# Efficient Memory Virtualization

Reducing Dimensionality of Nested Page Walks

Jayneel Gandhi† Arkaprava Basu‡ Mark D. Hill† Michael M. Swift†

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

‡AMD Research
Advanced Micro Devices, Inc.
Austin, TX, USA
arkaprava.basu@amd.com

***Abstract*— Virtualization provides value for many workloads, but its cost rises for workloads with poor memory access locality. This overhead comes from translation lookaside buffer (TLB) misses where the hardware performs a 2D page walk (up to 24 memory references on x86-64) rather than a native TLB miss (up to only 4 memory references). The first dimension translates guest virtual addresses to guest physical addresses, while the second translates guest physical addresses to host physical addresses. This paper proposes new hardware using direct segments with three new virtualized modes of operation that significantly speed-up virtualized address translation. Further, this paper proposes two novel techniques to address important limitations of original direct segments. First, self-ballooning reduces fragmentation in physical memory, and addresses the architectural input/output (I/O) gap in x86-64. Second, an escape filter provides alternate translations for exceptional pages within a direct segment (e.g., physical pages with permanent hard faults).**

**We emulate the proposed hardware and prototype the software in Linux with KVM on x86-64. One mode—VMM Direct—reduces address translation overhead to near-native without guest application or OS changes (2% slower than native on average), while a more aggressive mode—Dual Direct—on big-memory workloads performs better-than-native with near-zero translation overhead.**

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

## I. INTRODUCTION

Virtual machine monitors (VMMs)—such as KVM [33], Xen [6], and VMware [52]—provide an abstraction layer between operating systems (OSes) and hardware. The benefits of virtualization include resource management, server consolidation, security, and fault tolerance. Virtualization in the cloud provides the added benefit of on-demand access to hardware. To this end, cloud vendors like Amazon EC2 provide VMMs with up to 244GB memory and 32 cores [5].

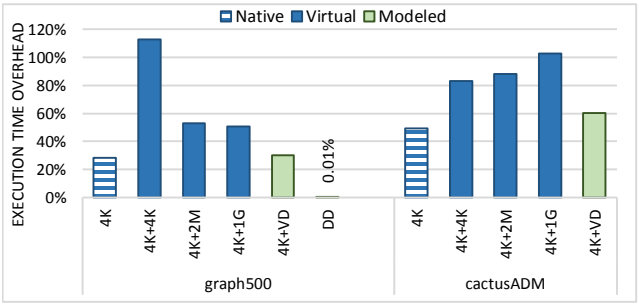


Figure 1 Overheads associated with virtual memory for selected workloads and configurations

*Motivation***:** Virtualization's benefits, however, come with overheads in processing, I/O, and memory. Fortunately, hardware advances in virtualization [11,40,41] (e.g., virtualizing I/O in hardware), have reduced these overheads substantially. However, overheads for virtualizing memory are not universally low. To quote a recent VMware paper:

*We will show that the increase in translation lookaside buffer (TLB) miss handling costs due to the hardware-assisted memory management unit (MMU) is the largest contributor to the performance gap between native and virtual servers. —Buell et al., 2013* [16]

Our results corroborate Buell *et al.* and show that virtualization degrades performance in workloads that use substantial memory. Figure 1 provides a preview of the overheads associated with virtual memory for the native case (bar: 4K) and the virtualized case with the guest OS using 4KB pages and the VMM using 4KB, 2MB and 1GB (bars: 4K+4K, 4K+2M, and 4K+1G, respectively). We observe that overheads increase drastically with virtualization and remain high even with larger VMM pages. This overhead makes virtualization less attractive for big-memory workloads that reference vast memory with poor locality, such as key-value stores and databases. Our proposed design (bars: DD and 4K+VD) mitigates these overheads.

The goal of our work is to make virtualization's benefits efficient for all workloads. To this end, we build on current x86-64 hardware support for memory virtualization [11,41] that logically performs two address translations on every memory reference:

**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-virtual machine nested page table (nPT).

Virtualized address translation performance depends critically on memory locality. In the best case, a TLB entry directly translates gVA to hPA with no overhead. In the worst case, a TLB miss performs a 2D page walk that "multiplies" overhead vis-à-vis native, because accesses to the gPT also require translation by the nPT. Figure 2 depicts how x86-64 page-table memory references can grow from a native 4 to a virtualized 24 references: 5 references to translate the root and each of 4 levels of the guest page table plus 4 references

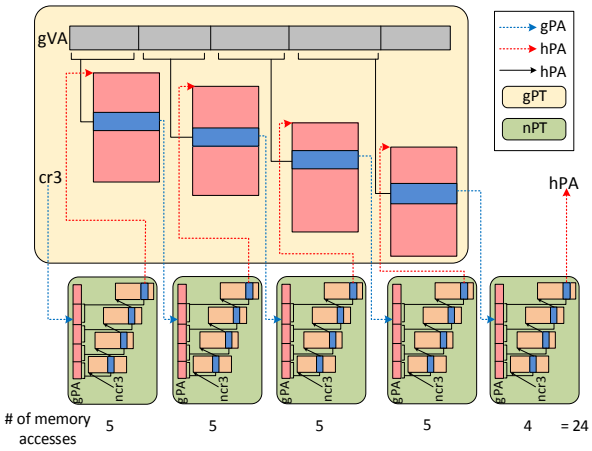


Figure 2 Concept of two-level nested page table walk state machine. The nested page tables are rotated by 90°

to obtain the final hPA: 5*4+4 references. Both levels of translation require a 4-level page table in x86-64, since each address space can potentially be as large as 256TB ($2^{48}$).

***Proposed Design*:** We propose new hardware that supports three new virtualized modes to lower the overheads of virtualized address translation. It extends direct segments [9], which were proposed to reduce TLB misses in unvirtualized (native) systems. As reviewed in Section II.B, a direct segment maps most of a process's linear virtual address space with a segment, while mapping the rest with page tables.

Figure 3 depicts the two base modes (native and virtualized), one unvirtualized direct segment mode (shaded) and three new virtualized modes (shaded). Large rectangular boxes represent an x86-64 page table walk, small squares represent direct segment accesses (used throughout the paper), and thick arrows depict the expected path of most translations. The left two modes show native (unvirtualized) 1D translation with and without direct segments. The right four modes are virtualized two-level translations with the existing 2D page walk (base virtualized) and our three new virtualized modes, representing different combinations of direct segments in the guest and nested address spaces. Note that Figure 3 shows nested translation linearly to simplify the illustration.

*Dual Direct* mode achieves almost zero translation overheads for big-memory workloads with modest changes to the VMM, guest OS, and applications. It uses direct segments to map directly from gVA to hPA, bypassing both dimensions

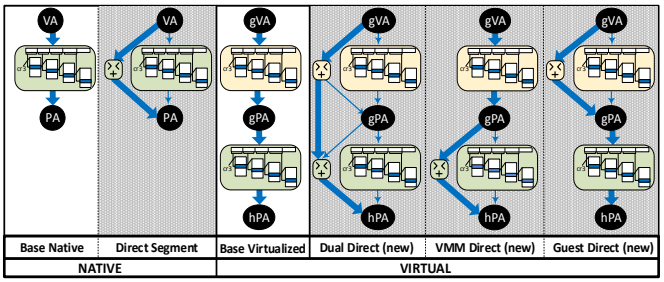


Figure 3 Native and virtualized address translation modes supported by proposed hardware

of virtualized address translation.

*VMM Direct* mode achieves near-native performance for arbitrary workloads with modifications confined to only the VMM. It uses a direct segment to map most of a guest's physical memory to host physical memory (2nd dimension: gPA→hPA).

*Guest Direct* mode provides near-native performance for big-memory workloads while using nested page tables in the VMM to facilitate services like live migration. It uses a direct segment to map most guest virtual addresses to guest physical addresses (1st dimension: gVA→gPA).

*Unvirtualized Direct Segment* mode provides identical behavior as the original direct segment proposal [9], but with less intrusive hardware.

