AMD-V™ Nested Paging

Issue Date: July, 2008 Revision: 1.0

Advanced Micro Devices, Inc.

Executive Summary

Operating systems use processor paging to isolate the address space of its processes and to efficiently utilize physical memory. Paging is the process of converting a process-specific linear address to a system physical address. When a processor is in paged mode, which is the case for modern operating systems, paging is involved in every data and instruction access. x86 processors utilize various hardware facilities to reduce overheads associated with paging. However, under virtualization, where the guest's view of physical memory is different from system's view of physical memory, a second level of address translation is required to convert guest physical addresses to machine addresses. To enable this additional level of translation, the hypervisor must virtualize processor paging. Current software-based paging virtualization techniques such as shadow-paging incur significant overheads, which result in reduced virtualized performance, increased CPU utilization and increased memory consumption.

Continuing the leadership in virtualization architecture and performance, AMD64 Quad-Core processors are the first x86 processors to introduce hardware support for a second or nested level of address translation. This feature is a component of AMD Virtualization technology (AMD-V™) and referred to as Rapid Virtualization Indexing (RVI) or *nested paging*. Under *nested paging*, the processor utilizes *nested page tables*, which are set up by the hypervisor, to perform a second level address translation. Nested Paging reduces the overheads found in equivalent shadow paging implementations.

This whitepaper discusses the existing software-based paging virtualization solutions and their associated performance overheads. It then introduces AMD-V™ Rapid Virtualization Indexing technology – referred to as *nested paging* within this publication - highlights its advantages and demonstrates the performance uplift that may be seen with nested paging.

Table of Contents:

1. Introduction

System virtualization is the abstraction and pooling of resources on a platform. This abstraction decouples software and hardware and enables multiple operating system images to run concurrently on a single physical platform without interfering with each other.

Virtualization can increase utilization of computing resources by consolidating workloads running on many physical machines into virtual machines running on a single physical machine. This consolidation can dramatically reduce power consumption and floor space requirements in the data center. Virtual machines can be provisioned on-demand, replicated and migrated using a centralized management interface.

Beginning with 64-bit AMD Opteron™ Rev-F processors, AMD has provided processor extensions to facilitate development of more efficient, secure and robust software for system virtualization. These extensions, collectively called AMD Virtualization™ or AMD-V™ technology, remove the overheads associated with software-only virtualization solutions and attempt to reduce the performance gap between virtualized and non-virtualized systems.

2. System Virtualization Basics

To allow multiple operating systems to run on the same physical platform, a platform layer implemented in software decouples the operating system from the underlying hardware. This layer is called the *hypervisor*. In context of system virtualization, the operating system being virtualized is referred to as *guest*.

To properly virtualize and isolate a guest, the hypervisor must control or mediate all privileged operations performed by the guest. The hypervisor can accomplish this using various techniques. The first technique is called *para-virtualization*, where the guest source code is modified to cooperate with the hypervisor when performing privileged operations. The second technique is called *binary translation,* where at run time the hypervisor transparently replaces privileged operations in the guest with operations that allow the hypervisor to control and emulate those operations. The third method is *hardware-assisted virtualization,* where the hypervisor uses processor extensions such AMD-V to intercept and emulate privileged operations in the guest. In certain cases AMD-V technology allows the hypervisor to specify how

the processor should handle privileged operations in guest itself without transferring control to the hypervisor.

A hypervisor using binary translation or hardware assisted virtualization must provide the illusion to the guest that the guest is running on physical hardware. For example, when the guest uses processor's paging support for address translation, the hypervisor must ensure that the guest observes the equivalent behavior it would observe on non-virtualized hardware.

3. x86 Address Translation Basics

A *virtual address* is the address a program uses to access data and instructions. Virtual address is comprised of *segment* and *offset* fields. The segment information is used to determine protection information and starting address of segment. Segment translation cannot be disabled, but operating systems generally use *flat segmentation* where all segments are mapped to the entire physical address space. Under flat segmentation the virtual address effectively becomes the *linear address*. In this whitepaper we will use linear and virtual address interchangeably.

If paging is enabled, the linear address is translated to a *physical address* using processor paging hardware. To use paging, the operating system creates and manages a set of page tables (See figure 1). The page table walker or simply the *page walker,* implemented in processor hardware, performs address translation using these page tables and various bit fields in the linear address. Figure 2 shows the high level algorithm used by the page walker for address translation.

