J Supercomput (2015) 71:1484–1504
DOI 10.1007/s11227-014-1375-7

Gem5v: a modified gem5 for simulating virtualized systems

Seyed Hossein Nikounia · Siamak Mohammadi

Published online: 5 February 2015

Abstract Virtualization is growing in different areas: from powerful servers in data centers to students' laptops and even cell phones. It can provide a more efficient use of hardware resources. Virtualization enables multiple virtual machines to run side-by-side in an isolated environment on a physical hardware. Modern processors are enhanced with technologies like Intel-VT and AMD-V that speed up virtual machines. However, there is still room for improving support of hardware for virtualization workloads. Gem5 is an open-source full system simulator capable of simulating a Chip-Multiprocessor with its caches, interconnection network, memory controllers among others. In its current state, gem5 does not support virtualized workloads. In this paper, we present a modified version of gem5, named gem5v, that simulates the behavior of a virtualization layer and can simulate virtual machines. We test this simulator in different scenarios using Parsec, Splash, MapReduce (Phoenix), SPEC and EEMBC benchmarks and compare its measured runtime with real systems. Results show 1–9 % difference between the simulated system and two virtualization softwares on a real hardware, namely KVM and VMware ESX. The comparison of vCPU overhead in VMware ESX and gem5v shows between 0.1 and 9 % difference.

Keywords Computer architecture · Virtualization · Consolidation · Simulation · Gem5

S. H. Nikounia · S. Mohammadi (✉)
School of ECE, University of Tehran, Tehran, Iran
e-mail: smohamadi@ut.ac.ir

S. H. Nikounia
e-mail: nikoonia@ut.ac.ir

S. Mohammadi
School of Computer Science, Institute for Research in Fundamental Science (IPM), Tehran, Iran

1 Introduction

Virtualization multiplexes a physical machine among multiple virtual machines. This enables a data center to run multiple virtual private servers (VPSs) on a single physical server to better utilize resources and reduce management costs and power consumption. For instance, a student can run Linux on its Windows machine side-by-side to do his/her OS homework. It can be used by a security researcher to investigate a new malware inside an isolated environment (virtual machine) without affecting production machines/network. In a software team, developers can run multiple virtual machines with different operating systems and configurations to test their new born software in different environments. This will reduce their development and test costs as well as their hardware setup times. One can even run multiple virtual phones on a physical phone to have different cell phones for work and home [15]. Use of virtualization is growing: Gartner has estimated over 50 % of server workloads worldwide are virtualized in 2012. That figure was 16 % in 2009 [2].

A special software layer runs virtual machines (VMs). It is called hypervisor or virtual machine monitor (VMM). VM is usually called *guest* while the machine that runs the hypervisor is called *host*. Figure 1 shows a hypervisor that runs 4 VMs side-by-side. Each of them runs its own OS and applications.

Virtualization better utilizes processor's computing power.

Chip-multiprocessors (CMPs) provide multiple processing cores within a single chip. They have become dominant in processor's market. For example, recent Intel Xeon processors provide up to 10 cores and up to 20 threads[6].

Modern processors have new features that enable hypervisors to speed up VMs. Technologies like Intel-VT [32] and AMD-V [1] add hardware support for an additional layer of address translation needed by hypervisors. See Sect. 2.1 for more details. ARM also has a virtualization extension [33].

Future CMPs can take more advantage of the information that hypervisors can provide about VMs status and needs. They may use it to fairly partition shared caches [10,22,26], optimize routing or cache coherency protocol in their network

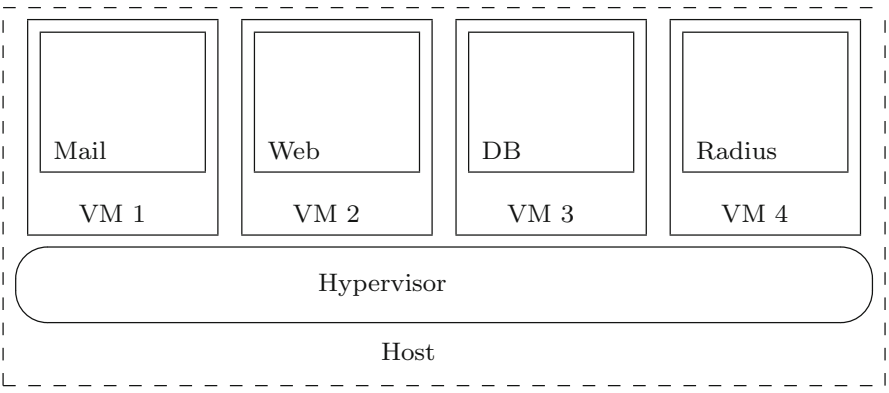

Fig. 1 Hypervisor runs Virtual Machines with their own OS and applications side-by-side in the *host* system

on chip (NoC) [17,19,23,24] or they can provide quality of service (QoS) in memory accesses [21,30]. These are just simple ideas but one could go beyond them to design a virtualization-capable processor.

Every new design needs extensive simulation for its functional and performance verification. There are many simulation tools from high-level task graph view of the application to low-level cycle-accurate NoC simulators or even transistor-level simulators, each designed to be used for specific part of the system: researchers could use transistor-level simulators to simulate a NoC router, a NoC simulator to simulate a NoC within a CMP or higher-level simulators to see a bigger picture of the system.

Gem5 is an event-driven full-system simulator. It simulates a shared-memory CMP that can boot from full Linux kernel in detailed micro-architectural level. It can simulate out-of-order (O3) pipelined CPU along with caches, NoC, bridges and memory controller among others (see Sect. 2.2 for more details). However, the current gem5 is not capable of simulating virtualized systems.

The main contribution of this paper is as follows:

- We have modified gem5 to simulate the behavior of a hypervisor that can simulate multiple virtual machines. We have named it gem5v. It has several features like vCPU, pipelined O3 CPU and ease of configuration using command line arguments—the usual style in gem5.
- We have tested the simulator with different scenarios using some MapReduce (Phoenix)[31], Parsec [12], Splash [34], SPEC CPU2006 [20] and EEMBC (coremark) [16] benchmarks and presented some of their results. We have also compared the simulated runtime with two modern virtualization softwares: KVM [25] and VMware ESX [9]. Results show 1–9 % difference between the simulated system and virtualization softwares on a real hardware.

The rest of this paper is organized as follows: We begin with a background on virtualization and gem5 simulator in Sect. 2. The simulation of a virtualized system is discussed in Sect. 3. Gem5v is described in Sect. 4. We present the simulation results and compare them with real systems in Sect. 5. Finally, we conclude in the last section.

2 Preliminaries

We start with a background on virtualization in Sect. 2.1 and we briefly present gem5 simulator in Sect. 2.2.

2.1 Virtualization

The hypervisor provides a portion of the physical machine as a virtual machine to the guest OS. It contains virtual CPUs, virtual RAM and virtual I/O.

