A Hash-TLB Approach for MMU Virtualization in Xen/IA64

Xiantao Zhang [1,2], Anthony X. F. Xu [2], Qi Li [2,3], David K. Y. Yau [4], Sihan Qing [5], Huanguo Zhang [1]
[1]*School of Computer, Wuhan University, Wuhan Hubei, 430079, China*
[2]*Intel OTC, SSG, Intel China Software Center, Shanghai, 200241, China*
[3]*Department of Computer Science, Tsinghua University, Beijing, 100084, China*
[4] *Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA*
[5]*Institute of Software, Chinese Academy of Sciences, Beijing, 100080, China*
[2]{xiantao.zhang,anthony.xu}@intel.com [3]liqi@csnet1.cs.tsinghua.edu.cn [4]yau@cs.purdue.edu

Abstract

With advances in hardware-assisted full virtualization technologies, system virtualization based on the virtual machine monitor (VMM) has received much recent attention. Using the Xen/IA64 hardware virtual machine implemented on Intel ® Virtualization Technology for Itanium(VT-i), we investigate the design of a virtual software hash translation lookaside buffer (TLB) based on the virtual hash page table (VHPT). Experimental results show that the proposed design can significantly improve the performance of the hardware virtual machine of Xen/IA64. Our contributions are the following. First, we design and implement in the VMM a virtual hash TLB algorithm to optimize the system performance of VT-i guest virtual machines. Second, we quantify experimentally the performance benefits of the hash TLB for VT-i guest virtual machines and analyze the performance impact of the software VHPT walker with the hash TLB algorithm. Lastly, we present experiments to verify, in an SMP virtual machine system environment, the superior scalability of the hash TLB approach.

1 Introduction

The concept of virtualization was first proposed by IBM in the 1960s as a means to share access to expensive hardware resources, and was widely applied in IBM VM360/VM370 systems [10]. Recently, large gains in computing system efficiency drive demands on the use of virtualization for different applications, such as resource multiplexing, performance isolation between applications, dynamic migration of workloads, and security research. A number of excellent projects have appeared in the area, including VMware [14], Xen [3, 6, 17, 1, 11], and KVM [18]. Among the projects, Xen developed at the University of Cambridge has found widespread use due to its flexible structure and excellent performance. The Xen VMM aims to enable the concurrent execution of multiple high-performance virtual machines through targeted modifications of the guest operating systems. Mutiple guest OS can run on the VMM layer of a single physical platform at the same time. While changes are made to the guest OS kernel, the application binary interface of the OS is not changed. Hence, the guest OS in Xen is fully binary compatible with higher layer applications. Xen's method of virtualization is called *paravirtualization*. Other projects employing paravirtualization include Denali [20, 21] and Disco [5], among others.

Paravirtualization requires kernel changes so that the guest OS is aware of its execution in a virtual environment, and can use the knowledge to achieve high efficiency and scalability. Hence, unmodified applications can run directly in the OS of the VM, and achieve performance close to direct execution on comparable native hardware. In contrast to paravirtualization is a virtualization approach known as *full virtualization*, which does not require changes to the OS of guest VMs.

Traditional CPUs, however, do not consider support for system full virtualization, and hence contain a number of *virtualization holes* [16]. Paravirtualization fills these holes by suitably modifying the guest OS. However, since changes to the OS are often infeasible in practice, certain systems realize full virtualization through a trap-and-emulate model [2]. The model uses dynamic binary instruction rewrite to force the execution of certain privileged instructions by the application to trap to the VMM, which then emulates the execution of the instructions. Clearly, the approach results in significant performance loss and increased system complexity. Current systems that employ the trap-and-emulate approach include VMware ESX Server [19] and Virtual Server [13].

Comparing paravirtualization and full virtualization, we notice that paravirtualization has better scalability, but its implementation will require modifications of the guest OS kernel, thereby limiting its applicability. In contrast, although full virtualization does not require kernel changes, its realization through the trap-and-emulate model in traditional CPUs results in significant performance loss and system complexity. These factors motivate Virtualization Technology, or VT for short, proposed by Intel. Intel VT aims to fill traditional virtualization holes in the CPU design, and provide hardware support for virtualization in the chip. It leads to improved performance and convenient implementation of virtualization platforms.