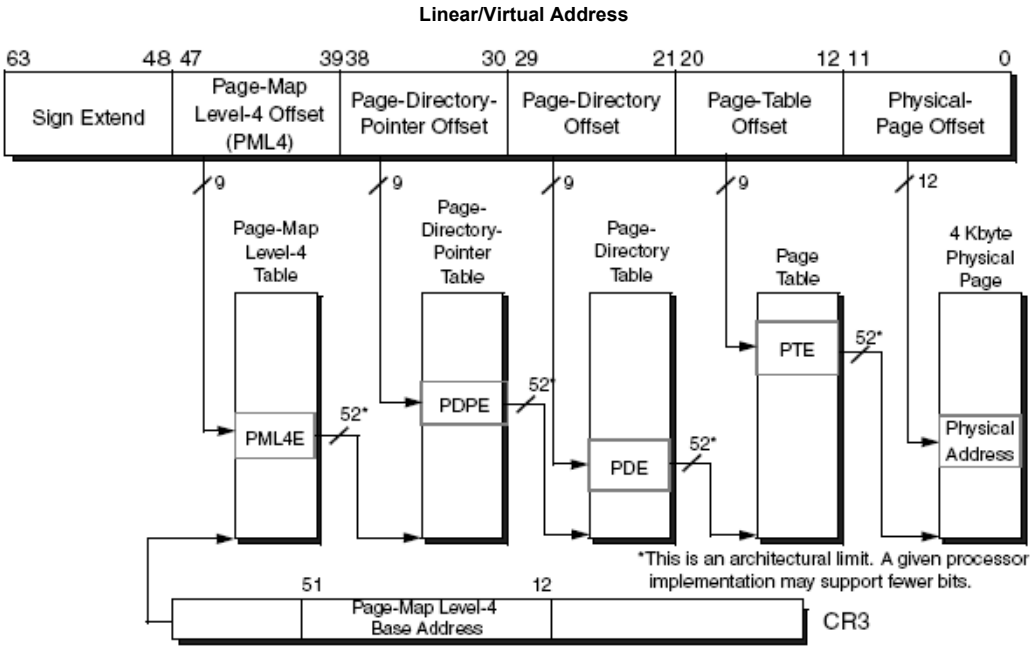

Figure 1: 4KB page tables in long mode

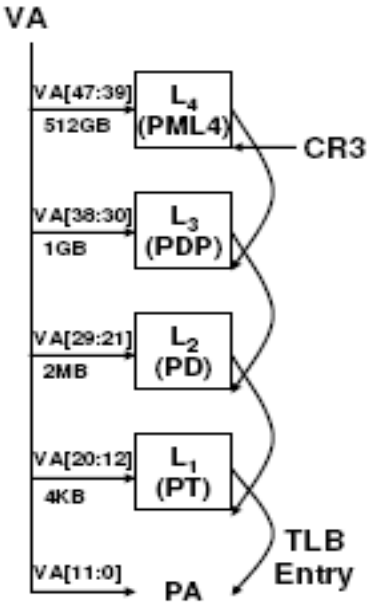

Figure 2: Linear/Virtual to physical address translation algorithm

During the page walk, the page walker encounters physical addresses in CR3 register and in page table entries which point to the next level of the walk. The page walk ends when the data or leaf page is reached.

Address translation is a very memory intensive operation because the page walker must access the memory hierarchy many times. To reduce this overhead, AMD processors automatically store recent translations in an internal *translation look-aside buffer (TLB).* At every memory reference, the processor first checks the TLB to determine if the required translation is already cached; if it is cached, the processor uses that translation; otherwise page tables are walked, the resulting translation is saved in the TLB and the instruction is executed.

Operating systems are required to cooperate with the processor to keep the TLB consistent with page tables in memory. For example, when removing an address translation, the operating system must request the processor to invalidate the TLB entry associated with that translation. On SMP systems where an operating system may share page tables between processes running on multiple processors, the operating system must ensure that TLB entries on all processors remain consistent. When removing a shared translation, operating system software should invalidate the corresponding TLB entry on all the processors.

The operating system updates the page table's base pointer (CR3) during a context switch. A CR3 change establishes a new set of translations and therefore the processor automatically invalidates TLB entries associated with the previous context.

The processor sets *accessed* and *dirty* bits in the page tables during memory accesses. The 'accessed' bit is set in all levels of the page table when the processor uses that translation to read or write memory. The 'dirty' bit is set in the PTE when the processor writes to the memory page mapped by that PTE.