To increase the flexibility of direct segments, we propose two novel techniques to address their limitations. First, a VM may lack contiguous physical memory needed to create a direct segment due to fragmentation. We address this with *self-ballooning*, which uses ballooning [52] and memory hotplug [38] to create contiguous physical guest memory from fragmented guest physical memory. Ballooning removes the fragmented memory from use and hotplug adds it back as contiguous guest physical memory that can be used to create a guest direct segment. We extend self-ballooning to further increase the amount of contiguous guest physical memory by relocating memory before the x86-64 I/O gap to contiguous memory at the end.

Second, we address the concern that a single faulty physical page can prevent the creation of a contiguous direct segment with an *escape filter* that allows holes in direct segments. The escape filter allows the OS to remap a few faulty pages within a direct segment through conventional paging. We find a 256-bit escape filter retains the performance gained by direct segments even in the presence of 16 faulty pages.

We emulate our proposed hardware and prototype our proposed software in the Linux® operating system with KVM/QEMU as the VMM on x86-64. We evaluate our designs with big-memory workloads (graph500, memcached, NPB:CG), the micro-benchmark GUPS, and compute workloads (SPEC® 2006 and PARSEC). VMM and Guest Direct modes reduce translation overheads (page walk times) to near-native, while Dual Direct mode makes overheads negligible.

This work is one of the first to study the memory overhead of virtualization for virtual machines with a large amount of memory. Our contributions are:

1. a quantitative analysis of address translation overheads for large virtualized workloads,
2. a new virtualized address translation design using direct segments with three new virtualized modes for reducing overhead to make virtualization more attractive,
3. a self-ballooning technique to increase contiguity of guest physical address, and
4. an escape filter to handle physical memory faults that may exist in a direct segment.

## II. BACKGROUND

### A. Virtualization and Memory Management

Virtualization has been used since the 1970s [25] to run multiple OSes on a single machine by introducing a layer of indirection between hardware and operating systems called a hypervisor or VMM. Virtualization's renaissance began with Disco in 1996 [17] and progressed without hardware support [1,6] (e.g., dynamic binary translation [1,3]) and, after gaining popularity, with hardware support [11,41].

Virtualization of memory management units followed the same software-to-hardware progression. The key software-only technique is shadow paging [52]: the VMM uses the guest page table (gVA→gPA) and nested page table (gPA→hPA) to build a new shadow page table (gVA→hPA). It points hardware to the shadow page table, so that TLB hits perform the translation (gVA→hPA) and TLB misses do a standard 1D page walk. However, changes to guest or host page tables incur substantial performance overhead to keep the shadow page table coherent [1]. All x86-64 processors now support the 2D page walk (gVA→gPA & gPA→hPA) in hardware [11,41], as depicted in Figure 2. Hence, the rest of the paper assumes this support. We compare against shadow paging in Section IX.D.

### B. Direct Segments

Direct segments use a form of segmentation along with paging to largely eliminate virtual memory overhead for big-memory workloads on unvirtualized (native) hardware [9]. A direct segment maps a portion of a process's linear address space with a segment rather than paging. Thus, a large chunk of a contiguous virtual address space can be mapped to contiguous physical addresses with only three registers per hardware context: BASE, LIMIT and OFFSET where BASE and LIMIT are the start and end of contiguous virtual address space and OFFSET is the difference between the virtual addresses and physical addresses of the direct segment. For compatibility, the rest of the linear address space is mapped using conventional paging. On a memory reference, the processor consults the segment registers and L1 TLB in parallel, with at most one match. A virtual address V within a direct segment (BASE ≤ V < LIMIT) gets translated to physical address V+OFFSET via simple addition avoiding the possibility of a TLB miss. Figure 4 illustrates an example address space mapping. Basu *et al.* showed that direct segments can eliminate 99% of address translation overhead for native big-memory applications.

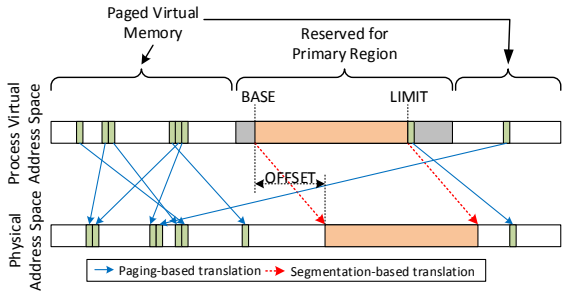


Figure 4 Address space layout using direct segment

To expose this hardware to programs, Basu *et al.* propose the *primary region* abstraction, which is a contiguous chunk of virtual address space that is mapped with the same access permissions. A primary region can be mapped fully or partially by a direct segment (shown in Figure 4).

## III. HARDWARE DESIGN AND SOFTWARE SUPPORT

The hardware proposed supports two levels of segment registers that can be controlled independently by the VMM and guest OS and used alone, together, or not at all. The hardware design we propose is shown in Figure 5. This hardware design supports *three new virtualized* modes, as was specified in Section I. At any point in time, each guest process (address space) uses one mode. We next explain the workings of each mode using the hardware proposed, along with the software support required for, each mode.

### A. Dual Direct mode

Dual Direct mode seeks a zero-D (0D) page walk. It uses two layers of direct segments: one, called the guest segment, for (most of) the first level of address translation (gVA→gPA) and the other, called the VMM segment, for (most of) the second-level of translation (gPA→hPA). Most L1 TLB misses are translated by segment registers and hence do not need to perform a page walk (a zero-D walk), which

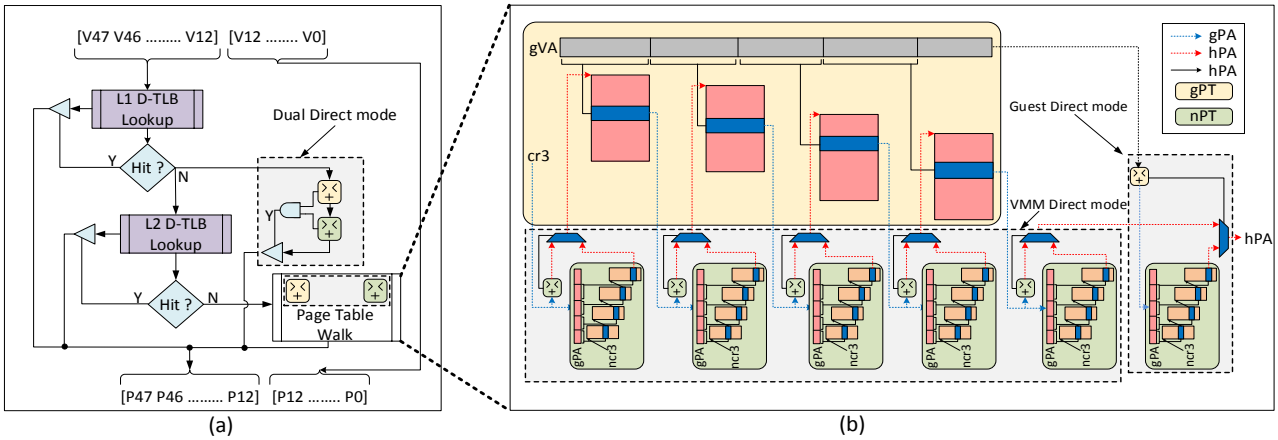


Figure 5 (a) Address translation flow chart (b) Steps of page walk state machine for various supported modes

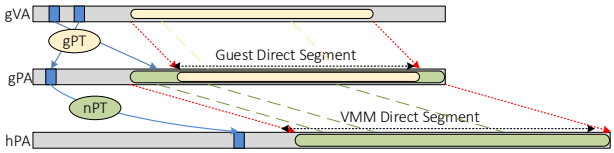


Figure 6 Memory layout for Dual Direct mode.

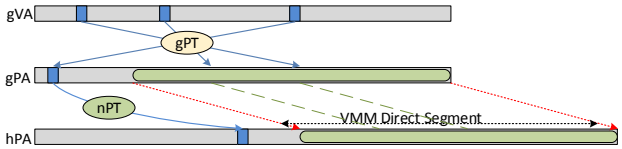


Figure 7 Memory layout for VMM Direct mode

achieves better-than-base-native execution. However, this mode suits big-memory applications with moderate guest OS and VMM changes.

Figure 6 shows the layout for gVA, gPA, and hPA address spaces with a guest segment in yellow and a VMM segment in green. The direct segments can be of different sizes, but here the guest segment is shown as a subset of the VMM segment.

