ORNL/TM-2021/1994

Time managed virtualization for simulating systems of systems

James Nutaro

November 24, 2021

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2021/1994

Computational Sciences and Engineering

Time managed virtualization for simulating systems of systems

James Nutaro

Date Published: November 24, 2021

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

ACRONYMS

ORNL	Oak Ridge National Laboratory
KVM	Kernel Virtual Machine
QEMU	Quick Emulator
pci	peripheral component interface
ioapic	input output advanced programmable interrupt controller
lapic	local advanced programmable interrupt controller
pit	programmable interval timer
adevs	a discrete event system simulator
irq	interrupt request line
tsc	time stamp counter
HLA	High Level Architecture
FMI	Functional Mockup Interface
IEEE	Institute for Electrical and Electronics Engineers

ABSTRACT

Software is an integral part of almost every non-trivial system. Yet simulation technologies to understand how software affects system performance lag behind what is available to understand physical behavior. This is not surprising. An abstract model of software tells us what it is supposed to do, not what it actually does. In this respect software is very different from a physical system. Kirchhoff's laws predict the behavior of an electrical circuit with great precision. The mathematics of fluids do an adequate job of anticipating how a wing will perform in flight. Only the software itself, observed in operation, can tell us how it behaves.

The use of simulated computers to place "as is" software into a simulated world is commonplace for embedded software systems. The term embedded is generally synonymous with small, both in form factor and in computational capability. Embedded computer systems are constrained by stringent requirements for reliability, responsiveness, power use, and other factors [9]. These factors favor simpler computers with greater reliability and predictability than can be had from computers typically used in information technology applications. One consequence of simplicity is that an engineer can usefully simulate the target hardware with their software development workstation. For many embedded software systems, these simulations are vital to verifying reliability and performance; see, for example, [32, 37].

There is a growing class of software that has many of the requirements associated with embedded systems but need the computational clout of a thoroughly modern computer. This class of software poses a significant challenge when simulations are being used to examine the system of which the software is a part; that is, when attempting to simulate the system of systems. Simulation techniques for smaller computer systems are impractical when the target computer is as computationally capable as the computer hosting the simulation. Nonetheless, simulation remains an important tool for assessing the system of systems, and this need prompts the creation of simulation models of software applications. The particular challenges of modeling software functionality makes these models costly to build, validate, and maintain.

Adaptations of existing virtualization tools have been proposed in various forms to address this problem; see [44, 47, 18, 13, 23, 43, 21] for a sampling of these efforts. Virtualization tools simulate the hardware needed to execute software and do so quickly by leveraging real hardware to accelerate the simulation. Executing instructions on a real processor, rather than a simulated processor, is the primary means of acceleration. Unfortunately, existing virtualization tools are not built for system of systems simulations. Their purpose is to share computational hardware between large numbers of software applications that must perform in the real world. Simulation research centered on existing virtualization tools seeks to swap real and simulated worlds within the confines of an existing design and implementation.

Many of the challenges faced when working with an existing virtualization tool disappear if we build a new computer system simulator. The obstacle to doing so is economic more than technical. This report examines the economic and technical considerations involved in building a virtualization tool that is intended specifically for system of systems simulations. The products of this examination are three. First, an elucidation of requirements imposed by the need for our simulator to participate in a larger simulation program, with a specific emphasis on federated simulations. Second, a computer system simulator that meets most of these requirements and a description of the technical advancements needs to satisfy the others. Third, a discussion of the cost of the new computer system simulator in relation to its alternative, which is to build models of the software rather than the hardware on which it executes.

1. INTRODUCTION

Models of software are essential to simulations that examine how well a system of systems will perform its mission. However, simulation models of software pose significant cost and schedule challenges. Software evolves rapidly. The delay between changes in software and updates to its model often means that simulation outcomes reflect out of date functionality and therefore are diminished in value. If the out of date model is not acceptable for a simulation exercise then schedule delays are inevitable. There are also costs associated with reprogramming key aspects of the software into a form suitable for simulation, and differences between actual functionality and its incarnation in the model are the inevitably consequence of simplifications and programmer errors. These differences may contribute to omissions and errors in the simulation outcomes, again diminishing their value.

Consequently, there is a strong incentive for using software "as is" rather than building models of software functionality. Though software may evolve rapidly, hardware typically does not. It is not uncommon for several generations of software to use identical hardware. When a change of hardware occurs, the change in relevant functionality is often very small and may be insignificant. Most commercially available hardware supports a vast amount of existing software, which ensures that new hardware is functionally identical to the old for most software systems. The relative stability of hardware engenders three motivating possibilities.

1. A model of hardware detailed enough to run software "as is" may be less expensive to build and maintain than a model of the software.

2. Many software systems have common requirements for hardware, permitting the cost of a hardware model or family of related hardware models to be spread across several simulation activities.

3. The software "as is" executing on a valid hardware model is up to date and contains all of the software's functionality.

At first glance, it may appear that there is nothing to do. Virtualization technology is readily available, and nearly two decades of research have explored its use within simulations [44, 47, 18, 13, 23, 43, 21]. Moreover, computer system simulators are routinely used by computer architects and embedded software programmers to examine software "as is" in a simulation. Gem5 and Open Virtual Platform are good representatives of such simulators [2, 29]. Surely, this prior work addresses our need?

Our focus on a system of systems [4], of which the software is only a part, distinguishes our aim from prior objectives. In a system of systems, the software is being used to fulfill a function that coincides with the original purpose of its designers but did not dictate the design. Before the software is integrated into a system of systems, we have substantial experience with the software in its primary role. Software testing and computer system performance are not our concerns. Rather, we wish to assess the contribution of the software to the purpose of the whole.

The computational demands of the computer system model are a deciding factor in its usefulness for these types of simulations. The turn around time for a system of systems simulation must be sufficiently short that a systems engineer can examine alternative configurations, sensitivity to changes in operating conditions, and a host of other issues that require repetition to address. A recent survey of computer architecture simulation tools by Akram and Sawalha [1] emphasizes time: hours or even days are needed to simulate a single benchmark application running on a model of a modern microprocessor. These turn

around times make computer architecture simulations impractical for addressing system of systems questions.

Virtualization technology can solve the problem of turn around time in principle. However, existing virtualization technologies are not intended to be used as components in a simulation model, and the limits to doing so appear rather quickly as we move away from a specific simulation and towards a general capability. This is not due to any deficiency of virtualization products in their intended role of sharing hardware resources between software applications. Rather, the mismatch comes from a virtualization tool enabling interaction between virtualized workloads and the real world whereas we wish to interact with a simulated world.

This report describes technology that permits us to move away from modeling software and towards using the software "as is" within a system of systems simulation. Our specific aims are three. First, to elucidate at least some of the requirements for a computer system simulator that is to be used within a system of systems model. Second, to demonstrate a working prototype of such a computer system simulator that offers a concrete illustration of the technical problems involved. Third, to demonstrate that it may be more economical to model computer hardware than to model software functionality.

2. REQUIREMENTS

System of systems simulations typically occur in a technological and organizational ecosystem that imposes constraints on the simulator. For the vast majority of simulation models, these constraints amount to no more than building a simulation program in "the usual way", which is almost certainly familiar to anyone who has programmed a computer. Virtualization technologies deviate substantially from a usual software application, and so an explicit statement of these constraints comprises new requirements for the virtualization tool when it is used as a simulator. We list these constraints before discussing the factors that make them pertinent. The simulator must

1. be linked with run-time infrastructure software that provides time management and data exchange services;
2. run as a regular operating system process, which can be
   (a) subjected to review by computer security tools and
   (b) hosted on a virtualized workstation;
3. operate correctly when the physical computer hardware is less capable than the simulated hardware;
4. operate correctly with unmodified guest binaries; and
5. execute at a rate sufficient to support systems engineering studies, which may include parameter sweeps, analysis of alternatives, and other activities requiring very large numbers of simulation executions.

A system of systems simulation often involves many participants from diverse engineering organizations. Each organization brings models of its own systems. These models are interconnected to form a federation, in which the individual component models are called federates. Technical inter-operation of federates within the federation relies on standards: the IEEE High Level Architecture (HLA) [15] and the Functional

Mockup Interface (FMI) [10] are typical examples. At the heart of a federation is its run-time infrastructure that provides an algorithm to coordinate the advancement of simulation time and the exchange of simulation data. These services are exposed to federates through an application programming interface. A federation runs on wall clock time if there are human actors or hardware in the loop and fast as possible if all of the federates are simulation models.

To operate within a federation, time in the computer simulator must advance in accordance with time advance grants issued by the run-time infrastructure. Naturally, this implies that the computer system simulator must present time to its software in a manner consistent with the federation's simulation clock. It is also necessary that data exchanged with other federates be coordinated with advances of the simulation clock. For example, a message generated at simulation time t must be offered to the federation at simulation time t. The run-time infrastructure services needed to meet these requirements are typically provided in the form of a shared object file or static link library. To use the run-time services, the simulator must link with these files. Consequently, the simulator, or some portion of it, must run as a process on the operating system for which that shared object or link library was built.

The computational environment in which the simulation will run may also impose constraints on the computer model. If the processor cores and memory of the physical computer hosting the simulation are shared by several simulation models, then our computer system simulator must run properly when it is under-provisioned with at least some hardware resources. For instance, a simulation of a computer with two processor cores must behave properly when only one physical core is available.

Policies, procedures, and economics in the organization where the simulator will be used impose their own constraints. For example, there is growing momentum within the Department of Defense towards "modeling and simulation as a service". Among the several motivators for this change is eliminating the costs of computer hardware dedicated solely to modeling and simulation; see, e.g., [24]. In practice, this means using cloud computing to allocate computational resources for modeling and simulation as they are needed, and to free these computational resources for other tasks when the simulation study is complete. Clearly, it is undesirable to create new simulation technologies that are incompatible with this aim.

Computer security policies also place limits on what is achievable. Approval processes are a barrier to deploying new software technology, and approval processes become increasingly stringent as technology moves down the software stack. An operating system presents a greater security risk and invites greater scrutiny than a software application that simulates an aircraft in flight. Getting new application software approved will be a time consuming but familiar process. Insurmountable barriers are likely to be encountered when seeking approval for a new operating system or hypervisor. This is particularly true if the sole justification for the technology is to support simulation exercises and models of software functionality are a technically viable, if expensive, alternative*.

