2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Agile Paging: Exceeding the Best of Nested and Shadow Paging

Jayneel Gandhi, Mark D. Hill, Michael M. Swift

Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI, USA
{*jayneel,markhill,swift*}*@cs.wisc.edu*

***Abstract*—Virtualization provides benefits for many workloads, but the overheads of virtualizing memory are not universally low. The cost comes from managing two levels of address translation—one in the guest virtual machine (VM) and the other in the host virtual machine monitor (VMM)—with either nested or shadow paging. Nested paging directly performs a two-level page walk that makes TLB misses slower than unvirtualized native, but enables fast page tables changes. Alternatively, shadow paging restores native TLB miss speeds, but requires costly VMM intervention on page table updates.**

This paper proposes *agile paging* that combines both techniques and exceeds the best of both. A virtualized page walk starts with shadow paging and optionally switches in the same page walk to nested paging where frequent page table updates would cause costly VMM interventions. Agile paging enables most TLB misses to be handled as fast as native while most page table changes avoid VMM intervention. It requires modest changes to hardware (e.g., demark when to switch) and VMM policies (e.g., predict good switching opportunities).

We emulate the proposed hardware and prototype the software in Linux with KVM on x86-64. Agile paging performs more than 12% better than the best of the two techniques and comes within 4% of native execution for all workloads.

***Keywords*-virtual memory; virtual machines; virtualization; translation lookaside buffer; nested paging; shadow paging.**

I. Introduction

Virtualization forms the foundation of our cloud infrastructure. It provides many benefits including security, isolation, server consolidation and fault tolerance. These benefits became achievable because various hardware and software advances have substantially reduced the cost of virtualization [5, 19, 20]. Nevertheless, even with hardware and software acceleration [10, 25, 32] the overhead of virtualizing memory can still be high.

To fully virtualize memory, two levels of address translation are used:

gVA⇒gPA: guest virtual address to guest physical address translation via a per-process guest OS page table (gPT)

gPA⇒hPA: guest physical address to host physical address via a per-VM host page table (hPT)

There are two state-of-the-art techniques to virtualize memory which provide different tradeoffs. First, the widely used hardware technique called nested paging [19] generates a TLB entry (gVA⇒hPA) to map guest virtual address directly to host physical address enabling fast translation. On a TLB miss, hardware performs a long-latency 2D page walk which walks both page tables. For example, in x86-64, TLB misses require up to 24 memory references [19] as opposed to a native 1D page walk requiring up to 4 memory references. However, this technique benefits from fast direct updates to both page tables without VMM intervention.

Table I
Trade-off provided by both memory virtualization techniques as compared to base native. Agile paging exceeds best of both worlds.

	Base Native	Nested Paging	Shadow Paging	Agile Paging
TLB hit	fast (VA⇒PA)	fast (gVA⇒hPA)	fast (gVA⇒hPA)	fast (gVA⇒hPA)
Max. memory access on TLB miss	4	24	4	∼(4—5) avg.
Page table updates	fast direct	fast direct	slow mediated by VMM	fast direct
Hardware support	1D page walk	2D+1D page walk	1D page walk	2D+1D page walk with switching

Second, the lesser-used technique called shadow paging [57] requires the VMM to build a new *shadow page table* (gVA⇒hPA) from both page tables. It points hardware to the shadow page table (sPT), so that TLB hits perform the translation (gVA⇒hPA) and TLB misses do a fast native 1D page walk (e.g., 4 memory references in x86-64). However, page table update requires VMM to perform substantial work to keep the shadow page table consistent [10].

Table I summarizes the trade-offs provided by the two techniques to virtualize memory as compared to base native. With current hardware and software, the overheads of virtualizing memory are hard to minimize because a VM exclusively uses one technique or the other.

Past work—*selective hardware software paging (SHSP)*—showed that a VMM could dynamically switch an entire guest process between nested and shadow paging to achieve the best of either technique [58]. It monitored TLB misses and guest page faults to periodically consider switching to the best mode. However, switching to shadow mode requires (re)building the *entire* shadow page table, which is expensive for multi-GB to TB workloads.

We take inspiration from and extend this approach with *agile paging* to *exceed* the best of both techniques. Intuitively, most of the updates to a hierarchical page table occur at the lower levels or leaves of the page table. With that *key intuition*, agile paging allows virtualized page walk to start with the shadow paging for stable upper levels of page table and allows switching in the same page walk to nested

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.67

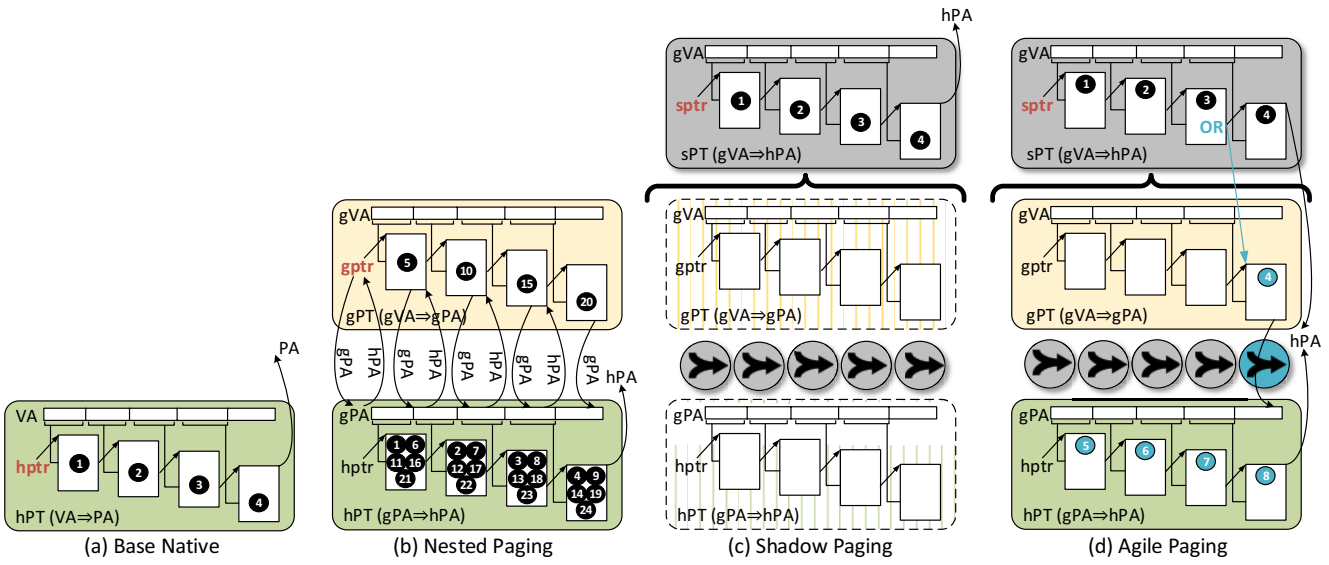

Figure 1. Different techniques of virtualized address translation as compared to base native. Numbers indicate the memory accesses to various page table structures on a TLB miss in chronological order. The merge arrows denotes that the two page tables are merged to create shadow page table. Colored merge arrows with agile paging denotes partial merging at that level. The starting point for translating an address is marked in bold and red.

paging for lower levels of page table which receive frequent updates. This allows a guest virtual address space to use both shadow and nested paging *at the same time* with varying degree of nesting and allows switching from one mode to the other in the middle of a page walk. This reduces the cost of a TLB miss since most of the TLB misses are handled fully or partially with shadow paging and reduces the costly VMM interventions by allowing fast direct updates to the page tables. Table II shows varying degrees of nesting and memory references for page walks in x86-64 depending on when the switch from shadow to nested paging occurs. Our evaluation in Section VII shows that agile paging requires fewer than 5 memory references per TLB miss on average. One can think of SHSP [58] as a temporal solution, while agile paging is both temporal and spatial, where spatial may grow in importance with increasing memory footprint.