4. Virtualizing x86 paging

To provide protection and isolation between guests and hypervisor, the hypervisor must control address translation on the processor by essentially enforcing another level of address translation when guests

are active. This additional level of translation maps the guest's view of physical memory to the system's view of physical memory.

With para-virtualized guests, the hypervisor and the guest can utilize para-virtual interfaces to reduce hypervisor complexity and overhead in virtualizing x86 paging. However for unmodified guests, the hypervisor must completely virtualize x86 address translation. This could incur significant overheads which we discuss in following sections.

4.1 Software techniques for virtualizing address translation

Software-based techniques maintain a *shadow* version of page table derived from *guest page table* (gPT). When the guest is active, the hypervisor forces the processor to use the *shadow page table* (sPT) to perform address translation. The sPT is not visible to the guest.

To maintain a valid sPT the hypervisor must keep track of the state of gPT. This include modifications by the guest to add or remove translation in the gPT, guest versus hypervisor induced page faults (defined below), accessed and dirty bits in sPT; and for SMP guests, consistency of address translation on processors.

Software can use various techniques to keep the sPT and gPT consistent. One of the techniques is *write-protecting* the gPT. In this technique the hypervisor write-protects all physical pages that constitute the gPT. Any modification by the guest to add a translation results in a *page fault* exception. On a page fault exception, the processor control is transferred to the hypervisor so it can emulate the operation appropriately. Similarly, the hypervisor gets control when the guest edits gPT to remove a translation; the hypervisor removes the translation from the gPT and updates the sPT accordingly.

A different shadow paging technique does not write-protect gPT but instead depends on processor's page-fault behavior and on guest adhering to TLB consistency rules. In this technique, sometimes referred to as Virtual TLB, the hypervisor lets the guest add new translations to gPT without intercepting those operations. Then later when the guest accesses an instruction or data which results in the processor referencing memory using that translation, the processor page faults because that translation is not present in sPT just yet. The page fault allows the hypervisor to intervene; it inspects the gPT to add the missing translation in the sPT and executes the faulting

instruction. Similarly when the guest removes a translation, it executes INVLPG to invalidate that translation in the TLB. The hypervisor intercepts this operation; it then removes the corresponding translation in sPT and executes INVLPG for the removed translation.

Both techniques result in large number of page fault exceptions. Many page faults are caused due to normal guest behavior; such those as a result of accessing pages that have been paged out to the storage hierarchy by the guest operating system. We call such faults *guest-induced* page faults and they must be intercepted by the hypervisor, analyzed, and then reflected into the guest, which is a significant overhead when compared to native paging. Page faults due to shadow paging are called *hypervisor-induced* page faults. To distinguish between these two faults, the hypervisor traverses the guest and shadow page tables, which incurs significant software overheads.

When a guest is active, the page walker sets the accessed and dirty bits in the sPT. But because the guest may depend on proper setting of these bits in gPT, the hypervisor must reflect them back in the gPT. For example, the guest may use these bits to determine which pages can be moved to the hard disk to make room for new pages.

When the guest attempts to schedule a new process on the processor, it updates processor's CR3 register to establish the gPT corresponding to the new process. The hypervisor must intercept this operation, invalidate TLB entries associated with the previous CR3 value and set the real CR3 value based on the corresponding sPT for the new process. Frequent context switches within the guest could result in significant hypervisor overheads.

Shadow paging can incur significant additional memory and performance overheads with SMP guests. In an SMP guest, the same gPT instance can be used for address translation on more than one processor. In such a case the hypervisor must either maintain sPT instances that can be used at each processor or share the sPT between multiple virtual processors. The former results in high memory overheads; the latter could result in high synchronization overheads.

It is estimated that for certain workloads shadow paging can account for up to 75% of overall hypervisor overhead.

