HyperBench: A Benchmark Suite for Virtualization Capabilities

Anonymous Author(s)*

ABSTRACT

Virtualization is becoming increasingly common in data centers due to its various advantages. However, how to choose among different platforms, including both software and hardware, is a considerable challenge. In this context, evaluating the virtualization capabilities of different platforms is critically important. Regrettably, the existing benchmarks are not qualified for meeting this requirement. Different hardware mechanisms and hypervisor designs introduce many different hypervisor-level events, such as transitions between VMs and the hypervisor, two-dimensional page walk, and binary translation. These events are key factors affecting virtualization performance. Existing benchmarks either overlook these changes or are tightly coupled to a particular hypervisor.

In this paper, we present HyperBench, a benchmark suite that focuses on the capabilities of different virtualization platforms. Currently, we design 15 hypervisor benchmarks covering CPU, memory, and I/O.

The virtualization-sensitive operation in each benchmark triggers hypervisor-level events, which examines the platform's ability in the target area. HyperBench is designed as a custom kernel which can adapt to different hypervisors and architectures. What's more, adding a new benchmark is pretty easy.

Finally, we perform a series of experiments on the host machine and several popular hypervisors, such as QEMU, KVM, and Xen, demonstrating that HyperBench is capable of revealing the performance implications of the hardware mechanism and hypervisor design.

CCS CONCEPTS

• **General and reference** → **Performance**; *Measurement*; • **Software and its engineering** → **Virtual machines**.

KEYWORDS

virtualization capabilities, benchmarks, performance, hypervisors

ACM Reference Format:

Anonymous Author(s). 2018. HyperBench: A Benchmark Suite for Virtualization Capabilities. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Virtualization is ubiquitous in modern data centers. By deploying applications on separate virtual machines hosted in a shared physical machine, it brings benefits over traditional systems in resources utilization[21, 36], system security[12, 13], and server management[28].

Despite the adoption and these advantages of virtualization, cloud providers, tenants, and developers face different challenges. The hardware and software options to build a virtualization platform pose a challenge for cloud providers. A variety of virtualization platforms make it difficult for tenants to make a choice. Developers are forced to continually improve the architecture and hypervisor design to better support virtual machines. In this context, evaluating the virtualization capabilities of different platforms is necessary.

Evaluating the capabilities of virtualization platforms is dramatically different from the traditional software stack. The introduction of the hypervisor changes the traditional software stack dramatically. In virtualized environments, double scheduling is a remarkable phenomenon that does not exist in native systems. Processes in the VM are scheduled first by the operating system (OS) and then by the hypervisor, which may increase the synchronization latency[32]. Memory virtualization requires a two-step address translations. One is guest virtual address (GVA) to guest physical address (GPA) translation. The other is GPA to host physical address (HPA) translation. The guest I/O requests go through the guest I/O stack and host I/O stack before accessing real devices. In addition to software changes, the prevalence of virtualization has spawned a revolution in architectures, especially hardware extensions for CPU and memory virtualization.

The virtualization-specific issues mentioned above are the key to study the capabilities of different virtualization platforms. However, current benchmarks that researchers use to evaluate virtualization performance may hide the effects caused by these issues. Application benchmarks, such as mysql, perform many complicated transactions and usually stress several subsystems simultaneously. The metrics that applications use, such as latency, throughput, utilization, do not have enough perception of the changes induced by the hypervisor. To make matters worse, performance optimization in one aspect is usually at the expense of performance degradation in other aspects, and the two cancel each other out eventually. An example of this problem is that some individual benchmarks in the SPEC CPU 2006[17] experienced performance fluctuations or even degradation among different QEMU versions[37]. Virtualization benchmarks, such as SPECvirt 2013[33], which are a combination of several common application benchmarks in data centers, focus only on server's consolidation capacity. Other benchmarks, such as lmbench[24], Unixbench[7], are initially designed for non-virtualized systems. Virtualization-specific issues are not within their consideration. What's more, almost all benchmarks need the support of a full Linux operating system. Whether the operating system itself has negative impacts on the test is uncertain.

To make up for the deficiencies of commonly used benchmarks, we present HyperBench, a benchmark suite for evaluating the virtualization capabilities of different platforms. It helps developers study the performance of hypervisors and architectures and helps

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

tenants and cloud providers choose the right virtualization platform. HyperBench is permitted to run on different fully virtualized platforms. This feature is achieved by designing HyperBench as a custom kernel which runs as a test VM. We make many efforts in the design of hypervisor benchmarks and the accuracy of measurement. To date, we design and develop 15 benchmarks covering CPU, memory, and I/O. Each benchmark in HyperBench triggers one or several hypervisor-level events and generally executes only tens of lines of C or assembler test code. To ensure the accuracy of measurement, we take out-of-order execution and interference from virtual CPU (VCPU) scheduling into account. In addition to this, we also amortize measurement errors by repeating benchmarks. With targeted benchmarks and reliable measurement results, one can verify whether the focused area induces much performance degradation.

This paper makes three contributions:

- We propose a cost model which formulates the time spent in handling hypervisor-level events.
- We design a set of hypervisor benchmarks to quantify important hypervisor-level events. Each benchmark contains one type of virtualization-sensitive operation that can trigger hypervisor-level events without touching underlying hypervisors.
- We develop a prototype of HyperBench. By designing the HyperBench as a custom kernel, it can accommodate different hypervisors. By carefully designing the memory layout of HyperBench, new benchmarks can be easily added. By defining a portable operating system interface for HyperBench (POSIH), HyperBench can be ported from x86 to other architectures.

The rest of this paper is organized as follows. Section 2 presents an overview of virtualization technologies. Section 3 proposes a cost model of the virtual machine. Section 4 describes the design of benchmarks in HyperBench. Section 5 describes the implementation of HyperBench. In section 6, we evaluate HyperBench. Section 7 summarizes some related works. Section 8 concludes the paper.

2 BACKGROUND

Virtualization is a technology that allows multiple guest operating systems to multiplex hardware resources within one physical machine. This section gives an overview of different techniques for virtualization, including CPU, memory, and I/O virtualization technologies.

2.1 CPU Virtualization

The hypervisor designers separate instructions into sensitive instructions and non-sensitive instructions. Sensitive instructions include control-sensitive and behavior-sensitive instructions. Control-sensitive instructions are those that attempt to change the configuration of resources in the system, such as the instruction to modify page table base register[30]. Behavior-sensitive instructions are those whose behavior or result depends on the configuration of resources, such as POPF in x86[30]. Different CPU virtualization techniques, such as trap-and-emulate, dynamic binary translation (DBT), hardware mechanism (AMD's SVM[11], Intel's VT[8]), are all developed to handle sensitive instructions correctly.

In a classically virtualizable architecture, all virtualization-sensitive instructions are a subset of privileged instructions[30]. The hypervisor on virtualizable architectures, which adopts the trap-and-emulate strategy, resides in privileged mode and runs guest OS in a low privileged mode. The hypervisor intercepts all privileged instructions and returns after finishing the emulation of the privileged instructions. Frequent traps consume much time of the virtual machine.

