A Case for Alternative Nested Paging Models for Virtualized Systems

Giang Hoang, Chang Bae, John Lange, Lide Zhang†, Peter Dinda and Russ Joseph
Northwestern University †*University of Michigan*

Abstract—Address translation often emerges as a critical performance bottleneck for virtualized systems and has recently been the impetus for hardware paging mechanisms. These mechanisms apply similar translation models for both guest and host address translations. We make an important observation that the model employed to translate from guest physical addresses (GPAs) to host physical addresses (HPAs) is in fact orthogonal to the model used to translate guest virtual addresses (GVAs) to GPAs. Changing this model requires VMM cooperation, but has no implications for guest OS compatibility. As an example, we consider a *hashed page table approach* for GPA→HPA translation. *Nested paging*, widely considered the most promising approach, uses unhashed multi-level forward page tables for both GVA→GPA and GPA→HPA translations, resulting in a potential $O(n^2)$ page walk cost on a TLB miss, for n-level page tables. In contrast, the hashed page table approach results in an expected $O(n)$ cost. Our simulation results show that when a hashed page table is used in the nested level, the performance of the memory system is not worse, and sometimes even better than a nested forward-mapped page table due to reduced page walks and cache pressure. This showcases the potential for alternative paging mechanisms.

Index Terms—Virtualization, Computer Architecture, Virtual Memory, Nested Paging

1 Introduction

Virtual machine monitors (VMMs) must efficiently map the address space seen by a guest operating system to the true physical memory of the host system. In x86 systems, the instruction set architecture (ISA) specifies page table structures, the paging model, and translation caching/invalidation, effectively exposing them as architected features [7]. This has a profound impact on the complexity and performance of the virtualization software. On the x86, TLB misses trigger the hardware to perform a *page walk* operation that fetches the missing translation via a traversal of the multi-level page table structure. The guest operating system expects these interactions with the hardware and the VMM consequently must virtualize these aspects of the memory system. These complexities make pure software support for memory virtualization for x86 architectures challenging to implement and may expose memory virtualization as a potential performance bottleneck [2].

In response to these concerns, both AMD and Intel have added an x86 hardware virtualization support feature known as *nested paging* to recent processors. Nested paging recognizes that memory virtualization requires two separate types of memory translation: (1) guest virtual address to guest physical address (GVA→GPA) and (2) guest physical address to host physical address (GPA→HPA). *The current nested paging approach applies the same address translation model to both types of translation.* The guest operating system controls the GVA→GPA translations by directly manipulating its own page tables and doing localized TLB invalidations. The VMM controls the GPA→HPA mapping by directly manipulating a second set of page tables and TLB. These translations "nest" inside of the steps of the guest's page-walk. If the page tables are n levels deep, an $O(n)$ page-walk from the guest's perspective can, in fact, turn into an $O(n^2)$ page-walk in the hardware. The address translation process for a 64-bit host VMM running a 64-bit guest operating system is illustrated in Figure 1. In a virtualized system, up to 24 separate memory accesses may be needed to perform complete translation as opposed to a maximum of five accesses for native execution.

Manuscript submitted: 26-Mar-2010. Manuscript accepted: 21-Apr-2010. Final manuscript received: 30-Apr-2010

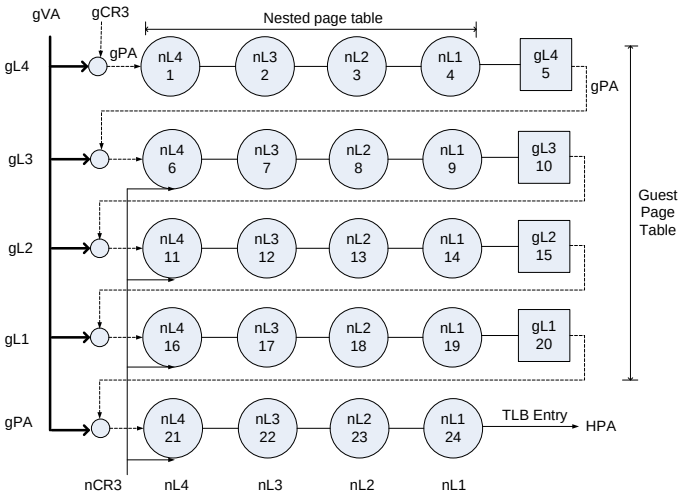

Fig. 1: Nested page walk using multi-level forward page tables for 64 bit guest and 64 bit VMM. A 4-deep GPA→HPA page walk is nested inside each step of the guest's own 4-deep page walk. This figure is a variant of that provided by Bhargava et al [2].