The economic and schedule benefits of using software "as is" are diminished if this requires modifications to an operating system or software libraries. Though these types of modifications are effective in many contexts [27, 30, 20], they are in conflict with the motivating possibilities enumerated in the introduction. A model created by modifying the operating system, software libraries, or both ties the model to those particular supporting elements. For mission critical applications, the operating system and software libraries are often treated as part of the software system and are updated with it. Moreover, these are likely

*The budget for cyber-security and the budget for M&S are rarely, if ever, controlled by the same parts of an organization, and often security requirements are imposed from without. In this environment, it is challenging to achieve a deliberated balance between cyber-risks and M&S budgets.

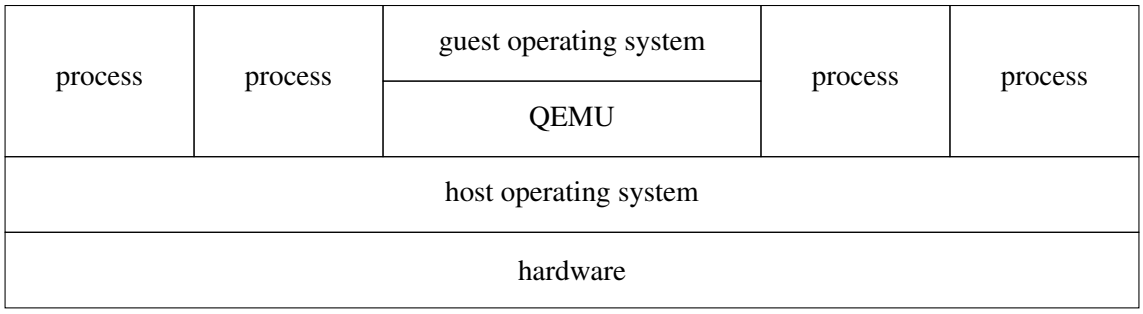

Figure 1. A type I hypervisor

to be specific to the particular software we wish to model. Hence, it may not be possible, or sensible from a modeling perspective, to use one operating system in system of systems simulations and another in operations. Even if we assume that the operating system or library source code is available to be modified, we incur the costs and technical risks of updating and maintaining a set of operating system and library patches as the application evolves.

The turn around time of the federated simulation is another element to consider. It is rarely the case that a single simulation execution will suffice. Most likely, many are needed to sample a parameter space, to estimate probabilities, or to explore variations of a scenario. Our computer system simulator should not unduly extend the running time of a federated simulation. If we wish to simulate a small computer using a much larger one, our options are many. For example, very detailed models of a micro-controller, like the popular 8051 and 8052, can be simulated rapidly using a typical desktop or workstation computer. On the other hand, to simulate a powerful server with your desktop workstation in a time frame suitable to support systems engineering tasks requires a compromise between detail and speed of execution.

3. RELATED WORK

Several related efforts seek to use existing virtualization technologies for the purpose of incorporating "as is" software into a simulation model. We describe the details of some of these technologies as they pertain to needs identified in Sect. 2., highlighting how specific features of the technology are incompatible with our goals. We do not suggest that suitable modifications, or possibly unrecognized aspects, of any of these technologies cannot make them suitable for system of systems simulations. Rather, our review is intended to identify specific technical problems that will form the focus of our development effort.

3.1 QBox, QEMU/adevs, and time dilation

QBox is an offshoot of the Quick Emulator (QEMU) computer system emulator, and its purpose is to add a virtualization capability to the SystemC simulation framework [8, 16]. SystemC is an IEEE standard describing a process oriented discrete event simulator intended for modeling computer hardware and other electronic systems [16]. QEMU is a virtual machine that permits the execution of guest software, including the operating system and applications, without modifications to the guest.

QEMU is a type I hypervisor. This class of hypervisors is characterized by running as a regular process within the host operating system. For this reason, type I hypervisors are also called hosted hypervisors.

The host operating system retains control over the physical hardware of the computer. Consequently, a hosted hypervisor must simulate physical devices that the guest operating system expects to manage. Figure 1 illustrates this arrangement.

QEMU and QBox support several microprocessor models by using a just in time compiler to translate guest machine instructions into equivalent sequences of host machine instructions, but the translation step can be very slow. When guest and host have the same instruction set architecture, the Kernel Virtual Machine (KVM) can be used in place of the just in time compiler. The KVM is a Linux kernel module that executes guest instructions directly on the host microprocessor [19]. This greatly improves performance, permitting simulations that approach the execution speed of software running directly on the host.

There are two variants of QBox. In one instance, QBox orchestrates the simulation and SystemC is used to model hardware within the simulated computer system. In the other, QEMU is modified to act as a component within the SystemC simulator. The latter permits QBox to simulate a computer system as one component of a larger SystemC model, and this mode of operation meets several of our requirements.

1. Guest software may be executed without modification.
2. Near native execution speeds contribute to rapid turn around.
3. Time perceived by the guest matches simulation time in the SystemC model.
4. The virtual machine is like any other software application; it may be easily linked to the federation run-time libraries, scanned by cybersecurity tools, and in all other respects is no different from a typical simulation model.

One deficiency of this approach, probably resolvable with a small effort, is that the SystemC standard does not include provisions for participating in a federation. The standard only addresses how the SystemC kernel orchestrates model execution without deferring to a higher authority; see [16], Sect. 4.2. Nonetheless, prototypical extensions have been created that enable SystemC to participate in a federation [35].

Our first attempt to meet the requirements in Sect. 2. borrowed heavily from the QBox approach, deviating from it in substantial ways only to resolve the problem of participating in a federation. Rather than using QBox within SystemC, we outfitted QEMU to act as a component within A Discrete EVent system Simulator (adevs) [25]. The adevs package evolved with a specific emphasis on federated simulations [27], an emphasis due in large part to the author's work with the IEEE HLA standard.

QEMU as an adevs model, which we will refer to as QEMU/adevs, emphasizes communication through the model's Ethernet card and serial port. General purpose support for memory mapped input/output and port input/output were omitted. However, the time synchronization mechanism in QEMU/adevs and in QBox have a common principle of operation. See [31] for a detailed treatment of the QEMU/adevs synchronization mechanism. In brief, it has four parts:

1. Allow QEMU to execute for h seconds of wall clock time and then pause QEMU.
2. Advance the adevs simulator by h seconds of simulation time.
3. Transform simulation messages destined for QEMU/adevs into Ethernet frames or serial data and vice versa.
4. Repeat these steps.

The deficiency of this approach to synchronization becomes apparent when we use KVM to accelerate the microprocessor model. The source of this deficiency is in the design of QEMU as a virtualization tool. A centerpiece of the design is to present the guest operating system with wall clock time when it uses the simulated timing hardware. Time keeping for a virtual machine that acts in the real world is surprisingly difficult, and the algorithms that accomplish this are deeply ingrained in QEMU and, to a lesser extent, the KVM. An article by VMware offers a concise introduction to the problem of time keeping in a virtual machine [42].

QEMU, and QBox, use an event list to manage time for the virtual hardware. This list contains an expiration time stamp for scheduled activities, such as generating an interrupt from a piece of timing hardware; for example, the advanced programmable interrupt controller (apic) or programmable interval timer (pit). When a hardware model places an event into the list, a timer is started that will expire after an interval of wall clock time commensurate with the scheduled expiration time. QEMU uses Linux's CLOCK_MONOTONIC for this purpose, which measures the real passage of time in relation to some arbitrary starting point; for example, when the system boots. The important point is that QEMU's modeled hardware always attempts to execute in real time when KVM is being used to accelerate the microprocessor simulation.

For each processor core in the virtual machine, there is an operating system thread executing that virtual core's machine instructions. Consequently, the number of machine instructions executed by the virtual core is determined by the amount of time allocated to the thread by the host operating system. If the host's hardware is supporting just a few virtual processor cores in relation to the available, physical processor cores then the time allotted to each thread will closely approximate the passage of wall clock time.

However, if the number of virtual processor cores is similar to, or greater than, the number of physical cores then the time allocated to each thread will be less than the wall clock time that passes. For example, suppose the operating system must schedule two threads for every physical core. One solution is to let each thread execute on the hardware for a short time, with each thread taking its turn. In this case, for every 1 second of wall clock time each thread receives $1/2$ second to execute machine instructions (discounting, of course, the time required for a context switch).

The result is an erratic relationship between time perceived by the model hardware and time perceived by the virtual processor core. In h seconds of wall clock time, a virtual core will be allocated αh seconds to execute, where $\alpha < 1$. The virtual core measures time by counting interrupts received from some device, such as the apic or pit, or reading a timer register, such as the the power management timer. If N interrupts are produced by a hardware model in the interval h, then the model's intended interrupt rate is N/h interrupts per second. However, the virtual core will actually receive interrupts at a rate of $N/(\alpha h)$ per second. Consequently, time perceived by the virtual core is proceeding much more quickly than it should.

For example, suppose the guess wants to approximate the number of instructions per second that the hardware is capable of executing; this is the BogoMIPS measurement made by the Linux kernel at boot time. To do so, the guest configures a timing device to generate an interrupt in h seconds. While the timer runs down, the guest executes in a loop and keeps a count of the instructions executed; the loop (in psuedo-assembly) might look something like this.

```
      mov  reg0, 0; # Store 0 to register zero
loop: incr reg0, 1; # Increment the value in register zero by 1
      jmp  loop;    # Repeat the loop
```

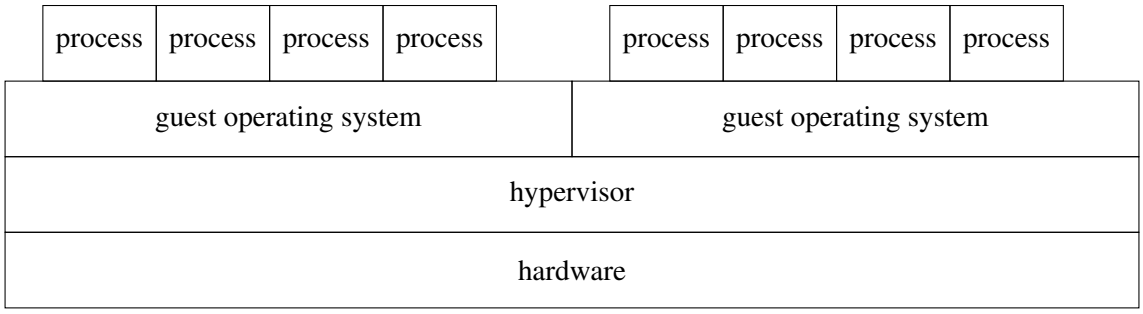

Figure 2. A type II hypervisor

What value does the register hold when the interrupt arrives? If the timer interval was set to h and the real processor core is capable of l loops in that time, then the guest counts αl loops. Our BogoMIPS measurement depends entirely on the workload of the host and how the host's operating system schedules threads. All such relationships between work done and time measured will suffer this distortion, to the detriment of our simulation.