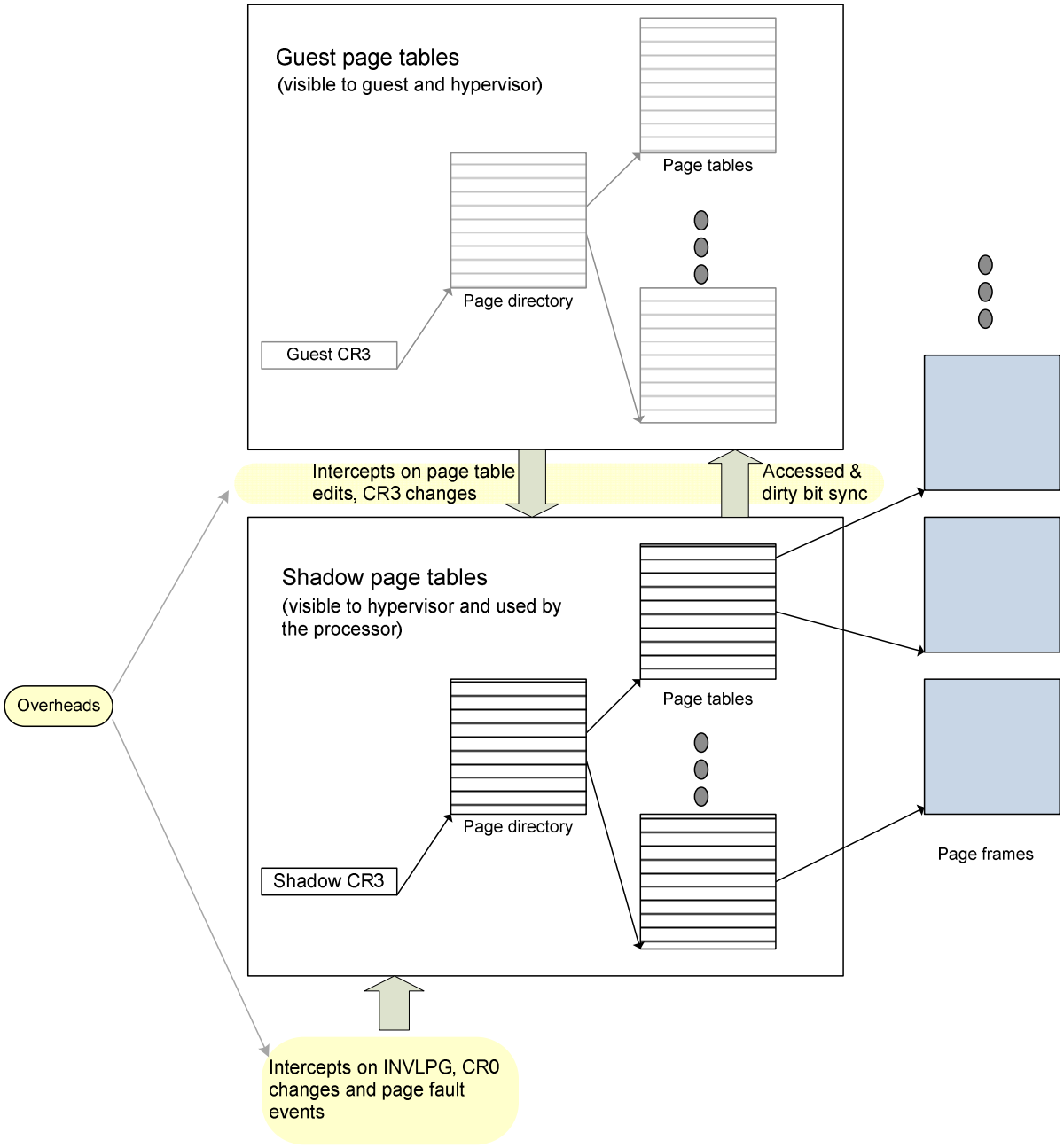

Figure 3: Guest and shadow page tables (showing two-level paging)

4.2 AMD-V™ Nested Page Tables (NPT)

To avoid the software overheads under shadow paging, AMD64 Quad-Core processors add *Nested Paging* to the hardware page walker. Nested paging uses an additional or *nested page table (NPT)* to translate guest physical addresses to system physical addresses and

leaves the guest in complete control over its page tables. Unlike shadow paging, once the nested pages are populated, the hypervisor does not need to intercept and emulate guest's modification of gPT.

Nested paging removes the overheads associated with shadow paging. However because nested paging introduces an additional level of translation, the TLB miss cost could be larger.

4.2.1 Details

Under nested paging both guest and the hypervisor have their own copy of the processor state affecting paging such as the CR0, CR3, CR4, EFER and PAT.

The gPT maps guest linear addresses to guest physical addresses. Nested page tables (nPT) map guest physical addresses to system physical addresses.