Agile paging builds on existing hardware for virtualized address translation, requiring only the modest hardware change of enabling switching between the two modes. In addition, to further reduce VMM interventions associated with the shadow technique within agile paging, we propose two optional hardware optimizations. Similarly, VMM support for agile paging builds upon existing support and requires modest changes.

Figure 1 shows address translation techniques for base native, nested paging, shadow paging, and our technique of agile paging. The numbers in the figure show the chronological order in which different levels of the page table structures are accessed on a TLB miss. The page tables are hashed in shadow paging since the hardware has no access to the guest or host page tables. Agile paging is color coded to show two of the options available from Table II. The black colored path

Table II
NUMBER OF MEMORY REFERENCES WITH VARYING DEGREE OF NESTING PROVIDED BY AGILE PAGING IN A FOUR-LEVEL X86-64-STYLE PAGE TABLE AS COMPARED TO OTHER TECHNIQUES.

Levels of Page Table	Base Native	Nested Paging	Shadow Paging	Agile Paging
PTptr: Page table pointer	0	4	0	0 or 4
L4: Page table level 4 entry	1	5	1	1 or 5
L3: Page table level 3 entry	1	5	1	1 or 5
L2: Page table level 2 entry	1	5	1	1 or 5
L1: Page table entry (PTE)	1	5	1	1 or 5
All	4	24	4	4-24

shows shadow paging in agile paging. With the blue colored escape path, agile paging allows the hardware to switch from shadow paging to nested paging for the leaf-level of page table, requiring up to 8 memory accesses per translation. The switch from shadow to nested can be performed at any level of the page table (not shown).

We emulate our proposed hardware and prototype our proposed software in KVM on x86-64. We evaluate our design with variety of workloads and show that our technique improves performance by more than 12% compared to the best of nested and shadow paging.

In summary, the contributions of our work are:

1) We propose a mechanism *agile paging* that simultaneously combines shadow and nested paging to seek the best features of each within a single address space.
2) We propose two optional hardware optimizations to further reduce overheads of shadow paging.
3) We show that agile paging performs better than the best of shadow and nested paging.

```
host_walk(VA, hptr)
PA = hptr;
for (i=0; i≤MAX_LEVELS; i++) do
  | PA = host_PT_access(PA + index(VA,i));
end
return PA;
```
(a) Base native page walk state machine

```
nested_walk(gVA, gptr, hptr)
//page fault in hPT will cause a VM exit
hPA = host_walk(gptr, hptr);
for (i=0; i≤MAX_LEVELS; i++) do
  | hPA = nested_PT_access(hPA + index(gVA,i), hptr);
end
return hPA;
```
(b) Nested page walk state machine

```
shadow_walk(gVA, sptr)
return host_walk(gVA, sptr);
```
(c) Shadow page walk state machine

```
host_PT_access(address)
PTE = *address;
PA = PTE.PA;
if PTE.valid == 0 then
  | raise host PAGE_FAULT;
return PA;
```
(d) Host page table access helper function

```
nested_PT_access(address, hptr)
PTE = *address;
gPA = PTE.PA;
if PTE.valid == 0 then
  | raise guest PAGE_FAULT;
//page fault in hPT will cause a VM exit
hPA = host_walk(gPA, hptr);
return hPA;
```
(e) Nested page table access helper function

Figure 2. Pseudocode of hardware page walk state machine for native and existing virtualized address translation techniques.

II. Background

A. Nested Paging

Nested paging is a widely used hardware technique to virtualize memory. The processor has two page table pointers to perform a complete translation: one points to the guest page table (gptr) and the other points to the host page table (hptr). The guest page table holds gVA to gPA translation and the host page table holds gPA to hPA translations.

In the best case, the virtualized address translation hits in the TLB to directly translate from gVA to hPA with no overheads. In the worst case, a TLB miss needs to perform a 2D page walk that multiplies overheads vis-à-vis native, because accesses to the guest page table also require translation by the host page table. Figure 1(b) depicts virtualized address translation for x86-64. It shows how the page table memory references grow from a native 4 to a virtualized 24 references: 4 access to translate gptr (since each gPA requires access to host page table) and each of the 4 levels of the guest page table (guest page table holds gPA) plus 4 references for the guest page table itself to obtain the final hPA: $4 \times 5 + 4$ references. Figure 2 describes the hardware page walk state machine for nested paging (see (b) and helper functions (a) and (e)). Note that various caching techniques in today's commodity processors, like caching of page table entries in data caches [36], MMU caches [15, 21] and caching intermediate translations [19, 20] can remove some of the memory references for a TLB miss.

Even though the TLB misses are longer than native, nested paging allows fast direct updates to both of the page tables without any VMM intervention.

B. Shadow Paging

Shadow paging is a lesser used software technique to virtualize memory. With shadow paging, the VMM creates a *shadow page table* (on demand) by merging the guest and host tables, then holds a complete translation from gVA⇒hPA.

In the best case, the virtualized address translation hits in the TLB to directly translate from gVA to hPA with no overheads. On a TLB miss, the hardware performs a 1D page walk on the shadow page table. The native page table pointer points to the shadow page table. Thus, the memory references required for shadow page table walk are the same as a base native walk. For example, x86-64 requires up to 4 memory references on a TLB miss for shadow paging as well as base native address translation (shown in Figure 1 (c)). In addition, as a software technique, there is no need for any extra hardware support for page walks. The hardware page walk is shown in Figure 2 (c) with helper functions (a) and (d).

Even though the TLB misses cost the same as native execution, this technique does not allow direct updates to the page tables since the shadow page table needs to be kept consistent [10]. Every page table update requires a costly VMM intervention (VMtraps) to fix the shadow page table by invalidating or updating its entries. These VMM interventions cause significant overheads in many applications. For example, to mark a page copy-on-write by a guest OS, shadow paging requires at least two VMtraps costing 1000s of cycles: one to write to the guest page table to mark the page read-only and one to force a TLB flush. Similarly, context switches require a VMtrap for the VMM to determine the shadow page table for the incoming process. These costly VMtraps are not required for nested paging or native paging. Note that we define VMtrap latency as the cycles required for a VMexit trap and its return plus the work done by the VMM in response to the VMexit.

III. Agile Paging Design

We propose *agile paging* as a lightweight solution to the cost of virtualized address translation. We observe that shadow paging has lower overheads than nested paging, except when guest page tables change. Our *key intuition* is that page tables are not modified uniformly: some regions of

an address space see far more changes than others, and some levels of the page table, such as the leaves, are updated far more often than the upper-level nodes. For example, code regions may see little change over the life of a process, whereas regions that memory-mapped files may change frequently.

We use this key intuition to propose agile paging that combines the best of shadow and nested paging by:

1) using shadow paging for fast TLB misses for the parts of the guest page table that remain static, and
2) using nested paging for fast in-place updates for the parts of the guest page tables that dynamically change.

