Devirtualizing memory in heterogeneous systems

Swapnil Haria Mark D. Hill Michael M. Swift
University of Wisconsin-Madison
{swapnilh, markhill, swift} @cs.wisc.edu

Abstract

Accelerators are increasingly recognized as one of the major drivers of future computational growth. For accelerators, unified virtual memory (VM) promises to simplify programming and provide safe data sharing with CPUs. Unfortunately, the overheads of virtual memory, which are high for general-purpose processors, are even higher for accelerators. Providing accelerators with direct access to physical memory (PM) in contrast, provides high performance but is both unsafe and difficult to program.

We propose the De-Virtualized Memory (DVM) scheme to combine the protection of VM with direct access to PM. By allocating memory such that physical and virtual addresses are almost always identical (PA==VA), DVM overlaps most read accesses with validation of access permissions. Moreover, DVM leverages the contiguity of permissions to expedite permission validation. DVM requires modest OS and IOMMU changes, and is transparent to the application.

Implemented in Linux 4.10 for a graph-processing accelerator, DVM reduces VM overheads to less than 2% on average. DVM also improves performance by 2.1X over a highly-optimized, conventional VM implementation, while consuming 3.9X less dynamic energy for memory management. We further discuss DVM's potential to extend beyond accelerators to CPUs, where it reduces VM overheads to 5% on average, down from 29% for conventional VM.

1. Introduction

The end of Dennard Scaling and slowing of Moore's law has weakened the future potential of general-purpose computing. To keep up with the ever-increasing computational needs of society, research focus has intensified on heterogeneous systems with special-purpose accelerators alongside conventional processors. In such systems, computations are offloaded by general-purpose cores to one or more accelerators.

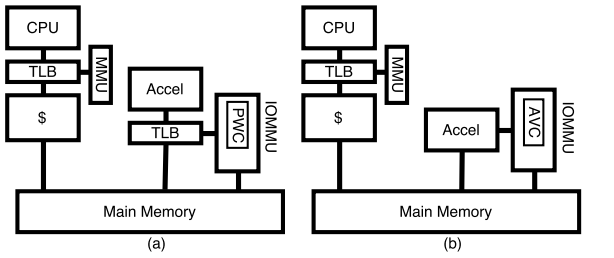

Figure 1. Heterogeneous systems with (a) conventional VM with address translation on critical path and (b) our proposal with access validation alongside direct access to PM.

Beyond existing accelerators like GPUs, accelerators for big-memory workloads with irregular access patterns are steadily gaining prominence [23]. In recent years, proposals for customized accelerators for graph processing [27, 1], data analytics [60, 61] and neural computing [15, 28] have shown performance and/or power improvements of several orders of magnitude over conventional processors. The success of industrial efforts such as Google's Tensor Processing Unit (TPU) [33] and Oracle's Data Analytics Accelerator (DAX) [58] further strengthens the case for heterogeneous computing. Unfortunately, existing memory management schemes are not a good fit for these accelerators.

Such accelerators ideally want to access host physical memory without needing address translation. Such direct access removes the need for data copies and facilitates sharing of data between accelerators and CPUs. Moreover, removing address translation simplifies memory management by eliminating large, power-hungry hardware structures such as translation lookaside buffers (TLBs). The low power and area consumption of this memory management scheme are extremely attractive for small accelerators.

However, direct access to physical memory (PM) is not generally acceptable. Applications rely on the memory protection and isolation of virtual memory (VM) to prevent malicious or erroneous accesses to their data [41]. Similar protection guarantees are needed when accelerators are multiplexed among multiple processes. Additionally, a shared virtual address space is needed to support 'pointer-is-a-pointer'

CONF'yy Month d–d, 20yy, City, ST, Country

ISBN 978-1-nnnn-nnnn-n/yy/mm. . . $15.00

DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

semantics. This allows pointers to be dereferenced on both the CPU and the accelerator. Enabling the use of pointer-based data structures increases the programmability of heterogeneous systems.

Unfortunately, the benefits of VM come with high overheads, particularly for accelerators. Supporting conventional VM in accelerators requires memory management hardware like page-table walkers and TLBs. For CPUs, address translation overheads have worsened with increasing memory capacities, reaching up to 50% for some big-memory workloads [5, 35]. These overheads occur in processors with massive two-level TLBs and may be more pronounced in accelerators with simpler translation hardware.

Is it possible to enforce the protection of VM with low-overhead access to PM? Conditions that required VM in the past are changing. In the past, swapping was crucial in systems with limited physical memory. Today, high-performance systems are often configured with sufficient PM to mostly avoid swapping. Vendors already offer servers with 64 TB of PM [53], and capacity is expected to further expand with the emergence of non-volatile memory technologies [31, 21].

Leveraging these opportunities, we propose a radical idea to *de-virtualize* virtual memory by eliminating address translation on most memory accesses (Figure 1). We achieve this by allocating most memory such that its virtual address (VA) is the same as its physical address (PA). We refer to such allocations as Identity Mapping (PA==VA). As the PA for most accesses is identical to the VA, accelerators can initiate read accesses in parallel with access validation. For identity-mapped pages, access validation simply verifies that the process holds valid permissions for the access. For pages which are not identity-mapped, conventional address translation is performed to generate the translated PA. Thus, we preserve the VM abstraction.

