AVMM: Virtualize Network Client with a Bare-metal and Asymmetric Partitioning Approach

Yuezhi Zhou, Yaoxue Zhang, Hao Liu
Dept. of Computer Science & Technology
Tsinghua University, China
Email: zhouyz@mail.tsinghua.edu.cn, zyx@moe.edu.cn

Naixue Xiong
Dept. of Computer Science
Georgia State University, USA
Email: nxiong@cs.gsu.edu

***Abstract*—This paper presents the design, implementation, and evaluation of AVMM, a symmetric partition-based bare-metal client virtualization approach that tries to achieve maximum near-native performance for end-users while supporting new *out-of-OS* mechanism for value-added services for network system administration. To achieve these goals, AVMM divides the underlying network client platform into two asymmetric partitions: user and service partitions. The user partition runs a commodity OS, which is assigned to most portions of the CPU and memory resources and a set of peripheral devices to retain the end-user experience. The service partition runs a specialized OS, which consumes only the essential resources for its tasks. By letting user OS possess the most part of resources and access some peripheral devices directly, the AVMM overhead is reduced greatly, improving the whole network system performance. We have implemented a preliminary network prototype that can support Windows and Linux. Our experimental evaluation results show that AVMM has achieved its designed goals and provides a feasible and efficient approach for client virtualization.**

***Index Terms*—Asymmetric Partitioning, Desktop Virtualization, Network Client Virtualization, Virtual Machine, Virtual Machine Monitor**

I. Introduction

The rapid advance in network desktop/personal computers (PCs) has greatly improved end-user productivity and flexibility by enabling a rich set of applications. PCs have been ubiquitously deployed in enterprise environments, such as universities, corporations, and governmental organizations, as the most common clients to access abundant applications.

However, the success of PCs also creates a set of challenges for enterprise network system administration and management. The high management cost for administrators and end users to maintain and manage the software, and to back up and secure the distributed state and data has resulted in a very high total cost of ownership (TCO). It is shown that in a typical enterprise scenario, the annual cost of managing a traditional PC can be multiple (up to five) times the cost of deploying it [1]. This situation will become more serious with the increased user demands on productivity and flexibility, e.g., accessing their applications and data anywhere, anytime, from any device with similar experiences, and those form factors, such as network ubiquity and mobility.

The power of virtualization technology has been recognized for long to address many challenges in computing systems. The hardware abstraction of virtual machine (VM), especially the decoupling between operating systems (OSes) and the underlying hardware, brings out many advantages that would be hard to achieve, such as server consolidation [2], debugging [3], and security [4]. Due to these distinguishing benefits, these years have witnessed widely deployments of server virtualization products in data-centers, enterprises, and other organizations [5].

Recently, inspired by the success of server virtualization, the client virtualization is also proposed and practised to address the challenges faced by traditional PCs, reducing the cost and improving the enterprise agility [6]. Despite the overwhelming potential advantages of client virtualization, it is more complex and difficult to adopt them widely in enterprise environments than server virtualization for two issues: performance and usability.

In this paper, we first present AVMM, a bare-metal native client (Type I) VMM with an asymmetric resources partitioning strategy to achieve the maximum near-native performance, while facilitating easy supporting of value-added features of management and security. To address the performance challenge, AVMM classifies the underlying platform resources into several types and allocates them to be monopolized or shared by different asymmetric partitions. The monopolized resources, for example, graphics or audio devices, can be directly accessed by user OS. Observing that end-users seldom need to run several commodity OSes (user OSes) at the same time, this monopoly strategy can improve the overall performance, especially the I/O performance sharply. To ease the development and deployment of value-added services, AVMM creates a dedicated service partition running a specialized service OS, in which more complex and *out-of-OS* service modules can be implemented and added to improve the client management and security. Second, we implemented a preliminary prototype that can support Windows and Linux on an Intel x86 machine, which is capable of newly emerged hardware-assisted virtualization technology, i.e., Intel VT [7]. Finally, we evaluated this prototype with several micro- and application benchmarks. The results have demonstrated that AVMM can achieve comparable performance of regular PC and better than the client-hosted client virtualization approaches, and it is an efficient bare-metal client virtualization with maximum near-native performance.