We refer to these two memory virtualization techniques as *constituent techniques* for the rest of the paper. We show that agile paging performs better than its constituent techniques and supports features of conventional paging on both guest OS and VMM.

In the following subsections, we describe the hardware mechanism which will enable us to use both constituent techniques at the same time for a guest process and discuss policies that are used by the VMM to reduce overheads.

A. Mechanism: Hardware Support

Agile paging allows using the constituent techniques for the same guest process—even on a single address translation—uses modest hardware support to switch between the two. Agile paging has three architectural page table pointers in hardware: one each for shadow, guest, and host page tables. If agile paging is enabled, virtualized page walk starts in shadow paging and then switches, in the same page walk, to nested paging if required. To allow fine grain switching from shadow paging to nested paging on any entry at any level of guest page table, the shadow page table needs to logically support a new *switching bit* per page table entry. This notifies the hardware page table walker to switch from shadow to nested mode. We choose not to support the switching in the other direction (nested to shadow mode) since the updates to the page tables are mostly confined to the lower levels of the page tables. When the switching bit is set in a shadow page table entry, the shadow page table holds the hPA (pointer) of the next *guest page table* level.

There are different degrees of nesting for virtualized address translation with agile paging: full shadow paging, full nested paging, and four degrees of nesting where translation starts in shadow mode and switches to nested mode at any level of the page table. These are shown in increasing order of page walk latency in Figure 3. The hardware page-walk state machine that uses the switching bit to allow this flexible paging mechanism is shown in Figure 4. A modest change needed to switch between the two techniques; the rest of the state machine is already present to support the constituent techniques. This change is shown in red in Figure 4.

Page Walk Caches: Modern processors have hardware page walk caches (PWCs) to reduce the number of memory

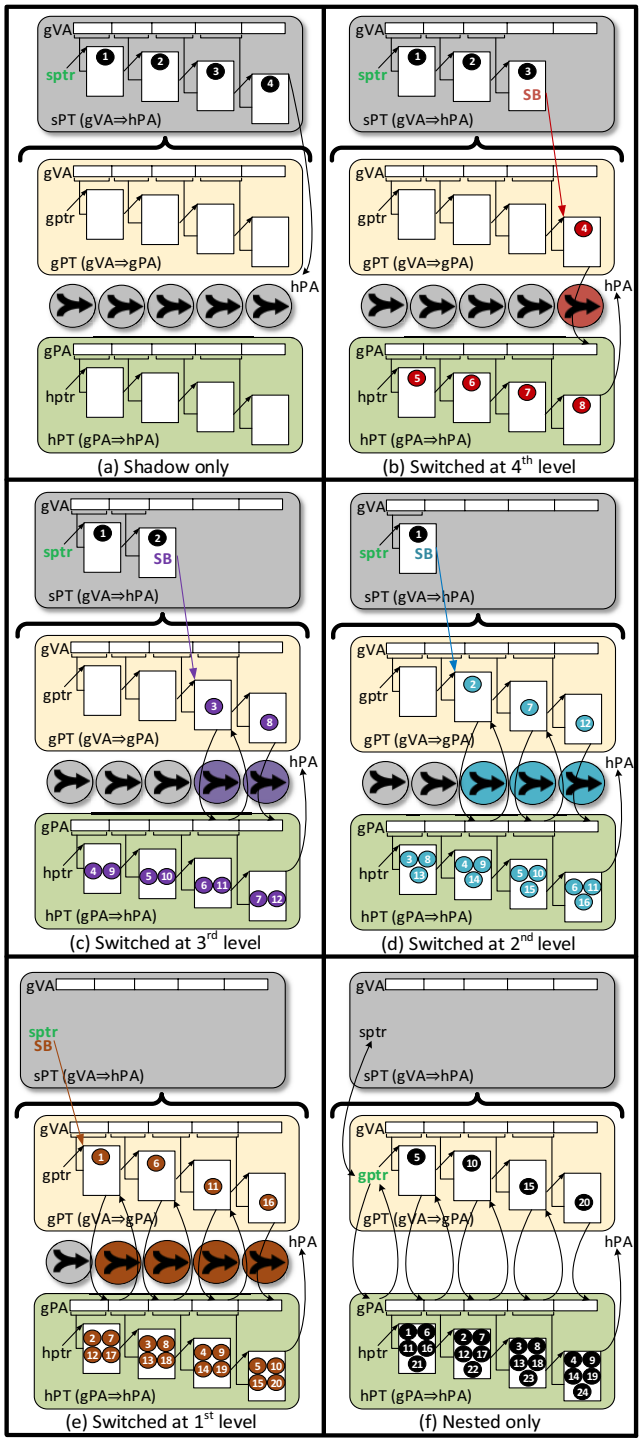

Figure 3. Different degrees of nesting with agile paging in increasing order of page walk latency. Starting point for each is marked bold and green.

accesses required for a page walk by caching the most-recently-used partial translations. For example, Intel processors use three partial translation tables inside PWCs: one table each to help skip the top one, two, or three levels of the page table [15, 21]. With shadow paging, PWCs store

```
agile_walk(gVA, gptr, hptr, sptr)
if sptr==gptr then
    return nested_walk(gVA, gptr, hptr);
else
    nested = sPT.switching_bit;
    hPA = sptr;
    for (i=0; i≤MAX_LEVELS; i++) do
        if nested then
            hPA = nested_PT_access(hPA + index(gVA,i), hptr);
        else
            hPA = host_PT_access(hPA + index(gVA,i));
            PTE = *(hPA + index(gVA,i));
            //Switching to nested mode
            if PTE.switching_bit then
                nested = true;
        end
    end
end
return hPA;
```

Figure 4. Pseudocode of hardware page walk state machine for agile paging. Note that agile paging requires modest switching support (shown in red) in addition to the state machines of nested and shadow paging.

the hPA as a pointer to the next level of the shadow page table and thus skip accessing a few levels of the the shadow page table. With nested paging, PWCs store the hPA as a pointer to the next level of the guest page table, and skip accessing some of the levels of guest page table as well their corresponding host page table accesses. With agile paging, PWCs can be used to store partial translations for up to three levels of the guest page table without any restrictions on which mode any of the levels may be in. The PWCs will store an hPA for the partial translation with a single bit to denote whether the hPA points to shadow or guest page table so that agile page walk can continue in the correct mode. While we explained extension to Intel's PWCs, agile paging can support other designs as well.

B. Mechanism: VMM Support

Like shadow paging, the VMM for agile paging manages three page tables: guest, shadow, and host. Agile paging's page table management is closely related to that of shadow paging, but there are subtle differences.

Guest Page Table (gVA⇒gPA): As with shadow paging, the guest page table is created and modified by the guest OS for every guest process. The VMM, though, controls access to the guest page table by marking them read-only. Any attempt by the guest OS to change the guest page table will lead to a VMM intervention, which is used to update the shadow page table to maintain coherence [10].

With agile paging, we leverage the support for marking guest page tables read-only with one subtle change. The VMM marks as read-only just the parts of the guest page table covered by the partial shadow page table. The rest of the guest page table (considered under nested mode) has full read-write access. Section III-C describes policies to choose what part of page table is under which mode. For example, KVM [43] allows the leaf level of a guest page table to be writable temporarily, called an unsynced shadow page, allowing multiple updates without intervening VMtraps. We extend that support to make other levels of the guest page table writable in our prototype.

