization of legacy real-time systems, we demonstrate that a lightweight introspection mechanism can be used to collect the real-time characteristics of a guest on the fly. Results indicate the our prototype introspection implementation is non-intrusive to the real-time performance of a virtualized guest.

7.1 Contributions

To summarize, the major contributions of this thesis are:

1. **We developed a framework for the offline allocation bandwidth for virtual machines that reduces resource consumption.** Each guest's taskset was modified to include a new execution demand cap via the Chebychev equation, given the expected value and variance for each task's distribution of execution times. Once calculated, we input these new execution time bounds to the compositional scheduling allocator,

which output a bandwidth for each guest. These allocations are then presented to the underlying host scheduler. These bandwidth allocations provide a guaranteed level of deadline satisfaction.

2. **We developed the virtualization infrastructure necessary to fully evaluate and demonstrate our proposed schedulers.** Making use of Linux's new SCHED_DEADLINE policy we were able to schedule guest virtual machines, via Linux's KVM and QEMU, using our bandwidth calculations. We used this infrastructure to perform extensive evaluations of our PHA scheduling policies. These evaluations included real-time synthetic benchmarks, using light uniform loads representing a broad scope of soft real-time system characteristics. These results showed that the guarantees provided by the Chebychev equation were met. We then constructed a soft real-time audio decoder scenario to evaluate the real-time performance of our solutions, when none synthetic distributions are generated on the fly. We lastly presented comprehensive offline analysis of the scalability of our solutions for multi-processor systems. Leaving us with the conclusion that our solution maintains high deadline satisfaction while requiring a small amount of CPU time. We also demonstrate the our PHA solution scales to multiple processors when adapting static partitioning techniques. As our main motivation for this work is to reduce the hardware resource requirement for each guest the migration from a uniprocessor to a multiprocessor platform is paramount. Our evaluations indicate that our solution successfully reduces the number of CPUs allocated without drastically decreasing the deadline satisfaction on each guest.

3. **We developed a lightweight introspection engine for use in dynamic real-time schedulers.** To support legacy real-time systems, it must not be necessary to modify the guest's operating system. That is, a full virtualized environment is required. For this reason, paravirtualized methods of dynamically scheduling virtual machines are not useful. Our introspection engine implements a generic real-time event trapping mechanism, which can easily be modified to support any real-time operating system on the guest. Existing introspection implementations trap all system events on the guest operating system. Therefore, the overhead incurred by these mechanisms can drastically alter system performance. This is especially true when introducing real-time applications to the guests. We show that our introspection implementation does not cause real-time deadline satisfaction to drastically decrease.

Chapter 8

Future Work

There are many avenues to pursue to expand upon the research presented in this thesis. In Section 1.3 of Chapter 1 we outline the restrictions in our scope in order to limit the variability and to identify specific characteristics of our proposed solutions. Due to the magnitude of variation one could perform on the scope of this thesis, this creates extensive opportunities for future research contributions.

8.1 Extended Evaluations

Due to the complex nature of our scope presented in this thesis, there are many tunable parameters for evaluation. Following the stochastic model, we generated synthetic distributions for our real-time simulations, as our target distribution generated by our soft real-time audio decoder scenario followed a normal distribution. We chose to evaluate a normal distribution of execution times. We then varied the expected value of the distribution from 30% to 70% the worst case execution time bound. However, the standard deviation for our evaluations was fixed to $\frac{WCET}{6}$. Therefore, for future work we propose evaluating our solutions using distributions generated with varying standard deviations, and in turn variations. We predict that this variation would provide a much wider range of c_i's generated by the Chebychev equation. To expand this concept, the evaluations could also be extended to support other types of distributions.

Specifically, extending the evaluations to support a continuous uniform distribution. This would represent a stochastic system where the execution time of a task is completely indeterministic. As the Chebychev equation only requires a distribution's expected value and variance, this would demonstrate that traditional worst case bounding techniques would not accommodate this kind of distribution.