Eliminating page-level translations for identity-mapped pages enables enables new page table structures to uncover and exploit the underlying contiguity of permissions. Permissions are typically granted and enforced at coarser granularities and are uniform across regions of virtually contiguous pages, unlike translations. Thus, we introduce Permission Entry (PE), which is a new page table entry format for storing coarse-grained permissions. PEs both reduce the size of the overall page tables and reduce page-walk latencies.

DVM for accelerators is completely transparent to applications, and requires small OS changes to identity map memory allocations on the heap. We modify the IOMMU for checking permissions, ensuring PA==VA and address translation for pages not identity-mapped. To minimize IOMMU overheads, we propose two structures for shortening page walks.

Furthermore, devirtualized memory can optionally also be used to reduce VM overheads for CPUs by identity mapping all segments in a process's address space. This requires additional OS and hardware changes. DVM for CPUs (cDVM) provides similar benefits on loads, and also optimizes stores.

However, the DVM scheme does have some limitations. Eager and contiguous memory allocation aggravates the problem of memory fragmentation, although we do not study this effect in this paper. Identity mapping requires the use of position independent executables (PIE), which is not the default option in most OSes yet. Finally, DVM precludes the use of Copy-On-Write (COW) and by extension, fork system call (which uses COW). Thus, DVM is not as flexible as VM, but avoids most of the VM overheads.

This paper describes a unified memory management scheme for heterogeneous systems and makes these contributions:

- We propose the DVM scheme to minimize the overheads of Virtual Memory, and implement OS support in Linux 4.10.
- We develop a compact page table representation by exploiting the contiguity of permissions through a new page table entry called the Permission Entry.
- We design the Access Validation Cache (AVC) to replace both TLBs and Page Walk Caches (PWC). For a graph processing accelerator, DVM with an AVC is 6.3x faster than a baseline VM implementation using a TLB with 4KB pages. More significantly, our design is 2.1X faster and while consuming 3.9X less dynamic energy for memory management than a highly-optimized VM implementation with 2M pages.
- We extend DVM to support CPUs (cDVM), thereby enabling a unified memory management scheme throughout the heterogeneous system. cDVM lowers the overheads of VM in big-memory workloads to 5% for CPUs.

2. Background

Our work focuses on accelerators running big-memory workloads with irregular access patterns such as graph-processing, machine learning and data analytics. As motivating examples, we use graph-processing applications like Breadth-First Search, PageRank, Single-Source Shortest Path and Collaborative Filtering with different input graphs as described in Section 6. First, we discuss why existing approaches for memory management are not a good fit for these workloads.

Accelerator programming models employ one of two approaches for memory management (in addition to unsafe direct physical memory access). Some accelerators use a separate physical address space for the accelerator [33, 40]. This necessitates explicit copies when sharing data between the accelerator and the host processor. Such approaches are similar to discrete GPGPU programming models. As such, they are plagued by the same problems: (1) the high overheads of data copying require larger offloads to be economical;

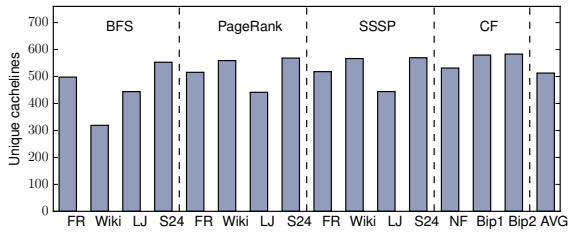

Figure 2. Number of unique pages/cachelines on coalescing 1000 references

Figure 3. TLB misses in our accelerator workloads with 128-entry TLB for 4K and 2M pages

and (2) this approach makes it difficult to support pointer-is-a-pointer semantics, which reduces programmability and complicates the use of pointer-based data structures such as graphs.

To facilitate data sharing, accelerators (mainly GPUs) have started supporting unified virtual memory, in which accelerators can access PM shared with the CPU using virtual addresses. This approach typically relies on an IO memory management unit (IOMMU) to service address translation requests from accelerators [2, 32], as illustrated in Figure 1. We focus on these systems, as address translation overheads severely degrade the performance of these accelerators [16].

In GPUs, several techniques have been proposed to lower VM overheads by exploiting the regular memory patterns of GPGPU applications [46, 45]. Coalescing memory requests reduces GPU address translation traffic by up to 85% [46]. However, coalescing is not effective for irregular memory accesses. For the graph workloads described in Section 6.1, coalescing memory accesses only reduces translation traffic by 50%, resulting in a high rate of translation traffic. Such a high frequency of requests will overwhelm TLBs. Furthermore, page-based TLBs have limited reach and their performance depends on the temporal locality of references. For our workloads, we observe high TLB miss rates of 21% on average even for a 128-entry TLB as shown in Figure 3. There is so little spatial locality that using larger 2MB pages only improves the TLB miss rates by 1% on average.