Intel and AMD's nested paging architectures merely duplicate the native paging model to cope with a second layer of address translation. However, in general, GPA→HPA translations are only visible to the VMM itself and not the guest operating system. This raises the question: *Is a multi-level forward page table the right approach for GPA→HPA translation?*

Our experiences with implementing support for shadow paging and nested paging in a new open source VMM suggest that, from the VMM implementation perspective, there is little benefit to having the GPA→HPA model behave like the GVA→GPA model [8]. *The model used for GPA→HPA translations need not match the model used for GVA→GPA translations.* A different GPA→HPA model could be used without breaking compatibility with guest operating systems in any way. This combination of VMM flexibility and the architectural separation of concerns at the guest and VMM levels presents an opportunity for hardware vendors to radically redesign the

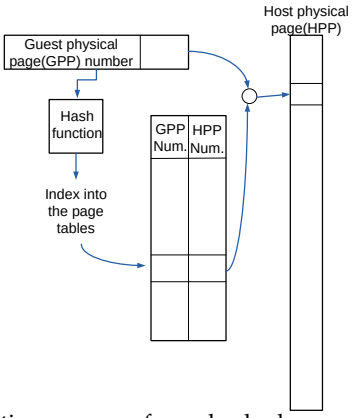

Fig. 2: Translation process for a hashed page table used for nested GPA→HPA translation. Here, each nested page walk is 1 step long, in expectation.

.

GPA→HPA translation model to more closely align with the needs and capabilities of VMMs.

This paper advocates that other nested paging or, more accurately, nested address translation, models be considered by the academic and industrial communities. As an example of an alternative paging model we consider a *hashed page table* approach [4] for GPA→HPA translation. We present a preliminary evaluation of the potential of this approach. The results show that hashed paging offers considerable promise for accelerating memory virtualization. This suggests that there is opportunity in revisiting paging models on x86 processors. We then discuss several additional alternatives.

2 Nested Paging with Hashed Page Tables

We have investigated the use of a one-level forward hashed page table for GPA→HPA translation. For each guest, a single page table is used where each entry corresponds to a frame in the guest physical memory (the memory map provided by the VMM). The size of the page table scales with amount of guest physical memory. In our description and evaluation, we assume a single general purpose hash function, but our VMM could also provide a per-guest perfect hash function. Using a hashed page table of this kind, while retaining the legacy paging model for GVA→GPA translation, a nested page walk would be an expected $O(2 \times n)$ steps long.

The translation process: The GPA→HPA translation is illustrated in Figure 2. It starts with applying a hash function on the guest physical frame number to obtain an index into the page table. Each entry in the table contains a guest physical frame number and a host physical frame number to which it maps. A comparison is made between the guest physical frame number given and the one contained in the entry. If there is no match, we advance to the next entry and repeat. When a match occurs, the host physical frame number is concatenated with the page offset from the guest physical address to obtain the exact location in host physical memory. If no match is found, a page fault occurs, which induces a VM exit. The VMM then attempts to resolve the situation. Since all usable guest physical pages should have their own entry in the page table, a page fault indicates an access to a region in guest physical address space that does not correspond to a location on usable guest physical memory, although it may correspond to a memory-mapped virtual device.

The VMM can arrange for the mapping from guest physical address to host physical address to be relatively simple, and, given knowledge or control over the hash function, can arrange that it results in few collisions. Therefore, a nested translation usually requires one hash computation and one memory access, compared to as many as four memory accesses given the model of Figure 1.

TABLE 1: Architectural Configuration

TLBs	ITLB/DTLB: 64-entry, fully-associative
Memory Hierarchy	IL1/DL1: 64 KB, 2 way, 2-cycle latency L2: 512 KB, 16 way, 9-cycles latency L3: 2 MB, 32 way, 50-cycle latency Memory: 4 GB, 250-cycle latency
Page Walk Cache	24-entry, fully associative, 2-cycle latency
Nested TLB	16-entry, fully associative, 2-cycle latency

The hashed page table organization requires more space per table entry, because it needs to store both guest and host physical frame numbers. We conservatively estimate the size of each entry to be 16 bytes (considering 64 bit address space for both guest and host). For 4GB of guest physical memory and a 4KB page size, we will need 2^{20} entries. The size of the hashed page table scales linearly with the size of guest physical memory. In our example, the size of the page table would be 16MB, a (constant) 0.4% overhead.

