The Dissertation Committee for Wing-Chi Poon
certifies that this is the approved version of the following dissertation:

Real-Time Hierarchical Hypervisor

Committee:

____
Aloysius K. Mok, Supervisor

____
James Browne

____
Mike Dahlin

____
Greg Plaxton

____
Deji Chen

Real-Time Hierarchical Hypervisor

by

Wing-Chi Poon, BEng, MSc

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2010

Dedicated to God.

Acknowledgments

Glory to God in the highest, and thank God for His loving kindness. I ventured into the journey for a PhD before I was saved by His grace, but ultimately it is God who helped me to finish this race. His love endureth forever!

The road to a PhD has been a long one for me. Countless number of people have helped me in the process. Without their support and encouragement, I would not have completed the program. Just as the holy Bible puts it: “I returned, and saw under the sun, that the race is not to the swift, nor the battle to the strong, neither yet bread to the wise, nor yet riches to men of understanding, nor yet favor to men of skill; but time and chance happeneth to them all” (Ecclesiastes 9:11, KJV) and “I know that, whatsoever God doeth, it shall be forever: nothing can be put to it, nor anything taken from it: and God doeth it, that men should fear before Him.” (Ecclesiastes 3:14, KJV).

I would like to thank my parents for their up-bringing. They cared for my academic success from a very early age, and set a good example of continuous learning. They supported my quest to continue my higher education in United States.

My advisor Prof. Aloysius K. Mok provided excellent guidance throughout the process. He allowed great freedom to his students in our pursuit of intellectual excellence. I would like to thank my committee members Prof. James Browne, Prof. Mike Dahlin, Prof. Greg Plaxton and Dr. Deji Chen for their time and valuable input. I would like to thank my graduate advisors and graduate coordinators for their help with the graduation process. I would like to thank people from my research group Pak-Ho Chung, Xiang Feng, Yi Feng, Song Han, Zhenting He, Ruiqi Hu, Raymond Lau, Chan-Gun Lee, Huiya Liu, Yanbin Liu, Jianping Song, Peiyu Wang, Weirong Wang, Hung-Uk Woo, Jianliang Yi, Weijiang Yu and Xiuming Zhu for their fruitful discussions. The former works of Xiang Feng and Zhenting He were especially important towards building up this dissertation. I also had good discussion with other people in my department who are not in the same research group, including but not limited to Huiling Gong, Yan Li, Yi Li, Jia Liu, Yu Sun, Yuk-Wah Wong, Hua Xiang, Xincheng Zhang, Jiandan Zhen. I would also like to thank my roommates Junwei Huang and Rongfeng Shen during these years.

Part of the idea of the dissertation originated from my summer co-op at AMD, when I had access to expertise in the x86 architecture. Kevin J. McGrath, Ben Serebrin and Erich Boleyn had been especially helpful to me in my learning. The dissertation could not be completed without the full support of my acting manager and principal engineer Ole Agesen, my current manager Jerri-Ann Meyer, all my colleagues in the monitor group and monitor verification group especially for technical help from Ole Agesen, Jim Mattson and Ben Serebrin from the monitor group, and Haoqiang Zheng and Carl Waldspurger from the vmkernel group. Alex Gathwaite, Jeffrey Sheldon, Ross Knippel, Wei Xu and my officemate Michael Ho, all from the monitor group, have also greatly encouraged me and/or shared some of my workload while I worked on the dissertation. Some people from other teams also encouraged me in the process, including but not limited to Limin Wang, Tony Huang and Lisa Liu.

These years, I have enjoyed wonderful times with friends from church. They gave me much support, encouragement and inspiration from time to time. I would like to thank the guidance, patience and good example of the counsellors of Austin Chinese Campus Christian Fellowship (ACCCF) Pastor Philip Hsu, Sylvia Hay, Phil and Ruth Chang, Morgan and Yuning Lin, Shin-Tower and Lie-Ching Wang, Hsing-Pang and Mei-Fang Wang, Charles and Esther Tang; the counsellors of Silicon Valley Christian Assembly (SVCA) Rainbow Fellowship James and Julie Chen, Pastors Hung-Chieh and Nancy Yu; and the counsellors of Stanford Campus Evengelical Fellowship (CEF) Sister Gan, Dr. Juan. Various people have also influenced me in non-trivial ways.

The final months of my PhD began with Urbana 2009 mission conference in St. Louis MO. People who are not listed above and greatly influenced me during these months include Tin-Ying Hsu (ACCCF), Yi-Ju Hsiao (ACCCF), Kevin Tung (ACCCF), Hsin-Yao Chen (ACCCF), Pastor and Mrs. James Liu (SVCA), Pastor John Shou and Mrs. Esther Shou (SVCA), Han Hu (SVCA), Auntie Christine Yu (SVCA) and Wenwei Zheng (CEF). I would like to thank Yang Li and Dandan Wang, Lan Tang and Lina Zhang for their hospitality. Last but not least, I would also like to thank Daniel and Karen Evans for proof-reading my dissertation for grammatical correctness.

Real-Time Hierarchical Hypervisor

Publication No. ______________

Wing-Chi Poon, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Aloysius K. Mok

Both real-time virtualization and recursive virtualization are desirable properties of a virtual machine monitor (or hypervisor). Although the prospect for virtualization and even recursive virtualization has become better as the PC hardware becomes faster, the real-time systems community so far has not been able to reap much benefits. This is because no existing virtualization mechanism can properly support the stringent timing requirements needed by real-time systems. It is hard to do real-time virtualization, and it is even harder to do it recursively. In this dissertation, we propose a framework whereby the hypervisor is capable of running real-time guests and participating in recursive virtualization. Such a hypervisor is called a real-time hierarchical hypervisor.

We first look at virtualization of abstract resource types from the real-time systems perspective. Unlike the previous work on recursive real-time partitioning that assumes fully-preemptable resources, we concentrate on other and often more practical types of scheduling constraints, especially the non-preemptive and limited-preemptive ones. Then we consider the current x86 architecture and explore the problems that need to be addressed for real-time recursive virtualization. We drill down on the problem that affects timing properties the most, namely, the recursive forwarding and delivery of interrupts, exceptions and intercepts. We choose the x86 architecture because it is popular and readily available, but it is by no means the only architecture of choice for real-time recursive virtualization. We conclude the research with an architecture-independent discussion on future possibilities in real-time recursive virtualization.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The performance of computer systems has been growing exponentially according to Moore's Law. Most new servers have seen very low average utilization. Hence it is more economical to consolidate different servers into one physical machine to reap the benefits of its increased performance. However, running multiple servers in one physical system poses new security concerns. Another layer of system software called virtual machine monitor or hypervisor is used between the hardware and OS to provide isolation and fault containment. While the hypervisor solves part of the problem, it does not deliver the full power of the underlying hardware to its virtual machine partitions, hence real-time application may miss the deadline when running inside a hypervisor. We propose a real-time hypervisor that could be stacked up hierarchically to allow for arbitrarily complex security constraints to be implemented.

There are two challenges to this research, one is to make the hypervisor real-time capable, and the other is to make the hypervisor hierarchical capable. Sections 1.1 and 1.2 give an introduction and related known results to the real-time resource sharing problem. Chapters 2, 3 and 4 will cover our development to the real-time resource sharing issue. Section 1.3 gives introduction and related works on the hierarchical virtualization problem, whereas chapters 5 and chapter 6 will cover our development to the x86 hierarchical virtualization issue. Finally, we conclude in chapter 7 with a summary of new results and a discussion of future possibilities.

1.1 Abstract Resource Virtualization

Computing resources could be exclusively-owned, space-partitioned, time-shared or software-emulated. Examples of each type of resource are shown in table 1.1. We try to lay down a theoretical framework for the real-time aspects of resource virtualization by abstracting all computing resources.

Exclusively Owned	Space Partitioned	Time Shared	Software Emulated
Floppy Disk Drive	Memory Hard Disk Space	CPU Hard Disk Controller	Virtual Ethernet

Table 1.1: Abstract resources and their real-life examples

Exclusively-owned and space-partitioned resources do not pose any real-time problems, because they are always available to their owner OS. Problems arise with time-shared or software-emulated resources because they may lengthen the critical path of execution and potentially induce a deadline miss which does not exist if running on a dedicated resource.

Timing properties of software-emulated resources could be obtained from Worst-Case Execution-Time (WCET) analysis. Puschner and Alan [43] have a detailed review of the available literature in WCET analysis that could be utilized.

Virtualization of time-shared resources could be fully-preemptable (e.g. processor), limited-preemptable (e.g. packet-switched network) or non-preemptable (e.g. printer).

Time sharing fully preemptiable resources has been well studied. We will quote some selected definitions and key results from the Bounded-Delay Resource Partition (BDRP) model in the next section. This related work is highly useful in real-time recursive virtualization for all fully preemptable resources. We will present our development on non-preemptable resources in the chapter 2 and chapter 3, followed by limited-preemptable resources and other variants in chapter 4.

1.2 Time Sharing Fully Preemptable Resource

Mok et. al. [37] [36] have developed a complete framework for partitioning fully-preemptable resources so that each partition is capable of running real-time application. For the sake of easy reference, this section is a reproduction of their key definitions and results.

1.2.1 The Bounded-Delay Resource Partition (BDRP) Model

In summary, each partition has two configurable parameters $0 < \alpha \leq 1$ and $\Delta > 0$. α is the percentage of time we want to assign the resource to the partition, and Δ is the maximum additional units of time a partition has to wait before it receives its full allocation $\alpha(t_2 - t_1)$ of the resource for any duration of time $(t_2 - t_1)$. Based on this model, Feng et. al. has shown that any preemptive scheduling policy that works with a given task set under the reduced resource availability (α of total), with Δ subtracted from all the deadlines, are guaranteed to continue to work on this partition.

Definition 1. *A **Periodic Resource Partition** Π is a tuple (Γ, P), where Γ is an array of N time pairs $\{(S_1, E_1), (S_2, E_2), \ldots, (S_N, E_N)\}$ that satisfies $(0 \leq S_1 < E_1 < S_2 < E_2 < \ldots < S_N < E_N \leq P)$ for some $N \geq 1$, and P is the partition period. The physical resource is available to a task set executing on this partition only during time intervals $(S_i + j \times P, E_i + j \times P)$, where $1 \leq i \leq N, j \geq 0$.*

The above definition enumerates every time interval that is assigned to a partition and is a general representation of periodic partitioning schemes, including those that are generated dynamically by an on-line partition scheduler. We will build onto this definition and arrive at a more useful one in terms of real-time scheduling.

Definition 2. *A **Bounded Delay Resource Partition (BDRP)** Π is a tuple (α, Δ) where α is the percentage of total time the resource is available to the partition and Δ is called the **Partition Delay**, which is the largest time deviation of a partition during any time interval with regards to a uniform uninterrupted allocate of the resource.*

Note that this definition defines a set of partitions because there are many different partitions in the static partition model that may satisfy this requirement. It provides a starting point upon which other approaches of defining partitions will be considered in sections 4.3.2 and 4.4.

Thus the problem of scheduling a number of task sets on a given resource could be split into two steps:

1. Scheduling of the given resource into BDRPs
2. Scheduling of one task set on each of these BDRPs

We call the first one resource level scheduling, and the second one task level scheduling.

1.2.1.1 Task Level Scheduling

Theorem 3. *Given a task set* τ *and a BDRP* $\Pi = (\alpha, \lambda_n)$, *let* S_n *denote a valid schedule of* τ *on the normalized execution of* Π, S_p *the schedule of* τ *on Partition* Π *according to the same execution order and amount as* S_n. *Also let* λ *denote the largest amount of time such that any job on* S_n *is completed at least* λ *time before its deadline.* S_p *is a valid schedule if and only if* $\lambda \geq \lambda_n$.

In Theorem 3, λ defines the maximum allowable output jitter [8] for S_n. Therefore, informally, Theorem 3 could be written as: A task set is schedulable on a partition if the maximum allowable output jitter is no less than the partition delay.

Theorem 3 provides a practical way to schedule a task set on a partition. If we could find a schedule on the normalized execution and the smallest λ is no less than λ_n, we could use this schedule on the partition and be guaranteed that no deadline will be missed on the partition. The schedule on the normalized execution is the same as the traditional task schedule, for which there are many known techniques.

1.2.1.2 Recursive Resource Level Scheduling

In recursive virtualization, a partition group is scheduled within another partition. This is the more general problem of recursive resource level scheduling. When we schedule a partition group on a dedicated resource, we could consider the dedicated resource as a partition with $\alpha = 1$ and $\Delta = 0$.

Theorem 4. *A partition group* $\{\Pi_i(\alpha_i, \Delta_i)\}$ $(1 < i \leq n)$ *is schedulable on a partition* $\Pi(\alpha, \Delta)$ *if* $\sum_{i=1}^{n} \alpha_i \leq \alpha$ *and* $\Delta_i > \Delta$ *for all* i*,* $(1 < i \leq n)$*.*

Theorem 4 provides a method to determine the schedulability of scheduling partitions (a partition group) on another partition. However, it does not explain how to perform the actual scheduling since the infinite time slice scheme that is used in the proof is impractical. Therefore, the question remains how to schedule partitions using methods with finite context switch overhead.

Theorem 5. *Given a partition group* $\{\Pi_i(\alpha_i, \Delta_i)\}$ $(1 < i \leq n)$ *to be scheduled on a partition* $\Pi(\alpha, \Delta)$*. Let* S_n *denote a scheduler of scheduling* $\Pi_i'(\alpha_i/\alpha, \Delta_i - \Delta)$ $(1 < i \leq n)$ *on a dedicated resource with capacity of the same as the normalized execution of* Π*. Also let* S_p *denote the virtual time* S_n *scheduler of scheduling* Π_i *on* Π*. Then* S_p *is valid if* S_n *is valid.*

Theorem 5 justifies the observation that we may use essentially the same algorithms of scheduling partitions on dedicated resources for hierarchical partitioning by applying the virtual time scheduling scheme.

With the ability of scheduling a partition inside another partition, we could build a hierarchy of resource partitions for fully-preemptable resources.

Chapter 2 continues with an anomaly we discovered in non-preemptive scheduling. We call this the robustness problem. It is being rigorously defined and formally analyzed. Then in chapter 3, we propose necessary and sufficient conditions to ensure non-preemptive robustness. Finally, this new result is combined with the BDRP model in chapter 4, together with new lights on limited-preemptive scheduling and other types of scheduling constraints.

1.3 The x86 Recursive Virtualization

The x86 architecture is chosen because it is popular and readily available. The abstract theory in resource virtualization discussed in section 1.2, chapter 2, chapter 3 and chapter 4 finds its application in a real-life scenario.

Before the advent of the Intel VT-x [27] and AMD SVM technology [4], the x86 architecture is not known to be virtualizable [42] [44]. Known problems include, but are not limited to, ring aliasing, ring compression, address space compression, non-faulting access to privileged state, high overhead in interrupt virtualization, and lack of access to processor hidden states.

The advent of hardware-assisted virtualization addressed these issues [50] but some of them would re-appear if we proceed to recursive virtualization. For example, the hardware-assisted virtualization solved the ring aliasing problem by essentially creating an extra set of rings for the hypervisor, known as the root mode operation. When we do recursive virtualization, the guest hypervisor has to be run in non-root mode although it was designed to be run in root mode. Instead of ring aliasing, we may call this mode aliasing problem. Basically, the outer hypervisor has to trap all root mode operations and emulate them sacrificing performance.

Although the hardware-assisted virtualization is very handy to use, we do not take it for granted as the only possible form of virtualization in our discussion. In each aspect of recursive virtualization, we presented the case when hardware-assisted virtualization is not available, when it is available, and when the hardware could be extended to do better. Chapter 5 integrates the abstract theories of resource virtualization in the context of the x86 architecture. Then chapter 6 deals with one specific problem that affects real-time workloads the most, namely, the recursive forwarding and delivery of interrupts, exceptions and intercepts. These chapters dive into a lot of the x86 technicalities. Readers are referred to appendix 1 for a list of the acronyms used.

Finally, we give an architecture-independent view of how real-time recursive virtualization might be achieved in chapter 7. This is useful in real-time recursively virtualizing non-x86 architecture, or designing new architecture specifically for use with real-time recursive virtualization.

Chapter 2

Non-Preemptive Robustness - Definition and Characterization

Unlike preemptive scheduling policies, non-preemptive real-time scheduling policies can exhibit anomalies even for the single-processor case. In particular, a task set that is schedulable by a non-preemptive scheduler may become unschedulable when the utilization of the task set decreases relative to the CPU speed, e.g., when a faster CPU is used to run the same task set. In this chapter, we define the notion of *robustness* to capture the essence of the scheduling anomaly on real-time system performance. We shall show that it is difficult to test for robustness in general but it could still be characterized. In chapter 3, we shall derive necessary and sufficient conditions for guaranteeing non-preemptive robustness.

2.1 Introduction

One problem in engineering large complex software systems is the sensitivity of a design to changes in the requirements. If we view each step of the design process as a mapping from a requirement space to an (abstract) design space, the sensitivity problem may be viewed as a relation. Let us call it the *tracking relation* between a *difference metric* in the requirements space and a corresponding difference metric induced in the design space. These difference metrics are appropriately defined to measure the magnitude of change within their respective space.

Some properties of the tracking relation are obviously desirable. For example, it should preserve *locality*: differences confined to a locality in the requirements space should induce differences confined to a locality in the design space, and *scalability*: a small difference in the requirements space should induce a small difference in the design space.

Of course, how a difference metric is defined should reflect the aspect of requirements captured under consideration. For example, the difference metric meant to capture locality in the requirements space may reflect the number of functionalities / components that are affected by a change in the requirement, and the difference metric meant to capture the scalability in the requirements space may reflect the increase in system load in a requirements change. The idea of tracking relation is illustrated in figure 2.1.

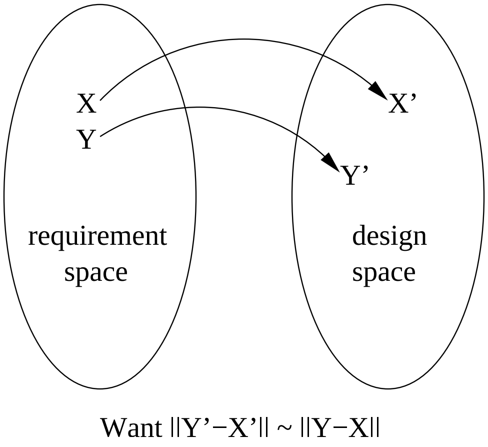


Figure 2.1: Tracking Relation

In the following, we shall illustrate the tracking relation concept by considering a specific aspect of real-time systems design, specifically, the relation between a change in the real-time performance requirements and the schedulability of the design solution. Intuitively, if we make the real-time performance requirement of an application less stringent, we should expect the design solution to require at most the same amount of computing resources. A mapping from requirement to design is *robust* if a less demanding requirement will not cause a performance failure in the design.

We believe that the notion of robustness will be important as long as CPU speed keeps on improving at a faster rate than memory I/O bandwidth (including L1, L2 and L3 caches). This is because the *worst-case* cost of preempting a task includes flushing caches, instruction pipelines and page tables all of which may incur I/O operations. Therefore unless all real-time tasks can be kept in fast memory all the time, the cost of preemption will be significant compared with task execution time.

2.1.1 Requirements Space - Periodic Task Set

The requirements space is the set of periodic tasks. A *periodic task* is characterized by a pair: $T_i = (C_i, P_i)$, where each service request of T_i requires C_i units of CPU time to satisfy and two successive requests must be separated by P_i time units. Suppose M is a set of n *periodic tasks* $\{(C_1, P_1), \ldots, (C_n, P_n)\}$ where C_i, P_i are respectively the computation time and the period for periodic task T_i. The first instance of all tasks in M arrive together at time 0.

2.1.2 Design - Priority Assignment

A design is a (fixed or dynamic) priority assignment to the tasks in the periodic task set. A Fixed-Priority (FP) scheduler or an Earliest-Deadline-First (EDF) scheduler is used to schedule tasks in M. An FP scheduler always selects for execution the task that has the highest priority. With Rate Monotonic Assignment (RMA) of priority, task T_i having a higher priority than task T_j implies $P_i \leq P_j$. We shall adopt the convention for FP scheduler that task T_i is assigned a higher priority than task T_j iff $i < j$. An EDF scheduler always selects for execution the task whose deadline is the nearest, hence the task priorities for EDF scheduler are dynamic, and change over time.

In this chapter, we talk about schedules for task set M that are produced by a Preemptive FP (PFP) scheduler, Non-Preemptive FP with RMA priority (NPFP/ RMA) scheduler, Preemptive EDF (PEDF) scheduler and Non-Preemptive EDF (NPEDF) scheduler. We call these schedules the PFP schedules, NPFP/RMA schedules, PEDF schedules and NPEDF schedules respectively.

	Preemptive	Non-Preemptive
EDF	PEDF	NPEDF
FP /w RMA	PFP PFP/RMA	NPFP NPFP/RMA

In general, FP/RMA and EDF produce different schedules (e.g. consider the task set $\{(2,3),(1,5),(1,8)\}$), but in order to save space, throughout the chapter, counter-examples are carefully chosen so that both EDF and FP/RMA have the same schedule.

A scheduler first computes an initial schedule or priority assignment based on the requirements specification of each task either statically (computed offline) or as part of the admission control process (computed online). Then a run-time dispatcher selects task instances for execution based completely or partially on the priority assignment information provided by the scheduler.

2.1.3 Requirement Change - Reduction in System Load

A (favorable) change in the requirements space is characterized by a reduction in system load, which is defined to be one or more of the following:

- Decrease in computation time of some task(s)
- Increase in period of some task(s)
- CPU upgrade, i.e. the use of a faster processor or CPU overclock

In particular, (3) is a special case of (1), where the computation time of all tasks are decreased by the same ratio. Also, deletion of a task is also a special case of (1) and (2), where the computation time becomes 0 and period becomes ∞. Note that we do not talk about scaling up resource nor increased workload in this dissertation (this is because in most of our target applications, e.g., mission critical embedded systems, an upgrade is often forced by the need for additional system functionality, and at best, you end up with the same if not greater system load).

We denote the task set after reduction in system load by M', and its constituent tasks $T_i' = (C_i', P_i')$, obeying the relations $C_i' \leq C_i$ and $P_i' \geq P_i$, with at least one inequality over all tasks being strictly less than (<).

There are two possibilities for reduction in system load, advertised and unadvertised. An advertised reduction in system load for a task means the actual values of (C_i', P_i') are made known to the run-time dispatcher prior to the arrival of the first instance of the task. Unadvertised reduction means the run-time dispatcher is never informed of any such changes. If the system load reduction is advertised, the scheduler could spend some time computing the optimal schedule; but the unadvertised ones are more common and do more harm. The unadvertised increase in period also represents a transitional model from periodic task to sporadic task.

2.1.4 Robustness

When there is a reduction in system load, we would normally expect the same design to work. In other words, a priority assignment that results in a task set being schedulable should preserve schedulability under reduction in system load. We say that a priority assignment is robust if schedulability is preserved in any reduction of system load. Robustness depends on the scheduling policy and type of timing constraints imposed on the system.

One of the hard problems in maintaining real-time systems requirements in mobile computing is to keep track of the impact of resource usage on the applications. Due to power consideration, CPU in mobile computing is often clocked at a range of frequencies, and adjusted at run time according to need. Often times, it is not sufficient to keep track of only the upper bounds on resource usage, since some requirements such as jitter are also sensitive to the lower bounds and the resource scheduling algorithm employed.

2.1.5 Preemptiveness vs Non-preemptiveness

Because of the NP-completeness of non-preemptive deadline scheduling [28], most extant work is about preemptive scheduling. However, non-preemptive scheduling is worth studying for a number of reasons, especially for resources that are inherently non-preemptable or when preemption cost is high.

Nowadays, processors are much faster, so jobs are much shorter. As processors are more pipelined, context switch overheads become relatively high. We want to resort to non-preemptive policies in an attempt to cut down this context switch overhead.

Also, in communication networks, synchronization and packet header processing overhead is relatively large. In order to deliver most of the available bandwidth to the end-user, batch processing of packets and messages is favored, thus limiting preemption.

Moreover, in open systems environment like mobile computing, we want non-interference among partitions, and jobs should not be preempted by other partitions. So it also necessitates the use of non-preemptive schedules.

2.1.6 Some Definitions and Notations

The p^{th} instance of a task T_i is denoted by T_i^p. Suppose r is the request for T_i^p that occurs at time t in a schedule s. Then the response time of r is defined to be $t' - t$ where t' is the time at which r is satisfied by the completion of T_i^p in s. Given a priority assignment, a task is schedulable if and only if all of its requests have response time no bigger than its period in the schedule. A task set is schedulable if every task in the set is schedulable.

A task T_i has no outstanding computation at time t if all the requests for T_i that arrived before t were satisfied by time t. A task set M does not have outstanding computation at time t if all the tasks inside M have no outstanding computation at time t. Time 0 is the time of the first request arrival, so by definition, it is a time of no outstanding computation.

2.1.7 Related Works

Scheduling anomalies have been known since [24]. Previous results pertain mostly to multiprocessor and list scheduling anomalies. The results reported in this dissertation pertain to real-time uniprocessor scheduling. The multiprocessor anomaly reported in previous work depends on processor assignment anomaly and do not apply to the uniprocessor case.

As far as we know, the definition of robustness in the sense of freedom from anomalies was first proposed by Mok in an invited lecture at both the NSF/ARO/CNR-Italy Workshop on Modelling Software System Structures and the 7th International Conference on Real-Time Computing Systems and Applications in 2000 (http://www.informatik.uni-trier.de/∼ley/db/conf/rtcsa/rtcsa2000.html). This and the next chapters contains results that answer some open problems proposed by Mok.

In [12], Buttazzo used Cyclical Asynchronous Buffers to avoid blocking on shared resources to avoid anomalies. However, the use of their approach is mainly limited to control applications, e.g. sensory acquisition task, because in their model, messages could be lost or read more than once. Their Rate Adaptation scheme works for processor overload, not reduced system load. In [10], Brandt et al showed that PEDF scheduling is anomaly-free. In [46], Sha eta al showed that RMA scheduling is anomaly-free under load reduction. Introduction to the general area includes [32], [47], [29] and [33]. Other related works include [28] and [52].

2.2 Robustness of Preemptive Schedulers

The schedulability problem for preemptive schedulers was first discussed in Liu & Layland [32]. We shall assume that time is discrete and all timing parameters are integers. Only slight modification is needed for the discussion below to apply to continuous time.