Other accelerators share parts of the address space with CPUs [1, 22, 37]. Such accelerators typically perform simple address translation using a base-plus-offset scheme similar to Direct Segments [5]. Using such a scheme, only memory within a single contiguous region of physical memory can be shared, providing limited flexibility. We can add complicated address translation schemes such as range translations [35], which support multiple address ranges. However, this adds a large and power-hungry Range TLB, which may be prohibitive given the area and power budgets of accelerators.

As a result, we see that there is a clear need for a simple (i.e., efficient) and performant memory management scheme for accelerators with irregular memory accesses.

3. Devirtualizing Virtual Memory

In this section, we present a high-level view of our Devirtualized Virtual Memory (DVM) scheme. Before discussing DVM, we enumerate the goals for a memory management scheme suitable for accelerators (as well as CPUs).

3.1 List of Goals

- **Programmability.** Simple programming models are important for increased adoption of accelerators. Data sharing between CPUs and accelerators must be supported, as accelerators are typically used for executing parts of an application. For this, a scheme must preserve pointer-is-a-pointer semantics. This improves the programmability of accelerators by allowing the use of pointer-based data structures without data copying or marshalling [50].
- **Power/Performance.** An ideal memory management scheme should have near zero overheads even for irregular access patterns in big-memory systems. Additionally, MMU hardware must consume little area and power. Accelerators are attractive when they offer large speedups under small resource budgets. Thus, large, power-hungry TLBs are unattractive for accelerators.
- **Flexibility.** Memory management scheme must be flexible enough to support dynamic memory allocations of varying sizes and with different permissions. This precludes approaches whose benefits are limited to a single range of contiguous virtual memory.
- **Safety.** No accelerator should be able to reference a physical address without the right authorization for that address. This is necessary for guaranteeing the memory protection offered by virtual memory. This protection attains greater importance in heterogeneous systems to safeguard against buggy or malicious third-party accelerators [42].

3.2 Devirtualized VM

The primary goal of DVM is to reduce VM overheads. Towards this end, DVM allows accelerators to directly access the majority of physical memory by eliminating the need for address translation on most memory accesses. DVM enforces memory protection by checking application permissions for each access using an efficient data structure. Hence, DVM satisfies the primary tenet of exokernels which is to al-

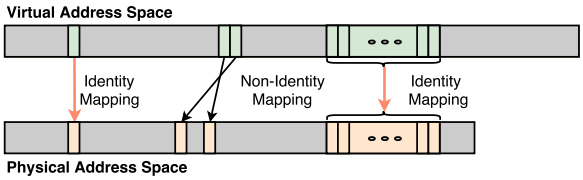

Figure 4. Identity mapping a single page as well as a large 2MB memory region.

low hardware resources to be accessed directly but securely by applications [20].

To eliminate address translation on a majority of accesses, DVM uses identity mapping for most of the address space. Identity mapping allocates most memory by assigning VAs equal to the backing PAs. Thus, on memory fetches, hardware can assume the physical address is equal to the virtual address (PA==VA) and retry with full translation if it is not, as shown in Figure 5. Permissions and identity mapping are checked in parallel with the data fetch. If PA is found to not be equal to VA, the translated PA is computed as part of checking without needing a separate address translation. On stores, permissions must be checked before writing data, but can leverage the more efficient data structure holding permissions rather than full translations. We refer to these functions, performed by the IOMMU, as access validation.

In the common case when PA==VA, accelerators and applications access PM directly without the overheads of VM. When PA!=VA, DVM behaves like conventional virtual memory and translates addresses. As a result, DVM offers the best of both worlds by combining the functionality of VM when needed with the performance of physical memory. This allows us to support low-overhead shared virtual memory in a heterogeneous system.

Programmability. DVM bridges the programmability gap between CPUs and accelerators by supporting shared regions of PM with common virtual addresses. Consequently, a pointer on a CPU remains a valid pointer on an accelerator. Thus, accelerators can operate directly on pointer-based CPU data structures without the need for calculating addresses. This lowers the granularity at which offloading becomes economical and facilitates fine-grained sharing between CPUs and accelerators.

Power/Performance. DVM reduces the overheads of shared virtual memory by replacing slow address translation with fast access validation. Information about permissions and identity mapping is small and can be stored at a coarser granularity than page translations. This information can be cached efficiently in a standard cache, whose reach can cover most of virtual memory, unlike the limited reach of conventional TLBs. Hence, access validation (Figure 5) has lower overheads than address translation.

Flexibility. DVM facilitates page-level sharing between the accelerator and the host CPU, as regions as small as a single page can be identity mapped with distinct permissions, as

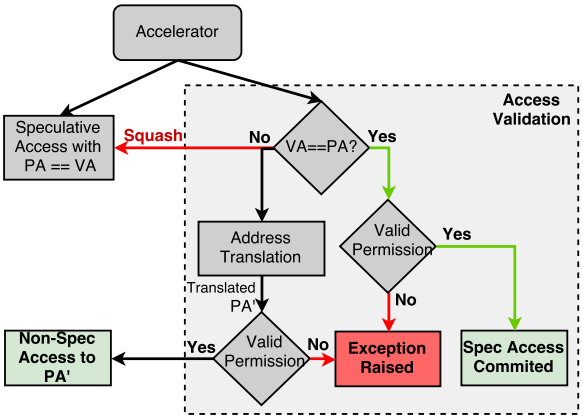

Figure 5. Memory Accesses in DVM. PA' is the translated PA when identity mapping does not hold.

shown in Figure 4. This allows DVM to benefit a variety of applications, including those that do not have a single contiguous heap. Furthermore, DVM is transparent to the application.