However, some architectures, such as x86, ARM, are not originally virtualizable. One solution is DBT. DBT translates non-sensitive instructions identically. The translator refers to a corresponding shadow structure when sensitive instructions try to change the privileged state. Guest code is translated only when it is about to execute, and the translated code is cached. DBT eliminates the traps in the classical hypervisors and replaces it with binary translation and jumps to the translated blocks. Another solution for virtualization on non-virtualizable architectures is hardware extensions. Take Intel's VT as an example. Intel introduces a new processor operation called Virtual Machine Extensions (VMX) operation. There are two kinds of VMX operation: VMX root operation and VMX non-root operation. Except for the VMX instructions, the two operations behave very similarly. In general, a hypervisor will run in VMX root operation, and guest software will run in VMX non-root operation. Thus, VMX operation eliminates both traps and translations for many sensitive instructions. However, some memory and I/O related exceptions still require the interference of the hypervisor, which involves a round-trip transition between the VMX non-root and root operation. Transitions into VMX non-root operation are called VM entries. Transitions from VMX non-root operation to VMX root operation are called VM exits. Transitions are implemented with a virtual-machine control data structure (VMCS) residing in normal memory. The frequency of transitions has a significant impact on the virtual machine performance[1, 2]. A guest will always run at native speed without transitions between VMs and the hypervisor.

Virtual interrupts are an integral part of CPU virtualization. Interrupt delivery in modern Intel processors is implemented through local Advanced Programmable Interrupt Controller (APIC). Each processor core is equipped with a local APIC that connected to other cores. Each local APIC has a set of registers that control the delivery of interrupts to the processor core and the generation of IPI messages. The primary local APIC register for issuing IPIs is the interrupt command register (ICR). The writing of virtual ICR causes a VM exit. Early local APIC(using xAPIC[8] architecture) registers are memory mapped. The new local APIC(using x2APIC[8] architecture), which is faster than xAPIC, uses model specific registers (MSR) programming interface to access registers. The virtual APIC is implemented through a virtual-APIC page resided in VMCS. To improve interrupt virtualization efficiency, Intel provides a posted-interrupt processing feature by which interrupt injection is done without a VM exit[27].

2.2 Memory Virtualization

For memory virtualization, almost all solutions require two sets of page tables. The guest operating system maintains the guest page tables which contain the mappings from GVA to GPA. Since the

GVA is ultimately mapped to the physical address, the mapping from GVA to HPA requires an additional set of page tables. There are two ways to accomplish this. One is to map directly from GVA to HPA, referred to as direct mapping. The other is to map from GVA to GPA/HVA and then map from GPA/HVA to HPA, referred to as indirect mapping. An example of the direct mapping is shadow paging. Although shadow page tables (SPT) can translate GVA directly into HPA, many hypervisor-level operations are required before SPT can be used. First, the hypervisor must trap the guest operation of switching page table root and point the hardware memory management unit (MMU) at a shadow page table. Second, when the guest updates its page tables, the hypervisor must intercept the update and modify the corresponding SPT entries. The third is the construction of shadow page table entries. The construction of page table entries in an additional set of page tables is a common source of performance degradation for both mapping methods. Intel's Extended Page Tables (EPT)[8] (AMD's Nested Page Tables (NPT)[11] has many similarities), an indirect mapping solution, eliminates synchronization between guest page tables and hypervisor-level page tables but introduces the two-dimensional (2-D) page walk[26] caused by TLB misses. Another example of indirect mapping is the software MMU in QEMU. QEMU allocates memory for virtual machines through the **mmap()** system call[3], during which the host virtual address (HVA) to HPA mapping is done. QEMU uses a software MMU to translate GVA into GPA that is equivalent to host virtual address[3]. The host virtual to physical address translation is done by the host OS with hardware MMU.

2.3 I/O Virtualization

With the standard device driver, all the I/O requests from the guest OSes are emulated by the hypervisor, which experiences both long data path and long control path. How to efficiently transfer I/O requests and data between virtual machines and devices is the key to optimize virtual I/O performance. Previous works also verify this point. virtio[31] is a popular solution for guest I/O. With the back-end driver in the hypervisor and the redesigned device drivers in VMs, the interactions between virtual machines and the hypervisor enjoy efficient event notification and data transfer. vhost[34] avoids the signaling between KVM and QEMU by moving back-end to kernel space. SplitX[20] and ELVIS[15] adopt hardware extensions and software scheme to realize exit-less notification between guests and the hypervisor, respectively. Direct device assignment simplifies the VM-to-device access path but interrupts from the device still hamper the performance of guest I/O. ELI[14] and DID[35] provide exit-less notification from the hypervisor to guests for direct device assignment.

3 COST MODEL

By reviewing the main virtualization techniques, we find that the hypervisor interposes on the guest execution frequently. The interference of the hypervisor consumes much time which is used to handle hypervisor-level events rather than running the application in the VM. Based on this point, we propose a cost model which describes the performance overheads of the virtual machine.

The cost model uses benchmark's execution time natively on the host as a baseline. As shown in equation 1, we define *GNR* (Guest Native Ratio) as the ratio of the benchmark runtime in the guest (T_{guest}) to the time taken natively on the host (T_{native}). In general, *GNR* is greater than 1. The smaller the *GNR*, the better the performance. *GNR* close to 1 indicates that the benchmark runs at native speed. In some cases, *GNR* is likely to be less than 1, which means the guest program outperforms the host program.

$$GNR = \frac{T_{guest}}{T_{native}} \tag{1}$$

The execution time of a guest application includes the necessary time taken on the host machine and additional time consumed by hypervisor-level events, which is represented by equation 2. T_{direct} represents the time spent on instructions that can be executed directly on the host machine. T_{virt} represents the runtime of instructions that need to be specially handled in a virtualized environment.

$$T_{guest} = T_{direct} + T_{virt} \tag{2}$$

The breakdown of T_{virt} is shown in equation 3. T_{cpu}, T_{memory}, and T_{io} represent the time for handling CPU, memory and I/O virtualization respectively. These components are described as follows. η denotes all the other overheads that are not mentioned here, for example, the mutual interference between co-located virtual machines. C represents the number of times a virtualization-sensitive operation occurred. The operator $\circledast$ represents the relationship between C and T and it will be discussed in section 6.

$$T_{virt} = T_{cpu} + T_{memory} + T_{io} + \eta \tag{3}$$

T_{cpu} represents the CPU virtualization overheads. The cost of CPU virtualization consists of the sensitive instruction cost (T_{sen}) plus the cost caused by virtualization extension instructions (T_{ext}), which can be represented by equation 4. Because the regular instructions are executed directly on the physical CPU, the performance of CPU virtualization is highly dependent on the platform's ability to handle sensitive instructions. In addition to sensitive instructions, the virtualization extension instructions induce pure performance overheads because they are used only by the hypervisor rather than operating systems.

$$T_{cpu} = C_{sen} \circledast T_{sen} + C_{ext} \circledast T_{ext} \tag{4}$$

T_{memory} represents the cost of memory virtualization. In the direct mapping, the additional latency comes from the switching of page table root (T_{switch}), the synchronization between the guest page tables and the page tables in the hypervisor(T_{sync}), and the construction of an extra set of page table entries (T_{cons}). The indirect mapping eliminates the synchronization but introduces 2-D page walk (T_{two}). Hence, T_{memory} can be represented by equation 5. For SPT, a direct mapping solution, T_{switch}, T_{sync}, and T_{cons} are the main cost and C_{two} is zero. For EPT, T_{cons} and T_{two} are the main cost, while C_{switch} and C_{sync} are zero.

$$\begin{aligned} T_{memory} = {} & C_{switch} \circledast T_{switch} + C_{sync} \circledast T_{sync} \\ & + C_{cons} \circledast T_{cons} + C_{two} \circledast T_{two} \end{aligned} \tag{5}$$

T_{io} represents the cost of I/O virtualization. The latency added to virtual I/O attributes to notifications in the two directions—from the I/O driver in the VM to the I/O device in the hypervisor (T_{out}), and the opposite direction (T_{in}). T_{io} can be represented by equation 6.