It would also be beneficial to perform full evaluations while randomizing the real-time parameters of the tasks for each guest. Specifically, as the periods of each task. For our evaluations we manipulated the periods of each task in order to provide a fixed utilization level for each guest's taskset. As the utilization levels of real-time systems may not be known beforehand, this would simulate such types of environments. Our solution does not depend on certain utilization levels of tasks and these tests would demonstrate the scalability of our solution.

8.2 Theory Extensions

As demonstrated in Section 5.2 of Chapter 5, our current derivation of Chebychev's inequality is restricted to a minimum c_i output equal to the distribution's expected value. That is, we would like to expand the range of allocations by using a formula without this limitation. An attractive quality of Chebychev's equation is that it only takes in two characteristics of the distribution, the expected value and the variance. Therefore, it does not require a certain type of distribution in order to calculate a solution. Using an equation that considers specific distribution types could be an interesting avenue and could possible produce tighter allocations.

Legacy systems can have the dependency of multiple CPUs. That is, once virtualized a guest should not be limited to a uniprocessor. We propose extending our model to support multiple virtual processors for each real-time guest. One proposition would be to support strict partitioned scheduling on each guest and later, to include global real-time schedulers. This would be more representative of real-world systems, as well it would introduce an appealing research scenario for a hierarchy of multiprocessor schedulers.

8.3 Introspection

As our introspection engine enables a host scheduler to be informed about the real-time characteristics of tasks running on virtualized guests, we encourage its use with hierarchical scheduling research and development. As we present a pluggable interface design with our introspection engine, we would like to see plugins for various other guest real-time operating systems developed and integrated. Using ChronOS as the only guest during evaluations limit the scope in several ways. One such way is that ChronOS relies on two system calls per real-time job. As we witness overheads associated with system call trapping, introducing real-time interfaces with a smaller number of system calls required is an interesting prospect. Granted, this is highly dependent on the needs of the scheduler implemented on the host.

As our PHA solutions are calculated and partitioned offline, they inherently cannot handle unpredicted events like early and late real-time task terminations. Therefore, we propose the integration of our introspection mechanism to create a dynamic solution for these events as future work. More specifically, we would like to see the support of online adjustments of bandwidth allocations, based on the on-the-fly real-time characteristics gathered by our introspection mechanism.

8.4 Scope/Model Expansions

As described in Chapters 1 and 3 the scope of this these was limited by several assumptions. One of these assumptions was using solely CPU intensive real-time tasks. That is, we assume there are not task interdependencies and no I/O operations residing inside the guest's real-time tasks. To make our work more generic and universally applicable, we propose the accommodation of such dependencies as future work. Our scope was also limited by our use of the periodic model, real-time tasksets that follow the sporadic and aperiodic models would still witness smaller resource reservations from our partitioning solution. Therefore, we propose the extension of our work to fit these models as future work.

Bibliography

[1] M. Dellinger, P. Garyali, and B. Ravindran. Chronos linux: A best-effort real-time multiprocessor linux kernel. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 474–479, New York, NY, USA, 2011. ACM.

[2] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmus rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 111–126, Washington, DC, USA, 2006. IEEE Computer Society.

[3] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An edf scheduling class for the linux kernel. In *Proceedings of the Real-Time Linux Workshop*, 2009.

[4] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari. Hierarchical multiprocessor cpu reservations for the linux kernel. In *Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT '09, pages 9–17, june 2009.

[5] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, January 1995.

[6] Z. Deng and J. W S Liu. Scheduling real-time applications in an open environment. In *Proceedings of The 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 308–319, Dec 1997.

[7] G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the Seventh Real-Time Technology and Applications Symposium*, RTAS '01, pages 26–, Washington, DC, USA, 2001. IEEE Computer Society.

[8] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, ECRTS '03, pages 151–158, July 2003.

[9] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 1(2):257–269, April 2005.

[10] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 249–258, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Wind River. Wind River Hypervisor. http://www.windriver.com/.