Safety. While DVM offers almost direct access to PM by skipping address translation, all accesses are still checked for valid permissions. This is similar to protection in single address-space operating systems [14]. Thus, DVM does not compromise on the safety and protection offered by conventional VM. Although DVM allows speculative data fetch assuming PA==VA, the access is only committed on successful access validation. If the permissions are not sufficient, an exception is raised on the host, as shown in Figure 5.

4. Implementing DVM for Accelerators

Having established the high-level model of DVM, we now dive into its hardware and software implementation. We add support for DVM in accelerators with modest changes to the OS and IOMMU but without any CPU hardware modifications.

In this section, we first describe two alternative mechanisms for fast access validation. Next, we show how access validation overheads can be minimized further by overlapping it with data fetch. Finally, we discuss our OS modifications to support identity mapping. Here, we use the term memory region to refer to a collection of virtually contiguous pages with the same permissions. Also, we use page table entries (PTE) to mean entries at any level of the page table, specifying level as needed.

4.1 Access Validation

We support access validation with (1) standard page tables and a bitmap for caching permissions, or (2) compact page tables and an access validation cache. For both mechanisms, we use the following 2-bit encoding for permissions—No permissions, 01:Read-Only, 10:Read-Write and 11:Read-Execute.

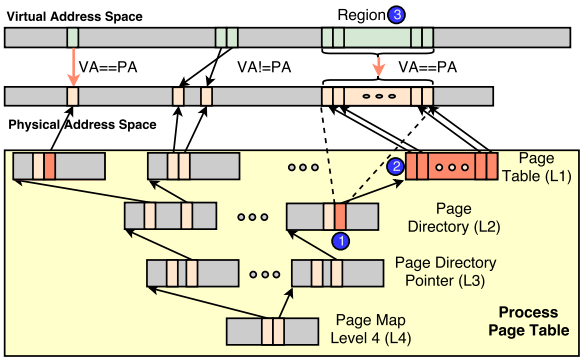

Figure 6. 4-level address translation in x86-64 for 48-bit virtual address space.

4.1.1 Bitmap

The Bitmap (BM) stores page permissions for identity-mapped pages. For such pages, virtual-to-physical address mappings are not needed. Instead, we apply the invariant that any page with valid permissions (!=00) in the BM is identity mapped. For each memory access from an accelerator, the IOMMU consults the BM to check permissions. For pages with 00 permissions, the IOMMU relies on regular address translation with a full page table walk. If the page is identity mapped, the IOMMU updates the BM with its permissions.

The Bitmap is a flat bitmap in physical memory, storing 2-bit permissions for each physical page. As with page tables, there is a separate bitmap for each process. It is physically indexed and lookups assume PA==VA. For a page size of 4KB, this incurs storage overheads of approximately 0.006% of the physical memory capacity for each active accelerator. Finding a page's permissions simply involves calculating the offset into the bitmap and adding it to the bitmap's base address. We cache BM entries along with L2-L4 PTEs in a simple data cache to expedite access validation.

The hardware design of the Bitmap is similar to the Protection Table used in Border Control (BC) [41], although the storage and lookup policies differ. The Protection Table in BC stores permissions for all physical pages, and is looked up after address translation. BM in DVM stores permissions for only identity-mapped pages, and is looked up to avoid address translation.

The simple BM has the benefit of leaving host page tables unchanged. But, it stores permissions individually for every physical page, which is inefficient for big-memory systems, especially with sparse memory usage. We address this drawback with our alternative mechanism described next.

4.1.2 Compact Page Tables.

We can leverage the contiguity of permissions to store permissions at a coarse granularity resulting in a compact page table structure. Figure 6 shows an x86-64 page table. An L2 Page Directory entry (L2PDE, the entries one level up from the leaves containing page translations), (1) maps a contiguous 2MB VA range (region 3). For storing pointers to PAs for each 4K page in this range, 512 L1 page table entries (PTEs) (2) are required, using 4KB of memory. However, if pages are identity mapped, PAs are already known and only permissions need to be stored. By storing permissions at a coarser granularity in the L2PDE itself, the page walk ends at the L2 level and the page table is much smaller. For larger regions, permissions can also be stored at the L3 and L4 levels.

We introduce a new type of leaf page table entry called the Permissions Entry (PE), shown in Figure 7. PEs are direct replacements for regular entries at any level, with the same size (8 bytes) and mapping the same VA range. PEs contain sixteen separate 2-bit permissions for 16-aligned regions constituting the VA range mapped by the PE. To distinguish between PEs and other PTEs, an identity mapping bit (IM) is added to all entries. The IM bit is set for PEs and unset for other entries.