In order to realize full hardware virtualization on the Itanium platform, Intel proposes the VT-i technical specification [9] for extending the design of the CPU and other components to remove the virtualization holes. Based on VT-i, the Intel OpenSource Technology Center (OTC) implements support for full virtualization of virtual machines on Xen/IA64, so that various OSes including Linux and Windows can run on the Xen VMM without modifications. The architecture of the Xen/IA64 full virtualization support is shown in Figure 1. Notice that, in the rest of the paper, we will refer to a full virtualization virtual machine as a VT-i virtual machine, also know as HVM domain.

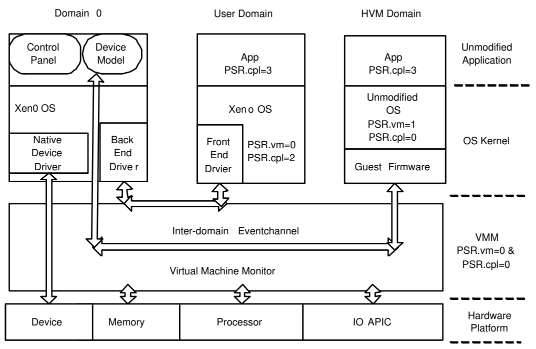

Figure 1. Xen architecture with VT-i domain support.

In this paper, we analyze the performance bottleneck of MMU virtualization on Xen/IA64, when Xen is extended to support full virtualization. Our analysis motivates us to propose a software *hash TLB* optimization algorithm assisted by the virtual hash page table (VHPT) [8], in order to optimize the TLB virtualization performance of VT-i virtual machines. In a Xen/IA64 implementation, because a guest VM under paravirtualization can work cooperatively with the VMM to virtualize MMU components, the VMM can obtain information for identifying the types of most virtual address accesses. The software TLB does not need to track changes in application data accesses. Hence, the software TLB implementation is simple, and can be accomplished through a *single TLB* approach.[1] In the single TLB design, each virtual processor of guest VM has only one virtual TC entry, which is shared by all the processes of an OS running on the VM. The design only implements the smallest number of TLBs required by the IA64 to ensure the VM's forward progress. While the single TLB works well with paravirtualization, it is inadequate for VT-i based full virtualization. In full virtualization, because the guest OS is unmodified, IO accesses are carried out through emulated execution according to the Qemu model.[2] A guest VM is unware of its execution in the virtualization environment. Hence, it depends on the ability of the guest VM TLB to store information about virtual memory accesses in the MMIO space. Without the information, the VMM is unable to distinguish between an IO access and a main memory access, and will have to inject a corresponding page fault into the OS of the guest OS for the decision. As a result, a guest VM will experience a significant performance loss when executing an IO intensive workload.

Therefore, in the full virtualization of the MMU, processing of memory accesses by a *multi*-software TLB algorithm is required for good performance. Driven by the efficiency concern, we propose a VHPT-assisted software *hash TLB* to fulfill the multi-software TLB design goal. Based on the design, by handling guest VM TLB misses through guest VHPT lookup by the software VHPT walker of the guest VM, we reduce by 65% the number of page faults injected into the OS of the guest VM under certain workloads.

2 Background Information

In this secition, in order to better understand our work, we discuss the implementation of TLB virtualization. Then, specifically we talk about single TLB approach in the implementation of paravirtualization.

2.1 TLB virtualization

The TLB is a critical component in the MMU design. TLB mapping is required in every virtual address access for either data or instruction. Hence, the TLB design directly affects the CPU throughput. The TLB in the Itanium architecture is divided into the data TLB (DTLB) and the instruction TLB (ITLB), which are responsible for resolving data and instruction virtual address accesses, respectively. The DTLB and ITLB are, in turn, divided into the translation register (TR) and the translation cache (TC). The TR allows operating systems to pin critical virtual memory translations in the TLB mappings, including accesses to those per-CPU

[1]On the Itanium architecture, the TLB is divided into the TR (Translation Register) and the TC (Translation Cache); we refer to the TC in this context.

[2]Qemu is an emulation model for a full computer system, including its processor and peripherals. Xen uses the peripheral system in Qemu to provide device emulation in full virtualization.

data, kernel memory areas, page tables, and instructions that cannot be allowed to cause a TLB miss. The TR is not affected by the CPU TLB hardware replacement strategy, unless the software uses explicit TR management instructions to purge it, while the TC replacement algorithm is implemented in hardware. Notice that for the Itanium, apart from the TC replacement in hardware and the VHPT TLB refill function, both the insertion and deletion of TLB entries are controlled by software. Such a design is in contrast to the x86 platform.