Shadow Page Table (gVA⇒hPA): As with shadow paging, for all guest processes with agile paging enabled, a shadow page table is created and maintained by the VMM. The VMM creates this page table by merging the guest page table with the host page table so that any guest virtual address is directly converted to a host physical address. The VMM creates and keeps the shadow page table consistent [10].

However, with agile paging, the shadow page table is partial and cannot translate all gVAs fully. The shadow page table entry at each switching point holds the hPA of the next level of guest page table with the switching bit set (as shown in Figure 3). This enables hardware to perform the page walk correctly with agile paging using both techniques.

Host Page Table (gPA⇒hPA): As with shadow paging, the VMM manages the host page table to map from gPA to hPA for each virtual machine. VMM merges this page table with the guest page table to create a shadow page table. The VMM must update the shadow page table on any changes to the host page table. The host page table is only updated by the VMM and during that update the shadow page table is kept consistent by invalidating affected entries.

For standard shadow paging, the host page table is never referenced by hardware, and hence VMM can use other data structures instead of the architectural page-table format. However, with agile paging, the processor will walk the host page table for addresses using nested mode (at any level), and hence the VMM must build and maintain a complete host page table for each guest virtual machine as in nested paging.

Accessed and Dirty Bits: As with shadow paging, accessed and dirty bits are handled by the VMM and kept consistent between shadow page table and guest page table. On the first reference to a page, the VMM sets the accessed bit in the guest PTE and in the newly created shadow PTE. The write-enable bit is not propagated to the new shadow PTEs from the guest PTE. This ensures that the first write to the page will cause a protection fault, which causes a VMtrap that checks the guest PTE for write enable bit. At this point, the dirty bit is set in both the guest and shadow PTEs, and the shadow PTE is updated to enable write access to the page. If the guest OS resets any of these bits, the writes to guest page table are intercepted by the VMM which invalidates (or updates) the corresponding shadow PTEs.

With agile paging, we use the same technique for pages completely translated by shadow mode. Pages that end in nested mode instead use the hardware page walker, available for nested paging, to update guest page table accessed and dirty bits. We describe an optional hardware optimization in Section IV that improves handling of accessed and dirty bits by eliminating costly VMtraps involved with shadow mode.

Context-Switches: Context switches within the guest OS are fast with nested paging, since guest OS is allowed to write to guest page table register. But with shadow paging, the VMM must intervene on context switches to determine the shadow page table pointer for the next process.

With agile paging, the context switching follows the mechanism used by shadow paging for all processes. The guest OS writes to the guest page table register, which triggers a trap to the VMM. The VMM finds the corresponding shadow page table and sets it in the shadow page table register. Hence, the cost of a context switch with agile paging is similar to shadow paging. We describe an optional hardware optimization in Section IV that improves context switches in a guest OS by eliminating costly VMtraps involved in shadow paging.

To summarize, the changes to the hardware and VMM to support agile paging is incremental, but they result in a powerful, efficient and robust mechanism. The design is applicable to architectures that support nested page tables (e.g., x86-64 and ARM) and any hypervisor can use this architectural support. The hypervisor modifications are modest if they support both shadow and nested paging (e.g., KVM [43], Xen [14], VMware [57] and HyperV [4]).

C. Policies: What degree of nesting to use?

Agile paging provides a mechanism for virtualized address translation that starts in shadow mode and switches at some level of the guest page table to nested mode. The purpose of a policy is to determine *whether* to switch from shadow to nested mode for a single virtualized address translation and at *which level* of the guest page table the switch should be performed.

The ideal policy would determine that the page table entries are changing rapidly enough and the cost of corresponding updates to the shadow page table outweighs the benefit of faster TLB misses in shadow mode and thus the translation should use nested mode. The policy would quickly detect the dynamically changing parts of the guest page table and switch them to nested mode while keeping the rest of the static parts of the guest page table under shadow mode. Note that programs with very few TLB misses should use nested paging for the whole address space, as shadow mode has no benefit.

To achieve the above goal, a policy will move some parts of the guest page table from shadow to nested mode and vice-versa. We assume that the guest process starts in full shadow mode. We propose one static policy to move parts from shadow to nested mode and two online policies to move parts back from nested to shadow mode.

Shadow⇒Nested mode: Detecting dynamically changing parts of a guest page table is convenient when these parts are in shadow mode. These parts are marked read-only, thus any attempt to change an entry requires a VMM intervention (Section III-B). Agile paging uses this to track the dynamic parts of the guest page table in the VMM and move those parts to nested mode.

To design a policy, we observed that updates to a page in page table are bimodal at a time interval of 1 second: only one update or many updates (e.g., 10, 50 or 500) within a second. Similar observations were made by Linux-KVM developers and used it to guide unsyncing a page of shadow page table [2]. For agile paging, if two writes to any level of the page table are detected by the VMM in a fixed time interval, then that level and all levels below it are moved to nested mode. This policy provides a small threshold like the one used in branch predictors for switching modes.

Nested⇒Shadow mode: The second, more complex, part of the policy is to detect when the workload changes behavior and stops changing the guest page table dynamically. This requires the switching parts of the guest page table back from nested to shadow mode to minimize TLB miss latency.

Our first simple online policy moves all the parts of the guest page table from nested back to shadow mode at fixed time interval and then use the above policy to move dynamic parts of the guest page table back to nested mode. While this policy is simple, it can lead to high overheads if the parts of the guest page table oscillate between the two modes.

A second more complex but effective policy uses dirty bits on the nested parts of the guest page table to detect changes to the guest page table itself. Under this policy, at the start of a fixed time interval, the VMM clears the dirty bits on the host page table entries mapping the pages of the guest page table. At the end of the interval, the VMM scans the host page table to which guest page table pages have dirty bits, which indicates the dynamic parts of the guest page table under nested mode. The non-dynamic parts of the guest page table (pages which did not have the dirty bit set) are switched back to shadow mode. The parent level of the guest page table is converted to shadow mode before converting child levels.

Short-Lived or Small Processes: Nested paging has been shown to be performing well for short-lived processes and for processes that have a very small memory-footprint since they do not run long enough to amortize the cost of constructing a shadow page table or do not suffer from TLB misses [20]. With agile paging, an administrative policy can be made to start the process in nested mode (no use of shadow mode) and turn on shadow mode after a small time interval (e.g., 1 sec) if TLB miss overhead is sufficiently large. The VMM can measure the TLB miss overhead with help of hardware performance counters and perform the switch to use agile paging.

To summarize, with our proposed policies, the VMM detects changes to the page tables and intelligently makes a decision to switch modes to reduce overheads.

IV. Hardware Optimizations

Shadow paging was developed as a software-only technique to virtualize memory before there was hardware support. We propose two optional hardware optimizations that can further reduce the number of VMtraps associated with shadow paging and agile paging's shadow mode.

Handling Accessed and Dirty Bits: Agile paging requires costly VMtraps to keep accessed and dirty bits synchronized for regions of guest page table under shadow mode. Unlike shadow paging, in agile paging, the hardware has access to all three page tables (guest, host, shadow). As a result, we propose to extend hardware to set the accessed/dirty bit in all three page tables rather than just in the shadow. The extra page walk required to perform the write of accessed/dirty bits requires a full nested walk (up to 24 memory accesses) will be faster than a long VMtrap and thus more efficient. In addition, recent Intel Broadwell processors introduced two hardware page walkers per-core to help handle multiple outstanding TLB misses and writing accessed/dirty bits. Similar hardware for page walkers can be leveraged to perform writes to all page tables in parallel.