Each PE stores separate permissions for sixteen aligned regions, together comprising the VA range mapped by the PE. Each constituent region is 1/16th the size of the range mapped by the PE, aligned on an appropriate power-of-two granularity. For instance, the 2 MB VA range mapped by an L2PE is made up of sixteen 128 KB (2 MB/16) regions aligned on 128 KB address boundaries. We treat unallocated memory in the mapped VA range as a region with no permissions (00).

A PE can replace a regular entry at any page table level and its entire sub-tree if all allocated addresses in the mapped VA range are part of identity-mapped regions, aligned and sized to a valid power of two. More simply, PEs implicitly guarantee that any allocated memory in the mapped VA range is identity-mapped. The L2PDE marked (1) in Figure 6 can be replaced by a PE, as all of region 3 is identity mapped. If region 3 is replaced by two adjacent 128 KB regions at the start of the mapped VA range with the rest unmapped, we could still use an L2PE to map this range, with relevant permissions for the first two regions, and 00 permissions for the rest of the memory in this range.

On an accelerator memory request, the IOMMU performs access validation by walking the page table. A page walk ends on encountering a PE, as PEs store information about identity mapping and permissions. If insufficient permissions are found in the PE, the IOMMU raises an exception on the host CPU. If a page walk encounters a leaf PTE, and the accessed VA is not identity mapped, the leaf PTE is then used to generate the PA using the page offset from the VA. This avoids a separate walk of the page table to translate the address.

The introduction of PEs significantly reduces the number of these L1PTEs and thus the size of the page tables. As shown in Table 1, L1PTEs comprise about 98% of the size of the page tables. PEs at higher levels (L2 or L3) replace entire sub-trees of the page table below them. For instance,

Input Graph	Page Tables (in KB)	% occupied by L1PTEs	Page Tables with PEs (in KB)
FR	616	0.948	48
Wiki	2520	0.987	48
LJ	4280	0.992	48
S24	13340	0.996	60
NF	4736	0.992	52
BIP1	2648	0.989	48
BIP2	11164	0.996	68

Table 1. Page table sizes with and without PEs for PageRank and CF. PEs reduce the page table size by eliminating most L1PTEs.

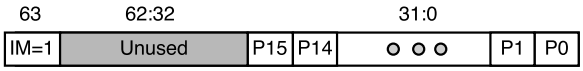

Figure 7. Structure of a Permission Entry. IM: Identity-Mapped, P15-P0: Permissions.

replacing an L3PE with a PE eliminates 512 L2PDEs and up to 512×512 L1PTEs, saving as much as 2.05 MB. Thus, PEs make page tables more compact.

4.1.3 Access Validation Cache.

The major value of smaller page tables is improved efficacy of caching PTEs. In addition to TLBs which cache PTEs, modern IOMMUs also include page walk caches (PWC) to store L2-L4 PTEs [4]. During the course of a page walk, the page table walker first looks up internal PTEs in the PWC before accessing main memory. In existing systems, L1PTEs are not cached to avoid polluting the PWC [8]. Hence, page table walks on TLB misses incur at least one main memory access, for obtaining the L1PTE.

We propose the Access Validation Cache (AVC), which caches all intermediate and leaf entries of the page table, to replace TLBs and PWCs for accelerators. On every memory reference by an accelerator, the IOMMU walks the page table using the AVC. In the best case, the AVC services page walks without any main memory accesses. Caching L1PTEs allows AVC to exploit their temporal locality, as done traditionally by TLBs. But, L1PTEs do not pollute the AVC as the introduction of PEs greatly reduces the number of L1PTEs. Thus, the AVC can perform the role of both a TLB and a traditional PWC.

The AVC is a standard 4-way set-associative cache with 64B blocks. The AVC caches 128 distinct entries, resulting in a total capacity of 1 KB. It is physically-indexed and physically tagged cache, as page table walks use physical addresses. With 64B blocks, eight 8B entries are fetched together from memory on a cache miss, exploiting the spatial locality of page tables. For PEs, this provides 128 sets of permissions. The AVC does not support translation skipping [4].

Due to the reduced size of the page tables, even a small 128-entry (1KB) AVC has very high hit rates, resulting in fast access validation. Using only PEs at L2, the AVC provides permissions for almost 2GB. As the hardware design is similar to a conventional PWC, the AVC is just as energy-efficient. Moreover, the AVC is more energy-efficient than a comparably sized, fully-associative TLB, as it has a less associative lookup.

4.1.4 Speculative data fetch.

If an accelerator supports the ability to squash and retry an inflight load, DVM allows speculative data fetch to occur in parallel with access validation. As a result, the validation latency for loads can be overlapped with the memory access latency. If the access is validated successfully (the page walk ends in a PE or a PTE that is identity mapped), the speculative fetch is ratified. Otherwise, it is discarded, and the access is retried to the correct, translated PA.

The speculative data fetch alleviates the effect of the IOMMU latency, as it is no longer on the critical path of most loads. While this technique is only possible for loads, it greatly improves performance as our workloads have ˜5X more loads than stores. For stores, this optimization is not possible because the physical address must be validated before the store updates memory.

4.2 OS Support for Accelerator-only DVM.

Only minor changes are required in the OS to support DVM in accelerators. This is because accelerators access few parts of the process address space like the heap and memory-mapped segments, but not typically the code or stack.

