2024

A Virtual Time Driven Simulator for DEVS Models

Ronald R. Stempien
Michigan Technological University, rrstempi@mtu.edu

A VIRTUAL TIME DRIVEN SIMULATOR FOR DEVS MODELS

By

Ronald R. Stempien

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2024

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Thesis Advisor:	*Dr. Jean Mayo*
Committee Member:	*Dr. Soner Onder*
Committee Member:	*Dr. Jianhui Yue*
Department Chair:	*Dr. Zhenlin Wang*

Contents

List of Figures

List of Tables

Acknowledgments

I would like to acknowledge and thank the teams present at both ARiA (Applied Research in Acoustics LLC) and MTRI (Michigan Tech Research Institute), who have worked diligently on this project with me.

List of Abbreviations

DEVS	Discrete Event System Specification
KVM	Kernel Virtual Machine
MSR	Model Specific Register
TSC	Timestamp Counter
VCPU	Virtual CPU
VM	Virtual Machine
VMCS	Virtual Machine Control Structure
VMM	Virtual Machine Manager
VMX	Virtual Machine Extensions
VTDEVS	Virtual Time Discrete Event System Specification

Abstract

When simulating a system that includes some software component, simulation authors are faced with the problem of how to appropriately model the software within the simulation. While many formal methods for modeling software exist, in some contexts these may not be appropriate or viable for a given simulation. Instead, simulation authors may model a computer within the simulation, and run the software in question "as is" on the modeled machine. In this work, we introduce a theoretical framework to allow for the use of hardware virtualization technologies as a hardware accelerator for CPU models in Discrete Event System Specification (DEVS) simulations. In addition, we cover the pragmatic issues with existing hypervisors that make them unfit for use in a simulation context, and provide a modification to Linux's Kernel Virtual Machine (KVM) hypervisor that would allow for this hypervisor to be used in a simulation context. Finally, we validate our new simulation via experiment with two separate benchmarks.

Chapter 1

Introduction

In this thesis, we introduce a theoretical framework for using virtualization as a hardware accelerator for executing code in a discrete event simulation. Specifically, we propose an extension of the Discrete Event System Specification (DEVS) [1], called the Virtual Time Driven Discrete Event System Specification (VTDEVS), which allows for the use of a virtual CPU (VCPU) to drive forward the progression of simulation time. The goal of this is to allow for the efficient execution of guest code when modeling software as-is (that is, running the software in the simulation, rather than formally modeling the software as a simulation component). In addition, this thesis covers the pragmatic issues of implementing virtual machines into a simulation environment, and provides a modification to Linux's Kernel Virtual Machine (KVM) that allows for the hypervisor to be used in a simulation context. Lastly, we present

empirical experimental evidence that this system allows for a simulated CPU to execute code in a simulation time context in a way that is comparable to how a physical CPU executes code in a physical time context.

The organization of this thesis is as follows. First, we introduce the problem of how to model software in a simulation, and what use case this approach is appropriate for. Second, we go over background information necessary to understand the details of theory and implementation, including an introduction to virtualization and timing hardware, as well as a brief introduction to DEVS. Third, we detail the theoretical component of this thesis, the VTDEVS extension to DEVS. Fourth, the implementation of the theoretical system is done via a modification to KVM, and used in a preexisting computer model. Fifth, we discuss the capabilities of this implementation through the experimental results of a micro-benchmark. Sixth, we cover related work in the broader area of using virtualization in simulation contexts. Lastly, we conclude by discussing future work that could stem for our results.

1.1 Problem

Simulations that must model some software component face a difficult challenge. Ideally, the software being scrutinized in the simulation would be modeled already, and it would simply be the case of plugging in the extant model of the software into the

simulation. However, this is not always feasible. Models aren't immediately available for many pieces of software, and developing these models in-and-of themselves takes time and effort. Further, any updates to the software itself may then require a rewrite of the software model.

Ideally, it would be great if we had a plug and play solution that would allow for the software to be run in the simulation as-is, without need for any dedicated software model. Instead of modeling the software itself, a simulation could model the computer the software runs on, and "run" the software on the simulated computer as needed in the simulation. This produces a lot of work upfront in the work that is required to produce a complete model of a computer; however, given such a model, it would be trivial to change the software running on the simulated system itself.

If this approach is taken, then there raises the issue of modeling the CPU. It is clear that this is one of the most important components of a computer model, being the part that runs the software we are attempting to model in the simulation. However, it is not clear how to approach modeling this component. While in some simulation contexts there may be trivial solutions, if we shift our focus onto DEVS simulations we run into a few issues.

Discrete event simulations do not rely on any real notion of time. That is to say, the advancement of simulation state is completely detached from the passage of physical time. Rather, the driving force of a discrete event simulation is the titular event.

That does not imply that the simulation has no concept of time, instead time is driven forward by the events generated in the simulation, rather than vice versa. The time perceived by the simulation is called simulation time.

Take, for example, a DEVS simulation (the exact definition of a DEVS simulation is discussed in section 2.3). The model in a DEVS simulation knows when the next event is scheduled to occur, and it is the simulator algorithm's task to advance the simulation state (and thus the simulation time) to that next event; if the event had not existed, then time would not have advanced forward. The great benefit of this approach is that it allows for the computation of the simulation to be completely unrestrained by physical time. A simulation may skip ahead far forward in time with very little actual compute time. Alternatively, if a simulation is incredibly complex, a simulator may spend a significant amount of compute time to simulate a relatively small amount of simulation time.

In a DEVS simulation, the simulation must know when the next event for each component of the simulation is going to occur. For example, when the CPU issues an `out` instruction to a certain port an event is produced, and this event may trigger the external transition function of another component in the simulation. The simulation is aware of some CPU events ahead of time, for example, it may know that an interrupt is scheduled to fire 2 milliseconds from the current simulation time. However, the general problem of knowing when the next event will occur during execution is

undecidable.

Aside from this event scheduling problem, several important implementation problems persist. Foremost is the interaction between the simulation and the simulated CPU. How should the simulation know the state of the simulated CPU, and vice versa? In the case of a DEVS simulation, where the simulator knows when the next event is set to occur, how does the simulation tell the simulated CPU to also proceed to that next event time? Similarly, if the VCPU produces an event unbeknownst to the simulation, how ought the simulation handle this event? Sadly, more thought must be put into the interaction of these two systems than simply "plugging it in".

The next issue is the actual implementation of the CPU in the model. Ideally, we would like for the simulation to be perfectly representative of the modeled system. If we wished to model the CPU to the best of our abilities, we may look towards the extremely granular models offered by microarchitecture simulators like simplescalar [2]. Similarly, tools and languages exist that allow for the creation of these cycle-by-cycle microarchitecture simulators, such as UPFAST with ADL [3]. The drawback of these simulators is that they are extremely slow. Similar speed concerns extend (albeit to a lesser extent) to simple CPU emulators like Bochs [4]. Virtualization, which allows VCPUs to execute guest code at near-native speeds, presents itself as a promising alternative.

The current methods of timekeeping available in many hypervisors make them unfit

to be used as the hypervisor for a hardware-accelerated CPU simulation. The reason for this is a matter of priorities. Hypervisors are typically designed to be used in a virtualization context, which has very different timing priorities compared to a simulation context, as such the developers of these hypervisors provide tools that conform to the virtualization use case.

In a virtualization context, it is generally desired that a guest perceive wall clock time. Wall clock time refers to the physical passage of time in the real world, that is, "the time a clock on the wall would show." There are many reasons this behavior is generally seen as beneficial for a virtual machine. For example, if a virtual machine wishes to use some network protocols on a physical network, it needs to have a correct view of wall clock time, or else these protocols may not work. Think of a DNS record with a set time-to-live. If a virtual machine with an incorrect view of wall clock time handles such a record, it may kill the record early, or perhaps allow an expired record to live longer than desired.

However, in a DEVS simulation context, wall clock time is irrelevant. This is because simulation time advances independently of wall clock time. If simulation time advances one second, we wish for the modeled computer to perceive a second has passed, regardless of how much time it actually took the simulator to calculate that simulation time slice.

Despite this, there is demand to use these hypervisors as accelerators in simulations.

KVM is no different from other hypervisors, there are several deficiencies with its current implementation that prevent its use in some simulation contexts. These deficiencies have been illustrated in [5]. Put simply, there exist two major hurdles that prevent KVM from being used in a simulation context.

First, the host does not have complete control over the value of the Time Stamp Counter (TSC) that is presented to the guest. The default behavior of the TSC has been described in section 2.2.2. While the host can set a TSC value, this value increments on its own accord relative to the advancement of the host's TSC. If the simulation sets a TSC value while the VCPU is paused and the host is in control, there is no guarantee that when the guest is resumed and eventually queries the value of the TSC that (A) the TSC is the same value as what the simulation set it to some time prior and (B) that value set is still a valid view of simulated time.

Second, there is no efficient or easy way to schedule, measure, or interrupt the execution of the VCPU in KVM. This leaves simulation developers relying upon thread time and sending thread signals to measure and manage the execution of a VCPU. On the part of measurement, thread time is not an accurate account of how long a VCPU had actually been executing guest code. This is because the thread time takes into account both time spent in host user space (for example, setting up the VCPU to run with the `KVM_RUN` ioctl) and time spent in the host kernel. On the part of managing VCPU execution, signals are processed at the leisure of the Linux kernel,

and there is no guarantee that it will deliver these to the recipient thread in a timely manner. This can result in a VCPU being scheduled for much longer than desired.

1.2 Limitations

There is no such thing as a "one size fits all" simulation, of course, and this limitation holds true for the approach given in this thesis. When modeling a system and running a simulation on a model, a modeler must make choices and compromises in what to model and what to abstract. These choices depend on what the goal of running the simulation is, what knowledge are we, as simulation runners, seeking to divine. Such compromises must be made in simulations that involve computers and software.

Take, for example, researchers and designers of CPU microarchitectures. When these researchers run simulations of a microarchitecture on a benchmark program, they are interested in having an extremely accurate picture of how the CPU behaves. They wish to know about every cache miss, every branch prediction, et cetera. A cycle by cycle simulation of a CPU is thus apt for this problem, as it will make clear the behaviors the researcher wishes to observe. However, such simulations are excruciatingly slow in the context of many other simulation use cases.

Contrast this to the approach of creating a bespoke software model, where the hardware is mostly (if not entirely) abstracted away. Many languages exist for creating formal models of software, such as PROMELA [6] and Alloy [7], as well as allowing a modeler to specify requirements the software must satisfy. These languages typically come with a model checker that allows a model developer to see if the model as specified satisfies the requirements. Model checkers and the modeling languages associated to them are incredibly powerful tools with strong theoretical backings that allow developers to easily reason about their software. Similar models can be constructed for DEVS simulations. Maintaining a model of a piece of software can be a massive commitment, however, particularly if said software is complex in functionality. This potentially creates a parallel stream of development, where changes to the software may require a change to the model, which takes time and effort. Such a parallel development stream may be unfavorable for software in active development, which may be constantly changing, thus requiring constant model changes. Alternatively, the software being modeled may not be open source, or poorly documented, thus requiring some reverse engineering to actually understand the semantic properties of the software, which is a whole other Herculean task.

1.3 An Example Use Case

To tackle the problem of congestion on freeways through a city, the city decides to implement a system which limits how often vehicles can enter the freeway via on-ramps. This system will monitor congestion via cameras and other sensors, and send this data to a central computer, which then determines via some algorithm how many cars to let onto the freeway. Before implementing this system, the city decides to create a simulation of the new system, and run empirical experiments to see if the new system will actually decrease congestion. This simulation thus has two parts, a physical part that models the traffic seen on the freeway, and a software part that models the decisions taken by the algorithm. Let us ignore the details of implementing the model of the freeway's traffic, and instead focus on the modeling of the second part, the software that makes the decisions. Ideally, the modeler would be able to look at the software making the decision and abstract it into some formal software model that can then be used as-is in the system's simulation. However, there may be several issues with this. Perhaps the software is a black box that the modeler cannot open to create an accurate model, maybe because the algorithm is proprietary, or that it uses some machine learning techniques that cannot easily be reasoned about. Alternatively, it may be that software is under active development, and any model the modeler develops would swiftly become obsolete, thus requiring a rewrite of the model. Whatever the reason may be, modeling the software formally is infeasible.

With this restriction, the modeler is left with the option to model the computer in the simulation, and run the software on the modeled computer. This allows for the software to be tested in the simulation, without needing to worry about modeling the software itself. Now, the modeler need only worry about modeling the computer. Part of modeling the computer is modeling the CPU, which executes the actual code of the software. Depending on the requirements of the simulation, for example how accurate the simulation must be, the modeler has many possible routes for implementing the CPU model. To maximize the accuracy of the simulation, the modeler may decide to make use of a preexisting CPU cycle-by-cycle simulation, such as those used in microarchitecture research. The great accuracy of these CPU microarchitecture simulators comes with the trade-off of computational complexity, which results in an extremely slow simulation. The modeler may come to the conclusion that this drawback is unacceptable, and that a quicker simulation is worth a loss in accuracy. This is where virtualization becomes an attractive option for modeling the CPU. If the central computer that is making the traffic decisions is of the same architecture as the computer the simulation is running on, the simulation machine can make use of virtualization hardware to run simulated code at native speeds.