Context-Switches: With every guest process context switch, the guest OS writes to the guest page table register, but is not allowed to set the shadow page table register since it does not have knowledge about the shadow page table. This results in costly VMtraps on context switches, which can degrade performance for workloads that do so frequently. In order to avoid these VMtraps, we propose adding a small 4-8 entry hardware cache to hold shadow page table pointers and their corresponding guest page table pointer, similar to how a TLB holds physical page numbers corresponding to virtual frame numbers. This cache can be filled and managed by the VMM (with help of new virtualization extensions) and accessed by the hardware on a context switch. So, if the guest OS writes to guest page table pointer register, hardware quickly checks this cache to see if there exists a shadow page table pointer corresponding to that guest process. On a hit, the hardware sets the shadow page table register without a VMtrap.

V. Paging Benefits

Agile paging is flexible and supports all features of conventional paging. We next describe how three important paging features that are supported with agile paging.

Large Page Support: Current processors support larger page sizes (2MB and 1GB pages in x86-64) by reducing the levels of the page tables and mapping larger regions of aligned contiguous virtual memory to aligned contiguous physical memory. Larger page sizes reduce the number of TLB misses for workloads with large working sets. Larger page sizes are supported at either or both stages for virtualized address translation with both shadow and nested paging; when large pages are used only in one stage of translation (e.g., guest only), they are in effect broken into smaller pages for entry into the TLB.

Table III
System configurations and per-core TLB hierarchy.

Processor	Dual-socket Intel Xeon E5-2430 (Sandy Bridge), 6 cores/socket 2 threads/core, 2.2GHz
Memory	96 GB DDR3 1066MHz
OS	Linux kernel version 3.12.13
VMM	QEMU (with KVM) version 1.6.2, 24vCPUs
L1 DTLB	4 KB pages: 64-entry, 4-way associative 2 MB pages: 32-entry, 4-way associative 1 GB pages: 4-entry, fully associative
L1 ITLB	4 KB pages: 128-entry, 4-way associative 2 MB pages: 8-entry, fully associative
L2 TLB	4 KB pages: 512-entry, 4-way associative 2 MB pages:

With agile paging, larger page sizes are supported using their current implementation in shadow paging and nested paging. The shadow page table, guest page table and host page table support larger page sizes as they all are multi-level page tables and can reduce their depth. Thus, agile paging supports larger page sizes using the same mechanisms and policies described in Section III.

Content-Based Page Sharing: Content-based page sharing is a technique used by both the guest OS and the VMM to save memory for many workloads. The guest OS or VMM scans memory to find pages with identical content. When such pages are found, the guest OS or VMM reclaims all but one copy and maps all the copies using copy-on-write [57]. This allows pages to be shared within a process, between two guest processes and even between two virtual machines. Reclamation by the guest OS requires changes to the guest page table (and shadow page table if applicable) whereas reclamation by the VMM requires changes to the host page table (and shadow page table if applicable).

As copy-on-write must update the page table, agile paging will naturally detect these changes and move parts of the page table to nested mode to reduce overheads. The overhead of copy-on-write is very high with shadow paging and will benefit from nested mode provided by agile paging.

Memory pressure: When free memory is scarce, a guest OS will frequently scan and clear the referenced bits of page tables looking for pages to reclaim (e.g., clock algorithm). With shadow paging, this scanning causes VMtraps, which increases overhead on an already stressed system. With agile paging, though, the VMM detects the page-table writes to clear referenced bits and converts leaf-level page tables to nested mode to avoid the VMtraps.

To summarize, existing page-based mechanisms blend naturally with our proposed technique. This shows that agile paging has the *agility* to adapt to changing environments and is powerful to reduce overheads of memory virtualization.

Table IV
PERFORMANCE MODEL BASED ON PERFORMANCE COUNTERS AND BADGERTRAP.

Ideal execution time (from base native)	$E^{ideal} = E^{2M} - T^{2M}$
Overhead of page walks (for both 4K and 2M)	$PW_{B/N/S} = [E_{B/N/S} - E^{ideal} - H_{B/N/S}]/E^{ideal}$
Overhead of VMM (for both 4K and 2M)	$VMM_{B/N/S} = H_{B/N/S}/E^{ideal}$
Avg. cycles per TLB miss (for both 4K and 2M)	$C_{B/N/S} = T_{B/N/S}/M_{B/N/S}$
Overhead of page walk (A) (for both 4K and 2M)	$PW^A = [C_N * \sum_{i=2}^{4}(F_{Ni}) + C_S * (1 - \sum_{i=1}^{4} F_{Ni}) + (C_N + C_S) * 0.5 * F_{N1}] * M_B$
Overhead of VMM (A) (for both 4K and 2M)	$VMM^A = O_S - \sum_{i=1}^{n}(F_{Vi} * C_{Ei})$

VI. METHODOLOGY

To evaluate our proposal, we emulate our proposed hardware with Linux and prototype our software in KVM.

State of the art: We compare against six configurations: base native (B), nested paging (N) and shadow paging (S), each with two page sizes 4KB and 2MB. The prior work, SHSP [58] performs similarly to the best of shadow and nested paging, so we do not evaluate it separately.

We run workloads to completion on real hardware as described in Table III. We use Linux's *perf utility* [6] to measure (i) total execution cycles for all six configurations: base native (E_B), nested paging (E_N) and shadow paging (E_S) for both page sizes, (ii) number of TLB misses ($M_{B/N/S}$) (iii) cycles spent on TLB misses ($T_{B/N/S}$) (iv) cycles spent in the hypervisor ($H_{B/N/S}$) (v) number of VMtraps ($V_{B/N/S}$) for each page size. For the virtualized setting, we use the same page size for both levels of address translation since they only reduce TLB misses when both have same page size. For using 2MB and 4KB page sizes transparently, we turn on transparent huge page support in Linux [9]. Note that the effects of various caching techniques like caching of PTEs in data caches [36], page walk caches [15, 21], Intel EPT caches and nested TLBs [19, 20] are already included in the performance measurement since they are part of the base commodity processor.

Linux does not use 1GB pages transparently and instead requires applications to explicitly use 1GB pages. Thus, we did not include that configuration. Agile paging supports 1GB page size (Section V) and has the potential to reduce overheads of page walks with 1GB page size.

We calculate the overhead of page walks and that of VMM interventions and report those for each of the six configurations based on the performance model described in Table IV.

Agile Paging: We use a novel two-step approach to report improvements for agile paging. We first generate a trace of page table updates from KVM and then use that trace with BadgerTrap [31] to calculate performance using a linear model. Next we describe our novel two-step methodology.

Step 1: The goal of this step is to create a list of dynamically changing gVAs to classify under nested mode and calculate the fraction of VMtraps (F_{Vi}) that agile paging eliminates with reason "i". This emulates the optional hardware optimizations for agile paging as well. The workload is run to completion with shadow paging on real hardware (described in Table III) with an instrumented VMM to create a per-vCPU trace of all updates to the guest and host page table that lead to shadow page table updates. We use the trace-cmd tool [8] with KVM, modified to print extra trace information, for creation of the trace in this step. We process the trace to find which areas of the page tables are changing by looking at the reasons for VMtraps. We record the gVAs being dynamically changed due to changes on any level of the page table. This helps us create four lists of gVAs under nested mode corresponding to their switching level of page table. The lists of gVAs are considered under nested paging for step 2. This step also finds the fraction of VMM interventions that agile paging will reduce (F_{Vi}) from the trace since areas of the page table under nested paging are known. Note that we emulate our shadow-to-nested policy in an offline fashion when processing the trace.