Guest and nested page tables are set up by the guest and hypervisor respectively. When a guest attempts to reference memory using a linear address and nested paging is enabled, the page walker performs a 2-dimensional walk using the gPT and nPT to translate the guest linear address to system physical address. See figure 4.

When the page walk is completed, a TLB entry containing the translation from guest linear address to system physical address is cached in the TLB and used on subsequent accesses to that linear address.

AMD processors supporting nested paging use the same TLB facilities to map from linear to system physical addresses, whether the processor is in guest or in host (or hypervisor) mode. When the processor is in guest mode, TLB maps guest linear addresses to system physical addresses. When processor is in host mode, the TLB maps host linear addresses to system physical addresses.

In addition, AMD processors supporting nested paging maintain a *Nested TLB* which caches guest physical to system physical translations to accelerate nested page table walks. Nested TLB exploits the high locality of guest page table structures and has a high hit rate.

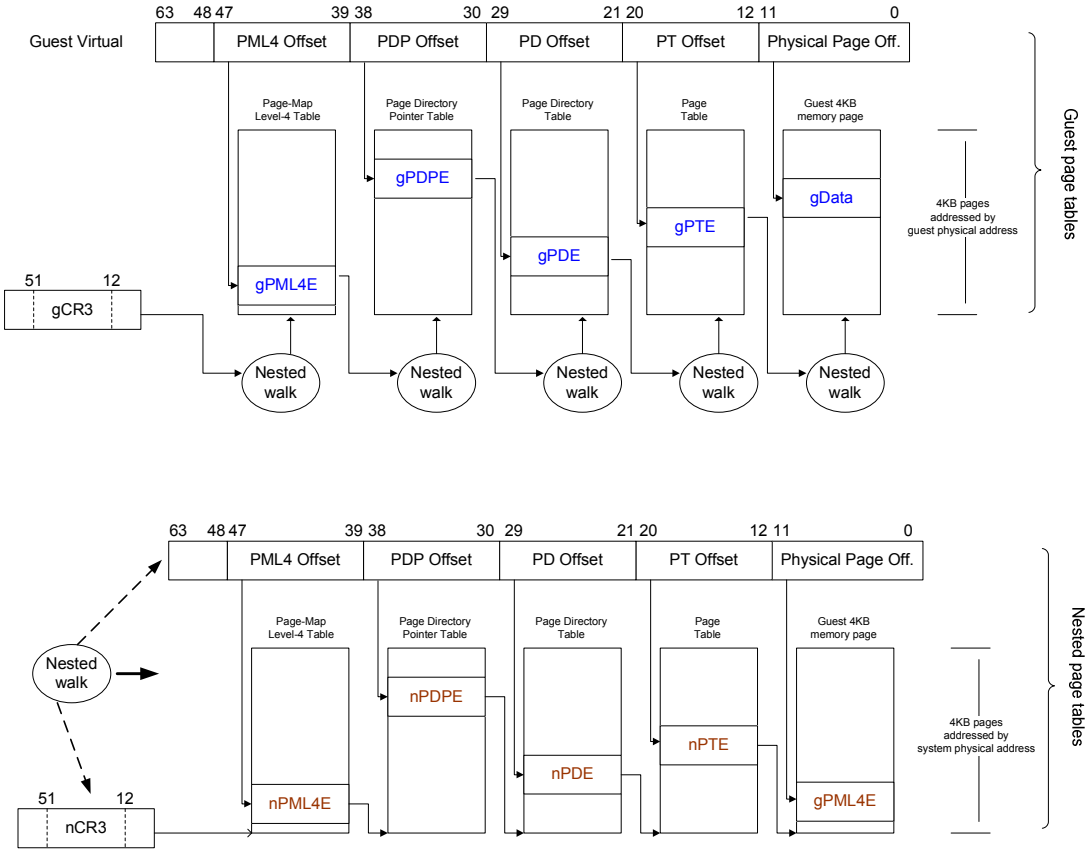

Figure 4: Translating guest linear address to system physical address using nested page tables

4.2.2 Cost of a Page Walk under Nested Paging

A TLB miss under nested paging could have a higher cost than a TLB miss under non-nested paging. This is because under nested paging, the page walker must not only walk gPT but also simultaneously walk nPT to translate the guest physical addresses encountered during guest page table walk (such as gCR3 and gPT entries) to system physical addresses.