Time dilation, introduced in [14], offers a limited capacity to address this problem. In one form of time dilation, we scale QEMU's measurement of real time by a factor β, so that a real interval h becomes an emulated interval βh. The relationship between thread time and scaled real time becomes $(\beta/\alpha)h$. With β we control the relationship between work accomplished by the virtual processor core and the passage of time in the emulator's hardware models. Indeed, we may cause the modeled hardware – timing sources, network devices, and so forth – to appear faster or slower in relation to the virtual core's processing power.

In practice, α is not a constant, but varies with the host's workload, the guest's workload, and the host operating system scheduler. Some mechanism to adjust β is needed to keep β/α a constant. This is the adaptive time dilation problem [21], the solution of which requires a feedback control in which α is measured and β adjusted to track some desired reference value for β/α. The approach in [21] very closely resembles a proportional control. The effectiveness of any time dilation scheme will depend on how well the control problem can be solved.

3.2 Specialized type II Hypervisors

The problem identified in Sect. 3.1 is rooted in the impracticality of controlling how the operating system's scheduler manages the computational workload. A type II hypervisor resolves this problem by placing the hypervisor between the guest operating systems and the host hardware. Now the hypervisor is analogous to an operating system and the guest operating systems are analogous to processes; the hypervisor manages hardware resources that are shared by several operating systems. Figure 2 illustrates this arrangement.

The open source Xen hypervisor is a popular choice for constructing simulators. A typical approach is to modify the hypervisor's scheduler to make it act as a future event list [44, 47]. We proceed with an analysis of the solution described in [47], which appears to be the most capable at present. The emphasis in [47] is on the simulation of mobile ad hoc networks and the modifications to Xen reflect that focus. Nonetheless, the solution could be extended into a general purpose simulation capability, at least conceptually. Therefore, we take some liberty in our description by omitting limitations of the simulator as it exists. Our supposition is that time and effort, and not some fundamental new insight, separates present capabilities

from what we envision. Consistent with this supposition, we assume a full set of simulated hardware devices associated with each guest operating system.

The Xen hypervisor is modified so that a time of next event is associated with each guest operating system. Each simulated computer comprises a single guest operating system and there is one additional guest operating system that hosts the network simulator. For brevity, the guests associated with our modeled computers will be referred to as guests and the guest holding the network simulator will be called the simulator.

The next event time for the simulator is the time stamp of the first pending discrete event. The time of next event for each guest is the smaller of the next hardware event that will impinge on it or a small slice of time to ensure forward progress when no external event is pending. A hardware event may be the arrival of a network frame (as described in [47]), an interrupt from a device, or some other activity by the simulated computer hardware associated with that guest.

The hypervisor proceeds just as a discrete event simulation would. It selects the simulator or guest that has the smallest time of next event. If a hardware event is pending, then that event is applied to the guest. The selected guest is permitted to execute instructions using the physical processor cores. The hypervisor allows the guest to execute until it issues an instruction that interacts with hardware or an event is pending for another guest or the simulator. At that time, the guest is evicted from the physical processor core.

Because the hypervisor has complete visibility into and control over the use of the physical processor cores, it can ensure that each instruction executed by a guest corresponds to an appropriate interval of simulation time. This solves the problem faced by QBox and QEMU/adevs of aligning time in the hardware models with time allocated to the the virtual processor cores. This event driven hypervisor meets nearly all of our requirements:

1. guest software may be executed without modification;
2. near native execution speeds contribute to rapid turn around;
3. time perceived by the guest matches simulation time;
4. the simulator may be easily linked with the federation run-time libraries; and
5. the virtual machines may be under-provisioned without impacting the simulation outcome.

In spite of its technical elegance, the new hypervisor faces difficult organizational barriers. First, it requires computer hardware to be dedicated for the modeling and simulation task. This it incompatible with a growing use of cloud computing to package “simulation as a service” [24]. Second, the cyber-security implications of a new hypervisor are likely to trigger stringent review and approval processes, the cost of which may not be justifiable given the perceived benefit to the organization in relation to existing, approved simulation solutions.

4. A NEW COMPUTER SYSTEM MODEL

A new computer system model has the advantage of not working around prior design decisions. The novel focus of our model on system of systems simulations makes this particularly attractive. We describe the major features of our novel design, sketch the issues involved in implementing models of hardware devices,

and review the mechanism by which the simulation clock and time perceived by the virtual processor cores are kept in alignment. Most features of the design will be familiar to persons already experienced with modeling computer systems, discrete event simulation, or both. The novelty of the design is in its particular use of established technologies and techniques to achieve a unique objective.

The new design and implementation has three main features that distinguish it from other computer systems simulators and virtualization tools. The first is construction of the model on top of an established simulation package. We use the adevs simulator, which offers support for participating in a federated simulation. This feature of adevs is documented elsewhere [25, 26, 27], and we do not discuss it further in this report.

Second, every aspect of our computer system simulator except the execution of machine instructions operates in simulation time. In this respect, our simulation model resembles the Bochs [38] model. Both Bochs and our model simulate only the functionality needed to run an operating system and applications without concern for detailed architectural elements that are transparent to most software. Both models also operate in simulation time, without reference to a wall clock. Indeed, elements of hardware models from Bochs open source code have been used in our new simulator. We differ from Bochs in two respects. First, Bochs was conceived and designed as a standalone simulation program; our new simulator operates as part of a federation. Second, Bochs uses binary translation to simulate an IA-32 microprocessor; we use the KVM to execute machine instructions.

The KVM virtual cores execute in their own threads, separate from the rest of the simulator. A primary concern is to ensure that the machine instructions per unit time allocated to a KVM thread and machine instructions per unit time in the simulator have a one to one correspondence. This is accomplished by having simulation time advance in tandem with time allocated for the KVM threads when they are active. In our present implementation, this relies on the "completely fair scheduler" that is used by the Linux operating system [17]. A detailed description of our microprocessor model is offered in Sect. 4.4.2. New features of the KVM that could offer a more robust solution to the synchronization problem are discussed in Sect. 5.

The third distinguishing feature of our simulator is the capability for multiple computer system models to exist within the same operating system process. This makes it possible to create a system model, which may comprise several computers, without the inefficiencies and complexities of inter-process communication. Moreover, if the other elements of this system model are realized with the adevs simulator, then the system model has intrinsic support for participating in a federated simulation.

The new computer system model is realized as a discrete event simulation expressed in terms of the Discrete Event System Specification (DEVS) [48]. DEVS (and, therefore, adevs) organizes a model into atomic and coupled (or network) components. An atomic component is an event driven state machine. In our computer system model, atomic models are used to realize an Advanced Host Control Interface disk controller (ahci), graphics card, programmable interval timer (pit), Local Advanced Programmable Interrupt Controller (lapic), Input/Output Advanced Programmable Interrupt Controller (ioapic), Peripheral Component Interface devices (pci), and other discrete hardware elements. Typically, one device data sheet is translated into one atomic model but there may be exceptions.

A coupled model comprises a set of component models and their interconnections. These components may be coupled and atomic models. In our computer system simulator, coupled models are used to create idealized data busses. Routing (coupling) within a bus model is based on memory address ranges for

memory mapped I/O, port addresses for port mapped I/O, and apic addresses for the apic bus.

A coupled model relays events between its components. In the computer system simulator, all communication between model components is via events, with four specific (and unfortunate) exceptions concerning the ahci drive controller, the graphics card, the pic and ioapic, and pci devices. The `Event` structure is consumed and produced by device models and routed by bus models. It contains a union that may be assigned one of seventeen types of events.

```
struct Event
{
  #define NO_EVENT -1
  // Type of event
  int type;
  // Which virtual core generated the event or should receive the event
  int core_id;
  union
  {
    #define EVENT_KVM_INT 0
    struct {
      unsigned irq;
      bool high;
    } kvm_gsi;
    #define EVENT_KVM_IO 1
    #define EVENT_ACK_IO 2
    struct {
      unsigned count;
      unsigned bytes;
      char* data_start;
      unsigned short port;
      bool read;
    } kvm_io;
    #define UART_16550_CHAR_ON_LINE 3
    char uart_16550_char_on_line;
    #define EVENT_KVM_MMIO 4
    #define EVENT_ACK_MMIO 5
    struct {
      unsigned long addr;
      unsigned bytes;
      char* data_start;
      bool read;
    } kvm_mmio;
    #define EVENT_GUI_KEY 6
    struct {
      unsigned key;
    } gui_key;
    #define EVENT_KVM_DISCARD 7
    #define EVENT_GUI_MOUSE 8
    struct {
      // pixel location change inside the gui window
      int dx, dy;
      // mm per pixel
```

```
      double mm_pixel_x;
      double mm_pixel_y;
      // bits show the state of the buttons
      unsigned button_state;
    } gui_mouse;
    #define EVENT_APIC_TO_APIC 9
    #define EVENT_MSI 10
    // Deliver the interrupt vector following an EVENT_ACK_INTR
    #define EVENT_APIC_TO_CORE 11
    struct {
      Bit32u vector;
      Bit64u dest;
      Bit8u delivery_mode;
      bool logical_dest;
    } apic_bus;
    // Set the INTR line to tell the core an interrupt is pending
    #define EVENT_INTR_TO_CORE 12
    // Core acknowledges the interrupt line
    #define EVENT_ACK_INTR 13
    // The interrupt has been injected. This is useful for MSIs
    #define EVENT_INTR_INJECTED 14
    struct {
      void* who; // Which PIC or APIC is this for
      Bit32u vector;
    } intr_line;
    #define EVENT_KVM_HALT 15
    #define EVENT_ETH 16
    struct {
      Bit8u* data;
      Bit32u* ref_cnt;
      unsigned bytes;
    } eth;
  };
};
```

Of these events, six are relevant to almost all models of devices.