*Hardware Operation*: Dual Direct performs the first level of address translation (gVA→gPA) with guest segment registers $BASE_G$, $LIMIT_G$ and $OFFSET_G$. Next, Dual Direct performs a second level of address translation (gPA→hPA) with segment registers $BASE_V$, $LIMIT_V$, and $OFFSET_V$. These registers are like those of unvirtualized direct segments (Section II.B), but for both levels of translation (see Figure 5 (a)).

A guest address can be in one of four categories in Dual Direct mode based on in which segment(s) it lies in. This is decided based on segment checks in hardware, $BASE_G \leq gVA < LIMIT_G$ and $BASE_V \leq gPA < LIMIT_V$. TABLE I shows the translation steps in each of the categories. In the best case, the guest address lies in both segments (Case: "Both") for which a L1 TLB miss will have a 0D page walk.

On VM-exit/entry, hardware must save/restore registers $BASE_V$, $LIMIT_V$ and $OFFSET_V$ along with other VM state.

*Software Support*: For Dual Direct mode, both the VMM and guest segments need modest software support. The VMM requires two key changes: (a) it must request contiguous host physical memory, and (b) it must set/clear segment registers when switching between guest VMs. The value of the $BASE_V$, $LIMIT_V$, and $OFFSET_V$, registers are the start and end of the guest physical contiguous memory region, and the difference between the guest and host physical addresses of the region.

Support for guest segments for the first level of translation (gVA→gPA) follows unvirtualized direct segments (Section II.B), (i.e., suiting big-memory applications with a primary region and OS support for allocating contiguous physical addresses). The guest segment register values are set per guest process and must be set during guest OS context switches. With Dual Direct, however, the OS becomes the guest OS and the physical address range is a gPA range. Like direct segments, restrictions imposed on primary regions make this less useful for compute workloads.

### B. *VMM Direct mode*

VMM Direct mode seeks a 1D (faster than 2D) page walk with no application or guest OS changes. It uses paging for the first level of address translation (gVA→gPA) and a direct segment for (most of) the second (gPA→hPA). This allows applications to use a dense or sparse gVA space, and leaves the guest OS unchanged with standard paging. Figure 7 shows the memory layout for gVA, gPA, and hPA address spaces with a VMM segment in green.

*Hardware Operation:* For VMM Direct mode, $BASE_G$ is set equal to $LIMIT_G$ to nullify the effect of the dashed boxes labeled Dual Direct and Guest Direct mode in Figure 5.

A guest address can be in one of two categories based on whether the guest address is in the VMM segment or not. The translation in each case is covered by the "VMM segment only" and "Neither" respectively from TABLE I.

The key advantage of VMM Direct mode is that TLB misses take only up to 4 memory accesses and 5 calculations, because guest page-table accesses (gPA) are translated with the direct segments rather than the nPT.

On VM-exit/entry, hardware must save/restore registers $BASE_V$, $LIMIT_V$ and $OFFSET_V$ along with other VM state.

*Software Support:* VMM Direct mode requires no application or guest OS changes, but only VMM support. VMM Direct requires only the two key VMM changes that were described for Dual Direct mode. With small guest OS changes, better performance can be achieved.

The x86-64 architecture has a gap in the physical address

TABLE I STEPS IN ADDRESS TRANSLATION OF A GUEST VIRTUAL ADDRESS IN DUAL DIRECT MODE.

| **Steps of Translation** | **Guest Virtual Address in Guest Segment or VMM Segment?** | | | |
|---|---|---|---|---|
| | **Both** | **VMM segment only** | **Guest segment only** | **Neither** |
| **L1 TLB Hit** | Translation complete | Translation complete | Translation complete | Translation complete |
| **L1 TLB Miss** | $hPA=gVA+OFFSET_G+OFFSET_V$ Insert L1 TLB entry | L2 TLB lookup | L2 TLB lookup | L2 TLB lookup |
| **L2 TLB Hit** | — | Insert L1 TLB entry | Insert L1 TLB entry | Insert L1 TLB entry |
| **L2 TLB Miss** | — | Invoke PTW | Invoke PTW | Invoke PTW |
| **PTW: gVA→gPA** | — | Walk guest OS page table | $gPA=gVA+OFFSET_V$ | Walk guest page table |
| **PTW: gPA→hPA** | — | For each gPA, $hPA=gPA+OFFSET_V$ or walk nested page table | Walk nested page table | For each gPA, walk nested page table |
| **PTW: End** | — | Insert L1 TLB entry | Insert L1 TLB entry | Insert L1 TLB entry |

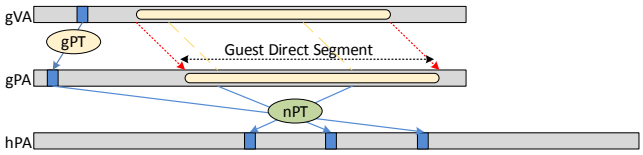

Figure 8 Memory layout for Guest Direct mode

space between 3-4GB for memory-mapped I/O [29]. In Section IV, we describe how a simple OS extension can relocate memory before the gap to create a larger contiguous region above the gap. Moreover, the guest OS must allocate page tables within the VMM direct segment, which can be achieved with a guest kernel module like VMware Tools.

### C. Guest Direct mode

Guest Direct mode seeks a 1D page walk while supporting features like page sharing and live migration that depend on 4KB nested pages. It uses a guest segment for (most of) the first level of translation (gVA→gPA) and nested paging for the second (gPA→hPA). This mode retains nested page table in the VMM while providing near-base-native speeds. It suits big-memory workloads along with small guest-OS changes. Figure 8 shows the memory layout for gVA, gPA, and hPA address spaces with a guest segment in yellow.

*Hardware Operation:* For Guest Direct mode, $BASE_V$ is set equal to $LIMIT_V$ thus nullifying the effect of dashed box labeled Dual Direct mode and VMM Direct mode in Figure 5. A guest address can be in one of two categories based on whether the guest address is in a Guest segment or not. These are covered by the cases "Guest segment only" and "Neither" respectively from TABLE I.

The key advantage of Guest Direct mode is that TLB misses make 4 memory accesses and 1 calculation, which costs closer to a native 1D page walk (4 accesses and 0 calculations) and much less than the 24-access 2D page walk.

On every guest OS context switch, hardware must save and restore $BASE_G$, $LIMIT_G$, and $OFFSET_G$, along with other guest process state.

*Software Support:* Guest Direct mode does not require any changes to the VMM, but requires the guest OS support proposed for Dual Direct mode.

### D. Unvirtualized Direct Segment mode

Direct segment mode operates just like the original direct segments [9] but with less intrusive hardware. The key advantage of Direct Segment mode is that TLB misses make only 1 calculation, as compared to a native 1D page walk (up to 4 accesses).

This mode is supported by using only guest segment registers to translate from VA→PA in parallel with the L2 TLB and introduces a L1 TLB entry. This hardware support is less intrusive than the originally proposed design [9], as it performs the segment calculation in parallel with the L2 TLB lookup instead of L1 TLB lookup, where it could affect pipeline timing. The software support remains the same.

### E. Summary

The new hardware for virtualized direct segments logically resides in two places. On the L1 TLB miss path, Dual Direct-mode hardware consults segment registers to bypass the page walk entirely. We also modify the page-walk hardware to flatten one or two dimensions of the walk based on the mode. The hardware along with software support allows switching between modes dynamically during execution.

TABLE II TRADE-OFFS IN VARIOUS VIRTUALIZED DESIGNS.

| Properties | Base Virtualized | Dual Direct | VMM Direct | Guest Direct |
|---|---|---|---|---|
| Page walk dimensions | 2D | 0D | 1D | 1D |
| # of memory accesses for most page walks | 24 | 0 | 4 | 4 |
| # of base-bound checks for page walks | 0 | 1 | 5 | 1 |
| Guest OS modifications | none | required | none | required |
| VMM modifications | none | required | required | none |
| Application category | any | big memory | any | big memory |
| Page sharing | unrestricted | limited | limited | unrestricted |
| Ballooning | unrestricted | limited | limited | unrestricted |
| Guest swapping | unrestricted | limited | unrestricted | limited |
| VMM swapping | unrestricted | limited | limited | unrestricted |

TABLE II summarizes the tradeoffs among the modes. The modes can achieve faster page walks at the cost of additional changes or restrictions. Dual Direct provides only limited support for memory overcommit, as it uses direct segments at both address translation levels. VMM Direct supports guest swapping whereas Guest Direct also supports page sharing, ballooning, and VMM swapping. All techniques can always be used for memory outside direct segments with all three modes.