To virtualize the TLB, the VMM can implement the TLB management functions in software, through catching the exceptions raised by the (modified) guest OS as it attempts to manage the TLB. (Notice that exceptions are raised because the guest OS in fact does not have sufficient privilege to perform TLB operations.) The TLB virtualization must satisfy the Itanium minimum requirement that there are at least eight TRs and one TC for both the instruction and data components. In addition, because the OS specification for using the TR is fixed, the VMM virtualization must use the same number of software TRs implemented by the array structure to store the corresponding information. On the other hand, because on the physical machine, the TC replacement is controlled by hardware which may implement various strategies, the TC virtualization can likewise use a flexible implementation strategy controlled by various software algorithms. The strategy is chosen so that the TC in the guest VM can maximize the efficiency of address translations in the VM.

2.2 The single TLB approach in paravirtualization

Paravirtualization in the Xen/IA-64 has used a single TLB approach based on the VHPT without collision chain. The VMM under such a design implements the smallest number of virtual TLBs required by the Itanium specification. For paravirtualization, the single TLB approach has the following characteristics:

- **Simple implementation logic:** Because the virtual ITLB and DTLB have only one TC entry, the insertion and replacement algorithms are relatively simple.
- **Performance benefits of the TLB flush-all virtualization:** In some cases, the OS may use the ptc.e instruction to purge all the TLB entries. In the virtualization environment, the VMM similarly needs to purge all the TC entries from the virtual TLB. If we can ensure that there is only one entry present in the virtual TLB, the execution time will be reduced and the VM will get better performance.

Although the single TLB approach works well with paravirtualization, it may not be suitable for full virtualization such as using Xen/VT-i, because it may cause many page faults to be seen by the guest OS. We used the single TLB implementation for the Xen/IA-64, and ported it to the Xen/VT-i. In the implementation procedure, we discovered the following shortcomings of the single TLB when it is implemented in VT-i. First, because an unmodified OS kernel is used in the guest VM, the guest VM is unable to cooperate with the VMM to reduce TLB misses. The VMM is required to cache many guest TC entries to cache more translations for the guest VM. Second, since all devices of VT-i VMs are emulated in software, it is necessary for the VMM to identify the MMIO and IO accesses among the virtual address accesses. The single TLB with one TC does not have enough capacity to store all such information. Hence, IO operations are often not identifiable by the VMM, and have to be forwarded to the guest OS for further investigation. Each forwarded entry may cause the replacement of an old entry. In the case of an IO-intensive application, the replacements will in turn lead to excessive changes in the virtual TLB entries, which produce more and more alternate TLB misses[3] for the guest OS and cause poor performance for the VM (since the Linux OS, for example, maps the MMIO/IO address space to region 6 and the VHPT is always disabled in region 6).

3 Hash TLB Virtualization based on VHPT

In Xen/IA-64, the VMM implements a TLB virtualization approach based on the VHPT. In the approach, the guest VM TLB is stored, after suitable P2M translations[4], in the long format VHPT, which can be looked up by the hardware VHPT walker. The approach fully leverages the characteristics of the Itanium MMU to maximize the performance of MMU virtualization in the guest VM. Compared with the hardware TLB, the VMM has the option of choosing the best storage format for each guest VM, including the bookkeeping data structures, the lookup algorithm, and the number of TLBs stored for the VM. For the long format VHPT, the VMM can also decide in software (1) whether the hash table has a collision chain and, if so, the length of the collision chain; and (2) the size of the VHPT.

Besides, the VMM can control the existence format of the VHPT: (1) One single system VHPT in the whole VMM; (2) one VHPT per logical processor (LP); (3) one VHPT per virtual CPU (VCPU); and (4) one VHPT per virtual machine. In this paper, we will not compare the performance of the four approaches, but provide the classification to distinguish between different implementation

[3]A kind of translation misses in the TLB and the VHPT is meanwhile disabled.