In each of the following sections, we are going to consider decrease in computation time and increase in period separately. In particular, we also highlight reduction in system load due to CPU upgrade, which is just a special case of decrease in computation time.

For the sake of clarity in proofs, we define an intermediate task set M'', whose constituent tasks $T_i'' = (C_i', P_i)$ have the same period P_i as the original task set T_i (hence the same priorities no matter if we use FP/RMA or EDF), yet with the decreased computation time C_i' as in T'.

A scheduling policy which is robust separately under both decrease in computation time and increase in period is also robust under any reduction in system load, because any reduction in system load could be represented as a combination of these two factors. Given that a task set M is schedulable under the policy concerned, the intermediate task set M'' is also schedulable because the policy is robust under decrease in computation time. Given M'' is schedulable, our target task set M' is also schedulable because the policy is robust under increase in period. Hence by transitivity, schedulability is preserved by this scheduling policy when system load is reduced from M to M'.

2.2.1 Preemptive Earliest-Deadline-First (PEDF)

By the Liu & Layland Model [32], a task set is schedulable by PEDF scheduling policy iff $\sum_{j=1}^{n} \frac{C_j}{P_j} \leq 1$

For decrease in computation time of some task C_k by δ $(1 \leq k \leq n, \delta > 0)$,

$$\sum_{j=1}^{n} \frac{C'_j}{P'_j} = \left(\sum_{j=1}^{n} \frac{C_j}{P_j} \right) - \frac{\delta}{P_k} < \sum_{j=1}^{n} \frac{C_j}{P_j} \leq 1$$

For increase in period of some task P_k by δ $(1 \leq k \leq n, \delta > 0)$,

$$\sum_{j=1}^{n} \frac{C'_j}{P'_j} = \left(\sum_{j=1}^{n} \frac{C_j}{P_j} \right) - \left(\frac{C_k}{P_k} - \frac{C_k}{P_k + \delta} \right) < \sum_{j=1}^{n} \frac{C_j}{P_j} \leq 1$$

(Note that this proof assumes that job priorities may change after period increase when the deadlines cross).

For CPU upgrade, where we reduce the computation time of all tasks by the same proportion α $(0 < \alpha < 1)$,

$$\sum_{j=1}^{n} \frac{C'_j}{P'_j} = \sum_{j=1}^{n} \frac{(1 - \alpha) \cdot C_j}{P_j} < \sum_{j=1}^{n} \frac{C_j}{P_j} \leq 1$$

Under any reduction in system load, the utilization factor

$$\sum_{j=1}^{n} \frac{C'_j}{P'_j} < \sum_{j=1}^{n} \frac{C_j}{P_j} \leq 1$$

remains smaller than 1. Hence the task set remains schedulable under PEDF scheduling policy. In order words, the PEDF scheduling policy is robust. (Note that this proof assumes that job priorities may change after period increase when the deadlines cross).

2.2.2 Preemptive Fixed-Priority (PFP)

By the Liu & Layland Model [32], a task set is schedulable by the PFP scheduling policy iff

$$\forall i(1 \leq i \leq n), \exists t_i \in (0, P_i], \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \leq t_i$$

Since the task set M is schedulable at first, we have a set of values $t_1, t_2, \ldots,$ t_n satisfying the above inequality. After reduction in system load, we want to find $t'_1, t'_2, \ldots, t'_n$ satisfying:

$$\forall i(1 \leq i \leq n), t'_i \in (0, P'_i] \wedge \sum_{j=1}^{i} C'_j \cdot \left\lceil \frac{t'_i}{P'_j} \right\rceil \leq t'_i$$

For decrease in computation time of some task C_k by δ $(1 \leq k \leq n, \delta > 0)$, then $\forall i$ ($k \leq i \leq n$, otherwise task T_k has no effect on the summation, which is only to i), we take $t'_i = t_i \in (0, P_i] = (0, P'_i]$

$$\begin{aligned} \sum_{j=1}^{i} C'_j \cdot \left\lceil \frac{t'_i}{P'_j} \right\rceil &= \left(\sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \right) - \delta \cdot \left\lceil \frac{t_i}{P_k} \right\rceil \\ &< \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \\ &\leq t_i \\ &= t'_i \end{aligned}$$

For increase in period of some task P_k by δ $(1 \leq k \leq n, \delta > 0)$, we first consider when job priorities do not change: $\forall i$ ($k \leq i \leq n$, otherwise task T_k has no effect on the summation, which is only to i), take $t'_i = t_i \in (0, P_i] \subset (0, P'_i]$, then

$$\begin{aligned} \sum_{j=1}^{i} C'_j \cdot \left\lceil \frac{t'_i}{P'_j} \right\rceil &= \left(\sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \right) - C_k \cdot \left(\left\lceil \frac{t_i}{P_k} \right\rceil - \left\lceil \frac{t_i}{P_k + \delta} \right\rceil \right) \\ &< \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \\ &\leq t_i \\ &= t'_i \end{aligned}$$

Hence in both cases, the inequality still holds.

If the period increase is advertised, we may change the RMA priorities accordingly. If it is unadvertised (which makes the periodic task begins to look like a sporadic task), the RMA priorities remain unchanged. We have shown that PFP is robust in general when priorities do not change, hence unadvertised PFP/RMA is robust with respect to increase in period.

Consider next advertised PFP/RMA. If the deadlines do cross when the periods increase, we introduce a series of intermediate reduced task sets $M^{(0)}, M^{(1)}, \ldots,$ $M^{(m+1)}$, where $M^{(0)} = M$, $M^{(m+1)} = M'$, and m is the number of swappings needed for bubble sort to sort the task sets from their original priority arrangement to the new one.

The intermediate reduced task sets are constructed as follows. Start with $M^{(i)}$, $M^{(i+1)}$ is obtained by picking the task $T_\tau^{(i)}$ with the lowest final priority (in M') whose period $P_\tau^{(i)} \neq P'_\tau$ and stretch it until either (1) $P_\tau^{(i+1)} = P'_\tau$ or (2) it hits the period of another task $T_\mu^{(i)}$ so that $P_\tau^{(i)} < P_\tau^{(i+1)} = P_\mu^{(i)} < P'_\tau$, whichever is earlier. Repeatedly pick a task this way and stretch its period until there are no more such tasks to pick, then the resulting task set is $M^{(i+1)}$.

It is easy to show that $M^{(m+1)} = M'$ based on an analogy with bubble sort. When system load is reduced from $M^{(i)}$ to $M^{(i+1)}$, task priorities are not changed, so our previous proof holds. Within $M^{(i+1)}$, swapping the priority of tasks $T_\tau^{(i+1)}$ and $T_\mu^{(i+1)}$ does not affect the schedulability of the task set because the periods of these two tasks are the same, i.e. $P_\tau^{(i+1)} = P_\mu^{(i+1)}$. Continuing this way, we see that M' remains schedulable for PFP/RMA even when the RMA priority changes after reduction in system load.

CPU upgrade or overclock is a special case of decrease in computation time. We provide the proof here for completeness. Here, the computation times of all tasks are reduced by the same ratio $\alpha < 1$. Take $t'_i = t_i$, then $\forall i (1 \leq i \leq n)$,

$$t'_i = t_i \in (0, P_i] = (0, P'_i]$$

and

$$\begin{aligned}\sum_{j=1}^{i} C'_j \cdot \left\lceil \frac{t'_i}{P'_j} \right\rceil &= \sum_{j=1}^{i} (1-\alpha) \cdot C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \\ &< \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \\ &\leq t_i \\ &= t'_i\end{aligned}$$

In any case, there exists t'_i satisfying the inequality after reduction in system load. Hence the task set remains schedulable under PFP scheduling policy. In other words, the PFP scheduling policy is robust.

2.3 Loss of Robustness in Non-preemptive Schedulers

Neither NPEDF nor NPFP/RMA scheduling policy is robust. In general, an anomaly may occur for any non-preemptive, eager scheduler which does not idle the CPU as long as there is a ready task.

- Decrease in Computation Time: Task set $\{T_1 = (3,5), T_2 = (2,10), T_3 = (4,20)\}$ is schedulable by an NPFP/RMA or an NPEDF scheduler. But it becomes unschedulable if we reduce the execution time of T_2 from 2 to 1. (figure 2.2)

- Increase in Period: Task set $\{T_1 = (1,4), T_2 = (3,8), T_3 = (6,16)\}$ is schedulable by an NPFP/RMA or an NPEDF scheduler. But it becomes unschedulable if we increase the period of T_1 from 4 to 5. (figure 2.3)

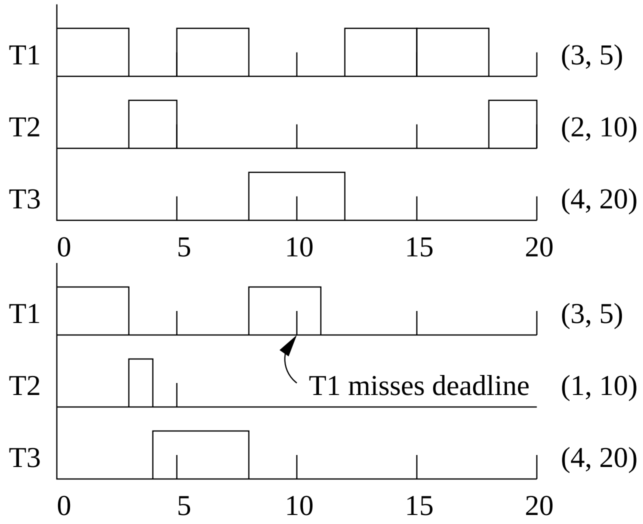


Figure 2.2: Loss of Non-Preemptive Robustness under Decrease in Computation Time (Top=Before; Bottom=After)

- CPU Upgrade or Overclock: Task set $\{T_1 = (30, 50), T_2 = (20, 100), T_3 = (40, 200)\}$ is schedulable by an NPFP/RMA or an NPEDF scheduler. But it becomes unschedulable if we reduce the execution times of all tasks by 10%. (figure 2.4)

Hence, the above counter-examples establish:

Theorem 6. *Neither the NPFP/RMA scheduler nor the NPEDF scheduler is robust with respect to any reduction in system load.*

2.4 How Bad is the Non-Preemptive Robustness Problem

The robustness problem occurs regardless of the CPU utilization factor, regardless of the number of different job sizes (length of job periods), and the problem

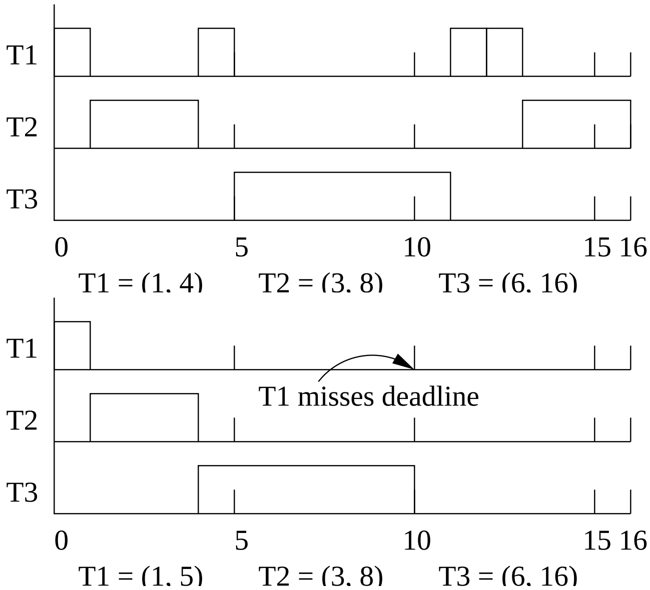


Figure 2.3: Loss of Non-Preemptive Robustness under Increase in Period (Top=Before; Bottom=After)

could not be solved by testing a finite number of workload reduction cases.

Theorem 7. *The loss of robustness for NPFP/RMA and NPEDF schedule exists even when the utilization factor of the task set tends to zero.*

Proof: A scenario suffices to demonstrate the fact here. For any given positive number ϵ, we construct a task set whose utilization factor $U < \epsilon$, yet neither NPFP/ RMA nor NPEDF schedule is robust on it.

Consider the task set $\{T_1 = (C, P), T_2 = T_3 = (2P - 2C, kP)\}$, where $k \geq 4$ is an integer. The NPFP/RMA and NPEDF schedules are not robust for this task set under any reduction in system laod. In figures 2.5 2.6 2.7 2.8, it causes T_1 to miss the deadline in the third period.

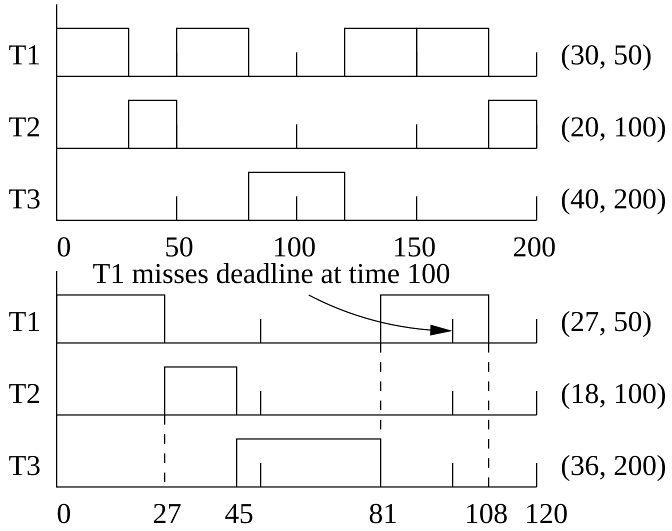


Figure 2.4: Loss of Non-Preemptive Robustness under CPU Upgrade or CPU Overclock (Top=Before; Bottom=After)

For this parameterized task set, take $k > \max(4, \frac{4}{\epsilon})$ and $P > \frac{(k-4)C}{\epsilon k-4}$, then we have utilization factor $U < \epsilon$. Hence when $\epsilon \to 0$, we have $k \to \infty$ and $P \to \infty$, utilization factor $U \to 0$, but task T_1 misses its deadline on the third period, so the anomaly still exists even when the utilization factor $U \to 0$. ■

Corollary 8. *Restricting job sizes (length of job periods) to a selected set won't avoid anomalies as long as there are more than one job size.*

Proof: The same example from theorem 7 shows that anomaly can occur when there are as few as only two job sizes. Restricting job sizes to harmonics won't help either, as illustrated in the scenario above. ■

Moreover, testing cannot solve the anomaly problem because no testing can detect all problems.

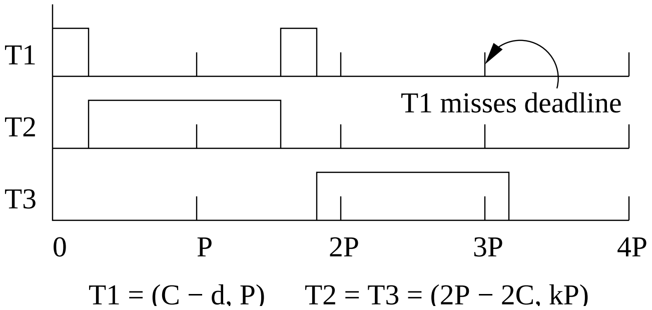


Figure 2.5: NPFP/RMA and NPEDF are not robust against decrease in a computation time of T_1

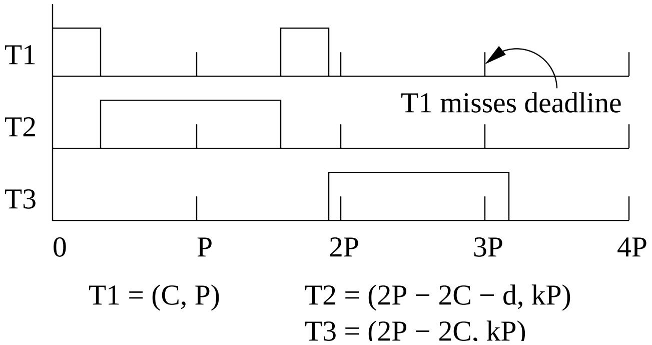


Figure 2.6: NPFP/RMA and NPEDF are not robust against decrease in a computation time of T_2

We will illustrate this with decrease in computation time. A good testing approach may go like this: For each task, try to decrease its computation time by a fixed factor δ each time and see if the anomaly occurs. If no anomaly occurs at all such testing points, we assume that the task set does not exhibit anomaly behavior for the scheduling policy concerned.

However, for any real-valued δ chosen, we can construct a task set such that it remains schedulable at all the testing points but shows anomalies in-between testing points:

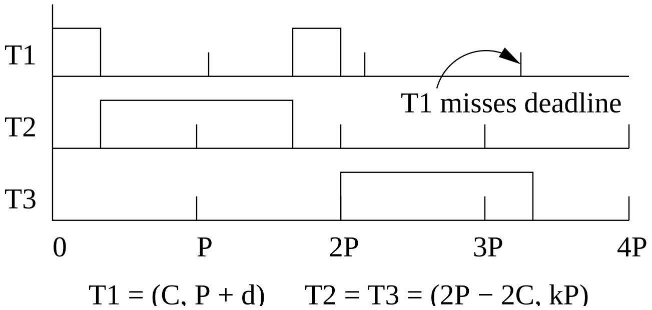


Figure 2.7: NPFP/RMA and NPEDF are not robust against an increase in period of T_1

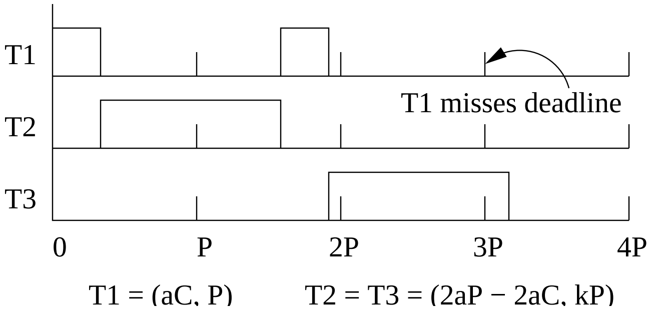


Figure 2.8: NPFP/RMA and NPEDF are not robust against CPU upgrade, $\alpha < 1.0$

T_1	$(2\delta, P)$
T_2	$(2P - 4\delta, kP)$
T_3	(δ, kP)
T_4	$(P - \delta, kP)$
$\vdots$	$\vdots$
T_n	$(P - \delta, kP)$

where $P \geq n\delta$ and is an even multiple of δ, integers $k \geq n$ and $n \geq 4$. This task set is schedulable by both NPFP/RMA and NPEDF scheduler (figure 2.9).

As shown in figures 2.10 to 2.13, whenever the computation time of T_2 is decreased by integral multiples of δ, the task set remains schedulable, but it may not be so when the computation time of T_2 is decreased by a non-integral multiple of

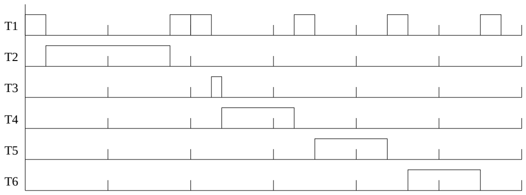


Figure 2.9: Original task set is schedulable

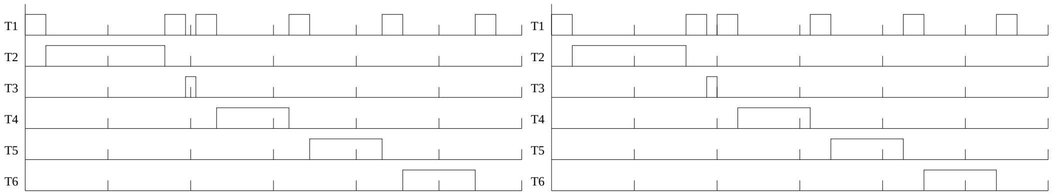


Figure 2.10: C_2' = left: $C_2 - \frac{1}{2}\delta$, right: $C_2 - \delta$

δ. In particular, the task set is not schedulable when the computation time of T_2 is decreased by $(m+\frac{1}{2})\delta$ and $(\frac{P}{\delta}-2+m+\frac{1}{2})\delta$, $\forall m : 1 \leq m \leq n-3$. Notice that even though an anomaly may not occur when the change in requirements specification tends to zero, it may still occur later.

Theorem 9. *There can be an infinite number of regions where an anomaly occurs.*

Proof: Consider the same parameterized task set we just constructed; for any positive integer m, we can choose $n = m + 3$ to achieve $2m$ number of anomalous regions. As $m \to \infty$, $n \to \infty$. ■

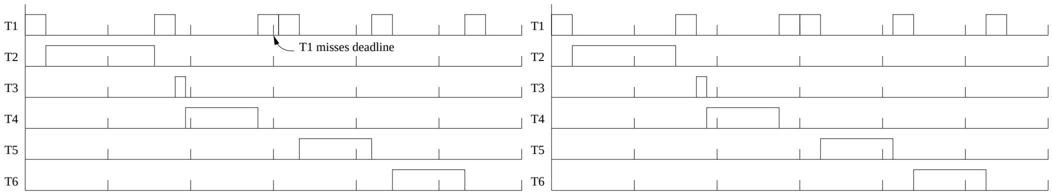


Figure 2.11: C_2' = left: $C_2 - 1\frac{1}{2}\delta$, right: $C_2 - 2\delta$

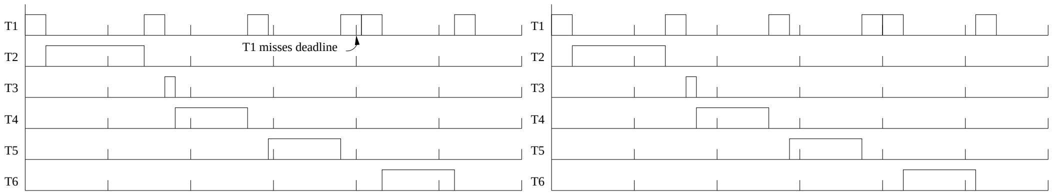


Figure 2.12: C_2' = left: $C_2 - 2\frac{1}{2}\delta$, right: $C_2 - 3\delta$

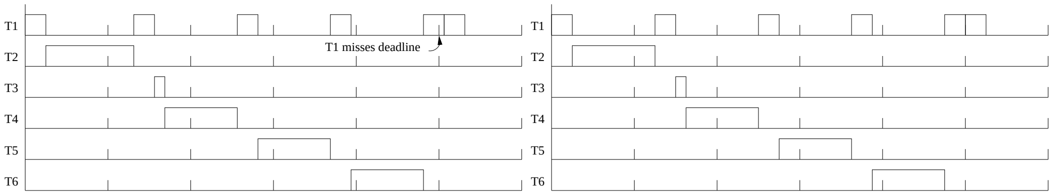


Figure 2.13: C_2' = left: $C_2 - 3\frac{1}{2}\delta$, right: $C_2 - 4\delta$

2.5 Properties of Non-preemptive Anomaly

In order to tackle the problem of non-preemptive scheduling robustness, we first gather a set of useful properties of non-preemptive scheduling.

Lemma 10. *For both NPFP/RMA and NPEDF, if the only kind of reduction in system load allowed is decrease in computation time, then the lowest priority task T_n will not miss its deadline.*

Proof: We will show that all instances of the task T_n would start no later than their respective start time in the original schedule.

Consider an arbitrary instance T_n^p of the task T_n. Let r be its request time, t be its start time and let t' be the latest time before t with no outstanding computation. Notice that t' remains a time of no outstanding computation under any decrease in computation time, because no task execution could cross t' under decrease in computation time. Tasks before t' could not cross because their computation time can only shrink but not expand to cross t'; tasks executed after t' cannot start earlier than their request time so they cannot cross t' too.

The number of requests for each task within the interval $[t', t)$ remains unchanged under reduction in system load, because the period is not changed. After reduction in system load, the total computation in the interval $[t', t)$ (excluding the task instance T_n^p) is decreased, hence the CPU must be idle during some time in this interval. Let the end of the last idle interval thus generated be t''. If $r \leq t'' < t$, then T_n^p starts earlier; otherwise, the total computation within the interval $[t'', t)$ is not greater than before, so T_n^p starts no later than before. Hence T_n^p will not miss its deadline. ■

For NPFP/RMA, observe that the start times of all instances of the lowest priority task T_n in a task set is the same no matter if we use preemptive or non-preemptive scheduling, it is natural to think that the critical instant of the lowest priority task T_n also occurs when the request for all tasks align at the same time. We will prove it formally in the following lemma. Note that this lemma is a study of the non-preemptive scheduling in general, and has nothing to do with reduction in system load.

Lemma 11. *For NPFP/RMA, the lowest priority task T_n has the longest response time when the request for all tasks align at the same time.*

Proof: Suppose the contrary is true that the lowest priority task T_n has the longest response time when the request for a certain task instance T_i^p is not aligned with that of T_n^q. Let r_p be the request time of T_i^p and r_q be that of T_n^q. Without loss of generality, let $r_p < r_q$ (in the case that the start time of T_i^1 is later than r_q, we let $p = 0$, and T_i^0 be a dummy instance which has no outstanding computation during its whole period). We consider the following cases:

If T_i^p still has outstanding computation at time r_q, or the CPU is busy between the completion time of T_i^p and r_q, then we move the request time of T_n^q to align with r_p. Doing so would not change the start time nor completion time of any task, yet the response time for T_n^q is increased, which is a contradiction.

Otherwise we consider the next instance T_i^{p+1}, and move its request time to align with r_q (and move all subsequent request times of T_i by the same amount too). This way, the number of requests of T_i during the period of T_n^q is not less than before, so the response time for T_n^q is at least as much as before.

So, the lowest priority task T_n has the longest response time when the request for all tasks align at the same time. ■

If a task meets its deadline when it has the longest response time, then it meets all deadlines. This is traditionally called the critical instant test.

Corollary 12. *For NPFP/RMA, when the first request for all tasks arrive together, if the only kind of reduction in system load allowed is increase in period, then the lowest priority task T_n will not miss its deadline.*

Proof: Special case 1: T_n is the only task whose period is increased, the completion time for the first instance of T_n remains unchanged but its deadline is extended, so T_n would still pass its critical instant test. Special case 2: The period of

T_n remains unchanged, then during the first period of T_n, the number of requests for higher priority tasks decreases (because some or all of their periods have increased), so the response time for T_n^1 is less than or equal to before, hence T_n still meets the critical instant test.