## IV. REDUCING MEMORY FRAGMENTATION

Guest and host physical memory fragmentation can prevent creation of direct segments at one or both levels. Moreover, the x86-64 architecture fragments the physical memory for memory-mapped I/O. We discuss few ways to reduce memory fragmentation to enable creation of direct segments.

*Self-ballooning:* To handle guest physical memory fragmentation and facilitate quick creation of guest segments, we propose a novel software optimization: self-ballooning.

The goal of this technique is to provide contiguous guest physical memory quickly from fragmented free guest physical memory without the cost of memory compaction. Self-ballooning applies to Guest Direct mode and creates contiguous memory in two steps:

First, our balloon driver runs in the guest OS, and like a standard balloon driver [52], asks the guest OS for a set of pages that can be reclaimed by the VMM. The balloon driver pins and reserves this memory so it cannot be used by guest applications nor swapped out.

Second, the balloon driver passes these pages to the VMM, which uses memory hotplug to add the same amount of memory back to the VM. Memory hotplug [38] is designed for hot swapping or powering off memory chips. The

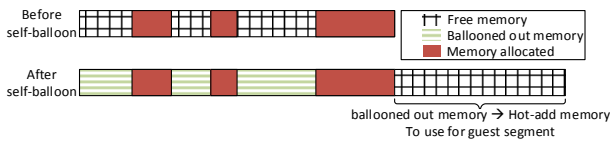


Figure 9 Illustration of guest physical memory with self-ballooning.

newly added contiguous guest physical memory can now serve as the guest segment in the VM. Thus, the VMM can create contiguous guest physical memory from an assortment of pages. Figure 9 illustrates self-ballooning.

While designed for Guest Direct segments, self-ballooning can also work with standard nested page tables to create more large pages in a guest OS.

*Reclaiming I/O gap memory:* A second source of fragmentation is architectural. The x86-64 architecture has an *I/O gap*, which is a ~1GB region at the high-end of the 32-bit (4GB) physical address space reserved for memory-mapped I/O [29]. This gap splits the addresses backed by physical memory to about 3GB before the I/O gap and the rest after, and prevents a single direct segment from mapping all guest physical memory. Normally, the chipset remaps contiguous physical memory to introduce the I/O gap.

We mitigate this effect with a variation of self-ballooning using hot-unplug instead of ballooning to remove the guest physical memory. We unplug most guest physical memory before the I/O gap and extend memory by the same amount. We use hot-unplug instead of ballooning because it supports removing specific addresses (those before the I/O gap), rather than an arbitrary set chosen by the kernel. Moving the memory allows a single direct segment to map almost all guest physical memory. More details on implementation can be found in Section VI.C. This technique is effective for both VMM Direct and Dual Direct.

*Memory compaction:* To address host physical memory fragmentation, we leverage the slower technique of memory compaction which slowly relocates pages and creates a VMM segment. Compaction is supported in many OSes including Linux® [20]. Dual Direct and VMM Direct modes can start without a VMM segment in Guest Direct mode and Base Virtualized mode respectively. Once the compaction daemon provides enough contiguous physical memory to create a VMM segment, the VMM can create a VMM segment achieving higher performance through Dual Direct.

*Summary:* TABLE III summarizes the modes used in fragmented systems.

## V. ESCAPE FILTER

Commodity OSes commonly put faulty pages on a bad-page list to prevent their use [26]. However, with direct segments, a *single* bad page can prevent creation of a large direct segment. We introduce a novel hardware technique, an *escape filter* that allows holes in direct segment. If an address is in a hole, it "escapes" segment-based translation to use conventional paging. Hence, the VMM or OS can still use paging to remap escaped pages to functioning memory.

We implement the escape filter using a hardware Bloom filter that is checked in parallel with the VMM's segment registers (in Dual or VMM direct modes) and the guest segment registers (in Direct Segment mode). In Guest Direct mode, the VMM still uses nested pages tables and can remap faulty pages. The VMM (or OS for Direct Segment mode) adds escaped pages to the Bloom filter and creates PTEs to map those pages. As the Bloom filter may have false positives (non-escaped pages that are falsely considered escaped), the VMM must create mappings for these pages as well. With an escape filter, an address is translated with direct segments if it lies within the direct segment, *but not* the filter. The filter is part of the context state and must be saved/restored with segment registers. However, it can be small: to tolerate 16 faulty pages, we show that a 256-bit filter has almost zero overhead from false positives.

The escape filter can also implement a limited number of pages with different protection, such as guard pages. For such uses, it may be useful to have escape filters at both levels of translation so the guest OS can escape pages as well.

## VI. PROTOTYPE IMPLEMENTATION

We design our prototype system for the Linux® operating system (x86-64) with LTS kernel v3.12.13 using QEMU v1.4.1 with KVM as the VMM. Our implementation has three pieces: (1) allocating contiguous physical memory, (2) emulating segmentation behavior, and (3) prototyping self-ballooning.

### A. Contiguous Allocation

To allocate contiguous physical memory, the OS reserves memory immediately after startup. We target machines running a stable set of long-lived virtual machines allowing us to use the mechanism naturally. The size of the virtual machine and memory requirements of big-memory applications [9] are often known *a priori* through their configuration parameters and allow us to reserve the required memory.

### B. Emulating Segments

The hardware design described in Section III requires

TABLE III VARIOUS MODES UTILIZED IN FRAGMENTED SYSTEMS.

| Applications | VM State | Modes utilized |
|---|---|---|
| Big-memory workloads | Host physical memory fragmented | Guest Direct mode slowly converted to Dual Direct mode with host memory compaction |
| | Guest physical memory fragmented | Dual Direct mode with self-balloon support |
| | Host+Guest physical memory fragmented | Guest Direct mode with self-balloon support slowly converted to Dual Direct mode with host memory compaction |
| Compute workloads | Host physical memory fragmented | Base Virtualized mode slowly converted to VMM Direct mode with host memory compaction |
| | Guest physical memory fragmented | VMM Direct mode |
| | Host+Guest physical memory fragmented | Base Virtualized mode slowly converted to VMM Direct mode with host memory compaction |

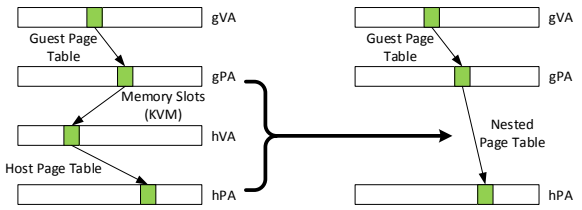

Figure 10 KVM memory slots to nested page tables.

new hardware support. To evaluate this design on current hardware, we follow a previously proposed technique [9] and emulate direct-segment functionality by mapping segments with 4KB pages. We modify the Linux page fault handler both in the VMM and guest OS. These changes identify page faults to direct segments, and compute physical addresses using segment offset. These computed addresses are added to the respective page tables by the fault handler. Thus, direct segments are mapped using dynamically computed PTEs.

This approach provides a functionally correct implementation of our designs on current hardware. However, it does not provide any performance improvement without new hardware. As described in Section VII, we count events to predict performance with new hardware.

### C. Self-Ballooning Prototype

We implemented self-ballooning in KVM. Each guest VM has an associated KVM process, which runs as a user-space process on the host OS. The guest physical addresses of a VM are mapped on to the host virtual addresses of the KVM process, and the host Linux maps host virtual addresses of the KVM process to host physical addresses. Figure 10 shows a typical address space mapping in x86-64 using KVM. The KVM kernel component, called the *KVM module*, creates nested page tables by computing combined gPA→hVA→hPA translations. The mapping gPA→hVA is handled through *memory slots*. A memory slot is a contiguous range of guest physical addresses that are mapped to contiguous virtual memory in the KVM process. There are only two large slots in KVM: one between 0-4GB, and another for 4GB and beyond.