In the processor's part, the guest OS needs to execute privileged instructions (e.g. TLB modification) as well as unprivileged instructions (e.g. an add instruction). While unprivileged instructions can be executed directly on the physical processor, the execution of privileged instructions should be handled with care to ensure that it does not affect other VMs or the host system. Some hypervisors handle this with on-the-fly

binary translation of privileged instructions [29]. To eliminate the translation overhead, some hypervisors like Xen require the guest OS to replace privileged instructions with hypercalls which are understood by hypervisor. This method needs source code modification of the guest OS. To eliminate the translation overhead without guest OS modification, modern processors feature hardware-assisted virtualization, where another protection ring is added and the hypervisor runs in the ring.

An OS running on a physical hardware needs to translate 'virtual addresses' to physical ones. In a virtualized environment, guest OSs have to do the same. They have their own address space. Since hypervisor runs several guest OSs, it needs to translate 'physical address' of the guest OS (i.e. the new 'virtual address' from hypervisor's point of view) to physical address from hypervisor's point of view—that is named 'real address' in the literature. VMs themselves are like individual applications in hypervisor's view. The guest OS translates 'virtual address' of its applications to 'physical address' and the hypervisor translates 'physical address' of guest OS to 'real address'. Modern processors help hypervisors with another level of TLB [1,32]. Without this feature, hypervisors must use *shadow page table* which has its own overhead [29].

Besides processor and memory, VMs need to access I/Os. Hypervisors provide virtual I/O devices to VMs. Some virtual I/O devices are backed by one physical I/O device (e.g. a printer), some of them are not connected to a physical device at all (e.g. a virtual monitor) and some of them share a physical I/O (e.g. network interface). In all the above cases, hypervisor handles I/O accesses. However, modern processors facilitate accesses to I/O with *direct I/O* which enables direct memory access (DMA) to the address space of the VM.

Modern hypervisors have advanced features like page sharing that reduces the memory footprint by sharing common pages among multiple VMs in read-only mode. This is useful when running multiple VMs with the same OS, where some parts of the code (like kernel) do not change in runtime. Hypervisor can save memory by storing these parts only once. They also support migration that enables a VM to migrate form one server to another.

Various parameters of VM are user configurable. Users can define the amount of RAM, number of CPUs and disk space of each VM. They can define multiple VMs with the total number of CPUs that could be more than available physical CPUs. In this case, the hypervisor schedules virtual CPUs (vCPUs) over the physical ones. Physical CPUs are shared among VMs. This is referred to as vCPU feature.

Gem5v simulates the behavior of a hypervisor inside the simulator itself. We assume a modern processor with hardware-assisted features like Intel-VT and AMD-V that provides a dedicated ring for the hypervisor, accelerates address translation and features direct I/O.

2.2 The gem5 simulator

"The gem5 simulator is a modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture" [3]. It simulates a shared-memory CMP with various ISAs: Alpha, x86, ARM and SPARC.

Different CPU models are also supported: from simple atomic, one clock per instruction (CPI) CPUs to pipelined out-of-order (O3) ones. It has two memory models: the classic memory model simulates multi-level cache hierarchies with snooping coherency protocol and the Ruby memory model where caches, directory controllers and memory controllers as well as processors are connected to an interconnection network. Multiple coherency protocols are supported. Note that not all CPU models work with all protocols.

Gem5 can work in two modes: syscall emulation (SE) and full system (FS); SE runs a statically compiled binary, while FS brings up a full kernel and disk image that can run applications in that environment. Multi-threaded programs are usually run in FS mode since `pthreads`[1] are not supported in SE mode. However, `m5threads` partially implements main functionality of `pthreads` and works in SE mode.

Gem5 features checkpointing. Users can take a checkpoint from a specific simulation point and start the simulation from that point several times with different configurations.

Gem5 is made of M5 and GEMS [13]. Historically, GEMS is used to simulate the memory parts (Ruby memory model) and M5 is used to simulate other parts (processors, I/Os, bridges, etc.). Gem5 is object oriented, written in C++ and Python. Python is used to configure and connect C++ objects to build up the simulation environment. Objects are connected by means of *port*; the concept is similar to ports in SystemC. Gem5's binary takes a python script that instantiates simulators' objects like CPUs, caches, etc. and connects them together to build up simulation. Objects are organized in a tree-like structure. The root of this tree is an object called *root*.

Gem5 is distributed under a Berkeley-style open source license.

3 Simulating virtualization workloads

Different methods could be used to simulate a virtualization workload. Based on the target performance metrics and level of abstraction, one could choose the appropriate simulation platform. For example, to simulate several machines in a cloud data center running virtualization workloads one could use CloudSim [14]. However, CloudSim is not intended to be used as a hardware architecture exploration tool; it does not model a detailed processor and does not count cache misses or clock per instruction (CPI). Gem5v is intended to simulate virtualization workloads at detailed micro-architectural level.

Since full system simulators like gem5 are built to run a full OS, theoretically they should be capable of running a hypervisor. Although this seems to be the most accurate way of simulating consolidation workloads, it has its own limitations: simulation time is long since an additional software layer runs on a detailed simulated hardware; configuration of the system is difficult since we have to configure parameters of benchmarks inside VMs' disk images, which reside inside disk image of the host system. Most importantly, this is difficult or even considered impossible [4].

[1] POSIX threads.

Because of the aforementioned limitations, researchers tried to modify existing simulators to simulate the behavior of a hypervisor inside the simulator itself. Virtual-GEMS presented in [18] is based on Simics [27] and GEMS [28]. It uses multiple instances of Simics for each VM that are connected to one Ruby memory model using their added interface.

Although Virtual-GEMS simulates detailed hardware, it lacks some features. Some of them like lack of out-of-order CPU are inherited from Simics. It does not support vCPU. Original GEMS no longer exists. It is part of gem5 and updated in gem5's source code. Our work is similar to Virtual-GEMS. We will compare gem5v with Virtual-GEMS in Sect. 4.1.

4 Gem5's new feature: virtualization support

Our modified version of gem5 simulates a hypervisor which can run multiple virtual machines. Just like any hypervisor, number of CPUs and the amount of dedicated RAM could be defined. VMs run in FS mode and any number of them could be defined. We have chosen FS mode instead of SE mode since it can run multi-threaded applications.

Users have control over assignment of processors to a VM in the interconnection network (e.g. Mesh). Each VM has its own kernel binary, disk images and I/Os. All VMs share a memory subsystem, Ruby, where memory requests are served and their timing is handled. We have chosen Ruby memory model over the classic one since it is more detailed.

We have modified Ruby to support translation of *physical* addresses (that is valid inside *system*) to *real* addresses. We assume that the simulated processor has hardware-assisted virtualization capability that can accelerate *real* to *physical* address translation in hardware. Thereby, the translation overhead is negligible compared to an ordinary processor that uses shadow page table [32]. The aforementioned model and the assumption are similar to those of Virtual-GEMS.