To ensure PA==VA for most addresses in memory, physical frames (and thus PAs) need to be reserved at the time of memory allocation. We refer to this as eager paging [35]. Next, a flexible address space is needed in which the heap can be mapped anywhere in the process address space as opposed to a hardcoded location. Below, we describe our implementation in Linux 4.10.

4.2.1 Eager Contiguous Allocations.

Identity mapping in DVM is enabled by eager contiguous allocations of memory. On memory allocations, the OS allocates physical memory then sets the VA equal to the PA. This is unlike demand paging used by most OSes, which allocates physical frames lazily at the time of first access to a virtual page. For allocations larger than a single page, contiguous allocation of physical memory is needed to guarantee PA==VA for all the constituent pages. We utilize the eager paging modifications to Linux's default buddy allocator developed by others [35] to allocate contiguous powers-of-two pages. Eager allocation can increase memory use if programs allocate much more memory then they actually use.

4.2.2 Flexible Address Space.

Operating systems historically dictated the layout of user-mode address spaces, specifying where code, data, heap, and stack reside. For identity mapping, our modified OS assigns

VAs equal to the backing PAs. Unfortunately, there is little control over the allocated PAs without major changes to the default buddy allocator in Linux. As a result, we could have a non-standard address space layout, for instance with the heap below the code segment in the address space. To allow such cases, the OS needs to support a flexible address space with no hard constraints on the location of the heap and memory-mapped segments.

Heap. Data sharing between the accelerator and CPU is done through the heap, which we identity map. We modify the default behavior of glibc `malloc` to always use the `mmap` system call instead of `brk`. This is because identity mapping of dynamically growing a region, as needed for `brk` is not possible without major changes to Linux. We initially allocate a memory pool to handle small allocations. Another pool is allocated when the first is full. Thus, we turn the heap into discontiguous memory-mapped segments, which we discuss next.

Memory-mapped segments. We modify the kernel to accommodate memory-mapped segments anywhere in the address space. Address Space Layout Randomization (ASLR) already allows randomizing the base positions of the stack, heap as well as memory-mapped regions (libraries) [57]. Our implementation further extends this to randomize the relative positions of the segments.

Low-memory situations. While most high-performance systems are configured with sufficient memory capacity, contiguous allocations can result in fragmentation over time and preclude further contiguous allocations.

In low memory situations, DVM reverts to standard demand paging. Furthermore, to reclaim memory, the OS could convert permission entries to standard PTEs and swap out memory (not implemented). We expect such situations to be rare in big-memory systems, which are our main target. Also, once there is sufficient free memory, the OS can reorganize memory to reestablish identity mappings.

5. Discussion

In this section, we address potential concerns regarding the devirtualized virtual memory scheme.

Security implications. While DVM sets PA==VA in the common case, this does not weaken any isolation properties. Just because applications can address all of physical memory does not give them permissions to access it [14]. This is commonly exploited by OSes. For instance, in Linux, all physical memory is mapped into the kernel address space, which is part of every process. Although this memory is addressable by an application, any user-level access will to this region will be blocked by hardware due to lack of permissions in the page table. Furthermore, even the strong isolation offered by conventional VM is still vulnerable to attacks such as the DRAM Rowhammer attack [36].

The semi-flexible address space layout used in modern OSes allows limited randomization of address bits. For instance, Linux provides 28 bits of ASLR entropy while Windows 10 offers 24 bits for the heap. The stronger Linux randomization has already been derandomized by software [25, 52] and hardware-based attacks [26]. A comprehensive security analysis of DVM is beyond the scope of this work.

Copy-on-Write (CoW). CoW is an optimization for minimizing the overheads of copying data, by deferring the copy operation till the first write. Before the first write, both the source and destination get read-only permissions to the original data. It is most commonly used by the `fork` system call to create new processes.

CoW can be performed with DVM without any correctness issues. Before any writes occur, there is harmless read-only aliasing. The first write in either process allocates a new page for a private copy, which cannot be identity-mapped, as its VA range is already visible to the application, and the corresponding PA range is allocated for the original data. Thus, the OS reverts to standard paging for the address. Thus, we recommend against using CoW for data structures allocated using identity mapping.

Unix-style Fork. The `fork` operation in Unix creates a child process, and copies a parent's private address space into the child process. Commonly, CoW is used to defer the actual copy operation. As explained in the previous section, CoW works correctly, but can break identity mapping.

Hence, we recommend calling `fork` before allocating structures shared with accelerators. If processes must be created later, then the `posix_spawn` call (combined `fork` and `exec`) should be used when possible to create new processes without copying. Alternatively, `vfork`, which shares the address space without copying, can be used, although it is typically considered less safe than `fork`.

Virtual Machines. DVM can be extended to work in virtualized environments as well. The overheads of conventional virtual memory are exacerbated in such environments as memory accesses need two levels of address translation (1) guest virtual address (gVA) to guest physical address (gPA) and (2) guest physical address to system physical address (sPA). This two-level translation results in as many as 24 memory references for each TLB miss in a virtual machine [7].