For example, a 4-level guest page table walk could invoke the nested page walker 5 times, once for each guest physical address encountered and once for the final translation of the GP of the datum itself. Each nested page walk can require up to 4 cacheable memory accesses to determine the guest physical to system physical mapping

and one more memory access to read the entry itself. In such a case a TLB miss cost can increase from 4 memory references in non-nested paging to 24 in nested paging unless caching is done. Figure 5 shows the steps taken by the page walker with 4 levels in both guest and nested page tables.

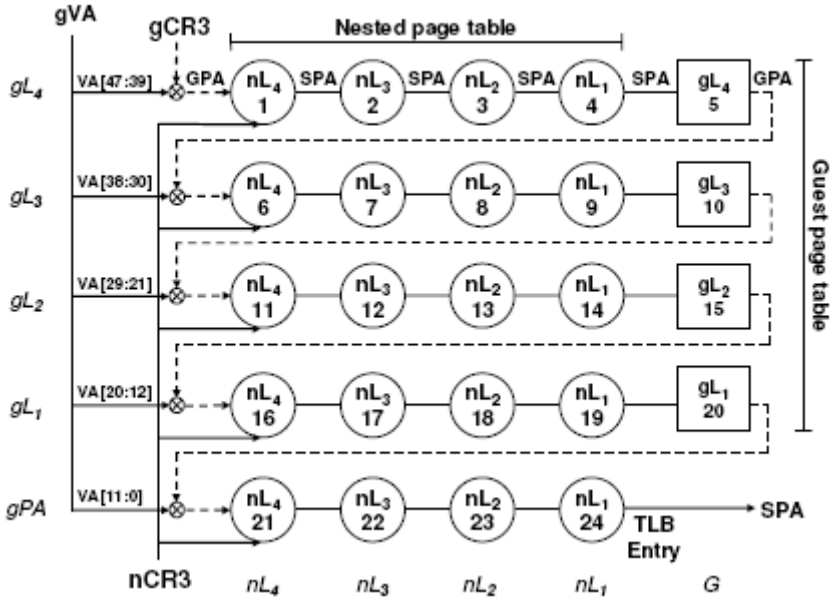

Figure 5: Address translation with nested paging. GPA is guest physical address; SPA is system physical address; nL is nested level; gL is guest level

4.2.3 Using Nested Paging

Nested paging is a feature intended for hypervisor's use. The guest cannot observe any difference (except performance) while running under a hypervisor using nested paging. Nested paging does not require any changes in guest software.

Nesting paging is an optional feature and not available in all implementations of processors supporting AMD-V technology. Software can use the CPUID instruction to determine if nested paging is supported on that processor implementation.

4.2.4 Memory savings with NPT

Unlike shadow-paging, which requires the hypervisor to maintain an sPT instance for each gPT, a hypervisor using nested paging can set up a single instance of nPT to map the entire guest physical address space. Since guest memory is compact, the nPT should typically consume considerably less memory than an equivalent shadow-paging implementation.

With nested paging, the hypervisor can maintain a single instance of nPT which can be used simultaneously at one more processor in an SMP guest. This is much more efficient than shadow paging implementations where the hypervisor either incurs a memory overhead to maintain per virtual processor sPT or incurs synchronization overheads resulting from use of shared sPT.

4.2.5 Impact of Page Size on Nested Paging Performance

Besides other factors, address translation performance is reduced when there is an increase in TLB miss cost. Other things being equal, TLB miss cost (under nested and non-nested paging) decreases if fewer pages need to be walked. Large page sizes reduce page levels needed for address translation.

To improve nested paging performance, a hypervisor can chose to populate nPT with large page sizes. Nested page table sizes can be different for each page in each guest and can be changed during guest execution. AMD64 Quad-Core processors support 4KB, 2MB, and 1GB page sizes.

Like large nested page size, large guest page size also reduces TLB miss cost. Many workloads such as database workloads typically use large pages and should perform well under nested paging.

An indirect benefit of large pages is TLB efficiency. With larger pages, each TLB entry covers a larger range of linear to physical address translation; effectively increasing TLB capacity and reducing page walk frequency.

4.2.6 Micro-architecture Support for Improving Nested Paging Performance

To reduce page walk overheads, AMD64 processors maintain a fast internal *Page Walk Cache* (PWC) for memory referenced by frequently

used page table entries. The PWC entries are tagged with physical addresses and prevent a page entry reference from accessing the memory hierarchy.