In the simplest form, physical address spaces of each VM are mapped one after another into real address spaces. There exist more complex methods. Hypervisor can map each newly addressed virtual page to the next free physical page. We have implemented both the above methods. Although the implementation of other methods is not hard, it is shown in [18] that they do not have significant differences.

Gem5v supports vCPU: physical processors can be shared between VMs. VMs share processing power in a weighted round-robin manner. The fraction of share is defined by the user. The context-switch overhead can also be configured.

Since gem5's code base is not multi-threaded, gem5v simulates VMs serially. Hence, the simulation time in gem5v approximately equals the sum of simulation time of its VMs (their OS and benchmarks) in gem5.

Like the original gem5, our version supports out-of-order and pipelined CPU, different ISAs, cache coherency protocols and interconnection networks. It coexists with other features of the simulator like checkpointing. It is open source and can be downloaded, used and modified freely.[2]

[2] We would like to submit it to the review board of gem5 so it could be part of the mainline.

Table 1 Comparison of gem5v and virtual-GEMS

Feature	Gem5v	Virtual-GEMS
Common features	Yes	Yes
Ruby memory model	Yes (updated)	Yes
Physical to real address translation methods	Two	Four
Virtual CPU	Yes	No
Full system simulation	Yes	Yes
Pipelined, in-order CPU	Yes	No
Pipelined O3 CPU	Yes	No
Virtualized I/O	No	No
Advanced features	No	No
Open source	Yes	Partially

In Sect. 4.1, we compare gem5v with Virtual-GEMS. We describe our architecture and technical details in Sect. 4.3. Section 4.2 discusses limitations of our current version.

4.1 Comparison

To the best of our knowledge, Virtual-GEMS [18] is the only comparable simulator for virtualized workloads: Both Virtual-GEMS and gem5v simulate VMs in full-system mode and with detailed hardware.

Besides the common features, both use Ruby memory model. However, GEMS (which provides Ruby) is now updated in gem5's source code.

Virtual-GEMS benefits from various physical-to-real address translation methods, while gem5v features vCPU and can simulate pipelined in-order and also pipelined out-of-order processors. Virtual-GEMS uses Simics which is an under commercial-license product; GEMS and their patch are open-source. Gem5v is fully open source. Both use dedicated I/O for VMs and do not implement advanced features (see Sect. 4.2). Table 1 summarizes comparison of gem5v with Virtual-GEMS.

4.2 Limitations

Some limitations of our current version are described here. Users cannot specify different kernel binary and disk images for VMs through command-line arguments. This could be implemented easily in the configuration script.

We have tested x86 and Alpha; Logically, other ISAs should also work. For cache coherency protocols and interconnection networks, *MOESI_CMP_token*, *MOESI_hammer* and *Mesh* are modified, other topologies and coherency protocols can be modified likewise.

We do not simulate virtualized I/Os. Each VM has its own I/Os. Our current implementation is similar to direct I/O feature of modern processors. However, implementation of shared virtualized I/O would be interesting.

We do not consider the situation where the VM puts the vCPU in *idle* state (see Sect. 4.3 for more details). Current version uses two methods for physical-to-real address translation and does not implement the page sharing. Other advanced features like migration are not implemented.

We have used FS mode. However, implementing it in SE mode would be easy as writing a new python configuration script. One can clone `se.py` and modify it like our added `hypervisor.py`.

4.3 Technical details

We simulate each VM by creating a new instance of *system* in gem5. Each *system* contains its own kernel binary, disk images and I/Os. All *system*s are connected to one instance of Ruby, which simulates memory parts: caches, DMA controllers, NoC, etc. Ruby is modified to support translation of *physical* addresses to *real* addresses.

Multiple instances of *system* is created in our provided python configuration script. Each *system* represents a VM. They are defined to be children of *root*. One Ruby is instantiated to simulate the memory part of the system. Ruby is defined to become the child of the first *system* in the tree.

The component that connects ports of each *system* to Ruby ports is modified, so that it is able to translate physical addresses within the *system* to real addresses that are valid inside Ruby. Figure 2 shows the aforementioned architecture.

Placement of VMs in the interconnection network (NoC) is defined by the way CPU ports in *system*s are connected to their routers. This is done in configuration python scripts.

For each VM, an instance of *system* is built which contains its own disk image and kernel with specified number of CPUs. One instance of *Ruby* is created to function as the memory system. Its ports are connected to the appropriate CPU ports. All these connections along with building interconnection topology are made inside configuration python scripts. Source codes are changed to make this configuration (multiple systems, one ruby) possible.

Each *system* has its own physical address space. We need to translate *physical addresses* to *real* addresses. It is implemented inside Ruby's `Sequencer`.

With the above architecture, a VM will either have 100 or 0 % share of a processor. In other words, physical CPUs are dedicated to VMs; processors cannot be shared between VMs and multiple VMs cannot be scheduled on a processor. Current hypervisors can allocate one physical CPU for more than one VM. As described in Sect. 2.1, this feature is called virtual CPU (vCPU).

In gem5v, we would like to be able to configure the share of each physical processor dedicated to a VM. Hypervisors achieve this by scheduling VMs on processors. This is similar to scheduling processes in a typical OS. To achieve this in gem5v, one solution could be to periodically move shared processors between *system*s (VMs) who share them. In this way, a physical processor is shared between *system*s in a time sharing manner. However, current code-base of gem5 does not allow it, thereby we implement the following solution: CPU component of gem5 is modified so that it can work in a specified *period* of cycles in a *hyperperiod*. This enables us to have multiple CPUs