VMM design considerations: To support hashed page tables for GPA→HPA translation, the VMM needs to be able to designed to build and manipulate these tables, as well as to emulate a nested page walk that uses them. As long as the hash function is known, or supplied by the VMM, and the hashed page table format is well documented, this can be readily done. For our VMM, we believe the implementation complexity would be on par with that of our implementation of nested paging support on AMD SVM, and simpler than that of our general shadow paging implementation.

We expect that a typical implementation will pre-allocate the hashed page table during initial guest construction. At this point, the VMM has an implementation-independent guest physical memory map and is aware of all GPAs that will map to physical memory. This map will remain constant for the life of the guest in almost all cases, and thus it could be readily used to populate the table, and even to construct a custom hash function, if desired. If the map[1] changes, the VMM would rebuild the table. The primary limitation of the hashed page table approach is that it makes lazy instantiation of nested page table structures much harder.

Hardware Complexity: Hashed page tables have been studied extensively by Huck et al [4], and an implementation of hashed page tables in the PA-RISC machine is described by Jacob et al [6]. Huck notes that the implementation of an HPT hardware miss handler has complexity similar to that of a 2-level TLB in the previous generation of PA-RISC. This suggests that a nested HPT approach is unlikely to be more complex than a nested forward page table approach. Hashed page tables do require extra hardware to implement a hash operation, but Huck et al conclude that a simple XOR function is sufficient and requires minimal additional hardware. Hashed page tables may rely on a PWC and nested TLB to accelerate translations, but these components are also found in the best current nested forward page table approach described by Bhargava et al [2].

3 Methodology

In evaluating the hashed page table model and other models, we consider the average memory latency seen by the guest. We assume a VMM-independent system, and do not model VM exits and entries.

1. It is important to note that we are referring here to the guest's physical memory map—the GPAs that are valid in the guest—not how those addresses map to host addresses. The latter can change dynamically.

We use Simics [10] to execute a benchmark in a copy of Linux Fedora Core 5, collecting traces of memory references from both the benchmark and the guest OS. Guest context switches are also captured in the trace, their effects being presented in the form of TLB invalidations (e.g., MOV to CR3, INVLPG*, etc). Simics also provides the specific memory accesses due to each page table walk.

We consider a range of benchmarks that have different memory system behavior. These include the TPC-C benchmark for on-line transaction processing [9] the TPC-H decision-support benchmark [3]), a subset of SPEC CPU2000 [11], and the compilation of the Linux 2.6.14 kernel. We also use a microbenchmark that stresses nested page translation. This microbenchmark accesses a very large array at a regular interval far apart enough that each access requires a new translation entry in both the TLB and nested TLB, causing faults in both translation levels.

To model the effects of nested paging, we developed a memory simulator which takes as input the GPA→HPA mapping, a nested paging model, and a memory reference trace. Our memory simulator models the memory hierarchy as closely as possible to a recent model AMD Opteron processor, specifically in the form described in Table 1, with the choice of nested paging model being optional. We compare several different nested paging models, as described below (a 4KB page size is used in both guest and host memory system).

- *NATIVE*: Native system (without any nested paging). In addition to the TLB configuration described in Table 1, a "1D" page-walk cache (PWC) (cf. [5]) is included. This gives us an upper limit on performance.
- *BASE*: Baseline system with nested multi-level forward page tables, but without page walk caching.
- *FPT2D+NTLB*: Nested multi-level forward page table with a nested TLB and a "2D" PWC. The nested TLB (NTLB) serves to significantly improve the performance of nested paging by directly caching GPA→HPA translations. This configuration is the one recently proposed by AMD to dramatically speed up the current nested multi-level forward page table model [2]. As far as we are aware, it represents the current state of the art in nested paging implementation for the multi-level forward page table model.
- *HPT1D*: Nested hashed page table with 1D PWC.
- *HPT1D+NTLB*: Nested hashed page table with nested TLB (NTLB) and 1D PWC.

It is important to realize that for each of these models, page walks are also accelerated (and thus memory latency reduced) by the fact that page table entries can reside in data caches.

4 Results

The results of our experiments can be summarized as follows.

- The hashed page table model is a viable alternative, performing comparably to the state-of-the-art model [2] for many benchmarks, and never performing worse.
- For some benchmarks, the hashed page table model performs significantly better than the state-of-the-art model.
- In these cases, the hashed page table model performs better because it is able to avoid nested page-walk cache references altogether, and reduce nested L2 cache accesses.