The remainder of the paper is organized as follows. In Section II, we present our main design goals and principles. Section III describes the general approaches of AVMM and presents the detailed design and implementation. In Section IV, we evaluate AVMM and present the experimental results. Finally, We conclude this paper in Section V.

II. Design Goals and Principles

In this section, we present the three design goals of AVMM: *Maximum near-native performance*, *Full compatibility with commodity OSes*, and *Facilitation of out-of-OS value-added services*. To achieve these above challenging goals, we adopt the following design principles: **1). Hardware-assisted virtualization.** To obtain the full compatibility while provide higher performance, AVMM employs the most recent hardware-assisted virtualization technologies. With this hardware virtualization, not only do the privilege instructions need no emulation, but also the whole VMM implementation can be simplified. Thus, the overall performance can be improved potentially. **2). Asymmetric and dedicated service partition.** The underlying client platform is partitioned into two asymmetric partitions (i.e., virtual machines): user partition to run commodity OS and service partition running a special OS that is devoted to service, control, or management functions for the system. **3). Single user OS and partial direct I/O device access.** The platform resources may be monopolized or shared among the partitions. To preserve the unmodified user experience, there is only one user partition that run a commodity OS for end-users at one time. In this way, most of the platform resources, such as most portions of CPU or memories can be assigned to the user partition, while the service partition runs with the minimum necessary resources.

III. General Approaches

In this section, we present the general approaches of AVMM, including the asymmetric partitioning strategy and the overview of AVMM.

A. Asymmetric Partitioning

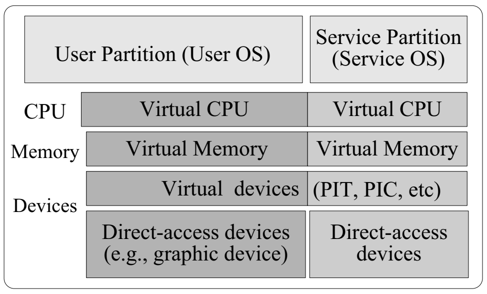


Fig. 1. Asymmetric partitioning

AVMM is based on the recent hardware-assisted virtualization technologies introduced by mainstream vendors, for example, Intel VT [7] or AMD-V [8]. With this hardware virtualization, the instruction execution, access to privileged CPU registers, and I/O port access of a VM can be selectively trapped into the VMM by configuration. Thus, the underlying platform resources, such as certain I/O devices, can be assigned and accessed directly by a VM as required without any VMM intervention.

Fig. 1 shows the asymmetric partitioning structure of AVMM. As shown in this figure, the platform resources of a client is divided into two partitions with the help of AVMM: a user partition which runs a commodity OS, such as Windows XP, and a service partition that runs a specialized service OS that will provide run-time services for user OS or other value-added functions. Considering the factors of performance, reliability, and security, the underlying platform resources of clients will be dedicated to one of the partitions or shared between them as required.

To obtain the maximum performance of the peripheral devices that are important for user experience, those resources, such as graphics or audio devices, can be assigned to the user partition exclusively for direct access. These accesses of the dedicated direct-access devices will pass-through the AVMM without any intervention. Of course, certain resources can also be assigned to service OS for direct access.

However, some critical platform resources that are very important for the system control and management, such as programmable interval timer (PIT), programmable interrupt controller (PIC), advanced programmable interrupt controller (APIC), must be virtualized and shared among partitions, and accessed via the AVMM. Of course, the CPU and memory must be shared among partitions too.

It is noted that the BIOS function should be modified to enable the loading of AVMM and service OS first and then the user OS.