General case: The period of T_n and one or more other tasks are increased, we consider an intermediate task set in which only the period of T_n is increased. By transitivity of the above two cases, T_n still passes its critical instant test. Hence in any case, T_n will not miss its deadline. ■

Notice that the proof is true for T_n even if the period of some other task T_x increases to beyond that of T_n, i.e. $P'_x > P'_n$. Hence corollary 12 is valid for both advertised and unadvertised period increase, or in other words, valid no matter if the RMA priorities are re-adjusted accordingly or not after period increase.

Theorem 13. *For NPFP/RMA, when the first request for all tasks arrive together, after reduction in system load, the lowest priority task T_n in M never misses the deadline.*

Proof: After reduction in system load, the new task set $M' = \{(C'_1, P'_1), (C'_2, P'_2), \ldots (C'_n, P'_n)\}$. We consider an intermediate task set $M'' = \{(C'_1, P_1), (C'_2, P_2), \ldots (C'_n, P_n)\}$. The lowest priority task T_n in M remains to be the lowest priority task as T''_n in M'' because the periods are not changed. By lemma 10, the lowest priority task T''_n in M'' does not miss its deadlines. By corollary 12, task T''_n does not miss its deadlines in M' too even though it may or may not remain to be the lowest priority task in M'. Hence by transitivity, the lowest priority task T_n in M never misses its deadline under reduction in system load. ■

Notice that the lowest priority task is not completely detached from the rest of the task set. Removing it may leave the task set unscehdulable, e.g. deleting T_3 from the task set: $\{T_1 = (2, 8),\ T_2 = (9, 14),\ T_3 = (3, 28)\}$ under NPFP/RMA or NPEDF. Similarly, decreasing the load (or even removal) of the highest priority task may also leave the task set unschedulable, e.g. removing T_1 or reducing it to $(1, 20)$ from the task set: $\{T_1 = (2, 20), T_2 = (6, 20), T_3 = (24, 80), T_4 = (21, 80)\}$ under NPFP/RMA or NPEDF.

Theorem 14. *For NPFP/RMA, when the first request for all tasks arrive together, then after any reduction in system load, let the request time of task T'_i be r' and its time of start of execution be s': if all the tasks that get executed between r' and s' have priorities higher than i, then task $T'p_i$ does not miss its deadline.*

Proof: Since all the tasks that get executed between r' and s' have priorities higher than i, then we may consider task T'_i as the task of lowest priority during this interval of time (by deleting all lower priority tasks, and for the remaining tasks, counting only requests whose deadlines are after r'). By lemma 11, task T'_i has the longest response time when the request of all higher priority tasks arrive together at r', so we only need to consider the first request of task T'_i. By theorem 13, task T'_i would not miss its deadline under any reduction in system load. ■

Theorem 15. *For NPEDF, after any reduction in system load, let the request time of task T'_i be r' and its time of start of execution be s': if all the tasks that get* *executed between r' and s' have deadlines earlier than that of T_i' (i.e. with higher dynamic priority), then task T_i' does not miss its deadline.*

Proof: It is known that any task set that is schedulable by NPEDF is also schedulable by PEDF. We have also proved that PEDF is robust against reduction in system load. A property of PEDF is that between r' and s', only tasks whose deadlines are earlier than that of T_i' get executed. By rearranging the PEDF execution order of task instances between r' and s' after reduction in system load, we get the corresponding schedule for NPEDF (which may or may not cause deadline miss for those higher priority tasks, but we are only concerned with task T_i'). This task instance swapping keeps the start time s' of task T_i'. Hence T_i' would not miss its deadline. ■

Theorem 16. *For both NPFP/RMA and NPEDF, assume the first request of all tasks arrive together. If a certain instance of victim task $T_v'^p$ whose request time is r' misses its deadline after reduction in system load, then there exists an instance of a culprit task $T_c'^q$ satisfying these properties:*

- *The culprit task T_c is of lower priority than the victim task T_v (For NPFP/ RMA, static priority means $v < c$; for NPEDF, dynamic priority means the deadline of T_c is later than that of T_v).*
- *The culprit task $T_c'^q$ is executing at time r'.*

Proof: Suppose the contrary is true that all the task instances that are executed between r' and the deadline miss are of (static or dynamic) priorities higher than

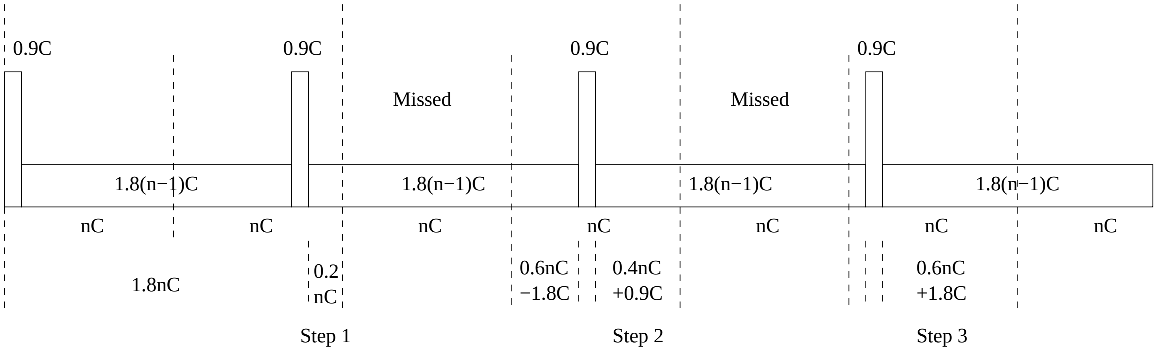


Figure 2.14: First seven periods of T_1 in the example task set

that of the victim task T_v, then by theorems 14 and 15, the victim task instance $T_v'^p$ would not miss its deadline, which is a contradiction. Hence an instance of the culprit task T_c of lower (static or dynamic) priority must be executing between r' and the deadline miss. Moreover, the culprit task must have started executing before r', otherwise by the time it starts execution, there is no outstanding computation for all higher priority tasks including the victim task T_v, and the victim task would not have missed the deadline. Hence the culprit task instance $T_c'^q$ must be executing at time r'. ■

2.6 Miss Ratio

We will look at a CPU upgrade scenario where the miss ratio for NPFP/ RMA or NPEDF scheduler can be as high as $\frac{1}{2}$. Miss ratio is defined for a task as the number of task instances whose deadlines are missed over the total number of task instances initiated in that interval of time.

We assume that if a task instance is not yet started when it misses its deadline, it will never be started. If an executing task instance misses its deadline, it does not matter whether it is killed right away or allowed to go to completion. When the run-time dispatcher picks a task instance for execution, the task is started even if the remaining time to its deadline is insufficient for its advertised computation time.

Consider the following set of tasks:

	Original	After CPU upgrade
T_1	(C, nC)	$((1-\alpha)C, nC)$
T_2	$(2(n-1)C, 2knC)$	$((1-\alpha)2(n-1)C, 2knC)$
$\vdots$	$\vdots$	$\vdots$
T_{k+1}	$(2(n-1)C, 2knC)$	$((1-\alpha)2(n-1)C, 2knC)$

where C is the computation time of T_1, n is the ratio of period to computation time of T_1, k is the number of ‘long’ tasks, α is a measure of increase in CPU performance. Figure 2.14 shows the first 7 periods of T_1. Here α is taken to be 0.1, i.e. the CPU becomes faster by 10%. For ease of explanation, we group every two periods into one step.

Consider the time left at the end of step i,

$$\textit{time left } = 2i\alpha nC + (i-1)(1-\alpha)C$$

Let $k = i$, the whole pattern repeats again when

$$\textit{time left } = (1-\alpha)2(n-1)C - nC$$

So we equate these two terms, and solve for i:

$$i = \frac{n(1-2\alpha) - (1-\alpha)}{2\alpha n + (1-\alpha)}$$

Taking the limit $n \to \infty$, $i \to \frac{1-2\alpha}{2\alpha}$. At the end of step i, the *total number of requests for* $T_1 = 2i$, the *total number of requests missed* $= i - 1$, so the *miss ratio* $= \frac{i-1}{2i}$. As $\alpha \to 0$, $i \to \infty$, so the *miss ratio* $\to \frac{1}{2}$.

Notice that the worst case miss ratio happens when $\alpha \to 0$, which may well be the case of improperly handled clock jitters!

Theorem 17. *A tight bound for the worst case miss ratio of NPEDF is* $\frac{1}{2}$.

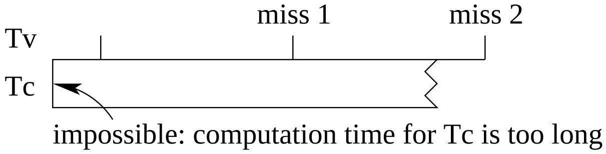
Proof: We have already seen a scenario where the worst case miss ratio for the NPEDF can be as high as $\frac{1}{2}$. So we only need to show that this is also the upper bound and the tightness follows. The intuitive observation is that there cannot be two consecutive misses for any task.

Suppose the contrary is true that some tasks miss deadline consecutively. Let T_v be the *first* task that misses its deadline consecutively in the execution.

By theorem 16, an instance of lower priority culprit task $T'^{p}_{c_1}$ must be executing at the time the first missed request of the victim task T'_v arrives ($c_1 > v$), and an instance of lower priority culprit task $T'^{q}_{c_2}$ must be executing at the time the second of the consecutive missed requests of the victim task T'_v arrives ($c_2 > v$).

These two task instances must be the same instance of the same task, i.e. $c_1 = c_2$ and $p = q$, otherwise by the time $T'^{q}_{c_2}$ starts, there are no outstanding computation for T'_v so it won't miss the first deadline. Notice that by our choice of treating deadline miss, T'_v would be started instead of $T'^{q}_{c_2}$ even if the remaining time to deadline is insufficient for its required computation time C_v.

Hence a certain instance of the lower priority culprit task $T_c'^{q}$ must be executing at the times the two consecutive missed requests of the victim task T_v' arrive ($c > v$).



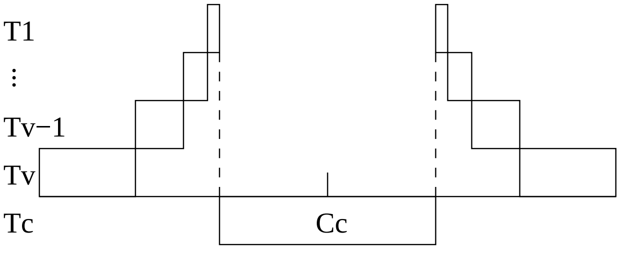
impossible: computation time for Tc is too long

But if T_c' starts before the arrival time of the first missed request, and ends after the arrival time of the second missed request, its computation time would be longer than possible. First, we have $C_c' > P_v'$, and for each task T_i, where $i < c$, $2P_i - 2C_i \geq C_c$, hence $2P_i' \geq 2P_i > 2P_i - 2C_i \geq C_c \geq C_c' > P_v'$, so that $\frac{P_v'}{P_i'} < 2$, i.e. $\left\lceil \frac{P_v'}{P_i'} \right\rceil \leq 2$. We will need this result at the last part of this proof.

Since the task set was originally schedulable before reduction in system load, by construction, the longest possible C_c we can have is:

$$C_c \leq 2P_v - \sum_{i=1}^{v} C_i \cdot \left\lfloor \frac{2P_v}{P_i} \right\rfloor$$

Notice that $\left\lfloor \frac{2P_v}{P_i} \right\rfloor \geq 2$ because $\forall i, 1 \leq i \leq v, P_i \leq P_v$. Hence when maximizing C_c, at least two instances of each higher priority task get executed during any two consecutive periods of P_v.

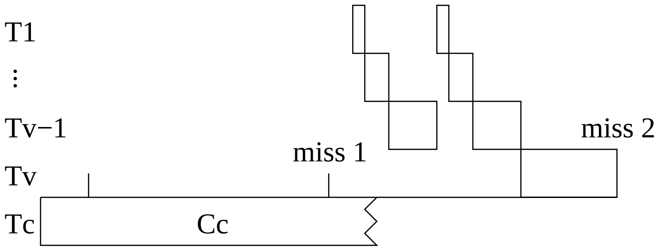


Hence we have

$$C'_c \leq C_c \leq 2P_v - \sum_{i=1}^{v} C_i \cdot \left\lfloor \frac{2P_v}{P_i} \right\rfloor \leq 2P_v - 2\sum_{i=1}^{v} C_i \leq 2P'_v - 2\sum_{i=1}^{v} C'_i$$

However, in order for T'_c to cause two consecutive misses, its computation time C'_c has to satisfy

$$C'_c > 2P'_v - \sum_{i=1}^{v} C'_i \cdot \left\lceil \frac{P'_v}{P'_i} \right\rceil \geq 2P'_v - 2\sum_{i=1}^{v} C'_i \geq 2P_v - 2\sum_{i=1}^{v} C_i$$



which is a contradiction. In the equation, $\left\lceil \frac{P'_v}{P'_i} \right\rceil$ counts the maximum number of deadlines of task T'_i that falls inside the second period of task T'_v in the graph.

So, there cannot be two consecutive misses for any task. Hence the miss ratio is at most $\frac{1}{2}$, and this is a tight bound for NPEDF scheduling policy. ■

Corollary 18. *With NPEDF and our choice of treating deadline miss as in the previous section, progress is guaranteed to all tasks in the task set.*

Proof: Since theorem 17 tells us that there could not be two consecutive misses for any task, there must be progress in any two adjacent periods for any task. ■

2.7 Conclusion

Non-preemptive scheduling is known to be NP-hard. Nevertheless, non-preemptable resources account for most of the I/O resources of a computing system. Therefore, a properly virtualizing non-preemptable resource is very important for any hypervisor design. However, the problem of time sharing non-preemptable resources to achieve real-time properties is not fully resolved. Notably, non-preemptive scheduling is subject to certain anomalies whereby a schedulable system may become unschedulable when the total system load is reduced (as opposed to increased).

We say that a scheduling policy is robust for a task set if it preserves the schedulability of the task set under reduction in system load. Both the PFP and PEDF schedulers are robust, while none of the NPFP/RMA nor the NPEDF scheduler is robust, even for the single processor case. In general, schedulability conditions do not necessarily guarantee robustness. Furthermore, a scheduling anomaly may happen even when the utilization factor tends to zero, when there are as few as only two job sizes in the task set, and when all the job sizes are harmonics. Testing (or simulation) cannot solve the problem, because for any testing approach, there exists a task set whose anomaly cannot be detected. There can be an infinite number of anomalous regions too. We then study the effect of the scheduling anomaly on deadline misses. We proved that a tight bound for miss ratio is $\frac{1}{2}$ under reduction in system load for the NPEDF scheduler.

Chapter 3

Solutions to the Non-Preemptive Robustness Problem

A real-time scheduler is robust (sustainable) for a certain task set if its schedulability is preserved under lighter system load by the scheduler. The first part of this chapter shows that non-preemptive robustness of a zero-concrete periodic task set against increase in period is sufficient to guarantee non-preemptive robustness for all variants of the task set. This proof includes the corresponding concrete or non-concrete periodic and sporadic task sets against any kind of reduction in system load.

Based on this result, the second part of this chapter gives the necessary and sufficient conditions for robustness for both Non-Preemptive fixed-priority (NPFP) and Non-Preemptive earliest-deadline first (NPEDF) schedulers under both discrete time and dense time assumption separately. We also look at some special cases where simplication could be made.

3.1 Introduction

A major advance in real-time scheduling theory started by the seminal work of Liu and Layland [32] has been based mostly on preemptive schedulers. In practice, however, non-preemptive schedulers have been widely used in the avionics industry and other safety-critical applications for various reasons such as ease of testing for timing compliance and the minimization of context switching overhead.

In recent years, a new concern increasingly makes non-preemptive schedulers attractive for use in real-time applications, namely the difficulty in obtaining accurate execution time bounds in modern processor architectures that exploit heavy pipelining and caching techniques. The result is that performance analysis has to be overly conservative in the choice of execution time numbers in the case where compliance to hard real-time constraints must be demonstrated to the certification authorities, or that a delicate tradeoff must be made between using more optimistic execution time numbers and accepting the possibility of missing some deadlines in the case of softer real-time constraints. This choice is increasingly difficult to make as the variance in execution time can be a factor of 10 or higher in modern computer architectures, especially where heavy context switching is incurred by preemptive schedulers. The end result is the loss of predictability in real-time performance.

This situation is ameliorated by the use of non-preemptive schedulers for which task interrupts are not allowed and the effect of caching on timing predictability is easier to analyze and control. However, there is a price to pay for the use of non-preemptive schedulers including weaker schedulability bounds and, less obviously, the loss of robustness against variation in resource usage parameters [38], a phenomenon that has also been referred to as sustainability [7]. Whereas the degradation in schedulability bounds could be somewhat counter-balanced by a decrease in the length of the blocking factor, the loss of robustness is a direct consequence of non-preemption and complicates the testing and verifiability of real-time systems.

In this chapter, we present some general results in non-preemptive real-time scheduling with respect to the issue of robustness. Chapter 2 has shown that non-preemptive schedulable task set may become unschedulable under reduction in system load. A scheduler is *robust* for a certain task set if it preserves schedulability under reduction in system load. We analyze the necessary and sufficient conditions for robustness of the non-preemptive earliest deadline first (NPEDF) and non-preemptive fixed priority (NPFP) schedulers for both periodic and sporadic task models, both discrete and dense time models, and both concrete and non-concrete task sets.

A preference for non-preemptive scheduling disciplines calls for formal investigation of the notion of robustness as we shall pursue in this chapter. In fact, many avionics applications already adopt at least limited non-preemption because of data locking issues. It is commonly held that by introducing a blocking term which is equal to the longest task, well known results such as those in RMA analysis can also be applied to non-preemptive priority schedulers, but as far as we know, this has never been justified in open literature. Our results in the last part of this chapter provide a formal justification for this "folk knowledge".

3.1.1 Task Model

A *periodic / sporadic task* is characterized by a pair: $T_i = (C_i, P_i)$, where each request for service of T_i requires C_i units of time and two successive requests are separated by *exactly* (periodic) / *at least* (sporadic) P_i time units. Each request results in a *task instance*. A *periodic / sporadic task set* is a set of n periodic / sporadic tasks $(C_1, P_1), \ldots, (C_n, P_n)$ where C_i, P_i are the computation time and period / minimum separation respectively for task T_i.

If a task set has a fixed release time for the first instances of all its tasks, it is called a *concrete task set*, otherwise it is a *non-concrete task set*. A non-concrete task set is a set of concrete task sets over all possible release times, i.e., arbitrary release times for its first task instances. A *zero-concrete task set* is a special case of concrete task set where the release times of all first task instances are at precisely time 0. (See figure 3.1).

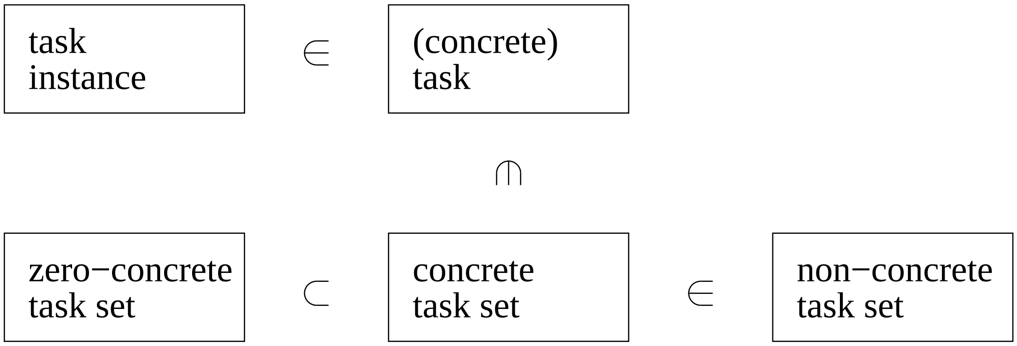


Figure 3.1: Set relationship in our task model

The deadline for each task instance is assumed to be at the end of their respective period or minimum separation. Time could be *discrete* with a minimum quantum, or *dense* by taking on values over the set of real numbers.

We use the following abbreviation in this chapter: CP (concrete periodic), CS (concrete sporadic), NCP (non-concrete periodic), NCS (non-concrete sporadic), ZCP (zero-concrete periodic) and ZCS (zero-concrete sporadic).

We only consider non-preemptive eager schedulers. By non-preemptive, we mean that a task instance must be allowed to go to completion uninterrupted once it is selected for execution. By *eager scheduler* (also called *non-idling* or *greedy scheduler* in some literature), we mean that the CPU must not be let idle whenever there is any ready (released) task instances. In particular, we explore the properties of a few eager schedulers like the Non-Preemptive Fixed-Priority (NPFP) scheduler and the Non-Preemptive Earliest-Deadline-First (NPEDF) scheduler in this chapter.

An NPFP scheduler always selects for execution the task that has the highest priority. With Rate Monotonic Assignment (RMA) of priority, task T_i has a higher priority than task T_j iff $P_i \leq P_j$. For simplicity, we shall adopt the convention for NPFP scheduler that task T_i is assigned a higher priority than task T_j iff $i < j$. An NPEDF scheduler always selects for execution the task whose deadline is the nearest, hence the priority assignment for NPEDF scheduler is dynamic, and changes over time.

3.1.2 Reduction in System Load Revisited

The definition of robustness [39] is closely coupled with the concept of reduction in system load. There are two basic components of reduction in system load:

- *decrease in computation time* ($\downarrow C$)
- *increase in period* (or *minimum separation*) ($\uparrow P$)

Many common forms are special cases or a combination of the above two components. For example, *deletion of task* is a special case where the computation time is decreased to 0 and the period is increased to ∞; *CPU upgrade or overclock* is another special case where the computation times of all tasks are reduced by the same factor α.

Blocking factor[1] could also be thought of as a special case of $\downarrow C$ (by treating the blocking factor as a task with the highest priority in the system). When a scheduler is robust for a certain task set with blocking factor, it remains schedulable when the blocking factor is reduced or removed.

A reduction in system load could be *advertised* (run-time dispatcher knows the exact amount of (decreased) computation time and (increased) period of all task instances by the time it dispatches them), or *unadvertised* (run-time dispatcher has no idea if a task instance would finish early once it is started, or if the next task instance would arrive late once the previous one finishes). If the system load reduction is advertised, the scheduler could spend some time computing the optimal schedule; however, the unadvertised ones are more common and does more harm.

There are two kinds of $\uparrow P$. (1) *Restrictive* $\uparrow P$ (usually happens when advertised) means that all task instances have the same (increased) period thus keeping its periodicity. (2) *General* $\uparrow P$ (usually happens when unadvertised) may leave the different instances of the same task having their periods increased to a different ex-

[1] *Blocking factor* is a term that is used in non-preemptive scheduling analysis to capture the maximum amount of time a released higher priority task has to wait before it could start execution due to another lower priority task that is executing and that cannot be preempted.

tent. Restrictive $\uparrow P$ is a special case of the general $\uparrow P$. The general $\uparrow P$ resembles the definition of a *sporadic task*, hence some known schedulability analysis results of sporadic task set could be borrowed. In this regard, sporadic task sets could also be considered as manifestations of the general $\uparrow P$.

3.1.3 Robustness Revisited

A task set is *robust* under a certain scheduler if its schedulability is preserved under reduction in system load. A scheduler is *robust* if it is robust for all task sets. Chapter 2 has shown that in non-preemptive scheduling, many schedulable task sets become unschedulable under reduction in system load, i.e. they are not robust.

We call it *general robustness* if schedulability is preserved under any reduction in system load. We also talk about special robustness, e.g. schedulability is preserved specifically under $\uparrow P$. If a task set under a certain scheduler is robust against both $\uparrow P$ and $\downarrow C$, then by transitivity, it is also robust against any reduction in system load, therefore qualifies for general robustness.

3.2 Related Work

Different words have been used in the literature to mean the same thing. For example, the concept of *robustness* was coined as *stability* in Deogun et. al [17] and *sustainability* in Baruah et. al. [7] [11]; the concept of *culprit task* was coined as *usurper task* in Deogun et. al. [17].

However, some of the previous work assumed task models different than ours. While we adopted the periodic and sporadic task models [32], Deogun et.

al. assumed exactly one task instance for each task, with fixed delay between the finish time of the previous task instance and the release time of the next instance. We believe the periodic and sporadic task models are so widely deployed that their non-preemptive robustness is worth studying. Also, they are not assuming eager scheduler, whereas using an eager scheduler is important in our model because under non-advertised reduction in system load, no scheduler has any fore-knowledge to insert idle time in anticipation of unarrived task instances.

No previous work addresses the necessary and sufficient conditions for non-preemptive robustness over the full spectrum of task models as does this chapter.

Burns and Baruah [7] [11] considered sustainability mainly for preemptive schedulers. Their non-preemptive test requires determination of a minimum R_i that could be obtained by exhaustive search in discrete time but impossible with dense time. Even with discrete time, our test has better running time.

Jeffay et. al. [28] provided a necessary and sufficient condition for schedulability of CS, NCS and NCP tasks. Their results are very useful and inspiring, but they are limited to the NPEDF scheduler under discrete time assumption. In this chapter, we explore both the NPEDF and NPFP schedulers under both discrete and dense time assumptions.

3.3 Properties of Non-Preemptive Robustness

Chapter 2 gave some useful properties of non-preemptive robustness. Let us explore more of these properties here. They lead to the conclusion that it is sufficient to check for non-preemptive robustness by looking at just the ZCP task set against $\uparrow P$.

This section explores non-preemptive robustness in general without reference to any particular scheduler. We will add some famous non-preemptive schedulers to the picture in section 3.4.

3.3.1 Concrete Robustness and Non-Concrete Schedulability

Jeffay et. al. [28] has shown that the non-preemptive schedulability conditions of NCP task set is exactly the same as that of the NCS task set. In the next theorem, we are going to show that the non-preemptive schedulability conditions in this case are equivalent to its non-preemptive robustness conditions.

Theorem 19. *Non-preemptive robustness of a CP/CS task set against $\uparrow P$ is equivalent to non-preemptive schedulability of the corresponding NCS task set.*

Proof: Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a NCS task set with arbitrary release times r'_i, and where $T_i = (C_i, P_i)$, for $1 \leq i \leq n$. Let $T = \{(T_1, r_1), (T_2, r_2), \ldots, (T_n, r_n)\}$ be the corresponding CP/CS task set where r_i is the release time for the first instance of T_i.

($\rightarrow$) Given that T is non-preemptively robust against $\uparrow P_i$, we construct a task set T' of increased P'_i as follows. Let $t = \max_{1 \leq i \leq n}(r_i + P_i)$. (See figure 3.2).