[4]P2M translation is a Xen term, which refers to the translation of an address from the guest physical address space to the machine physical address space.

methods for the paravirtualization guest VM and the VT-i guest VM. We used the first approach for the PV domain in an early implementation. It was changed to the second approach with the support of the SMP guest VM implementation. Finally, we provided the implementations of both approaches 2 and 3, and allow the user to select the implementation to run at system startup time. For the VT-i guest VM, we fully evaluated the scalability of the SMP guest VM in the early implementation phase, and adopted approach 3, namely to provide one VHPT per CPU. The approach avoids the overheads of locking when the VHPT is shared by multiple virtual machines.

In the TLB virtualization, the paravirtualization adopts the 8 TR + 1 TC model. For both the ITLB and DTLB, the model represents the smallest number of TLBs to ensure the system's forward progress according to the IA64 specification. Such a model is the *single TLB* approach mentioned in the previous sections. For the VT-i virtualization implementation, a 8 TR + x TC, $x > 1$, hash TLB model is called for by the earlier analysis. That is, for both ITLB and DTLB, we implement the number of TRs required by the IA64 specification and, in addition, use multiple TCs to allow the storage of more TLB entries for each VCPU of VM. In this approach, the VMM uses a hash table, similar to the VHPT, to manage the TC entries of a guest VM TLB. We call this the *hash TLB* approach. The design of the hash TLB has the following characteristics:

- **Unified management of the ITLB and DTLB:** On the Itanium, the TLB of a processor is divided into ITLB and DTLB. Although the split TLB has certain performance advantages, the split virtual ITLB and DTLB design will also increase the system complexity significantly under virtualization. In addition, Since the mainline OSes, such as Windows, Linux, use TLB in the unified way, with our hash TLB approach, we can unify the ITLB and DTLB under the same TLB model. The management workflow is similar to the VHPT. By not distinguishing between the ITLB and DTLB, the system design is simplified.

- **Storage of more guest VM TLB entries at high efficiency:** By storing more TLB entries for the guest VMs, the VMM can handle TLB misses in guest VMs by looking up the required information in the hash TLB. This reduces the number of TLB misses and page faults injected by the VMM into a guest VM. According to our experimental results (Table 1), under the same workload (system bootup and a three-minute kernel build), the hash TLB can reduce by 95.2% the number of page faults injected into the guest OS for processing. In addition, the results show that a large number of ALT DTLB misses seen by the single TLB approach can be reduced to a minimal level by using the hash TLB. The large number of ALT DTLB misses in the case of the single TLB is due to the following reason. When the guest VM is processing IO, the single TLB is unable to store the multiple TLB entries that contain information about MMIO accesses by the guest VM. Hence, the VMM cannot identify whether a guest VM memory access is in the IO space or not. It is then required to inject an ALT DTLB miss into the guest VM for resolving the memory access.

- **Flexible management:** The hash TLB is able to support the huge TLB mode, and furthermore, flexible management strategies for the huge TLB. For example, all the 256M TLB entries in the guest VM can first be inserted into the hash TLB. Then, according to the VMM implementation strategy, the 256M entries can be divided into smaller 16K pages and are either (1) inserted into the VHPT or TLB in a single batch, or (2) inserted into the VHPT or TLB one page at a time. Flexible implementation is therefore possible based on the results of functional analysis.

- **Intercept of information accesses in MMIO space:** MMIO/IO accesses in VT-i are realized through a software emulation model. The VMM must therefore be able to identify all the IO accesses in the guest VM, and direct the accesses to the emulation algorithms accordingly. If mappings to the IO space are stored by the VMM, the VMM will be able to decide whether a memory access should be handled as an IO operation or not. Because the hash TLB is able to store multiple TLB entries, TLB translations in the IO space of the guest VM are very likely to be found in the hash table. Hence, the VMM can easily identify an IO access, greatly reducing the number of page faults injected into the guest OS.

- **Effective hash table collision management:** To effectively manage possible hash collisions, we introduce a collision chain mechanism in the hash TLB implementation. The collision chain reduces the chance that a TLB entry is completely replaced as a result of collision.

- **Excellent scalability:** In the hash TLB design, we ensure that a hash TLB and VHPT are allocated for each virtual CPU. This ensures the scalability of the guest VM system for supporting guest SMP support.

3.1 Two hash TLB implementation strategies

There are two methods to implement the hash TLB based on the VHPT. We call the first one a *full track* method. Essentially, when the guest VM inserts a TLB entry, the

TLB Virtualization Approach	Page Fault Type			
	TLB Miss	Alt TLB Miss	VHPT Miss	Page Not Present
Single TLB	29.069	1147.6097	11.5133	39.3397
Hash TLB	2.6183	0.017	12.1177	44.7206

Table 1. Number of page faults (x10^4) injected into the guest OS for the two virtualization approaches.