Figure 3 shows the number of cycles spent in the memory system, including both cache stalls and TLB translations, normalized to native system performance. In comparison to the base system, the nested hashed table model reduces the memory access latency significantly for all our benchmarks. Its performance is on par with the current state-of-the-art nested

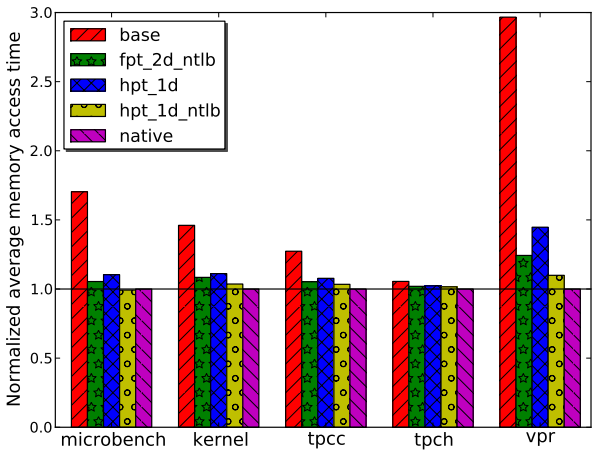

Fig. 3: Performance Comparison.

.

forward page table [2]. Even *without* the support of an NTLB, the hashed page table model provides performance very close to the state-of-the-art approach. With an NTLB, performance is slightly closer to native performance than that approach.

The SPEC2000 vpr benchmark shows the most differentiation between the various models. Even with an NTLB and a 2D PWC, the nested multi-level forward page table model performs 24% worse than native. In contrast, the nested hashed page table model with a 1D PWC and an NTLB suffers only a 10% loss of performance.

It may be surprising that SPEC2000 vpr shows a more pronounced difference between the native and baseline virtualized systems than does the microbenchmark. A detailed analysis of the number of cycles spent in page walks reveals that, while the microbenchmark is designed to cause misses in the nested TLB, it also results in significant TLB misses in the native case. The execution time of the microbenchmark is dominated by TLB miss handling in both cases, making the difference between them less pronounced than we see for SPEC2000 vpr.

To better understand the source of the performance difference for SPEC2000 vpr, we analyzed the number of memory accesses due to page walks for each scheme as well as the location in the memory hierarchy where these walks are serviced. Figure 4 plots the number of page walks generated by each model, normalized to the number performed by the baseline nested paging model. These page walks are further decomposed into where they occur and how they are satisfied: guest(g) or nested(n) paging levels; and service in PWC, L2, L3 cache or the main memory.

Figure 4 shows that the primary reason for the hashed page table model's improvements result from a reduction in page walks. Hashed paging with an NTLB gives a number of page-walks that approaches that of the native system. Even without an NTLB, the number of page walks in in hashed paging is very close to multi-level forward paging supported with a NTLB. The addition of an NTLB to hashed paging eliminates many of the nested page walks originally serviced by the L2 cache. This means that hashed paging significantly reduces the pressure that page translation places on the memory system during page walks. Overall, this translates into significantly improved memory access times.

Beyond the SPEC2000 vpr results, we have also noticed an across-the-board reduction in the total number of page table entry accesses. This accounts for a noticeable reduction in memory stall time for many of the workloads.

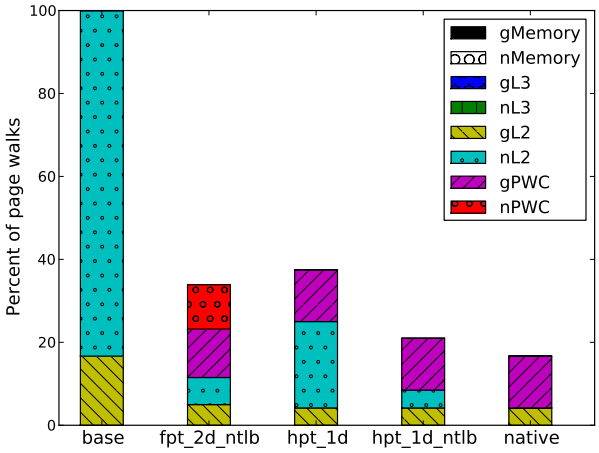

Fig. 4: Cache-ability of SPEC CPU 2000 vpr benchmark.

5 Other models

The results of the previous section suggest that there is considerable promise to exploring alternative paging models in virtualized systems. We now consider other paging models that could be employed.