For any values of release times r'_i (from task set τ), where $1 \leq i \leq n$, let the release time of the second instances of the i-th task in T' be $t + r'_i$. Afterwards, all task requests in T'_i assume the same pattern as in task set τ. (See figure 3.3).

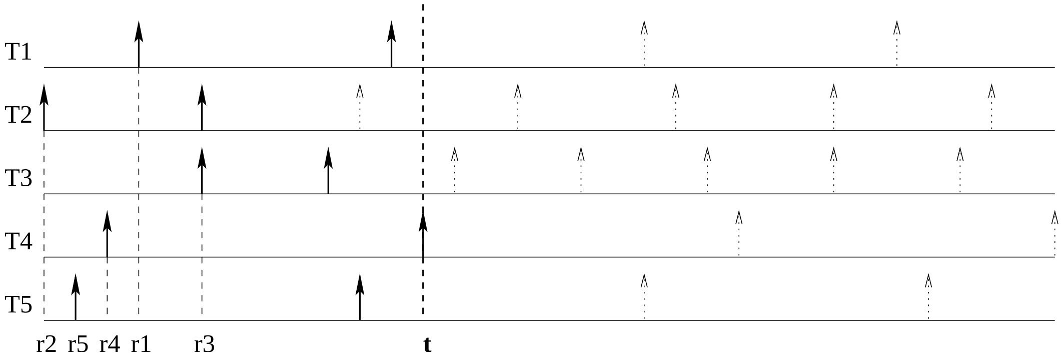


Figure 3.2: Determination of t from task set T

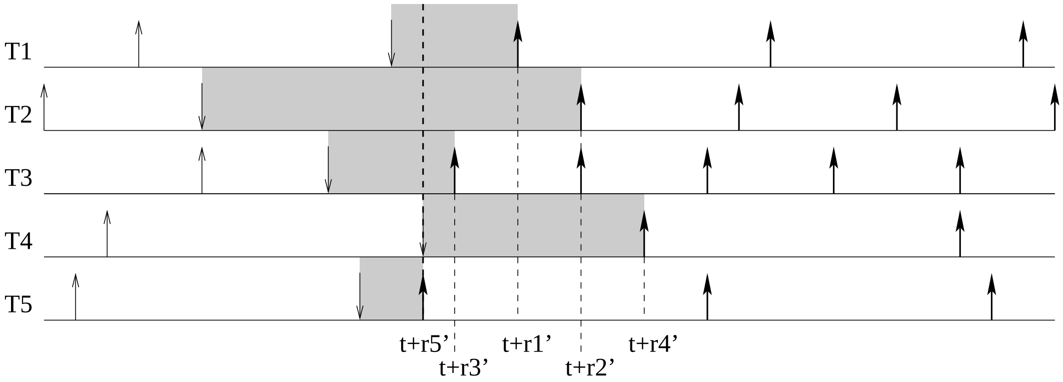


Figure 3.3: Construction of task set T'

Since T is non-preemptively robust, we conclude that T' is non-preemptively schedulable. By our construction, there is no outstanding computation at time t, so the task set (whose release times are r_i') formed by chopping off the first t units of time from task set T_i' is also non-preemptively schedulable. (See figure 3.4).

Since the derivation is valid for all r_i', the NCS task set τ is non-preemptively schedulable. Therefore, non-preemptive robustness of CP/CS task set against $\uparrow P$ implies non-preemptive schedulability of the corresponding NCS task set.

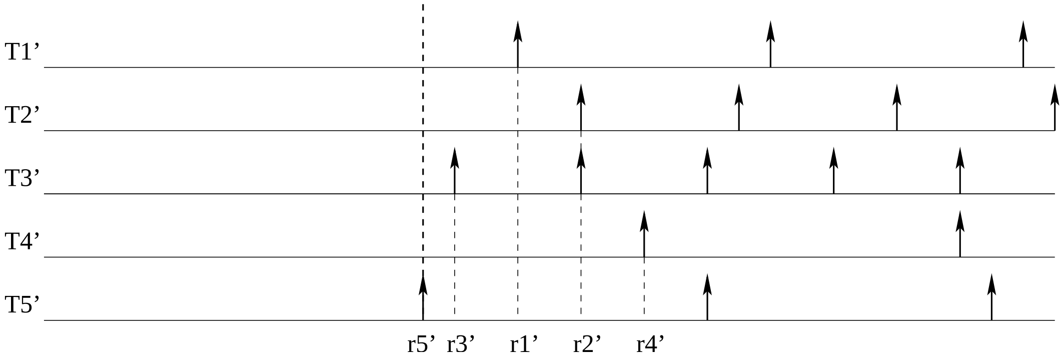


Figure 3.4: Back to τ with arbitrary r'_i

($\longleftarrow$) A CP/CS task set T' of $\uparrow P$ is itself an instance of the NCS task set τ. Given that τ is non-preemptively schedulable, we conclude that T' is also non-preemptively schedulable. Hence the original CP/CS task set T is non-preemptively robust. Hence non-preemptive schedulability of NCS task set implies non-preemptive robustness of the corresponding CP/CS task set against $\uparrow P$. ■

Corollary 20. *The following statements are equivalent:*

1. *Non-preemptive robustness of CP task set against* $\uparrow P$
2. *Non-preemptive robustness of CS task set against* $\uparrow P$
3. *Non-preemptive robustness of NCP task set against* $\uparrow P$
4. *Non-preemptive robustness of NCS task set against* $\uparrow P$

Proof: By theorem 19, items 1 and 2 are equivalent to non-preemptive NCS schedulability (let’s call it item 0). When rephrased, non-preemptive robustness of *any instance* of a NCP task set against $\uparrow P$ (i.e., item 3) is equivalent to the same item 0. The former becomes non-preemptive robustness of NCP task set against $\uparrow P$ (item 3).

Finally, by the nature of a sporadic task set, item 4 is equivalent to its own non-preemptive schedulability. We show this by contrapositivity: ($\rightarrow$) If such a sporadic task set (items 4) is not non-preemptively robust, then there exists a concrete instance of $\uparrow P$ that is non-preemptively unschedulable. This instance is itself also a concrete instance of the original NCS task set. Thus the NCS task set is non-preemptively unschedulable. ($\leftarrow$) And anything that is unschedulable is automatically not robust. By transitivity, all the above items 1 to 4 are equivalent to one another. ■

Since non-preemptive robustness of periodic / sporadic task sets against $\uparrow P$ does not depend on the first release times of each task, in what follows we only look at ZCP task set, i.e. the release time of the first instances of all tasks are at time 0. The result is automatically applicable to all CP/CS/NCP/NCS task sets against $\uparrow P$.

3.3.2 Increase in Period / Minimum Separation ($\uparrow P$) vs. Decrease in Computation Time ($\downarrow C$)

$\downarrow C$ causes anomaly only when it creates a priority inversion. We are going to show in the next theorem that any such priority inversion could be simulated by $\uparrow P$. If a task set is non-preemptively robust against $\uparrow P$ for a scheduler, then any such priority inversion cannot result in anomaly, including those induced by $\downarrow C$. Hence the task set is also non-preemptively robust against $\downarrow C$ for the same scheduler.

Theorem 21. *Non-preemptive robustness of a CP/CS task set against $\uparrow P$ is sufficient to guarantee non-preemptive robustness of the same task set against $\downarrow C$.*

Proof: Suppose the contrary is true that a CP/CS task set $T = \{(C_1, P_1), (C_2, P_2), \ldots, (C_n, P_n)\}$ is non-preemptively robust against any $\uparrow P$: $\{(C_1, P_1'), (C_2, P_2'), \ldots, (C_n, P_n')\}$ but misses its deadline for certain $\downarrow C$: $T' = \{(C_1', P_1), (C_2', P_2), \ldots, (C_n', P_n)\}$. Let t be the time of the earliest deadline miss. Let T_v' be the task that misses its deadline at time t and let $r'(= t - P_v)$ be the request time of that task instance. Let T_c' be the task executing at time r' and let s' be the start time of that task instance. (See figure 3.5).

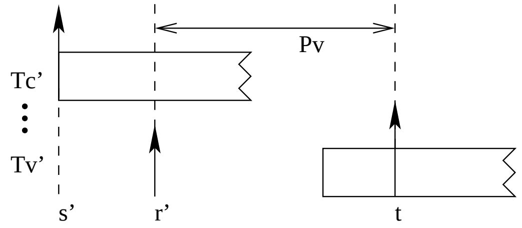


Figure 3.5: Identification of time points s', r' and t

Consider another CP task set T'' with the same C_i and P_i but with the following release times. Let the release time of task T_c'' be at time s', and the release time of all other tasks be at time r'. T_c'' will start execution at time s' because it is the only released task then. Between r' and t, the total demand for computation time is no less than in T', because there are now the maximum number of requests for each task, and the computation time for each task is also restored to C_i. Hence if T_v' misses its deadline at t, then the task set T'' would also miss its deadline no later than time t.

However, as T is non-preemptively robust against $\uparrow P$, according to corollary 20, T'' is non-preemptively schedulable as an instance of the NCP task set, which is a contradiction. So T must also be non-preemptively robust against $\downarrow C$.

■

Corollary 22. *Non-preemptive robustness of CP task set against $\uparrow P$ (item 0) is equivalent to the following general non-preemptive robustness:*

1. *Non-preemptive robustness of CP task set*

2. *Non-preemptive robustness of CS task set*

3. *Non-preemptive robustness of NCP task set*

4. *Non-preemptive robustness of NCS task set*

Proof: This follows directly from corollary 20 and theorem 21. By theorem 21, item 0 is equivalent to item 1. By corollary 20, item 0 is equivalent to non-preemptive robustness of CS task set against $\uparrow P$, which by theorem 21 is equivalent to item 2.

This could be generalized to items 3 and 4 because the derivation above is valid for *any instance* of the NCP/NCS task set. So, non-preemptive robustness of NCP task set against $\uparrow P$ is equivalent to item 3; and that non-preemptive robustness of NCS task set against $\uparrow P$ is equivalent to item 4. Corollary 20 shows them to be equivalent to item 0 too. Hence all the above items are equivalent to one another.

■

In what follows, we will consider the non-preemptive robustness of only the ZCP task set against $\uparrow P$, and the result automatically applies to all task sets against both $\uparrow P$ and $\downarrow C$.

3.4 Common Non-Preemptive Schedulers

Next, we derive necessary and sufficient conditions on task set to guarantee its robustness for both NPEDF and NPFP under discrete time or dense time separately.

3.4.1 Discrete-Time, NPEDF Scheduler

Jeffay et. al. [28] proposed a necessary and sufficient condition for NPEDF schedulability of NCP task, NCS task and CS task. We want to prove that this is also the necessary and sufficient condition for NPEDF robustness under discrete time. A ZCP task is a special case of the CS task. For the sake of easy reference, Jeffay et. al.'s NPEDF schedulability condition is specialized to the case of ZCS task and reproduced here:

Theorem 23. (special case of theorem 4.1 in [28]) *Let $T = \{T_1, T_2, \ldots, T_n\}$, where $T_i = (C_i, P_i)$, be a set of ZCS tasks sorted in non-decreasing order by the minimum separation P_i (i.e. for any pair of tasks T_i and T_j, if $i > j$, then $P_i \geq P_j$). If T is schedulable, then*

1. $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$
2. $\forall i, 1 < i \leq n; \quad \forall L, P_1 < L < P_i :$

$$L \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{P_j} \right\rfloor \cdot C_j$$

A direct application of the theorem does not establish robustness because the $\forall L$ term above expands under $\uparrow P$. It is easy to see why this condition is a sufficient condition for NPEDF robustness of CP task set. We next prove that this is also a necessary condition for ZCP task set.

Theorem 24. *The condition in theorem 23 is also a necessary condition for NPEDF robustness of ZCP task set against $\uparrow P$.*

Proof: Suppose the condition does not hold, we construct a task set T' of reduced system load where there is a deadline miss. Let $t = \text{lcm}(P_1, P_2, \ldots, P_n)$ is the least common multiple of all the periods. Consider when the release time of the second instance of task T_i' is delayed to time $2t-1$. By our construction[2], there is no outstanding computation at time $2t-1$, so task T_i' begins execution at time $2t-1$. Consider the time interval from $2t-1$ to $2t+L-1$. Since the second condition does not hold, the total available time is less than the requested amount of time, so there is a deadline miss no later than time $2t+L-1$. ■

Now, consider the case where a reduction in system load is $\downarrow C$. Both inequalities in Jeffay et. al.'s conditions continue to hold when C_i decreases. It means that once the above schedulability conditions are met, it remains schedulable under $\downarrow C$. It also means that under $\downarrow C$, a ZCP task set remains robust against $\uparrow P$ for

[2] With discrete time, $C_i \geq 1$. With eager scheduler and all first task instances arrive at time 0, there cannot be outstanding computation at time $2t-1$ when task T_i is taken out of the time interval $[t, 2t)$, otherwise the original task set would not have been schedulable.

NPEDF scheduler. Put it together, Jeffay et. al.'s schedulability condition in theorem 23 is exactly the necessary and sufficient conditions for NPEDF robustness of ZCP task set if discrete time is adopted. (The same conclusion could also be derived by applying theorem 21).

Alternatively, when translated to the terminology of this chapter, NPEDF schedulability of ZCS task sets is equivalent to NPEDF robustness of ZCP task sets, if the only kind of reduction in system load allowed is $\uparrow P$. Hence if discrete time and NPEDF scheduler are adopted, and the only kind of reduction in system load allowed is $\uparrow P$, then Jeffay et. al.'s necessary and sufficient condition for ZCS schedulability is exactly the necessary and sufficient condition for ZCP robustness. In fact, by corollary 22, we conclude that theorem 23 is also the necessary and sufficient conditions for NPEDF robustness of CP/CS/NCP/NCS task sets.

The running time of this test is $O(n^2 P_{max})$, where $P_{max} = \max_{1 \leq i \leq n} P_i$.

3.4.2 Dense Time, NPEDF Scheduler

Theorem 23 does not work for dense time because the clause $\forall L, P_1 < L < P_n$ makes it computationally intractable. Also, $L - 1$ in the numerator would become $L - \delta$ for infinitesimally small δ. This is in accordance with section 2.4 that testing cannot solve the anomaly problem because no testing can detect all problems.

According to theorem 16 (theorem 11 of [39]), whenever there is a deadline miss, a culprit task could be identified. For easy reference, the theorem is reproduced as follows:

Definition 25. (reproduction of theorem 16) *For both NPFP/RMA and NPEDF, assume the first request of all tasks arrive together. If a certain instance of victim task $T_v^{\prime p}$ whose request time is r' misses a deadline after reduction in system load, then there exists an instance of a culprit task $T_c^{\prime q}$ satisfying these properties:*

- *The culprit task T_c is of lower priority than the victim task T_v (For NPFP/ RMA, static priority means $v < c$; for NPEDF, dynamic priority means deadline of T_c is later than that of T_v).*
- *The culprit task $T_c^{\prime q}$ is executing at time r'.*

As implied directly from the definition, for NPEDF, the deadline miss (deadline of $T_v^{\prime p}$) must occur before the end of the period of $T_c^{\prime q}$ (deadline of $T_c^{\prime q}$), assuming relative deadlines are equal to periods.

Theorem 26. *Let $T = \{T_1, T_2, \ldots, T_n\}$, where $T_i = (C_i, P_i)$, be a zero-concrete task set of n tasks scheduled by NPEDF. The task set is robust against $\uparrow P$ iff $\forall i, 1 \leq i \leq n$, T remains schedulable during the interval $[0, P_i]$ when task T_i is given the highest absolute priority, with all other priority assignment remaining the same as EDF.*

Proof: ($\longleftarrow$) If task T_i could ever be the culprit task, it would cause a deadline miss within P_i time when being promoted to the highest priority. So if the task set remains schedulable when T_i is promoted to the highest priority, then T_i cannot be a culprit task. If no task could act as a culprit task, the task set is robust for NPEDF.

($\rightarrow$) Suppose the condition does not hold for a certain task T_i in the task set, i.e., task T_i causes deadline miss in task T_j when promoted to the highest priority, we construct a scenario with $\uparrow P$ where the task set becomes unschedulable. Let the first instances of all tasks start at time 0. Let the second instances of all tasks except task T_i start at time $2P_{max}$ while the second instance of task T_i start at time $2P_{max} - \delta$ for a sufficiently small δ. By the time $2P_{max} - \delta$, there is no outstanding computation, so task T_i is the only released task and it begins execution. Then all tasks request at the maximum frequency. This causes task T_j to miss the deadline. Hence it is also a necessary condition. ■

According to corollary 22, theorem 26 also holds for all of CP/CS/NCP/ NCS task sets and under all kinds of reduction in system load.

Since both C_i and P_i remain the same when evaluating the condition above, the number of steps in the evaluation is finite and computationally tractable. The running time for this algorithm is $O(\frac{P_{max}}{C_{min}} \cdot n \log n)$, where $P_{max} = \max_{1 \leq i \leq n} P_i$ and $C_{min} = \min_{1 \leq i \leq n} C_i$ are the maximum period and minimum computation time of all tasks in the task set respectively. $\frac{P_{max}}{C_{min}}$ is the maximum number of non-preemptive scheduling events during each test. Each scheduling event involves one priority queue operation which is $O(\log n)$ and the test is repeated n times once for each task in the task set.

3.4.3 Discrete or Dense Time, NPFP Scheduler

Let $P_{max} = \max_{1 \leq i \leq n} P_i$. When there is a priority inversion, let $T_c = (C_c, P_c)$ be the *potential* culprit task that may cause a deadline miss to the *potential* victim task $T_v = (C_v, P_v)$. Instead of checking for a deadline miss up to P_c from the priority inversion as in the case of NPEDF, we need to check up to $2P_c$ for NPFP/RMA, and up to $2P_{max}$ for general NPFP.

Theorem 27. *Let $T = \{T_1, T_2, \ldots, T_n\}$, where $T_i = (C_i, P_i)$, be a zero-concrete task set of n tasks schedulable by an NPFP scheduler with priority assignment w. The following constitutes the necessary and sufficient condition for robustness:*

$\forall i, 1 \leq i \leq n$, τ remains non-preemptively schedulable during the interval $[0 \ldots 2P_{max}]$ (or $[0 \ldots 2P_c]$ for NPFP/RMA) when task T_i is promoted to the highest priority, with all other priority assignments the same as in w.

Proof: According to theorem 21, we consider only $\uparrow P$. Since the culprit task must be executing by the time the victim task is released, the time interval between start of execution of culprit task instance to deadline miss of victim task instance is less than $P_v + C_c$. For NPFP/RMA, we have $P_v < P_c$ and $C_c < P_c$, so $P_v + C_c < 2P_c$ and for general NPFP, we have $P_v \leq P_{max}$ and $C_c < P_{max}$, so $P_v + C_c < 2P_{max}$.

If task i could ever be a victim task, then it would have caused deadline miss within a time interval of $2P_{max}$ for general NPFP and $2P_i$ for NPFP/RMA when it is promoted to the highest priority. If there is no deadline miss when T_i is promoted to the highest priority, then T_i cannot be a culprit task. If none of the tasks could be a culprit task, there could not be any deadline miss under reduction in system load, hence the task set is robust.

On the contrary, if task i causes task j to miss a deadline within the said time interval when task i is promoted to the highest priority, we can construct a task set of reduced system load in which task i is the culprit task and task j is the victim task that misses the deadline:

Let the first instances of all tasks start at the same time. Let the second instances of all tasks except task T_i start at time $2P_{max}$ while the second instance of task T_i start at time $2P_{max} - \delta$, where $\delta = 1$ for discrete time and δ is a sufficiently small value for dense time. By the time $2P_{max} - \delta$, there is no outstanding computation, so task T_i as the only released task is selected for execution. Then all tasks request at their maximum frequency. This causes T_j to miss its deadline. Hence it is also a necessary condition. ■

According to corollary 22, theorem 27 also holds for all of CP/CS/NCP/ NCS task sets and under all kinds of reduction in system load.

The running time of this test is $O(\frac{P_{max}}{C_{min}} \cdot n \log n)$, where $P_{max} = \max_{1 \leq i \leq n} P_i$ and $C_{min} = \min_{1 \leq i \leq n} C_i$ are the maximum period and minimum computation time among all tasks in the task set respectively. $\frac{2P_{max}}{C_{min}}$ is the maximum number of non-preemptive scheduling events during the time period under test. Each scheduling event involves a priority queue operation which is $O(\log n)$. The test is repeated n times over all tasks.

3.5 Special Cases for Non-preemptive Robustness

In order to not miss deadlines for non-preemptive real-time application, it is important to know the criteria for non-preemptive robustness. While a necessary and sufficient condition is often too complicated to use, we present some sufficient conditions with reasonable utilization bounds.

3.5.1 Geometric Envelope Task Set

When the sum of computation time of all tasks in the set do not exceed the shortest period, the task set is obviously robust. The utilization factor for such a task set could still be reasonably high. Consider the following parameterized task set normalized to $P_1 = 1$

T_1	$\left(\beta_1 \cdot \left(1 - \frac{1}{x}\right), 1\right)$
T_2	$\left(\beta_2 \cdot \left(1 - \frac{1}{x}\right) \cdot \frac{1}{x}, x\right)$
$\vdots$	$\vdots$
T_i	$\left(\beta_i \cdot \left(1 - \frac{1}{x}\right) \cdot \frac{1}{x^{i-1}}, x^{i-1}\right)$
$\vdots$	$\vdots$

where $x > 1$ and $\forall i, 0 < \beta_i \leq 1$. Let $\beta = \min(\beta_i)$. We can see that:

$$\begin{aligned} \sum_{i=1}^{\infty} C_i &= \sum_{i=1}^{\infty} \beta_i \cdot \left(1 - \frac{1}{x}\right) \cdot \frac{1}{x^{i-1}} \\ &\leq \sum_{i=1}^{\infty} \left(1 - \frac{1}{x}\right) \cdot \frac{1}{x^{i-1}} \\ &= \frac{1 - \frac{1}{x}}{1 - \frac{1}{x}} = 1 = P_1 \end{aligned}$$

so the task set is robust against reduction in system load for NPFP/RMA and

NPEDF scheduling policies. The utilization factor

$$\begin{aligned} U = \sum_{i=1}^{\infty} \frac{C_i}{P_i} &= \sum_{i=1}^{\infty} \frac{\beta_i \cdot \left(1 - \frac{1}{x}\right) \cdot \frac{1}{x^{i-1}}}{x^{i-1}} \\ &\geq \beta \cdot \sum_{i=1}^{\infty} \frac{\left(1 - \frac{1}{x}\right) \cdot \frac{1}{x^{i-1}}}{x^{i-1}} \\ &\geq \beta \cdot \frac{1 - \frac{1}{x}}{1 - \frac{1}{x^2}} \\ &= \beta \cdot \frac{x^2 - x}{x^2 - 1} \\ &= \beta \cdot \frac{x}{x + 1} \end{aligned}$$

Taking the limit $x \to 1$, $U \to \beta \cdot \frac{1}{2}$ and taking the limit $x \to \infty$, $U \to \beta$. As an example, suppose $x = 2$ and $\beta = 0.9$, meaning that the anomaly-free task set deviates from the geometric series envelope by at most 10%, then the utilization factor can be as high as $\frac{3}{5} \leq U \leq \frac{2}{3}$.

3.5.2 No-Blocking Test

Theorem 28. *Let an arbitrary task T_i within a task set M be schedulable under NPFP/RMA. Task T_i remains schedulable under reduction in system load if*

$$\max_{i+1 \leq r \leq N} C_r \leq P_i - \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{P_i}{P_j} \right\rceil$$

If all tasks within the task set M remain schedulable under reduction in system load, then NPFP/RMA is robust for this task set M under reduction in system load.

Proof: Suppose the contrary is true that a certain miss occurs for task T_i after reduction in system load. Let the request time and the deadline for the miss be r

and d respectively. There could be at most one lower priority task between r and d, otherwise by the time the second lower priority task is scheduled to execute, there is no outstanding computation for T_i and hence it would not have missed the deadline.

Let t be the latest time no later than r with no outstanding computation. By theorem 16, there exists at least one culprit task between t and d. Let the start time of the latest culprit task between r and d be s. At time s, there could be no outstanding computation for tasks 1 to i, so $s < r$ otherwise the miss could not have occurred.

Consider the time between s and d. Except the culprit task T_c, the victim task T_v is the lowest priority task between s and d. So by lemma 11, the response time of T_v is the longest when the requests of all the higher priority tasks are aligned with s. After reduction in system load, the maximum amount of computation time demanded between s and d is:

$$\begin{aligned}
& C'_c + \sum_{j=1}^{i} C'_j \cdot \left\lceil \frac{d-s}{P'_j} \right\rceil \\
\leq \quad & C_c + \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{d-s}{P_i} \cdot \frac{P_i}{P_j} \right\rceil \\
\leq \quad & C_c + \frac{d-s}{P_i} \cdot \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{P_i}{P_j} \right\rceil \\
< \quad & \frac{d-s}{P_i} \cdot \left(\max_{i+1 \leq r \leq N} C_r + \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{P_i}{P_j} \right\rceil \right) \\
\leq \quad & \frac{d-s}{P_i} \cdot P_i = d - s
\end{aligned}$$

The total required computation time is less than the available time, so there could not be a deadline miss at d, which is a contradiction. Hence the task set remains schedulable using NPFP/RMA under reduction in system load. ■

Note that this is a sufficient but not necessary condition.

3.5.3 Necessary and Sufficient Condition of Robustness for Task Set of Successively Divisible Period

A task set with N tasks is said to exhibit successively divisible period if

$$\forall i, j, \quad 1 \leq j \leq i \leq N, \quad \left\lfloor \frac{P_i}{P_j} \right\rfloor = \frac{P_i}{P_j} = \left\lceil \frac{P_i}{P_j} \right\rceil$$

A common example would be when the periods form a subset of the geometric series.

Theorem 29. *Let an arbitrary task T_i within in task set M with successively divisible periods be schedulable under NPFP/RMA. A necessary and sufficient condition for task T_i to remain schedulable under reduction in system load is*

$$\max_{i+1 \leq r \leq N} C_r \leq P_i - \sum_{j=1}^{i} C_j \cdot \frac{P_i}{P_j}$$

If all tasks in the task set M remains schedulable under reduction in system load, then NPFP/RMA is robust for this task set under reduction in system load.

Proof: Substituting the definition of successively divisible period that $\forall i, j$: $\left\lceil \frac{P_i}{P_j} \right\rceil = \frac{P_i}{P_j}$ into theorem 28, we see that the above inequality is a sufficient condition of robustness. So in what follows, we only need to prove that this is also a necessary condition.