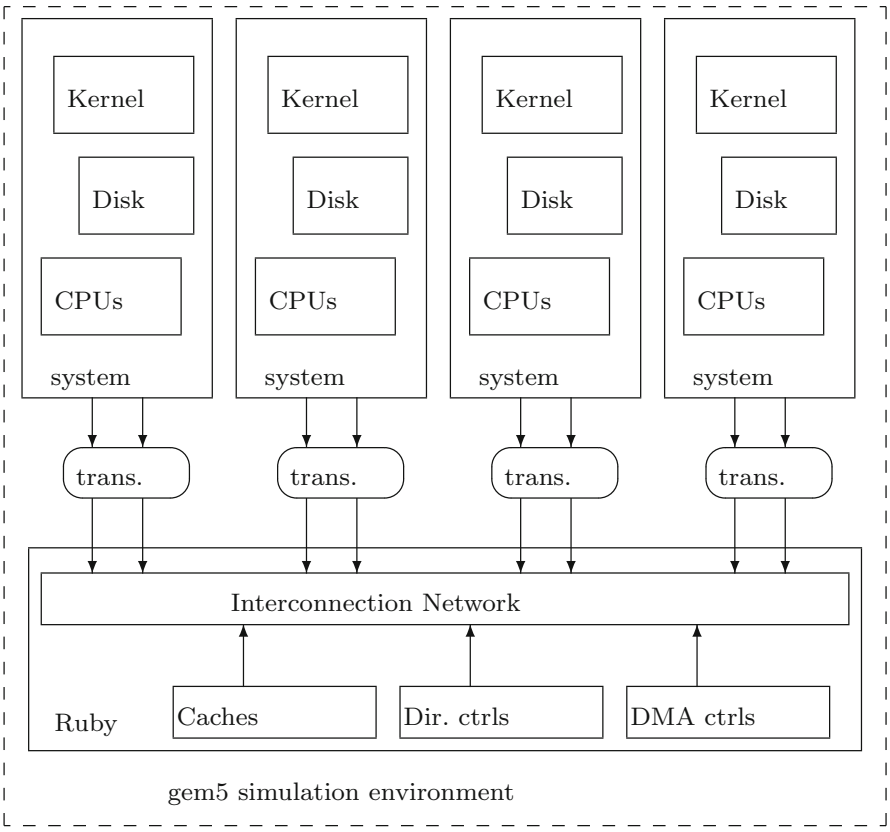

Fig. 2 We create multiple instance of systems for each VM. Processors of the systems are connected to one Ruby with a module that translates physical addresses within a system to real addresses valid in Ruby

that work in non-overlapping periods of time. Suppose these CPUs are part of different VMs (gem5's *system*s), it looks as if they share one CPU in a round-robin manner.

When a fraction of a physical CPU is assigned to a VM, we create an instance of a CPU inside its *system*. This CPU is configured to work in a specific period of times in a hyperperiod. The duration of the period is assigned based on the share of the VM from that physical CPU. The duration of the hyperperiod is defined by the user. Other VMs that share this physical CPU also get a CPU in the same way. However, we set location of their period in the hyperperiod in the python configuration script automatically, so that their periods are non-overlapping.

We provide the context switch time by shortening the duration of assigned periods based on the required context switch overhead. In this way, VMs do not have the physical CPU during context switches. The context switch time could be defined by the user.

In a real virtualization system, when a physical CPU is assigned to a vCPU, the VM will have it in the assigned period. If the VM has nothing to do and puts the vCPU in *idle* state, it effectively releases the physical CPU. Our current implementation

does not consider this situation. We know that our method does not simulate a 100 % accurate vCPU. However, our comparison between the overhead of scheduling four VMs on one CPU instead of four in gem5v and in VMware ESX shows a good match between simulated and the real system (see Sect. 5.4).

To support vCPU, we add new parameters (specifying the hyperperiod, start and stop of active period of the vCPU) to `ClockedObject` class that makes the CPU active in a portion of a *hyperperiod*. We do this by changing the way the `ClockedObject` schedules the next event. VCPU that represents a physical CPU is connected to one router.

The default values for our added parameters are set so when virtualization-specific parameters are not set, modified components can perform their function in non-virtualized environment as before.

To build up the simulation environment, gem5 uses some python configuration scripts. A python configuration script is added. It is called `hypervisor.py`.[3] Different options of this script are described in Appendix 7.

5 Testing simulator

To test the simulator, we have simulated several scenarios using some of MapReduce (Phoenix) [31], Parsec 2.1 [12], Splash [8], SPEC CPU2006 [20] and EEMBC (coremark) [16] benchmarks.

We have simulated several scenarios with multiple VMs where each VM runs a benchmark. We have also simulated scenarios that represent a host environment running one benchmark. We compared performance statics of various scenarios in the following subsections. The simulations are launched on an Intel Core i7 machine with 10GB of RAM. More detailed description of each scenario is described in each subsection.

We continue this section by testing the correct run of multiple VMs, presenting the effect of co-scheduled VMs, comparing with two real hypervisors and testing the vCPU feature in gem5v.[4]

5.1 Running multiple virtual machines

If a disk image, kernel and workload(s) can run on original gem5, gem5v can run it as a VM as well. To confirm this, besides careful code modification, we have run several tests with different workloads from MapReduce, Parsec, Splash, SPEC CPU2006 and EEMBC; it successfully runs them. Memory requests (i.e. Every read and write from CPUs) are directed through one Ruby instance where memory requests' timings are handled. This ensures correct timing of memory requests.

[3] Located in `configs/example`.

[4] As we do not have a license for Simics, we are unable to do tests for comparison of Virtual-GEMS and gem5v.

Table 2 System specification when studying the effect of co-scheduled VMs in multi-threaded benchmarks

	Host		Hypervisor	
	x86	Alpha	x86	Alpha
Cores	8	16	32	64
Core per VM			8	16
RAM per VM			512MB	512MB
RAM	512MB		2GB	
Simulator	Original gem5		gem5v	
CPU Type	TimingSimpleCPU			
CPU Frequency	2 GHz			
Line Size	64B			
L1I Cache	$32kB$ 2-way set associative			
L1D Cache	$64kB$ 2-way set associative			
L2 Cache Size	$\{32k, 64k, 128k, 256k, 512k, 1M, 2M\}B$ 8-way set associative			
Replacement Policy	PSEUDO_LRU			
Interconnect	Mesh			
Coherency	MOESI_hammer			

5.2 The effect of co-scheduled virtual machines

Co-scheduled VMs share common resources of the processor like the shared cache. In gem5v, memory requests are served by one Ruby instance. This ensures an accurate timing of memory accesses.

To see the effect of co-scheduled VMs, we present some of our simulation results. We simulate three *hypervisor*s and three *host*s using gem5v and the original gem5, respectively. For multi-threaded benchmarks (Parsec and MapReduce), gem5v simulated four multi-core VMs and for single-threaded benchmarks (SPEC CPU2006 and EEMBC coremark) it simulated two single-core VMs. Hypervisors co-schedule VMs that run their own benchmark while hosts run a single benchmark. Tables 2 and 3 present configurations of single-threaded and multi-threaded environments, respectively. We have used both x86 and Alpha with various L2 cache sizes. Each core has its own L1I and L1D caches. Most of these configurations are gem5's defaults. Note that the results shown here are part of our experiments and only intended to show the authenticity of our implementation by exploring the effect of co-scheduled VMs on IPC and cache statistics. Interference study is out of scope of this paper.

In hypervisor's experiments, we use the notation $b_1@b_2 - b_3 - b_4$ to refer to the scenario where b_1 is running in a VM and b_2, b_3 and b_4 are running in neighboring VMs.[5] Actually, we run b_i benchmark in vm_i. Similarly, we use $b_1@b_2$ for two VM scenarios.

[5] That is why we have used the @ notation.