Table V
WORKLOAD DESCRIPTION AND MEMORY FOOTPRINT.

Suite	Description	Input	Memory
SPEC 2006	compute and memory intensive single-threaded workloads	astar	350 MB
		gcc	885 MB
		mcf	1.7 GB
PARSEC	Shared-memory multi-threaded workloads	canneal	780 MB
		dedup	1.4 GB
BioBench	Bioinformatics single-threaded workloads	tigr	610 MB
Big Memory	Generation, compression and search of graphs	Graph500	73 GB
	In-memory key-value cache	Memcached	75 GB

Step 2: The goal of this step is to find the fraction of TLB misses that would be serviced under nested mode (F_{Ni}) for each level "i" the switch occurs. The workload is again run to completion, but this time with nested paging along with BadgerTrap [31]: a tool that converts all x86-64 TLB misses to a trap, which allows us to analyze TLB misses and classify TLB addresses, while enabling full-speed execution of instructions with TLB hits. We instrument the TLB misses and classify them under shadow mode or nested mode at each switching level. We compare TLB miss addresses against the gVAs for nested mode to find the fraction of TLB misses serviced in nested mode at each switching level i (F_{Ni}). We conservatively assume that when a TLB miss is serviced in nested mode F_{N1} pays half the cost of a nested TLB miss beyond native and the rest F_{N2}, F_{N3} and F_{N4} pays full cost of nested paging. This assumption leads to higher overheads for agile paging than with real hardware.

Cost of VMtraps: We measure VMtrap latency as the cycles to complete the VMM intervention: the VMexit operation and return plus the work done by the VMM in response to the VMexit. The total cost varies and can be 1000s of cycles. We use LMbench [3] and microbenchmarks to measure the cost of the VMtrap for a context switch, page

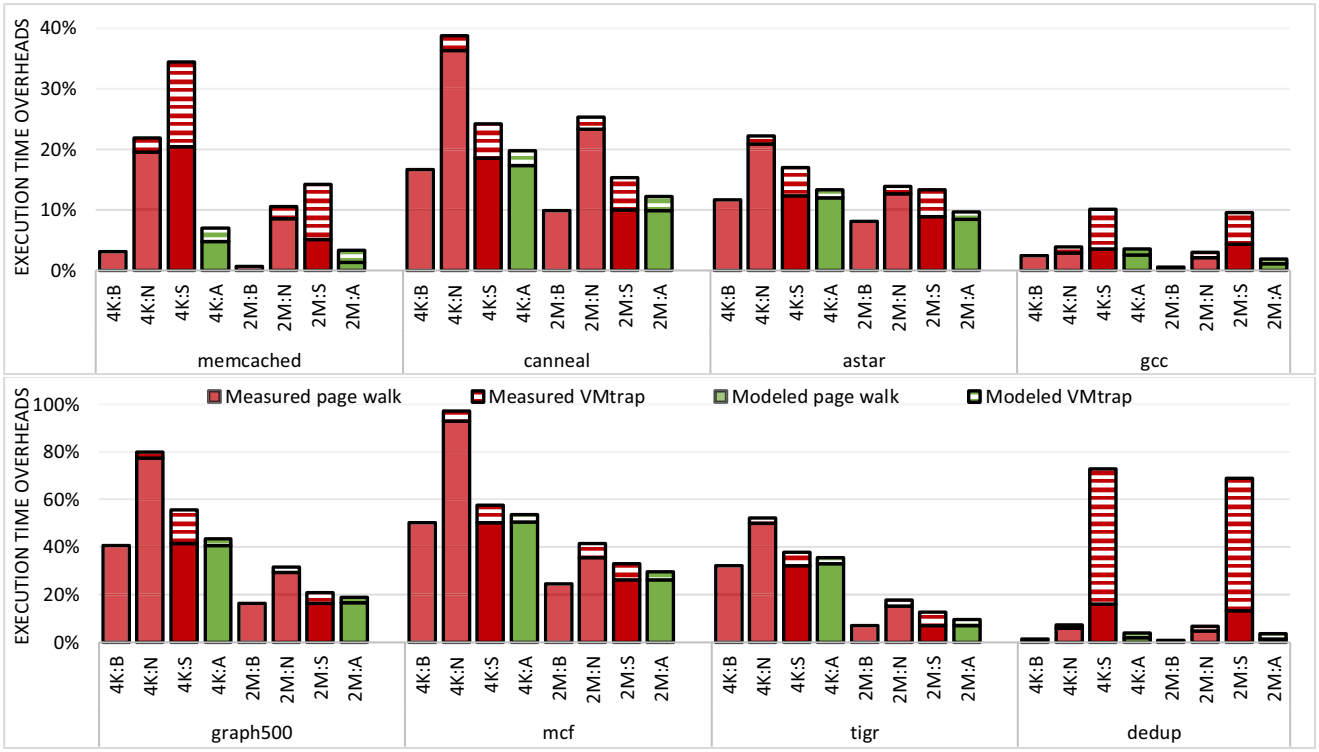

Figure 5. Execution time overheads due to page walks (bottom bar) and VMM interventions (top dashed bar) for all workloads.

table update and page fault. We calculate reduction in cycles spent in the VMM, by subtracting the number of VMtraps of each type multiplied by its cost.

Performance Model: We use the fractions calculated using the above two step approach to develop a linear model to project performance for agile paging (A) with both page sizes (i) fraction of VMtraps reduced (F_{Vi}) and, (ii) fraction of TLB misses serviced under nested mode (F_{Ni}) for both page sizes. The linear model takes the performance of shadow paging, subtracts the cost of VMtraps avoided, but adds in the higher cost of nested TLB misses. This linear performance model is similar to previous research [21, 32, 40, 49].

The new two step approach with linear model enables us to evaluate realistic workloads (in time and data footprint). We use a linear performance model to illuminate trends. Our two-phase approach includes the real system effects of page sharing, zapping of shadow entries, page walk cache, etc., and approximates the performance impact of these effects on agile paging.

Benchmarks: Our proposal is applicable to a wide variety of workloads from desktop to big-memory workloads. To evaluate our proposal, we select workloads with high TLB-miss overhead (more than 5MPKI) from SPEC [35], PARSEC [24], BioBench [13], and big-memory workloads [17] as summarized in Table V.

VII. Results

This section evaluates the cost of address translation and VMM interventions of agile paging, and shows that agile paging outperforms state-of-the-art techniques.

A. Performance analysis

Figure 5 shows the execution time overheads associated with page walk and VMM interventions for each workload in eight different configurations: base native paging (bars 4K:B and 2M:B), nested paging (bars 4K:N and 2M:N), shadow paging (bars 4K:S and 2M:S) and agile paging (bars 4K:A and 2M:A). Each bar is split into two segments. The bottom segment represents the overheads associated with page walks and the top dashed segment represents the overheads associated with VMM interventions.