The scenario above has all the qualities that make it amenable to the approach suggested in this thesis. First, the simulation contains some software part. This point is important, as a simulation without a software component need not go through this trouble in the first place. Second, this software part cannot easily be modeled

formally. If the software were easily modeled, then it may be more beneficial to model the software directly, and use the model as a component in the simulation. A piece of software may be difficult to model because it is a black box of some sort, or the nature of the software's development cycle makes it difficult to keep a model in parity with the actual software itself. Third, the simulation allows for some margin of error when it comes to the simulation of the software. A simulation may be this way because the software being modeled is non-critical, or if the software is not the main focus of what is being investigated by the simulation runners. Given all these qualities, we may then proceed with the method of software simulation proposed in this thesis.

Chapter 2

Background

A review of several virtualization concepts, simulation concepts, and hardware features are merited before discussing the problem at hand.

2.1 Hardware Features

This project relies upon the use of several specific hardware features unique to some x86 processors.

2.1.1 Time Stamp Counter

The PC platform has a wide variety of time keeping mechanisms. Many of these mechanisms date back to the original IBM PC itself. These include, but are not limited to, the Programmable Interval Timer (PIT), the Advanced Programmable Interrupt Controller (APIC) timer, and the High Precision Event Timer (HPET). However, the most commonly used primary timing mechanism available on most modern x86 CPUs is the Time Stamp Counter (TSC) [8].

The TSC is a Model Specific Register (MSR) present on most modern x86 CPU microarchitectures. In practice, the TSC (MSR index $0x10$) is an unsigned 64-bit integer that increments at a certain rate. Given that the TSC is a Model Specific Register, the fine details of its behavior are very specific to a given microarchitecture. The rate at which the TSC increases may differ from one CPU to another, depending on the microarchitecture. For example, on older microarchitectures, the TSC increased at a rate determined by the CPU clock speed, however modern microarchitectures tend to have the TSC increment at a constant rate divorced from the CPU clock [9].

There are a very limited number of ways that a user can interact with the TSC. These are limited to a few specific instructions and auxiliary MSRs. Perhaps the most basic interaction is reading directly from and writing directly to the MSR itself.

Reading can be performed using the `rdtsc`, `rdtscp`, and `rdmsr` instructions. These instructions populate the `edx:eax` registers with the value of the TSC (or, in the case of `rdmsr`, the MSR indicated by the value of the `ecx` register) at the time they are executed. The `rdtscp` instruction differs from the `rdtsc` instruction, in that it also populates the `ecx` register with a value to represent which specific CPU core the instruction was executed from (set by the `TSC_AUX` MSR). Writing is done solely through the `wrmsr` instruction. One may also use the `rdmsr` and `wrmsr` instructions to read from and write to a number of auxiliary TSC MSRs. First among the auxiliary MSRs is the `TSC_ADJUST` MSR, which is used to increase or decrease the TSC by the value written to it. The `TSC_ADJUST` MSR is also updated whenever the TSC is written to directly, holding the difference between the new and old TSC value. The `TSC_DEADLINE` MSR is used as a timer. When the TSC has passed the value written to the `TSC_DEADLINE` MSR, an interrupt is triggered. Finally, there is the `tpause` instruction, which causes the CPU to sleep (as though in a halted state) until the TSC has surpassed the value specified in the operands (or some other interrupt occurs) [10].

It should be noted that each core of the CPU has its own TSC value. Thus, if the TSC is modified on one core of the CPU, the other cores will remain unaffected [11]. However, in most cases, an operating system works to keep each core's TSC in sync with every other core.

2.2 Virtualization Basics

Virtualization refers to the execution of a computer system, referred to as the guest, within another computer system, called the host. A hypervisor is a process on the host that provides virtualization to one or many guests. In some cases the hypervisor is the base layer of the host, with its main purpose being to provide an environment for virtualization. Alternatively, a hypervisor can be built on top of a traditional operating system, such as Linux, and merely be a process running on said operating system. These are known as type-1 and type-2 hypervisors, respectively.

Typically, virtualization is supported by the hardware of the host computer system, usually built into the microarchitecture of the CPU. This is sometimes referred to as hardware-assisted virtualization. On the x86 platform, hardware-assisted virtualization is supported by Intel's Virtual Machine Extensions (VMX) [12], and AMD's Secure Virtual Machine (SVM) [13]. Hardware-assisted virtualization is one of the primary factors that allows for the efficient execution of guest code in virtual machines, as executing guest code via hardware will be quicker than having to execute guest code via an emulator.

The Linux operating system supplies its own type-2 hypervisor module that allows for user space applications (called virtual machine managers) to take advantage of

hardware-assisted virtualization. This module is called the Kernel-based Virtual Machine (KVM). Being a module of the kernel, KVM executes in kernel space. User space can interact with KVM through a set of `ioctl`-based system calls [14]. KVM supports both VMX and SVM for hardware-assisted virtualization on x86 systems.

The virtual machine manager (VMM) is another layer on the virtualization stack. While there are several differing definitions of what exactly a VMM is, in this paper we use the term to refer to the user space process using a hypervisor to run some virtual code. KVM is a hypervisor, while QEMU (which may use KVM) is a VMM, as an example.

2.2.1 VMX Preemption Timer

The VMX preemption timer is a 32 bit register that ticks down after each change of a given digit of the host's TSC while the guest is running. The timer is set by writing to the `VMX_PREEMPTION_TIMER_VALUE` field of the Virtual Machine Control Structure (VMCS, shared memory with the hypervisor that specifies VCPU behavior), and the digit of the TSC that is tracked is defined by bits 4:0 of the `VMX_MISC` MSR [15] [12]. The digit that the VMX timer keeps track of is dependant on hardware implementation, and cannot be changed by the user. When the VMX preemption timer reaches 0, an interrupt is triggered that forces VMX to exit guest execution and return to

the hypervisor. VMX exits with the reason `EXIT_REASON_PREEMPTION_TIMER`, which is then handled by the kernel before guest reentry.

2.2.2 Virtualized Timekeeping

Many computer systems rely on some notion of the passage of time. Naturally, this requirement extends to those systems running in a virtualized environment. Thus, it is the onus of the hypervisor to provide guest operating systems with some method to tell the time.

There are many ways to approach this problem. Typically, a hypervisor will try to provide a guest with as accurate a view of wall clock time as possible. That is to say, a second for the guest is equal to a second of wall clock time. Such an approach is beneficial for most virtualization use cases. If you are running a Microsoft Windows guest on a Linux host, for example, it is generally desirable that the guest and the host both perceive the same time.

Both Intel and AMD [13] provide support for the virtualization of timekeeping methods in hardware. One way that this is done is via hardware pass-through of the TSC. Here we will focus on how this is implemented on Intel chips. When the guest queries the value of the TSC in some way, say via the `rdtsc` instruction, rather than simply providing the value of the physical register, it will provide a modified value. This

modified value is calculated in hardware by offsetting and scaling the value of the physical register by some value defined in the VCPU's VMCS, a structure in memory that controls several features of a VCPU [16]. The modified virtual TSC (vTSC) that is delivered to the VCPU is defined by the following equation:

$$\text{vTSC} = \text{TSC} \times s + o$$

where o is an unsigned 64-bit integer defined in the `TSC-offset` field of the VCPU's VMCS, and s is a 64 bit fixed point with 48 bits for the fractional part defined in the `TSC-multiplier` field of the VMCS [12]. KVM has built in support for this hardware TSC pass-through feature, and abstracts it automatically for user space virtual machine managers that make use of KVM. By default, KVM does not scale the value of the TSC, and as such, the default value of s is 1. However, KVM does offset the value of the vTSC in order to present the VCPU with its own view of time relative to creation. As such, on creation of the VCPU $o = -TSC$, unless another VCPU has already been created for the VM, in which case the extant VCPU's offset will be used so as to synchronize the view of the TSC between the VCPUs.

2.2.3 Synchronization

As mentioned in the previous section 2.2.2, there exists an offset and scale for each VCPU. While this is consistent with the fact that the TSC on physical hardware is local to each logical processor, it can make synchronizing TSC values across VCPUs difficult. Presumably in order to make the lives of VM developers easier, the KVM developers added a way of synchronizing any particular VCPU's TSC with a VM-wide TSC.

This synchronization procedure is done in the `kvm_synchronize_tsc` function within the kernel. Each VCPU has a pointer to an external `kvm` struct. This `kvm` struct is where the VM-wide TSC is stored. KVM stores a "generation" counter for each VCPU's TSC, and the VM-wide TSC (both represented by a scaling value and offset from the physical TSC). If any two VCPUs are on the same generation, then they are synchronized. When a write to a VCPU's TSC is done by the host (and only the host), KVM will calculate the time delta between the new and the old time. If the time delta is less than a second, or if the value written to the TSC is 0, KVM will interpret the write as a request to synchronize the VCPU's TSC with the VM-wide TSC. As such, the VCPU's TSC offset and scaling value are set to the VM-wide TSC offset and scaling value. Otherwise, if the TSC delta is more than a second, that new TSC value is written to the VCPU's TSC (really the VCPU's TSC offset is changed),

Host TSC	**VM TSC**		**V1 TSC**		**V2 TSC**	
Value	*Value*	*Offset*	*Value*	*Offset*	*Value*	*Offset*
1000	0	-1000	0	-1000	0	-1000
Some time passes.						
1100	100	-1000	100	-1000	100	-1000
VCPU1's TSC is written to with a large delta by the host.						
1100	100	-1000	200	-900	100	-1000
Because the time delta was large, the VM's TSC updates.						
1100	200	-900	200	-900	100	-1000
Some time passes.						
1200	300	-900	300	-900	200	-1000
VCPU2's TSC is written to with a small delta by the host.						
1200	300	-900	300	-900	210	-990
However, this is interpreted to be a request to synchronize.						
1200	300	-900	300	-900	300	-900

Figure 2.1: A simplified example of TSC synchronization for a VM with two VCPUs.

as well as the VM-wide TSC, creating a new TSC generation. An example of this can be found in figure 2.1.

2.3 Discrete Event System Specification

Discrete Event System Specification (DEVS) is a formalism, among many others, for the modeling and simulation of systems. Ziegler et al's literature on modeling and simulation is a good point of reference for the details of the specification [1]. Here we present a slight alteration to the classical DEVS system, known as Parallel DEVS

[17]. A parallel DEVS model is defined by the octuple:

$$M = \langle X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{conf}}, \lambda, ta \rangle$$

where,

† X is a set of input events,

† S is a set of sequential states for the model,

† Y is a set of output events,

† $\delta_{\text{int}} : S \to S$ is the internal event transition function,

† $\delta_{\text{ext}} : Q \times X^b \to S$ is the external event transition function (where X^b is a "bag" of 0 or more input events, which can be thought of as a collection of events),

† $\delta_{\text{conf}} : S \times X^b \to S$ is the confluent event transition function,

† $\lambda : S \to Y^b$ is the output function, and

† $ta : S \to T \cup \{\infty\}$ is the time advance function.

Two sets in the above definition remain unaccounted, Q and T. Time is an implicit component of a DEVS model, and it is represented using the set T. This set is called the *time base*. Model designers are given much leeway in their choice of time base for a model, as there are only a few requirements. Firstly, the set used for the time base

must be a total ordering. That is, there must be some ordering operation, $<$, that for any $a, b, c \in T$ is:

1. transitive, $(a < b \wedge b < c) \rightarrow a < c$;
2. irreflexive, $a \not< a$;
3. antisymmetric, $(a < b \wedge a \neq b) \rightarrow b \not< a$;
4. and total, $(a < b) \vee (b < a) \vee (a = b)$.

Secondly, the set must be an Abelian group. For some operation, $+$, there is an identity element, 0, such that $\forall a \in T, a + 0 = a$. Similarly, each element in the set has an inverse element: $\forall a \in T, \exists -a \in T, a + -a = 0$. The operation must also be order preserving, where $\forall a, b, c \in T, a < b \rightarrow a + c < b + c$.

With the requirements of the T set put forward, one can easily see how these requirements conform to how we perceive time. Finding a set that satisfy these requirements is not difficult. Common choices are the set of real numbers, $\mathbb{R}$, and the set of integers, $\mathbb{Z}$. However, one can think of a number of alternative sets. The only meaningful differentiation between these sets is whether they are continuous or discrete. A time base isomorphic to $\mathbb{R}$ creates a continuous time base, and an time base isomorphic to $\mathbb{Z}$ produces a discrete time base.

The set Q is the set of *total states*, and is a subset of $S \times T$.

$$Q = \{(s, e) : s \in S \wedge 0 \leq e \leq ta(s)\}$$

We can think of this set as a set of intermediate states, where s is a sequential state, and e is the elapsed time that the model has spent in that state. This set is only relevant when an external event triggers the δ_{ext} function.

This definition of a DEVS model is not the classical definition, but rather a common extension to classical DEVS called *Parallel DEVS*. This extension defines the δ_{conf} function, the confluent transition function. It is entirely possible that a model could experience an external and internal event at the same time. This extension gives the modeler explicit control over how a model should handle this edge case [17].