Suppose the inequality above does not hold, we have

$$\exists i, r, \quad 1 \leq i < r \leq N, \quad C_r > P_i - \sum_{j=1}^{i} C_j \cdot \frac{P_i}{P_j}$$

and we want to construct a certain reduction in system load, where there is a deadline miss in some task T_j, where $1 \leq j \leq i$.

Let $x = P_r$. From time 0 to $2x$, there are two instances of task T_r. Consider if we increase the period of task T_r from P_r to $P_r' = 2x - \delta$ for some very small δ.

Given that δ is very small, we can conclude that $\delta < C_r$, so there could be no outstanding computation for any task at time $2x - \delta$ after this reduction in system load. Now, the task instance T_r^2 would start right at its request time $2x - \delta$, while requests for all other tasks arrive simultaneously at time $2x$.

From time $2x$ to $2x + P_i$, the exact amount of works that needs to be done in order to avoid any deadline miss for tasks T_j, $1 \leq j \leq i$, would be greater than the total available time

$$C_r + \sum_{j=1}^{i} C_j \cdot \frac{P_i}{P_j} > P_i$$

Here P_i represents a deadline for all tasks T_j, where $1 \leq j \leq i$. The required computation time is more that the total available time, so there must be one or more deadline miss in the tasks T_j, where $1 \leq j \leq i$. The situation is worse if there is other culprit task or idling time during this period of time.

Hence the inequality is also a necessary condition of robustness. Putting together, the inequality is a necessary and sufficient condition of robustness for task set of successively divisible period. ■

3.6 Conclusion

The notion of robustness will become increasingly important as CPU speed keeps on improving such that the temporal length of control tasks becomes relatively short compared to the cost of context switching overhead which may incur I/O actions and thus does not scale with CPU speed. The robustness problem is particularly important to mobile computing because such devices are often clocked at a range of frequencies, and adjusted at run time according to need. Hence it is not sufficient to keep track of only the upper bound on resource usage. Our results in non-preemptive robustness provides formal justification for the "folk knowledge" that the use of the longest task in the blocking factor in RMA (preemptive priority scheduling) analysis can also be used in non-preemptive schedulability analysis.

In this chapter, we prove that non-preemptive robustness of a ZCP (zero-concrete periodic) task set against increase in period ($\uparrow P$) is sufficient to guarantee robustness of the corresponding concrete or non-concrete, periodic or sporadic task set against any kinds of reduction in system load.

Based on this result, we derived the necessary and sufficient conditions of robustness for both Non-preemptive fixed-priority (NPFP) and non-preemptive earliest-deadline-first (NPEDF) schedulers under both discrete time and dense time assumptions separately. It has pseudo-polynomial running time and is potentially practical for use in adaptive real-time systems where not only the timing parameters but the task sets themselves may change at run time to adapt to environmental conditions.

Besides the general necessary and sufficient conditions for non-preemptive robustness, we also formulated a set of sufficient conditions for special cases, with reasonable performance bounds. In particular, interesting properties could be derived when the task set exhibits a property we called successively divisible periods.

Chapter 4

Time Sharing Limited-Preemptable Resources and Mixed-Type Resources

In this chapter, we would like to examine the issue of robustness for hybrid schedulers where task preemption is allowed in limited ways in order to strike a balance between optimizing schedulability bounds and preserving robustness. This can be done, for example, by restricting the frequency of task preemption by both compilation methods and run-time enforcement mechanisms.

4.1 Introduction

Most real life examples do not fall entirely into fully preemptive nor fully non-preemptive task models, but somewhere in-between. In sections 3.4.1 and 3.4.3, we already discussed the admission control for non-preemptive robustness when time is discrete.

Under discrete time, events could only happen at certain time points, separated by integer multiples of a time quantum q. Without loss of generality, the discussion in sections 3.4.1 and 3.4.3 assumed $q = 1$ clock cycle. We shall relax the assumption in this chapter by adopting wall clock time instead of cycle time and thereby allowing an arbitrary value for q. Scheduling decision, being a class of events, is allowed to happen only on a subset of these time points.

Depending on the exact model where scheduling decisions are allowed to be made, our scheduling analysis needs to be massaged to suit for a variety of task models. We are going to explore the robustness for some common cases in the this chapter, including limited preemption, aperiodic tasks and mixed resource types.

4.2 Robustness for Limited Preemption

If we allow limited preemption, the performance can be much better. Limited preemption is characterized by a granularity g, which is the minimum number of time units that a task, once scheduled, is allowed to run before it is allowed to be preempted. Note that g is an integer multiple of the time quantum q.

Theorem 30. *If we allow limited preemption with a granularity of g, then by adopting EDF scheduler, a task won't miss its deadline by more than $g - 1$ time units under reduction in system load.*

Proof: Mok et. al. [35] have proved that if the utilization factor $U \leq 1$, then by adopting EDF scheduler with granularity g, no task would ever miss its deadline by more than $g - 1$ time units. After reduction in system load, the utilization factor becomes smaller, hence by the same argument, no task would miss its deadline by more than $g - 1$ time units. ■

Corollary 31. *If a task set is schedulable by the limited-preemptive EDF scheduler when $g-1$ time units are added to all the computation time of the tasks (or better, if deadline is administered separately from period, that $g-1$ time units are subtracted* *from the deadline of all tasks), then no task would ever miss its deadline under reduction in system load.*

Proof: This follows directly from theorem 30. ■

4.3 Robustness on a Bounded Delay Resource Partition (BDRP)

In chapter 2 and 3, we discussed the preemptive and non-preemptive robustness problem on dedicated resource. In this section, we are going to apply the robustness analysis on a BDRP that has been introduced in definition 2.

For the sake of easy reference, definition 2 is reproduced here:

Definition 32. *A* ***Bounded Delay Resource Partition (BDRP)*** Π *is a tuple* (α, Δ) *where* α *is the percentage of total time the resource is available to the partition and* Δ *is called the* ***Partition Delay****, which is the largest time deviation of a partition during any time interval with regards to a uniform uninterrupted allocate of the resource.*

Consider task level scheduling (see section 1.2.1.1), we want to know if schedulability is preserved under a reduction in system load when the task set is scheduled on a BDRP instead of on a dedicated resource. Again, we consider the two common schedulers Earliest Deadline First (EDF) and Fixed Priority (FP).

For a formal definition and analysis of reduction in system load, please refer to sections 2.1.3 and 3.1.2. For a formal definition of robustness, please refer to sections 2.1.4 and 3.1.3.

4.3.1 Preemptive Robustness on a Bounded Delay Resource Partition (BDRP)

A task set $M = \{(C_1, P_1), \ldots, (C_n, P_n)\}$ is schedulable on a BDRP $\Pi = (\alpha, \Delta)$ by a certain scheduling policy if all task execution finishes at least Δ time units before their respective deadlines.

4.3.1.1 Preemptive Earliest-Deadline-First (PEDF)

For the PEDF scheduling policy, finishing task execution before their respective deadlines on a given BDRP means

$$\forall i (1 \leq i \leq n), \exists t_i \in (0, P_i - \Delta], C_i + \sum_{j=1}^{i-1} \frac{t_i \cdot C_j}{P_j} \leq \alpha \cdot t_i$$

Since that task set M is schedulable on the BDRP, we have a set of values $t_1, t_2, \ldots, t_n$ satisfying the above inequality.

In addition to the criteria on Δ above, the task set M is schedulable with PEDF on the BDRP $\Pi = (\alpha, \Delta)$ if the total utilization factor is not more than available rate of the resource α for the partition, i.e. $U = \sum_{j=1}^{n} \frac{C_j}{P_j} \leq \alpha$.

After reduction in system load, we want to make sure that the total utilization factor remains less than α of the partition, i.e. $U = \sum_{j=1}^{n} \frac{C'_j}{P'_j} \leq \alpha$ and that there exists a set of values $t'_1, t'_2, \ldots, t'_n$ satisfying:

$$\forall i (1 \leq i \leq n), t'_i \in (0, P'_i - \Delta] \wedge C'_i + \sum_{j=1}^{i-1} \frac{t'_i \cdot C'_j}{P'_j} \leq \alpha \cdot t'_i$$

For decrease in computation time of a certain task C_k by δ $(1 \leq k \leq n, \delta >$

$0)$, the total utilization factor is less than before.

$$\sum_{j=1}^{n} \frac{C'_j}{P'_j} = \left(\sum_{j=1}^{n} \frac{C_j}{P_j} \right) - \frac{\delta}{P_k} < \sum_{j=1}^{n} \frac{C_j}{P_j} \leq \alpha$$

We also need to look at the worst case response time. For task T_k, we take $t'_k = t_k \in (0, P_k - \Delta] = (0, P'_k - \Delta]$, then

$$C'_k + \sum_{j=1}^{k-1} \frac{t'_k \cdot C'_j}{P'_j} = C_k - \delta + \sum_{j=1}^{k-1} \frac{t_k \cdot C_j}{P_j} < C_k + \sum_{j=1}^{k-1} \frac{t_i \cdot C_j}{P_j} \leq \alpha \cdot t_k = \alpha \cdot t'_k$$

and $\forall i$ ($k < i \leq n$, otherwise if $i < k$ the response time of task T_i clearly remains the same), we take $t'_i = t_i \in (0, P_i - \Delta] = (0, P'_i - \Delta]$, then

$$\begin{aligned} C'_i + \sum_{j=1}^{i-1} \frac{t'_i \cdot C'_j}{P'_j} &= \left(C_i + \sum_{j=1}^{i-1} \frac{t_i \cdot C_j}{P_j} \right) - \frac{t_i \cdot \delta}{P_k} \\ &< C_i + \sum_{j=1}^{i-1} \frac{t_i \cdot C_j}{P_j} \\ &\leq \alpha \cdot t_i \\ &= \alpha \cdot t'_i \end{aligned}$$

Hence PEDF is robust against decrease in task computation time on a given BDRP.

For increase in period of some task P_k by δ $(1 \leq k \leq n, \delta > 0)$, we first consider when job priorities do not change. The total utilization factor is less than before.

$$\sum_{j=1}^{n} \frac{C'_j}{P'_j} = \left(\sum_{j=1}^{n} \frac{C_j}{P_j} \right) - \left(\frac{C_k}{P_k} - \frac{C_k}{P_k + \delta} \right) < \sum_{j=1}^{n} \frac{C_j}{P_j} \leq \alpha$$

We also need to look at the worst case response time. For task T_k, we take $t'_k = t_k \in (0, P_k - \Delta] \subset (0, P'_k - \Delta]$, then

$$C'_k + \sum_{j=1}^{k-1} \frac{t'_k \cdot C'_j}{P'_j} = C_k + \sum_{j=1}^{k-1} \frac{t_k \cdot C_j}{P_j} \leq \alpha \cdot t_k = \alpha \cdot t'_k$$

and $\forall i$ ($k < i \leq n$, otherwise if $i < k$ the response time of task T_i clearly remains the same), we take $t'_i = t_i \in (0, P_i - \Delta] \subset (0, P'_i - \Delta]$, then

$$\begin{aligned} C'_i + \sum_{j=1}^{i-1} \frac{t'_i \cdot C'_j}{P'_j} &= \left(C_i + \sum_{j=1}^{i-1} \frac{t_i \cdot C_j}{P_j} \right) - \left(\frac{t_i \cdot C_k}{P_k} - \frac{t_i \cdot C_k}{P_k + \delta} \right) \\ &< C_i + \sum_{j=1}^{i-1} \frac{t_i \cdot C_j}{P_j} \\ &\leq \alpha \cdot t_i \\ &= \alpha \cdot t'_i \end{aligned}$$

Hence PEDF is robust against increase in task period on a given BDRP, provided that task priorities do not change.

Using the same construction as in section 2.2.2, it can be shown that PEDF is also robust against increase in task period on a given BDRP when the task priorities change accordingly.

In other words, PEDF scheduling policy is robust against any reduction in system load when applied on any given BDRP.

4.3.1.2 Preemptive Fixed Priority (PFP)

For the PFP scheduling policy, finishing task execution before their respective deadlines on a given BDRP means

$$\forall i (1 \leq i \leq n), \exists t_i \in (0, P_i - \Delta], \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \leq \alpha \cdot t_i$$

Since that task set M is schedulable on the BDRP, we have a set of values $t_1, t_2, \ldots, t_n$ satisfying the above inequality.

After reduction in system load, we want to find a set of values $t'_1, t'_2, \ldots, t'_n$ satisfying:

$$\forall i(1 \leq i \leq n), t'_i \in (0, P'_i - \Delta] \wedge \sum_{j=1}^{i} C'_j \cdot \left\lceil \frac{t'_i}{P'_j} \right\rceil \leq \alpha \cdot t'_i$$

For decrease in computation time of a certain task C_k by δ $(1 \leq k \leq n, \delta > 0)$, then $\forall i$ ($k \leq i \leq n$, otherwise if $i < k$ the response time of task T_i clearly remains the same), we take $t'_i = t_i \in (0, P_i - \Delta] = (0, P'_i - \Delta]$

$$\begin{aligned}
\sum_{j=1}^{i} C'_j \cdot \left\lceil \frac{t'_i}{P'_j} \right\rceil &= \left(\sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \right) - \delta \cdot \left\lceil \frac{t_i}{P_k} \right\rceil \\
&< \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \\
&\leq \alpha \cdot t_i \\
&= \alpha \cdot t'_i
\end{aligned}$$

Hence PFP is robust against decrease in task computation time on a given BDRP.

For increase in period of some task P_k by δ $(1 \leq k \leq n, \delta > 0)$, we first consider when job priorities do not change: $\forall i$ ($k \leq i \leq n$, otherwise if $i < k$ the response time of task T_i clearly remains the same), we take $t'_i = t_i \in (0, P_i - \Delta] \subset (0, P'_i - \Delta]$, then

$$\begin{aligned}
\sum_{j=1}^{i} C'_j \cdot \left\lceil \frac{t'_i}{P'_j} \right\rceil &= \left(\sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \right) - C_k \cdot \left(\left\lceil \frac{t_i}{P_k} \right\rceil - \left\lceil \frac{t_i}{P_k + \delta} \right\rceil \right) \\
&< \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t_i}{P_j} \right\rceil \\
&\leq \alpha \cdot t_i \\
&= \alpha \cdot t'_i
\end{aligned}$$

Hence PFP is robust against increase in task period on a given BDRP, provided that task priorities do not change.

Using the same construction as in section 2.2.2, it can be shown that PFP is also robust against increase in task period on a given BDRP when the task priorities change accordingly.

In other words, PFP scheduling policy is robust against any reduction in system load when applied on any given BDRP.

4.3.2 Non-Preemptive Robustness on a Bounded Delay Resource Partition (BDRP)

Notice that when we assign a non-preemptive task to a BDRP, the non-preemptive task may not always be running once it is scheduled for execution because the partition itself could be de-scheduled by the resource level scheduling (section 1.2.1.2). In that case, once the partition is brought back for execution, the non-preemptive task that has been running inside the partition picks up where it left off and continues until completion (or another de-scheduling of the partition). The task could still be treated as non-preemptive because it is a prerequisite that there are no dependencies between different partitions in the system, so partitions could be freely preempted even if the tasks running inside are in their respective critical sections.

Non-preemptive Earliest Deadline First (NPEDF) and Non-preemptive Fixed Priority (NPFP) schedulers are known to be not robust against reduction in system load even on dedicated resources (see section 2.3. Hence they are also not robust on

BDRP. Instead of looking at how un-robust they are, we would instead determine if the necessary and sufficient conditions given in sections 3.4.1 and 3.4.3 could be adapted for use on BDRP. We consider only discrete time with time quantum q in this chapter.

We state without proof in this section the necessary and sufficient conditions for non-preemptive robustness of concrete or non-concrete, periodic or sporadic task sets using NPEDF on a BDRP.

Theorem 33. *Let $T = \{T_1, T_2, \ldots, T_n\}$, where $T_i = (C_i, P_i)$, be a set of concrete or non-concrete, periodic or sporadic tasks sorted in non-decreasing order of the period or minimum separation P_i (i.e. for any pair of tasks T_i and T_j, if $i > j$, then $P_i \geq P_j$). If T is schedulable by NPEDF with time quantum q on a BDRP $\Pi = (\alpha, \Delta)$, then*

1. $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \alpha$
2. $\forall i, 1 < i \leq n; \quad \forall L, P_1 < L < P_i :$

$$\alpha \cdot L \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L - q}{P_j} \right\rfloor \cdot C_j$$

4.4 Resource Level Scheduling with Aperiodic Tasks under Discrete Time

Construction of BDRP by Mok et. al. [37] [36] was based on the periodic task model. Up till now, this dissertation also assumed either the periodic task model or the sporadic task model. We are going to show in this section that we may as well construct BDRP with an aperiodic task model.

When task sets are aperiodic, we do not have the notion of period nor minimum separation. Each task is equivalent to having only a single on-going instance that needs to be scheduled and de-scheduled from time to time. Each aperidic task is characterized by a percentage α of resource that the task demands, much like the utilization factor in the periodic task model.

It is meaningless to talk about non-preemptive scheduling for aperiodic task sets because each aperiodic task in the task set would go on forever. If time is dense, we could have infinite time-slicing [32] so there is no robustness problem. However, the robustness problem becomes interesting when we allow only limited preemption by imposing the discrete time requirement.

4.4.1 The Problem

An aperiodic partition group consists of n aperiodic partitions, $\{(\Pi_1, \Pi_2, \ldots, \Pi_n)\}$ where each aperiodic partition is specified by $\Pi_i = (\alpha_i, \Delta_i)$. α_i which is the percentage of the resource demanded by partition i, and Δ_i is the partition delay.

Let g be the scheduling granularity, which is an integer multiple of the time quantum q. Scheduling decisions could only be made once every g time units. We will consider two cases, namely, when g is a system-fixed constant, and when g is a variable that could be adjusted per resource.

Our goal is to find a way to partition the resource among a set of aperiodic tasks in discrete time with scheduling granularity g such that real-time application could be run within each such partition.

Note that our partitions Π_i ($1 \leq i \leq n$) does not satisfy the definition of Periodic Resource Partition (see definition 1) as developed by Mok et. al. [37] [36] because ours are aperiodic. However, we still satisfy their definition of a BDRP (see definition 2) because the notion of BDRP does not intrinsically require the presence of a constant and finite period.

The algorithm presented by Mok et. al. [37] [36] to compute the BDRPs assumed the existence of a constant and finite period, therefore they adopted exhaustive search to find the optimal solution. We cannot use their offline exhaustive search algorithm because our task model is aperiodic. We need to find an online heuristic algorithm to compute the BDRPs with their Δ values good enough for real-time workloads.

4.4.2 VMware ESX Server Case Study

The problem was motivated by the scheduling model in VMware ESX server. A number of virtual CPUs (VCPU) are to be scheduled on a fixed number of physical CPUs (PCPU). Basically, a proportional share scheduler is implemented assuming an aperiodic task model. The scheduler kicks in once every 50ms on every PCPU to determine if the current VCPU should be preempted and another VCPU should be selected for execution. Therefore this is a limited preemptive aperiodic task model.

As a simplification, if all VMs in the system are uniprocessor VMs, then

VCPU migration between PCPUs are minimal and negligible. In this dissertation, we only consider time sharing a single PCPU among many VCPUs. We want to be able to reap the same real-time benefits with an aperiodic task model in the same way as we did in periodic or sporadic task models in our previous discussion.

4.4.3 A Parametric Delay Bound Δ Solution

Given a partition group with n partitions $\{\Pi_i = (\alpha_i, \Delta_i)\}$ for $1 \leq i \leq n$, Mok et. al. [37] [36] have given a dynamic schedule whereas the partitions could be computed using the Earliest Deadline First (EDF) scheduler if we consider each partition Π_i as a periodic task (C_i, P_i) with $P_i = \frac{\Delta_i}{2(1-\alpha_i)}$ and $C_i = \alpha_i P_i = \frac{\alpha_i \Delta_i}{2(1-\alpha_i)}$.

In our limited preemptive model, C_i must be an integral multiple of the time quantum q. Hence $\exists n, n \in \mathbb{Z}^+, C_i = nq$. We do the derivation on C_i instead of P_i because C_i is a real quantity whereas P_i is imaginary in our aperiodic model. Re-arranging and solving for Δ_i, we have:

$$\Delta_i = \frac{2nq(1-\alpha_i)}{\alpha_i}$$

We could adopt EDF on an imaginary deadline P_i for each partition i when we compute the partition schedule. Note that this imaginary period P_i does not have to be an integral multiple of q because it does not have a real manifestation in our aperiodic model.

$$P_i = \frac{\Delta_i}{2(1-\alpha_i)} = \frac{2nq(1-\alpha_i)}{2\alpha_i(1-\alpha_i)} = \frac{nq}{\alpha_i}$$

This solution keeps the resource demand α_i of each partition intact, but makes the partition delay Δ_i dependent upon the resource demand of the same partition α_i as well as the time quantum q. Thus the partition delay Δ_i is no longer a constant, and also no longer a parameter that could be freely specified by the user. The only factor the user is still free to influence on the partition delay Δ_i is the integer n. The ability to attain high system utilization is retained because α_i is kept intact.

4.4.4 Why Isn't Parametric Granularity a Good Idea?

One could argue that the situation could be further improved if we make the granularity g roughly inversely proportional to the resource demand α. This way, within a certain operating range (limited by the time quantum q as g cannot drop below q), we could achieve BDRP with essentially the same parameters α and Δ that are fully and independently specifiable by the user, thus not sacrificing any performance.

However, this is not a good idea because it breaks VM isolation. Granularity g is a parameter of the system whereas α and Δ are parameters of the partition. There are many partitions in a system. If we make a system parameter dependent upon the parameters of the (often changing) partitions running on the system, we are breaking the isolation between the partitions. When we add or remove a partition, or change the parameters of a partition, the behavior of other partitions are also affected because the system parameter g that affects all partitions changes accordingly. This is undesirable for real-time systems.

4.5 Time Sharing Mixed Set of Resources

The realistic computing system contains some instances of all the above resource types, so it is important to be able to properly integrate the above paradigms into a unified real-time resource scheduling framework.

Assume there is time quantum q in the system (which is realistic), periodic and aperiodic partitions could be easily mixed because the imaginary periods derived in the aperiodic partitions are compatible with those real periods from the periodic partitions. They can be scheduled together with the same online Earliest Deadline First (EDF) scheduler, provided that we treat the deadlines of periodic partitions are half their respective periods. The division by 2 is to guarantee the maximum separation of executions between two adjacent periods to be less than the partition delay Δ requirement.

Real-time and non-realtime workloads could also be mixed by dedicating a BDRP for non-realtime workloads. Thus failure or deadline miss from the non-realtime workloads would not affect any other real-time workloads because they isolated from each other by residing in different partitions.

4.6 Conclusion

All computing systems carry intrinsic minimum addressable time quantum. When this time quantum is significantly large, or when the nature of the workload demands minimal interruption, we resort to using limited-preemptive scheduling. The robustness of limited-preemptive scheduling discussed in this chapter completes the spectrum of real-time resource scheduling from fully-preemptable resources to non-preemptable resources.

There are a variety of limited-preemptive scheduling. A periodic or sporadic task set may allow preemption of any running task only if the task has been executing for at least a specific number of time units. An aperiodic task set may have its scheduling decision updated only at specific time points separated from each adjacent ones by a fixed time interval. The common theme in real-time limited preemptive schedulability analysis is the presence of a delay bound. Schedulability by common schedulers like Earliest Deadline First (EDF) or Fixed Priority (FP) is preserved in the limited-preemptive scenario if all task instances finish their execution ahead of their respective deadlines by at least a certain number of time units equal to the delay bound.

The delay bound concept is the same as the one in Bounded Delay Resource Partition (BDRP), first introduced in chapter 1 for fully-preemptable resources. This chapter generalizes the concept by adapting it to non-preemptive scheduling and limited-preemptive scheduling as well. We looked at a specific case study of VMware ESX server as an example of aperiodic limited-preemptive scheduling.

Together with the previous chapters on fully-preemptive scheduling (chapter 1) and non-preemptive scheduling (chapters 2 and 3, this discussion of limited-preemptive scheduling completes the real-time scheduling picture of all time-shared resources. With all these pieces in place, they could be composed together for scheduling mixed-type resources in the same system.

Chapter 5

The x86 Hierarchical Hypervisor

5.1 Introduction

5.1.1 Definition

A *Virtual Machine* (VM) is an abstraction of an execution environment where system and/or application programs can be run. The piece of software that provides this interface is called a *Virtual Machine Monitor* (VM Monitor) or *hypervisor*. There are many flavors of VM Monitors: some present a different machine interface from that of the host (e.g. the Java Virtual Machine (JVM) [31]), others provide an identical machine interface (e.g. VMware Workstation or ESX server).

Just as an OS abstracts the underlying hardware details to provide a software interface for simultaneously running one or more application programs, a hypervisor virtualizes the underlying hardware interface to provide (virtual) instances of the hardware interface to simultaneously run one or more OSes. Inside a hypervisor, each OS is run in a separate VM instance. In theory, a VM instance should be able to run another copy of the hypervisor, resulting in *recursive virtualization*. We call a hypervisor participating in recursive virtualization a *hierarchical hypervisor*. In theory, there could be an arbitrary number of hierarchical hypervisors participating in recursive virtualization between the actual hardware and the ultimate guest OS.

We call the hypervisor that is the closest to the hardware the bottommost level (level 1), and the one that is the closest to the OS the topmost level (level n). For different OSes in the same system, the topmost level may reside at a numerically different level number. From the point of view of level i, level $i - 1$ is the parent level, level $i + 1$ is the next level or child level, all successive parents levels are collectively called ancestor levels, and all successive children levels the descendant levels.

5.1.2 Related Works

Machine virtualization has been studied for a long time [23]. Popek and Goldberg [42] listed separate criteria for a machine architecture to be virtualizable and recursively virtualizable at the system level. Specifically, the AMD/Intel x86 CPU architecture has been shown to be not virtualizable [44] according to the requirements. Despite that, a number of techniques have been used in various successful attempts to virtualize the x86 architecture, notably by VMware and Xen. VMware also has begun to support running VMware ESX server or Workstation inside another VMware ESX server or Workstation, but this is limited to only one level of nesting. We explore arbitrarily nested virtualization in this dissertation.

The problem of recursive virtualization has been tackled in various ways before. Some adapted the machine-OS interface [30] [15] [25] while others tackled the OS-application interface [19] [6] to make way for recursive virtualization. In this dissertation, we base our hypervisor on unmodified x86 hardware, taking an unmodified OS (e.g. Linux, Microsoft Windows) or “bare-metal” hypervisor as our guest.