*Fragmented memory:* We modify QEMU-KVM [33] and the virtio balloon driver [35] to prototype self-ballooning with the KVM hypervisor. KVM currently does not support hot-adding memory to guest OSes. Instead, we extend the second KVM slot by the largest amount of memory (96GB for our machine) that can be used for self-ballooning when required. This extra guest physical memory is ballooned out during startup and cannot be used by the guest OS.

The guest OS invokes the modified balloon driver when it cannot create a guest segment due to fragmentation. The driver removes the required amount of memory and provides that to the VMM. The VMM in return informs the driver to release the memory from the reserved portion of guest physical memory. The guest OS can now create a guest segment from the newly released guest physical memory.

*I/O gap*: We modify the guest OS to remove as much memory as possible from the first KVM slot using hotplug [38]. Using hotplug is similar to ballooning, and causes the guest OS to ignore the removed addresses. We extend the second KVM slot by the same amount of memory. Our experiments show that 256MB is enough to boot Linux correctly and the rest (3.1 GB) can be removed from the first KVM slot. This gives us a long address range in the gPA starting at 4GB that can be mapped using a single segment to hPA, and a small 256MB range for the kernel mapped using pages.

*Memory compaction*: We use the memory compaction daemon present in Linux [20] to aggressively perform compaction when required to create a direct segment.

## VII. Evaluation Methodology

We evaluate the proposed hardware using VMM and kernel modifications and hardware performance counters since workload size and duration makes full-system simulation less appropriate. We use the counters to measure the number of TLB misses in native ($M_n$) and virtual ($M_v$) environments, and the page walk cycles spent on TLB misses. This provides us with page walk cycles spent per TLB miss ($C_n$ and $C_v$, respectively) for each program. We modify the guest OS kernel to capture all TLB misses using BadgerTrap [24], a tool that instruments all DTLB misses, as these DTLB misses benefit from our proposed hardware.

We instrument the guest OS to extract gVA and gPA for a DTLB miss to determine if the address lies in a VMM or guest segment. We classify the miss depending on the mode used to calculate DTLB misses affected by a direct segment.

*Native/Virtualized baseline:* We run the workloads (Table V) to completion on native hardware and in a virtual machine described in Table VI and use Linux® *perf* [34] to collect the performance counter data. By fixing the amount of work done by the programs, we compare cycles across different configurations. For each proposed mode, we developed linear models to predict its performance (Table IV).

*Direct Segment:* To compare with native direct segments, we developed a model to determine the unvirtualized performance. We find the fraction of TLB misses ($F_{DS}$) that lie in the direct segment [9], which would be eliminated.

*VMM Direct/Guest Direct:* We determine the fraction of TLB misses that lie in the respective direct segment ($F_{VD}$ or $F_{GD}$). This fraction of TLB misses would spend near-native cycles per TLB miss. We estimate the cycles per TLB miss of this fraction as ($C_n+\Delta$) where $\Delta$ represent the overhead of performing base-bounds check in addition to cycles per TLB miss on native hardware ($C_n$). As an estimate, we use 1 cycle per *base-bound check,* thus $\Delta_{VD}=5$ for VMM Direct and $\Delta_{GD}=1$ for Guest Direct modes. The rest of the TLB misses would suffer $C_v$ cycles per TLB miss due to 2D page walk.

Table IV Linear Model for Cycles Spent on Page Walk.

| Design | Model |
|---|---|
| Direct Segment | $C_n*(1-F_{DS}) * M_n$ |
| Dual Direct | $[(C_n+\Delta_{VD})*F_{VD} + (C_n+\Delta_{GD})*F_{GD} + C_v*(1-F_{GD}- F_{VD}-F_{DD})]*M_n$ |
| VMM Direct | $[(C_n+\Delta_{VD})*F_{VD} + C_v*(1-F_{VD})]*M_n$ |
| Guest Direct | $[(C_n+\Delta_{GD})*F_{GD} + C_v*(1-F_{GD})]*M_n$ |

TABLE V WORKLOAD DESCRIPTION.

| Workload | Description |
|---|---|
| **Graph500** | Generation, compression and breadth-first search (BFS) of very large graphs, as often used in social networking analytics and HPC computing. |
| **Memcached** | In-memory key-value cache widely used by large web-sites, for low-latency data retrieval. |
| **NPB:CG** | NASA's high performance parallel benchmark suite. CG workload from the suite. |
| **GUPS** | Random access benchmark defined by the High Performance Computing Challenge. |
| **SPEC® 2006** | Compute single-threaded workloads: cactusADM, GemsFDTD, mcf, omnetpp (ref inputs) |
| **PARSEC3.0** | Compute multi-threaded workloads: canneal, streamcluster (native input set) |

TABLE VI DETAILS OF THE NATIVE AND VIRTUALIZED SYSTEMS.

| **Native System** | |
|---|---|
| Processor | Dual-socket Intel Xeon E5-2430 (SandyBridge) 6 cores/socket, 2 threads/core, 2.2 GHz |
| Physical Memory | 96 GB DDR3 1066MHz |
| Operating System | Fedora-20 (Linux LTS Kernel v3.12.13) |
| L1 Data TLB | 4KB: 64 entries 4-way associative<br>2MB: 32 entries 4-way associative<br>1GB: 4-entry fully associative |
| L2 TLB | 4KB: 512 entries 4-way associative |
| **Fully-Virtualized System with KVM** | |
| VMM | QEMU (with KVM) v1.4.1, 24vCPUs |
| Physical Memory | 85GB |
| Operating System | Fedora-20 (Linux LTS Kernel v3.12.13) |
| EPT TLB/NTLB | Shares the TLB (no separate structure) |

**Dual Direct:** We split the TLB misses in 4 ways according to TABLE I:

1. The fraction of misses that lie in both direct segments ($F_{DD}$). These page walks are eliminated by Dual Direct.
2. The fraction of misses that lie only in VMM segment and not in guest segment ($F_{VD}$). These misses are sped up by VMM Direct mode.
3. The fraction of misses that lie only in guest segment and not in VMM direct segment ($F_{GD}$). These misses are sped up by Guest Direct mode.
4. The rest of the TLB misses suffer $C_v$ cycles per TLB miss for the 2D page walk.

The cycles per TLB miss for Dual Direct Design can be calculated by adding the above four categories. Since we cannot measure cycles spent accessing L1 and L2 TLBs, our model does not account for improvements due to faster L2 hits when using the Dual Direct design or pollution of the L2 TLB from pages in a direct segment.

## VIII. COST OF VIRTUALIZATION

We show that the cost of virtualization can be very high and corroborate some of the findings of Buell *et al.* [16]. Similar earlier studies used much smaller virtual machines running desktop applications [4,14,41]. Our study quantifies how applications behave in large virtualized environments.

Our experimental setup is as follows:

- We use several multithreaded big-memory workloads, a GUPS micro-benchmark, and some SPEC2006 and PARSEC work-loads (TABLE V). The big-memory workloads were executed with 60-75GB datasets on virtual machines with 80GB of guest physical memory.
- We run the workloads on an x86-64 system (TABLE VI), both natively and in a Linux KVM virtual machine.
- We examine four native configurations. For big memory workloads, it is straightforward to have big-memory applications explicitly request 4KB, 2MB, or 1GB pages. Since SPEC and PARSEC are less suited to these changes, we use 4KB pages and enable transparent huge pages (THP) [19], which seeks to dynamically promote aligned groups of 512 4KB pages to a 2MB page.
- We examine eight virtualized configurations that vary both guest and VMM page sizes (e.g., 4K+2M means the guest uses 4KB pages and the VMM uses 2MB pages).

Figure 11 and Figure 12 present execution time overheads for address translation for base-native (hashed bars) and virtualized (solid). If an execution E runs in time $T_E$, we calculate its address-translation overhead as $(T_E - T_{2Mideal})/T_{2Mideal}$, where $T_{2Mideal}$ is the same benchmark's native execution time with 2MB pages minus the time the 2MB run spends in page table walks. TLB misses may be overlapped with other processor stalls, so for workloads with very high miss rates, subtracting page walk time from total execution time may be inaccurate. We try to minimize this effect by using as our base execution time, the 2MB results ($T_{2Mideal}$), but cannot remove it completely. Note that execution times include the effects of all TLBs and page walk caches. The GUPS micro-benchmark uses the scaled right-hand y-axis and is shaded separately.