Table 3 System specification when studying the effect of co-scheduled VMs in single-threaded benchmarks

	Host	Hypervisor
	x86	x86
Cores	1	2
Core per VM		1
RAM per VM		512MB
RAM	512MB	1GB
Simulator	Original gem5	gem5v
CPU Type	TimingSimpleCPU	
CPU Frequency	2 GHz	
Line Size	64B	
L1I Cache	$32kB$ 2-way set associative	
L1D Cache	$64kB$ 2-way set associative	
L2 Cache Size	$1MB$ 8-way set associative	
Replacement Policy	PSEUDO_LRU	
Interconnect	Mesh	
Coherency	MOESI_CMP_token	

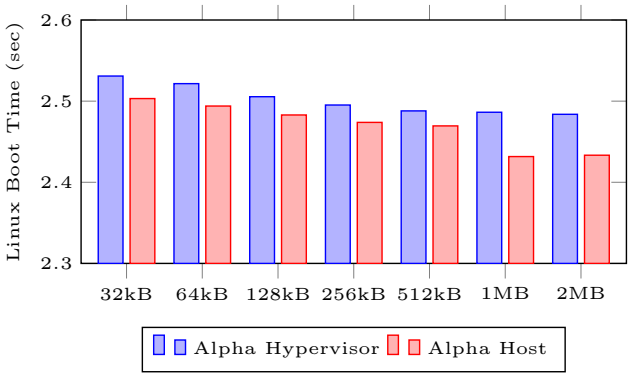

Fig. 3 Linux kernel boot time in Alpha host and Alpha hypervisor running 4 VMs for different L2 sizes

5.2.1 Linux kernel boot time

In this part, we compare the Linux kernel boot time in host and 4-VM hypervisor environment. Figure 3 shows Linux kernel 2.6.27 [7] boot time that boots under Alpha in about 5.5 s and Fig. 4 shows boot time of about 2.5 s for kernel 2.6.22.9.smp [3] in x86.

Boot time in x86 and Alpha environments are not comparable since kernel versions, configuration options of the kernel and processor's architectures are different. We want to compare the boot time in the host and hypervisor environments of each configuration.

Linux kernel in our Alpha configuration boots faster with larger L2 caches. With the same L2 cache size, the kernel boots about 0.7–2.5 % faster in the host environment. This is due to the cache capacity being shared with other VMs.

This is slightly different in our x86 configuration. The interesting point here is that for 64 to 512 kB L2 size, the kernel boots around 14 % faster in the hypervisor

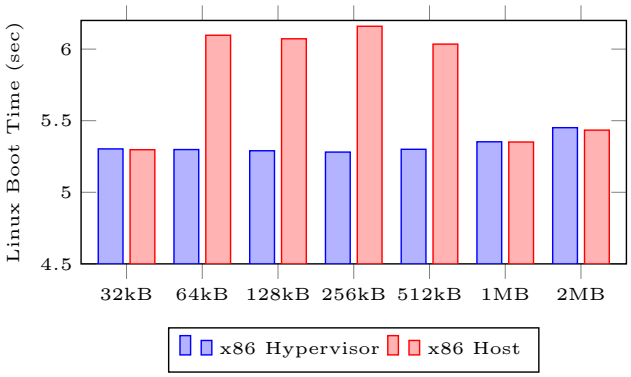

Fig. 4 Linux kernel boot time in x86 host and x86 hypervisor running 4 VMs for different L2 sizes

environment. This is also due to cache sharing between VMs that results in lower effective cache capacity for each VM. Note that the kernel in our x86 configuration boots faster with lower cache sizes (e.g. 32 kB).

5.2.2 The effect of co-scheduled VMs on L2 misses

We have run Parsec 2.1 benchmarks with small input size in simulated Alpha hypervisor configured as shown in Table 2. In a virtualized system, all running VMs are using the shared cache and their access pattern causes different miss rates. We now discuss L2 cache misses in some of our experiments.

Figures 5 and 6 show a portion of L2 miss rate in time for different scenarios for L2 cache size 64 and 128 kB, respectively. Each figure arranges different scenarios in a grid so that benchmark of vm_1 is fixed in each row and benchmarks of vm_2, vm_3 and vm_4 are fixed in each column. We have abbreviated benchmark names in figures. Table 4 lists them.

Since benchmarks vary in different experiments, we expect different L2 misses and that is what we see in figures. However, changing only one benchmark (i.e. moving in one column) does not necessarily mean similar L2 miss rates. For example, @streamcluster-blackscholes-vips for 64 kB L2 size have almost similar L2 miss rates while these experiments with 128 kB L2 size do not. This observation shows that in cache contention studies of virtualized systems, the cache size of the target platform should be considered.

When a VM with a large working set is running, we expect high L2 miss rate. That is the case for the middle column in two figures where dedup is running as a neighbor.[6] According to [11], dedup has a large working set.

[6] According to Table 4, *d* in *ffd* means dedup.

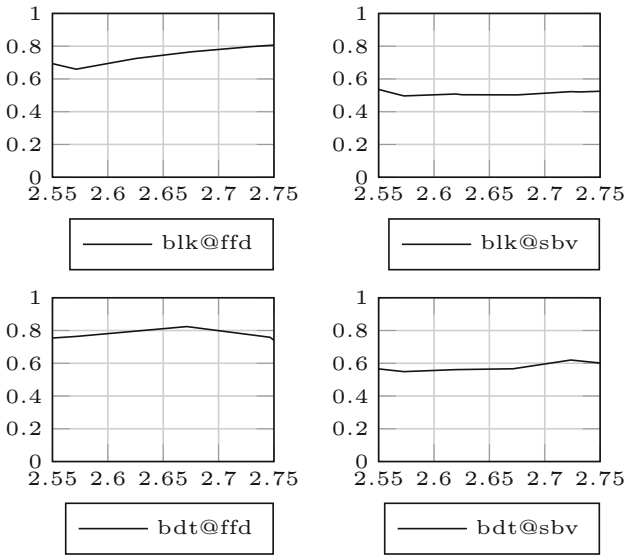

Fig. 5 L2 miss rate in time (s) for 64 kB L2 cache. See Table 4 for abbreviation

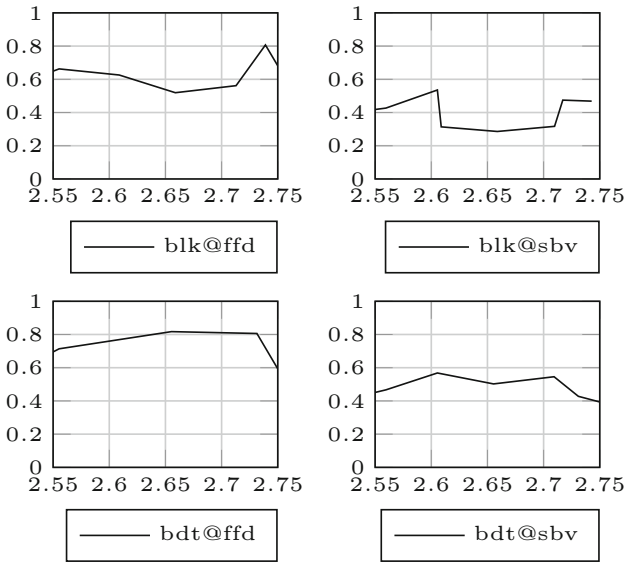

Fig. 6 L2 miss rate in time (s) for 128 kB L2 cache. See Table 4 for abbreviation

5.2.3 The effect of co-scheduled VMs on IPC

As VMs share common resources of the processor, they might affect the Instruction Per Cycle (IPC) of their neighbors. To see this effect, we present the results of two sets of our experiments.