Agile paging outperforms its constituent techniques for all workloads and improves performance by 12% over the best of nested and shadow paging on average, and performs less than 4% slower than unvirtualized native at worst. We find:

1) *For most workloads, the base native system with 4KB pages incurs high overheads*. For example, mcf, tigr and graph500 spend 50%, 32% and 41% respectively.
2) *With virtualization, the overheads increase drastically under nested paging.* For example, the overheads increase from 50% to 97% for mcf (bar 4K:B vs. bar 4K:N). Compared to native, translation overheads in-

crease by 2.5× with nested paging using 4KB pages (geometric mean).

3) *Shadow paging has high cost due to VMM interventions for some workloads.* While shadow paging generally performs better than nested paging, dedup, memcached and gcc have high overheads with shadow paging. Dedup has 57% overhead spent in the VMM servicing page table updates. Compared to native, translation overheads increase by 3.1× with shadow paging using 4KB pages (geo. mean).
4) *Agile paging outperforms the best of shadow paging and nested paging for all workloads.* Agile paging achieves low cost of VMM interventions as well low latency TLB misses. Agile paging is only 2.3% slower than native on average (up to 3.7%). This includes the benefit of hardware optimizations.
5) *2MB large pages help reduce overheads of virtual memory. Agile paging helps reduce overheads further.* Large pages consistently improve performance with agile paging as compared to the 4KB page size.

B. Insights into Performance of Agile Paging

We report the fraction of TLB misses covered by each mode of agile paging in Table VI for 4KB pages. For this table alone, we assume no page walk caches are present. We see that more than 80% of TLB misses are covered under complete shadow mode allowing fast TLB misses. Thus, few of the pages suffering TLB misses also have frequent page-table updates. By converting the changing portion of the guest page table to nested mode, agile paging is able to prevent most of the VMexits that makes shadow paging slower. We also note that most of the upper-levels of the page table remain static after being initialized and hence use shadow mode. Overall, the average number of memory accesses for a TLB miss comes down from 24 to between 4-5 for all workloads.

C. Discussion: Selective Hardware Software Paging

SHSP seeks to select the best paging technique dynamically [58]. Their results for a 64-bit VM running SPEC workloads showed that SHSP can achieve approximately the best of the two techniques. While SHSP, with no hardware support, improves on a single system-wide choice, it is limited by the high cost of each virtualization techniques alone. Agile paging breaks that limitation and exceeds the best of shadow and nested paging by more than 12% on average, even with a pessimistic model. Moreover, Table VI shows that only part of the address space needs switching from one mode to the other.

Table VI
PERCENTAGE OF TLB MISSES COVERED BY EACH MODE OF AGILE PAGING WHILE USING 4KB PAGES ASSUMING NO PAGE WALK CACHES. MOST OF THE TLB MISSES ARE SERVED IN SHADOW MODE.

Switch Level / Mem. accesses	Shadow 4	L4 8	L3 12	L2 16	L1 20	Nested 24	Avg.
memcached	88.2%	4.5%	7.3%	0%	0%	0%	4.76
canneal	94.7%	4.6%	0.7%	0%	0%	0%	4.24
astar	92.3%	7.5%	0.2%	0%	0%	0%	4.32
gcc	81.6%	11.7%	6.7%	0%	0%	0%	5.00
graph500	99.8%	0.2%	0%	0%	0%	0%	4.01
mcf	99.1%	0.9%	0%	0%	0%	0%	4.04
tigr	88.3%	7.6%	3.1%	0%	0%	0%	4.51
dedup	91.4%	2.2%	6.4%	0%	0%	0%	4.60

VIII. RELATED WORK

We already discussed closely related work on memory virtualization techniques: nested paging [19], shadow paging [57], and selective hardware software paging [58, 61].

Virtualization: Virtualization has been used since the 1970s [34] to run multiple operating systems on a single machine by introducing a layer of indirection between hardware and the operating systems, called the hypervisor or virtual machine monitor (VMM). Virtualization's renaissance began with Disco in 1996 [26], and since then the overheads of virtualization have been decreasing [10, 19]. Hardware support for virtualization has greatly reduced the number and latency of VMM interventions [5, 7, 19]. Binary translation can help reduce some of these overheads further [10, 11]. A 2013 industrial study showed that virtualizing memory still has very high overheads [25].

Virtual Memory: Virtual memory has been an active area of research and various previous work have shown that TLB misses lead to performance loss for many applications [17, 18, 19, 22, 23, 32, 40, 42, 44]. The performance degradation is exacerbated when running in a virtualized system [12, 19, 25, 32].

One way of reducing the overheads of virtual memory is to accelerate the page walk, which reduces the latency of a TLB miss. Commodity processors cache the page table entries in the their data caches to reduce page walk latency [36]. Most processors today have memory management unit (MMU) caches that cache partial translations or top levels of the page table to reduce the TLB miss latency [15, 21]. Other proposals like cooperative caching help increase the effective size of the TLB [44]. Also, shared last-level TLB [22] or cooperative TLBs [53] accelerate the page walk for multithreaded applications. Prefetching of PTEs or TLB entries can hide TLB miss latency [19, 39, 50]. Various software-defined caching structures like TSBs in SPARC [54], and software managed sections of TLB in Itanium [1] pin entries into the TLB to improve performance. These techniques are orthogonal to our work and could be used to accelerate agile paging

For virtualized systems, there are caching structures like MMU caches, to store partial or intermediate translations. A special cache called the nested TLB was proposed to cache the nested translations (gPA⇒hPA) to reduce latency of TLB miss with virtualization [19]. Intel hardware calls this EPT TLB, but uses the same TLB structure physically [20]. Our results include some of these caching effects as we run on real hardware. Software managed TLBs have also been

proposed for virtual machines [27]. However, our focus is on hardware managed TLBs.

Ahn et al. [12] proposed a flat page table to replace x86-64 page tables for the second stage of translation (gPA⇒hPA). This reduces the number of nested page table memory accesses from 4 to 1 and a 2D page walk from 24 to 8 accesses. While this work is promising, it does not support large pages and still requires up to 8 memory references for a TLB miss, while agile paging needs as few as 4 accesses.

Another way of reducing the overheads of virtual memory is to reduce the number of TLB misses. Larger page size have been the primary driver to reduce TLB misses [29, 30, 52, 55, 56]. Most modern processors like x86-64, ARM, MIPS, ALPHA, PowerPC and UltraSPARC support multiple page sizes [37]. Alternatively, Barr et. al. proposed speculative translation based on huge pages [16]. Similarly, Papadopoulou et al. [46] proposed a prediction mechanism that allows all page sizes to share a single set-associative TLB. Also, Du et al. [28] proposed mechanisms to allow huge pages to be formed even in the presence of retired physical pages. Recently, Pham et al. [49] proposed to use the maximum of the two page sizes instead of minimum when different page sizes are used with two-levels of address translation. These approaches can help reduce TLB misses and can be used even with agile paging.

Operating system support for large pages haves been shown to be tricky [30, 45, 56]. There have been various proposals on coalescing contiguous adjacent TLB entries or clustering TLB entries to increase the reach of the TLB [47, 48]. All of these proposals come with their own limitations [40].

Recent proposals for using segments along with paging like direct segments [17], virtualized direct segments [32] and redundant memory mappings (RMM) [33, 40, 41] show a promising new direction to reduce overheads of virtual memory. Both techniques based on direct segments [17, 32] trade off various features of paging to extract performance, whereas agile paging supports all features of paging. RMM adds multiple ranges in addition to paging, but has not been extended to support virtualization. These techniques rely on additional hardware and some software support beyond what is needed for agile paging, but in return promise better performance.