To reduce these costs, DVM can be extended in three ways. With guest OS support for multiple non-contiguous physical memory regions, DVM can be used to map the gPA to the sPA directly in the hypervisor, or in the guest OS to map gVA to gPA. These approaches convert the two-dimensional page walk to a one-dimensional walk. Thus, DVM brings down the translation costs to unvirtualized levels. Finally, there is scope for broader impact by using DVM for directly mapping gVA to sPA, eliminating the need for address translation on most accesses.

Comparison with Huge Pages. Here we offer a qualitative comparison, backed up by a quantitative comparison in Section 6. DVM breaks the serialization of translation and data

CPU	
Cores	1
Caches	64KB L1, 2MB L2
Frequency	3 GHz
Accelerator	
Processing Engines	8
TLB Size	128-entry FA
TLB Latency	1 cycle
PWC/AVC Size	128-entry, 4-way SA
PWC/AVC Latency	1 cycle
Frequency	1 GHz
Memory System	
Memory Size	32 GB
Memory B/W	4 channels of DDR4 (51.2 GB/s)

Table 2. Simulation Configuration Details

Graph	# Vertices	# Edges	Working Set
Flickr (FR) [19]	0.82M	9.84M	288 MB
Wikipedia (Wiki) [19]	3.56M	84.75M	1.26 GB
LiveJournal (LJ) [19]	4.84M	68.99M	2.15 GB
RMAT Scale 24 (RMAT)			6.79 GB
Netflix (NF) [6]	480K users, 18K movies	99.07M	2.39 GB
Synthetic Bipartite 1 (SB1)	969K users, 100K movies	53.82M	1.33 GB
Synthetic Bipartite 2 (SB2)	2.90M users, 100K movies	232.7M	5.66 GB

Table 3. Graph datasets used for evaluation

fetch, unlike huge pages. Also, DVM exploits finer granularities of contiguity by having 16 permission fields in each PE. Specifically, 128KB (=2MB /16) of contiguity is sufficient for leveraging 2MB L2PEs, and 64MB (=1GB/16) contiguity is sufficient for 1GB L3PEs.

Moreover, huge pages are not scalable with growth in memory capacity. As memory sizes grow, new page sizes must be supported in hardware and system software while retaining support for the existing sizes. For instance, TLBs have to be redesigned to support multiple page sizes [54, 17].

Finally, huge pages do not change the fact that TLB performance depends on the locality of memory references. TLB performance can be an issue for future big-memory workloads with irregular or streaming accesses [43, 47], as we show in Figure 3. In comparison, DVM reduces exploits the locality in permissions which is found in most applications as a virtue of how memory is allocated.

6. Evaluation

6.1 Methodology

We quantitatively evaluate the DVM scheme using a heterogeneous system containing OOO cores and the Graphicionado graph-processing accelerator [27]. Graphicionado is a programmable graph accelerator optimized for the low computation-to-communication ratio of graph applications. In contrast to software frameworks, where 94% of the executed instructions are for data movement, Graphicionado uses an application-specific pipeline and memory system design to avoid such inefficiencies. Its execution pipeline and datapaths are geared towards graph primitives— edges and vertices. Furthermore, by allowing concurrent execution of multiple execution pipelines, the accelerator is able to exploit the available parallelism and memory bandwidth.

To match the flexibility of software frameworks, Graphicionado uses reconfigurable blocks to support the vertex programming abstraction. With this abstraction, a graph algorithm is expressed as operations on a single vertex and its edges. With three custom functions—processEdge, reduce and apply—most graph algorithms can be specified and executed on Graphicionado. The graph itself is stored as a list of edges, each in the form of a 3-tuple (srcid, dstid, weight). It also maintains a list of vertices where each vertex is associated with a vertex property (i.e., distance from root in BFS or rank in pagerank). The vertex properties are updated during execution. Graphicionado also maintains ancillary arrays for efficient indexing into the vertex and the edge lists.

We simulate a heterogeneous system with one CPU and the Graphicionado accelerator in the open-source, cycle-level gem5 simulator [12]. We implement Graphicionado with 8 processing engines and no scratchpad memory as an IO device with its own timing model in gem5. The computation performed in each stage of a processing engine is executed in one cycle, and memory accesses are made to a shared physical memory. We use gem5's full-system mode to run workloads on our modified Linux operating system. The configuration details of the simulation are shown in Table 2. For energy results, we use access energy numbers from Cacti [38] and access counts from our gem5 simulation.

6.2 Workloads

We run four common graph algorithms on our graph-processing accelerator—PageRank, Breadth-First Search, Single-Source Shortest path and Collaborative Filtering. We run each of these workloads with multiple real-world as well as synthetic graphs. The details of the input graphs can be found in Table 3. The synthetic graphs are generated using the graph500 RMAT data generator [13, 39]. To generate synthetic bipartite graphs, we convert the synthetic RMAT graphs following the methodology described by Satish et al [51].

6.3 Results

This section evaluates the performance of DVM for accelerator and CPU workloads. We also analyze the energy-efficiency of DVM versus conventional VM for accelerators. Lastly, we report the efficacy of identity mapping in terms of the percentage of overall system memory successfully allocated.