Software-managed nested TLB: Instead of enforcing a specific page table model in the instruction set architecture, the ISA could instead provide for a software-managed nested TLB, exiting to the VMM whenever an NTLB miss occurred. This would allow the VMM to adopt whatever paging model would be most appropriate for the current workload. The guest would continue to see the legacy (hardware-based) paging model. While at the present time this concept would appear quite expensive as VM exit/entry costs are very high in AMD and Intel's virtualization extensions, there doesn't appear to be any intrinsic reason why this will remain so.

Segments/extents: Some VMMs employ a simple GPA→HPA mapping, trying to keep physical memory contiguous at the host and guest levels. One advantage of this, particularly in high performance computing, is to make the memory system maximally predictable at the guest level. Such a mapping could be compactly represented as a collection of run-length-encoded extents provided by the VMM to the hardware. For example, in our VMM, when creating a memory map comparable to a commodity PC on an HPC system, it could be represented with $O(10)$ extents.

With the decline of segmentation in 64 bit x86, one could imagine mapping these extents into segmentation. Each segment described in the GDT or an LDT is already defined by a base address and length in pages. A special segment type that also contained a guest physical base address as a tag, and a corresponding host physical base address, could be created. Nested translation would then consist of a lookup through those special segments. With a small number of entries in the segment table, a small and very fast cache close to the core could be dedicated to these lookups, for example extending the existing segment descriptor cache. Using a fully associative cache with logic to support the $\leq$ relation, all comparisons could be done in parallel in a single cycle. Since the VMM is in full control of the GPA→HPA mapping, it could limit the number of extents used, balancing between memory traffic due to missing in this cache and flexibility in its mapping.

Further discussion of software alternatives and of adaptive paging is given in a separate technical report [1].

6 Conclusion

The overhead associated with memory translation, especially in page walks, results in significant slowdown in virtualized x86 systems. We have made the observation that the translation model applied to map from a guest physical address to a host physical address is orthogonal to the translation model used to map from a guest virtual address to a guest physical address. Current hardware uses identical models for these two very different translations. Adopting a different model for the guest physical to host physical translation would not present any compatibility challenges for the guest operating system, and existing VMMs could be readily modified to support it. This creates an opportunity to revisit paging models even in the widely used commodity x86.

We next considered an example of an alternative model, a hashed page table, and compared it to several implementations of the conventional model, including an implementation that is considered to be state-of-the-art. Our results show that hashed paging achieves significant reduction in memory access time compared to the alternatives. Even compared to the state-of-the-art, it never performs worse and in some instances performs considerably better due to reduced numbers of page walks and more efficient handling of page walks that do occur. We considered the implementation of hashed paging in our own VMM and found that it's complexity would be on par with that of the conventional model.

Our results suggest that there are potential performance gains to be had in revisiting address translation in virtualized systems and discarding the assumption that the "nested" layer of translation should employ the same model as the (backward compatible) "guest" layer. We described several other alternatives, and we advocate further exploration of different models.

References

[1] Chang Bae, John Lange, and Peter Dinda. Comparing approaches to virtualized page translation in modern VMMs. Technical Report NWU-EECS-10-07, Department of Electrical Engineering and Computer Science, Northwestern University, April 2010.

[2] Ravi Bhargava, Ben Serebrin, Francesco Spanini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

[3] TPC benchmark H (decision support) standard specification, revision 2.8.0 (2008). http://www.tpc.org/tpch/spec/tpch2.8.0.pdf.

[4] Jerry Huck and Jim Hays. Architectural support for translation table management in large addres sapce machines. *ACM SIGARCH Computer Architecture News*, 21(2):39–50, May 1993.

[5] Intel Corporation. *TLBs, Paging-Structure Caches, and Their Invalidation (revision 003)*, December 2008.

[6] Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 295–306, 1998.

[7] Paul Karger. Performance and security lessons learned from virtualizing the alpha processor. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, June 2007.

[8] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Steven Jaconette, Mike Levenhagen, Ron Brightwell, and Patrick Widener. Palacios and kitten: High performance operating systems for scalable virtualized and native supercomputing. Technical Report NWU-EECS-09-14, Department of Electrical Engineering and Computer Science, Northwestern University, July 2009.

[9] Diego R. Llanos. TPCC-UVa: An open-source implementation for global performance measurement of computer systems. *ACM SIGMOD Record*, 35(4):6–15, December 2006.

[10] P. Magnusson, M. Christensson, J. Esklison, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2), February 2002.

[11] SPEC CPU 2000 benchmark. http://www.spec.org/cpu2000/.