Another option to reduce TLB overheads is to remove the TLB from critical path with virtual caches [18, 38, 51, 59, 60]. These techniques perform well for certain workloads, but introduce challenges for address aliasing. In addition, the common use of content-based deduplication in virtualized system makes synonyms a larger problem.

IX. Summary

We and others have found that the overheads of virtualizing memory can be high. This is true, in part, because currently guest processes must choose between (i) nesting paging with slow 2D page table walks or (ii) shadow paging wherein page table updates cause costly VMM interventions. Ideally, one would want to use nested paging for addresses and page table levels that change, and use shadow paging for addresses and page table levels that are relatively static.

Our proposal—*Agile Paging*—seeks the above ideal. With Agile Paging, a virtualized address translation usually starts in shadow mode and then switches to nested mode only if required to avoid VMM interventions. Agile paging requires a new hardware mechanism to switch modes during a page walk in response to a page-table-entry bit set by the VMM. The VMM policy seeks to invoke the mode change only on more dynamic page table entries.

We emulate the proposed hardware and prototype the software in Linux with KVM on x86-64. We find that, for our workloads, agile paging robustly performs better than the best of nested paging and shadow paging.

Acknowledgments

We thank our anonymous reviewers, Arkaprava Basu, Vasileios Karakosas, Jason Lowe-Power, Benjamin Serebrin, Samuel Peters and Hongil Yoon for their insightful comments and feedback on the paper. This work is supported in part the US National Science Foundation (CCF-1218323, CNS-1302260, CCF-1438992, and CCF-1533885), Google, and the University of Wisconsin (Kellett award and Named professorship to Hill). Hill has a significant financial interest in AMD.

References

[1] "Intel itanium architecture developer's manual, vol. 2."

[2] "KVM MMU Virtualization," https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_guangrong.pdf.

[3] "LMbench: Tools for Performance Analysis," http://www.bitmover.com/lmbench/.

[4] "Microsoft Virtualization: Hyper-V," https://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx.

[5] "PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology." http://www.intel.eu/content/www/eu/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html.

[6] "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page.

[7] "SPCS001: Intel Next-Generation Haswell Microarchitecture," http://blog.scottlowe.org/2012/09/11/spcs001-intel-next-generation-haswell-microarchitecture/.

[8] "trace-cmd: A front-end for Ftrace," https://lwn.net/Articles/410200/.

[9] "Transparent huge pages in 2.6.38," http://lwn.net/Articles/423584/.

[10] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 2–13.

[11] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, "Software Techniques for Avoiding Hardware Virtualization Exits," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012, pp. 373–385.

[12] J. Ahn, S. Jin, and J. Huh, "Revisiting Hardware-assisted Page Walks for Virtualized Systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012, pp. 476–487.

[13] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, 2005, pp. 2–9.

[14] 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 the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[15] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don'T Walk (the Page Table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 48–59.

[16] ——, "SpecTLB: A Mechanism for Speculative Address Translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 307–318.

[17] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 237–248.

[18] A. Basu, M. D. Hill, and M. M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012, pp. 297–308.

[19] R. Bhargava, B. Serebrin, F. Spadini, and S. 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*, 2008, pp. 26–35.

[20] N. Bhatia, "Performance evaluation of Intel EPT hardware assist," *VMware, Inc*, 2009. [Online]. Available: http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf

[21] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 383–394.

[22] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[23] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 29–40.

[24] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[25] J. Buell, D. Hecht, J. Heo, K. Saladi, and R. Taheri, "Methodology for performance analysis of VMware vSphere under Tier-1 applications," *VMware Technical Journal*, vol. 2, no. 1, 2013.

[26] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 412–447, Nov. 1997.

[27] X. Chang, H. Franke, Y. Ge, T. Liu, K. Wang, J. Xenidis, F. Chen, and Y. Zhang, "Improving Virtualization in the Presence of Software Managed Translation Lookaside Buffers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 120–129.

[28] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 223–234.

[29] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee, "Reevaluating Online Superpage Promotion with Hardware Support," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[30] N. Ganapathy and C. Schimmel, "General Purpose Operating System Support for Multiple Page Sizes," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 1998.

[31] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "BadgerTrap: A Tool to Instrument x86-64 TLB Misses," *SIGARCH Comput. Archit. News*, vol. 42, no. 2, pp. 20–23, Sep. 2014.

[32] ——, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 178–189.

[33] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Range Translations for Fast Virtual Memory," *IEEE Micro*, May 2016.

[34] R. P. Goldberg, "Survey of Virtual Machine Research," *Computer*, vol. 7, no. 9, pp. 34–45, Sep. 1974.

[35] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[36] Intel Corporation, "Intel® 64 and IA-32 Architectures Optimization Reference Manual," April 2012.

[37] B. Jacob and T. Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE Micro*, vol. 18, no. 4, pp. 60–75, Jul. 1998.

[38] ——, "Uniprocessor Virtual Memory Without TLBs," *IEEE Trans. Comput.*, vol. 50, no. 5, pp. 482–499, May 2001.

[39] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002, pp. 195–206.

[40] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant Memory Mappings for Fast Access to Large Memories," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 66–78.

[41] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Energy-Efficient Address Translation," in *Proceedings of the 2016 IEEE 22nd International Symposium on High Performance Computer Architecture*, 2016.

[42] V. Karakostas, O. Ünsal, M. Nemirovsky, A. Cristal, and M. M. Swift, "Performance analysis of the memory management unit under scale-out workloads," in *IEEE International Symposium on Workload Characterization*, Oct 2014.

[43] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proceedings of the Linux Symposium*, vol. 1, Ottawa, Ontario, Canada, Jun. 2007, pp. 225–230.

[44] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 2:1–2:38, Apr. 2013.

[45] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, 2002, pp. 89–104.

[46] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, Feb 2015, pp. 210–222.

[47] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014.

[48] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 258–269.

[49] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 1–12.

[50] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB Preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 117–127.

[51] A. Sembrant, E. Hagersten, and D. Black-Shaffer, "Tlc: A tag-less cache for reducing dynamic first level cache energy," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 49–61.

[52] A. Seznec, "Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB," *IEEE Trans. Comput.*, vol. 53, no. 7, pp. 924–927, Jul. 2004.

[53] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[54] Sun Microsystems, "UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007."

[55] M. Swanson, L. Stoller, and J. Carter, "Increasing TLB Reach Using Superpages Backed by Shadow Memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998, pp. 204–213.

[56] M. Talluri and M. D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 171–182.

[57] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, 2002, pp. 181–194.

[58] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li, "Selective Hardware/Software Memory Virtualization," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2011.

[59] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton, "An In-cache Address Translation Mechanism," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 1986, pp. 358–365.

[60] H. Yoon and G. S. Sohi, "Revisiting Virtual L1 Caches: A Practical Design Using Dynamic Synonym Remapping," in *Proceedings of the 2016 IEEE 22nd International Symposium on High Performance Computer Architecture*, 2016.

[61] Y. Zhang, R. Oertel, and W. Rehm, "Paging Method Switching for QEMU-KVM Guest Machine," in *Proceedings of the 2014 International Conference on Big Data Science and Computing*, 2014, pp. 22:1–22:8.