B. Overview of AVMM

In this section, we will present the general architecture of AVMM, as illustrated in Fig. 2. AVMM runs directly on the hardware platform and provides a full-virtualization-alike interface to the user OS, meaning that there isn't any modification of the user OS kernel. As in other VMMs, the main interfaces of AVMM can also be classified into three aspects: CPU, memory, and I/O devices. In the following we address each subsystem and their general design in AVMM. Note that although our implementation is specific to an x86 machine with Intel VT, these aspects can be readily applied to other platform and hardware virtualization with or without minor modifications.

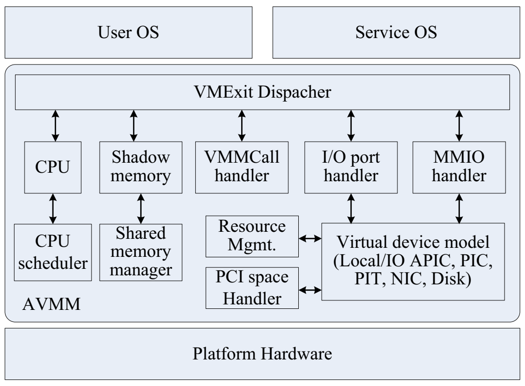


Fig. 2. Overview of the AVMM architecture

1) CPU: In AVMM, the CPU is virtualized through architectural extensions of the modern CPU, such as Intel VT or AMD-V. With the help of Intel VT, it is straightforward and more robust to virtualize the CPU than the software-based virtualization. The user or service OS can be directly executed without any simulation or emulation of privileged instructions.

The virtual CPU module in AVMM provides the abstraction of a processor to partitions, managing the virtual processor and associated virtualization events. It saves and resumes the corresponding physical processor's state when execution switches between partitions as a VM Exit or VM Entry. To get the full control of the platform, AVMM must intercept or trap some special instructions or events critical for the system control or management by specifying the hardware-enable data structure, i.e., the VMCS structure in Intel VT. These instructions or events that may incur an VMExit to be handled by AVMM are configured as follows:

- Instructions whose execution will change the CPU state or status must be trapped and/or virtualized to avoid the affection on the other partition's normal execution. These instructions includes CPUID, HLT, PAUSE, INVD, INVLPG, etc.
- Accesses to privileged processor states, for examples, MOV CRx (CR0, CR3, CR4), MOV DRx, RDMSR, WRMSR, etc., have to be intercepted. Read/write for those involved control or status registers, such as CR0, CR4, and time stamp counter (TSC) may need to be shadowed, that is, the read/write access to such registers will not cause an VM Exit, but will return/write to the shadowed values.
- Exceptions and faults. Certain exceptions and faults, such as page fault, need to be intercepted as VM Exit. Accordingly, the corresponding virtualized exceptions and faults need to be injected into related partitions on VM entry.
- External interrupts are all intercepted on VM Exit and the corresponding virtualized one is injected on VM Entry. Note that the external interrupt vector cannot be configured to be intercepted selectively, thus every interrupt will first be handled by the AVMM and then injected to one partition accordingly.

The physical CPU are shared among AVMM, user or service OSes. To guarantee the performance of user OS, most portions of the CPU time will be assigned to it. This is implemented by the CPU scheduler in AVMM.

2) Memory: Due to the fixed number and functions of partitions in AVMM, the main memory is subdivided into several contiguous areas and allocated to each partition. For each partition, it appears to have a 0-origin memory. The memory accesses may be trapped or checked as required by the AVMM to ensure security or enable virtual memory-mapped I/O (MMIO) functions.

The user partition is allocated to a contiguous physical memory starting from the absolute location zero to the maximum that can be allocated to it. This *virtual=real* mapping [9] enables that the performance overhead associated with address relocation and paging is avoided. In addition, the DMA operation can be executed directly without any memory translation and thus improve I/O performance.

The service partition and AVMM are also allocated to a contiguous physical memory, but starting from a fixed offset. Similarly, this allocation can also reduce the performance overhead of memory paging and DMA operations, only with a fixed memory offset.