Both AMD [4] and Intel [27] introduced hardware support for virtualization. Although the present hardware only supports a single level of virtualization, we show in this dissertation that they are extensible to recursive virtualization. Adams & Agesen published an analysis [1] comparing performance of single-level virtualization with and without hardware support. We do the analysis for multi-level virtualization with and without hardware support.

5.2 Why Recursive Virtualization?

There are a number of scenarios for which we might need recursive virtualization.

5.2.1 Debugging and Upgrading to New Hypervisor

A hypervisor could be used during the development of an OS to probe into the machine state, capture and restore the machine state, etc. Similarly, it is also desirable to have the aid of such a hypervisor in the development of new hypervisors. A hypervisor to debug another hypervisor naturally means recursive virtualization.

When a hypervisor is upgraded, it is desirable to keep both the old and new versions up and running for an extended period of time while guests are migrated, so as to make sure that the new hypervisor performs correctly, or at least as well as the old one. Recursive virtualization is needed if two versions of hypervisors must be up and running simultaneously on a single host for comparison.

5.2.2 Testing Hypervisor Management Software

A specific example, encountered at VMware, is the need to test hypervisor management software at a large scale. Large numbers of hypervisors must be managed. Without recursive virtualization, this requires an expensive setup with as many hosts as hypervisors. With recursive virtualization, significant hardware cost savings can be achieved by running multiple copies of VMware ESX servers within another VMware ESX server or Workstation.

5.2.3 Hardware Feature Prototyping

The introduction of new hardware features is currently prototyped in simulators, which are often slow, and do not simulate all hardware features (e.g. a simulator for the CPU may not simulate cache effects and hence ignores instructions like `WBINVD` that invalidate the cache). If a new hardware feature is prototyped in a hypervisor layer, real systems could be tested faster and the performance results obtained may be more representative of the real system. If a hardware prototyping hypervisor is able to run all present system programs including a hypervisor, it itself has to support recursive virtualization.

Hardware features that can be prototyped this way include but are not limited to new instructions, more memory or processors, and enhancements to I/O controller units.

5.2.4 GENI Net

The Global Environment for Network Innovations (GENI) [40] is a major planned initiative of the US National Science Foundation to build open, large-scale, realistic experimental facility for evaluating new network architecture. One of the requirements of GENI is its sliceability, i.e. GENI must be able to be shared among many researches running different experiments. Virtualization is at the heart of such provision. Recursive virtualization allows for even greater flexibility (some experiments are guaranteed greater access to high-cost resources) and better isolation (experiments shall not interfere with each other).

5.2.5 Real-Time Resource Partitioning

In an open system environment, real-time resource partitioning must be performed by local decisions and is therefore recursive by nature. If we can divide resource into real-time capable partitions (see sections 1.2, 4.3 and 4.4), then the division scheme should naturally be able to work recursively. Our next chapter (chapter 6) deals with the real-time aspects of recursive virtualization by seeing if the greatest timing-sensitive overhead, interrupt and exception forwarding, could also be taken into account in recursive virtualization.

5.3 Design Issues on the x86 Architecture

There are two challenges to this research, one is to make the hypervisor real-time capable, and the other is to make the hypervisor hierarchically capable. This section explores how the hierarchical framework could be built with common

PC processors (the x86 architecture). Much of the results here could be extensible to other processors including those in embedded systems.

Before we dive into details of how to recursively virtualize the processor, the memory management, the interrupt/exception/intercept handling and the I/O subsystem, we first look at two flavors of hierarchical hypervisor, namely, the trap-and-emulate paradigm and the paravirtualization paradigm.

5.3.1 Trap-and-Emulate vs. Paravirtualization

There are many ways a hypervisor could be designed. Some take a trap-and-emulate approach, others take a paravirtualization approach. The trap-and-emulate approach is gradually replacing paravirtualization as hardware assisted virtualization becomes more and more common. It allows the hypervisor to interoperate with other hypervisors (developed by others or even developed in the future).

In designing our algorithm for forwarding interrupts and exceptions, we assume the trap-and-emulate approach. The paravirtual approach simply requires some centralized bookkeeping to directly deliver the interrupts or exceptions to the correct guest hypervisor. The problem is harder in the trap-and-emulate approach because of the assumptions we make:

- The hypervisor has no way to tell whether it is running inside another hypervisor or on top of real hardware (except possibly by timing analysis with the aid of an external time source). Guests do not have any communication with the parent hypervisor.

- The hypervisor does not assume anything about its guests. It does not even know whether its guests are OSes or hypervisors.

5.3.2 Processor Virtualization

The x86 architecture provides 4 rings of privileges. The OS kernel resides in ring 0 which is the most privileged, while application programs run in ring 3 which is the least privileged. Both AMD and Intel removed support for non-zero segment base in their 64-bit mode. Intel further removed support for segment limit checking in 64-bit mode. Thus the only protection we could get is from paging. However, paging does not distinguish between rings 0, 1 and 2 as they are collectively treated as kernel mode. This is known as the *ring compression* problem. For this reason and for the sake of portability, we choose to stick to the use of only ring 0 (kernel mode) and ring 3 (user mode) for all levels of the hypervisor in our design.

The challenge is to properly assign processor privilege to all levels of hypervisor, the OS and application program and can still achieve isolation between them. The hypervisor does this by properly shadowing the Global Descriptor Tables (GDT) and Local Descriptor Tables (LDT) of its guests.

5.3.2.1 Time Multiplexing

Real-time properties of each VM partition is guaranteed by adopting the Bounded Delay Resource Partition (BDRP) scheme from section 1.2 because CPU is a fully preemptable resource. This scheme allows the CPU resource to be recursively virtualizable in the real-time systems context.

The same real-time recursively virtualizable scheme has been shown in sections 4.3.2 and 4.4 to be extensible to non-preemptive scheduling and limited-preemptive aperiodic scheduling.

5.3.2.2 Binary Translation

Some x86 instructions have non-faulting access to privileged machine states, others may incur excessive performance penalty to the hierarchical hypervisor while propagating exceptions up and down the hierarchy. These instructions need to be re-written with online binary translator for safe and efficient execution.

The binary translator needs to be just-in-time and translates only those guest instructions that are about to be executed. Due to the complex control flow structure in the x86 architecture, we cannot reliably tell ahead of time which instructions will never get executed, and we cannot force control flow to always respect instruction boundaries and never jump into the middle of an instruction. An online just-in-time binary translator solves this issue by translating only those instructions that are actually needed.

Following this line of thought, when we reach a control flow instruction, we cannot predict which branch the program execution would take. The binary translator stops translation and resumes guest execution of translated instruction once we reach a control flow instruction in the original guest instruction stream.

Binary translation is applied only to guest code that is intended to be run in kernel mode. Application programs do not have direct access to the privileged states, therefore could be allowed to execute directly without translation.

Problem arises with a binary translating of the binary translator. A hypervisor cannot distinguish between the binary translator code and the translated code of its guest. So from the point of view of a hypervisor, a guest doing binary translation is a self-modifying code. Binary translating a self-modifying code incurs heavy performance penalty. It is not clear which performance penalty dominates, hierarchically binary translating self-modifying code, or propagating an exception up and down the hierarchy. We will look at exception propagation across the hierarchy in chapter 6.

5.3.3 Memory Management

Since non-zero segment base addresses is no longer supported in 64-bit mode, and that paging is mandatory in 64-bit mode, much of the memory management in hypervisor is done through paging. There is a separate shadow page table per guest, which either supersets or is different from the page table the hypervisor uses when it is running itself.

5.3.3.1 Shadow GDT/LDT (for implementation without hardware support)

Address translation in x86 architecture starts with segmentation, which translates virtual address into linear address. This is accomplished by taking the segment selector and indexing it into eith GDT or LDT to obtain the base, limit and permission flags for the segment. After limit check and permission check, the base is added to the offset part of the virtual address to arrive at the linear address. Linear address will be fed into paging, which is the next step in the x86 address translation.

For implementation without hardware support, since we choose to push all hypervisors except the bottom-most one in user mode, we have to shadow the GDT and LDT that any guest hypervisor / OS creates by changing all of their descriptor privilege levels (DPL) to ring 3. Hypervisor needs to add at least 2 entries to the shadow GDT, one kernel code and the other kernel data for the hypervisor itself. There must be enough spare entries in the guest GDT at every level in the hierarchy.

The distinction of the same entry being kernel mode in guest GDT/LDT and user mode in shadow GDT/LDT is important for determining which level of hypervisor in the hierarchy should handle an interrupt or exception (see section 6.3.4 for pseudo-code).

From the point of view of a hypervisor, there is one shadow GDT/LDT per guest. It is necessary to switch shadow GDT/LDT for all ancestor levels of hypervisors when a certain hypervisor in the hierarchy decides to switch guest. This is accomplishing by all hypervisors intercepting any execution of `LGDT` instruction at user mode.

5.3.3.2 Shadow Page Tables (for implementation without nested paging)

The next step of address translation in x86 architecture involves paging, which translates the linear address to the physical address. For a virtualized guest, the result of paging is called guest physical address, which needs to be further translated to become the system physical address, also known as the machine address.

When nested paging is not available, in order to control the memory access of guest OS, hypervisor maintains its own page table on behalf of the guest OS.

Since this page table mirrors the contents of the page table that the guest OS builds, it is called the shadow page table. Given a linear address, the hypervisor parses the guest page table to determine the linear to guest physical address translation, at the same time translating each guest physical address of the page translation table entry into its system physical address, and doing the same translation again at the end to obtain the system physical address to be put into the leaf node entry that corresponds to the linear address in the shadow page table for the guest.

The handling of #PF exceptions, which are essential to shadow page table maintenance, requires reverse propagation, and is discussed in section 6.3.4. Although different levels in the hierarchy may have different shadow page table implementation, they should still be inter-operable by virtue of our exception and interrupt handling framework.

The use of shadow page tables ensure that the memory regions allocated to one guest is not reachable by any other guests. When hardware support is not available, since the hypervisor is going to intercept all interrupts and exceptions, at least the entry points of the hypervisor's interrupt and exception handlers should be reachable by the guest, but they should be write-protected. (Being in kernel mode, the hypervisor's handlers are free to change the page tables so as to access the rest of the handler code as well as the hypervisor's protected data structures.) The linear address region associated with memory mapped I/O could be marked as "not-present" so that any read or write access to it becomes a #PF and properly emulated by the hypervisor. When hardware support is available, none of the hypervisor-exclusive memory region need to be visible in the guest linear address space (this makes the hypervisor more secure), because the hypervisor's page table (value of CR3) is automatically swapped in upon #VMEXIT.

5.3.3.3 Shadow Nested Page Tables (for implementation with nested paging)

Nested paging is available on AMD RVI and Intel EPT. Without loss of generality, we will use AMD's terminology for the rest of this section.

When nested paging is available, two levels of page tables exist. The guest page table (pointed to by gCR3) maps linear address into guest physical address, then the nested page table (pointed to by nCR3) maps guest physical address into system physical address. We consider two cases, whether we are running an immediate guest (i.e. code from the next level), or we are running on behalf of our immediate guest (i.e. code from the descendents of next level).

In the first case, we just create a nested page table (nCR3) that maps the guest physical addresses to our allocated range of system physical addresses for the guest, marking all memory-mapped I/O addresses as not-present. For maximum security, the hypervisor's own memory region should not be reachable from any guest physical address in the nested page table.

level-i (L_i) VMRUN		level-j (L_j) VMRUN	simulated L_i VMRUN
		L_k gCR3	L_k gCR3
L_j gCR3	$\rightarrow$	L_j nCR3	combined L_i nCR3
L_i nCR3			

Table 5.1: Combining nested page tables ($i = 1, j > 1, k = j + 1$ for paravirtual hierarchy; $i \geq 1, j = i + 1, k > j$ for trap-and-emulate hierarchy)

In the second case, where we are running on behalf of our immediate guest under nested paging, things are more complicated. Refer to table 5.1, the level i hypervisor is running a certain guest (at level j), and a #VMEXIT occurs due to guest's attempt to execute VMRUN, at which point it replaces its own page table (L_j gCR3) with a nested version (L_j nCR3). Now the correct address translation sequence should be L_k gCR3 → L_j nCR3 → L_i nCR3. We need to combine two of them because hardware facilitates only two levels of nested paging but not three. I am going to explain why we should combine the bottom two and leave the L_k gCR3 alone.

In a paravirtual hierarchy, imagine when the $L_{k(=j+1)}$ guest is going to trigger another VMRUN, its $L_{k(=j+1)}$ gCR3 is going to be replaced. It takes less work to combine something that won't be changed ($L_{i(=1)}$ nCR3 and L_j nCR3) as the hierarchy grows deeper. For trap and emulate hierarchy, the simulated L_i VMRUN would be presented as the L_j VMRUN to the parent level hypervisor as the VMRUN unfolds across the hierarchy. In this case, although it doesn't matter that much which two to combine, it takes a lot less effort when the hypervisor has full knowledge and control to one of the page tables to combine (L_i nCR3, which the hypervisor created itself). In this case, the hypervisor doesn't even have to create L_i nCR3 at all but just knowing the allocated memory base and limit for the current guest as well as the range of memory addresses that correspond to memory-mapped I/O.

Table 5.2 shows the mapping of each level of page tables before and after this combination exercise. The level-k guest page table (L_k gCR3) maps the guest linear address (gV) to guest physical address of level-k (g_kP), and so on, with level-

		mapping		
keep	L_k `gCR3`	gV	$\rightarrow$	g_kP
combine	L_j `nCR3`	g_kP	$\rightarrow$	g_jP
	L_i `nCR3`	g_jP	$\rightarrow$	g_iP
result	L_i `nCR3`	g_kP	$\rightarrow$	g_iP

Table 5.2: Mapping of combined nested page tables ($i = 1, j > 1, k = j + 1$ for paravirtual hierarchy; $i \geq 1, j = i + 1, k > j$ for trap-and-emulate hierarchy; finally, g_1P is the system physical address sP)

1 physical address (g_1P) equivalent to the system physical address (sP). The way combined L_i `nCR3` is created is very much the same as the way a shadow page table is created, except that we are creating the whole page table at once and cannot opt for the virtual TLB alternative.

For each linear address (g_kP), the L_i hypervisor parses the L_j `nCR3` page table, passing the guest physical address (g_jP) of the page translation table entry to the L_i `nCR3` page table before each read to obtain the host physical address (g_iP) of the entry (which is equal to the system physical address if $i = 1$), and then passing the final guest physical address (g_jP) translated into the L_i `nCR3` again to obtain the final linear address to host physical address translation ($g_kP \rightarrow g_iP$). It then creates the combined L_i page table using these translation. This is basically a software implementation of nested page table walk.

If a certain address is marked not-present in either table, it is marked not-present in the combined table. The L_j MTRR values are also parsed and reflected in the corresponding PAT values in the combined page table. When combining the L_j MTRR values, L_j `nCR3` and L_i `nCR3` PAT values, the memory type combining rules set forth by AMD and Intel's documentation need to be observed.

5.3.3.4 ASID Remapping (for implementation with hardware support)

Hypervisor has to maintain a list of unused ASID numbers and a mapping of (guest ID, guest ASID) → ASID. When it encounters a new guest ASID number, it assigns the next unused ASID number and add to the mapping. When guest issues `MOV CR3` or `INVLPGA`, it reassigns all entries corresponding to the guest ID to new unused ASIDs. When there are not enough unused ASID, the hypervisor issues `INVLPGA`, recycles all ASID numbers to the unused list and start again. This ASID remapping ensures correct TLB caching of page translations across the hierarchy and among different guests, while at the same time trying to reduce the number of (costly) TLB flushes.

5.3.4 I/O Subsystem Virtualization

I/O subsystem virtualization consists of I/O access control and I/O scheduling. Access control for CPU processes could be done by properly marking the shadow or nested page table entries of the memory-mapped I/O region and properly setting up the I/O permission maps for individual I/O ports.

Access control for external device cannot be securely implemented without the IOMMU (AMD) or VT-d (Intel). Take AMD as an example, for implementation with hardware support, a Device Exclusion Vector (DEV) could be programmed to achieve limited control but there is no guaranteed enforcement if IOMMU is not involved. IOMMU provides the I/O page table for external devices. In the context of recursive virtualization, the hypervisor needs to virtualize the IOMMU for its guest, which means that it has to do shadow paging for I/O page tables anyway, because nested paging is not available with IOMMU.

I/O scheduling that preserves real-time properties of each VM partition is done by following the scheduling analysis of chapter 3, and in particular the non-preemptive part because I/O subsystem is mostly non-preemptable in nature.

5.4 Conclusion

This chapter gives a brief introduction to recursive virtualization for the x86 architecture. Recursive virtualization is useful in a number of scenarios, for example, when we debug and upgrade to a new hypervisor, when we test the hypervisor management software and when we prototype new hardware features.

We briefly analyzed the recursive aspect of processor virtualization, memory management virtualization and I/O subsystem virtualization for the x86 architecture. For the real-time aspects of these virtualization, CPU is a fully-preemptable resource, memory is a space-partitioned resource, I/O subsystem is a non-preemptable resource. They have been dealt with in the previous chapters 1, 2, 3 and 4.

A key aspect of real-time recursive virtualization has been deliberately left out in this chapter, namely, the recursive forwarding and delivery of interrupt, exceptions and intercepts. This topic directly affects the real-time timeliness of the recursive virtualization, and is the main topic of our next chapter (chapter 6).

Chapter 6

Interrupt and Exception Forwarding in x86 Recursive Virtualization

Virtualization has been a key technology in enhancing interoperability and in making systems more secure. However, the question remains whether virtualization can be used in the context of real-time systems because of efficiency and schedulability issues. This question is even more controversial when recursive virtualization is considered.

In this chapter, we explore one of the biggest challenges of bringing recursive virtualization to the real-time systems community, namely bounding the time for interrupt and exception forwarding across the hierarchy of hypervisors. We analyze the problem and propose non-paravirtualized algorithms in the context of the x86 architecture, both with and without the latest hardware virtualization support. Though the performance is severely limited by the current hardware features, we show that a simple hardware extension could speed up recursive interrupt and exception delivery significantly.

6.1 Introduction

6.1.1 Motivation

Two important features of OS support for real-time applications are predictable, efficient interrupt handling and exception forwarding (in general, event handling). In the context of virtualization, it is especially challenging to run a hypervisor inside another hypervisor (known as *recursive virtualization* or *nested virtualization*). Recursive virtualization is important for future system design (see section 5.2), especially if we can provide real-time guarantees despite of the recursive resource partitioning involved. This chapter concentrates on how to correctly and efficiently support forwarding of interrupts and exceptions in recursive virtualization in the context of the x86 architecture and to provide time bounds. We shall suggest specific hardware support that is needed for an efficient solution after analyzing why a purely software solution will likely fall short.

6.1.2 The Problem

A hypervisor needs to forward suitable interrupts and exceptions to its guest, which could itself be another instance of the hypervisor. In the context of nesting hardware-assisted virtualization, we also need to forward some of the intercepts to the guest hypervisor.

When there are multiple levels of hypervisors in recursive virtualization, we need a correct and efficient algorithm to forward these interrupts, exceptions and intercepts to the hypervisor sitting at the correct level. We assume that each hypervisor is unaware of whether it sits directly on top of hardware, or within another hypervisor. (Please see section 5.3.1 for more details on this assumption.)

The proposed algorithms need to provide performance guarantees suitable for real-time analysis. We also restrict our discussion to the x86 architecture.

6.1.3 Our Contribution

In this chapter, we propose the concept of *forward propagation* and *reverse propagation* for interrupt / exception / intercept delivery in recursive virtualization in the context of the x86 architecture, and formulate distributed software algorithms for both cases that come with and without hardware-assisted virtualization supports. The running time for each proposed algorithm is analyzed to provide a parameterized bound for the worst-case execution time, and the analysis is verified with simulation. Finally, we propose a possible future hardware extension to improve the performance.

6.2 Design Issues

6.2.1 Statically Determined Interrupt and Exception Handling Sequence

In recursive virtualization, we observe that the level which gets to handle an exception or an interrupt is always well defined. Basically, if the exception or interrupt is generated internally by software, it is handled top-down (from higher to lower numerical level number); but if the exception or interrupt is generated externally, it is handled bottom-up (from lower to higher numerical level number).

Specifically in the context of the x86 architecture, all fault-type (including hidden page fault `#PF` in shadow paging), trap-type, abort-type exceptions (except machine check exception `#MC`), all instruction intercepts, all I/O intercepts, all software interrupts (INT) and processor shutdown (triple-fault) are *internal events*; while all non-maskable interrupts (NMI), system management interrupts (SMI), maskable external interrupts (INTR), external processor initialization (INIT), machine check exception `#MC` and processor freeze (FERR) are *external events* [1].

6.2.2 Forward Propagation

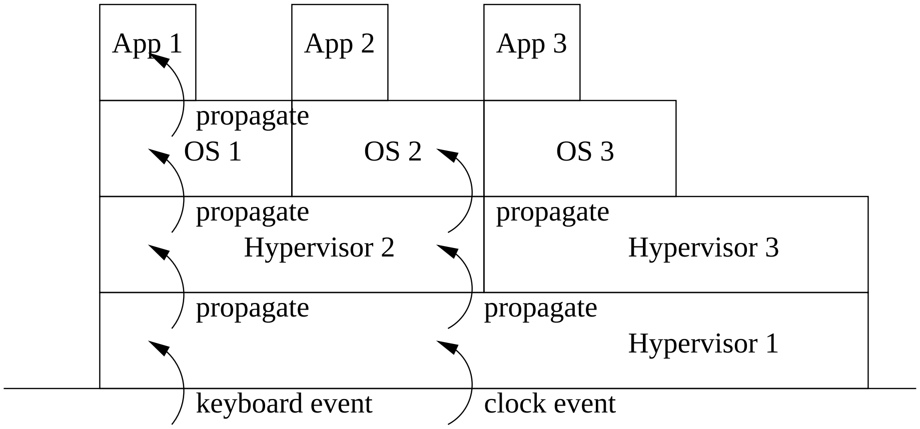


Figure 6.1: Forward propagation

When a hypervisor receives an external event, it either consumes the event itself, or needs to forward the event to the correct guest in the hierarchy. We call this *forward propagation*. Refer to figure 6.1, and suppose the processor is currently assigned to App 2 while the keyboard is assigned to App 1. Now, external timer

[1] An SMI could be caused by internally trapping I/O instructions, or asserted externally. Ideally, we would like it to be handled top-down in the first case, and bottom-up in the second case.

Debug exceptions `#DB` is another special case. When the use of recursive virtualization is to debug a new hypervisor or OS, the bottommost hypervisor may wish to own the exception together with the debug registers, in which case the exception should be handled bottom-up. In all other cases, the hypervisor should leave the debug registers to its guests, and handle the exception top-down.

interrupts should be forwarded to OS 2 while at the same time external keyboard interrupts should be delivered to App 1.

6.2.3 Reverse Propagation

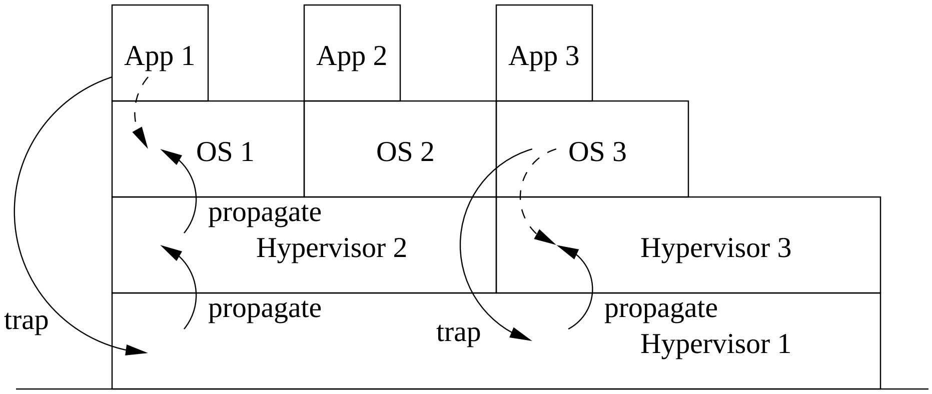


Figure 6.2: Reverse propagation

Internal events should be delivered in a top-down manner. However, without hardware support, such events are always delivered to the bottommost hypervisor first. Thus the hierarchy as a whole needs to simulate a top-down delivery of such events. In figure 6.2, exceptions generated in App 1 should be delivered to OS 1, and those from OS 3 should first be triaged by hypervisor 3, as shown by the dotted arrows. However, real exceptions travel according to the solid arrows, with hypervisors jointly executing a non-paravirtualized algorithm to forward each exception back to its correct level.

Our non-paravirtualization requirement forbids the bottommost hypervisor from intervening and acting as a proxy for all subsequent levels. It also mandates that each hypervisor can make decisions based only on its own state, not the state of its parent or children. Hence we need a "distributed" algorithm for the implementation of top-down delivery. We call this *reverse propagation*.

As each intermediate hypervisor processes an event (both internal and external), it generates more (internal) events that are reverse propagated to the next lower level hypervisor. This pattern continues until the bottommost hypervisor finally gets to process the event.

6.2.4 Performance Measurement Methodology

The performance of each proposed algorithm is first analyzed mathematically and then simulated empirically. Current hardware is not performant enough for real implementation, so we verify the mathematical analysis with simulation. The approximate number of clock cycles spent in interrupt / exception delivery is graphed against the hypervisor level number for each of the proposed algorithms (figures 6.3, 6.4 and 6.5). The mathematical analysis is shown as solid line whereas the empirical measurements from simulation are shown as data points with error bounds.

Both the mathematical analysis and the empirical simulation require some data about the speed of certain hardware instructions (e.g. `IRET`) and events (e.g. `#GP` exception). We obtained some of the numbers from AMD published data [3], and have performed measurements to determine the rest. All measurements are done on a 6-core 2200MHz SVM-enabled AMD Istanbul processor (Opteron 2427). The numbers we obtained are recorded in table 6.1.