[12] National Instruments. National instruments Real-time Hypervisor. http://www.ni.com/.

[13] TenAsys. TenAsys Real-time Hypervisor. http://www.tenasys.com/.

[14] Real Time Systems GmbH. RTS Real-time Embedded Hypervisor. http://www.real-time-systems.com/real-time_hypervisor/index.php.

[15] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *ACM Sigplan Notices*, volume 45, pages 97–108. ACM, 2010.

[16] A. Masrur, S. Drössler, T. Pfeuffer, and S. Chakraborty. VM-based real-time services for automotive control applications. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (Short Paper)*, Macau SAR, P.R.China, 2010.

[17] S. Xi, J. Wilson, C. Lu, and C. Gill. Rt-xen: towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 39–48. IEEE, 2011.

[18] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky. Realizing compositional scheduling through virtualization. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 13–22. IEEE, 2012.

[19] J. Kiszka. Towards linux as a real-time hypervisor. *RTLWS11*, 2009.

[20] T. Cucinotta, F. Checconi, and D. Giani. Improving responsiveness for virtualized networking under intensive computing workloads. In *Proceedings of the 13th Real-Time Linux Workshop*, 2011.

[21] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig. Flattening hierarchical scheduling. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 93–102. ACM, 2012.

[22] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Proceedings of the 6th International Conference on Advances in Information and Computer Security*, IWSEC'11, pages 96–112, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] F. Azmandian, M. Moffie, M. Alshawabkeh, J. Dy, J. Aslam, and D. Kaeli. Virtual machine monitor-based lightweight intrusion detection. *SIGOPS Oper. Syst. Rev.*, 45(2):38–53, July 2011.

[24] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[25] J. Hizver and T. Chiueh. Real-time deep virtual machine introspection and its applications. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 3–14, New York, NY, USA, 2014. ACM.

[26] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 14–24, New York, NY, USA, 2006. ACM.

[27] T. Cucinotta, G. Anastasi, and L. Abeni. Respecting temporal constraints in virtualised services. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 02*, COMPSAC '09, pages 73–78, Washington, DC, USA, 2009. IEEE Computer Society.

[28] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli. Adaptive real-time scheduling for legacy multimedia applications. *ACM Trans. Embed. Comput. Syst.*, 11(4):86:1–86:23, January 2013.

[29] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 1973.

[30] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 149–163, 2003.

[31] H. Wu, B. Ravindran, E.D. Jensen, and P. Li. Cpu scheduling for statistically-assured real-time performance and improved energy efficiency. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 110–115, 2004.

[32] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems. In *Proceedings of the 4th ACM International Conference on Embedded Software*, pages 64–73, 2004.

[33] Insik Shin and I Lee. Compositional real-time scheduling framework with period model. *ACM Trans. Embed. Comput. Syst.*, 2008.

[34] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 4–, Washington, DC, USA, 1998. IEEE Computer Society.

[35] L. Abeni. Server mechanisms for multimedia applications. Technical report, 1998.

[36] T. P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and EDF scheduling for hard real time. Technical report, 2005.

[37] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 1958.

[38] C. Xian, Y. Lu, and Z. Li. Energy-aware scheduling for real-time multiprocessor sustems with uncertain task exection time. In *Proceedings of the 44th Annual Design and Automation Conference*, pages 664–669, 2007.

[39] K. Burns, A. Barbalace, V. Legout, and B. Ravindran. KairosVM: Deterministic introspection for real-time virtual machine hierarchical scheduling. *VtRES*, 2014.

[40] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Proceedings of the 6th International conference on Advances in Information and Computer Security*, pages 96–112, 2011.

[41] B. D. Payne, M. D. P. De Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conf. (ACSAC)*, pages 385–397, 2007.

[42] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 586–600, 2012.

[43] The netwide assembler: Nasm. https://courses.engr.illinois.edu/.

[44] Russell Coker. Bonnie++ benchmark suite. http://www.coker.com.au/bonnie++/.