For QEMU-KVM, the notification from VM to a device involves a VM exit which is an essential event affecting I/O virtualization performance[40]. For Xen, the notification from VM to Dom0 involves a VM exit, a hardware IPI, and an interrupt injection.

$$T_{io} = C_{in} \circledast T_{in} + C_{out} \circledast T_{out} \quad (6)$$

In section 4, we will follow the cost model to design corresponding benchmarks. Each benchmark exercises one or several components of the cost model.

4 HYPERBENCH BENCHMARKS

According to the cost model in section 3, we designed 15 micro-benchmarks and conducted an in-depth analysis of these benchmarks on some typical virtualization platforms. These benchmarks are listed in Table 1 and divided into several groups, including privileged sensitive instruction, unprivileged sensitive instruction (critical instruction), exception, memory, and I/O. Table 1 also shows the recommended range of iteration count used for each benchmark. Beyond this range, HyperBench will run for a long time or the measurement will be inaccurate. We now discuss each of these benchmark groups or the individual benchmarks in each group. The purpose of the Idle benchmark is to help determine the iteration count of other benchmarks, which will be described in section 5.2.

4.1 Sensitive Instruction

Sensitive instruction benchmarks mainly exercise the virtualization events caused by handling sensitive instructions. Sensitive instructions can be separated into two categories: privileged sensitive instructions and unprivileged sensitive instructions (critical instructions). We select several of the privileged sensitive and unprivileged sensitive instructions as benchmarks, which are listed in Table 1. Although sensitive instruction benchmarks are architecture specific, the sensitive instruction benchmarks can be added easily (see section 5).

Unprivileged Sensitive Instruction: Unprivileged sensitive instructions only exist in non-virtualizable architectures. In the non-virtualizable architecture, there is no trap for these critical instructions. With the hardware-assisted virtualization, the VM executes critical instructions just as the native operating system does. With DBT, the execution of critical instructions triggers the translation process.

Privileged Sensitive Instruction: Except for hardware-assisted virtualization which allows privileged instructions to execute directly on the physical CPU, privileged instructions stress the virtualization system in different ways. For trap-and-emulate strategy, privileged instructions trigger traps. For dynamic binary translation, privileged instructions are translated into a sequence of regular instructions that refer to shadow structures.

4.2 Exception

Exception benchmarks trigger the virtualization-specific exceptions which require the interference of the virtualization extension instructions. These virtualization-specific exceptions include, for example, Hypercall, virtual inter-processor interrupt (IPI).

Hypercall: Hypercall is the transition between a VM and the hypervisor. For x86, the VMCS is saved and restored by hardware automatically when performing a Hypercall. It is a fundamental operation in trap-and-emulate hypervisors. Any operation that requires the interference of the hypervisor involves the Hypercall, such as the virtual IPI. Architecture developers are committed to reducing the hardware cost of Hypercall, and hypervisor developers are working to reduce the frequency of Hypercall[2].

IPI: IPI is a frequent operation in multi-core systems. The native system performs IPI through local APIC interface. When a core wants to send an IPI, it writes ICR with the interrupt vector, destination, etc. Once the write is complete, the IPI message appears on the system bus or the APIC bus. The receiving core determines if it is the specified destination or not. If it is the specified destination, it accepts the IPI message and calls interrupt handler routine. Once the handler routine is complete, the receiving core writes the end-of-interrupt (EOI) register in the local APIC. In addition to a necessary hardware IPI, the virtual IPI involves many world switches between the VM and the hypervisor. First, the VM exit caused by writing ICR is necessary. Second, when the destination core receives this IPI, a VM exit will occur on a system that doesn't have posted interrupts. When the posted-interrupt processing feature is enabled, the destination core receives the virtual IPI without a VM exit.

4.3 Memory

Memory benchmarks mainly activate the GVA to HPA translation processes and related operations, such as the creation of hypervisor-level page tables, synchronization between guest page tables and hypervisor-level page tables. Doing different patterns of memory access is a straightforward method to trigger these events. Accessing a page that has ever accessed before may stress the virtual TLB. Accessing a memory region that has never accessed before may cause the construction of hypervisor-level page table entries. However, accurately measuring T_{memory} is a huge challenge without modifying the underlying hypervisor. For example, T_{two} is dependent on a number of factors, namely TLB and cache hierarchy, page mapping size in both the guest and the host, paging structure caches, and benchmark working set size[26]. Nevertheless, we can roughly evaluate the items in the cost model by examining custom micro-benchmarks on both native and virtual execution environments. Since POSIH (see section 5) provides functions for guest configuration, such as installing page table entries with different mapping granularity, we can verify the impact of the factors mentioned above by controlling variables.

Hot Memory Access: The spatial locality and temporal locality of memory make hot memory access universal in real applications. The virtual TLB should ensure that hot memory access has a higher hit rate as soon as possible. The intuition is that the higher the TLB hit rate, the smaller the performance difference between the hardware TLB and the virtual TLB. In this benchmark, the working set size, page mapping size is tunable. As the working set size grows larger, it will overwhelm the TLB and more memory accesses trigger a page walk. While using large page mappings can reduce TLB misses, both the guest and the host must use 2MB pages to allow the processor to use 2MB TLB entries in a virtualized environment[26]. One can verify this effect by adjusting the guest page size and the hypervisor page size.

Table 1: Benchmarks in HyperBench

Category	Benchmark	Iteration Range	Description
Idle	Idle	10-1M	Idle benchmark performs two consecutive reads of the time counter. It is used to check the stability of the measurement results. Ideally, the result is zero.
Unprivileged Sensitive Instruction	SGDT SLDT SIDT	10K-10M 10K-10M 10K-10M	Store the corresponding register value into memory repeatedly.
	SMSW	10K-10M	Store the low-order 16 bits of register CR0 into memory.
	PUSH-POPF	10K-10M	The PUSHF and POPF instructions execute alternately on the current stack. The time between the first PUSHF instruction and the last POPF instruction is measured.
Privileged Sensitive Instruction	LGDT Set CR3	10K-10M 10K-10M	Read the current value of the register during the initialization phase and load the value into the corresponding register repeatedly in the test phase.
Exception	Hypercall	1-1K	Execute an instruction in the VM which leads to a transition to the hypervisor and return without doing much work in the hypervisor.
	IPI	1-1K	Issue an IPI from a CPU to another CPU which is in the halt state. IPI benchmark measures the time between sending the IPI until the sending CPU receives the response from the receiving CPU without doing much work on the receiving CPU. In the virtualized environment, this benchmark emulates an IPI between two VCPUs running on two separate physical CPUs (PCPUs).
Memory	Hot Memory Access	10-100K	Read many different memory pages twice and the time of the second memory access is measured. The default guest page size is 4KB.
	Cold Memory Access	10-100K	This benchmark reserves a large portion of memory that has never been accessed before and performs one memory read at the start address of each page. The reading over different pages eliminates TLB hits due to the prefetcher, as the prefetcher cannot access data across page boundaries. The default guest page size is 4KB.
	Set Page Tables	1	Map the whole physical memory 1:1 to the virtual address space. This benchmark creates a lot of page table entries, which is a frequent operation in heavy memory allocation. The default guest page size is 4KB.
I/O	IN	1K-10M	Polling and interrupt are two main approaches for notifications from host to guest. This benchmark reads the register of the serial port through the register I/O instructions repeatedly, which emulates the polling mechanism.
	OUT	1K-10M	OUT benchmark outputs a character to the register of the serial port repeatedly.
	Print	10-1K	This benchmark outputs a string to the serial port through the I/O address space, which is handled through the string I/O instructions.