3) I/O Devices: The virtualization of other devices is more complicated than that of CPU and memory. The other devices in an AVMM client are shared among partitions or assigned to one of the partitions exclusively as desired. Specifically, from the perspective of the guest OS, its I/O devices can be classified into three categories as follows:

Virtual device. The critical platform resources for system control and management, such as PIT, PIC, and APIC, must be virtualized by AVMM. To avoid guest OS kernel modification, these devices are full virtualized in AVMM and can be accessed by the user or service OS as normal with their existing physical device drivers. The virtual device models and physical drivers of these critical devices are implemented only in AVMM and can be accessed through standard I/O interfaces by guest OSes.

Direct-access device. These devices in AVMM client are assigned exclusively to one partition and are monopolized by it. To improve the end-user's experienced performance, especially the access performance of I/O devices, most devices other than the above mentioned critical devices can be assigned to and directly accessed by user OS. The I/O port and MMIO operations of the direct-access device won't be trapped into AVMM, avoiding the AVMM overhead. It is noted that the DMA device is not virtualized and accessed directly to improve the operation performance. Due to its *virtual=real* memory mapping, the user OS can access the DMA devices as normal. However, the service OS has to take care of the operation, but only needs to add an offset to the allocated original memory address of the DMA operation.

Disguised device. Different from the virtual devices, the virtual device model of a disguised device is split into two parts in the AVMM and service OS respectively. The part of device model in the AVMM just traps the corresponding port I/O operations and passes them to the part of device model in the service OS, which completes the virtual device operations with the existing physical device drivers in the service OS. There are two advantages of disguised devices. First, it can support diverse devices easily by leveraging the existing device drivers in service OS, and thus simplify the construction of AVMM and improve the system reliability. Second, considering the two main sources of system weakness and vulnerability: disk and network, we can disguise them for the user OS to provide enhanced features. For example, the split device driver in service OS can facilitate to develop value-added applications to enhance the system reliability and security, such as soft devices [10].

C. Dynamic proportional-share CPU scheduling

As described before, the platform's PIT is virtualized by AVMM. Thus, the sharing and scheduling of CPU can be implemented by assigning the timer's slice to user or service OS. To acquire the similar performance of user OS, we assign most of the CPU, for example, 90 percent to the user OS, only a small proportion to the service OS, about 10 percent. The service OS can set the parameters to specify the proportions of CPU cycles assigned to the guest OSes by calling into AVMM.

The limitation of this fixed proportional-share scheduling is the lack of flexibility. For example, the idle time of one guest OS cannot be allocated to the other. Also, if the critical tasks to be carried out by service OS, it cannot be executed timely, resulting in a reduced overall performance. To address this problem, we extend it to a dynamic proportional-share mechanism by applying two heuristics to improve the efficiency. These two heuristics are outlined as follows: **1). Identify the idle states and remise the cycles.** If the user OS is idle, the AVMM will let the service OS run. And if the service OS is idle, AVMM will schedule the time to user OS. If both are idle, the AVMM will enter idle. **2). Detect the busy states and seizure the cycles.** The AVMM will detect the busy state or potentially busy state of service OS. If it finds, AVMM will let the service OS to snatch up more CPU cycles. For example, when the user OS issues a disguised hard disk operation which will be trapped into AVMM and completed by the service OS, AVMM will let the service OS run instantly in the next time slice to finish the request timely.

D. Shadow page table mechanism

To retain the ultimate control of the memory resources and protect the access from and between partitions, AVMM has to virtualize the memory access. Due to the fixed and 1:1 virtual-to-physical allocation strategy in AVMM, it doesn't allocate and reclaim the physical memory dynamically and just validates and redirects memory accesses of partitions, simplifying the implementation of shadow memory module. That is, the guest virtual address (i.e., guest physical address) installed in the virtual CR3 and the guest PD and PT entries by the user OS is directly mapped into the real physical address and the service OS is mapped to the physical address only with a fixed offset.