We make the following observations:

1. *Native address translation overheads with 4KB pages can be high and grow drastically when virtualized.* [e.g., the overheads for graph500 goes up from 28% in native to 113% under virtualization (bar 4K vs 4K+4K)]. The geometric mean increase with virtualization is ~3.6x.
2. *Virtualization overheads can be reduced by using 2MB pages to map gPA to hPA at the VMM. However, overheads can still be large. Even with 2MB pages at both levels of translation, overheads are substantially higher than native execution with 2MB pages* (e.g., bar 4K+2M: 53% vs bar 4K+4K: 113% and bar 2M: 6% vs. bar 2M+2M: 13% for graph500).
3. *Even with 1GB pages at guest OS and VMM, the overhead does not reduce greatly as compared to 2MB pages and are higher than native execution with 1GB pages. In addition, using 1GB pages in the VMM can also hurt performance due to limited 1GB TLB entries* (e.g., for graph500, bar 1G: 3% vs. bar 1G+1G: 11% and 13% overhead with bar 2M+2M and 14% with bar 2M+1G).
4. *Similar trends are observed in compute workloads.* CactusADM and mcf have high overheads even with transparent huge pages (THP).

These observations demonstrate that many workloads suffer substantial virtualization overheads, rendering virtualization less attractive today. Moreover, we conjecture that overheads will get considerably worse with larger datasets and virtual machines [9]. This analysis corroborates a recently published study from VMware[16].

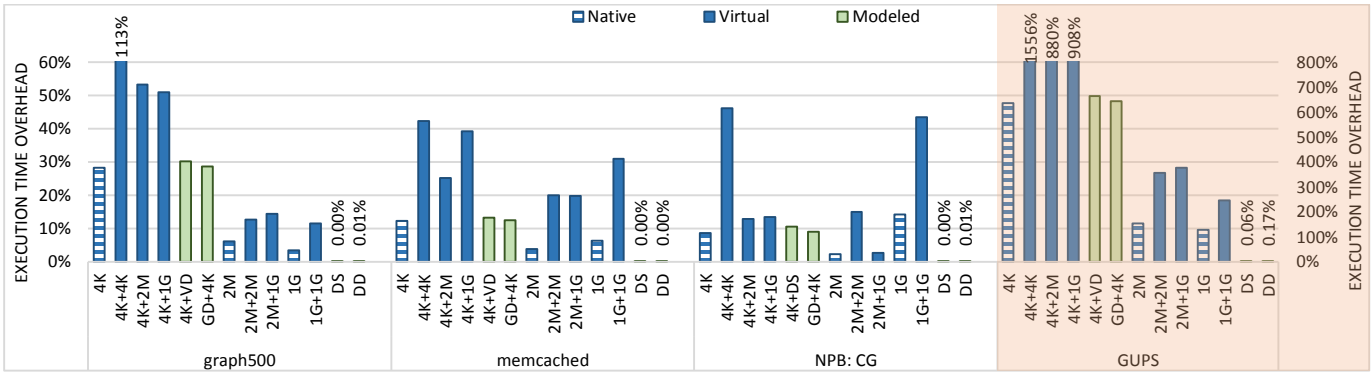


Figure 11 Virtual memory overhead for each configuration per big-memory workload.

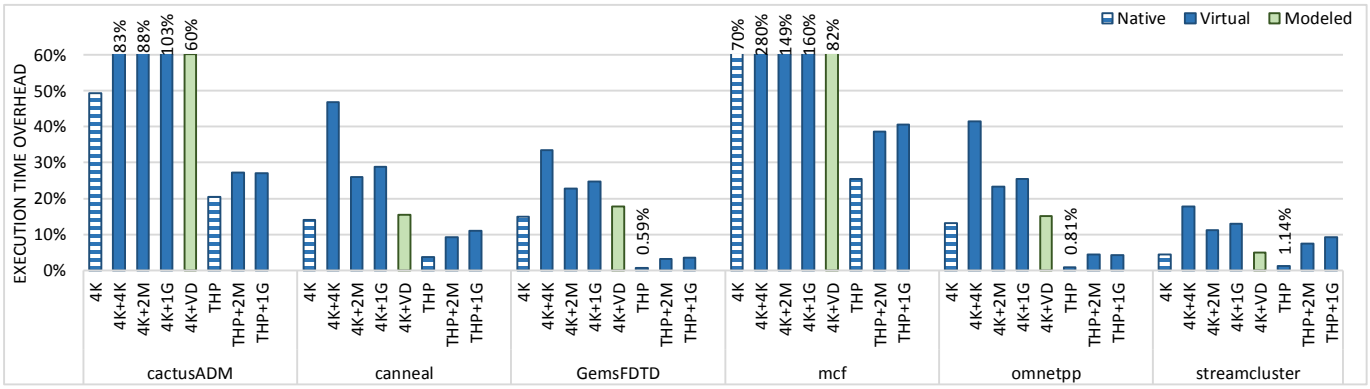


Figure 12 Virtual memory overhead for each configuration per compute workload.

## IX. Evaluation of New Design

In this section we discuss the various benefits of our proposed hardware with its various modes of operation.

### A. Performance Analysis

*Performance benefits*: Figure 11 and Figure 12 depict execution time overheads for address translation for our new designs in green. Recall that our models are pessimistic due to our assumption of flat $\Delta_{VD} = 5$ cycles and $\Delta_{GD} = 1$ based on the number of base-and-bound calculations. This overhead will be reduced by techniques like translation caching [7] and shared MMU caches [12] that store intermediate translations by reducing the computations required.

From Figure 11 and Figure 12, we conclude:

1. *For our big-memory and compute workloads, VMM Direct achieves overheads close to native execution.* [e.g., graph500 suffers 30% (bar 4K+VD) overhead, close to the native overhead of 28% (bar 4K). Overall, VMM Direct is only 2% slower than native execution (geo mean)].
2. Similarly, *Guest Direct is able to achieve native performance for big-memory workloads* (bar 4K+GD).
3. *Dual Direct achieves negligible address translation overheads for big-memory workloads, similar to unvirtualized direct segments*. Dual Direct mode reduces address translation overheads to at most 0.17% (bar DD).

The performance benefits of VMM Direct are further enhanced by using 2MB (bar 2M+VD) or 1GB pages (bar 1G+VD) or THP to translate from gVA to gPA. We do not evaluate these due to lack of support for large pages in our prototype. Guest Direct can also be enhanced similarly.

*Performance Breakdown:* We analyzed two key factors affecting the translation overheads: 1) number of TLB misses and 2) avg. cycles spent per TLB miss.

We make two observations about execution under virtualization vs. executing natively. First, we expect the TLB misses for a given application to remain the same across execution under virtualization and native execution. However, *we found that virtualization can lead to a significant increase in the number of TLB misses*. For example, TLB misses 4K+4K increased by 1.38x for graph500, 1.62x for memcached, 1.41x for GUPS, 1.33x for canneal, and 1.29x for streamcluster. The extra misses occur because nested TLB entries (gPA→hPA) share the same physical structure as the normal TLB entries, reducing the effective capacity of the TLB. We confirmed the behavior with a micro-benchmark. Second, *as expected for all workloads, the average cycles per TLB miss grows significantly with virtualization due to 2D page walks*. For example, it can be as high as 3.5x for NPB:CG with 4K+4K. On average, cycles-per-miss increase 2.4x, 1.5x, 1.6x for 4K+4K, 4K+2M, and 4K+1G, respectively.

With our proposed hardware, we find that VMM Direct and Guest Direct achieve cycles per TLB miss close to native execution. A TLB miss costs only 13% higher (on average) with VMM direct and 3% higher (on average) with Guest Direct compared to native 4K. In contrast, Dual Direct gets most of its benefits from reduction in L2 TLB misses (~99.9% reduction in L2 TLB misses).

### B. Energy Discussion

Alternate address translation modes affect system energy in two ways. Most importantly, if the mechanism reduces execution time by some percentage X, it can reduce whole-system static energy by about X%. For example, Dual Direct reduces execution time by 11-89% compared to 4K+2M pages (Figure 11 and Figure 12) across our benchmarks and thus reduces system static energy by a similar fraction.