Cold Memory Access: Cold memory access intends to trigger TLB misses or hypervisor-level page faults. The cold memory access in the QEMU virtual machine only causes the two-dimensional page walk because the virtual machine memory is allocated during the startup phase. Since some hypervisor-level page tables, such as SPT and EPT, is empty at the beginning, cold memory access causes the hypervisor to construct the corresponding page table entries. Take EPT/NPT as an example. Cold Memory Access causes two-dimensional page walk and the construction of EPT/NPT entries. By running this benchmark on both native and virtual execution environment, we can evaluate T_{memory} for EPT/NPT.

Set Page Table: This benchmark essentially writes a large number of page table entries to a memory region, which involves both hot memory access and cold memory access. Set Page Table benchmark emulates a frequent operation in heavy memory allocation.

4.4 I/O

I/O benchmarks mainly exercise the notification mechanisms between guest and host. The key to the I/O benchmark is to find a standard test interface. A hypervisor may support I/O requests from guests through emulation, para-virtualization, assignment, and I/O device sharing[9]. One standard test interface among these models

is the interface between applications and the operating system. However, applications are so high-level that complex operations may swallow performance differences between these models. Any model ends up accessing real physical devices through port I/O (PIO) or memory-mapped I/O (MMIO). Performing a series of I/O ports accesses within HyperBench kernel can activate notification mechanisms.

IN: Polling and interrupt are two main approaches for the notification from host to guest. This benchmark reads the register of the serial port through the register I/O instruction repeatedly, which emulates the polling mechanism. For QEMU, I/O instructions are emulated by itself. For KVM with QEMU, before emulating the I/O instruction in QEMU, the VM traps to host kernel and then transfer control to QEMU. For Xen, IN benchmark triggers the following actions: (1) the VM traps to Xen, (2) Xen issues an IPI to Dom0, (3) transfer control to QEMU and emulate the I/O instruction, (4) return to the VM.

OUT: As a counterpart of IN, OUT benchmark outputs a character to the register of the serial port repeatedly. This benchmark focus on the added latency caused by notification mechanism from guest to host.

Print: While OUT focuses on a single character, Print focuses on a string. This benchmark outputs a string in memory to the serial port through the I/O address space, which is handled through the string I/O instructions.

5 IMPLEMENTATION

HyperBench possesses two properties of interest: extensibility and portability. Extensibility allows for the addition of new benchmarks, while portability makes it possible to port HyperBench to other architectures. This section describes the architecture of HyperBench and how to accurately measure the benchmarks.

5.1 Architecture

The cost model described in section 3 requires HyperBench benchmarks to run in both native and virtualized environments. Figure 1 shows HyperBench in different scenarios. HyperBench kernel is an ELF file in multiboot format, which allows grub to boot it directly. HyperBench kernel resides in a hard disk in which grub is correctly configured. In the native environment, HyperBench kernel runs directly on the hardware. The benchmark measurements are finished within the HyperBench kernel, referred to as internal measurement. In the virtualized environment, HyperBench kernel runs as a test VM. On a standalone bare-metal hypervisor, such as Xen, the timing application runs in the privileged VM. On a hosted hypervisor, such as KVM, the timing application runs as a regular application on the host OS. The benchmark measurements can be implemented either by internal measurement or by the timing application. The latter is referred to as external measurement. With external measurement, HyperBench kernel sends timing signals to UART immediately before and after the benchmark. The UART is redirected to the external timing application through the pipe between processes. For QEMU, timing signals are output to the emulated serial port which is redirected to the console with *-nographic* option. Once the timing signal is received, the timing application will read the current value of the counter. In a virtualized environment, internal measurement is mainly for regular users because they don't have enough permissions to start the timing application on the host.

We designed and implemented HyperBench kernel as a standalone kernel without any modification of the underlying hypervisor. To reduce the difficulty of development, we borrowed some code of kvm-unit-tests[6], xv6[38] and simbench[37]. A standalone kernel was facilitated by a number of reasons. First, each micro-benchmark consumes very few clock cycles and is susceptible to resources contention within a regular Linux kernel, such as lock contention[19], packet scheduling[18, 39]. With a standalone kernel, we can ensure that the current benchmark is not interfered by other tasks. Second, the custom kernel provides many HyperBench specific functions, such as benchmark measurements, setting whole memory, accessing I/O ports, etc., which allows us to customize micro-benchmarks to the greatest extent. Last but not least important, HyperBench is no more than 10K lines of C and assembly code. However, building all benchmarks and supporting functions as a standalone kernel makes HyperBench less portable, harder to maintain, and unlikely to ever support new processor features. We solve this problem in the traditional UNIX way of defining a portable operating system interface for HyperBench (POSIH) which is actually an API. Some important functions in POSIH are shown in Table 2. There are four POSIH categories in HyperBench: (1) symmetric multiprocessing (SMP), (2) interrupt, (3) memory, (4) I/O.

Portability. The architecture of the HyperBench kernel is displayed in Figure 2. HyperBench benchmarks span CPU, memory, and I/O (see section 4). While different benchmarks focus on different features, all benchmarks share the underlying POSIH. POSIH is based the underlying drivers that interact with the hardware directly. Therefore, HyperBench can run on any architecture as long as the drivers on the target architecture is implemented. Although some benchmarks that are closely related to instruction set architecture (ISA) require modification, the benchmark usually has only tens of lines of C or assembler test code.

Extensibility. A significant advantage of Hyperbench over existing benchmarks is that adding a new benchmark is pretty easy. This advantage is achieved by POSIH and carefully designed memory layout. Benchmarks are based on POSIH. When adding a new benchmark, many functions in POSIH can be reused. Under the control of a carefully designed linker script, the memory layout of HyperBench kernel is shown in Figure 3. Address space from **_HEAP_START** to the end is used for memory allocation and memory test. The **.benchmarks** section is a descriptor table into which all data structures that describe HyperBench benchmarks are mapped. The benchmark structure looks like so:

```
typedef void (*function_t)();
typedef struct {
    const char *name;
    const char *category;
    function_t init;
    function_t benchmark;
    function_t benchmark_control;
    function_t cleanup;
    uint64_t iteration_count;
} benchmark_t;
```

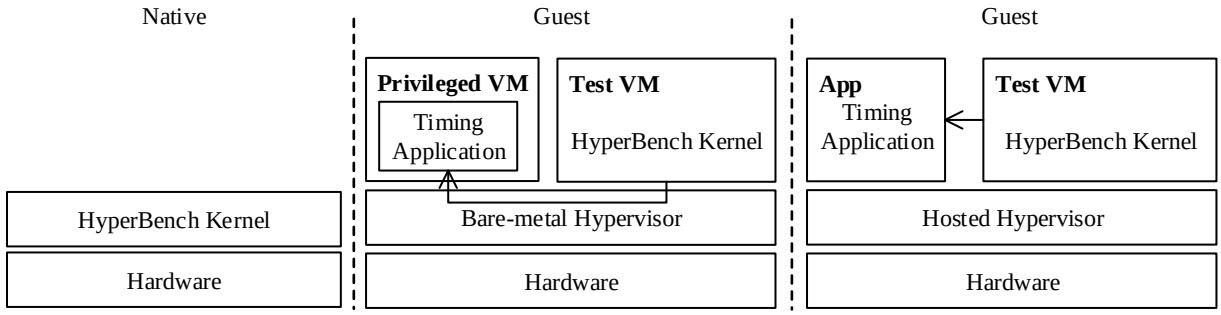


Figure 1: HyperBench in Native and Virtualized Environments.