Notably, the DEVS formalism is closed under composition. One may compose several *atomic* DEVS models (that is, as described earlier in this section) together to create a new *composite* DEVS model. Further, composite models may be composed with other composite or atomic models to create a new composite model.

2.3.1 DEVS Simulation

A DEVS model in and of itself is a purely static thing, it does not change state by its own means. That is to say, a model defines the structure and the mechanics of a given system; however, it does not reveal any dynamic behavior. Rather, it is the task of the simulation algorithm to change the state of the model (in accordance to the model's definition), thus revealing the dynamic behaviors we may wish to observe. It is good to think of these as two distinct but symbiotic elements of a greater whole. Just as a recipe and ingredients produce nothing without the hands of the chef, a model reveals little without the mechanical actions of the simulation algorithm.

At a high level, the simulation algorithm is simple, the simulation advances to the soonest next event scheduled by any of the components, which it is aware of via the components' *ta* function. The component with the next event is then updated (based off its δ_{int} function). In this update, however, the component may produce any number of output events (from the λ function), destined for some other components in the model. These recipient components are then updated in accordance to the event generated and the time elapsed since they had last updated, through the δ_{ext} function. At this point, the simulation time is advanced to the time of that soonest event, and the cycle is repeated.

A simplified example of a DEVS simulation would do well to illustrate the high-level actions of the algorithm. Let Alice and Bob be components of a DEVS simulation, and let the current simulation time be Monday. Alice knows her current state, she is currently at work, and she knows the time of her next event, which is on Saturday. Bob similarly knows his current state, he is in class. Bob's next event is scheduled for Tuesday. The simulation thus advances to Tuesday, and Bob's δ_{int} function is called. Bob's event was to call Alice and invite her to the Hockey game on Friday. This updates Bob's internal state, but also produces a new event destined for Alice. After being produced, the event is immediately sent to Alice, as a parameter to her δ_{ext} function, along with the time that has elapsed since her last event. When this function executes, Alice's internal state is updated, and thus her next event time is updated to Friday, when she will be going to the game.

Structurally, the DEVS simulation algorithm is a bit more complicated. The simulation is the composition of 3 different classes of algorithms, atomic simulation algorithms, coordinator algorithms, and a root-coordinator algorithm. These three classes of algorithms are organized in a tree structure, and communicate via message passing.

Atomic simulation algorithms are, as the name alludes, responsible for advancing the state of atomic DEVS components. Each atomic component has a single atomic simulation algorithm associated to it. The atomic simulation algorithm directly modifies the component's state according to the model and any incoming messages sent by the

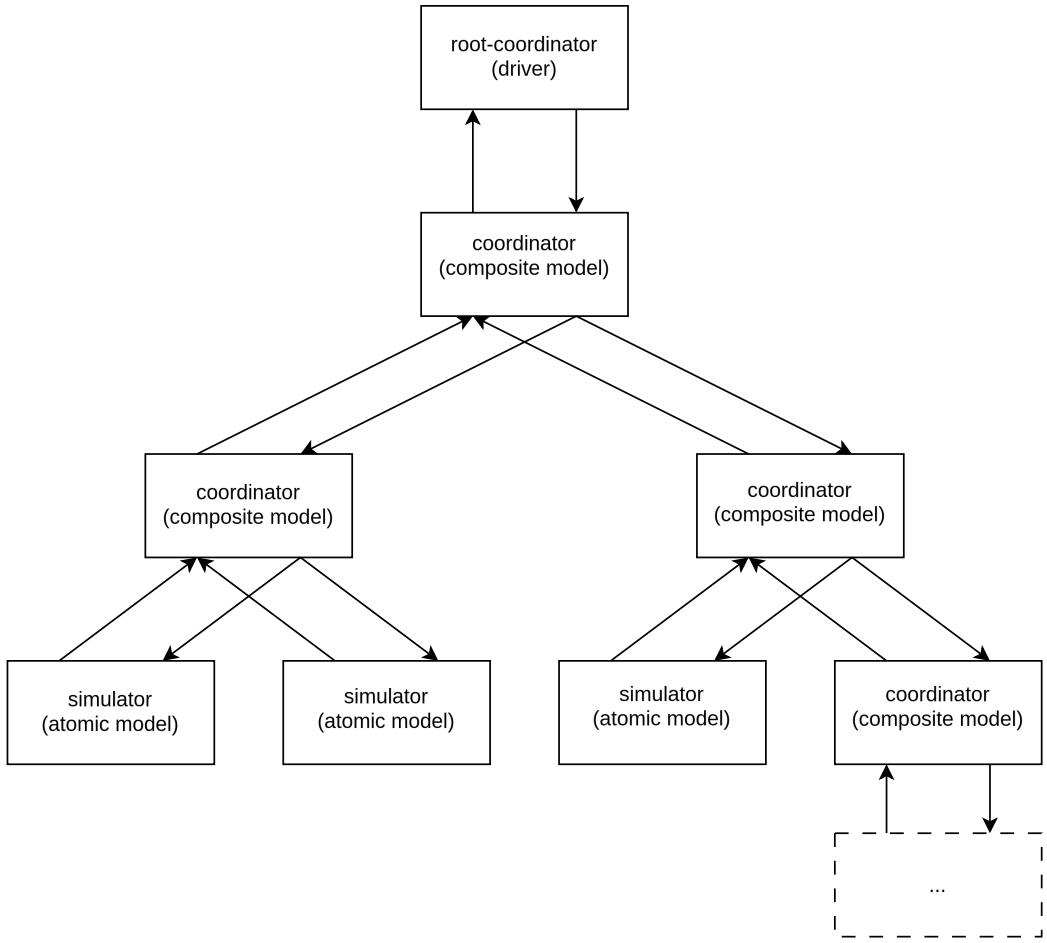


Figure 2.2: Hierarchy of the DEVS simulation algorithm

component's parent. For example, if the component is in state $s \in S$ and receives a message from its parent of the form $(*, t)$ (t being the time the component is to advance to, and $*$ representing that this time is the next internal event), the atomic algorithm will update the component's state to $\delta_{\text{int}}(s)$ and return the message $(\lambda(s), t)$ to the parent. In a DEVS model, each component is aware of only a few basic things. For example, it knows what its current state is (from the set S), and it knows when its next internal event will occur (based off the ta function). However, a component is not aware of when it may receive an external event, this algorithm relies on its parent

Algorithm 1 DEVS root-coordinator

$tn_c :: T$ ▷ Next event time of child, $tn_c = ta(c)$
$t_0 :: T$ ▷ Initial time

$t \leftarrow t_0$ ▷ Initialize time
send (i, t) to c ▷ Send initialization message to child
$t \leftarrow tn_c$
loop
    send $(*, t)$ to c ▷ Advance to the next internal event
    $t \leftarrow tn_c$
end loop

component to send it external events when they appear (via the (x, t) message).

Similarly, coordinator simulation algorithms are associated to composite components. These coordinators do not directly modify the state of any given atomic component. Rather, coordinators are primarily responsible for forwarding messages between their parent node and their child nodes, or from one child node to one of its siblings.

Last of these classes of simulation algorithms is the root-coordinator simulation algorithm. Found at the root of the simulation hierarchy tree, this algorithm is responsible for issuing the messages that drive the simulation forward. This algorithm has only one direct child node, be it a coordinator or atomic simulation algorithm. The workings of the root-coordinator algorithm are simple. Pseudocode of the algorithm can be found in Algorithm 1, as defined in [1].

There are 4 classes of messages used in the simulation algorithm. First are messages of the form (i, t) (i in this context signifies that this is an initialization message, it

does not hold any data). This message class is the initialization message, sent by a parent to its children at the start of a simulation, to reset the children's states to their initial state. Second are messages of the form $(*, t)$. This message class tells a child to advance to its next internal event. Third are messages of the form (x, t). This message class tells its children to advance a certain amount of time and to handle any external event in $x \in X^b$. Lastly, there are messages of the form (y, t). Messages of this form are the only messages transmitted from a child component to its parent, being used to inform the parent component of any output events $y \in Y^b$ generated during the execution of an internal event.

Chapter 3

VTDEVS

3.1 Definition

We created an extension to the DEVS standard, called Virtual Time Driven Discrete Event System Specification, or VTDEVS. The structure of a component model in a VTDEVS system is a nonuple, similar to Parallel DEVS as described in section 2.3:

$$M = \langle X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{conf}}, \lambda, ta, V \rangle$$

where the new symbol V represents a set of VCPUs associated to a component. In this thesis we only consider models where $|V| \leq 1$. See section 7.1 for a discussion on the difficulties of integrating multiple VCPUs into a simulation.

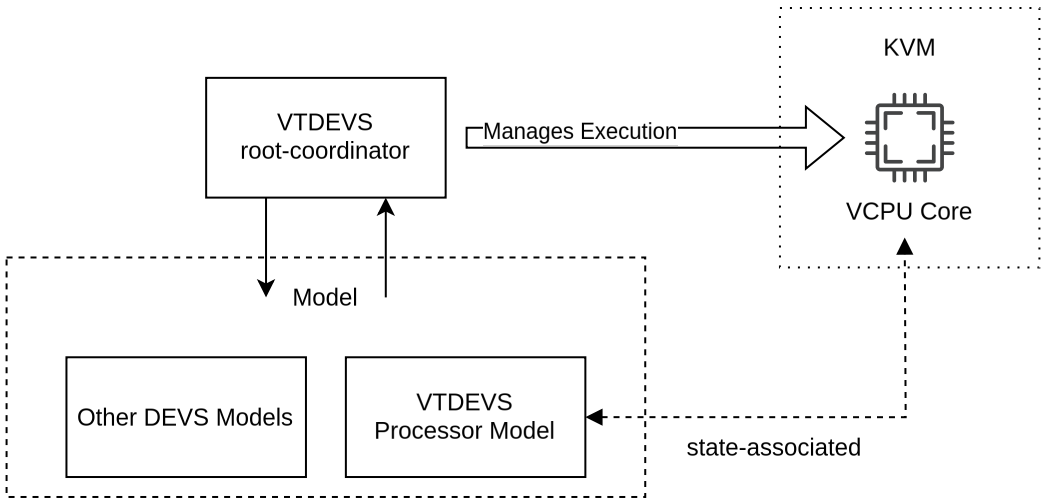


Figure 3.1: The structure of a VTDEVS system

VTDEVS introduces a VCPU to the DEVS model. This VCPU is used to model the state of the CPU in the modeled computer. Figure 3.1 shows the intended structure of a model that uses a VTDEVS component.

In this model, the CPU in a modeled computer is modeled as a VTDEVS component, where other components are modeled using the traditional Parallel DEVS specification. Because VTDEVS maintains all the typical functionality of a normal Parallel DEVS component, a VTDEVS component may be used in a "mixed" DEVS model. That is, a normal DEVS parent interacts with the VTDEVS child in the same way it would interact with a normal DEVS child.

There are implications for what is actually meant by the "state" of a VTDEVS component. The VTDEVS component now has two de facto components that comprise its state - it's classical DEVS state (some $s \in S$), and the state of the VCPU (its

registers, memory, configuration, et cetera). These two parts have influence over each other. The classical DEVS part could result in some state change in the VCPU, for example, changing the value of some register upon receiving an external event, or vice versa, the VCPU could execute the `hlt` instruction, which would change the VTDEVS component's classical state (changing to a not-running state).

3.1.1 Simulation Algorithm

When one looks at the static model described in VTDEVS, it is very similar to a normal Parallel DEVS model. Statically, very little has changed; however, the introduction of a VCPU has consequences for the DEVS simulation algorithm.