Second, the translation mechanism itself uses energy, (e.g., 20-38% of L1 cache dynamic energy [10] and a smaller fraction of whole system energy). Page-based address translation uses dynamic energy to: (a) access the L1 TLB, (b) on an L1 TLB miss, accesses the L2 TLB, and (c) on an L2 TLB miss, accesses the page walker and MMU cache.

Our new virtualized design: (a) leaves the L1 TLB access unchanged; (b) on an L1 TLB miss, accesses the L2 TLB as well as small virtualized direct-segment hardware; and (c) on a L2 TLB miss and on a miss in the both direct segments, accesses the modified page walker and MMU cache. We expect the reduction in term (c) dominates the small cost increase to term (b), thus potentially reducing address translation dynamic energy as compared to virtualized baseline.

The original direct segment design uses energy to: (a) access the L1 TLB as well as small direct-segment hardware, (b) on a L1 TLB miss and on a miss in the direct-segment, accesses the L2 TLB, and (c) on L2 TLB miss, accesses the page walk hardware and MMU cache. If the reduction in the term (b) dominates the small cost increase to term (a), then the original direct segment hardware further improves on address translation dynamic energy compared to the new virtualized design.

If this improvement is important, our design can be modified to perform the base-bound check for Dual Direct with its two comparators in parallel with L1 TLB lookup. We do not advocated this design, because it affects the timing critical L1 TLB hit path (a drawback of the original design).

### C. Escape Filter

Our proposed escape filter enables a few pages within a direct segment to escape to page-based translation. We use this mechanism to retain most of a direct segment's performance benefits even when 1-16 pages have hard faults.

We studied the impact of using a 256-bit hardware parallel bloom filter with four H3 hash functions [44]. For each number of bad pages (1-16), we ran each application with 30 different random sets of bad pages. Figure 13 depicts execution time overhead (compared to Dual Direct mode with no bad pages), as well as 95% confidence intervals.

*Dual Direct mode retains almost all its performance benefits even with some hard faults.* With a pessimistic 16 faults, execution impact is less than 0.06% (except micro-benchmark GUPS 0.5%). We observe similar trends with compute workloads running in VMM Direct mode.

### D. Shadow Paging: An Alternative

Shadow paging offers a way to eliminate 2D page walks, but we observe that it works well for only some of our workloads, while our new design works for all. Recall that with shadow paging (Section II.A), the VMM uses the guest page


Figure 13 Normalized execution time for big-memory workloads in presence of bad pages

table (gVA→gPA) and nested page table (gPA→hPA) to build a shadow page table (gVA→hPA) walked by the hardware, and changes to guest or host page tables incur substantial performance overheads.

We use shadow paging by disabling extended page tables in KVM. We measure the slowdown in execution time compared to native execution for shadow paging using 4KB and 2MB pages (both guest and VMM).

We observe that our workloads fall in two categories:

1. Workloads for which shadow paging incurs high virtualization overheads: memcached (4K: 29.2%, 2M: 11.1%), GemsFDTD (4K: 12.2%, 2M: 4.9%), omnetpp (4K: 8.7%, 2M: 3.4%), and canneal (4K: 6.63%, 2M: 2.5%). The increase in execution time corresponds to extra VMexits to keep shadow page tables coherent.
2. Workloads for which shadow paging incurs relatively low overheads. For all other workloads, we observed slow-down of less than 5% for both page sizes.

Shadow paging does well due to the static nature of memory allocation in workloads from the second category. For workloads with frequent memory allocations and deallocations (first category), shadow paging provides poor performance due to frequent updates to guest page tables [53].

In contrast, VMM Direct allows page table updates to proceed without any VMM intervention. Thus, our techniques provide near-native performance for both sets of workloads. Shadow paging can be up to 29.2% slower, whereas VMM Direct is only up to 7.3% slower than the native execution. With small guest OS and application changes, Dual Direct provides much lower overhead than shadow paging.

### E. Content-Based Page Sharing

Content-based page sharing saves memory for compute workloads, but we observe that it provides less benefit for our big-memory workloads. Content-based page sharing scans memory to find pages with identical contents. When such pages are found, the VMM can reclaim all but one copy and maps the others using copy-on-write [52].

We studied the impact of content-based page sharing, since VMM segments preclude page sharing. We co-scheduled two smaller instances (40 GB) of KVM, each running one of our big-memory workloads (all possible pairs) to measure the potential memory saving from page sharing.

We observed that page sharing does not save more than 3% memory for our big-memory workloads since the bulk of

memory is for data structures unique to the workload. These include *OS code pages that can be easily shared with our modes, as they are mapped with pages*. Thus, restricting page sharing may be less important when virtualizing big-memory workloads than others.

For compute workloads, earlier studies have shown page sharing to be useful when there are large numbers of VMs.

## X. Related Work

*Performance Impact of Virtualization:* Since the resurgence of virtualization [17], the overheads of virtualization have been decreasing [1,11]. Hardware support for virtualization has greatly reduced the number and the latency of VMM interventions [11,36,41]. Binary translation can reduce these interventions further [3]. A recent study shows that workloads may have high virtualization overheads [16].

*Flat Page Table:* Ahn *et al.* [4] proposed replacing x86-64's four-level nested page table (gPA→hPA) with a flat one-level nested page table. This closely related work reduces the number of nested page table accesses from 4 to 1 and a 2D page walk (gVA→gPA & gPA→hPA) from 24 to 8 accesses. While this is promising for the small virtual machines they studied, it may be less suited for big-memory workloads. Our design reduces 2D page walks from 24 accesses to 4 or 0, which is better than 8 with the flat nested page table.

*Virtual Memory:* TLBs can cause performance degradation on native machines [9,10,11,13,14,37]. This is exacerbated when the workload is running in a virtualized system [1,4,11,16]. There have been two classical approaches to reducing overheads of virtual memory: reducing TLB miss latency and reducing TLB misses.

*Reducing TLB miss latency:* To reduce TLB miss latency, PTEs are cached in data caches [7,11,39] or TLB miss latency is hidden [13,15]. Prefetching PTEs can also improve performance [11,31,32]. A special structure was proposed to cache the nested translations (gPA→hPA) to reduce the latency of a TLB miss with virtualization [11,41]. Translation caching [7] and Large-Reach MMU caches [12] improved performance by caching any intermediate translation.

Some ISAs like SPARC use software managed TLBs and use a software-defined translation buffer (TSB) to service TLB misses faster [39]. Intel Itanium had a software managed section in TLB to pin critical address translations [27]. A recent proposal supported virtualization with software-managed TLBs [18]. We focus on hardware managed TLBs.

*Reducing TLB misses:* Large-pages improve TLB coverage, thus reducing TLB misses [22,23,47,49,50,51]. Most processors support multiple page sizes today [30]. However, applications and OS have been slow to support multiple page sizes [23,50]. Hardware support for large pages is difficult to design [8,48,50]. Coalescing contiguous or clustered PTEs have been shown to improve effective TLB size [42,43].

The other common approach for reducing TLB misses is virtual caching [10,31,45,46,54]. However, big-memory workloads often have poor cache performance, so virtual caching merely moves, but does not solve the problem.

*Segmentation:* Segmentation has been used in various processors. In general, it is used without any paging like in the early 8086 [28], or on top of paging, as in IA-32 [30] and MULTICS [21]. Segmentation on top of paging in x86-64 has been used with virtual machines [2]. A flat memory was also used in Blue Gene Linux for HPC workloads [55]. We use segmentation along with paging, but never for the same address. Segmentation in virtualized systems has been mentioned, but without any hardware design or evaluation [9].

## XI. Summary

This paper brings low-overhead virtualization to workloads with poor memory access locality. This is achieved with three new virtualized modes that improve on 2D page walks and direct segments. In addition, we propose two novel optimizations that greatly enhance the flexibility of direct segments. This design can greatly lower memory virtualization overheads for big-memory and compute workloads.


## Acknowledgment

We thank our anonymous reviewers, shepherd, K. McKinley, D. Gibson, and S. Reinhardt for their insightful comments and feedback on the paper. We thank Wisconsin Computer Architecture Affiliates and B. Serebrin for their feedback on an early version of the work. We thank R. Addai-Mununkum for proof-reading our drafts.