Initially, the shadow PTS is created with all of its entries marked as invalid, using the present (*P*) flag bit in them. However, in an x86 machine, the processor will set the accessed (*A*) bit and dirty (*D*) bit in the PD and PT entries automatically. These various events initiated by guest OS needed to be trapped into the AVMM can be classified into the following groups.

1. **Page fault.** The page fault may be generated by a guest OS normally, for example, if the requested guest page is not in the guest PTS. In this case, the fault should be raised to and handled by the guest OS. As described before, the page fault may also be generated by the inconsistency between the shadow and guest PTS, for example, the shadow PD or PT entries are marked as *not present* or their *R/W* bits are not consistent with the guest one. In this case, AVMM needs to maintain the shadow PTS according to the guest PTS, by allocating new PT pages, and/or update the relevant flag bits in PD or PT entries as described before, and then re-execute the page-faulted instruction.

To intercept the MMIO operations of certain PCI devices or the PCI configuration transactions that needed to be virtualized from the guest OS, the MMIO memory pages are also set with *not present*. Thus, when a guest OS accesses those MMIO pages, it will create a page fault. Of course, these page faults should be passed to and handled by the MMIO handler module in AVMM.

2. **TLB operation.** the TLB operation, for example, INVLPG, issued by the guest OS, must also be trapped and handled by AVMM due to the possible inconsistency incurred between the guest and shadow PTS. After the AVMM takes control, it first modifies the shadow PTS to emulate the desired effect of INVLPG by setting the relevant shadow PD or PT entries as *not present* and then executes INVLPG with the faulting address to remove the invalid physical TLB entries. Noted that if all entries in the shadow PT are not present, this shadow PT can be deallocated and the parent PD entries is set as *not present*. After that, it needs to reloads CR3 with its current value to flush the physical TLB.

3. **Address space switch.** When a guest OS attempts to load from or store to CR3 or initiate a task switch, the address space will be changed, resulting in the invalidation of the entire TLB. Therefore, the AVMM must take control and disable the whole shadow PTS to emulate the desired effects.

E. Resource discovery and allocation

As described before, AVMM can assign an I/O device to one partition for exclusive usage. This means that the assigned device must not be used or shared by other partitions. In AVMM, we use a resource hidden technique to assign and hide resources for guest OSes. The memory is allocated and hidden to guest OS by intercepting the access of advanced configuration and power interface (ACPI) tables [11]. When a guest OS accesses the PCI configuration space of devices by accessing the 0xCF8 and 0xCFC I/O port or the MMIO memory[1], these accesses will be trapped into AVMM. If the device is needed to be assign to it, AVMM will return the real configuration space, otherwise it will return *Null* to hide them from the guest OS. If a device is hidden to one guest OS, its corresponding MMIO resources will also be hidden by the shadow memory module. If a device is virtualized or disguised by AVMM, the space reading operation will return the virtual configuration space of it. AVMM must emulate and virtualize their PCI base address register (BAR) and other registers to be accessed by the guest OS later.

IV. Implementation and Evaluation

We have implemented AVMM on an Intel Dual Core machine. We also carried out several experiments to evaluate the performance of AVMM using established benchmarks. Here we are primarily interested in the following questions: (1) What is the performance of the virtual CPU and memory? And how does it compare against the performance of native and client-hosted virtualization? (2) What is the performance of the direct-access device and how does it compare with other approaches? (3) What is the value-added disguised device performance? And how does it impact on the system performance?