With nested paging, the significance of the PWC becomes even more important. AMD processors supporting nested paging can cache guest page table as well as nested page table entries. This would convert the unconditional memory hierarchy access for data referenced by both guest and nested page table entries to likely PWC hits. Reuse of page table entries at the top most page level is the highest while that at the lowest level is the least. PWC takes these characteristics into consideration when caching entries.

As we discussed previously, the TLB plays a major role in reducing address translation overheads. When TLB capacity increases, the number of costly page walks needed decreases. The AMD "Barcelona" family of processors are designed to cache up to 48 TLB entries in their L1 Data TLB for any page size; 4KB, 2MB or 1GB pages. They can also cache 512 4KB TLB entries or 128 2M entries in their L2 TLB. A TLB with large capacity improves performance under nested as well as shadow paging.

Similar to the regular TLB which caches linear address to system address translations, AMD processors supporting nested paging support a *Nested TLB (NTLB)* to cache guest physical to system physical translations. The goal of NTLB is to reduce the average number of page entry references during a nested walk.

The TLB, PWC and NTLB work together to improve nested paging performance without requiring any software changes in guest or the hypervisor.

4.2.7 Address Space IDs

Starting 64-bit AMD Opteron Rev-F processors support *Address Space IDs (ASIDs)* to dynamically partition the TLB. The hypervisor assigns a unique ASID value to each guest scheduled to run on the processor. During a TLB lookup, the ASID value of the currently active guest is matched against the ASID tag in the TLB entry and the linear page frame numbers are matched for a potential TLB hit. Thus TLB entries belonging to different guests and to the hypervisor can coexist without causing incorrect address translations, and these TLB entries can persist during context switches. Without ASIDs, all TLB entries must be flushed before a context switch and refilled later.

Use of ASIDs allows the hypervisor to make efficient use of processor's TLB capacity to improve guest performance under nested paging.

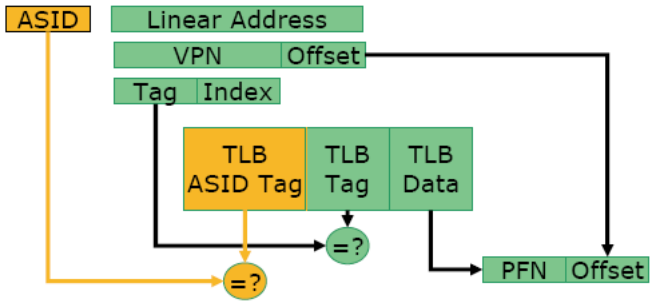

Figure 6: Address Space ID (ASID)

4.2.8 Nested Paging Benchmarks

This whitepaper includes benchmark results collected from an early revision of the AMD "Barcelona" family of processors and early hypervisor implementations. It is possible that as hypervisors get optimized for nesting paging, the overall performance will improve. Furthermore, the performance may improve with enhancements to micro architecture. As benchmark results from later revisions of AMD64 Quad-Core processors and later hypervisor versions become available, they will be added or linked to this document.

The benchmarks discussed here were collected on systems with the following configuration:

Processors: 2-socket, 2.0GHz/1800MHz-NB Barcelona (Model 2350), 95W.
Memory: 32GB of 4GB DDR2-667MHz.
HBA: QLA2432 (dual-port PCIe 2Gb Fiber) HBA: 1 port used.
Disk Array: MSA1500, 1 controller, 1 fiber connection, 512MB cache. 15 drives 15K rpm SCSI 73GB disks.

Figure 7 shows benchmark data collected with and without nested paging using an experimental build of VMware ESX. With nested paging, the performance increased by approximately 14 and 58 percent for SQL DB Hammer and MS Terminal Services workloads respectively.