1. `EVENT_KVM_INT` is an interrupt generated on an irq line that is connected to the pic and ioapic. This event is generated by a device model to send an interrupt to the microprocessor.

2. `EVENT_MSI` is a message signaled interrupt (msi) produced by a pci device.

3. `EVENT_KVM_IO` is received when software issues a port I/O instruction directed at a port claimed by a device model. An `EVENT_ACK_IO` is sent by the device in response after completing the I/O operation.

4. `EVENT_KVM_MMIO` is received when software issues a memory mapped I/O instruction directed at a memory address claimed by a device model. An `EVENT_ACK_MMIO` is sent by the device in response after completing the I/O operation.

Figure 3 shows the top most level of the computer system model. This coupled model acts as a system bus that routes the six types of events listed above. At this level of the model, the processor is presented as a

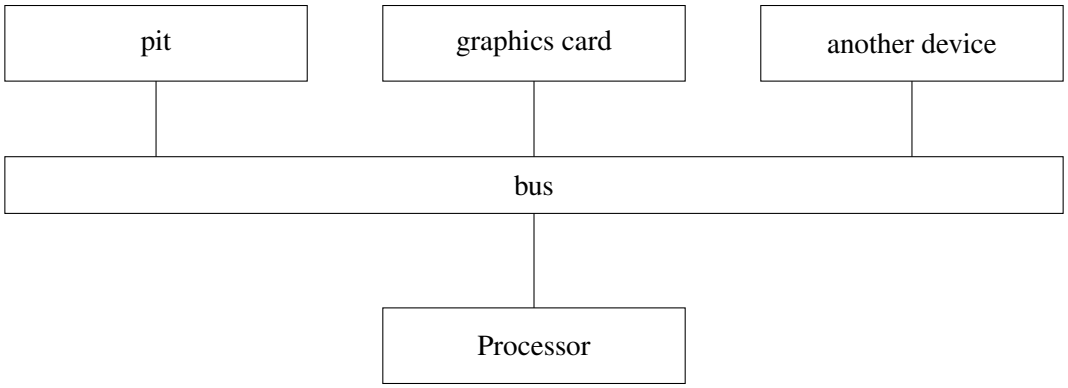

Figure 3. Elements of the coupled model that acts as a system bus.

monolithic entity that issues port and memory mapped I/O requests and that reacts to interrupts.

Devices other than the microprocessor are atomic models. The common mechanics of consuming and producing events are realized by the `DiscreteDevice` class, which extends the adevs `Atomic` class. Device models are created by extending the `DiscreteDevice` class. The derived class implements virtual methods that are invoked by the base class to handle device specific aspects of an I/O request. The `DiscreteDevice` class also offers functionality for scheduling local timers, generating interrupts, and claiming port and memory regions that are used by software to communicate with the device. The interface for using these aspects of the `DiscreteDevice` class is shown below.

```
class DiscreteDevice:
  public adevs::Atomic<Event,adevs::sd_time<long> >
{
  public:
    // If a processor is provided then this device can request physical memory access
    DiscreteDevice(Processor* proc = NULL);
    ~DiscreteDevice();
    void gc_output(EventBag& gb){}
    void delta_int();
    void delta_ext(adevs::sd_time<long> e, const EventBag& xb);
    void delta_conf(const EventBag& xb);
    void output_func(EventBag& yb);
    adevs::sd_time<long> ta();
    bool pio_in_range(Bit32u port);
    bool mmio_in_range(Bit64u addr, int core_id = ANY_CORE);
    virtual std::string name() = 0;
    // Register a port to receive io
    void register_pio(Bit32u port);
    // Remove a port
    void unregister_pio(Bit32u port);
    // Register a memory window beginning at the given address
    void register_memory(Bit64u start_addr, unsigned long bytes, int key,
        int core_id = ANY_CORE, int flags = 0);
```

```
  // Remove the memory window starting at the given address
  void unregister_memory(int key, int core_id = ANY_CORE);
  void unregister_all_memory();
protected:
  // Override to read from mapped memory io. Result is copied to data.
  virtual void read_mmio(Bit64u addr, char* data, unsigned long bytes) = 0;
  // Override to respond write to memory mapped io. Data to write is in data.
  virtual void write_mmio(Bit64u addr, const char* data, unsigned long bytes) = 0;
  // Port mapped reads
  virtual void read_pio(Bit32u port, char* data, unsigned bytes) = 0;
  // Port mapped writes
  virtual void write_pio(Bit32u port, const char* data, unsigned bytes) = 0;
  // Handle the expiration of a timer
  virtual void timer_expires(unsigned which) = 0;
  // Handle miscellaneous events that do not have a specific handler
  virtual void handle_event(Event x) = 0;
  // Generate an event
  void generate_event(Event x);
  // Set a timer to expire at the given duration
  void set_timer(unsigned which, long duration);
  // Cancel a timer
  void cancel_timer(unsigned which);
  // Set an irq line
  void set_irq(unsigned irq, bool level);
  // Get the current time in nanoseconds
  long now() const { return t_now.real(); }
```

4.1 Programmable interval timer

The programmable interval timer (pit) offers a relatively compact example of how the `DiscreteDevice` class is extended to model a specific device. The pit is the timing device used in 1980's era IBM PC compatibles. The pit uses a 1.193182 MHz clock signal to implement three timer numbered 0, 1, and 2. Remarkably, modern Linux operating systems still use timer zero as a clock while booting, and it is also used by the SeaBIOS BIOS software. Timers one and two are anachronisms that we omit from our model.

The pit has several modes of operation, but the general operating principles are the same in each. Software sets the pit's mode and an interval counter via port I/O on ports 0x40 and 0x43. The interval counter is decremented at each clock tick. Upon reaching zero, the pit generates an interrupt on irq line zero. Our model updates the interval counter upon a read, write, or at the moment it reaches zero. Upon modifying the counter, we scheduling a timer event for the future instant when the clock will drive the counter to zero.

An abbreviated source code listing for our model of the pit is given below. Ports 0x40 and 0x43 are claimed by the pit in its constructor. The receipt of a `EVENT_KVM_IO` events causes the base class to invoke the `read_pio` or `write_pio` method as appropriate. The pit does not use memory mapped I/O, but registration of the memory region and the response to I/O requests would be nearly identical to the port I/O that is illustrated.

Use of the timer and irq services provided by `DiscreteDevice` are apparent at several locations in this source code listing. Calling the `set_irq` method causes the base class to schedule a `EVENT_KVM_INT` event, which will be relayed to the microprocessor. When `set_timer` is called, the base class responds by invoking `timer_expires` at the specified moment in the future. A call to `cancel_timer` revokes the preceding `set_timer` request. Every device model has access to four of these timers, numbered 0-3, and not to be confused with the pit timer, which is a feature of the device being modeled.

```
/** The programmable interval timer */
class PIT:
  public DiscreteDevice
{
  public:
    PIT();
    std::string name() { return "pit"; }
  protected:
    void timer_expires(unsigned which);
    void read_pio(unsigned port, char* data, unsigned bytes);
    void write_pio(unsigned port, const char* data, unsigned bytes);
    void read_mmio(Bit64u,char*,unsigned long){}
    void write_mmio(Bit64u,const char*,unsigned long){}
    void handle_event(Event x){}
  private:
    Bit16u
      counter,
      reload,
      latch,
      operating_mode,
      access_mode,
      latch_read_count,
      counter_read_count,
      reloaded;
    Bit8u status_latch;
    bool read_back, null_count_flag, out_gate, status_latched;
    // Keep track of elapsed nanoseconds between updates to the tick counter
    long tick_update_ns;

    void update_counter();
    long load_counter();
};

PIT::PIT():
  DiscreteDevice()
{
  // Only channel 0
  register_pio(0x40);
  register_pio(0x43);
    /** Some omitted initialization stuff here */
}

void PIT::timer_expires(unsigned which)
{
```

```
  /* Counter has reached zero */
  counter = 0;
  tick_update_ns = now();
  /* We support modes 0 and 2 only */
  switch(operating_mode)
  {
      /** Some omitted modes here */
    case 2:
        // Pulse the irq line
      set_irq(0,false);
      set_irq(0,true);
      // Reset the counter and schedule a timer event
      // for when the counter reaches zero
      set_timer(0,load_counter());
      break;
        /** More omitted modes
  }
}

void PIT::read_pio(unsigned port, char* data, unsigned bytes)
{
    // Decrement the counter to account for elapsed time
  update_counter();
  switch(port)
  {

    case 0x40:
        /** Omitted stuff here to handle read at port 0x40 */
        break;
    case 0x42:
        // Read on port 0x42 always returns 0xff because
        // our model doesn't implement any of the 0x42 features
      *data = 0xff;
      break;
  }
}

void PIT::write_pio(unsigned port, const char* data, unsigned bytes)
{
    // Decrement the counter to account for elapsed time
  update_counter();
  Bit8u cmd;
  switch(port)
  {
    case 0x40:
      switch(access_mode)
      {
          /** Omitted access modes. We showcase just mode 1 here. */
        /* Write to the reload register */
        case 1:
          assert(bytes == 1);
```

```
          reload &= 0xff00; // Clear the lower bits
          reload |= 0x00ff & (Bit16u)(*data);  // Set the lower bits
          reloaded = 0x03;
          break;
      }
        }
        /** More omitted cases */
  }
}

void PIT::update_counter()
{
  /* Account for the elapsed time */
  const unsigned ticks = double(now()-tick_update_ns)*freq_ghz;
  if (ticks > 0)
  {
    tick_update_ns = now();
    if (ticks < counter) counter -= ticks;
    else counter = USHRT_MAX - ticks%USHRT_MAX;
  }
}
```

4.2 Peripheral Component Interface devices

Peripheral Component Interface (pci) devices are modeled in the same manner as other devices by inheriting from the `DiscreteDevice` class. However, to facilitate management of the device by the pci bus controller, a pci device model must also inherent from the `PCIDevice` class. This class offers services to register memory mapped I/O regions using the pci base address registers, access the device configuration space, and generate message signaled interrupts.

The term bus in pci bus controller is misleading. The pci bus controller is not a bus that routes data, and it is not implemented as a coupled model. Rather, the pci bus controller is a device model called `PCIBus` that inherits from `DiscreteDevice` and `PCIDevice`. From the perspective of software using the pci sub-system, a primary use of the bus controller is to discover and configure pci devices.