To understand why this is an issue, we first must recall the nature of the DEVS simulation algorithm. By the structure of a DEVS model, the DEVS simulator algorithm has perfect knowledge of when the next event will occur in the simulation. Such perfect knowledge is bestowed upon the simulator by the ta function of the model being emulated, as well as having knowledge of when it will inject its own external events into the model. To understand this, let us look at an example. Say we have a DEVS model M, and we are simulating it using the normal DEVS simulation algorithm. The current time in the simulation is $t = 500$. M is in state s, and M's time advance function, ta, reports that the next internal event is scheduled for time $t = 600$. We

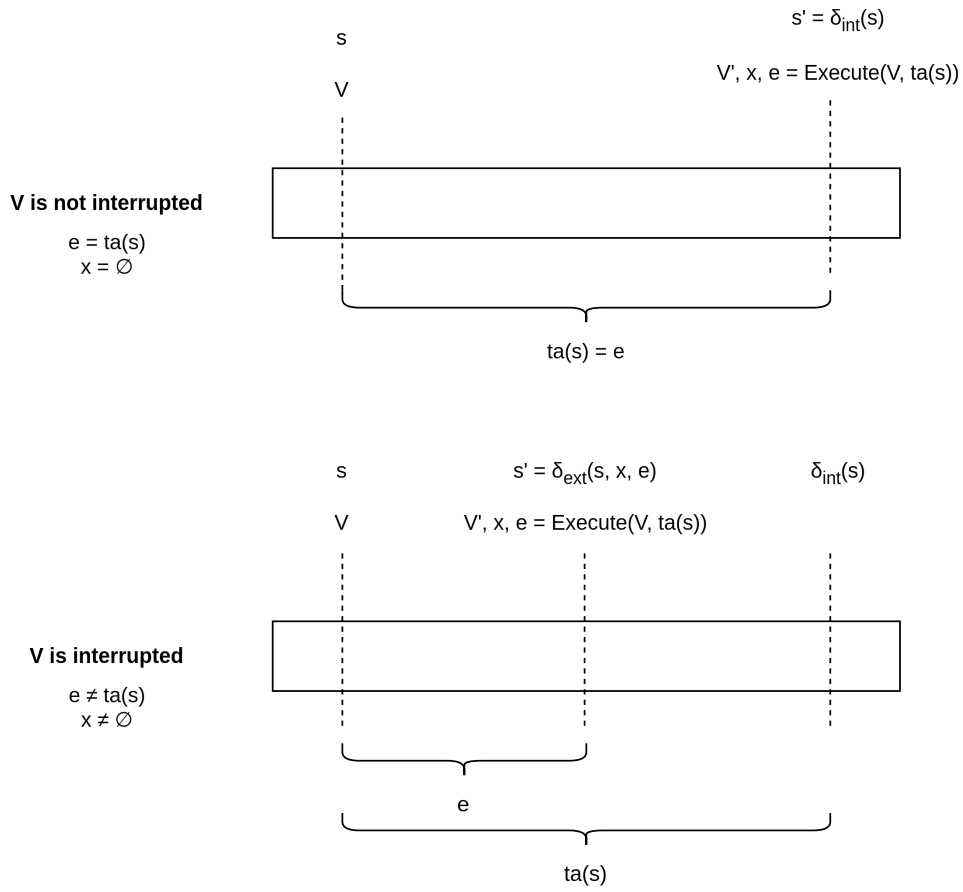


Figure 3.2: Advancing the total state of a VTDEVS model

will assume for now the simulation also has no external events it wishes to inject into the simulation. Given this information, we know that nothing of interest can possibly happen until time $t = 600$. As such, there is no reason to update the model to an intermediate state between now and its next scheduled event. It is possible to update the model to an intermediate state between $t = 500$ and $t = 600$, but nothing is gained from this, as the model is still in sequential state s if we were to do this. The introduction of code execution breaks this perfect knowledge of when the next event will occur. This is due to the undecidable nature of code execution, as discussed

before in section 1.1. If we have some simulation with a VTDEVS component at time $t = 500$, with the next event at time $t = 600$, we can't simply skip ahead to that time, due to the VTDEVS model's imperfect knowledge of its own state. Recall that the VTDEVS component has two de facto states, the classical state s_{CPU} and the VCPU's state. The component has perfect knowledge of the classical state, and as such when the next state transition will occur based solely off this classical state. But, the de facto state of the VTDEVS component also includes the state of the VCPU, which the component does not have complete knowledge of. If we say $ta(s_{CPU}) = 100$, this assumes that the VCPU will not experience some event in those 100 microseconds that results in a state change for the component, for example if it were to issue an IO instruction. As such, to advance the state of the VTDEVS component forward any given amount, we need to execute the VCPU in tandem for that amount of time, in order to catch any unexpected state changes caused as a side effect of code execution.

Given this, the execution of the VCPU needs to be integrated into the DEVS simulation algorithm somewhere. At first glance, this would seem a trivial choice, the atomic simulation algorithm for the CPU component should be responsible for executing the VCPU (that is, when the CPU is told to advance its state, it will execute the VCPU before actually advancing), this does not work due to the lazy nature of the classical DEVS simulation algorithm. In a DEVS simulation, a component's state is only updated when it is needed, as updating a classical component to a partial state for no reason produces no interesting results and is wasted compute. However, the same

cannot be said for a simulation with a VTDEVS component. For sake of an example: take a model with two components, a CPU VTDEVS component and an $OTHER$ classical component. Say $ta(s_{CPU}) = 2000$, and $ta(s_{OTHER}) = 1000$. As such, the simulation algorithm tells $OTHER$ to advance to the next internal transition, $s'_{OTHER} = \delta_{\text{int}}(s_{OTHER})$. Let us also assume $\lambda(s'_{OTHER}) = \emptyset$ and $ta(s'_{OTHER}) = 3000$. As such, there is no reason to do a partial state transition for the CPU component, and as such the VCPU is not executed at all. Now, when the simulation goes to execute the next event, which in this case is the CPU's event, it will tell the CPU model to advance to its next internal state. In doing so, the CPU model will try to execute the VCPU for 2000 microseconds. However, let's say the VCPU was interrupted after only 500 microseconds of execution, changing the classical state of the VTDEVS component, and forcing a new output event x targeting $OTHER$. Here we see the issue, CPU is now in state s'_{CPU} at $t = 500$, with an output event for $OTHER$. As such, $OTHER$ should transition to the new state $\delta_{\text{ext}}((s_{OTHER}, 500), \{x\})$, however $OTHER$ is not in that state, it is in the state s'_{OTHER} in the future at $t = 1000$.

To avoid this aforementioned problem, the solution is simple, we move the execution of the VCPU (and thus the updating of the associated VTDEVS component's non-classical state) to the root-coordinator algorithm. In a classical DEVS simulation algorithm, the state of a component is managed completely by the component itself. That is to say, the atomic model tracks the value of its current state $s \in S$, and this s is only updated by the model itself when it receives a message from its parent telling

Algorithm 2 VTDEVS root-coordinator

$tn_c :: T$ ▷ Next event time of child, $tn_c = ta(c)$
$t_0 :: T$ ▷ Initial time
$V :: VCPU$
Execute $:: VCPU \times T \to P(X) \times T$

$t \leftarrow t_0$ ▷ Initialize time
send (i, t) to c ▷ Initialize model
loop
  $x \leftarrow \emptyset$
  $e \leftarrow tn_c - t$ ▷ e is time to next event
  if V is not halted **then**
    $(x, e) \leftarrow$ EXECUTE(V, e) ▷ Execute V for e time
  end if
  if $x \neq \emptyset \vee t + e < tn_c$ **then** ▷ We were interrupted early
    send $(x, t + e)$ to c ▷ Move the simulation forward partially
    $t \leftarrow t + e$
  else ▷ V executed without interrupt, or was halted
    send $(*, tn_c)$ to c ▷ Advance to the next internal event
    $t \leftarrow tn_c$
  end if
end loop

it to do so. Moving the execution of the VCPU to the root-coordinator destroys this presumption. Now, when the root coordinator wishes to advance the simulation forward any amount, it must first execute the VCPU for that given amount of time. The full algorithm can be seen in Algorithm 2.

If the VCPU is in a halted state (a state where it need not execute code), then the VTDEVS component acts in the same way as a normal Parallel DEVS component, and the simulation algorithm may advance without executing the VCPU. Thus, the simulation at that point has perfect knowledge of next events again, and may move forward at full speed, without unexpected events occurring from code execution.

Chapter 4

Implementation

Having covered the theoretical basis of this thesis, we may move on to the practical implementation. There are two distinct parts to implementing this system in a way that can be used and tested. The first part consists of preparing the hypervisor to be used in a simulation context. This part discusses modifications made to the Linux kernel's KVM module in order for it to be used as an accelerator in a VTDEVS simulation. The second part of the implementation moves from kernel space back to user space, where the simulation lives. Here, we discuss the practical implementation of the VTDEVS root-coordinator algorithm in a pre-existing DEVS simulation. We also discuss changes made to make use of the new kernel features.

4.1 Kernel Changes

As introduced in chapter 1, off-the-shelf virtualization technologies aren't fit to be used in a DEVS simulation context. Our KVM modifications have two primary purposes. The first is TSC isolation, and the second is VCPU management.

4.1.1 TSC Isolation

The aim of TSC isolation is to remove a guest's ability to perceive time on its own, and to be able to substitute this view of wall clock time with some arbitrary value. When we have gained the ability to do this, we can convince the guest to perceive time in any arbitrary way, notably we can present the guest simulation time.

To illustrate our goals in this, let us look at an analogy. Imagine a prisoner locked in a cell. If the prisoner has access to a clock, he can always tell what the time is, he need simply look at the clock. Let us say that the warden of this terrible prison is particularly cruel, and wishes to play with the prisoner's sense of time. Thus, in this imaginary prison there are no clocks, windows, or any other method of telling time available to the prisoner. The only method the prisoner can tell the time is if he goes to the door of his cell and asks the guard what time it is. In this way, the prisoner's

sense of time is captive to whatever the guard tells him. The prisoner might query the guard about the time at 2 AM, yet the guard informs the prisoner that it is 6 AM. At 3 AM the prisoner may again query the guard, and this time the guard responds that it is half past 6 AM. In this way, regardless of the time that had actually passed, the prisoner blindly accepts that only 30 minutes had passed between his two queries. We, as the simulation runners, are the guard, and the guest is the prisoner. If we provide our guest with only methods of time keeping that conform to simulation time, the guest will accept that as the actual time.

To achieve this, we need to remove any external source of time that we, the simulation, do not have direct control of. For most time keeping mechanisms this is quite trivial, due to the nature of virtualization. Mechanisms like the APIC and HPET can simply not be presented to the guest, this can be done by modifying the CPUID presented to the guest to indicate that these chips do not exist on the machine. The PIT is also a trivial case as, being a chip separate from the CPU, it is not virtualized at all, and as such must be emulated in software for a guest to have access to its features. This is typically done in KVM by using the module's built in PIT emulator, created using the `KVM_CREATE_PIT2` ioctl. However, we model the PIT directly as a pure DEVS component under the larger computer model, as such there is no need for KVM to emulate the device. Because our PIT device is modeled directly within the simulation, it is inherently beholden to simulation time.

This leaves us with the TSC. While it would be trivial to simply disable the TSC (for example, via the CPUID method previously mentioned), this leaves the guest operating system without one of the most reliable timekeeping mechanisms available to it. A better approach is to have a fine and arbitrary control over the way the TSC is presented to the guest. If this can be done, then the TSC can be presented to the guest in a way that conforms to simulation time.

Luckily, there exists hardware features that allow hypervisor developers to do just this. Referring back to section 2.1.1, we can see that there are a very limited number of ways in which a program can query the value of the TSC. Two of these methods, the `TSC_DEADLINE` MSR and the `tpause` instruction, reveal the value of the TSC in an indirect way. Neither of these functionalities are particularly important to the execution of Linux guests, so in the project we simply disable these capabilities in the CPUID. This leaves the two main ways of querying the value of the TSC, the `rdmsr` instruction and the `rdtsc` instruction.

4.1.1.1 `rdmsr` Capture

KVM already provides an interface for capturing execution of the `rdmsr` and `wrmsr` instructions via the `KVM_X86_SET_MSR_FILTER` ioctl. This ioctl allows a host to choose what course of action the hypervisor should take upon the execution of a given MSR. Before using this ioctl, we need to enable user space MSR handling by enabling

KVM's KVM_CAP_X86_USER_SPACE_MSR capacity. Specifically, in this capacity we set the KVM_MSR_EXIT_REASON_FILTER to 1, to indicate that filtered MSRs should be redirected as an exit to user space in order to be handled by the VMM (that is, the simulation). [18]

A filter for both reads and writes is set on the TSC's MSR index (0x10). This forces the hypervisor to exit to user space with the exit reason KVM_EXIT_X86_RDMSR or KVM_EXIT_X86_WRMSR, depending on the instruction that was executed. When an exit to user space occurs, depending on the type of exit, the user space application can supply KVM with instructions on what should be done in kernel space to handle the exact exit condition. This communication between kernel space and user space occurs in the shared kvm_run structure. For rdmsr and wrmsr exits, the msr field structure in kvm_run is used to supply (in the case of a read instruction) what value should be populated in the VCPU's edx:eax register, or (in the case of a write instruction) what the VCPU attempted to write to the MSR.

For capturing reads of the TSC using this rdmsr instruction, we simply populate the edx:eax registers with the simulation time at the moment of the instruction's execution. In more complicated simulations, we can supply the registers with a modified version of simulation time. Simulation time can be scaled and offset, similar to what is described in section 2.2.2.

4.1.1.2 `rdtsc` Capture

Of our modifications to the KVM module, the ability to capture and redirect guest execution of the `rdtsc` instruction to the host is the most important. This allows us to provide the guest with any arbitrary TSC value we wish, via the most common method by which the TSC is queried.

Implementation is quite straightforward. First, VMX's `rdtsc` capture feature must be enabled. This is done by setting the 12th bit of the VMCS's `VM-Execution Controls` field to 1, this being the "`rdtsc` exiting" bit. This is done in a handler for a new capacity added to KVM to facilitate this, `KVM_CAP_X86_RDTSC_EXITING`.

When the aforementioned bit has been set, VMX will exit upon the execution of the `rdtsc` instruction with the `EXIT_REASON_RDTSC` exit reason. In KVM we add a handler for this exit that forces an exit to user space. This new KVM exit, `KVM_EXIT_X86_RDTSC`, informs the user space VMM that the `rdtsc` instruction has been executed. Upon this exit occurring, it is the responsibility of userspace to populate the `msr.data` field of the `kvm_run` structure associated to the VCPU with the TSC value the VMM wishes to provide to the guest. When the guest returns from the exit, the value written by the VMM is populated into the the VCPU's `edx:eax` register.