Table 4 Short form of benchmark names used in figures

Abbreviation	Name
blk	Blackscholes
bdt	Bodytrack
cnl	Canneal
fld	Fluidanimate
ddp	Dedup
vps	Vips
chl	Cholesky
bbr	Blackscholes-bodytrack-rtview
bcr	Blackscholes-canneal-rtview
frf	Ferret-rtview-facesim
ffd	Freqmine-fluidanimate-dedup
sbv	Streamcluster-blackscholes-vips
bfc	Bodytrack-fft-cholesky
bbl	Blackscholes-bodytrack-lu_ncb

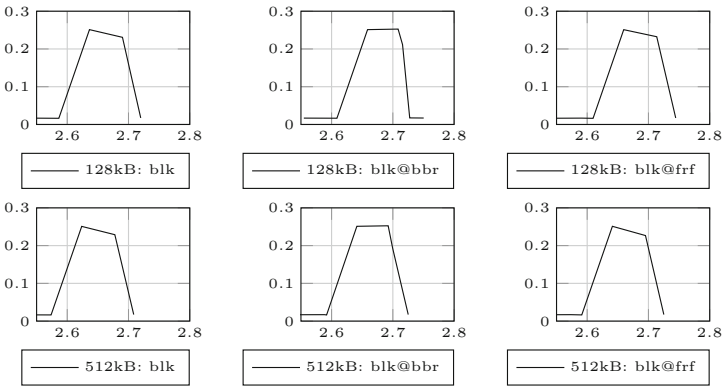

Fig. 7 Average IPC in time (s). See Table 4 for abbreviation

We measure the average IPC of vm_1's processors running on our simulated 64-core Alpha hypervisor and compare it with the average IPC of the same benchmark running in our simulated host. Table 2 shows configuration details. Note that the host system is simulated using the original gem5 while the hypervisor is simulated with gem5v. Benchmarks shown here are from Parsec 2.1 benchmark suite with small input sizes.

Figure 7 compares the IPC of blackscholes benchmark running in a host with the situation where it is running inside a VM with three neighboring VMs running other Parsec benchmarks. It can be seen that in our configuration, neighboring VMs slightly affect the shape of blackscholes' IPC: between 0.2 and 32.9 % difference with the host environment for 128 kB L2 size; that is 10.2 % on average. Although we have only shown the IPC for two cache sizes, our results with other cache sizes have shown the same pattern.

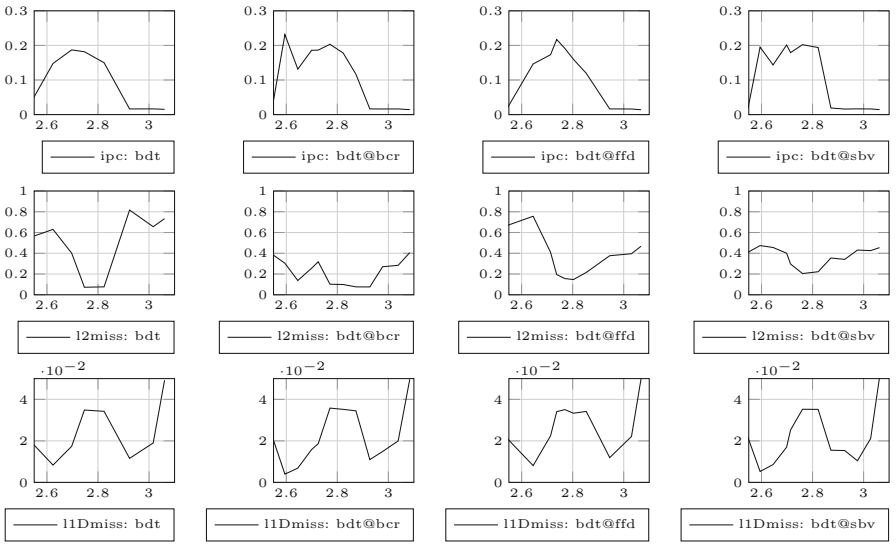

Fig. 8 L2, L1D and L1I miss as well as IPC in time (s) for 256 kB L2 cache. See Table 4 for abbreviation

In contrast, neighboring VMs affect more the IPC of bodytrack: between 15.5 and 41.3 % difference with IPC of running in the host; that is 27.0 % on average. Figure 8 shows this effect for L2 size of 256 kB. This figure also shows L2, L1I and L1D cache miss rates. It is clear that L1I and L1D cache miss rates are also affected. The L2 miss rates differ significantly since different VMs are accessing it. Other L2 sizes show similar trend (not shown here).

Bodytrack is more sensitive to reduction in memory bandwidth than blackscholes [11]. This explains the above difference in the effects of neighbors on blackscholes and bodytrack.

We continue this subsection with IPC analysis of four SPEC CPU2006 and EEMBC Coremark benchmarks. Table 3 shows configuration details of our host and hypervisor environment for these tests. We have selected four benchmarks from SPEC CPU2006 benchmarks: libquantum and mcf that are memory intensive and bzip2 and gcc that are compute intensive. We have run every combination of these four benchmarks in the simulated hypervisor to see the effect of neighboring VM on the L2 cache. We run our experiments for about 4×10^9 cycles. We have used *test* inputs in SPEC benchmarks. Figure 9 shows the percentage of reduced IPC in simulated hypervisor compared to simulated host for some SPEC benchmarks. The reduced IPC depends on how benchmarks suffer from reduced shared resources (like shared L2 cache) and the behaviour of the neighboring VM. For example, bzip2 has large L2 miss ratio, 29 %, in our simulations and caused up to 3.5 % reduction in IPC. We also expect to see performance degradation when two memory intensive benchmarks are co-located and that is what we see in mcf@libquantum.