The `PCIDevice` and `PCIBus` classes conceal the first instance of model components communicating directly, rather than through events. The pci bus controller and its attached devices share a memory region called the configuration address space. Each device is assigned a region within this address space. Information concerning the identity and functionality of the device is stored here, as are several other important items, such as the base address registers that are used to map the device into a memory mapped I/O region.

The pci specification requires that the configuration address space for every device be accessible through the pci bus controller. In our model, this means that memory mapped I/O or port I/O operations directed to the `PCIBus` may need to access the configuration data for any of its attached devices. At the same time, a device may read and write in its own configuration address space.

To accommodate this shared space in our model, the `PCIBus` has a pointer to each device for which it is

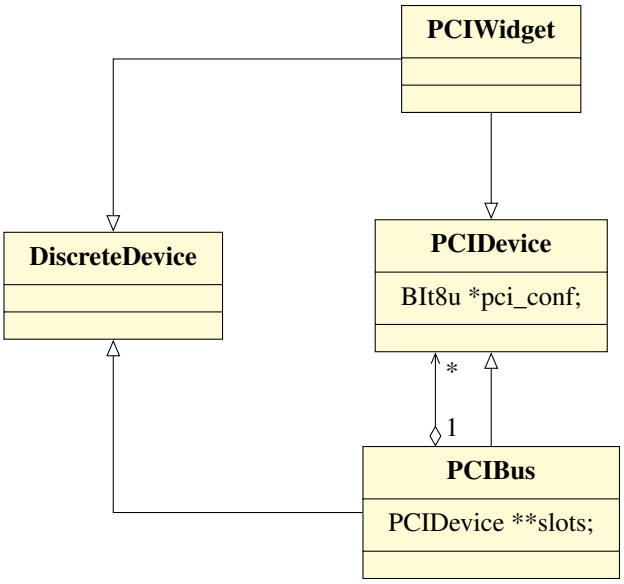

Figure 4. Class relationships between the pci bus controller and a pci device, called the widget, assigned to the controller. The pci bus controller and widget may communicate implicitly through the widget's configuration address space.

responsible. Figure 4 shows a fictitious `PCIWidget` model that illustrates the the class relationships. These pointers are used to access the device's configuration address space. Hence, implicit communication occurs when the shared configuration space is accessed by the controller or a device.

4.3 Direct memory access

Model of devices and the microprocessor model may use direct memory access to communicate without exchanging `EVENT_KVM_MMIO` and `EVENT_ACK_MMIO` events. There are two mechanisms for direct memory access. Both require the `Processor` object that models the microprocessor be passed to the constructor of the `DiscreteDevice` base class. This capability has two purposes. First, it offers a significant performance improvement when memory mapped I/O is very frequent. Second, it can be used to model direct memory access (DMA) transfers data between a device and microprocessor.

A convenient aspect of the KVM is that the simulated random access memory used by the virtual cores exists in the address space of the operating system process that constitutes our simulation model. Similarly, the threads assigned to the KVM core are indistinguishable, for all practical purposes, from other threads in our process. The KVM limits virtual cores to accessing that part of the process address space set aside to simulate the memory system of our virtual machine. However, any part of our program that is not simulating a KVM virtual core may access any part of the processes address space, including the simulated memory of the virtual machine.

This arrangement is illustrated in Fig. 5. The upper rectangle depicts the timeline of two threads with shaded parts showing where a KVM virtual core is being simulated. During the shaded intervals of time, the thread may only access the shaded KVM simulated memory region. In the non-shaded intervals, the entire memory region is accessible to the thread. To communicate with a device, the virtual core writes

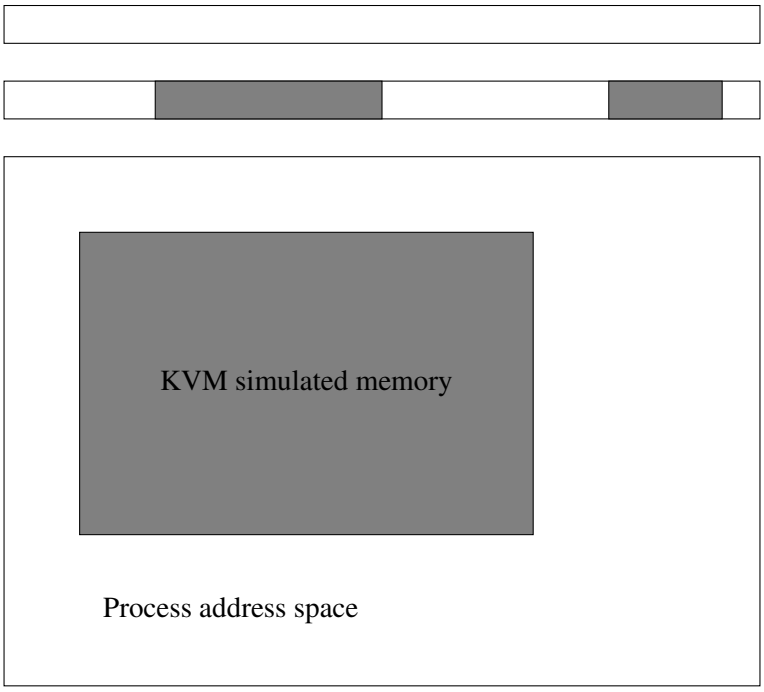

Figure 5. A map of the KVM virtual cores and their simulated memory within the operating system process that contains our simulation model.

data to the KVM simulated memory and the device model reads that data. In the other direction, the device model may write to the KVM simulated memory and the virtual core reads that data[†].

The first mechanism for direct memory access is to supply the flag `REGISTER_PHYSICAL_MEMORY` to the method `register_memory` of the `DiscreteDevice` class. A pci device may use this flag when calling `init_bar` to register a memory mapped I/O region that is associated with one of its base address registers; the `init_bar` method then supplies this flag to `register_memory`.

The `REGISTER_PHYSICAL_MEMORY` flag instructs the `Processor` model to allocate within the KVM a region of memory that is accessed by software without generating requests for memory mapped I/O. The device model can obtain a pointer to this memory region from the `Processor`, which is used to read and write in the region. The graphics card model uses this mechanism to access the memory region that holds graphics data, which is shown on a user's display. In this way, the software may update the graphics data without the overhead of issuing memory mapped I/O requests. The modeled graphics card polls this memory region in simulation time at a rate consistent with the refresh rate of the simulated graphics terminal, updating the display to reflect changes written by software since the previous refresh.

The second mechanism for direct access memory is to obtain from the `Processor` a pointer to the memory region used by the KVM to simulate the random access memory of the virtual cores. This method is used by the ahci drive controller model to implement the DMA data transfers required by the ahci specification. A sketch of the procedure for writing data to the disk drive gives some insight into how DMA is simulated with this mechanism.

[†]I've ignored important details about how this is implemented and the caveats these details imply. But the main point is valid. Communication between modeled devices and virtual cores is possible without using events.

The ahci is a pci device that receives commands via memory mapped I/O in an address space indicated by its base address registers. The I/O address space used to issue commands to the ahci does not overlap the address space used for random access memory by the virtual core. To initiate a write to the drive, the software does the following.

1. Reserves a region of random access memory and places the data to be written into this region.
2. Issues a write command to the ahci by using memory mapped I/O. Among other items of information, the command contains (1) the start address of the random access memory region that contains the data and (2) the number of bytes of data.
3. The ahci model fetches the data from the indicated region in the random access memory of the `Processor` and then writes that data to a file, which simulates the physical disk drive.

In an actual computer, several pieces of electronic equipment are needed to perform the data transfer from random access memory to the ahci device. However, these electronics are transparent to the software and so we omit them from our model. This simplification avoids the complexities of simulating the actual process of a DMA transfer, replacing it with a simple, three step procedure.

4.4 The processor

The `Processor` is a coupled model that contains atomic models realizing each of the virtual cores, the lapic attached to each core, and the ioapic and pic. Event routing by the `Processor` simulates the apic bus that connects the local apics and ioapic; it models the irq lines connected to the pic and ioapic; it connects each lapic to its processor; and it routes memory mapped and port I/O events between the virtual processor cores and the system bus model and between the virtual processor cores and the other processor elements. The lapic, ioapic, and pic are derived from the `DiscreteDevice` class and are realized in the same manner as other device models. The virtual processor core is unique in that it does not derive from `DiscreteDevice`. Its design and implementation will be addressed later.

4.4.1 Interrupts

Figure 6 shows how the devices models within the `Processor` are connected for the purpose of interrupt handling and apic to apic communication. An irq line is set when a device attached to the system bus generates an `EVENT_KVM_INT`. This event enters the `Processor`, which routes it to the pic and ioapic. If the pic is configured to handle the interrupt, then the pic begins a sequence of events that will culminate in the delivery of an interrupt vector to a core. If the ioapic is handling the event, then the ioapic generates an `EVENT_APIC_TO_APIC` to deliver the irq data to an lapic. The lapic then begins the sequence of events that delivers an interrupt vector to the lapic's core.

An message signaled interrupt (msi) is created when a pci device generates an `EVENT_MSI` event. This event enters the `Processor`, where it is routed to directly to an lapic. The lapic then initiates the sequence of events needed to deliver an interrupt vector to the lapic's core.

The sequence of events by which a pic or lapic delivers an interrupt vector to the core has five steps.

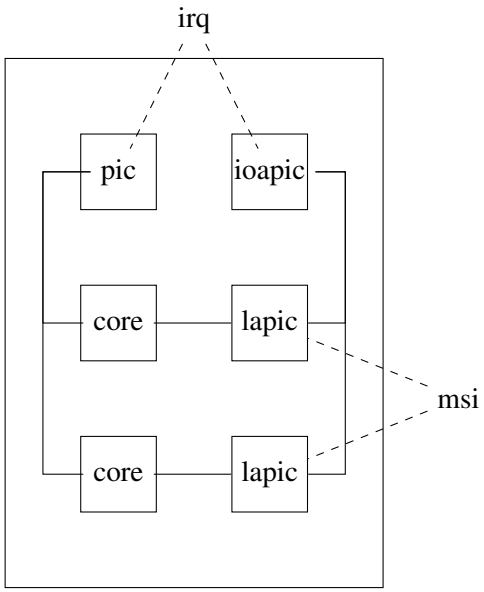

Figure 6. Interrupt routing in a model of a processor with two virtual cores.