4.1.2 VCPU Management

Our modifications relating to VCPU management revolve around being able to schedule and interrupt the VCPU accurately. In addition, our modifications aim at providing us with a reasonably accurate account of guest execution time.

Issues arise when we reply upon simply using Linux thread time for management of how long a VCPU runs. Firstly, this makes us completely beholden to the whims of the Linux Scheduler. Secondly, we limit ourselves to relying upon pthread signals as our primary method of manually interrupting a guest. These signals are delivered to the guest whenever the kernel gets around to it, which may not be in any timely manner.

4.1.2.1 VMX Preemption Timer

The primary mechanism we use for scheduling and interrupting the VCPU is the VMX preemption timer. To achieve our scheduling goals, we make several changes to how the preemption timer is used within KVM. The goal of these changes is to provide the user space VMM (in our case the DEVS simulation), with the ability to directly control the value of the VMX preemption timer.

In unmodified KVM, the preemption timer has two primary purposes. First, the VMX preemption timer is used by KVM to immediately force an exit upon starting the VCPU. Second, the VMX preemption timer is used to emulate the `TSC_DEADLINE` MSR. To make full use of the VMX preemption timer, we need to be able to disable these two functionalities, in order to make the timer always available to the user space program. Starting with the latter is incredibly simple, the emulation of the `TSC_DEADLINE` MSR is disabled by default on the creation of a VCPU, to enable it one must first enable the `KVM_CAP_TSC_DEADLINE_TIMER` capacity. In addition, the VCPU's CPUID is masked to disable the `TSC_DEADLINE` MSR by default, and one must enable it by setting the 24th bit of the `ecx` register on CPUID leaf 1 to 1 [12]. The `TSC_DEADLINE` MSR emulation feature can thus simply be disabled by never enabling the `KVM_CAP_TSC_DEADLINE_TIMER` capacity. In the case of the former functionality, the use of the VMX preemption timer to force an immediate exit upon entering VMX, the kernel must be modified to resort to using the generic immediate exit functionality for KVM. This is done by chaining the `request_immediate_exit` function pointer to point to the generic implementation function `__kvm_request_immediate_exit`.

VMX by default resets the value of the VMX preemption timer upon guest exit. This functionality is undesirable for our use case. Being able to query the value of the VMX preemption timer at the time of the last exit allows us to calculate how long the guest actually executed for. We, as the VMM that set the value of the VMX preemption timer, know the value the timer started at, if we have the value of the

timer at the last exit, we can take the difference between these two values, and know exactly how long the guest has executed for since our last write to the timer. If this value is automatically reset by VMX upon guest exit, we cannot find this difference, which hinders our ability to know how long the guest actually executed. To prevent this from occurring, we set the 22nd bit of the `VM-Exit Controls` field of the VMCS to 1, the "Save VMX-preemption timer value" bit.

From here, it is simple to add new behavior to trigger an exit to user space upon a `EXIT_REASON_PREEMPTION_TIMER` VMX exit. For the kernel modifications, a new KVM exit, `KVM_EXIT_X86_PREEMPTION_TIMER`, is triggered upon the expiration of the VMX preemption timer. This exit has no required action on the part of the user space VMM, rather its usage is simply to indicate to the VMM that the timer had actually expired.

In addition to the KVM exit, user space must be provided with a means of querying and modifying the value of the VMX preemption timer. As such, two new VCPU ioctls were added. The first is the `KVM_X86_SET_PREEMPTION_TIMER` ioctl, which takes as input a 32bit integer value, and directly writes this value into the VMX preemption timer field of the VMCS. The second is the `KVM_X86_GET_PREEMPTION_TIMER` ioctl, which directly returns the value of the VMCS's VMX preemption timer field.

4.2 Simulator Changes

The other side of implementation is the simulation part, without which we cannot test our theoretical VTDEVS specification, nor our kernel modifications. Rather than producing a wholly new simulation that makes use of DEVS, we chose to base our implementation off a piece of extant software, described in [5]. This simulation (which from here on will be referred to as the DEVS simulator, or the unmodified simulator), provides a simple computer model. It makes use of unmodified KVM to model the CPU. To provide time to the guest, since KVM cannot provide access to the TSC, the simulated computer relies upon a modeled PIT as its clock source. The unmodified simulator attempts to manage VCPU execution by working in time slices. The VCPU is scheduled to execute for a given time slice, with all events occurring during that time slice being injected into the simulation at the end of the time slice. To manage the execution of these time slices, the unmodified simulator uses Linux's `CLOCK_MONOTONIC` to measure how long the guest has been running, and uses pthread signals to interrupt the VCPU when the time slice has expired.

4.2.1 Root-Coordinator Algorithm

In order to produce the desired timing logic, the modified root-coordinator algorithm seen in Algorithm 2 needed to be implemented in our computer simulation. The implementation of the algorithm is similar to the pseudocode given. Primarily, the main implementation details are concerning the hardware mechanisms used to actually implement parts of the algorithm.

The EXECUTE subroutine in the pseudocode is responsible for advancing the state of the VCPU. Implementing this function in the simulation is quite simple. The simulation already knows the time of the next event (tn_c), as well as the current time of the simulation (t). Here we find the time to the next event by taking the difference $tn_c - t$. We then scale this simulation time value into a VMX timer value, this scaling value is explained in section 4.2.2. This VMX timer value we can write directly to the appropriate VMCS field using the new `KVM_X86_SET_PREEMPTION_TIMER` ioctl. At this point, the VMX timer has been set to the desired amount of time we wish the VCPU to run for. We can thus trigger the execution of guest code using the `KVM_RUN` ioctl. This ioctl must eventually return to user space. Because we have the VMX timer saved upon VMX exit, we know we don't need to worry about time spent in kernel space but not spent executing guest code. As such, when the ioctl returns, we can query the value of the VMX preemption timer. The difference between the set

timer value and the ending timer value reveals the amount of time the VCPU ran for in terms of VMX preemption timer ticks. We reverse the scaling process to convert this preemption timer value back into a simulation time value, which is thus the amount of time we must advance the simulation forward. If the KVM exit reason was `KVM_EXIT_X86_PREEMPTION_TIMER`, then we know that the VCPU was able to execute for the entire duration of time to the next simulation event without being interrupted in a way that would require a change of the simulation's classical state. As such, the VTDEVS root-coordinator can advance the simulation forward to the next event time as normal, like the classical root-coordinator. Otherwise, if the VCPU exited with some other KVM exit reason, such as performing MMIO, then there may be some classical-state altering action that needs to occur. In this case, the VCPU will generate and return an event, as well as the amount of time (in simulation ticks) that it had actually executed for. The root-coordinator then advances the simulation time forward by the amount of time returned by the VCPU, as well as injects the generated event into the simulation.

4.2.2 Time Scaling

One practical issue to overcome is the fact that the value of a simulation tick is different from that of a VMX preemption timer tick. Every DEVS simulation has some time base, as described in section 2.3. While in some simulations this time base

may be abstract, in our simulation it is very much a representation of the passage of physical time. In this simulation, we use the unsigned `long` integers as our time base, and we define a single increment in this discrete time base to represent the passage of one microsecond of physical time. We call the number of simulation ticks in a simulated second the simulation tick frequency, which in this case is the number of microseconds in a second, or one million.

If the next event is 1000 simulation ticks (or one millisecond) in the future, then we need to execute the VCPU for one millisecond. As discussed in section 4.1.2.1, we can do this by using the VMX preemption timer. However, we can't simply write 1000 into the VMX preemption timer field of the VMCS because the VMX preemption timer has a different frequency. Per section 2.2.1, the VMX preemption timer ticks at a frequency relative to the TSC's frequency (specifically by inspecting a certain bit of the TSC). The TSC itself has some frequency determined by the hardware.

In converting a simulation time value to a value VMX timer can understand, we have to go through several steps. First, we must multiply the simulation time value by the number of TSC ticks per microsecond (or whatever the simulation time frequency is), resulting in the number of TSC ticks the simulation time represents. From here we divide this value by the number of TSC ticks per VMX preemption time tick, to get the number of VMX ticks the simulation time represents, which may be written to the VMX preemption timer field.

There are 3 scaling coefficients present in our simulation:

1. The VMX preemption timer tick to TSC tick scale, p.
2. The TSC tick to simulation tick scale, t.
3. The simulated CPU scale, c.

Two of these, p and t, are determined by the hardware running the simulation. p is a scaling value introduced by the nature of the VMX preemption timer. As described in section 2.2.1, the VMX preemption timer is designed to tick down every time a certain digit in the TSC is flipped. The particular digit being watched by VMX is defined in the `VMX_MISC` MSR; this value is CPU-specific, and cannot be changed by the user. In this way, our VMX timer is scaled by some power of 2 to the TSC. We thus need to read this value and account for this scaling difference in our simulation.

Modern CPUs, those with a constant TSC, have a set TSC tick speed. The speed of the TSC, in kHz, is calculated by Linux at boot up. This TSC rate is readable from within the kernel, being the `tsc_freq_khz` variable. Unfortunately, this value is not exposed to user space in the unmodified kernel. Third-party kernel modules exist to expose this value to user space [19], which can be used to retrieve this value for use in the simulation.

The last of the scaling values is the simulated CPU scale. CPU scaling is done to make

the CPU faster or slower to the simulation. For example, a CPU scale of 2 means that if the CPU was told to run for 100 microseconds of simulation time, the CPU will actually execute for 200 microseconds, yet only perceive that 100 microseconds have passed. In this way, the computer thinks it has done 200 microseconds of work in 100 microseconds of time, thus perceiving the CPU as being twice as fast as it actually is. Similarly, a CPU scale of 0.5 will have the simulated computer believe that the CPU is twice as slow as it actually is.

Chapter 5

Validation

In this chapter, we evaluate the validity of our implementation via the use of two different benchmarks. We execute both of these benchmarks over three different systems, a DEVS simulation that does not use VTDEVS (as described in section 4.2), our VTDEVS simulation, and hardware (as a control to compare to). The two benchmarks used in this evaluation are exect and Cyclictest. The former is a micro benchmark created specifically for this thesis, and the latter is an industry standard benchmark for evaluating the latency of Linux systems.

An explanation of "validity" in this context is warranted. In this paper, we are currently only interested in the correctness of the simulation, rather than the performance of the simulation. We take for granted that a simulation using virtualization

to execute code will run faster than the same simulation using emulation or a proper CPU simulation. We find this to be a reasonable assumption. Rather than comparing the amount of time it takes to execute a simulation, we are attempting to compare how close a simulation is in dynamic behavior to the real system being modeled.

Validation of a simulation is done via experiment. The modeled system and the physical system are given the same inputs, and produce some outputs. The outputs of these systems may then be processed and compared to each other. Simulation error is the difference between the outputs of the modeled and physical system. If the error measured is beneath some threshold, the tolerance level, we say that the simulation has had its validity confirmed [1]. Validation is not a proof of correctness for the whole modeled system with all possible inputs, rather it is a judgment of "good enough" for some context. For our validation tests, we perform this form of experiment. Our inputs are the benchmarks themselves, and our outputs we are comparing are the results of these benchmarks. However, rather than setting a discrete tolerance level for simulation error, we are simply comparing the two alternatives (a DEVS simulation versus a VTDEVS simulation) to each other.

5.1 Exect Benchmark

The first benchmark we have used to evaluate our implementation is the exect ("*exec*ute for *t*ime") synthetic benchmark. This benchmark was created specifically to evaluate this project. Mechanically, this benchmark is incredibly simple, it measures the number of iterations of a loop that can be performed in a set amount of time. The idea being that we may use this count as a measure of the amount of work that can be done in a set period of time. The variance of this value over multiple runs will indicate how correlated the execution of code is in the system (either real or modeled) to the passage of time (either simulation time or wall clock time). We can then compare the value of both the mean and variance in the number of iterations performed for all three systems.

5.1.1 Algorithm

First, let us introduce the mechanism of the test. The test is composed of two threads, a timer thread and a counter thread, each having elementary tasks. The counter thread spawns the (later described) timer thread, then simply moves to a while loop that repeatedly increments an unsigned long. This unsigned long represents the number of loop iterations that have been completed. The while loop terminates when

a Boolean shared between the two threads is set to true. When the wait timer thread is spawned by the guest, it has two simple tasks. First, it sleeps for a designated amount of time, using the POSIX `nanosleep` function (which is based off the Linux kernel's `CLOCK_MONOTONIC`), with the sleep time being defined by the user in the command line arguments. After waking up from sleep, the timer thread sets the shared Boolean to true, consequently stopping the counter thread's while loop. At this point, the timer thread has done its job, allowing us to destroy it. Finally, the counter thread prints the number of loops executed to standard out (where it may be redirected to a file for future analysis). This process is repeated a number of times, defined by the user in the program's command line arguments.

A short defense of this test may be necessary. Synthetic workload benchmarks such as this are not necessarily the greatest choice of workload when doing performance analysis on a system. Exect's workload does not perform any IO, it does not make any system calls, and it only uses a negligible amount of memory. It simply iterates a loop that consists of a handful of instructions. These factors would make this workload a rather poor choice when analyzing the performance of two separate systems.