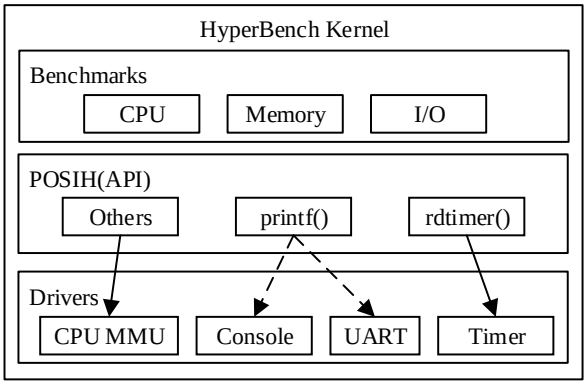


Figure 2: HyperBench Kernel.

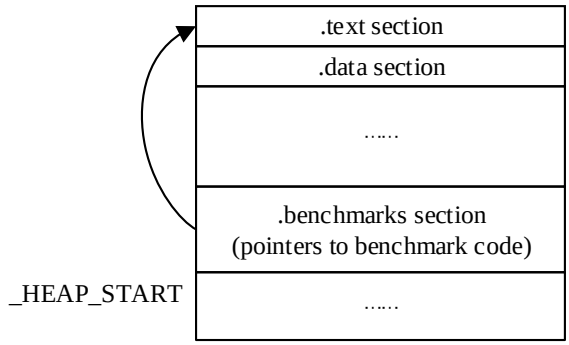


Figure 3: The Memory Layout of HyperBench Kernel.

Table 2: A Brief Description of POSIH

SMP	
startothers	Boot and initialize secondary cpus.
spinlock/spinunlock	Synchronization functions.
Interrupt	
set_idt_entry	Set up exception handler and enable interrupt.
on_cpu	Send an IPI to a CPU.
Memory	
early_mem_init	Get the memory layout and prepare free memory for allocation.
setup_mmu	Map the physical memory to the virtual memory.
install_pte	Install a page table entry with the specified granularity, such as 4KB, 2MB, etc.
heap_alloc_page	Allocate one 4KB or 2MB physical memory page.
mem_tlb_flush	Flush TLB.
I/O	
in/out	Read/write I/O ports.
write/read	Read/write memory-mapped I/O.
printf	Output to the console or UART.
rdtimer	Read the value of the virtual/hardware counter.

Each descriptor contains the name of the benchmark, its category, iteration count, and several function calls. The **init** call is used to prepare a runtime environment for the benchmark. The **benchmark** call repeats the virtualization-sensitive operation that triggers hypervisor-level events. Usually, the virtualization-sensitive operation is wrapped by a **for** loop and the iteration count is determined by the **iteration_count**. The **benchmark_control** points to an idle loop function whose iteration count is the same as the current benchmark. It is used to offset the runtime of loop statements which must not be counted into the runtime of each benchmark. The **cleanup** call cleans up the runtime environment. The **main** function in HyperBench kernel will retrieve each descriptor in this

section. Hence, a ready benchmark decorated with **.benchmarks** will be executed.

5.2 Measurement

There are two alternatives to measure benchmarks, namely internal measurement, and external measurement. However, neither method is inherently perfect. The timer within the HyperBench kernel may not be accurate or adequately calibrated in virtualized environments. Although the host timer is more accurate than the virtual timer, the notification from HyperBench kernel to the timing application has a certain degree of variation. The stability of the time between HyperBench kernel sends a timing signal until the timing application finishes reading the hardware counter determines the accuracy of measurement results. To settle this problem, we designed an Idle benchmark which performs two consecutive reads of the counter. According to the result of the Idle benchmark, iteration of the same or higher order of magnitude can counteract fluctuations. Table 3 gives the recommended iteration count used for each benchmark. The value is obtained by running each benchmark as many times as possible in a reasonable amount of time.

In the context of measuring performance on multi-core systems, variations caused by interrupts and scheduling can skew measurements by thousands of cycles. To further improve the accuracy of external measurements, out-of-order execution and the interference from multi-core scheduling are taken into account. We isolate a subset of physical CPUs by configuring grub and pin each VCPU to a PCPU in that subset. Instruction barriers are used immediately before and after taking timestamps to avoid out-of-order execution or pipelining from skewing our measurements[29].

To exclude as many operations as possible that we are not interested in, we compare the time consumed by **benchmark** function and **benchmark_control** function. The only difference between the two functions is whether they contain virtualization-sensitive operations. As shown in Figure 4, the runtime of each benchmark is measured in eight phases. First, the **init** function prepares the runtime environment for each benchmark. Previous to that, the HyperBench kernel boots and each VCPU is pinned to a specific PCPU that has been set aside to ensure that no other work is scheduled on that PCPU. At step 3, the benchmark is repeated at a predefined number of times. Immediately before and after that, a timing signal is delivered to timing application when external timing application is enabled. On receiving a timing signal, the timing application obtains the cycle counts by reading hardware counter (step 2b and step 4b). When the internal timing application is enabled, HyperBench kernel measures benchmarks by itself (step 2a and step 4a). A similar timing process is applied to **benchmark_control** (step 5, step 6 and step 7) to eliminate the impact of the supporting operations, such as the **for** statement. As a final step, HyperBench reports the test results in cycles and all benchmark specific environments are cleaned up. The difference between the runtime of the **benchmark** function and the **benchmark_control** function is the benchmark runtime about which we care.

6 EVALUATION

We ran HyperBench on both native and virtual execution environments. The hypervisors under test include QEMU, KVM, and Xen. In KVM, we also compared EPT with SPT. After introducing the experimental environment, we first discuss the operator $\circledast$ in the cost model, which represents the relationship between C and T. We then demonstrate the effectiveness of HyperBench in sensing the hardware acceleration for virtualization and differences in the hypervisor design.

6.1 Experimental Setup

For a fair comparison, the hardware settings were the same for all hypervisors by using the same server machine which is a Lenovo RQ940 with four octa-core 2.2GHz Intel Xeon E7-4820v2 (based on Ivy Bridge micro-architecture) processors, 32GB of RAM, and three 1TB disks. Xen occupied one disk, QEMU and KVM shared one same disk. The host used Ubuntu14.04 with Linux 4.2.0-27-generic kernel and GNU GRUB 0.97. Hugepage size is 2MB, which is disabled by default. The versions of hypervisors under test were QEMU 2.0.0, KVM in Linux 4.2.0-27-generic, and Xen 4.4.2. Xen was configured to use HVM type of guests. All VMs were configured as SMP system with 2 CPUs and 4GB of RAM. In QEMU and KVM, HyperBench kernel was assigned to run on a separate set of PCPUs, and each VCPU was pinned to a PCPU. In Xen, we have similarly configured DomU. In addition, we configured Dom0 with one VCPU to run on a dedicated PCPU. In KVM and Xen, QEMU was used as the device emulator. Measurements were performed by the HyperBench kernel itself on the host machine and by the host timing application in virtualized environments.

6.2 Operator Determination

According to the cost model described in section 3, we can conclude that one of the hypervisor optimization approaches is reducing the number of virtualization-sensitive events. Merging some hardware virtualization exits with software techniques breaks the one-to-one relationship between exits and the number it occurs within intact hypervisors[2]. Another example is para-virtual IPI[22], which uses a hypercall to send IPIs to multiple VCPUs. A batch of IPIs require only one VM exit. Therefore, the relationship between C and T, which is represented by the undetermined operator in the cost model, is an important characteristic of hypervisors. In order to verify whether the tested hypervisors has similar characteristics, we ran HyperBench with variable iterations on both host machine and hypervisors under test. Figure 5 shows the results with three different iterations. The results are normalized relative to $iteration1$. Intuitively, one would expect the virtualization overheads are proportional to the number of virtualization-sensitive operations. Figure 5a and Figure 5c show that the change of iteration numbers does not have much effect on the average cycles consumed by each virtualization-sensitive operation. Figure 5b and Figure 5d show a slight inconsistency. To determine the cause of the inconsistency, we conduct the experiment with more iterations. As the iteration numbers increase, the inconsistency shrinks. Hence, the cost model of QEMU, KVM, and Xen is roughly linear.