1. The pic or lapic indicates a pending interrupt by generating an `EVENT_INTR_TO_CORE` event, which is routed to the virtual core.
2. When an interrupt window opens at the virtual core, it responds to the pic or lapic with an `EVENT_ACK_INTR`.
3. Upon receipt of `EVENT_ACK_INTR`, the pic or lapic supplies the interrupt vector via an `EVENT_APIC_TO_CORE` event.
4. The core model injects the interrupt into the KVM thread, which initiates the software interrupt handler. The core indicates that the interrupt has been received and injected by generating an `EVENT_INTR_INJECTED` event.
5. Upon receipt of the `EVENT_INTR_INJECTED` event, the pic and lapic repeat this process if more interrupts are pending.

The fourth and final case of direct communication between model components involve the pic and ioapic. The ioapic may operate in several modes, which are set by the guest operating system as it boots. One of these modes requires the ioapic to use the mapping of irq line to interrupt vector that, in our model, is maintained by the pic. It is a matter of convenience to have the ioapic model keep a pointer to the pic which is used to access the map. The present intent is to remove the direct coupling in some future version of the model.

4.4.2 Virtual core

The `Core` class is an adevs `Atomic` model that responds to interrupts and executes machine instructions. Associated with each `Core` object is a thread that executes a KVM virtual core. There are two parts of the virtual core model. One part is a pure simulation model acting solely on simulation events, just like any

other hardware model. The other part is a KVM thread that executes machine instructions. This threaded part originates requests for memory mapped I/O, port I/O, and other events that are gathered up by the simulation part of the core model to be resolved in simulation time.

It is useful to distinguish the simulator thread, which has the adevs simulator, and the core threads that execute machine instructions for the guest software. The simulation thread has a single instance of an adevs `Simulator` object that orchestrates the execution of the hardware model components, including the simulation parts of the core. The core threads manage a KVM virtual CPU, and these may be active or idle. Active core threads have machine instructions to execute and no pending I/O requests. Every core thread begins in the active state. A core thread may become idle in three ways.

1. The software has requested port I/O or memory mapped I/O and the acknowledgement of completed I/O is pending.
2. The software has issued a halt instruction, indicating it has no more work to do until an interrupt arrives.
3. An `EVENT_APIC_TO_CORE` is pending from the pic or lapic attached to the virtual core.

We begin an iteration of the computer system simulator with a maximum allowable time advance h. Each core thread that is active is permitted to begin execution. In parallel, the simulator thread advances time for the adevs models at the same pace that Linux's `CLOCK_THREAD_CPUTIME` advances. This clock reports the amount of time allocated by the Linux operating system for the execution of the thread that requests the time.

The Linux operating system uses a "perfectly fair scheduler" that seeks to ensure every thread at the same priority receives equal amounts of execution time. Assuming the adevs simulation can keep pace with the thread clock, the scheduler ensures that time for the simulation clock paces the execution time allocated to the virtual CPU cores. This resolves the problem identified in Section 3.1 with respect the QBox and QEMU when the available physical hardware is less capable than the virtual hardware. Moreover, because simulation time and thread time maintain a constant 1 to 1 relationship, a non-adaptive time dilation scheme is sufficient to model a core that is faster or slower than the hardware actually available.

While advancing the simulation clock, the simulator thread monitors the status of the core threads. Should a core thread become idle, it is not scheduled for execution until switching back to the active state through the actions of the simulated hardware. Upon becoming active, the core thread is permitted to execute. If all of the core threads are idle, then the simulator thread no longer needs to pace the virtual core thread. In this case, the simulator thread advances the adevs simulator as fast as possible, only resuming its paced execution when one or more of the core threads become active. When the adevs simulation clock reaches its maximum permissible time advance, the core threads are halted and the simulation thread waits for a new time advance grant from the federation run-time. Upon receiving the grant, execution resumes as described above.

5. KERNEL VIRTUAL MACHINE

The KVM appears to have been created without considering its use as a tool in simulations. When seeking to use KVM as an accelerator for a simulation model that is not tied to a wall clock, we find that two of its features pose a particular challenge. These are the use of software signals to evict the KVM thread from the

physical processor core and the use of wall clock time to update the Time Stamp Counter (tsc) register of the virtual processor.

5.1 The software signal problem

To initiate the execution of machine instructions on the virtual core, the thread managing that KVM virtual core issues a `KVM_RUN` command via the Linux kernel's `ioctl` function. The `ioctl` call will return of its own accord if the guest software issues an instruction that requests I/O, a window opens to inject an interrupt, and under various other circumstances that require service by a simulated hardware device. If we wish to stop execution of the virtual core before it returns on its own, the simulation thread must issue a software signal using the Linux kernel's `signal` facilities.

The need to issue a signal raises two problems. The first problem is minor, but irksome. In Sect. 4.4.2 we describe the unsatisfactory solution of relying on the fairness of the Linux thread scheduler to keep a 1 to 1 relation between simulation time and time allotted to the virtual cores. We assume that if t units of time have been allotted to the simulation thread, then t units of time have also been allotted to the active guest threads.

If an interrupt should be injected into the processor model, then it is necessary to stop the virtual core before carrying out the exchange of events described in Sect. 4.4.1. Suppose this interrupt will occur k units of simulation time into the future. To know when k units of time have have elapsed, the simulation thread runs in a loop polling the thread clock. The number of events executed by this loop is typically small compared to the number of iterations and so we are wasting considerable computational resources.

A more serious problem is posed by the operating system and its handling of signals. A signal is issued under two circumstances. At the end of a simulation time quantum, as measured by elapsed thread time, the simulation thread issues a signal to halt the active virtual cores. A signal is also issued when an interrupt must be injected into an active virtual core; this occurs when the core's model receives an `EVENT_INTR_CORE` event.

A signal issued by the simulator is queued by the operating system to be delivered "when it gets to it". In practice, the delay is quite short, but no guarantees are offered concerning the timing of the delivery. Therefore, the virtual core may receive substantially more time than it should. Because this aspect of signal delivery is outside of our simulator's control, it degrades our ability to manage the relationship between simulation time and machine instructions executed by the active virtual cores.

5.2 The time stamp counter problem

The tsc register exists on virtually all modern processors used to create workstations, desktops, and servers. The purpose of the tsc is to permit the software to track time precisely, efficiently, and easily by reading it from a register. In many circumstances, this means that time can be tracked without the bothersome use of interrupts from an off chip device. When an interrupt is needed, the tsc may be configured to trigger an interrupt from the lapic at programmable intervals.

At present, the KVM does not offer a capability for the simulation programmer (or virtual machine programmer) to control the value reported when the guest software reads the tsc register, nor is it possible

to be notified by the KVM if the tsc should induce the lapic to generate an interrupt. Rather, the tsc is managed transparently by the KVM. Advancement of the counter value proceeds at a rate linked to the passing of wall clock time. Similarly, the in kernel lapic model, which is managed as a real time device by KVM, must be used if the tsc is to induce the lapic to generate interrupts.

This solution is attractive to the builders of virtual machines. The tsc may be read frequently by the guest software, and so computational efficiency is greatly improved by not having the virtual core cease execution so that the tsc read can be resolved. Similarly, interrupts from the lapic may occur with great frequency, and simulation of the lapic within KVM allows the virtual core to avoid halting when these interrupts must be injected.

However, these features of the KVM make the tsc unusable by a simulation model. Unfortunately, the tsc is the preferred timing device for nearly all modern operating systems. Our present version of the computer system simulator reports to the guest that the virtual core does not possess a tsc register, forcing it to use some alternative, simulated timing source. The power management timer is a popular choice, which is simulated as part of the q35 chip set model.

5.3 KVM remedies

The signal problem could be resolved if it was possible to schedule an exit of KVM when the `KVM_RUN` command is issued. Specifically, we would like to instruct KVM to exit after executing N instructions, and that upon return of the `ioctl` call KVM would inform us of how many instructions were actually executed, which could be less than N if I/O was requested by the guest, an interrupt window opened, or under some other circumstance that would normally result in an exit. Suppose we have this feature, that the desired average instruction rate is I machine instructions per second, and that the next simulation event (e.g., an interrupt) is scheduled for k seconds into the future. The simulator would issue a `KVM_RUN` command with a request to return after no more than Ik instructions have been executed.

Upon the return of the `ioctl` call we find that $M \leq N$ instructions have been executed. The corresponding amount of simulation time that has elapsed is M/I. The simulation thread advances the simulation clock by this amount, issuing an interrupt if one was scheduled by a hardware model. This use of the proposed feature removes our dependence on the Linux scheduler, permits precise management of the time allotted to virtual cores, and the simulation thread no longer needs to poll the thread clock.

The tsc becomes useful to the simulator if the `ioctl` call returns when the tsc is accessed by the guest. Upon being notified that a read of the tsc is occurring, the simulator would supply a counter value consistent with the present value of the simulation clock. Moreover, because time reported by the tsc and simulation time are precisely synchronized, the simulator can schedule interrupts from the lapic to coincide in simulation time with the advancement of the tsc.

6. CHALLENGING HARDWARE

The ability to model hardware rests on two key assumptions. First, documentation for the hardware is available and sufficiently detailed. Second, the hardware model can be simulated at a rate sufficient to not seriously impact simulation turn around times. Graphical Processing Units (GPUs) provide an example of where both assumptions are violated. GPUs are increasingly common as accelerators for software that

needs to achieve a high rate of floating point operations. Numerical linear algebra, machine learning, digital signal processing, and a host of other computational tasks make use of GPU technology.

Documentation for the PTX instruction set forms the basis for GPU simulators designed to explore architectural questions, and these simulators offer insight into the computational challenges posed by a software model of GPU hardware. The GPU architecture is entirely dissimilar from a conventional microprocessor. The GPU uses thousands of threads executing in lockstep as they apply identical instructions to disparate data. For specific types of data processing workloads, the GPU architecture offers tremendous processing rates, which cannot be matched by a conventional processor. Because of this, simulation models of a GPU execute thousands of times slower than their real world counterparts; see, e.g., the benchmarks reported in [22]. These slow simulation speeds make it impractical to model a GPU for use in system of systems simulations.