VMM, while making insertions into the VHPT and the physical TLB, makes also an insertion into the guest VM hash TLB for tracking. Hence, all the mappings in the VHPT and the physical TLB can be found also as corresponding entries in the guest VM hash TLB. We call the second method a *partial track* method. Essentially, a fraction of the TLB entries in the guest VM are inserted into the hash TLB, while the remaining fraction are inserted into the system VHPT and physical TLB. The two methods require different algorithms for TLB entry insertion/removal, and for deallocating a guest VM TLB.

3.1.1 Full track method

An evaluation of the full track method is as follows. On the one hand, it is extremely flexible. It can track the TLB entries for all the current activities of the guest VM. Global purge of the TLB under SMP guest VM support is also more simple. On the other hand, the memory overhead can be high in order to ensure that all the entries in the guest VM TLB are able to fully cover the VHPT and the TLB. The overhead is in spite of the fact that managing a guest VM TLB miss requires only the guest virtual TLB hash table to be searched for an available translation. In addition, in recycling the collision chain of VTLB entries, we must repeatedly reset the region register [8] to clear the corresponding entries in the VHPT.

3.1.2 Partial track method

The partial track algorithm makes use of the hash TLB, VHPT, and machine TLB resources. It stores one part of the guest TLB in the hash TLB, and the other part in the VHPT and machine TLB. The main disadvantage of the partial track method is the need to look up target mappings in potentially three different places, namely the hash TLB, VHPT, and machine TLB. Our current system implements partial track. In future work, we will compare the performance of the full track method with that of the partial track method in aspects of flexibility, efficiency, and scalability.

3.2 Enhancing the hash TLB algorithm by using the software VHPT walker

The virtual hash page table (VHPT) in the Itanium architecture is an extension of the limited hardware TLB. The VHPT can be looked up in hardware by a *VHPT walker*. Mappings in the VHPT are automatically inserted after they are resolved, similar to the x86 MMU for refilling TLB entries. When the processor fails to find a requested TLB mapping and the VHPT walker is enabled, the CPU can search the VHPT in memory and, if an entry is found, insert the entry into the TLB, to avoid future TLB misses. In fact, the VHPT in Itanium has two hardware support formats: the short format and the long format. The long format is a superset of the short format. It supports the protection key, region id, and a collision chain structure implemented in software. The hardware VHPT walker can be completely turned off in principle, but doing so will greatly reduce the system performance. By the results in Gray *et al.* [12] for a Linux system, the average TLB refill overhead is only about 45 CPU cycles when the the VHPT walker is enabled, and is about 160 CPU cycles when it is disabled. Hence, the VHPT walker is typically enabled on Linux and Windows systems.

As noted above, the OS with VHPT enabled can greatly reduce the CPU time of reloading the TLB. In a virtualization environment, we can implement a software VHPT walker for each VCPU, and use it to look up the requested mappings in the guest VHPT table, when the guest VM experiences a TLB miss. The software VHPT walker implements all the functions of the hardware VHPT walker, and performs in a functionally identical manner. To simplify the system design in our initial implementation, we did not inspect the guest VHPT on a TLB miss, but rather injected a TLB miss directly into the guest VM for further processing. In this case, unnecessary page faults will be injected, which increases the average time in resolving a TLB miss. Our experimental results in Table 2 verify that on TLB misses, using the guest VM VHPT can greatly reduce the number of page faults, by about 64.42% in a kernel build. The time for the kernel build is also reduced by about 5.4%.

Guest VHPT	Page Fault Type			
	ITLB Miss	DTLB Miss	VHPT Miss	Page Not Present
OFF	3920721	7540324	432567	16532
ON	27340	62342	328746	3820342

Table 2. Total times of page faults injected into the guest OS in a kernel build.

4 Hash TLB Performance Evaluation

To quantify the performance of the VHPT-based hash TLB algorithm relative to the single TLB, we have implemented, for each of VT-i virtualization and PV virtualization, the two virtualization approaches. We abbreviate the single TLB for PV implementation as STPV, the single TLB for VT-i implementation as STVT, the hash TLB for PV implementation as HTPV, and the hash TLB for VT-i implementation as HTVT.