In all our experiments, the AVMM client machine is configured as an Intel Dual Core E6300 1.86 GHz machine, with a 1 GB DDR2 667 MHz RAM, a 1 Gbps Intel network card, and

[1]PCI Express defines the enhanced configuration access mechanism, which allows an OS to access the configuration registers via memory-mapped address space [12].

an Intel G965 Express Chipset Family Graphics Card. Its hard disk is virtualized as an NBD device that is provided with an NBD server. The NBD server is configured as an Intel Xeon Quad Core 1.6 GHz machine, with a 4 GB DDR2 667 MHz RAM, a single Hitachi 160 GB 15000 rpm SATA hard disk, and a 1 Gbps onboard network card. The AVMM machine and the NBD server are connected with a TP-Link TL-SG1048 full 1 Gbps Ethernet switch.

We also compare the AVMM client performance with a regular PC that has the same hardware configuration as the AVMM client but with a local 250 GB Seagate 7200 rpm SATA hard disk, and a client-hosted VM (CHVM), which is virtualized as with 256 MB memory (the optimal size recommended by VMware) and 20 GB hard disk using VMWare Workstation 6.5 hosted by Windows XP professional (SP2) with NTFS 3.1 file system on the same regular PC hardware[2].

The AVMM client, regular PC and the VMware VM all run Windows XP Professional SP2 with NTFS 3.1 or a desktop Linux (kernel version 2.6.21) with ext3 file system to test their performance in Windows or Linux environment respectively. The AVMM service partition runs a customized Linux (kernel version 2.6.21) and an NBD client (version 2.9.9). The NBD server runs Red Hat Enterprise Linux Server 5.1 (kernel version 2.6.18) on ext3 file system and NBD server (version 2.9.9).

A. CPU

We test the CPU performance in Linux, by running lmbench 3.0 [13]. The CPU subset of lmbench consists of 34 arithmetic microbenchmarks, which can be divided into two groups: a simple tests group with arithmetic like add/mul and a complex tests group with arithmetic like div/mod. A shorter complete time means better performance. We find that the simple tests performance of both AVMM and CHVM are very close to native, even without enough precision to differentiate them. However, AVMM shows a similar performance to native in the complex tests, and much better than CHVM, as shown in Fig. 3. CHVM encounters an overhead of 10% to 15% over native in the following 8 tests: integer div (ID), integer mod (IM), int64 div (ID64), int64 mod (IM64), float div (FD), double div (DD), float bogomflops (FB), double bogomflops (DB). Worse in the test of integer mod parallelism (IMP), CHVM needs 3 times as long as native to complete the operation. However, AVMM has an overhead of less than 4% over native in all these tests.

Our CPU performance evaluation shows that, by using a bare-metal VMM approach, AVMM can achieve a closely near-native performance and better than client-hosted VMM approaches. This may be because in a client-hosted virtual machine, user-level applications have to compete against both VMM and host OS for the CPU resources, resulting in more overhead than in AVMM.

[2]Since VMware Workstation is a very mature and common commercial VM solution, we argue that it can represent a general client-hosted VM approach.

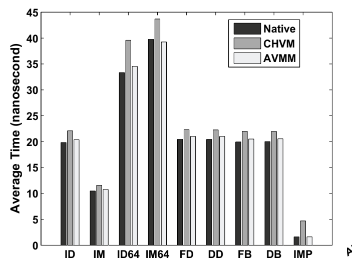


Fig. 3. Lmbench CPU benchmarks.

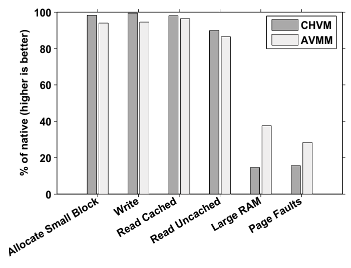


Fig. 4. Memory benchmarks.

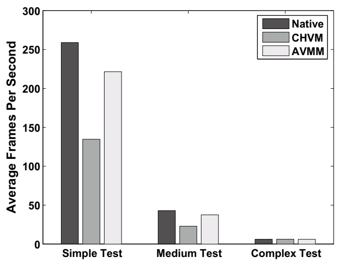


Fig. 5. PerforamceTest 3D graphics.