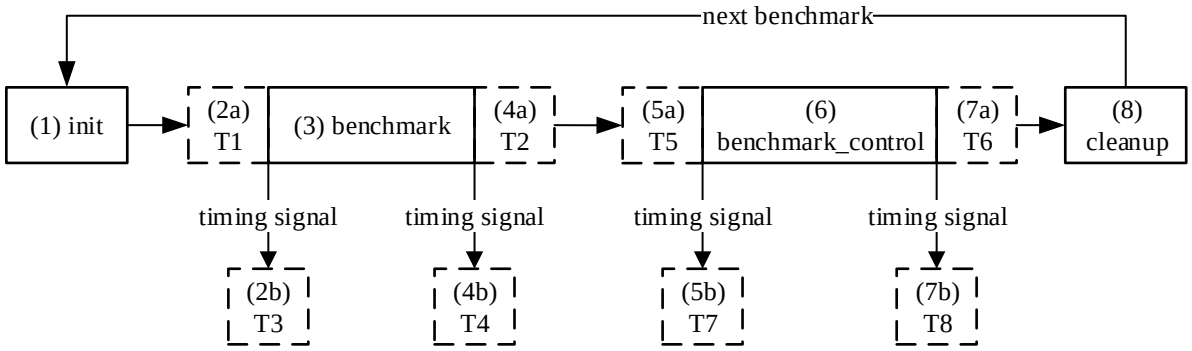
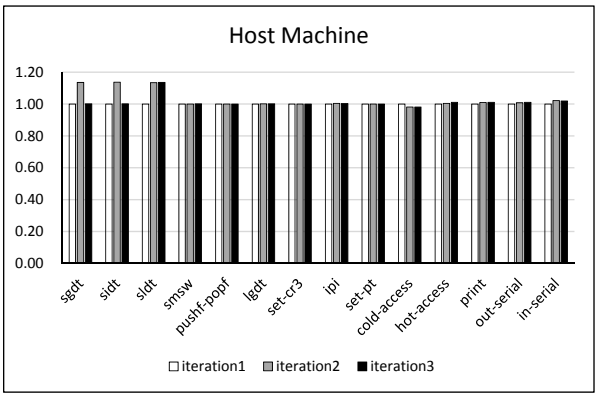
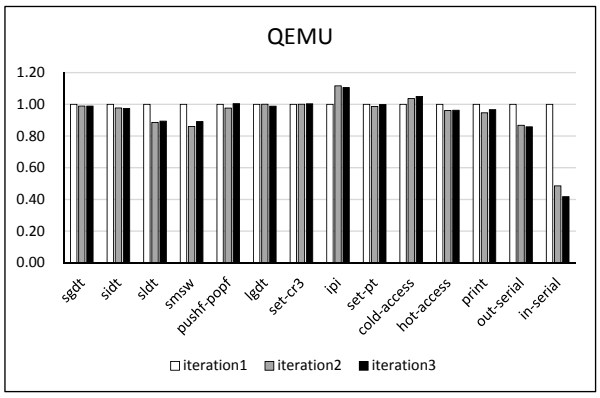


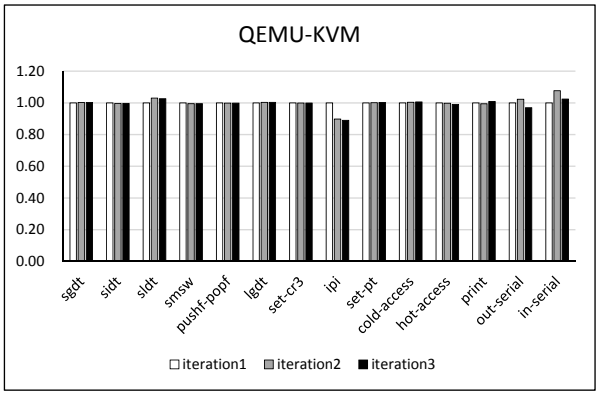
Figure 4: HyperBench Execution Flow.



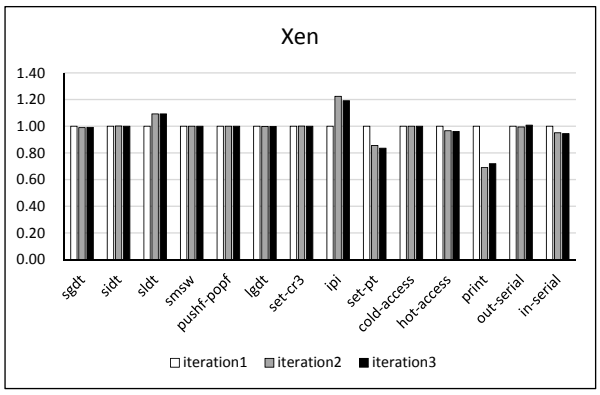
(a) Host Machine



(b) QEMU



(c) QEMU-KVM



(d) Xen

Figure 5: Normalized Performance with Different Iterations (iteration1 < iteration2 < iteration3).

6.3 Effectiveness

Using HyperBench, we ran 15 benchmarks that measure various aspects of low-level hypervisor performance, as listed in Table 1. Table 3 shows the results from running these benchmarks on both a physical machine and several popular hypervisors. These results are shown in the format: ((cycles per iteration) (slowdown relative to native execution)).

The Idle benchmark results in Table 3 indicate that the fluctuation of measurements is at the level of a few thousand to tens of thousands of cycles. The iteration count of majority benchmarks is over 100K, which largely reduces the effect of fluctuation. The physical machine does better than virtualization platforms for the Idle benchmark because the host reads the hardware counter directly.

The previous study has shown that compute-intensive benchmarks run at nearly native speed on both software and hardware hypervisors[1]. But this does not mean that the virtualization platform and the native system behave exactly similarly. The differences in CPU virtualization capabilities between software and hardware hypervisors can be reflected in HyperBench benchmarks. Host machine, KVM, and Xen behave similarly and outperform QEMU in all sensitive instruction benchmarks except for SMSW. The similarity among host machine, KVM, and Xen is determined by the same hardware support for CPU virtualization. The vast differences between QEMU and other platforms in sensitive instruction benchmarks are due to code translation and the quality of translated code. The behavior of SMSW is determined by the CR0 guest/host mask and the CR0 read shadow in a VMCS. In KVM, every bit is set in the CR0 guest/host mask during the initialization of VMCS. SMSW returns the value of the CR0 read shadow on KVM and reads normally from CR0 on the host machine. Therefore, SMSW benchmark behaves differently on host machine and KVM.

For Hypercall benchmark, KVM and Xen do not show significant performance differences because both KVM and Xen leverage the same x86 hardware mechanism for transitioning between the VM and the hypervisor. Since the receiving VCPU is in the halt state, IPI on KVM costs more than one hardware IPI and one Hypercall. The results of IPI and Hypercall on KVM confirm this point. We also send an IPI to a running CPU on KVM. The IPI consumes 3375 clock cycles, which costs a Hypercall and a hardware IPI. This indicates that the posted-interrupt processing feature is enabled on KVM x86. However, Xen takes more cycles than KVM for IPI. With the same hardware support, this is attributed to the difference in hypervisor settings. By running the **xl demsg** command, we found that Xen failed to enable interrupt remapping and will not enable x2APIC. For QEMU, IPI is a much more expensive operation than KVM and Xen, which indicates that DBT behaves poorly in handling IPI.