However, consider that in this project we are not comparing the performance of different systems, rather we are comparing the association of code execution to the perceived passage of time. We are not asking "Which of these systems is the fastest?" Rather, we ask "How consistent are these systems in the amount of work they can

do in a second?", and "How similar is this to how a physical system would execute the code?" For this question, the simplistic workload is a benefit, as it should help minimize variance between runs caused by issues like cache, IO, et cetera. We are merely concerned with how many instructions it can retire in a set amount of time, and how much that changes between runs. Of course, a certain amount of variation between runs is expected, due to things outside the program's fine control, such as the scheduler. Anything beyond this, however, could be attributed to the system's ability to control and manage its own execution of code. A poorly managed system will have a much greater observed variance, while a more tightly managed system should have a variance similar to that of a physical system.

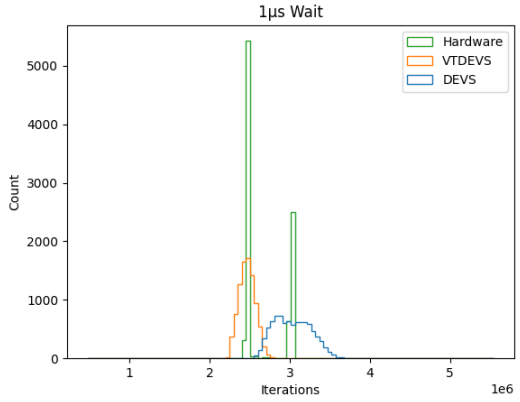
Because (in case of our emulator) the perceived guest time is simulation time, we are using this test to evaluate the correlation of the execution of code to the passage of simulation time. Testing hardware, where there is not any simulation time, gives us a good baseline of what we would ideally see in a "perfect" simulation, that is a simulation without simulation error. Of course, the results of hardware testing will inherently differ from our simulation in some ways. The physical CPU will not have to worry about a hypervisor destroying its cache, for example, it also has to deal with physical devices, rather than simulated devices.

5.1.2 Setup

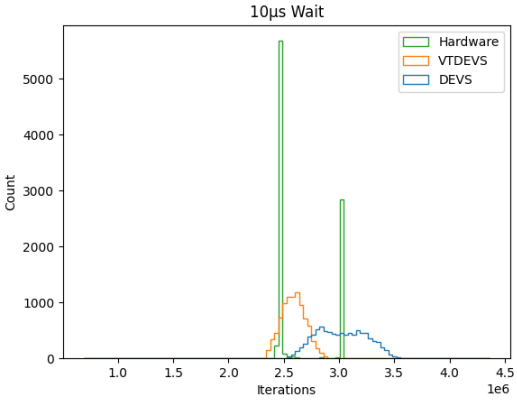
This test was performed on a machine equipped with an Intel i7-12700H and 30 GB of memory. The operating system used across tests is Ubuntu Server 20.04, using kernel version 5.4.161, with the real-time patch rt67-rc1. Tests performed in the simulators use a computer model with 1 VCPU core and 4 GB of memory. Hardware tests limited the number of cores available to the operating system using the `maxcpus=1` boot parameter. Shorter wait tests (that is, those with less than a second wait) were repeated 9000 times. Longer wait tests (one and two second waits) were repeated 1000 times. The binary was compiled with `gcc` version 9.4.0.

5.1.3 Results

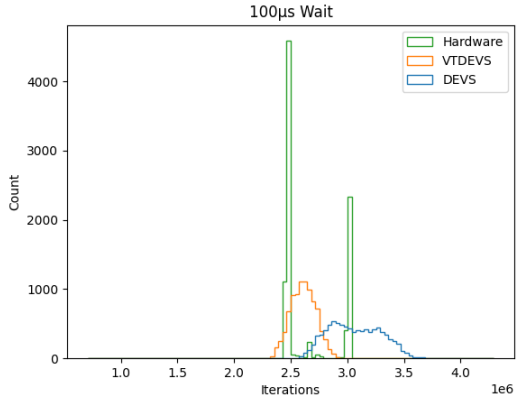
Histograms for the results of each experiment can be found in figure 5.1. The x-axis represents the number of iterations of the counter loop exect was able to execute prior to being interrupted. The y-axis represents how many instances of the exect experiment result in this number of iterations. In these histograms, the green line shows the results for tests on hardware, orange for the VTDEVS simulation, and blue for the DEVS simulation. There are a few interesting qualitative results to be seen in these histograms.



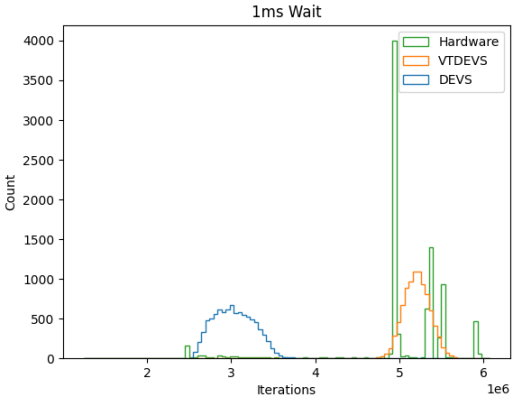
(a) $1\mu s$



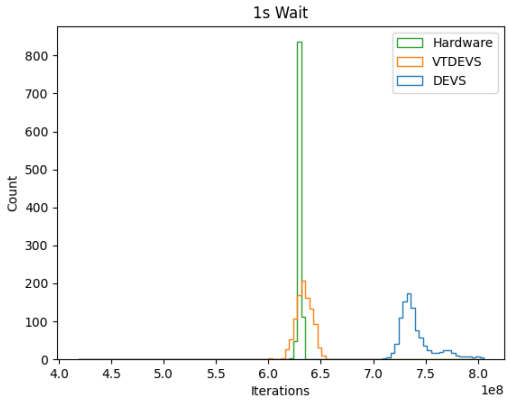
(b) $10\mu s$



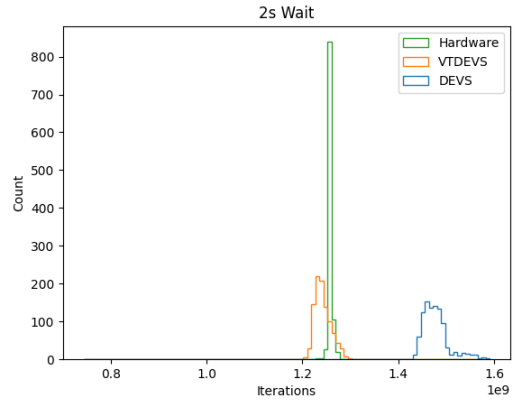
(c) $100\mu s$



(d) 1ms



(e) 1s



(f) 2s

Figure 5.1: Histograms of exect results

First, and most apparent, is that the hardware results for wait times less than 1 second are multi-modal. However, in each of these modes there is very little local variation. These modes remain in roughly the same location for waits less than one million nanoseconds (one millisecond). This suggests that while hardware is quite precise in its association of compute time to the passage of time, it is inaccurate and has several bins that it tends to fall into. The reasons for this could boil down to simply an inherent limitation in this type of benchmark, or perhaps a limit to the consistency of the Linux scheduler at such short timeframes. Notably, both simulations fail to emulate this strongly multi-modal behavior, with all simulator histograms being near-Gaussian in shape.

Second, the results for wait times less than one million nanoseconds are all very similar. This can also be seen in the mean and standard deviations of the results, seen in table 5.2. We believe this to be due to the inherent latency of the timers in the test. See the discussion of Cyclictest, section 5.2, for more information on this latency.

	Hardware	DEVS	VTDEVS
1μs	2 665 836	3 034 796	2 469 244
10μs	2 648 697	3 020 495	2 588 235
100μs	2 648 248	3 064 045	2 610 089
1ms	5 033 898	3 036 105	5 198 443
1s	629 264 022	741 220 640	633 527 054
2s	1 256 601 143	1 476 547 580	1 241 445 708

Table 5.1
Mean of exect results

	Hardware	DEVS	VTDEVS
1μs	262 860	231 331	106 871
10μs	257 666	223 951	110 687
100μs	247 378	233 448	111 354
1ms	648 832	238 254	197 136
1s	2 498 648	17 372 072	10 410 657
2s	4 585 299	38 064 135	19 202 032

Table 5.2
Standard deviations of exect results

	Hardware	DEVS	VTDEVS
1μs	0.0986	0.0762	0.0432
10μs	0.0972	0.0741	0.0427
100μs	0.0934	0.0761	0.0426
1ms	0.1288	0.0784	0.0379
1s	0.0039	0.0234	0.0164
2s	0.0036	0.0257	0.0154

Table 5.3
Coefficient of variance of exect results

The mean iteration counts of each test is listed in table 5.1. Consulting this statistic reveals some insights. Here we see that the mean iteration count for VTDEVS is much closer to the mean observed in hardware compared to the DEVS simulation. The DEVS simulation is observed to consistently run more iterations than both DEVS and VTDEVS, except for wait times of 1 millisecond, where the DEVS iteration count is far below the other two.

Table 5.2 and table 5.3 list the standard deviation and coefficient of variance (standard deviation divided by mean), respectively. Here we see that VTDEVS outperforms

DEVS in every test, having a consistently lower distribution. However, DEVS does have a much closer standard deviation to hardware in the sub-one millisecond tests, and neither being particularly close to hardware in the one millisecond test. This is due to the multi-modal nature of the hardware results at these low wait time values.

5.1.3.1 *t*-Testing

In order to assess the accuracy of the systems, we first perform several simple t-tests to find if the differences in test results are statistically significant. To accomplish this, we perform 3 t-tests between the 3 systems, hardware against the DEVS simulation, hardware against the VTDEVS simulation, and the VTDEVS simulation against the DEVS simulation. We do these three t-tests over all the experiments we perform. The t-tests all result in a near-zero p value, as seen in table 5.4. As such, we may conclude that the observed average iteration count is (statistically) significantly different between the three systems.

Comparison	p-value
HW vs. DEVS	0.0
HW vs. VTDEVS	0.0
VTDEVS vs. DEVS	0.0

Table 5.4
t-testing p-values (rounded)

	Factor	Error
1μs	54.05	45.95
10μs	41.45	58.55
100μs	43.53	56.47
1ms	73.93	23.07
1s	77.35	22.65
2s	82.17	17.83

Table 5.5
Allocation of Variation

5.1.3.2 Analysis of Variation

In a perfect system, one without error, we would expect for the exect test to result in the same number of iterations performed in a given wait time. Pragmatically, however, such a perfect system is not possible. There are many sources of error that lead to the variation of the gathered results. A small amount of this variation is inherent to the hardware the test is being performed on, for example, the speed of the CPU at a given moment, or whether the program was cached. Timer accuracy, that is the latency of timers, is another source of error to consider. Finally, there is the scheduler as the source of error. The iterator thread is not guaranteed to be scheduled for the entirety of the timer's duration. Indeed, since multiple processes are sharing a single core, there is ample opportunity for the thread to be descheduled, as such the thread will have less of an opportunity to increment the counter.

In a single factor experiment, we find the grand mean of all observed results. The

variation over all observations is either explained (that is, from differences in the systems we are measuring themselves), or unexplained (that is error, as listed in the prior paragraph). We can then find the allocation of variation, which indicates what percentage of variation is explained and unexplained in an experiment. Explained variation is called variation due to factor, and unexplained variation is variation due to experimental error. A high percentage variation due to factor indicates that the experiment is mostly measuring actual differences between the subject systems. Conversely, a high percentage due to error indicates that an experiment is mostly measuring noise from error inherent to the systems. The allocation of variation can be found in table 5.5.

Here we find that the allocation of variation due to error is quite high in the short sleep tests (less than one millisecond sleeps). This seems to show a lower bound for the utility of this benchmark, as in these tests we are mostly measuring the "random" noise of the systems from their sources of error.

This confirms the validity of the higher wait value ($\geq$1ms) results. Thus, we may confirm the validity of the VTDEVS simulation in this context. We may extend this to say that, with some amount of error, the VTDEVS simulator is capable of executing a consistent amount of code similar to hardware in a given amount of time greater than a millisecond.

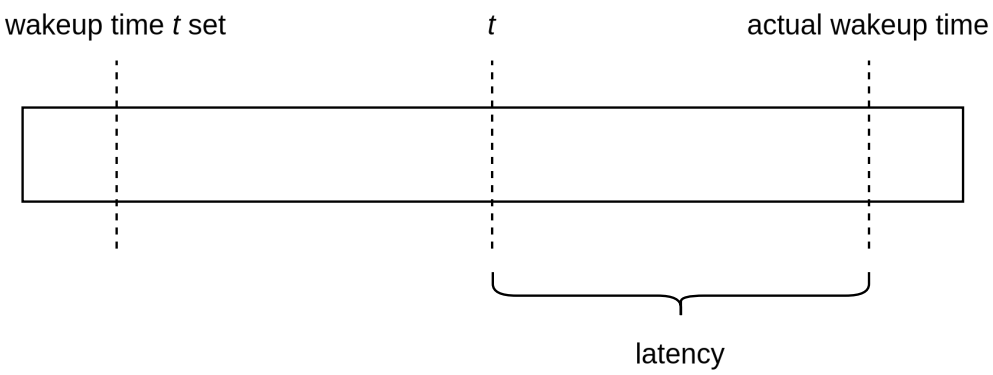


Figure 5.2: Latency

5.2 Cyclictest Benchmark

The Cyclictest benchmark is a part of the rt-tests suite of benchmarks. These benchmarks are developed by the real-time Linux kernel community for measuring the performance of real-time Linux systems in various metrics. Cyclictest, in particular, measures the latency of a real-time Linux system. Latency is defined as the difference between when something was intended to be handled, and when it was actually handled. Figure 5.2 provides an illustration of latency.