Coremark experiences about 0.02 % less IPC when it is co-running with another instance of coremark in our hypervisor environment (simulated with gem5v) compared to its alone run in the host environment (simulated with original gem5). That is

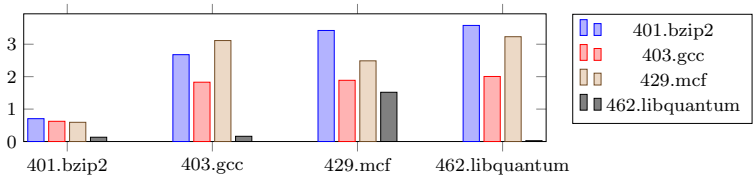

Fig. 9 Percentage of reduced IPC of some of SPEC CPU2006 benchmarks when co-located with a VM running another SPEC CPU2006 benchmark

Table 5 System specification when comparing simulation results with a real system

	Gem5	Real
CPU per VM	1 In-order, pipelined x86	1 Core of AMD Opteron 6172
Frequency	2 GHz	2.1 GHz
RAM per VM	512 MB	512 MB
Line size	64B	64B
L1I cache	32 kB 2-way set assoc.	64 kB 2-way set assoc.
L1D cache	64 kB 2-way set assoc.	64 kB 2-way set assoc.
L2	12 MB shared 8-way set assoc.	Per core 512 kB 16-way set assoc.
L3	N/A	12 MB shared 96-way set assoc.
Replacement policy	Pseudo LRU	Pseudo LRU
Coherency protocol	MOESI_CMP_token	MOESI
Interconnect	2×2 Mesh	Bus

understandable since coremark is intended to grade processor's core capabilities and is not memory intensive.

5.3 Comparing with two real systems

To compare simulation results with real virtualization softwares, we have run 4 VMs on a 4-core x86 CMP on gem5v and compared the results with KVM [25] and VMware ESX [9] on a real CMP. We have done this experiment with blackscholes@bodytrack-fft-cholesky and cholesky@blackscholes-bodytrack-lu_ncb configurations. Fft, cholesky and lu_ncb are from Splash benchmarks.

We have averaged the runtime in the real system over five runs. Machine specifications are described in Table 5. We have tried to make the specification of the simulated machine similar to that of the real system. However, in order to make a fair comparison, benchmarks are run with two input sizes: *small* and *medium* and we have compared t_{medium}/t_{small} as runtime. Figures 10 and 11 report the runtime normalized to simulated results for blackscholes@bodytrack-fft-cholesky and cholesky@blackscholes-bodytrack- lu_ncb, respectively. It can be seen that the simulated results match the results from the real system with little margin: between 1 and 9 % of difference on average in our tests.

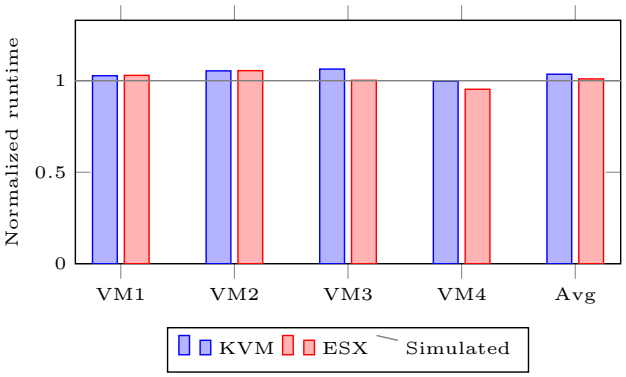

Fig. 10 Comparing normalized runtime of blackscholes@bodytrack-fft-cholesky benchmarks in KVM, VMware ESX and simulated system

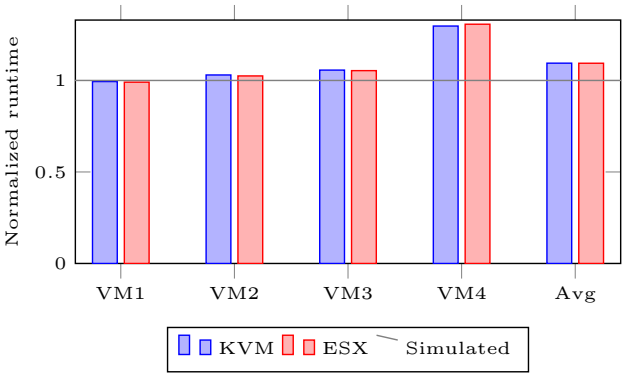

Fig. 11 Comparing normalized runtime of cholesky@blackscholes-bodytrack-lu_ncb benchmarks in KVM, VMware ESX and simulated system

5.4 Testing vCPU

We want to study the overhead of running four VMs on one CPU instead of four CPUs in the simulated environment using vCPU feature and compare the results with VMware ESX. Specifications of real and simulated machines are described in Table 5 for 4-CPU configuration. We assign 25 % of each core to a VM in 1-CPU configuration. We perform this experiment for blackscholes@bodytrack-fft-cholesky and cholesky@blackscholes-bodytrack- lu_ncb configurations. The overhead is calculated for the shortest benchmark. Runtimes in the real system are averaged over five runs. Figure 12 compares the overhead in VMware ESX and the simulated system. It shows that overheads are similar: There are 0.1 and 9 % differences between overhead of the simulated system and ESX for two configurations.

We could not run this experiment for KVM: it seems that Debian Linux that runs KVMs has a bug in setting affinity of processes.

This experiment shows that vCPU's feature of gem5v simulates the behavior of vCPU in a real hypervisor with a good match with regard to overheads.

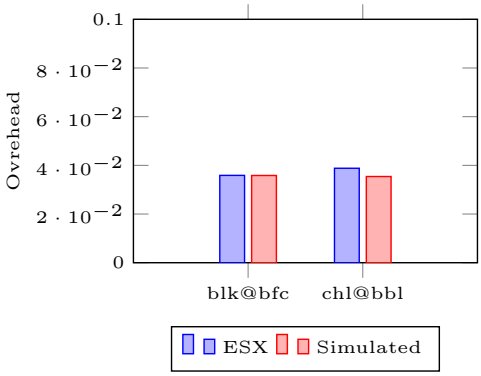

Fig. 12 Comparing overhead of running 4 VMs in 1 CPU instead of 4 CPU. See Table 4 for abbreviation

6 Conclusion and future work

This paper presented a modified version of the gem5 simulator, named gem5v, that supports simulation of virtualized systems: It simulates the run of multiple isolated virtual machines with their own OSs and benchmarks. Gem5v supports multiple ISAs, out-of-order CPU, various interconnection topologies and coherency protocols. Our modified gem5 simulates multiple virtual CPUs that allow multiple VMs share a single physical CPU. It is open source.

Gem5v is able to run the same benchmarks as gem5, in a VM. To confirm this, besides careful code modification, we tested this simulator with some of MapReduce (Phoenix), Parsec, Splash, SPEC CPU2006 and EEMBC benchmarks in different scenarios and gem5v was successful in those tests.

We have also compared gem5v with real hypervisors. Results showed 1–9 % differences between benchmark runtime in the simulated system and two virtualization softwares: KVM and VMware ESX. Comparison of overhead of vCPU in VMware ESX and gem5v showed 0.1 and 9 % difference in two configurations. We have also presented a comparison between features of Virtual-GEMS and gem5v.

Our gem5v has some limitations including not supporting virtualized I/O (we simulate dedicated I/O per VM), vCPU's idle state, page sharing and migration. We consider adding support for advanced methods like page sharing and supporting both SE and FS mode of gem5 as our future work.