HyperBench also performs well in exposing the memory virtualization capabilities of the platform. When KVM and Xen both use EPT, they suffer a certain degree of performance degradation for the construction of EPT entries and 2-D page walks. The Cold Memory Access results show an approximately 1.4x slowdown. For Cold Memory Access, T_{memory} is at the level of 300 clock cycles on KVM and Xen. For QEMU, although the translation from GPA to HPA is done by both a relatively slow software MMU and a hardware MMU, there are no VM exits during memory accesses. The final result is that QEMU enjoys a slight performance advantage over KVM and Xen. Hot Memory Access benchmark shows that x86 KVM and x86 Xen have the same capability as the host in TLB virtualization. With the help of EPT, the direct mappings from GVA to HPA is likely to be cached[8]. In QEMU, only the mappings from host virtual address (equivalent to GPA) to HPA may be cached. For Hot Memory Access on QEMU, the GVA to GPA translations miss TLB while the HVA to HPA translations is likely to hit TLB. Therefore, the performance of Hot Memory Access on QEMU is still poor but better than Cold Memory Access. The Set Page Table benchmark, most of which are hot memory access, is more than an order of magnitude more expensive for QEMU than for KVM and Xen.

Many memory-related benchmarks suffer performance degradation when EPT is turned off in KVM. The Set CR3 benchmark with SPT is accompanied by a hypercall every time the guest loads CR3 and is about 10x slower than that with EPT. This hypercall points hardware MMU at the shadow page table and the GVA to HPA mappings may be cached during memory accesses. With the GVA to HPA mappings in TLB, Hot Memory Access is not affected too much by turning off EPT. IPI and Hypercall consume more clock cycles due to access to VMCS which resides in normal memory. Thus, I/O benchmarks which involve IPI or Hypercall are affected to varying degrees.

The I/O benchmarks reflect the impact of hypervisor design on the virtualization capabilities of the platform. In both KVM and Xen, the I/O devices are emulated by QEMU in user space. While the serial port emulated by QEMU performs better than the physical port, I/O benchmarks are much worse on KVM and Xen than on host machine. The signaling mechanism between the VM and the hypervisor offsets the advantage of QEMU and results in the large difference. In KVM, there are two main context switches after the HyperBench kernel initiates an I/O request. First, a transition between HyperBench kernel and KVM on the same physical core. Second, because KVM runs in kernel space, signaling QEMU, which runs in user space, requires another context switch. In Xen, as the HyperBench kernel and Dom0 run on different physical cores, Xen must send a physical IPI from the CPU running the HyperBench kernel to the CPU running Dom0. Xen has a longer path of notifying QEMU to emulate I/O requests than KVM. Hence, Xen is much slower than KVM for I/O benchmarks.

HyperBench can sense different CPU configurations. In KVM, as the number of iterations increases, the average time consumed by IPI changes irregularly without any VCPU pinning. The reason for this phenomenon is that the VCPU is scheduled on different PCPUs over time. In the case where all VCPUs are pinned, the average time of IPI is stable. In Xen, we conduct the experiment with two different configurations: **SEPARATE** represents that Dom0 and the HyperBench kernel are pinned on two separate physical CPU, while **SHARE** represents that Dom0 and the HyperBench kernel share the same physical CPU. Surprisingly, IPI with **SHARE** configuration consumes 3021x more clock cycles than that with **SEPARATE** configuration. Figure 6 compares the rest of the benchmarks under the two settings. IPI is a part of the control path between I/O driver in the VM and QEMU in Dom0, which results in about 50% performance degradation for I/O benchmarks.

Table 3: Cycles Spent on Each Micro-benchmark. The Results are in the Format: ((Cycles per Iteration) (Slowdown Relative to Native Execution))

Category	Benchmark	Host machine		QEMU		KVM (EPT)		KVM (SPT)		Xen	
Idle	Idle	-1684	-	23646	-	24206	-	-5546	-	-3256	-
Critical Instruction	SGDT	9	1	2178	242	9	1	9	1	9	1
	SIDT	9	1	2186	243	9	1	9	1	9	1
	SLDT	9	1	79	8.8	9	1	9	1	9	1
	SMSW	10	1	1	0.1	7	0.7	7	0.7	7	0.7
	PUSHF-POPF	24	1	130	5.4	24	1	24	1	24	1
Privileged Instruction	LGDT	127	1	31	0.2	127	1	127	1	127	1
	Set CR3	215	1	7716	35.9	297	1.4	3178	14.8	296	1.4
Exception	Hypercall	-	-	-	-	1470	-	2677	-	1430	-
	IPI	2409	1	128486744	53336	5741	2.4	6435	2.7	6368	2.6
Memory	Hot Memory Access	87	1	892	10.3	89	1	88	1	91	1
	Cold Memory Access	834	1	1017	1.2	1150	1.4	11245	13.5	1134	1.4
	Set Page Table	141254312	1	1725967620	12	141628640	1	185716972	1	145163668	1
I/O	IN	3351	1	328	0.1	8370	2.5	12325	3.7	36060	10.8
	OUT	3351	1	388	0.1	6239	1.9	9684	2.9	35532	10.6
	Print	84397	1	94503	1.1	513149	6	663393	7.9	2113049	25

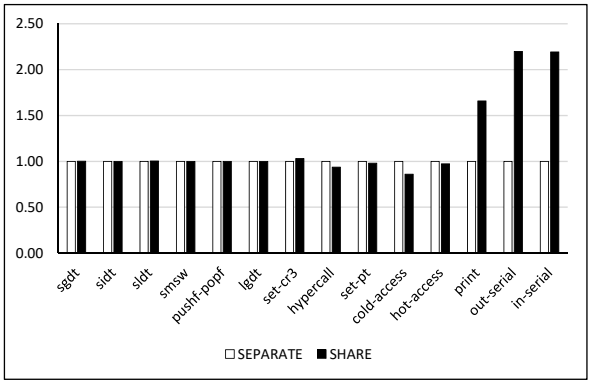


Figure 6: Normalized Xen Performance with Different CPU Configurations.

Table 4: Conventional Benchmarks

Category	Benchmark
compute intensive	SPEC CPU 2006, ByteMark, PARSEC[4]
memory intensive	Stream, Ramspeed, Cachebench
disk I/O intensive	fio, bonnie++, IOZone
network I/O intensive	netperf, Apachebench
other benchmarks	lmbench, Unixbench, Sysbench

7 RELATED WORK

After decades of development, virtualization has become a well-researched area. Many benchmarking approaches in evaluating the performance of various virtualization platforms have been developed. We roughly classify the existing benchmarks as micro-benchmarks and macro-benchmarks according to granularity. Macro-benchmarks evaluate the overall performance of actual applications under loosely defined conditions. Micro-benchmarks measure the performance of individual primitive operations under well-defined conditions.

7.1 Macro-benchmarks

Many researchers generally use traditional benchmarking tools to evaluate the performance of the virtualization platforms. The most widely used conventional benchmarking tools are listed in Table 4, such as lmbench[24], SPEC CPU 2006, etc. Benchmarks used in the virtualized environment are entirely different from that in the bare-metal system. Due to the introduction of the hypervisor, running benchmarks in a guest operating system may cause many hypervisor-level events, such as the transitions between the hypervisor and VMs, 2-D page walk, and code translation. However, traditional benchmarks, which are initially designed for the non-virtualized system, are not aware of hypervisor-level events. The unawareness makes it difficult to answer questions related to virtualization.