We re-purposed a custom OS (FrobOS, that VMware developed for VMM

# of clock cycles	AMD [3]	FrobOS
ALU† *mem reg/imm*	4	
ALU† *reg reg/imm*	1	
BT *mem imm*	4	
BT *mem reg*	7	
CLI	3	
#`GP` exception		120
INC *reg*	1	
IRET	80	80
Jcc/JMP *disp*	1	
LIDT	36	
MOV *mem reg/imm*	3	
MOV *reg reg/imm*	1	
MOV *reg mem*	3	
MOV *reg* `SS`	4	
PUSH *mem/reg/imm*	3	
PUSHF		5
VMLOAD		102
VMSAVE		59
(round-trip) World Switch‡		794

Table 6.1: Characterization for AMD Istanbul (family 10h) † ALU instructions include ADD, AND, CMP, OR, SUB, XOR, etc. ‡ *Round-trip world switch time* is measured as the combined time for a `VMRUN` instruction followed immediately by a `#VMEXIT` event that is triggered by an intercepted `#GP` exception in the first guest instruction.

testing) to measure the number of clock cycles these instructions or events take natively in 64-bit long mode. This hardware characterization test is extended from the same nanobenchmark used by Adams & Agesen [1]. Wherever our measurement overlaps with AMD's (e.g. `IRET` instruction), our results are in agreement with AMD.

In the simulation, the algorithms are rewritten in assembly code and then implemented with each x86 instruction converted to a function that accumulates the number of simulated clock cycles spent according to table 6.1. Instructions or events that require reverse propagation are converted into recursive function calls that follow the actual propagation sequence while accumulating the simulated clock cycles. Randomization is used when control flow depends on external factors.

We run the simulation 65536 times for each algorithm at nesting levels 1 to 5, and plot the minimum, average and maximum number of clock cycles spent, overlaying the mathematical analysis result.

6.3 Implementation without Hardware Support

Even though both AMD and Intel currently offer hardware support for writing a hypervisor, it is still interesting and useful to look at implementation without hardware support for the following reasons. Firstly, the current hardware support is mainly for single-level hypervisors; many software techniques are still needed for deeper nesting. Secondly, the current hardware support represents only one of the many possible things the x86 hardware could do, so we do not wish to restrict our discussion to the status quo.

6.3.1 Processor Operating Modes

When there is no hardware support for virtualization, we want to preferentially protect the hypervisor from its guests rather than the OS from its application programs. So for simplicity, without affecting the validity of our results, we put the hypervisor in ring 0 (kernel mode) and leave its guests including the OS in ring 3 (user mode). (The guest OS could still be protected from the application program by a combination of other techniques like binary translation and proper memory management, but those are outside the scope of this chapter). In recursive virtualization, only the bottommost hypervisor enjoys ring 0 while all other descendant levels reside in ring 3. The hypervisor achieves this by properly shadowing the Global Descriptor Tables (GDT) and Local Descriptor Tables (LDT) of its guests.

6.3.2 Hypervisor IDTs and Shadow IDTs

The hypervisor maintains its own Interrupt Descriptor Table (IDT), which is called the *hypervisor IDT*. In general, it also maintains a *shadow IDT* for each of its guests.

In the shadow IDTs, any exception or interrupt that the hypervisor does not wish to meddle with is directly forwarded to the guest; otherwise the IDT entry points to the hypervisor's own handler.

	64-bit shadow		32-bit shadow		32/64 hyper visor
	retain guest	install new	retain guest	install new	
gate `DPL`	keep	3	keep	3	3
`CS.DPL`	3	0	3	0	0
`CS.C`	0	0	1	1	0

Table 6.2: Shadow IDT and Hypervisor IDT Access Control Bits

Table 6.2 shows the access control bits that should be set for each entry in the shadow IDT and hypervisor IDT. Owing to our simplication in the choice of processor operating modes for the hypervisor and the guest, we pick conforming code segment in 32-bit protected mode and non-conforming code segment in 64-

bit long mode. This decision gives us uniformity in saving and restoring the stack pointer `SS:RSP`. [2] In section 6.3.3, we will explain the use of the code segment's Descriptor Privilege Level `CS.DPL` to decide when we should stop forwarding.

6.3.3 Forward Propagation

The discussion in this section refers to the pseudo-code `forward_propagation()`. The reader is referred to inline sub-routine `propagate_to_guest()` in section 6.3.5.

Forward propagation does not propagate events beyond the level where ex-

[2] When an interrupt occurs in 64-bit long mode, the stack segment and stack pointer `SS:RSP` are always pushed onto the stack, thus interrupt forwarding across the hypervisor hierarchy could be done seamlessly, and we allow non-conforming code segments so that the processor's Current Privilege Level (`CPL`) changes to 0 at the bottommost hypervisor and 3 in all its descendant levels.

When an interrupt goes to a 32-bit protected mode handler, the stack pointer `SS:RSP` may or may not be pushed depending on whether there is a `CPL` change or a switch from the virtual-8086 mode. In recursive virtualization, we want to control exactly when this `SS:RSP` is pushed, otherwise the correct stack frame does not get properly restored after the interrupt is serviced and control returns to the application program.

For guests in virtual-8086 mode, the stack pointer `SS:RSP` and many other segment registers are pushed when the bottommost hypervisor's shadow interrupt handler is called. They remain in the stack until the top-level hypervisor either forwards the event to the virtual-8086 guest's handler, or consumes the event and returns to the virtual-8086 guest. Thus as long as we forward all interrupts in the TSS Interrupt Redirection Bitmap of the virtual-8086 guest, we are fine. (We want virtual mode extension `CR4.VME` $= 1$ because we still need the I/O Redirection Bitmap and `EFLAGS.VIF`)

For guests in protected mode, the stack segment and stack pointer `SS:RSP` are pushed when the bottommost hypervisor's shadow interrupt handler is called (`CPL` changes from 3 to 0). CPU tries to pop them when the event is forwarded to level 2 handler (`CPL` changes from 0 to 3), which is not where it should get popped. When the top-level hypervisor either forwards the event to the guest handler or consumes the event and returns control to the guest, `SS:RSP` are not restored when they should be (`CPL` remains at 3). Here we use conforming code segments in shadow IDT to force the `CPL` to stay the same across the hierarchy. When the bottommost level hypervisor services the interrupt, it raises exception to its own hypervisor IDT, which is the only place where `CPL` changes. Thus when the interrupt is forwarded across the hierarchy, we do not have to worry about saving and restoring `SS:RSP`.

```
01 forward_propagation() {
02   cli
03   if (event is solely for me) {
04     consume the event
05   } else { // guest needs this event
06     if (event needs processing) {
07       preprocess the event
08     }
09     if (saved CS.DPL=0 in shadow GDT) or
10       (shadow RFLAGS.IF=0) {
11       add event to guest pending INTRs
12     } else if (guest in INTR shadow) {
13       set RFLAGS.TF on stack frame
14       add event to guest pending INTRs
15     } else inline propagate_to_guest()
16   }
17   iret
18 }
```

ecution was interrupted, so that the guest does not see any code segment (CS) descriptor that it does not recognize. This is checked by indexing CS from the interrupt stack frame into the shadow GDT of the current guest. If the entry has a Descriptor Privilege Level (DPL) $= 0$, current level code was interrupted, so the guest should wait until the current level finishes execution before it receives this event. In this case, the event is inserted into the (sorted) list of guest pending interrupts according to interrupt priority levels (IPL). These pending interrupts are taken immediately when the current level finishes execution and passes control onto its guest, see the pseudo-code and explanation of propagate_to_guest() in section 6.3.5.

Forward propagation should also observe the provision of interrupt shadow (line 12), where interrupt delivery is temporarily disabled before the completion of the next instruction. If the guest is currently in interrupt shadow (e.g. just after executing the instructions STI or MOV SS), the hypervisor sets the trap flag so that control returns to the hypervisor immediately after the interrupt shadow, at which point the hypervisor can safely propagate the pending interrupts.

6.3.4 Reverse Propagation

```
reverse_propagation() {
  cli
  if (saved CS.DPL=0 in shadow GDT) or
    (saved CS.DPL=0 in guest GDT) {
    lidt hypervisor IDT
    call actual handler
    lidt shadow IDT
  } else inline propagate_to_guest()
  iret
}
```

The discussion in this section refers to the pseudo-code reverse_propagation(). The reader is referred to inline sub-routine propagate_to_guest() in section 6.3.5).

A hypervisor handles the exception or interrupt if the event is triggered by itself (applicable to bottommost level hypervisor only), or if the event occurs at precisely the next level (i.e. in the kernel of its immediate guest). The CS pushed onto the stack should have a DPL $= 3$ in the shadow GDT and DPL $= 0$ in the guest GDT for the latter case. If DPL $= 3$ in both the shadow and guest GDT, the event occurs in one of the descendant levels of the guest, hence should be forwarded to the guest.

Consider a general protection fault (#GP) as an example of an event that requires reverse propagation. When the actual handler at level i is invoked, it is going to generate another #GP which could be (reverse) propagated to level $i - 1$. The use of return-from-interrupt instruction IRET to call the guest handler unwraps tail recursion and eliminates the need for the hypervisor's interrupt handling routines to be *re-entrant*, on the premise that the hypervisor's interrupt routine cannot fault.

6.3.5 Interrupt-Enable Flag RFLAGS.IF Shadowing

If any level other than the bottommost level hypervisor executes an instruction that may change the state of interrupt-enable flag RFLAGS.IF (e.g. CLI/ STI/IRET instructions), it causes #GP exception which is reverse propagated (except for virtual-8086 mode where shadowing of EFLAGS.IF is done in hardware). Upon receiving this exception, the parent hypervisor sets, clears or returns a copy of RFLAGS.IF bit (called shadow RFLAGS.IF) for its guest.

For both forward and reverse propagation, when the event is propagated to its next level guest, the hypervisor sets its own RFLAGS.IF bit to enable external interrupt. Referring to the pseudo-code for propagate_to_guest(), any pending interrupts (from forward propagation, see section 6.3.3) are checked and propagated to the guest at this moment too.

From the point of view of a hypervisor (at any level), the net effect is that no guest can grab a processor forever and prevent the hypervisor scheduler (which hooks onto timer interrupt and I/O events, etc) from running. This guarantees that no guest can steal allocated CPU time from other guests and adversely affect the availability of processor resources to other guests.

```
inline propagate_to_guest() {
  // Sets up the IRET frame for caller
  clear RFLAGS.TF on stack frame
  if (guest in virtual-8086 mode) {
    push saved EFLAGS to saved SS:SP
    push saved CS:IP to saved SS:SP
    saved SP := saved SP - 4
    saved EFLAGS.IF := 1
    saved CS:IP := guest handler's
    if (guest has pending INTRs) {
      saved EFLAGS.VIP := 1
    }
  } else if (64-bit mode) {
    push current SS:RSP
    push current RFLAGS(IF:=1)
    push guest handler's CS:RIP
    forall (guest pending INTRs) {
      push current SS:RSP
      push current RFLAGS(IF:=1)
      push pending handler's CS:RIP
    }
  } else { // protected mode
    push current RFLAGS(IF:=1)
    push guest handler's CS:RIP
    forall (guest pending INTRs) {
      push current RFLAGS(IF:=1)
      push pending handler's CS:RIP
    }
  }
}
```

6.3.6 Running time Analysis

Let us analyze the time it takes to propagate an interrupt or exception. The actual time to service the interrupt or exception does not affect the effectiveness of the propagation.

For the bottommost hypervisor, the running times for both forward propagation and reverse propagation without hardware-assisted virtualization support are dominated by the time it takes to raise each interrupt / exception t_{INT}, and the time it takes for each interrupt return IRET instruction t_{IRET}, assuming for simplicity that all required memory to propagate the interrupt or exception is pinned so we do not have page fault #PF exceptions adding further costs and complexity.

For higher-level hypervisors, some privileged instructions in the propagation itself require reverse propagation, which adds dramatically to the total running time as the number of levels nest deeper.

We make the following simplification according to the worst case scenario:

- Although we disabled interrupt (CLI) for the bottommost hypervisor during propagation, we cannot prevent non-maskable interrupt NMI and system management interrupt SMI from occurring. We are not considering effects from NMI and SMI in this analysis. Interested readers could add them to the final worst-case total cost.
- Except for the bottommost hypervisor, interrupts are actually enabled in hardware during propagation. External interrupts could occur. They are forward propagated and queued as pending at the appropriate level. It adds to the latency of the original propagation but does not increase the total time overhead spent to propagate that many number of interrupts and exceptions. In fact, the worse total time occurs when there are no other guest pending interrupts each time we execute propagate_to_guest(), so that each interrupt has to be propagated by itself and no piggyback optimization can be done. Thus we can omit lines 17-21 and 25-28 of propagate_to_guest() in the analysis. We assume the frequency of external interrupts to be f_{INTR}.

- We calculate only the time it takes to propagate an interrupt to the handler in the appropriate level, and the time to return to the interrupted instruction, i.e. the round-trip time to a null handler. The handler itself may invoke other privileged calls that require reverse propagation, but those are beyond the scope of this analysis.

Let T_n^f and T_n^r be the times it takes to forward propagate and reverse propagate an interrupt / exception to level n respectively. Obviously, $T_1^f \approx T_1^r \approx 2(t_{INT} + t_{IRET})$. The constant 2 is due to the extra logic in lines 5-7 of `reverse_propagation()` (or line 4 in `forward_propagation()`) to get the current privilege level `CPL` correct (see the discussion on conforming code segment in section 6.3.2).

For `propagate_to_guest()`, lines 15 and 23 require reverse propagation. Each time this subroutine is called, the worst case running time is approximately T_{n-1}^r. For `forward_propagation()`, lines 2, 17 require reverse propagation. The worst case occurs when lines 2, 15 and 17 are executed. For `reverse_propagation()`, lines 2, 5, 7 and 9 require reverse propagation. So $T_n^f \approx T_n^r \approx 4T_{n-1}^r = 4^{n-1}2(t_{INT} + t_{IRET})$.

From table 6.1, $t_{INT} + t_{IRET} = 120 + 80 = 200$ cycles. The equation is graphed as solid line in figure 6.3, overlaid with simulation result as data points with uncertainty range.

The mathematical analysis closely matches but slightly underestimates that from the simulation. This is because the mathematical analysis considers only the

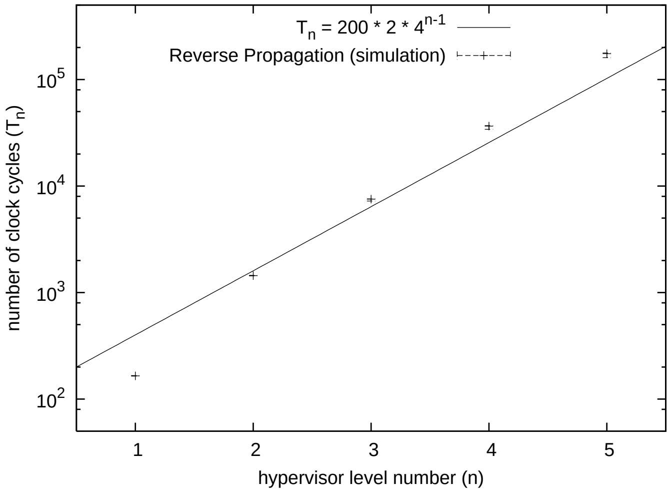


Figure 6.3: Reverse propagation without hardware-assisted virtualization takes exponential time.

key steps that take up the largest number of clock cycles, and ignores others.

6.4 Implementation with Hardware Support for Single-Level Hypervisor

AMD has Secure Virtual Machine (SVM, also known as AMD Pacifica Technology) while Intel has Virtual Machine Extension (VMX, also known as Vanderpool Technology x86, or VT-x). They provide direct hardware support for a single-level hypervisor.

6.4.1 Processor Operating Modes

With hardware-assisted virtualization, the hypervisor and the OS both reside in ring 0 (kernel mode) while the application program is in ring 3 (user mode).

In recursive virtualization, all hypervisors and OSes are in ring 0, and only the application program is in ring 3. Thus, the *ring aliasing* problem is slightly different from the case without hardware supports.

6.4.2 Intercept Handling

Instead of relying mainly on `#GP` exception, a hypervisor using hardware support can specify precisely which events to intercept in the `VMCB` Control Block or VMX Controls in `VMCS`. An intercepted event results in `#VMEXIT`, which is handled by the code immediately after the `VMRUN` (for AMD) or the instruction specified in the `VMCS` when invoking `VMLAUNCH/VMRESUME` instruction (for Intel). The guest is not restarted until the hypervisor executes `VMRUN` or `VMRESUME` again.

We could extend this architecture to recursive virtualization in two ways. In the first approach, paravirtualization would call for an omnipotent bottom-level hypervisor that does all the work and keeps track of all state information for all levels. However, we prefer the second approach, where each hypervisor takes care of only its next level (i.e. immediate guests). According to section 6.2.1, any event is either forward propagated or reverse propagated across the hierarchy (see figure 6.2). Here we present the pseudo-code for reverse propagation. `reverse_propagation_svm()` is for AMD's SVM Architecture. The corresponding one for Intel's VMX Architecture, and the ones for forward propagation under both architectures are very similar.

`gVMCB/gVMCS` is the guest VMCB/VMCS used to run the guest (hyper-

```
reverse_propagation_svm() {
  initialize gVMCB
  clear proxy flag
  while (true) {
    RAX := proxy ?  pVMCB : gVMCB
    VMLOAD
    while (true) {
      restore additional registers
      VMRUN
      save additional registers
      if (handling #VMEXIT is easy) {
        handle #VMEXIT
      } else break
    }
    VMSAVE
    if (proxy) and (guest intercepts this) {
      clear proxy flag
      gVMCB.State := pVMCB.State
    } else if (guest trying to VMRUN) {
      set proxy flag
      gVMCB.rip := gVMCB.rip + 3
      pVMCB.Ctrl := gVMCB.Ctrl
         bitwise-or gVMCB.rax->Ctrl
      pVMCB.State := gVMCB.rax->State
    } else handle other #VMEXIT
  }
}
```

visor or OS), while pVMCB/pVMCS is a proxy VMCB/VMCS for simulating the guest's attempt to VMRUN or VMLAUNCH/VMRESUME. While gVMCB/gVMCS contains all the machine state of the guest, pVMCB/pVMCS contains the machine state of the guest's gVMCB/gVMCS (pointed to by gVMCB.rax upon #VMEXIT when the guest tries to do VMRUN, or given in VMX-Instruction Information Field of VMCS upon #VMEXIT due to guest's attempted execution of VMPTRLD). In pVMCB/pVMCS, we intercept anything that the guest wants to intercept or we ourselves want to intercept. The proxy flag is used to distinguish whether the hypervisor is running the immediate guest or running an image on behalf of the immediate guest.

If the hypervisor is running an immediate guest, it deals with whatever #VMEXIT it catches. But if the hypervisor is running an image on behalf of the immediate guest, it appropriately decides whether to forward the #VMEXIT event to the guest handler or consumes the event itself. If it wants to forward the event to the guest handler, it simply re-starts the guest at the instruction following VMRUN or at the location specified in VMCS when invoking VMLAUNCH/VMRESUME, with the gVMCB/gVMCS updated with the state information from pVMCB/pVMCS.

Since it is mandatory to intercept VMRUN in AMD SVM and VMLAUNCH/VMRESUME in Intel VMX, the hypervisor would only proxy VMRUN or VMLAUNCH/VMRESUME for its immediate guest. Hence it takes care of only the next level in the hierarchy. When a level 3 hypervisor tries to VMRUN or VMLAUNCH/VMRESUME, the level 1 hypervisor intercepts it and forwards it to the level 2 hypervisor. The level 2 hypervisor then sets up a proxy VMRUN or VMLAUNCH/VMRESUME for the level 3 hypervisor, which is again caught by the level 1 hypervisor. And now the level 1 hypervisor sets up a proxy for the level 2 proxy. The level 1 hypervisor has no knowledge that the VMRUN or VMLAUNCH/VMRESUME of the level 2 hypervisor it tries to proxy for is itself a proxy for the level 3 hypervisor!

6.4.3 Running Time Analysis

Similar to the running time analysis of the previous section, we determine the round-trip time to a null handler for the effectiveness of the propagation algorithm, and disregard any time spent inside the actual handlers.

For the bottommost hypervisor, the running time is dominated by the #VMEXIT and VM resume events. These are the events that involve heavy weight world switch between the hypervisor and the guest. Let t_{WS} be the time it takes for each world switch. We have $T_1 = 2t_{WS}$. For all higher levels, the entry point when an intercept is forwarded to the next level hypervisor is the #VMEXIT event at line 10 (immediately following the VMRUN instruction) in reverse_propagation_svm(), and the running time is measured until control loops back to the VMRUN instruction at line 9. The VMSAVE, VMLOAD and VMRUN instructions on lines 15, 6 and 9 respectively require reverse propagation of their own. Thus, $T_n^r = 3T_{n-1}^r = 3^{n-1} \cdot 2t_{WS}$.

From table 6.1, $t_{WS} = 794$ cycles. The equation is graphed as solid line in figure 6.4, overlaid with simulation results as data points with uncertainty range.

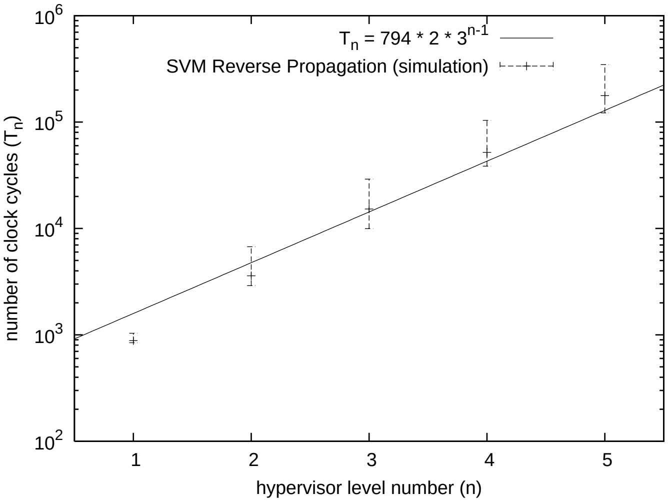


Figure 6.4: Reverse propagation with hardware-assisted virtualization also takes exponential time.

With the current hardware-assisted virtualization support, running time for propagating intercepts in recursive virtualization is still exponential in terms of the level number that the intercept needs to be forwarded to. Though the exponential factor is less than the case without hardware-assisted virtualization, the base case t_{WS} is still prohibitively expensive ($t_{WS} \gg t_{INT} + t_{IRET}$).

The mathematical analysis again closely matches but slightly underestimates the running time than the simulation result. This is because we only consider the key steps that take up the largest number of clock cycles in the mathematical analysis and ignore the rest.

6.5 Possible Hardware Extensions to Support Recursive Intercept Delivery

In this section, we suggest hardware improvement that can drastically bring down the running time for intercept delivery in recursive virtualization.

6.5.1 Hardware Intercept Delivery

The whole interrupt, exception and intercept forwarding could be done better if we adopt a simple hardware algorithm. This algorithm accomplishes correct delivery of intercepts to the hypervisor at the correct level.

6.5.1.1 Ancestor and Descendant Linked Lists

First of all, hardware needs to keep track of the chain of hierarchical hypervisors loaded at any moment. This can be achieved by adding a pointer to the `VM_HOSTSAVE_AREA` (for AMD) and `VMCS` Host-State Area (for Intel) to point back to their parent `VM_HOSTSAVE_AREA` or `VMCS` Host-State Area respectively. Thus, no matter which level in the hierarchy is currently running, this pointer chain links all the ancestor `VM_HOSTSAVE_AREA` or `VMCS` Host-State Area in a linked list. Similarly, from the parent area, hardware can find the next level by keeping their currently running `VMCB` or `VMCS` Guest-State Area in their parent's area.

6.5.1.2 Intercept Redirect Bit

Instead of forcing all intercepts to statically fall into either forward propagation or reverse propagation, we can leave this option open to the hypervisor. We propose that along with each intercept bit that a hypervisor specifies in the `VMCB/VMCS` we define a redirect bit. For backward compatibility, the redirect bit could be specified as follows:

When the intercept bit is not set, the value in the redirect bit is ignored, and the hypervisor won't get this intercept anyway. If the intercept bit is set but the corresponding redirect bit is cleared (which is the default case), then the hypervisor has priority over its guest in intercepting this event, which is what happens with current hardware. If both bits are set, then the processor checks whether the guest is intercepting this event. If it does, then the intercept goes to the guest, otherwise it goes to the hypervisor.

6.5.1.3 The Hardware Algorithm

Thus the processor algorithm to determine which level of hypervisor to deliver an event is unified in pseudocode `intercept_delivery()`. The originating level is defined as the level where the currently executing code (pointed to by the instruction pointer CS:RIP) resides when the intercept occurs.

```
intercept_delivery() {
  $i$ := 0; $j$ := 1
  while ($j <$ originating level) and
        ((level $j$ intercept bit is 0) or
         (level $j$ redirect bit is 1)) {
    if (level $j$ intercepts) {
      $i$ := $j$
    }
    $j$ := $j$ + 1
  }
  if ($j <$ originating level) or ($i$ == 0) {
    deliver the event to level $j$
  } else {
    deliver the event to level $i$
  }
}
```

The algorithm always finds a definite level to deliver the event, so it itself will not generate double fault (#DF) or triple fault (SHUTDOWN) exceptions. A #DF or SHUTDOWN exception occurs only when a certain level has been selected to handle an event, and further faults occur while locating the corresponding handler in that same level.

For exceptions and interrupts (both internal and external), the top-level OS (as well as any hypervisor which is currently not running any guest, during which it behaves like a top-level OS) is poised to handle it anyway, as if it has the intercept bit set and the redirect bit cleared. So if each underlying hypervisor decides either not to handle an event or preferentially let its guest handle the event, then j equals the originating level at the end of the while-loop, and the top-level gets to handle the event.

For new intercepts that come only with the introduction of hypervisor (e.g. instruction intercept), the intercept event is generated only if at least one underlying hypervisor decides to intercept it. Hence j equals originating level implies $i \neq 0$ at the end of the while loop. The originating level will never get to receive the event (which it does not expect to receive).

If this algorithm is implemented in software, it would have the same linear running time (see section 6.5.1.4), but it would violate the isolation requirement between adjacent levels of hypervisors (see section 5.3.1). This requirement is often needed in real-time systems. It is not possible to achieve linear running time in software without paravirtualization because each forwarding step would incur more reverse propagation that avalanche down the hierarchy of hypervisors. Implementing this algorithm as a hardware extension avoids steering the hypervisor into paravirtualization.

6.5.1.4 Running Time Analysis

The running time is still dominated by the world switch cost t_{WS}. From table 6.1, $t_{WS} = 794$ clock cycles. Now, hardware walks the ancestor and descendants linked lists to determine the correct level where intercept should be delivered. This walk is $O(n)$, where n is the numerical value of the originating level. Thus the total running time is $O(n + t_{WS})$. This hardware algorithm is a great performance improvement (figure 6.5) to the exponential running time software solutions, and still keeps the hypervisors isolated from each other.