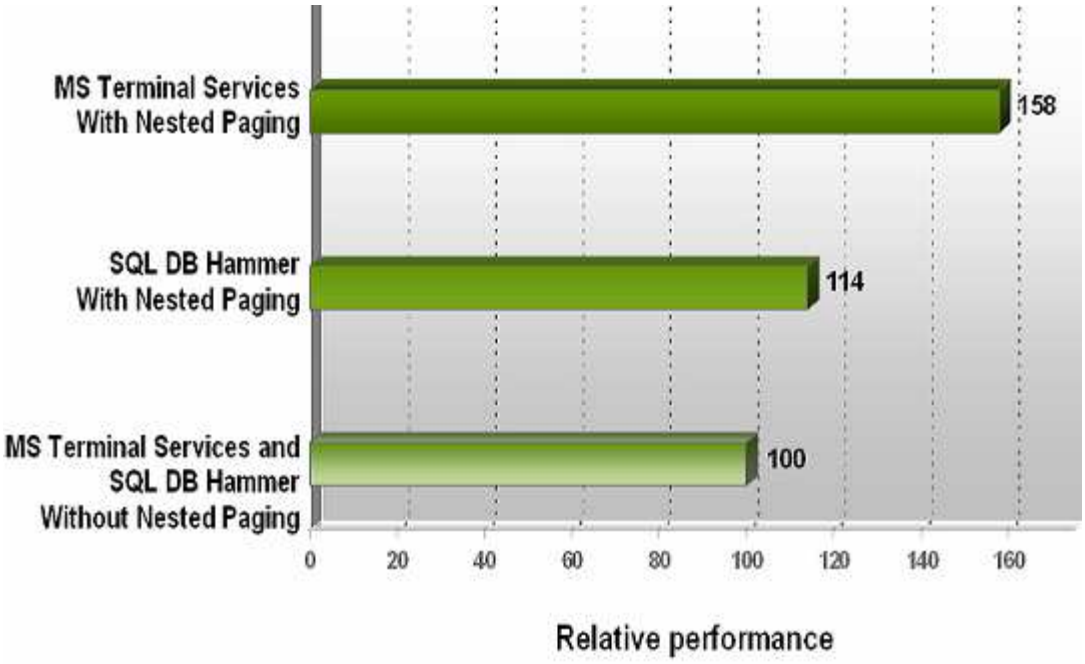

Figure 7: Performance with and without nested paging with an experimental build of VMware ESX hypervisor on AMD64 Quad-Core Model 2350

Figure 8 shows Oracle 10G OLTP with and without nested paging with RHEL 4.4 running under Xen 3.1. With nested paging, the performance increased by approximately 94%. With para-virtualized (PV) drivers for NIC and storage, the performance increased by 249%.

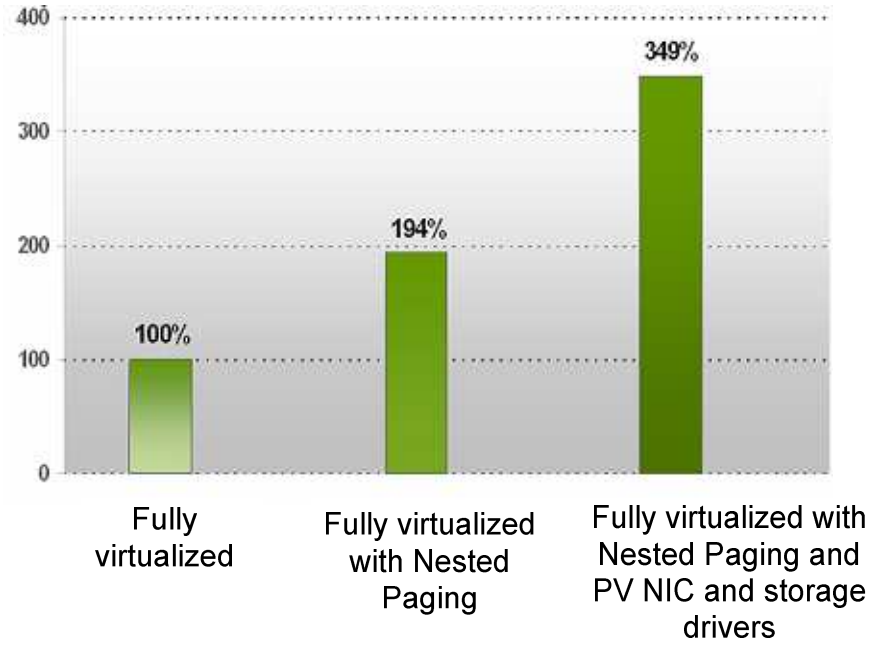

Figure 8: Oracle 10G OLTP performance with and without nested paging with RHEL 5.5 under Xen on AMD64 Quad-Core Model 2350.

5 Conclusion

Nested Paging removes the virtualization overheads associated with traditional software-based shadow paging algorithms. Together with other architectural and micro-architectural enhancements in AMD64 Quad-core processors, Nested Paging helps deliver performance improvements, specifically for memory intensive workloads with high context-switch frequency. Servers based on these processors can provide outstanding scalability, leading edge performance-per-watt, high consolidation ratios and great headroom for server workloads.