In this context, latency is in reference to the wake-up time of a thread. Cyclictest works via spawning two (or more) separate threads. One thread is a master non-real time thread that runs the experiment. The other threads are real-time threads where the experiment is performed. The real-time threads are put to sleep, and scheduled to wake up after a set interval of time (defined by the user) using an alarm.

The difference between when this alarm was set for and when it is awoken is the measured latency of the system. Cyclictest repeats this process for several iterations (again, set by the user), and reports the minimum, maximum, and average latency (in nanoseconds) [20].

An explanation of the rationale behind this choice of benchmark would be beneficial. Real time operating systems (RTOSs) seek to provide a predictable execution environment for the processes running on them. Interrupts should trigger exactly when they are scheduled to trigger, and code should run for exactly as long as expected, with little room for error. This trait is amenable to safety-critical and timing-critical systems. There are multiple methods RTOSs go about providing this sort of execution environment, such as allowing all processes (including the kernel itself) to be preempted, or attempting to minimize the nondeterministic nature of the process scheduler, for two examples. These strict timing requirements suggest that this would be an apt choice of benchmark for our system. That is, the simulated machine should have a similar latency to the physical machine, which would show an aptitude for this use case.

This benchmark is incredibly similar (mechanically) to the previously covered bespoke exect benchmark. However, the two are measuring something very different. Exect is concerned with the consistency of how much work can be done in a given amount of time, whereas Cyclictest is concerned with how consistently the OS can handle a

timer. These two measurements are related, but not equivalent. We can imagine a system that has a consistently poor latency, but that poor latency allows it to perform a very consistent amount of work. Alternatively, we may imagine a system that has little to no latency, but the ways it achieves this results in an inconsistent amount of work for that given time.

5.2.1 Setup

As with exect, this test was performed on a laptop equipped with a Intel i7-12700H and 30 GB of memory, running Ubuntu Server 20.04 with kernel version 5.4.161 and real-time patch rt67-rc1. The instances of the VTDEVS and DEVS models were limited to 4 GB of memory with 1 VCPU, while hardware was limited to one core with the `maxcpus=1` boot parameter. In addition to hardware and the simulators, we ran Cyclictest on a normal VirtualBox VM with the same setup. Cyclictest was configured to execute for 100000 iterations with a timer interval of 1000 microseconds (one millisecond), with test threads set to thread priority 90. No additional load was placed on the systems beyond the normal processes running on Ubuntu and the test itself.

	Minimum	Average	Maximum
Hardware	0	0	4
VTDEVS	3	40	40
DEVS	0	0	2
VirtualBox	3	4281	706 933

Table 5.6
Cyclictest latency measurements (μs)

5.2.2 Results

The summary results for these experiments are given in table 5.6. This table lists the minimum, maximum, and average latency as reported by Cyclictest. In these results we see that hardware has almost no observed latency, with an average latency of 0 microseconds, and a maximum latency of 4 microseconds. The DEVS simulation measured an average latency of 0 microseconds as well, with an even lower maximum latency of 2 microseconds. VTDEVS has an overall worse minimum, maximum, and average over the prior two.

VirtualBox, which is not a simulation, but rather a normal virtualization tool, has by far the worst average and maximum latency, with an average latency of 4 milliseconds, far beyond what would typically be deemed acceptable. This test was included to illustrate how unmanaged virtualization, which doesn't provide any notion of virtual or simulation time, is unfit for a simulation use case.

Shifting focus to the other results, it is clear that the VTDEVS implementation has the worst statistics of the three by a not unreasonable amount. While a 40 microsecond average latency is not terrible by any means, it is much larger than what is seen on hardware. This calls into question the validity of the VTDEVS simulation in this context. We may conclude that there is a notable amount of simulation error present in guest latency for the VTDEVS simulator. The reason why this latency is present in the VTDEVS implementation but not present in the DEVS simulation needs further investigation.

Chapter 6

Related Work

6.1 Hardware-Assisted Simulation

A plethora of prior work exists in the area of virtualization accelerated simulation, allowing for the execution of code within a simulation at near-native speeds [21][22][23]. This extant group of prior work seeks to use virtualization in a simulation context in order to model computers (or even networks of computers), either due to the convenience of virtualization as an existing computer model [23], or for the speed at which virtual machines may execute code compared to traditional simulations [22]. In this thesis, we use virtualization to achieve the latter of these goals, as a hardware accelerator for execution of code. However, as discussed prior in section 1.1, there

are many problems inherent to virtualization, such as sharing cache with the host, as well as the difficulty of scheduling VCPUs, that make this approach difficult. Despite this, the topic of virtualization assisted simulation has already been well explored, and there already exists many off-the-shelf solutions for this approach. An example of this is QBox [21], which integrates the Quick Emulator (QEMU) into the SystemC discrete event simulator. However, several problems arise when attempting to make QEMU beholden to simulation time. Primarily, this has to do with how QEMU handles events for it's emulated hardware internally. QEMU places a time to live on events destined for emulated hardware that is kept in check with wall clock time. Thus, situations may arise where events are not properly delivered to emulated hardware. Alternatively, events arrive too early (in terms of simulation time) to emulated hardware, as QEMU (being a virtualization platform) is attempting to work in real time [5].

6.2 Virtualized Time

The concept of virtual time driven simulation, as described in this thesis, is not new. It has been described before by [24], and has been used in simulation contexts [25], to much success. However, this solution relies upon the modification of the Xen hypervisor, and particularly Xen's scheduler. For this, it relies upon Xen's accounting of domain time for determining when and for how long to schedule a VCPU core. We

should be able to get a more accurate accounting of domain time using the TSC, and should be able to get a more precise control of VCPU scheduling by using the VMX preemption timer. Xen being a Type-1 hypervisor may also be a drawback for some use cases, for example, consider the situation that this simulation will be executed on a developer's workbench, this would require that their main working environment be just another guest of the Xen hypervisor. Further, there is little prior work in introducing virtual time driven simulation to DEVS simulations specifically.

6.3 Time Dilation

Another common approach towards using virtual machines in a simulation context is the use of time-dilated virtual machines [26][27][28]. This approach scales the advancement of wall clock time to trick the VM into perceiving itself as having a slower or faster CPU than the one it is actually executing on. The amount by which the passage of time is scaled is called the time dilation factor. For example, we may use a time dilation factor of 2 to make the VM perceive itself as having a CPU half the speed it actually does. Time dilation has found its use in simulation contexts, particularly in the testing of physical networks, as well as the simulation of networks.

Time dilation has the benefit of being simple, and having the potential of working with real-world components. For example, in [26] the authors modify the Linux kernel

to provide Linux Containers a dilated view of wall clock time. These time dilated containers interact with physical components in a network, however the time dilation results in the computer perceiving the network as being faster or slower than it is in reality, allowing for an accurate model of the network at different speeds. In [27], the authors provide a similar view of dilated time to virtualized guests, however in this case, the modifications have been made to the Xen Hypervisor.

Time dilation differs inherently from our approach. In time dilated systems, the guest's view of time is still inherently linked to the passage of wall clock time, it is simply that said view of time is scaled by the time dilation factor. This works well for some use cases, for example when a simulated system needs to operate and interact with real-world devices. However, in the context of a complete simulation of systems, where a simulated guest need not interact with the physical world, then any connection to wall clock time is undesirable. Some work has gone into time dilation to try to avoid this problem by using an adaptive time dilation factor, an example of this can be found in [28].

6.4 Resource Dedication

One of the most common solutions used to improve the fidelity of code execution time is by dedicating CPU cores to the virtual machine. In this method, every VCPU

core in the virtual machine is assigned to some specific hardware CPU thread. This solution removes some problems introduced by the unpredictable nature of the Linux scheduler. Any given thread dedicated to the VCPU will spend much of its time executing guest code. When this is done, we can simply use the vTSC as is, with a reasonable assumption that the passage of guest time is highly proportional to the execution of guest code. Resource dedication is a pretty standard feature for most virtual machine managers. QEMU is capable of assigning a VCPU to a given processor via libvirt [29][30]. An example of this can be found in [31], which seeks to provide an adequate environment for the virtualization of real-time operating system.

However, this does not dedicate the physical thread to solely executing guest code, there are still times in which virtual TSC is still ticking (as physical time is still ticking), but guest code is not being executed. For example, when the guest exists execution is returned to the host kernel. In the time it takes to handle that exit condition, guest time is still advancing, but no guest code is being executed. Thus, it is more of a band-aid solution for our specific use case.

This solution also has another, more practical, problem. It requires that for each VCPU, there is a CPU thread to dedicate to it. This requirement makes the solutions impractical. Take a situation where we are simulating a computer system with more cores than we have physically on the machine we are running the simulation on. Imagine we are running a simulation with many computing systems that add up to

have more cores than we have physically. These use cases are prohibited due to the solution's inability to over dedicate computing resources. It may simply be a developer is running this simulation on her workbench, and does not wish to dedicate a large amount of her processing power to the simulation, reasonably wishing to multitask while the simulation is executing.

Chapter 7

Future Work & Conclusion

7.1 Support for Multiple VCPUs

VTDEVS, as described in this paper, is limited to one VCPU core. This, however, is not a practical limitation for multiple reasons. First among these is that multicore computing has become common place, and a considerable amount of software may require multiple cores to run properly. As such, the simulation must be able to support simulating multicore systems if it wishes to model such software. Second is the fact that a simulation could model two separate computing systems at once. For example, think of a simple satellite, with a computer on the satellite controlling local hardware, and a ground station computer that issues commands to the satellite computer. Even

if both these computers could be modeled with single core processors, VTDEVS (as described) could not handle such a model, as, in total, there would still be multiple VCPUs.

7.1.1 The Interrupt Problem

Let us, for the moment, ignore the host OS's scheduler. We will assume that we can at any instant tell all the VCPUs in our simulation to start executing in harmony with one another, as though we have just fired the starting pistol at the beginning of a race. Let us look at the example in figure 7.1. In this example the simulation has two VCPUs, both at simulation time $t = S$, with the next event being at time $t = E$. Thus, the simulation schedules the VCPUs to run to $t = E$, and using our magic scheduler we can do this perfectly. However, at point $t = I$, VCPU0 encounters an interrupt that generates a simulation event. This event, when handled by the

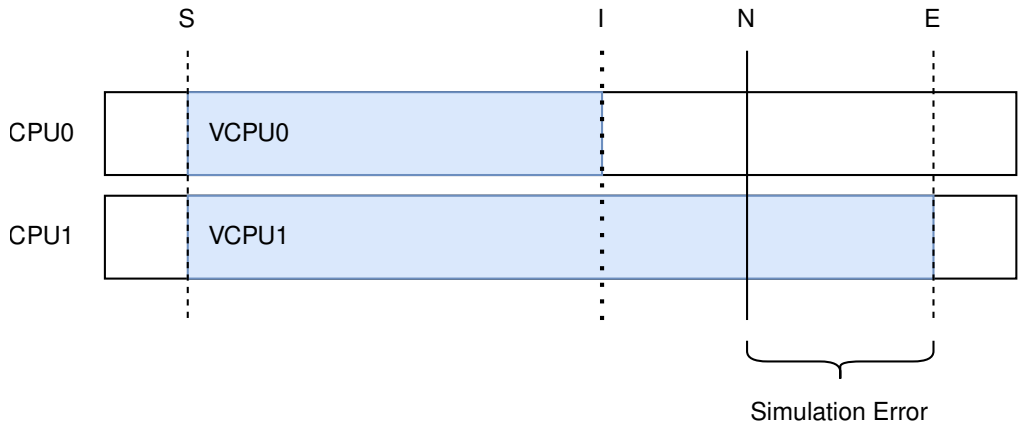


Figure 7.1: Interrupt error

simulation, ends up moving the next event time forward to $t = N$. Thus, to prevent the VCPUs from over-executing guest code past what the simulation has allotted, they should only be run to $t = N$. In the scenario we have given in figure 7.1, the simulator is lazy, and when VCPU0 is interrupted it simply allows VCPU1 to continue executing to the originally scheduled point $t = E$. However, this means that when the simulation is updated, and the next event time is moved to $t = N$, VCPU1 has over-executed by $E - N$ ticks.

The best way to avoid this type of error is to configure the system that when a VCPU is interrupted with an event that may update the state of the simulation, all other VCPUs are interrupted as well. If we could stop VCPU1 the moment VCPU0 is interrupted at $t = I$, this error would not have occurred. Sadly, we are limited by the mechanism present to us. There seemingly isn't any way to instantly have all the other VCPU stop executing whenever we want. Instead, we are reliant on thread signals or some other measure to interrupt the VCPUs. These will inherently have some latency to them, thus leaving interrupt errors as a possibility, although a mitigated one.