As the depth of nesting increases, the maximum delivery time increases

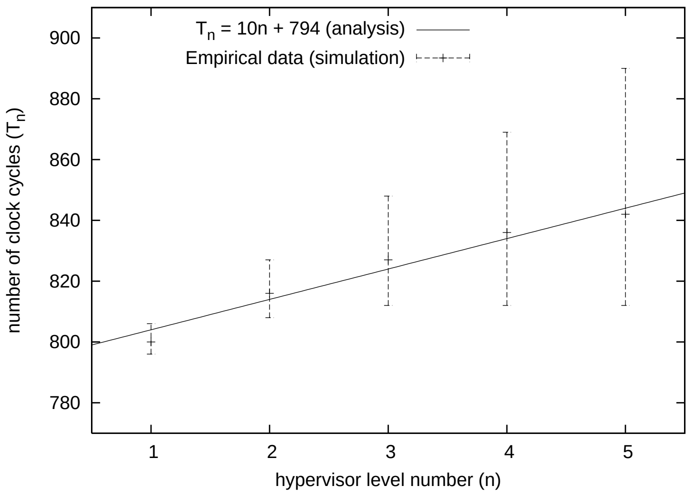


Figure 6.5: Propagation with hardware extension takes linear time.

linearly but the minimum delivery time flattens out. Our mathematical analysis closely matches the average case.

6.5.2 Avoid Avalanche of Intercepts Cascading down the Hierarchy

If the aforementioned hardware solution is not implemented, there are still other ways we could improve performance of intercept delivery in recursive virtualization, albeit to a lesser degree.

In recursive virtualization, intercepts often avalanche down the hierarchy before they are completely serviced, with each handler generating more than one additional intercept to its parent hypervisor. In order to improve performance, either the world switch cost or number of intercepts need to be greatly reduced, or both.

For reducing the world switch cost, we propose the option of lightweight `#VMEXIT`. Adams et. al. [1] found that the current architecture of hardware support for hypervisors assumed too much of the trap-and-emulate paradigm, leaving little room for other approaches like binary translation to be effectively implemented. With the option of lightweight `#VMEXIT`, some hypervisor handlers could run in guest context, thus avoiding costly world switches back and forth.

Alternatively, for reducing the number of instruction intercepts, SVM/VMX could allow a hypervisor to specify a mapping of instruction to sequence of instructions in the `VMCB/VMCS`, so that whenever the processor encounters an instruction defined in the mapping while executing the guest, it executes the mapped sequence of instructions instead. The hypervisor effectively binary translates instructions to avoid excessive instruction intercepts.

6.6 Conclusion

There are many practical uses for recursive virtualization. We would like to reap the same benefits in running real-time guests. Interrupt and exception forwarding is a key issue in building a real-time capable hypervisor. We proposed the concept of *forward propagation* and *reverse propagation*, and formulated a hypervisor-level distributed algorithm for its correct implementation. We have shown in this chapter that its performance bound can be reasonably predicted.

With the current x86 architecture, running time is exponential in the number of nesting levels, whether we adopt the hardware-assisted virtualization or not. This exponential running time may be acceptable because the number of users to a hierarchical hypervisor also decreases exponentially with the number of nesting levels. However, the situation can be improved to linear running time if we assume simple hardware extension as outlined in the last section.

Chapter 7

Conclusion

7.1 Future Work

With all the different types of resources virtualized recursively for real-time workloads using the framework constructed in this dissertation, including the fully preemptable, non-preemptable and limited preemptable resources under the periodic, sporadic or aperiodic task models, the future of real-time recursive virtualization looks promising. From our analysis on the x86 architecture, we are going to generalize our works so that it may apply to broader computer architectures. We are going to review the existing methods and toss a wild guess at future possibilities in this section.

7.1.1 Architectural Constraints to Real-Time Recursive Virtualization

The current computer architecture has a lot of limitation making it difficult to be virtualized, difficult to be virtualized for real-time workloads, and difficult to be recursively virtualized.

Section 1.3 reviewed the literature of known problems that make the x86 virtualization and x86 recursive virtualization difficult. These include, but not limited to, ring aliasing, ring compression, address space compression, non-faulting access to privileged state, etc.

As we have discussed in chapter 6, the biggest deterrent to real-time virtualization in the x86 architecture is the unbounded frequency of occurrence of interrupts and exceptions, and the unbounded amount of time to service each interrupt / exception when it arrives.

The need to maintain backward compatibility makes the x86 instruction set architecture difficult to evolve into one that could seamlessly support real-time recursive virtualization.

7.1.2 Principles for Adapting Any Architecture for Real-Time Recursive Virtualization

Despite all the constraints mentioned, any given architectures could still be used for real-time recursive virtualization, of course, with some hardware and/or software adaptation. We have discussed in chapters 5 and 6 how the x86 architecture could be adapted. The adaptation principles is then generalized.

Generally speaking, adapting a given architecture for real-time recursive virtualization is a three-step process.

1. Make the architecture virtualizable

2. Make the architecture real-time capable

3. Make the architecture recursively virtualizable in a real-time perspective

We are going to look at each of these three steps in turn. Within each step, we need to look at three aspects: the instruction set architecture, the memory management unit (MMU) and the interrupt / exception delivery mechanism.

7.1.2.1 Make the architecture virtualizable

There are a number of ways to virtualize any given architecture. If the architecture meets the requirement as listed by Popek et. al. [42], then a simple trap-and-emulate approach is sufficient. Otherwise, theoretically speaking, full interpretation is always available where all the machine states are stored and emulated in memory. However, this is not performant enough to be of any practical use. So, depending on what is lacking, three common approaches were used to tackle the problem, namely, paravirtualization, binary translation with software MMU, and hardware assisted virtualization. Not all approaches are applicable in all situations.

1. *Paravirtualization.* The hypervisor and guest OS cooperate with each other using predefined protocols. Problematic instructions (like those that involves non-faulting access to privileged state) and/or memory management routines are replaced with system call to the hypervisor.

   This approach is the most performant but it also involves a great amount of work to tweak every guest OS at the source code level. In other words, it has the fewest requirements on hardware capabilities but it is possible only if the guest OS source code is legally available for modification and redistribution. The other pros and cons of paravirtualization has been discussed thoroughly in section 5.3.1.

2. *Binary translation*. It is a clever invention that strikes the balance between the fast speed in native execution and the flexibility of a full interpretation. Guest kernel binary code (instead of source code) is decoded and translated on-the-fly so that problematic instructions and/or memory management routines are translated into sequences of safe instructions, or even callouts to the hypervisor.

   Binary translation makes the fewest assumption about the guest OS so it has the advantage of being able to run unmodified guest OSes. However, it does put some hardware requirements on the architecture to enable the implmentation of binary translation. For example, the binary translator must be able to fully protect itself from inadvertent or even malicious attack from the guest OS. As a counter-example, the 64-bit Intel CPUs do not offer segment limit check, and segment limit check is too expensive to be done in software, therefore binary translation cannot be used on 64-bit Intel CPUs when the guest OS is also 64-bit. Some more discussion of binary translation is available in section 5.3.2.2.

3. *Hardware assisted virtualization*. Last but not least, we could modify the hardware to support virtualization instead of restricting ourselves to software solutions only. Hardware assisted virtualization generally adds a new mode of processor execution called the root mode, which is dedicated to the hypervisor. Guest OSes run in non-root mode. Hardware makes all the necessary distinction between the two modes and acts accordingly.

> Existing hardware assisted virtualization all brings us back to the trap-and-emulate paradigm, although this need not necessarily be the only option. Alternatives include hardware design to support more efficient binary translation or even a combination of the above mentioned methods. Section 6.5.2 has a brief discussion of how hardware assisted virtualization could be made to better support binary translation.

Except for paravirtualization, when we virtualize a given architecture, we always want our hypervisor to be transparent for security reasons, but it does not mean that the presence of the hypervisor is totally undetectable, otherwise a transparent and undetectable viral hypervisor could wreck havoc in the computer system.

If a guest cannot tell, *without external knowledge*, whether it is running directly on hardware or within a hypervisor, then we say that the hypervisor is transparent. It could, however, detect the presence of the hypervisor if it knows, for example, the hardware configuration or CPU speed externally (i.e. not probed by the code, e.g. being told by the end-user), or has access to an external timer. Thus faking the return values of `CPUID` instruction to indicate less features available to the guest does not in itself constitute non-transparency, but an inconsistency in return values when the guest is probing the total amount of available memory by different means, or an incorrect hypervisor behavior in emulating a virtualized hardware are examples of non-transparency.

Hence, as a security issue, a viral hypervisor could be transparent, but not totally undetectable. Transparent hypervisor ensures the correct functioning of all proper guests, be they OSes or hypervisors themselves. An undetectable hypervisor is a theoretical curio but practically impossible in a real machine.

7.1.2.2 Make the architecture real-time capable

In order to make a given architecture real-time capable, we need to identify and classify all the resources into fully preemptable, non-preemptable or limited preemptable types, as in section 1.1. Virtualization of fully preemptive resources follow the summary outlined in section 1.2. Interested readers are also referred to Mok et. al. [37] [36] for a detailed analysis. Virtualization of non-preemptable resources are given in great length in chapters 2 and 3. There are more variation to limited-preemptable resources. Some common types and a VMware case study is provided in chapter 4. One may need to develop similar solutions if the task models are different from what have considered in the chapter.

Basically, a (virtual or physical) resource is real-time capable if its availability is highly predictable. The Bounded Delay Resource Partition (BDRP) approach we adopted in this dissertation provides the predictability and enables some well-known scheduling algorithms to be run unmodified inside such partitions.

Known existing architectures have a big blow to timing predictability of any running software. The incoming rate of external interrupts are unbounded, and the processing time of each interrupt is also unbounded. These interrupt handlers could stack on one another during execution for an arbitrarily long time. In order to make any given architecture real-time capable, we need to enforce some hardware contract on the maximum frequency of incoming external interrupts, and instill some software discipline to keep each interrupt handler quantifiably short.

7.1.2.3 Make the architecture recursively virtualizable in a real-time perspective

There are two parts to this problem. (1) The resource virtualization needs to be capable of doing so recursively; (2) the computer architecture needs to have a predictable and reasonable time bound on all of its activities, particularly the interrupt and exception delivery across the hierarchy of hypervisors.

The Bounded Delay Resource Partition (BDRP) model adopted in this dissertation can be stacked up in recursive virtualization. Theorem 5 and the discussion that follows give the details of how this recursion could be done.

When virtualization becomes recursive, some operations become prohibitively costly. This could cause real-time workloads, which are timing-sensitive, to fail miserably. The most important thing would be the forwarding and delivery of interrupts, exceptions and intercepts across the hierarchy of hypervisors. It is involves every time the hypervisor regains control of the system in the trap-and-emulate paradigm. It is also heavily relied upon by the software MMU when doing binary translation. Chapter 6 specifically deals with how the interrupt / exception delivery could be made predictable.

7.2 Conclusion

Abstract resources are classified into fully-preemptable, non-preemptable or limited-preemptable types and analyzed independently for recursive virtualization with a real-time perspective on the workloads. Specifically, the non-preemptive scheduling is found to suffer anomalies whereby an originally schedulable task set may become unschedulable under reduction in system load. This anomaly is coined as *robustness*. Non-preemptive scheduling in general, and non-preemptive robustness in particular is analyzed in depth, leading to some necessary and sufficient conditions to guarantee non-preemptive robustness.

The Bounded Delay Resource Partition (BDRP) model is borrowed from the fully-preemptable resources and applied to non-preemptable and some variants of limited-preemptable resources with some promising results. The model allows for recursive virtualization with real-time workload because existing common schedulers like Earliest-Deadline-First (EDF) and Fixed Priority (FP) can be applied within such a partition without modification, therefore integrates seamlessly with the whole framework. The application of the model is discussed in a VMware ESX server case study.

With all the theoretical models of real-time recursive virtualization of abstract resources in place, we look at the challenges from a real computer architecture. The x86 architecture is chosen because it is popular and readily available. We looked at various aspects of recursive virtualization on the x86 architecture and then drilled into the one that affects real-time performance the most, namely, the recursive forwarding and delivery of interrupts, exceptions and intercepts across the hierarchy of hypervisors in recursive virtualization. Experiments were done to characterize the timing properties of various schemes, including two software schemes one with and one without the latest hardware assisted virtualization technology, and a hardware scheme proposed for future hardware extension. Finally, we distilled the whole process and discussed how it could be applied to real-time recursively virtualize any computer architectures.

Appendix

Appendix 1

Acronyms

The following acronyms were used throughout this dissertation.

[#DB] Debug Exception

[#DF] Double Fault

[#GP] General Protection Fault

[#MC] Machine Check Exception

[#PF] Page Fault

[#VMEXIT] Virtual Machine Exit Event

[ADD] Addition (Instruction)

[ALU] Arithmetic and Logic Unit

[AMD] Advanced Micro Devices, Inc.

[AND] Bitwise AND (Instruction)

[ASID] Address Space Identifier (AMD)

[BDRP] Bounded Delay Resource Partition

[BT] Binary Translation / Bit Test (Instruction)

[CLI] Clear Interrupt Enable Flag (Instruction)

[CMP] Compare (Instruction)

[CP] Concrete Periodic

[CPL] Current Privilege Level

[CPU] Central Processing Unit

[CR3] Control Register 3, for physical address of top-level page translation table

[CR4] Control Register 4

[CS] Code Segment / Concrete Sporadic

[DEV] Device Exclusion Vector

[DPL] Descriptor Privilege Level

[EDF] Earliest Deadline First

[FERR] Processor Freeze (Event)

[EFLAGS] 32-bit Extended Flags Register

[FP] Fixed Priority

[FrobOS] Frob OS (VMware)

[gCR3] Guest CR3

[gVMCB] Guest VMCB (AMD)

[gVMCS] Guest VMCS (Intel)

[GDT] Global Descriptor Table

[GENI] Global Environment for Network Innovations

[IDT] Interrupt Descriptor Table

[IF] Interrupt Enable Flag (RFLAGS)

[INC] Increment (Instruction)

[INIT] External Processor Initialization (Event)

[INT] Software Interrupt (Instruction)

[INTR] Maskable External Interrupts (Event)

[INVLPG] Invalidate TLB Page (Instruction)

[INVLPGA] Invalidate TLB Page Global Address Space (Instruction, AMD)

[I/O] Input/Output

[IOMMU] Input Output Memory Management Unit

[IPL] Interrupt Priority Level

[IRET] Interrupt Return (Instruction)

[Jcc] Conditional Jump (Instruction)

[JMP] Unconditional Jump (Instruction)

[JVM] Java Virtual Machine

[LDT] Local Descriptor Table

[LGDT] Load Global Descriptor Table (Instruction)

[LIDT] Load Interrupt Descriptor Table (Instruction)

[MMU] Memory Management Unit

[MOV] Move (Instruction)

[MTRR] Memory Type Range Register

[NCP] Non-Concrete Periodic

[nCR3] Nested CR3 Register

[NCS] Non-Concrete Sporadic

[NMI] Non-Maskable Interrupt (Event)

[NPEDF] Non-Preemptive Earliest Deadline First

[NPFP] Non-Preemptive Fixed Priority

[NPFP/RMA] Non-Preemptive Fixed Priority Scheduler with Rate Monotonic Assignment of Priority

[NSF] National Science Foundation

[OR] Bitwise Inclusive OR (Instruction)

[OS] Operating System

[PAT] Page Attribute Table

[PC] Personal Computer

[PCPU] Physical CPU

[PEDF] Preemptive Earliest Deadline First

[PFP] Preemptive Fixed Priority

[PFP/RMA] Preemptive Fixed Priority Scheduler with Rate Monotonic Assignment of Priority

[PUSH] Push to Stack (Instruction)

[PUSHF] Push RFLAGS to Stack (Instruction)

[pVMCB] Proxy VMCB (AMD)

[pVMCS] Proxy VMCS (Intel)

[RAX] 64-bit Accumulator Register

[RFLAGS] 64-bit Flags Register

[RIP] 64-bit Instruction Pointer

[RMA] Rate Monotonic Assignment of Priority

[RSP] 64-bit Stack Pointer

[RVI] Rapid Virtualization Indexing (AMD)

[SMI] System Management Interrupt (Event)

[SS] Stack Segment

[STI] Set Interrupt Enable Flag (Instruction)

[SUB] Subtract (Instruction)

[SVM] Secure Virtual Machine (AMD)

[TF] Trap Flag (RFLAGS)

[TLB] Translation Lookaside Buffer

[TSS] Task State Segment

[VCPU] Virtual CPU

[VIF] Virtual Interrupt Enable Flag (RFLAGS, AMD)

[VM] Virtual Machine

[VMCB] Virtual Machine Control Block (AMD)

[VMCS] Virtual Machine Control Structure (Intel)

[VME] Virtual Mode Extension (CR4, AMD)

[VMLAUNCH] Virtual Machine Launch (Intel)

[VMLOAD] Virtual Machine Load (AMD)

[VMM] Virtual Machine Monitor, i.e. Hypervisor

[VMPTRLD] Virtual Machine Pointer Load

[VMRESUME] Virtual Machine Resume (Intel)

[VMRUN] Virtual Machine Run (AMD)

[VMSAVE] Virtual Machine Save (AMD)

[VMX] Virtual Machine Extension (Intel)

[VT-d] Virtualization Technology for Directed I/O (Intel)

[VT-x] Virtualization Technology x86 (Intel)

[WCET] Worst-Case Execution Time

[XOR] Bitwise Exclusive OR (Instruction)

[ZCP] Zero-Concrete Periodic

[ZCS] Zero-Concrete Sporadic

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, California, USA, October 2006.

[2] *BIOS and Kernel Developer's Guide for AMD NPT Family 10h Processors (Publication number 31116; revision 3.46)*. AMD, March 2010.

[3] *Software Optimization Guide for AMD Family 10h Processors (Publication number 40546; revision 3.11)*. AMD, May 2009.

[4] *AMD64 Architecture Programmer's Manual, Volume 1, 2 and 3 (Publication number 24592, 24593, 24594; revision 3.15)*. AMD, November 2009.

[5] Madhukar Anand and Insup Lee. Robust and sustainable schedulability analysis of embedded software. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 61–77, Tucson, Arizona, USA, June 2008.

[6] Paul Barhem, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating*

System Principles (SOSP), pages 164–177, Bolton Landing, New York, USA, October 2003.

[7] Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, pages 159–168, Rio de Janeiro, Brazil, December 2006.

[8] Sanjoy K. Baruah, Deji Chen, and Aloysius K. Mok. Jitter concerns in periodic task systems. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, San Francisco, California, USA, December 1997.

[9] Gerald Belpaire and Nai-Ting Hsu. Formal properties of recursive virtual machine architectures. In *Proceedings of the 5th ACM Symposium on Operating System Principles (SOSP)*, pages 89–96, Austin, Texas, USA, November 1975.

[10] Scott A. Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, page 396, Cancun, Mexico, December 2003.

[11] Alan Burns and Sanjoy Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):72–94, 2008.

[12] Giorgio C. Buttazzo. Scalable applications for energy-aware processors. In *Proceedings of the 2nd International Conference on Embedded Software (EMSOFT)*, pages 153–165, Grenoble, France, October 2002.

[13] Ya-Shu Chen, Lin-Pin Chang, Tei-Wei Kuo, and Aloysius K. Mok. Real-time task scheduling anomaly: Observations and prevention. In *Proceedings of the 20th ACM Symposium on Applied Computing*, pages 897–898, March 2005.

[14] Ya-Shu Chen, Lin-Pin Chang, Tei-Wei Kuo, and Aloysius K. Mok. An anomaly prevention approach for real-time task scheduling. *Journal of Systems and Software*, 82(1):144–154, 2009.

[15] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[16] Robert I. Davis and Alan Burns. Robust priority assignment for fixed priority real-time systems. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS)*, pages 3–14, Tucson, Arizona, USA, December 2007.

[17] J. S. Deogun, R. M. Kieckhafer, and A. W. Krings. Stability and performance of list scheduling with external process delays. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 15:5–28, 1998.

[18] Renato Figueiredo, Peter A. Dinda, and José Fortes. Resource virtualization renaissance. *IEEE Computer*, pages 28–31, May 2005.

[19] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, Shantanu Goel, and Steven Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–151, Seattle, Washington, USA, October 1996.

[20] Laurent Georges, Paul Mühlethaler, and Nicolas Rivierre. Optimality and non-preemptive real-time scheduling revisited. Research Report 2516, INRIA, April 1995.

[21] Laurent Georges, Paul Mühlethaler, and Nicolas Rivierre. A few results on non-preemptive real time scheduling. Research Report 3926, INRIA, May 2000.

[22] Robert P. Goldberg. Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, Cambridge, Massachusetts, USA, March 1973.

[23] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, June 1974.

[24] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, March 1969.

[25] Peter H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[26] *Intel 64 and IA-32 Architecture Optimization Reference Manual (Order number 248966-016)*. Intel, November 2007.

[27] *IA-32 Intel Architecture Software Developer's Manual, Volume 1, 2A, 2B, 3A and 3B (Order Number 253665-025US, 253666-025US, 253667-025US, 253668-025US and 253669-025US)*. Intel, November 2007.

[28] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th IEEE Real-Time Systems Symposium (RTSS)*, pages 129–139, San Antonio, Texas, USA, December 1991.

[29] Mark H. Klein, John P. Lehoczky, and Ragunathan Rajkumar. Rate monotonic analysis for real-time industrial computing. *IEEE Computer*, pages 24–33, January 1994.

[30] Hugh C. Lauer and David Wyeth. A recursive virtual machine architecture. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 113–116, Cambridge, Massachusetts, USA, March 1973.

[31] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, April 1999. ISBN 0-20-143294-3.

[32] C. L. Liu and James W. Layland. Scheduling algorithms for multi-programming in a hard-real-time environment. *Journal of ACM*, 20(1), January 1973.

[33] Jane W. S. Liu and Rhan Ha. *Efficient Methods of Validating Timing Constraints*, chapter 9, pages 199–224. Advances in Real-Time Systems. Prentice Hall, 1995. ISBN 0-13-083348-7.

[34] Aloysius K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT Department of EECS, MIT/LCS/TR-297, May 1983.

[35] Aloysius K. Mok, Prasanna Amerasinghe, Moyer Chen, Supoj Sutanthavibul, and Kamtorn Tantisirivat. Synthesis of a real-time message processing system with data-driven timing constraints. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 133–143, December 1987.

[36] Aloysius K. Mok and Alex Xiang Feng. Real-time virtual resource: A timely abstraction for embedded systems. In *Proceedings of the 2nd International Conference on Embedded Software (EMSOFT)*, pages 182–196, Grenoble, France, October 2002.

[37] Aloysius K. Mok, Xiang (Alex) Feng, and Deji Chen. Resource partition for real-time systems. In *Proceedings of IEEE Real Time Technology and Applications Symposium (RTAS)*, pages 75–84, Taipei, Taiwan, June 2001.

[38] Aloysius K. Mok and Wing-Chi Poon. Non-preemptive robustness testing is not finite. In *WiP session of IEEE Real-Time Systems Symposium (RTSS)*, Austin, Texas, USA, December 2002.

[39] Aloysius K. Mok and Wing-Chi Poon. Non-preemptive robustness under reduced system load. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS)*, pages 200–209, Miami, Florida, USA, December 2005.

[40] Larry Peterson, Tom Anderson, Dan Blumenthal, Dean Casey, David Clark, Deborah Estrin, Joe Evans, Dipankar Raychaudhuri, Mike Reiter, Jennifer Rexford, Scott Shenker, and John Wroclawski. Geni design principles. *IEEE Computer*, pages 102–105, September 2006.

[41] Wing-Chi Poon and Aloysius K. Mok. Necessary and sufficient conditions for non-preemptive robustness. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Macau, China, August 2010.

[42] G. J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third-generation architectures. *Communications of ACM*, pages 412–421, July 1974.

[43] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution time analysis. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 18:115–128, 2000. Kluwer Academic Publishers.

[44] John Scott Robin. Analyzing the intel pentium's capability to support a secure virtual machine monitor. Master's thesis, Naval Postgraduate School, Monterey, California, USA, September 1999.

[45] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Computer*, pages 39–47, May 2005.

[46] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–265, December 1989.

[47] L. Sha, R. Rajkumar, and L. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computer*, 39(9),

1990.

[48] James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE Computer*, pages 32–38, May 2005.

[49] James E. Smith and Ravi Nair. *Virtual Machines - Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005. ISBN 1-55860-910-5.

[50] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *IEEE Computer*, pages 48–56, May 2005.

[51] Richard T. Wang and James C. Browne. Virtual machine-based simulation of distributed computing and network computing. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS Performance Evaluation Review)*, volume 10(3), pages 154–156, Las Vegas, Nevada, USA, September 1981.

[52] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *6th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, page 328, Hong Kong, China, December 1999.

Index

Vita

Wingchi Poon won Gold Medal in International Olympiad in Informatics (IOI) in 1997. He graduated his Bachelor of Engineering in Computer Engineering with First Class Honour from the University of Hong Kong in 2000, and received Master of Science in Computer Sciences from the University of Texas at Austin in 2001.

He accepted Christ in 2002 and was baptized in 2003. In his spare time, he likes visiting national parks and taking nature photos. He is now working on core virtualization technology in the virtual machine monitor group in VMware, Inc. in Palo Alto, California.

Wingchi Poon has contributed to the following publications:

1. Aloysius K. Mok and **Wing-Chi Poon** “Non-Preemptive Robustness under Reduced System Load” *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS), pp.200-209, Miami, Florida, USA, December 2005*

2. **Wing-Chi Poon** and Aloysius K. Mok “Necessary and Sufficient Conditions for Non-Preemptive Robustness” *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Macau, P.R. China, August 2010*

3. **Wing-Chi Poon** and Aloysius K. Mok “Bounding the Running Time of Interrupt and Exception Forwarding in Recursive Virtualization for the x86 Architecture” *Manuscript to be submitted*

4. **Wing-Chi Poon** and Aloysius K. Mok “Architecture Independent Real-Time Recursive Virtualization” *Manuscript in preparation*

5. **Wing-Chi Poon** and Aloysius K. Mok “Robustness on Bounded Delay Resource Partition” *Manuscript in preparation*

Permanent address: 2123 Oakland Road
San Jose CA 95131

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth’s TeX Program.