Apart from problems of computational speed, commercial GPU technology is proprietary and no documentation is provided by the vendor describing the software/hardware interface. The driver software for a GPU is supplied as a compiled binary object, and this is the sole means for applications to interact with the hardware. Existing models of a GPU overcome this problem by relinking the GPU application software with their own versions of the application programming interface; see, e.g., the solution described in [34]. Absent documentation, it is infeasible to model the low level operations used by the vendor software to access the GPU, which prevents use of the unmodified GPU driver software in a simulation model.

Should the software system require access to a GPU, it becomes necessary to offload work targeted at the GPU to real hardware while simulating the rest of the computer system. A single hardware GPU can be made available to a single virtual machine by using PCI pass through. This method models memory mapped I/O targeted at the GPU by forwarding simulated access requests to the physical device. Mature support for PCI pass through is supplied by the Linux operating system, and it is relatively uncomplicated to add this feature to a computer system simulation model. However, a one to one mapping of simulated computer to real GPU hardware may present other challenges, particularly if several computer system models with GPUs are needed for a simulation run.

Increasingly, GPU vendors are offering support for several virtual machines to share one or more physical GPUs; e.g., the NVIDIA vGPU product [28]. In principle, technology that enables hardware sharing for virtual machine applications could support hardware sharing between simulated computers. If the simulation will be executed on dedicated hardware, and not in a virtualized cloud, then existing commercial offerings may be adequate. Otherwise, a lack of support for nested virtualization in these products limits the circumstances where they can be used to support simulations.

Using real hardware in place of simulated hardware poses validation problems relating to the mismatched passage of wall clock time for the hardware and simulation time for the rest of the model. The issue is very similar to the one raised in Sect. 3.1. If the GPU does not constitute a performance bottleneck in the software system, then the fact that the GPU is operating in real time rather than simulation time may not pose a problem. The GPU will appear to be overpowered when the simulation clock is advancing slowly in relation to the wall clock. If, in operation, the time required to complete a job submitted to the GPU is negligible, then an apparent acceleration is unlikely to cause the simulation to deviate significantly from behavior that would be observed in operation. However, if the GPU is a performance bottleneck for the application, the changes in its apparent performance may lead to significant deviations from operational behavior, thereby invalidating the model.

Unfortunately, the workload offered by an application to the GPU is likely to be determined by the simulation scenario. Hence, validation of the computer system simulator must be specific about the scenario to the extent that it influences the GPU workload. Another complication is the possibility of overtaxing the GPU because it is being shared across scenarios. In this case, the workload offered by a particular scenario may not exceed the validation limits, but the collective workload offered by scenarios running in parallel could invalidate the simulation results. These issues must be carefully considered when real hardware must be used in the simulation model.

7. COSTS

A large part of the motivation for a new computer system simulator is economic, and now we revisit the cost of building the simulator. In its present form, the simulator is capable of booting the Linux operating system and running applications. It includes models of an Ethernet card and serial port to simulate connectivity with other models of computer systems or hardware. The sloccount [45] program reports 11,691 software lines of code if we include just the source code for the computer model, excluding test cases and the adevs simulation engine. The simulator is implemented in the C++ programming language.

Without information concerning a similar level of effort metric for building a simulation model of a "typical" software system, we cannot offer a direct comparison of hardware and software modeling. Nonetheless, the basic COCOMO model [5] suggests that 11K software lines of code should require about 30 person-months of effort. The first commit to the software repository for our computer system simulator was two years ago from the time of writing and involved one developer working approximately half time, which places the level of effort at 12 person-months. However, our simulator was produced as research software and is presently short on quality assurance activities. Suppose we cut the difference and assume a level of effort of 21 person-months, or approximately 2 person-years.

The U.S. Department of Defense is the largest user of modeling and simulation technology in the United States, and so a cost estimate for organizations in the defense industry may find the quickest comparison with other, publicly available modeling and simulation expenditures[‡]. The average salary for software engineer working in the defense industry is $87,358 annually [12]. If we assume a rather high overhead rate of 100%, then two years of development would cost approximately $350,000. Of course, this does not account for the lifetime costs associated with maintaining the model and its evolution to account for new hardware. However, if our assertion that hardware evolves more slowly than software is true, then we may expect lifetime costs for the hardware model to be smaller than for the software model, all other things being equal.

The Department of Defense Software Fact Book [7] offers two points of comparison by which to judge the effort expended on a model of hardware. The first is a measure of Equivalent Source Lines of Code (ESLOC) and the second is person-months effort. The vast majority of the software in our model is new, and so our software lines of code metric overlaps reasonably well, but not precisely, with the ESLOC measure. However, the 212 projects analysed in the fact book reflect several domains of application, and not just modeling and simulation. Software lines of code is a notoriously poor measure of complexity and, therefore, of cost, which can only be made worse if we look across domains of application. Person-months is a better metric in this regard, but the data available do not allow us to normalize this figure specifically

[‡]Although costs for "small" modeling and simulation efforts, like the one described here, are difficult to find.

for M&S applications. Nonetheless, anecdotal evidence and the author's own experience with M&S software suggest that a comparison with average ESLOC and person-months presented in [7] is not entirely devoid of information.

The average ESLOC reported in [7] for a software system is 57, 199 and the median is 28, 598. By this metric, our computer system simulator is 20% the average size and less than half the median size. For the purpose of analyzing level of effort, the software projects are sub-divided into four domains of application: real-time, mission support, engineering, and automated information systems. Modeling and simulation could fall into either mission support or engineering. The average person-months effort reported for mission support software is 127, and for engineering software the average level of effort is 215 person-months. The estimated level of effort for our simulator is 2 person-years, or 24 person-months. This is 11% of the average effort of engineering software and 18% the average effort of a mission support application. The broad picture painted by these numbers is that a computer hardware simulator sufficient for system of systems simulations is a relatively inexpensive undertaking.

Model validation and accreditation may also offer opportunities for cost reduction [33]. By using the software "as is" it becomes possible to validate the hardware model in relation to a particular software system by checking that the software satisfies its test and accreditation requirements when executed on the simulated hardware. This can be done using existing testing resources for the software system, and so largely eliminates the cost of designing and applying new verification, validation, and accreditation material that would be needed for a new model. Moreover, it would offer very strong evidence of the model's operational validity [36] by showing that the software as a model and the software in operation are indistinguishable.

Technical and organizational considerations by themselves are sufficient to merit a new model, as we have argued in Sects. 2. and 3. We offer an economic incentive now. The modifications to QEMU created by QBox, in both of its variants, and QEMU/adevs were not incorporated into QEMU's main line of development. Neither have any other efforts to modify QEMU for use as a simulator. Similarly, modifications of the Xen hypervisor to support simulation have not been adopted as a permanent feature. All of these proposed modifications have been available for several years and are, at least in some cases, readily available as patches to the version of the software package that was modified. The caveat concerning version is important; continuing development of the parent virtualization tool means that these patches must be maintained over time. This has not been done, which suggests (1) simulation is not useful to most users of virtualization tools and (2) long term support for a simulation capability is non-trivial.

The consequences of (1) and (2) are that the modified virtualization tool becomes, for all practical purposes, a new software package that must be maintained and evolved separately from the original line of development. To grasp the magnitude of this endevour, Figure 7 compares level of effort measured in ESLOC and labor hours across the engineering software and mission support categories from the Department of Defense Software Fact Book [7], our new simulator, and the QEMU and Xen software packages. The latter measurements of lines of code are obtained with sloccount [45] and the labor hours are estimated using sloccount's built in level of effort estimation tool, which is the standard COCOMO model.

The time and effort needed to maintain and evolve QEMU or Xen is staggering. It is commonly estimated that 60% of software engineering activities involve maintenance. If we use the measurements in Fig. 7 as a baseline, then tens of thousands of hours would be needed to pair down, fix, and modify millions of lines of code as our simulation needs evolve. Worse still, most of that code will be irrelevant to our interests – for example, support for user mode emulation in QEMU – but nonetheless drives up our costs, even if only to

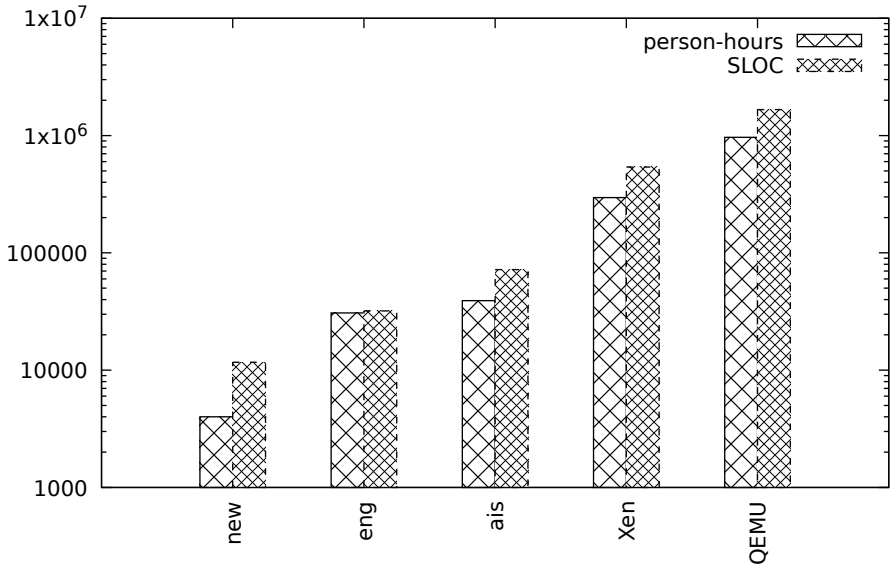

Figure 7. Measures of level of effort. The x-axis labels are: new for the new simulator, eng for engineering software, and ais for information systems.

remove the feature. This level of effort for Xen and QEMU as virtualization tools is justified by a massive base of users, a smaller but still substantial number of user-developers that contribute to the software, and the commensurate financial interests of large corporations with businesses intimately tied to these tools [3]. Such efforts are unlikely to be justifiable for a simulation tool.

In Sect. 5.3 we discussed new features for the KVM that would facilitate its use as a simulation tool. KVM also has a massive user base interested almost exclusively in virtualization (albeit, indirectly as component of several virtualization systems), and so efforts to modify it for simulation may result in a new software package that evolves separately from its parent. Consequently, future efforts to augment KVM for simulation must address the question of whether to reuse KVM or build anew.