4.1 Experimental platform

We use a large number of micro-benchmark tools, including Kernel Build, Byte, CPU2000, System Bench, and Specjbb2005, for evaluating each of the above four implementations. Our evaluation environment is as follows. We use the Intel IA64 Tiger 4 platform with four 1.6 GHz Itanium 2 Montecito processors (each of which has 4 logical processors) and 32 GBytes of memory. The host machine and the guest VMs all run the Redhat Enterprise Linux 4 Update 4 OS and the Xen version ChangeSet 12014. A guest VM is configured to have 10 GBytes of physical memory and four virtual processors. In evaluating the scalability of the hash TLB design, each virtual machine and the native system is equipped with 6 GBytes of memory and correspondingly the same number of virtual processors for equal comparison. Other relevant system parameters of the benchmark tools will be given for the detailed subsystems when needed.

4.2 STVT vs HTVT

We evaluate the performance of STVT and HTVT, and compare it with that of the native OS running on an identically equipped physical machine. The results are shown in Figure 2. They show that the hash TLB algorithm, consistent with its design objectives, can significantly improve the efficiency of both IO intensive and memory intensive applications.

Because Kernel Build requires a large amount of hard disk reads/writes, its performance is increased by about 44.34% through the hash TLB approach. The Byte benchmark is designed to test the CPU and FPU performance under a large number of memory accesses; HTVT improves its performance by 32.94%. The FileIO application in Sysbench tests the IO performance of the system by requiring a lot of emulated IO operations; it similarly achieves a significant performance improvement (by about 31.92%) as a result of the HTVT algorithm. The OLTP application in Sysbench is also an IO intensive case, and the HTVT algorithm improves its performance by 37.25%. SPECjbb2005 needs to make large footprint memory accesses in the JVM because of its emulated database operations; HTVT improves its performance by up to 50.91%. In summary, the results show that the hash TLB algorithm can greatly improve the performance of IO intensive and large footprint memory intensive applications.

4.3 STPV vs HTPV

We have implemented the hash TLB design for PV virtualization. We measure the overall performance of the PV VMs under both the STPV and HTPV approaches. The results are shown in Figure 3. They show that the hash TLB does not improve the performance of the virtual machines under PV virtualization. The analysis is as follows. It is not hard to see that under the specialized IO mode of the PV guest VM, the number of TLB misses as a result of IO operations is reduced. Hence, it is not necessary for the guest VM TLB to track these accesses. Moreover, because the guest OS is modified, instructions for manipulating the TLB can, after paravirtualization, use relatively simple logic to insert information into the system TLB and VHPT. As a result, the complex logic of the hash TLB virtualization can negatively impact the performance of the paravirtualization approach. From the results, notice that for the CPU2000 Integer and Byte benchmarks, HTPV enhances performance by 0.42% and 3.66%, respectively. For the Kernel Build and CPU2000 floating point benchmarks, however, the performance is reduced in the HTPV implementation. Hence, it is not clear whether it is beneficial to integrate the HTPV algorithm into the mainline Xen distribution. We notice that, although the HTPV algorithm can cause small performance degradations for the PV domain, it is helpful from the point of view of code maintenance, because it uses the same algorithm for both the VT-i and PV virtualizations.

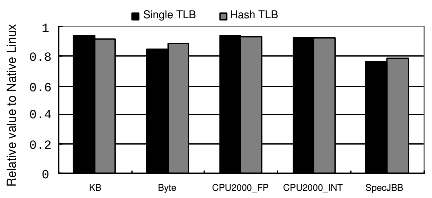

Figure 3. Performance comparison for the PV domain.

4.4 Performance scalability of HTVT for SMP guest machines

Because of the interdependence between core resources in SMP systems, support for SMP virtual machines and the resulting system scalability is a design/implementation challenge for virtualization systems. We now answer the question of how well VHPT-based HTVT can achieve the scalability objective. We present experiments to evaluate the performance of the VT-i guest machine when it is allocated different numbers of virtual CPUs. We compare the results relative to the scalability of the unmodified native OS on a physical system platform.