This work is supported in part by the National Science Foundation (CNS-1117280, CCF-1218323, CNS-1302260 and CCF-1438992), Google, and the University of Wisconsin (Kellett award and Named professorship to Hill). A. Basu's contribution to the paper occurred while at UW-Madison.




## References


[1] Adams, K. and Agesen, O. A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, (2006), pp. 2–13.

[2] Agesen, O., Garthwaite, A., Sheldon, J., and Subrahmanyam, P. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev. 44*, 4 (2010), pp. 3–18.

[3] Agesen, O., Mattson, J., Rugina, R., and Sheldon, J. Software techniques for avoiding hardware virtualization exits. *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX Association (2012), pp. 35–35.

[4] Ahn, J., Jin, S., and Huh, J. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. *Proceedings of the 39th Annual International Symposium on Computer Architecture*, (2012).

[5] Amazon Elastic Compute Cloud (Amazon EC2), Cloud Computing Servers. http://aws.amazon.com/ec2/.

[6] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. *Proceedings of the nineteenth ACM symposium on Operating systems principles and practice (SOSP '03)*, (2003), pp. 164–177.

[7] Barr, T.W., Cox, A.L., and Rixner, S. Translation caching: skip, don't walk (the page table). *Proceedings of the 37th Annual International Symposium on Computer Architecture*, (2010).

[8] Barr, T.W., Cox, A.L., and Rixner, S. SpecTLB: a mechanism for speculative address translation. *Proceedings of the 38th Annual In-*

*ternational Symposium on Computer Architecture*, (2011).

[9] Basu, A., Gandhi, J., Chang, J., Hill, M.D., and Swift, M.M. Efficient Virtual Memory for Big Memory Servers. *Proceedings of the 40th Annual International Symposium on Computer Architecture*, IEEE Computer Society (2013).

[10] Basu, A., Hill, M.D., and Swift, M.M. Reducing Memory Reference Energy With Opportunistic Virtual Caching. *ISCA '12: Proceedings of the 39th annual international symposium on Computer architecture*, (2012), pp. 297–308.

[11] Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. Accelerating two-dimensional page walks for virtualized systems. *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, (2008).

[12] Bhattacharjee, A. Large-Reach Memory Management Unit Caches. *Proceedings of the 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society (2013).

[13] Bhattacharjee, A., Lustig, D., and Martonosi, M. Shared last-level TLBs for chip multiprocessors. *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, IEEE Computer Society (2011), pp. 62–63.

[14] Bhattacharjee, A. and Martonosi, M. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society (2009), pp. 29–40.

[15] Bhattacharjee, A. and Martonosi, M. Inter-core cooperative TLB for chip multiprocessors. *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, (2010).

[16] Buell, J., Hecht, D., Heo, J., Saladi, K., and Taheri, R.H. Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications. *VMware Technical Journal*, Summer 2013.

[17] Bugnion, E., Devine, S., Govil, K., and Rosenblum, M. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems 15*, 4 (1997), pp. 319–349.

[18] Chang, X., Franke, H., Ge, Y., Liu, T., Wang, K., Xenidis, J., Chen, F., and Zhang, Y. Improving virtualization in the presence of software managed translation lookaside buffers. *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ACM (2013), pp. 120–129.

[19] Corbet, J. Transparent huge pages. 2011. www.lwn.net/Articles/423584/.

[20] Corbet, J. Memory compaction. http://lwn.net/Articles/368869/.

[21] Daley, R.C. and Dennis, J.B. Virtual memory, processes, and sharing in Multics. *Proceedings of the first ACM symposium on Operating System Principles*, ACM (1967), 12.1–12.8.

[22] Fang, Z., Zhang, L., Carter, J.B., Hsieh, W.C., and McKee, S.A. Reevaluating Online Superpage Promotion with Hardware Support. *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, IEEE Computer Society (2001), pp. 63–.

[23] Ganapathy, N. and Schimmel, C. General purpose operating system support for multiple page sizes. *Proceedings of the annual conference on USENIX Annual Technical Conference*, USENIX Association (1998), pp. 8–8.

[24] Gandhi, J., Basu, A., Swift, M.M., and Hill, M.D. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *SIGARCH Computer Architecture News*, (2014).

[25] Goldberg, R.P. Survey of virtual machine research. *Computer 7*, 9 (1974), pp. 34–45.

[26] Hwang, A.A., Ioan A. Stefanovici, and Schroeder, B. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, (2012), pp. 111–122.

[27] Intel® Itanium® Architecture Developer's Manual, Vol. 2. http://www.intel.com/content/www/us/en/processors/itanium/itanium-architecture-software-developer-rev-2-3-vol-2-manual.html.

[28] Intel 8086 - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Intel_8086.

[29] Intel Corp. Intel Chipset 4GB System Memory Support. 2005. http://www.polywell.com/us/support/faq/4gb_rev1.pdf.

[30] Jacob, B. and Mudge, T. Virtual Memory in Contemporary Microprocessors. *IEEE Micro 18*, 4 (1998), pp. 60–75.

[31] Jacob, B. and Mudge, T. Uniprocessor Virtual Memory without TLBs. *IEEE Trans. Comput. 50*, 5 (2001), pp. 482–499.

[32] Kandiraju, G.B. and Sivasubramaniam, A. Going the distance for TLB prefetching: an application-driven study. *Proceedings of the 29th Annual International Symposium on Computer Architecture*, (2002).

[33] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. kvm: the Linux Virtual Machine Monitor. *Proceedings of the Linux Symposium*, (2007), pp. 225–230.

[34] Linux Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page.

[35] Linux Virtio Balloon Driver. http://lxr.free-electrons.com/source/drivers/virtio/virtio_balloon.c.

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

[37] Lustig, D., Bhattacharje, A., and Martonosi, M. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Transactions on Architecture and Code Optimization*, (2013).

[38] Memory Hotplug. https://www.kernel.org/doc/Documentation/memory-hotplug.txt.

[39] Microsystems, S. UltraSPARC T2™ Supplement to the UltraSPARC Architecture 2007. 2007.

[40] PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. 2011. http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html.

[41] VMware. *Performance Evaluation of Intel EPT Hardware Assist.* 2008.

[42] Pham, B., Bhattacharjee, A., Eckert, Y., and Loh, G.H. Increasing TLB reach by exploiting clustering in page translations. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, (2014), pp. 558–567.

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

[44] Sanchez, D., Yen, L., Hill, M.D., and Sankaralingam, K. Implementing Signatures for Transactional Memory. *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, (2007).

[45] Sembrant, A., Hagersten, E., and Black-Schaffer, D. The Direct-to-Data (D2D) Cache: Navigating the Cache Hierarchy with a Single Lookup. *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, IEEE Press (2014), pp. 133–144.

[46] Sembrant, A., Hagersten, E., and Black-Shaffer, D. TLC: A Tagless Cache for Reducing Dynamic First Level Cache Energy. *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM (2013), pp. 49–61.

[47] Seznec, A. Concurrent Support for Multiple Page Sizes on a Skewed Associative TLB. *IEEE Transactions on Computers 53(7)*, (2004), pp. 924–927.

[48] Subramanian, I., Mather, C., Peterson, K., and Raghunath, B. Implementation of Multiple Pagesize Support in HP-UX. *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, USENIX Association (1998), 9–9.

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

[50] Talluri, M. and Hill, M.D. Surpassing the TLB performance of superpages with less operating system support. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, (1994).

[51] Talluri, M., Kong, S., Hill, M.D., and Patterson, D.A. Tradeoffs in Supporting Two Page Sizes. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (1992).

[52] Waldspurger, C.A. Memory Resource Management in VMware ESX Server. *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, (2002).

[53] Wang, X., Zang, J., Wang, Z., Luo, Y., and Li, X. Selective hardware/software memory virtualization. *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ACM (2011), pp. 217–226.

[54] Wood, D.A., Eggers, S.J., Gibson, G., Hill, M.D., and Pendleton, J.M. An in-cache address translation mechanism. *Proceedings of 13th annual international symposium on Computer architecture*, (1986).

[55] Yoshii, K., Iskra, K., Naik, H., Beckman, P., and Broekema, P. Characterizing the Performance of "Big Memory" on Blue Gene Linux. *International Conference on Parallel Processing Workshops, 2009. ICPPW '09*, (2009), pp. 65–72.