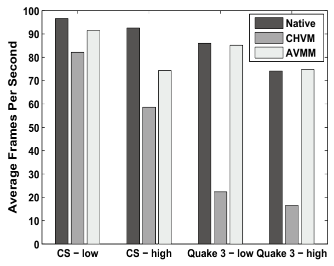


Fig. 6. Computer games benchmarks.

B. Memory

We next evaluate the memory performance of AVMM and also compare it against native and CHVM, by using the PerformanceTest 7.0 Memory Standard Suite. Fig. 4 shows the relative test results of native, CHVM, and AVMM.We observe that in the first four sub-tests, AVMM shows a slowdown of 2% to 4% against CHVM. This overhead of AVMM is due to its primary implementation of the shadow page table algorithm. We also observe that both CHVM and AVMM encounter a sharp performance decrease in Large RAM test. AVMM achieves a performance 157% better than CHVM. This shows that AVMM has a better memory performance than CHVM when encountering too much paging due to its static memory allocation strategy. We also ran lmbench 3.0 in Linux environment to measure the page fault time. This benchmark times how fast a page of file can be faulted in. We ran the benchmark for five trials and the average is: Native (1.33 ms), CHVM (8.53 ms), and AVMM (4.69 ms). We can see that AVMM achieves a performance nearly two times as good as CHVM in page fault test.

C. Graphics

In this section, we study the direct-access device performance in AVMM. As an example, we evaluate the graphics card performance. We first ran PerformanceTest 7.0 Graphics Suite in Windows environment to measure the performance of 3D graphics card, using Microsoft Direct3D 9.0 graphics library. Three test scenarios have been provided: Simple, Medium and Complex. There are two resolutions in our experimental setup: low resolution (800x600) and high resolution (1024x768). We evaluate the graphics by the term of frames per second (FPS) [14], which is a standard evaluation metric for 3D Graphics.

Fig. 5 shows the performance of native, CHVM, and AVMM respectively in low resolution. We can see that CHVM encounters a 50% slowdown over native in both Simple and Medium tests. This overhead mainly comes from a lot of virtualization works involved with graphics card and DMA. Since both graphics card and DMA device are direct-access devices in AVMM and can be accessed directly by the user OS, AVMM do not have any graphics related virtualization overhead except the interrupts injection. Therefore, AVMM only has a performance dropping of 15% over native in Simple test, and 12% in Medium test. We attribute the 12% to 15% dropdown to the CPU and memory overhead and the related interrupt injections. For 3G games test shown in Fig. 6, CHVM encounters a slowdown of 15% to 88% over native in different cases. However, AVMM shows a slowdown of no more than 19% in all these cases. The graphics performance evaluation shows that AVMM can achieve a very close performance to native in the direct-access device access, thus increasing the end-users experiences sharply, especially in the graphics- and video-intensive applications.

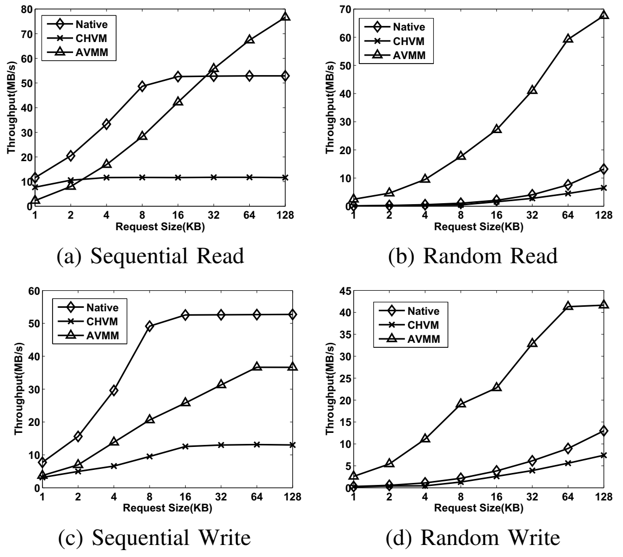
D. Disk

We finally evaluate the performance of the disguised NBD-based hard disk in AVMM environment, by using the Iometer [15] performance tool. Here we also compare its performance against native hard disk and client-hosted VM disk.