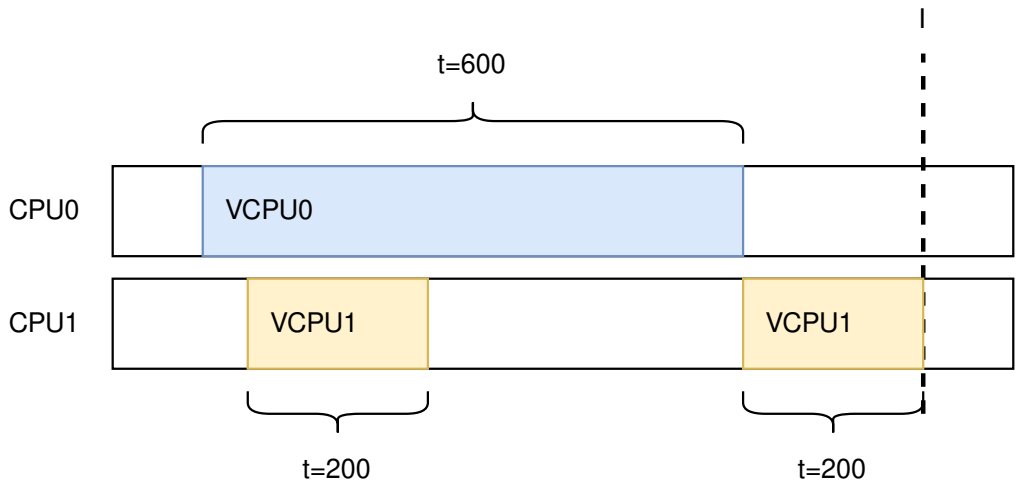


Figure 7.2: Over-execution error

7.1.2 The Host Scheduler Problem

The next, and perhaps most difficult problem, stems from the host operating system's process scheduler. In essence, there are two basic types of error that can occur due to the host scheduler. These are over-execution and under-execution error.

Over-execution error is when a VCPU generates an event that occurs in the past for another VCPU. An example of over-execution error is shown in figure 7.2. In this example, let us assume that the simulation has two VCPUs, and that it has determined the next event to be at time $t = 1000$, as such it schedules both VCPUs to execute for 1000 ticks. From this point onward, the actual execution of the VCPUs is at the whims of the host scheduler. The scheduler happens to schedule VCPU0 for 600 ticks of time, and as such, to that CPU it has perceived simulation time as

having advanced 600 ticks. However, in this same amount of time the host scheduler has only executed VCPU1 for 200 ticks of time. After both of these blocks of execution occur, the scheduler reschedules VCPU1 to continue executing. Another 200 ticks of execution pass, and VCPU1 is interrupted in a way that generates a simulation event. Because this event could change the state of the model, we need to have it handled immediately. However, there is no clear way that the simulation should handle this. If the simulation uses VCPU1's time, then the event will have already occurred in the past for VCPU0. If the simulation uses VCPU0's time, then the event will have occurred late to VCPU1. If the simulation simply advances VCPU1's clock to meet VCPU0's, then time would have passed without any execution occurring (breaking our association of execution time with simulation time). Similarly, if the simulation rolls back VCPU0's clock to meet VCPU1's, we will have over executed for the simulation time we are setting. The observant reader will notice that the error in figure 7.1 is also a case of over-execution error.

The second type of error is under-execution error. This error occurs when an event is generated by one VCPU at a time that has not yet occurred for another VCPU, that is the other VCPU has under-executed at the time the event is generated. An example of this is seen in figure 7.3. The setup to this example is similar to that seen in the previous example, the simulation has two VCPUs, and they have both been scheduled to execute for 1000 ticks. In this scenario, however, the host's scheduler happens to prioritize the execution of VCPU0 before VCPU1. After 600 ticks of execution time,

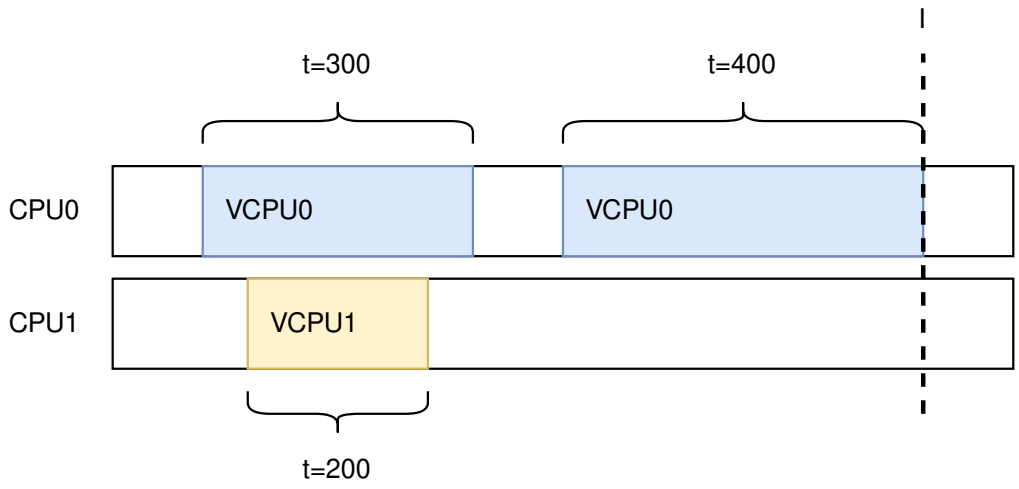


Figure 7.3: Under-execution error

VCPU0 is interrupted that generates a simulation event. Again, this event must then be handled by the simulation, as it may update the state of the model. However, in this time, VCPU1 has only executed for 200 ticks. We find ourselves in a similar situation to before, however, in this case the "incorrect" VCPU is in the past. Unlike the previous situation, there appears to be a trivial solution. Because VCPU1 is in the past by 500 ticks, simply schedule VCPU1 to run for another 500 ticks to catch up to VCPU0. Consider that if we do this, in the time it takes to execute those 500 ticks, VCPU1 could possibly encounter an interrupt that generates an event of its own, which would turn this into an over-execution error.

The problems of over- and under-execution error are exacerbated as the number of VCPUs introduced into the simulation grows. As the number of processes fighting over a limited supply of physical cores upon which to execute increases, the likelihood of an out-of-order interrupt increases, just due to the nature of only a small number

of VCPU cores being able to execute at a given time. A further complication is that the complexity of the simulation increases as the number of VCPU cores increases as well, a simulation with 5 VCPU cores needs at least 5 seconds worth of compute time to advance the simulation forward 1 second. As such, it is incredibly important that a solution for handling multiple VCPUs scales well enough to handle these issue.

7.2 Distributed VTDEVS Simulation

DEVS lends itself very well to being distributed over multiple systems. DEVS simulation nodes, as seen in section 2.3, need only communicate by passing simple messages with their parent and direct child. It is thus possible for multiple computers in a network to distribute the execution of a DEVS simulation. This allows for large and complex models, which if simulated on a single computer may constrain resources, to be run over a network of computers [1]. The main theoretical issue that must be solved prior to proper distributed VTDEVS is the aforementioned support for multiple VCPUs. From there, one must also solve several non-trivial implementation issues. How would distributed VCPUs share virtual memory? How do you handle the possibility of the computers running the distributed simulation having different CPU microarchitectures?

7.3 Conclusion

Modeling and simulation are, in some ways, an art. The simulation author is faced with trying to balance several conflicting goals while creating the simulation. Accuracy and granularity, abstraction and performance; these goals contrast and may conflict with each other. Thus, it is necessary that the simulation writer make decisions on what is valued in the simulation, and what is not. This dilemma extends to simulations that include software as a component. If the simulation author eschews the abstraction of the formal software model approach for the fine granularity of the computer model approach, then we would expect that there should be some corresponding deficit in the performance or accuracy of the model. For this problem, we introduce VTDEVS. VTDEVS, in this situation, prioritizes performance over accuracy, foregoing the computational cost of a formal model of a CPU for the performance of hardware virtualization. In practice, in order for simulations to take advantage of the VTDEVS formalism, we also needed to retrofit an existing hypervisor, KVM, to allow for the fine control of VCPU execution and presentation of time, which would minimize the loss of accuracy from using virtualization. Finally, we validate an example model to test our VTDEVS approach, where we show that the VTDEVS approach is reasonably accurate in regards to real world behavior for two benchmarks.

References

[1] Zeigler, B. P.; Muzy, A.; Kofman, E. *Theory of Modeling and Simulation;* Academic Press, third edition ed., 2019.

[2] SimpleScalar. SimpleScalar. `https://pages.cs.wisc.edu/~mscalar/simplescalar.html`, **Last Accessed: 2024-07-10**.

[3] Onder, S.; Gupta, R. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*, pages 80–89, 1998.

[4] Bochs IA-32 Emulator Project. Bochs. `https://bochs.sourceforge.io/`, **Last Accessed: 2024-07-10**.

[5] Nutaro, J. J. Time managed virtualization for simulating systems of systems Technical report, Oak Ridge National Laboratory, **2021**.

[6] Spin. Spin. `https://spinroot.com/spin/whatispin.html`, **Last Accessed: 2024-07-10**.

[7] Alloy. Alloy. `http://alloytools.org/`, **Last Accessed: 2024-07-10**.

[8] Amsden, Z. Timekeeping virtualization for x86-based architectures Technical report, Red Hat.

[9] Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide, part 2. Intel.

[10] Intel 64 and ia-32 architectures software developer's manual volume 2 (2a, 2b, 2c, & 2d): Instruction set reference, a-z. Intel.

[11] Intel 64 and ia-32 architectures software developer's manual volume 4: Model-specific registers. Intel.

[12] Intel 64 and ia-32 architectures software developer's manual volume 3c: System programming guide, part 3. Intel.

[13] Amd64 architecture programmer's manual volume 2: System programming. AMD.

[14] The Definitive KVM (Kernel-based Virtual Machine) API Documentation. Linux. `https://docs.kernel.org/virt/kvm/api.html`, **Last Accessed: 2024-07-10**.

[15] Intel 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1. Intel.

[16] Intel. Timestamp-Counter Scaling for Virtualization White Paper Technical report, Intel.

[17] Chow, A. C. H.; Zeigler, B. P. In *Proceedings of Winter Simulation Conference*, pages 716–722, 1994.

[18] KVM-specific MSRs. Costa, G. `https://docs.kernel.org/virt/kvm/x86/msr.html`, **Last Accessed: 2024-07-10**.

[19] A tsc_freq_khz driver for everyone. Trail of Bits. `https://github.com/trailofbits/tsc_freq_khz`, **Last Accessed: 2024-07-10**.

[20] Cyclic Test. The Linux Foundation. `https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start`, **Last Accessed: 2024-07-10**.

[21] Delbergue, G.; Burton, M.; Konrad, F.; Le Gal, B.; Jego, C. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, 2016.

[22] Falcon, A.; Faraboschi, P.; Ortega, D. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 72–83, 2007.

[23] Rösch, D.; Nicolai, S.; Bretschneider, P. In *2021 6th International Conference on Smart and Sustainable Technologies (SpliTech)*, pages 1–6, 2021.

[24] Yoginath, S. B.; Perumalla, K. S.; Henz, B. J. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 68–77, 2012.

[25] Yoginath, S. B.; Perumalla, K. S.; Henz, B. J. *The Journal of Defense Modeling and Simulation* **2015**, *12*(4), 439–456.

[26] Lamps, J.; Nicol, D. M.; Caesar, M. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '14, page 179–186, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Gupta, D.; Yocum, K.; McNett, M.; Snoeren, A. C.; Vahdat, A.; Voelker, G. M. In *3rd Symposium on Networked Systems Design & Implementation (NSDI 06)*, San Jose, CA, 2006. USENIX Association.

[28] Lee, H. W.; Thuente, D.; Sichitiu, M. L. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '14, page 167–178, New York, NY, USA, 2014. Association for Computing Machinery.

[29] libvirt. libvirt. `https://libvirt.org/`, **Last Accessed: 2024-07-10**.

[30] Configuring Real-Time Compute. Red Hat. `https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/13/html/instances_and_images_guide/realtime-compute`, **Last Accessed: 2024-07-10**.

[31] Scordino, C.; Savino, I. M.; Cuomo, L.; Miccio, L.; Tagliavini, A.; Bertogna, M.; Solieri, M. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1, pages 353–360, 2020.

[32] KVM PVclock. Red Hat. `https://www.linux-kvm.org/page/KVMClock`, **Last Accessed: 2024-07-10**.

[33] Clocks and Timers. Linux. `https://docs.kernel.org/virt/hyperv/clocks.html`, **Last Accessed: 2024-07-10**.

[34] Kong, X.; Kong, X. *CSAE* **2021**, *October*(5), 1–5.

[35] Zhang, J.; Chen, K.; Zuo, B.; Ma, R.; Dong, Y.; Guan, H. In *5th International Conference on Computer Sciences and Convergence Information Technology*, pages 421–426, 2010.

[36] García-Valls, M.; Cucinotta, T.; Lu, C. *Journal of Systems Architecture* **2014**, *60*(9), 726–740.