From Figure 4, notice that HTVT achieves a level of scalability comparable to that of the native OS. The results give convincing evidence that HTVT based on the VHPT is flexible and can adapt well to the level of the processor resources. It can be seen from Figure 4(a) that, for the SPECjbb2005 benchmark, the scalability of the guest machine and that of the native system is quite close. When there are seven CPUs, VT-i in fact achieves slightly better

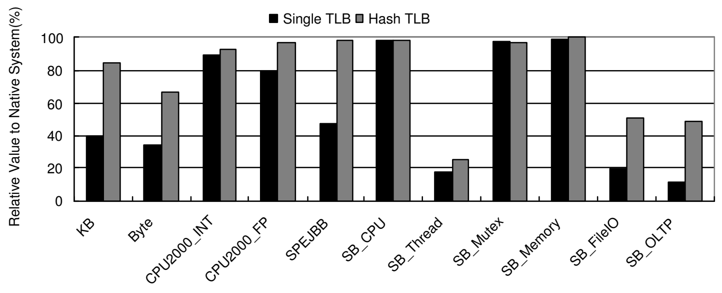

Figure 2. Performance comparison between Hash TLB and Single TLB

performance than the native system. Similar results are obtained for the Kernel Build experiments which are shown in Figure 4(b).

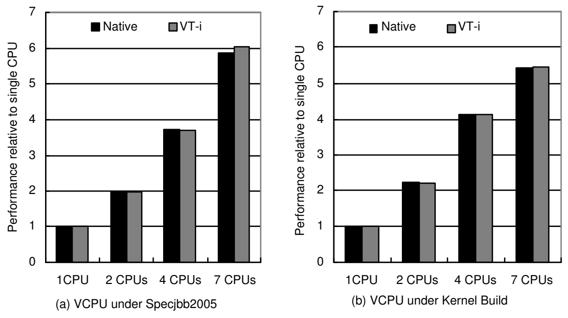

Figure 4. VCPU Scalability of VT-i guest OS with HTVT in a guest SMP environment.

5 Related Work

Xen [3, 6, 17, 1, 11] is a notable VMM project developed in the Computer Lab of Cambridge University. The VMM model and PV scheme used in Xen is shown to obtain competitive performance with the native platform. Daniel *et al.* report the implementation of vBlade [15] as a port of Xen to IA-64, employing the PV approach. In vBlade and its successor Xen/IA64, the single TLB approach is developed to trace TLB translations in the guest VMs, and the VCPU in every guest VM has one TC entry, and behaves well for Xen/IA64. However, single TLB is not a good solution in the case of full virtualization. This is because if a guest OS is not modified, all IO devices in the guest OS will be emulated by the Qemu device model [4, 11], and the guest OS cannot detect if it is running in the virtual environment. In this case, it has to depend on the VMM to cache previous MMIO and IO accesses. Otherwise, many page faults will happen and have to be injected into the guest OS, greatly degrading the performance of IO-intensive applications. In the single TLB approach, IO spaces assigned to the guest VMs are not uniform, and it is hard to ensure that the whole IO space is covered by one virtual TC entry. Most IO accesses fail to be detected when a TLB miss happens for the first time, and many page faults will be injected into the guest OS. Because of the special memory management unit (MMU) virtualization model in VT-i, a new TLB virtualization technique is required to overcome the bottleneck in the single TLB approach. Instead of a single TLB solution, we use a hash TLB approach to improve the efficiency of managing a guest TLB. The hash TLB is highly scalable in the context of full hardware virtualization; its overhead is less than half that of the single TLB.

In the Xen/x86 architecture, the VMM uses the shadow page table [6, 17, 11] to implement MMU virtualization for the guest VM. Because in the x86, the OS and other software has no control over the TLB operations, we can view the shadow page table as the guest VM virtual TLB. The efficiency of the shadow page table design directly impacts the overall performance of the system. In addition, to further enhance the capabilities of the MMU virtualization, the extended page table (EPT) [16] has been proposed for the IA32 processor in the future VT by Intel. The basic design rationale of the EPT is to enable the IA32 CPU to directly use the guest VM page table to map from the guest VM linear address space into the guest VM physical address space. As a result, before the guest VM physical address is released to the system convergence layer, it is passed into the EPT to map from the guest VM physical address into the host machine physical address. The design greatly reduces the number of VM exits [5], which are the key factors of performance issues. The AMD virtualization specification [7] proposes the NPT, which implements functions similar to the EPT. It is definitely possible to implement a component like the x86 EPT in the Itanium architecture to improve the efficiency of the MMU virtualization. Such an implementation will, however, increase the system complexity. For example, one will have to adding the Region ID hardware virtualization support, in order to avoids address space conflicts between the guest VMs as well as side effects in the CPU internal pipeline design.