VMmark[23], vConsolidate[5] and SPECvirt 2013[33] are virtualization benchmarks that focus on server's consolidation capacity. These tools combine several typical datacenter workloads, such as mail server, java server, etc., as a unit of work. The total number of units that a physical system and virtualization layer can accommodate gives a coarse grain measure of that system's consolidation capacity. This kind of virtualization benchmarks focuses only on server's consolidation capacity. It is challenging to change the workload type according to specific test requirements. Workloads used in the work unit are real applications. The synthetic results cannot capture small variations among different hypervisors.

7.2 Micro-benchmarks

To better discover the performance differences between native and virtual systems, two performance evaluation toolkits that are specific to the Xen hypervisor are published. XenMon[16] is a QoS monitoring and performance profiling tool for the Xen-based virtual environment. It makes use of the existing Xen tracing feature to provide a fine-grained report of various domain related metrics. XenMon reports several metrics, such as time metrics(CPU usage, blocked time, waiting time), execution count, and I/O count. Xenoprof[25] is an extension of Oprofile. By virtualizing the hardware performance counters, Xenoprof can monitor the hardware events from the view of each domain. Unfortunately, the hardware performance counter virtualization is extremely costly, which will affect the accuracy of measurements in turn. One major limitation of this type of tools is that the comparison between different hypervisors is not permitted.

In addition to profiling and monitoring tools, there are also some works that studied the virtualization performance through fine-grained benchmarks. Dall et al.[10] proposed several micro-benchmarks to qualify important low-level interactions between the hypervisor and the ARM hardware support for virtualization. Adams and Agesen[1] wrote a series of nano-benchmarks that each exercises a single hypervisor-level event. While these fine-grained benchmarks can help to understand the performance anomalies of the hypervisor, there is no standard benchmarking tools or the hypervisor source code is modified. What's more, whether these benchmarks are compatible with different hypervisors is questionable.

8 CONCLUSION

We have proposed and evaluated HyperBench, a hypervisor benchmark suite for analyzing how much hardware mechanisms and hypervisor designs support a virtual machine. HyperBench is designed and implemented as a purpose-built kernel, which eliminates the side effects from OS.

A challenge in the design of HyperBench is the selection of the benchmarks. By reviewing techniques used in hypervisors, we proposed a cost model that formulates the time stolen by hypervisor-level events. Each benchmark in HyperBench exercises one component of the cost model. What's more, only tens of lines of C or assembler code is executed in each benchmark. With targeted benchmarks and short executable code, developers can verify whether the focused area incurs much performance degradation.

The evaluation, which is conducted by running HyperBench on QEMU, KVM, Xen, and host machine, demonstrates the HyperBench's capabilities to sense the hardware acceleration for virtualization and differences in hypervisor designs. HyperBench is a complement to the application benchmarks. Application benchmarks focus on high-level metrics, while HyperBench focuses on hypervisor-level events.

HyperBench is designed to be extensible and portable. Currently, HyperBench is implemented on x86 and contains 15 micro-benchmarks. Future work includes the development of new micro-benchmarks as well as porting HyperBench to other architectures (e.g. ARM). New micro-benchmarks may focus on the issues that are not illustrated in detail in the cost model, such as performance interference caused by co-resident VMs.

To facilitate further exploration, we release HyperBench and more details at https://bitbucket.org/Second2None/hyperbenchv2/src/master/.

REFERENCES

[1] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGARCH Computer Architecture News* 34, 5 (2006), 2–13.

[2] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. 2012. Software Techniques for Avoiding Hardware Virtualization Exits. In *USENIX Annual Technical Conference*. 373–385.

[3] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 72–81.

[5] Jeffrey P Casazza, Michael Greenfield, and Kan Shi. 2006. Redefining server performance characterization for virtualization benchmarking. *Intel Technology Journal* 10, 3 (2006).

[6] KVM contributors. 2017. KVM-unit-tests. Retrieved February 18, 2019 from https://www.linux-kvm.org/index.php?title=KVM-unit-tests&oldid=173824

[7] Unixbench contributors. 2018. Unixbench. Retrieved February 18, 2019 from https://github.com/kdlucas/byte-unixbench

[8] Intel Corporation. 2017. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[9] Intel Corporation. 2017. *Intel® Virtualization Technology for Directed I/O*.

[10] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. 2016. ARM virtualization: performance and architectural implications. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 304–316.

[11] Advanced Micro Devices. 2005. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*.

[12] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 51–62.

[13] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. 2003. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 193–206.

[14] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: bare-metal performance for I/O virtualization. *ACM SIGPLAN Notices* 47, 4 (2012), 411–422.

[15] Abel Gordon, Nadav Har'El, Alex Landau, Muli Ben-Yehuda, and Avishay Traeger. 2012. Towards exitless and efficient paravirtual I/O. In *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 10.

[16] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. 2005. Xenmon: Qos monitoring and performance profiling tool. *Hewlett-Packard Labs, Tech. Rep. HPL-2005-187* (2005), 1–13.

[17] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[18] Chi-Yao Hong, Matthew Caesar, and P Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 127–138.

[19] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. 2012. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 9.

[20] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. 2011. SplitX: Split Guest/Hypervisor Execution on Multi-Core. In *WIOV*.

[21] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. 2011. Validating heuristics for virtual machines consolidation. *Microsoft Research, MSR-TR-2011-9* (2011), 1–14.

[22] Wanpeng Li. 2018. Torwards a more Scalable KVM Hypervisor. KVM Forum.

[23] Vikram Makhija, Bruce Herndon, Paula Smith, Lisa Roderick, Eric Zamost, and Jennifer Anderson. 2006. *VMmark: A scalable benchmark for virtualized systems*. Technical Report. Technical Report TR 2006-002, VMware.

[24] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.

[25] Aravind Menon, Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Willy Zwaenepoel. 2005. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM, 13–23.

[26] Timothy Merrifield and H Reza Taheri. 2016. Performance implications of extended page tables on virtualized x86 processors. *ACM SIGPLAN Notices* 51, 7 (2016), 25–35.
[27] Jun Nakajima. 2012. Enabling optimized interrupt/APIC virtualization in KVM. KVM Forum.
[28] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. 2005. Fast Transparent Migration for Virtual Machines. In *USENIX Annual technical conference, general track*. 391–394.
[29] Gabriele Paoloni. 2010. *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*.
[30] Gerald J Popek and Robert P Goldberg. 1974. Formal requirements for virtualizable third generation architectures. *Commun. ACM* 17, 7 (1974), 412–421.
[31] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.
[32] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. 2013. Schedule processes, not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*. ACM, 1.
[33] SPEC. 2017. SPEC virt_sc 2013. Retrieved September 28, 2018 from http://www.spec.org/virt_sc2013/index.html
[34] Michael S. Tsirkin. 2010. vhost-net and virtio-net: Need for Speed. KVM Forum.
[35] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. 2015. A comprehensive implementation and evaluation of direct interrupt delivery. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 1–15.
[36] Werner Vogels. 2008. Beyond server consolidation. *Queue* 6, 1 (2008), 20–26.
[37] Harry Wagstaff, Bruno Bodin, Tom Spink, and Björn Franke. 2017. SimBench: A portable benchmarking methodology for full-system simulators. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 217–226.
[38] xv6 contributors. 2018. xv6. Retrieved February 18, 2019 from https://github.com/mit-pdos/xv6-public
[39] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. 2012. DeTail: reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 139–150.
[40] Binbin Zhang, Xiaolin Wang, Rongfeng Lai, Liang Yang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. 2010. Evaluating and optimizing I/O virtualization in kernel-based virtual machine (KVM). In *IFIP International Conference on Network and Parallel Computing*. Springer, 220–231.