For sequential disk access, the native sequential read/write throughput increases with request sizes, and saturates to about 53MB/s when the request size is larger than 16KB (see Fig. 7). However, CHVM has a big slowdown over native, with a throughput of no more than 13MB/s. Surprisingly, AVMM can achieve a throughput two times as high as CHVM in most cases, and even outperforms native when the request size is larger than 32KB in sequential read. The reason may be that the NBD server has a faster hard disk (15000 rpm) than the native (7200 rpm). For random disk access, the read/write throughput of AVMM increases with the request size far more sharply, which is at least 3 times higher than native or CHVM on average. One reason is that the NBD server has a faster hard disk than native. Since seek time dominates disk random access, a faster disk can save a lot of disk access time. Another reason could be that the NBD server has a larger memory (4 GB) than native(1 GB), so that it can have a large cache.

V. Conclusions

We have developed AVMM, a bare-metal client-side VMM for network desktop virtualization. AVMM can achieve maximum native performance by assigning most portions of CPU and memory resources and a set of peripheral devices to be used exclusively by a user OS. In addition, AVMM supports to add new value-added services easily in a dedicated partition. We have implemented a preliminary prototype based on an Intel-VT enabled platform and performed several experiments to evaluate AVMM. We showed that by leveraging the recent hardware-assisted virtualization, the bare-metal AVMM can achieve closely comparable performance to native and better than network client-hosted VM approaches. Future work includes further optimizing performance, implementing and demonstrating the value-added functions, and applying AVMM in real-world scenarios.



(a) Sequential Read
(b) Random Read
(c) Sequential Write
(d) Random Write

Fig. 7. Disk read/write throughput.

References

[1] A. Gillen, F. W. Broussard, R. Perry, and S. Dowling, "Optimizing Infrastructure: The Relationship Between IT Labor Costs and Best Practices for Managing the Windows Desktop, IDC White Paper," http://www.microsoft.com/virtualization/en/us/products-desktop.aspx, 2006.

[2] W. Vogels, "Beyond server consolidation," *Queue*, vol. 6, no. 1, pp. 20–26, 2008.

[3] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the 2005 Annual USENIX Technical Conference*. USENIX, April 2005, pp. 1–15.

[4] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 193–206, 2003.

[5] L. McLaughlin, "Virtualization in the Enterprise Survey: Your Virtualized State in 2008, CIO," http://www.cio.com/article/168401/, 2008.

[6] J. McKendrick, "The 2009 Share Survey: Total Enterprise Virtualization, Unisphere Research," http://www.microsoft.com/virtualization/en/us/products-desktop.aspx, 2009.

[7] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith, "Intel virtualization technology," *IEEE Computer*, vol. 38, no. 5, pp. 48–56, May 2005.

[8] AMD Inc., "AMD Virtualization (AMD-V) Technology ," http://www.amd.com/us/products/technologies/virtualization/Pages/amd-v.aspx, 2009.

[9] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk, "Multiple operating systems on one processor complex," *IBM Syst. J.*, vol. 28, no. 1, pp. 104–123, 1989.

[10] A. Warfield, S. Hand, K. Fraser, and T. Deegan, "Facilitating the development of soft devices," in *Proceedings of the USENIX 2005 Annual Technical Conference*, April 2005, pp. 378–382.

[11] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Tech. Ltd., and Toshiba Corp., "Advanced configuration and power interface specification, revision 3.0b," 2006.

[12] PCI-SIG, "PCI Firmware Specification, Revision 3.0," 2005.

[13] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23.

[14] Motherboards.org., "How to Benchmark a Videocard," http://www.motherboards.org/articles/guides/1278_1.html, 2009.

[15] Iometer., "Iometer project," http://www.iometer.org/, 2009.