8. LOOKING AHEAD

Modeling a modern, high performance processor core presents a challenge that cuts across simulation objectives. The KVM addresses two difficult problems. The first is achieving an acceptable rate of execution. The second is to avoid modeling a large, complicated instruction set. However, technical drawbacks inherent in the design of KVM as a virtualization tool limit its usefulness in simulations. To correct these deficiencies, modifications to KVM would be necessary. However, the probable need to support a new fork of the KVM software and its inclusion of significant features not useful for simulation diminishes the attractiveness of KVM as a solution.

In many respects, simulation is much simpler than virtualization. Performance of the simulator is a factor only to the extant that it negatively impacts the turnaround time of systems engineering studies. Time keeping is greatly simplified by tracking a simulation clock instead of a wall clock. These simplifications in our requirements suggest a design different from KVM. The demotion of performance makes a

micro-kernel implementation very attractive because of its potential for simpler development and improved security [46, 40].

In [46], KVM is estimated to contain 33K software lines of code; [40] estimates it to be 20K software lines of code. The jailhouse hypervisor [39] offers a minimalist's alternative to KVM with the explicit goal of having a code base smaller than 10K lines of code. A new microprocessor model that is realized as a type I hypervisor and built exclusively for use as a simulator should require still fewer lines of code. The possibilities for a small code base and targeted feature set suggest that the construction and long term maintenance of a new hypervisor may offer significant economical and technological advantages over modifying an existing hypervisor technology.

The ability to host a complete software stack within a system of systems simulation opens up several new possibilities for analysis. Immediately apparent is the potential for a simulated cyber-range where software may be attacked in circumstances too risky for experimentation in the real world. A particularly novel use of these simulations would be to integrate cyber-security concerns into the early parts of the system engineering process where risks can be assessed and mitigated before the system is put into service. An interesting survey of current industry practices related to cyber-security was recently conducted by Venson *et. al.* [41]. The results suggest a growing interest by industry in these sorts of security oriented engineering practices, at least in relation to a similar (albeit smaller) survey [11] conducted a decade ago.

Simulations could also offer a relatively inexpensive way to accumulate operational hours for complex software systems. The simulated environment could perform continuous testing of the software in operational circumstances that may be rare, dangerous, or both. The variety of scenarios that could be explored with such a simulator increases the likelihood of uncovering software flaws that may not otherwise appear until a critical moment. The particularly well documented failure of a Patriot missile in 1991 offers an intriguing example of where voluminous, inexpensive scenario exploration might have uncovered a software fault prior to its appearance in the field [6]. The ever expanding role of software in systems implies a growing risk of software failures. Simulation could become an important tool for the analysis and mitigation of these risks.

9. REFERENCES

References

[1] A. Akram and L. Sawalha. A survey of computer architecture simulation techniques and tools. *IEEE Access*, 7:78120–78145, 2019.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[3] Jürgen Bitzer and Philipp J.H. Schröder. The economics of open source software development: An introduction. In Jürgen Bitzer and Philipp J.H. Schröder, editors, *The Economics of Open Source Software Development*, pages 1–13. Elsevier, Amsterdam, 2006.

[4] J. Boardman and B. Sauser. System of systems - the meaning of *of*. In *2006 IEEE/SMC International Conference on System of Systems Engineering*, page 6, 2006.

[5] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of software engineering*, 1(1):57–94, 1995.

[6] Ralph V. Carlone, Michael Blair, Sally Obenski, and Paula Bridickas. PATRIOT MISSILE DEFENSE: Software Problem Led to System Failure at Dhahran, Saudi Arabia. Technical Report GAO-IMTEC-92-26, United States General Accounting Office, February 1992.

[7] Bradford Clark, James McCurley, and David Zubrow. *DoD Software Factbook, Version 1*. Carnegie Mellon University, Dec 2015.

[8] Guillaume Delbergue, Mark Burton, Frederic Konrad, Bertrand Le Gal, and Christophe Jego. QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, January 2016.

[9] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.

[10] Functional Mockup Interface. `https://fmi-standard.org/`, 2021.

[11] David Geer. Are companies actually using secure development life cycles? *Computer*, 43(6):12–16, 2010.

[12] glassdoor. `https://www.glassdoor.com/Salary/US-Department-of-Defense-Software-Engineer-Salaries-E14798_D_KO25,42.htm`, accessed March 2021.

[13] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M Voelker. DieCast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems*, 29(2):4–48, May 2011.

[14] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To infinity and beyond: Time-warped network emulation. In *3rd Symposium on Networked Systems Design & Implementation (NSDI 06)*, San Jose, CA, May 2006. USENIX Association.

[15] IEEE. IEEE Standard for Modeling and Simulation (M & S) High Level Architecture (HLA) – Framework and Rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pages 1–38, 2010.

[16] IEEE. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2012.

[17] Marty Kalin. CFS: Completely fair process scheduling in Linux. `https://opensource.com/article/19/2/fair-scheduling-linux`, February 2019.

[18] Yosuke Kurimoto, Yusuke Fukutsuka, Ittetsu Taniguchi, and Hiroyuki Tomiyama. A hardware/software cosimulator for Network-on-Chip. In *Proceedings of the 2013 International Soc Design Conference (ISOCC)*, pages 172–175. IEEE, 2013.

[19] KVM. Kernel virtual machine. `https://www.linux-kvm.org/`, March 2021.

[20] Jereme Lamps, Vignesh Babu, David M. Nicol, Vladimir Adam, and Rakesh Kumar. Temporal integration of emulation and network simulators on linux multiprocessors. *ACM Trans. Model. Comput. Simul.*, 28(1), January 2018.

[21] Hee Won Lee, David Thuente, and Mihail L. Sichitiu. Integrated simulation and emulation using adaptive time dilation. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '14, pages 167–178, New York, NY, USA, 2014. Association for Computing Machinery.

[22] S. Lee and W. W. Ro. Parallel GPU architecture simulation framework exploiting work allocation unit parallelism. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 107–117, 2013.

[23] Marius Montòn, Antoni Portero, Marc Moreno, Borja Martinez, and Jordi Carrabina. Mixed SW/SystemC SoC emulation framework. In *IEEE International Symposium on Industrial Electronics*, pages 2338–2341, 2007.

[24] Specialist Team MSG-131. Modelling and simulation as a service: New concepts and service-oriented architectures. Technical Report STO TR-MSG-131, Science and Technology Organization, North Atlantic Treaty Organization, May 2015.

[25] James Nutaro. A discrete event system simulator. `https://sourceforge.net/projects/adevs/`.

[26] James Nutaro. *Building sofware for simulation*. Wiley, 2010.

[27] James Nutaro and Phil Hammonds. Combining the model/view/control design pattern with the devs formalism to achieve rigor and reusability in distributed simulation. *The Journal of Defense Modeling and Simulation*, 1(1):19–28, 2004.

[28] NVIDIA. `https://www.nvidia.com/en-us/data-center/virtual-gpu-technology/`.

[29] Open Virtual Platform. `https://www.ovpworld.org/`, Accessed March 2021.

[30] OpenVZ. `https://openvz.org/`, accessed March 2021.

[31] Ozgur Ozmen, James J. Nutaro, Jibonananda Sanyal, and Mohammed M. Olama. Simulation-based testing of control software. Technical Report ORNL/TM-2017/45, Oak Ridge National Laboratory, Oak Ridge, United States, Feb 2017.

[32] Marta Pantoquilho. Challenges in testing and validating operational spacecraft simulators. In *Proceedings of the 2010 Conference on Grand Challenges in Modeling & Simulation*, GCMS '10, pages 165–172, 2010.

[33] Mikel D. Petty. Verification,validation,and accreditation. In John A. Sokolowski and Catherine M. Banks, editors, *Modeling and Simulation Fundamental*. John Wiley & Sons, Inc., 2010.

[34] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015.

[35] Christoph Roth, Oliver Sander, Matthias Kühnle, and Jürgen Becker. HLA-based simulation environment for distributed SystemC simulation. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 108–114, Brussels, BEL, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[36] R. G. Sargent. Verification and validation of simulation models. In *Proceedings of the 2010 Winter Simulation Conference*, pages 166–183, 2010.

[37] S. Sumith Shankar, K. Desai, S. Dutta, R. R. Chetwani, M. Ravindra, and K. M. Bharadwaj. Mission critical software test philosophy a SILS based approach in Indian Mars Orbiter mission. In *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, pages 414–419, November 2014.

[38] Stanislav Shwartsman and Darek Mihoka. How Bochs works under the hood, 2nd edition. `http://bochs.sourceforge.net/How%20the%20Bochs%20works%20under%20the%20hood%202nd%20edition.pdf`, June 2012.

[39] Valentine Sinitsyn. Understanding the jailhouse hypervisor, part 1. `https://lwn.net/Articles/578295/`, January 2014.

[40] Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 209–222, April 2010.

[41] Elaine Venson, Reem Alfayez, MarÃŋlia M. F. Gomes, Rejane M. C. Figueiredo, and Barry Boehm. The impact of software security practices on development effort: An initial survey. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019.

[42] VMware. Timekeeping in VMware virtual machines. `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf`.

[43] Elias Weingärtner, Florian Schmidt, Tobias Heer, and Klaus Wehrle. Synchronized network emulation: Matching prototypes with complex simulations. *SIGMETRICS Perform. Eval. Rev.*, 36(2):58–63, August 2008.

[44] Elias Weingärtner, Florian Schmidt, Hendrik vom Lehn, Tobias Heer, and Klaus Wehrle. Slicetime: A platform for scalable and accurate network emulation. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[45] David A. Wheeler. `https://dwheeler.com/sloccount/sloccount.html`, August 2004.

[46] Chiachih Wu, Zhi Wang, and Xuxian Jiang. Taming hosted hypervisors with (mostly) deprivileged execution. In *20th Annual Network and Distributed System Security Symposium*, February 2013.

[47] Srikanth B Yoginath, Kalyan S Perumalla, and Brian J Henz. Virtual machine-based simulation platform for mobile ad-hoc network-based cyber infrastructure. *The Journal of Defense Modeling and Simulation*, 12(4):439–456, 2015.

[48] Bernard Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of Modeling and Simulation, 3rd Edition*. Academic Press, 2018.