[5]VM exit is an hardware exception which stands for switching context from guest to VMM

6 Conclusion

In this paper, we design and implement the hash TLB approach for Xen/IA64 TLB virtualization to address the performance issues caused by only one single translation entry in the single TLB approach in Xen with VT-i. Through the performance comparison between the single TLB and hash TLB approaches in paravirtualization and VT-i full virtualization, we show that the hash TLB can remove the performance bottleneck faced by the single TLB in Xen/VT-i. The performance study shows that the hash TLB can achieve a factor of two overall performance for VT-i guest VMs. The VM performance in the VT-i and PV domains is more than 80% of the native Linux performance in most cases, and is close to 100% in some cases. Also, our approach resolves the performance problem caused by too many page faults being injected into the guest OS. These page faults are generated by IO accesses in the guest VM. The proposed hash TLB virtualization has much better adaptability, flexibility, and scalability than the single TLB. In addition, we analyze the performance of the hash TLB approach in an SMP environment. In this environment, the hash TLB has similar (or sometimes better) scalability compared with the native platform.

Acknowledgments

We are grateful to Fleming Feng, Susie Li, Fred Yang from Intel for their valuable suggestions and review comments. Especially, we thank Eddie Dong for his helpful discussion. In addition, other members from Intel OTC VMM team also gives us many helps.

The research of D. Yau was supported in part by the U.S. National Science Foundation under grant number CNS-0305496; the research of S. Qing was supported in part by the National Science Foundation of China under grant number No.60573042; the research of H. Zhang was supported in part by the National Science Foundation of China under grant number No.60673071 and Hi-Tech. Research and Development Program of China under grant number 2006AA01Z442 and 2007AA01Z411.

References

[1] HPC virtualization with xen on itanium. *Master thesis, Norwegian University of Science and Technology*, 2005.

[2] K. Adams and O.Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of international conference on Architectural support for programming languages and operating systems*, 2006.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2003.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference*, 2005.

[5] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[6] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Matthews. Xen and the art of repeated research. In *Proceedings of USENIX Annual Technical Conference*, 2004.

[7] AMD Corp. *AMD64 Virtualization Codenamed ”Pacifica” Technology: Secure Virtual Machine Architecture Reference Manual*, 2005.

[8] Intel Corporation. *Intel Itanium Architecture Software Developer’s Manual*. http://www.intel.com/design/itanium/manuals.htm.

[9] Intel Corporation. *Intel Virtualization Specification for the Intel Itanium Architecture (VT-i)*.

[10] R. Creasy. The origin of the vm/370 time-sharing system. *IBM Systems Research Journal*, 1981.

[11] Y. Dong, S. Li, A. Mallick, J. Nakajima, K. Tian, X. Xu, F. Yang, and W. Yu. Extending xen with intel virtualization technology. *Intel Virtualization Technology*, 10(3), 2006.

[12] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser. Itanium - a system implementor’s tale. In *Proceedings of USENIX Annual Technical Conference*, 2005.

[13] Connectix Inc. *Product Overview: Connectix Virtual Server*. http://www.connectix.com/products/vs.html.

[14] VMware Inc. *VMware virtual machine technology*. http://www.vmware.com/.

[15] D. Magenheimer and T. Christian. vblades: Optimized paravirtualization for the itanium processor family. In *Proceedings of Virtual Machine Research and Technology Symposium*, 2004.

[16] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Virtualization Technology*, 10(3), 2006.

[17] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the art of virtualization. In *Proceedings of Ottawa Linux Symposium*, 2005.

[18] Inc Qumranet. Kernel-based virtual machine driver for linux. http://www.qumranet.com/wp/kvm_wp.pdf.

[19] C. Waldspurger. Virtual machines: Memory resource management in vmware esx server. In *Proceedings of Symposium on Operating Systems Design and Implementation*, 2002.

[20] A. Whitaker, M. Shaw, and S. Gribble. Denali: A scalable isolation kernel. In *Proceedings of ACM SIGOPS European Workshop*, 2002.

[21] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of USENIX Annual Technical Conference*, 2002.