7 Options for users

`hypervisor.py` adds the following options: `--vm-cpu-placements` defines assignment of VMs to CPUs as well as their share from CPUs.

`--vm-context-switch-hyperperiod` defines the hyper period of the round-robin. This is used in scheduling of vCPUs. Since the default scheduling quantum in Linux kernel is 100 ms, we set the default value for hyperperiod to 1 s (for 10 VMs that are scheduled on a processor).

`--vm-context-switch-overhead` defines the context switch overhead used in vCPU. Our study shows process context switch overhead of 5,993 ns for an in-order

pipelined x86 CPU in gem5 with default configuration. Hence we set the default value of this parameter to 6 ms. We have used the method described in [5] for this study.

Memory size of VMs is defined by `--vm-mem-sizes`. The `.rcS` script of each VM can be defined using `--vm-scripts`. If not set, the usual `bash` prompt will be given. See `hypervisor.py --help` for detailed syntax.

Acknowledgments This research was in part supported by a Grant from IPM (No. CS1393-4-17).

References

1. AMD-V. http://www.amd.com/virtualization. Accessed on Nov 2012. Accessed 5 May 2014
2. Gartner: estimate of virtualized workloads. http://gartner.com/it/page.jsp?id=1211813. Accessed 1 May 2014
3. Gem5. http://gem5.org. Accessed 5 May 2014
4. [gem5-users] Running Xen in gem5. http://www.mail-archive.com/gem5-users@gem5.org/msg00367.html. Accessed 1 Jan 2014
5. How does it take to make context switch. http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html. Accessed 1 May 2014
6. Intel Xeon. http://www.intel.com/content/www/us/en/servers/server-products.html. Accessed on Nov 2012. Accessed 1 May 2014
7. Parsec 2.1 for m5. http://www.cs.utexas.edu/cart/parsec_m5. Accessed 5 May 2014
8. Splash 2. http://www-flash.stanford.edu/apps/SPLASH. Accessed 1 Jan 2014
9. VMware. http://vmware.com. Accessed 5 May 2014
10. Apparao P, Iyer R, Newell D (2008) Implications of cache asymmetry on server consolidation performance. In: IEEE international symposium on workload characterization, pp 24–32
11. Bhadauria M, Weaver VM, McKee SA (2009) Understanding PARSEC performance on contemporary CMPs. In: IEEE international symposium on workload characterization, pp 98–107
12. Bienia C, Li K (2009) Parsec 2.0: a new benchmark suite for chip-multiprocessors. In: Workshop on modeling, benchmarking and simulation
13. Binkert N, Beckmann BM, Black G, Reinhardt S, Saidi A, Basu A, Hestness J, Hower D, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill M, Wood D (2011) The gem5 simulator. SIGARCH Comput Archit News 39(2):1–7
14. Calheiros RN, Ranjan R, Beloglazov A, De Rose CAF, Buyya R (2010) CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw Pract Exp 41(1):23–50
15. Dall C, Andrus J, Hof AV, Laadan O, Nieh J (2012) The design, implementation, and evaluation of cells: a virtual smartphone architecture. ACM Trans Comput Syst 30(3):1–31
16. Gal-On S, Levy M Exploring CoreMark—a benchmark maximizing simplicity and efficacy. EEMBC Whitepaper. http://www.eembc.org/techlit/coremark-whitepaper.pdf
17. Garcia-Guirado A, Fernandez-Pascual R, Garcia J (2010) Analyzing cache coherence protocols for server consolidation. In: International symposium on computer architecture and high performance computing, pp 191–198
18. Garcia-Guirado A, Fernandez-Pascual R, Garcia JM (2009) Virtual-GEMS: an infrastructure to simulate virtual machines. In: International workshop on modeling, benchmarking and simulation
19. Garcia-Guirado A, Fernandez-Pascual R, Ros A, Garcia J (2011) Energy-efficient cache coherence protocols in chip-multiprocessors for server consolidation. In: International conference on parallel processing, pp 51–62
20. Henning JL (2006) SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput Archit News 34(4):1–17
21. Iyer R, Zhao L, Guo F, Illikkal R, Makineni S, Newell D, Solihin Y, Hsu L, Reinhardt S (2007) QoS policies and architecture for cache/memory in CMP platforms. In: ACM SIGMETRICS international conference on measurement and modeling of computer systems, pp 1–12. ACM request permissions
22. Jin X, Chen H, Wang X, Wang Z, Wen X, Luo Y, Li X (2009) A simple cache partitioning Approach in a virtualized environment. In: IEEE international symposium on parallel and distributed processing with applications, pp 519–524

23. Kim D, Ahn J, Kim J, Huh J (2010) Subspace snooping: filtering snoops with operating system support. In: International conference on parallel architectures and compilation techniques, pp 111–122, ACM
24. Kim D, Kim H, Huh J (2010) Virtual snooping: filtering snoops in virtualized multi-cores. In: IEEE/ACM international symposium on microarchitecture, pp 459–470
25. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A (2007) kvm: the Linux virtual machine monitor. In: Linux symposium, pp 225–230
26. Koller R, Verma A, Rangaswami R (2011) Estimating application cache requirement for provisioning caches in virtualized systems. In: IEEE annual international symposium on modelling, analysis, and simulation of computer and telecommunication systems, pp 55–62. IEEE Computer Society
27. Magnusson PS, Christensson M, Eskilson J, Forsgren D, Hallberg G, Hogberg J, Larsson F, Moestedt A, Werner B (2002) Simics: a full system simulation platform. Computer 35(2):50–58
28. Martin MMK, Sorin DJ, Beckmann BM, Marty MR, Xu M, Alameldeen AR, Moore KE, Hill MD, Wood DA (2005) Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. SIGARCH Comput Archit News 33(4):92–99
29. McDougall R, Anderson J (2010) Virtualization performance: perspectives and challenges ahead. SIGOPS Oper Syst Rev 44(4):40–56
30. Srikantaiah S, Kandemir M, Wang Q (2009) Sharp control: controlled shared cache management in chip multiprocessors. In: IEEE/ACM international symposium on microarchitecture, pp 517–528, IEEE
31. Talbot J, Yoo RM, Kozyrakis C (2011) Phoenix++: modular MapReduce for shared-memory systems. In: MapReduce '11: Proceedings of the 2nd international workshop on MapReduce and its applications, pp 9–16. ACM request permissions
32. Uhlig R, Neiger G, Rodgers D, Santoni A, Martins F, Anderson A, Bennett S, Kagi A, Leung F, Smith L (2005) Intel virtualization technology. Computer 38(5):48–56
33. Varanasi P, Heiser G (2011) Hardware-supported virtualization on ARM. In: Second Asia-Pacific workshop on systems, pp 1–5, ACM
34. Woo S, Ohara M, Torrie E, Singh J, Gupta A (1995) The SPLASH-2 programs: characterization and methodological considerations. In: International symposium on computer architecture, pp 24–36