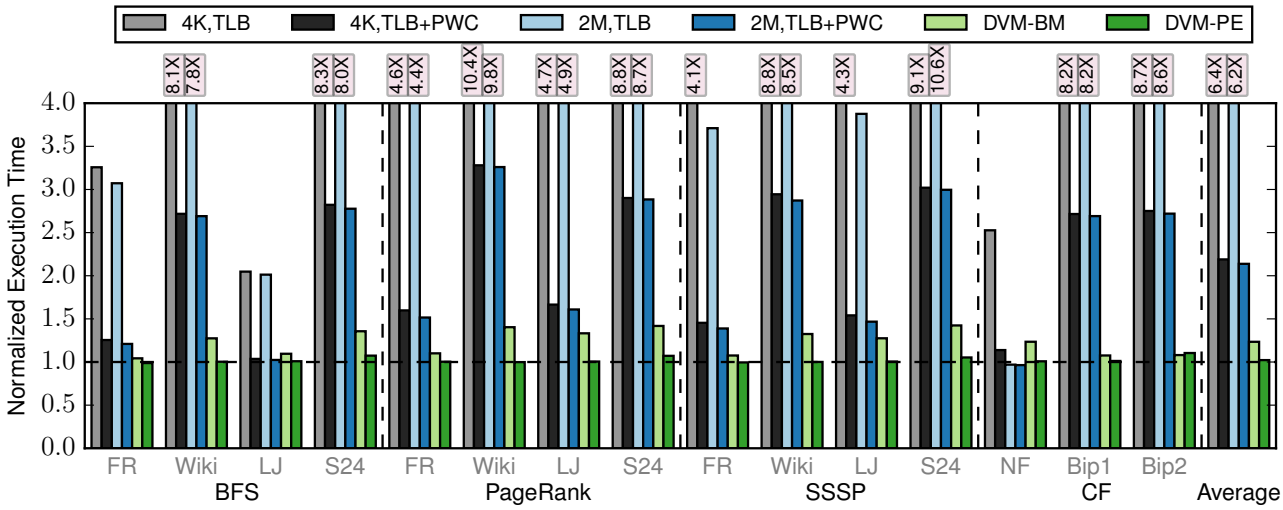

Figure 8. Execution time for accelerator workloads, normalized to runtime of ideal implementation.

6.3.1 Performance.

To evaluate the performance of DVM in accelerators, we compare the following systems having accelerators with:

(i) 128-entry TLB (4KB pages)

(ii) 128-entry TLB and 1KB PWC (4KB pages)

(iii) 128-entry TLB (2MB pages)

(iv) 128-entry TLB and 1KB PWC (2MB pages)

(v) DVM-BM, with a 2MB bitmap, 128-entry BM cache and 128-entry TLB

(vi) DVM-PE, with compact page tables and 1KB AVC (4KB pages)

(vii) Ideal (no translation or protection).

Figure 8 shows the execution time of our graph workloads for different input graphs for the above systems, normalized to the ideal implementation. Note that for some workloads, some systems perform slightly better than ideal due to delays in the interconnects and memory subsystem, which can vary depending on timing and other factors.

DVM-PE, with 1KB AVC and no TLB outperforms all other VM implementations and comes within 2% of the ideal system, which does not support VM. The performance improvements come from having smaller page table structures, which significantly improves the hit rate of the PWC. DVM-BM also outperforms the other VM implementations. Unfortunately, the hit rate of the BM cache is not as high as the AVC, due to the much larger size of the standard page table and use of 4KB pages instead of 128KB or larger regions.

VM, with 128-entry TLB is the slowest measured system with 4KB and 2MB pages. As seen in Figure 3, the irregular access patterns of our workloads result in high TLB miss rates. Moving to 2MB pages does not help much, as the TLB reach is still limited to 256 MB (128*2MB), which is smaller than the working set s of most of our workloads. NF has high TLB hit rates due to higher temporal locality of accesses. Being a bipartite graph, all its edges are directed from 480K users to only 18K movies. The small number of destination nodes results in high temporal locality. As a result, moving to 2MB pages exploits this locality showing near-ideal performance.

VM, with 128-entry TLB and 1KB PWC performs well for smaller graphs such as FR, NF and LJ. As 2MB pages need shorter page walks (only till L2PDEs), this results in slightly better performance than systems with 4KB pages. However, the larger size of the page tables limits the effectiveness of the small 1KB PWC. Overall, the PWC is critical to achieving high performance with a TLB, given the high cost of page walks. But, the combination of a TLB and PWC exhibit much high energy use, as described next.

6.3.2 Energy.

Energy is a first-order concern in modern systems, particularly for small accelerators. Here, we consider the impact of DVM on reducing the dynamic energy spent in MMU functions, like address translation for conventional VM and access validation for DVM. We calculate this dynamic energy by adding the energy of all TLB accesses, PWC accesses, and main memory accesses by the page table walker [34]. We show the dynamic energy consumption of VM implementations, normalized to the baseline system with TLB+PWC (4KB pages), in Figure 9.

DVM-PE offers a 76% reduction in dynamic translation energy over the baseline. This mainly comes from removing the fully associative TLB. Also, the high hit rate of the AVC reduces main memory accesses during a page walk, significantly decreasing the energy consumption.

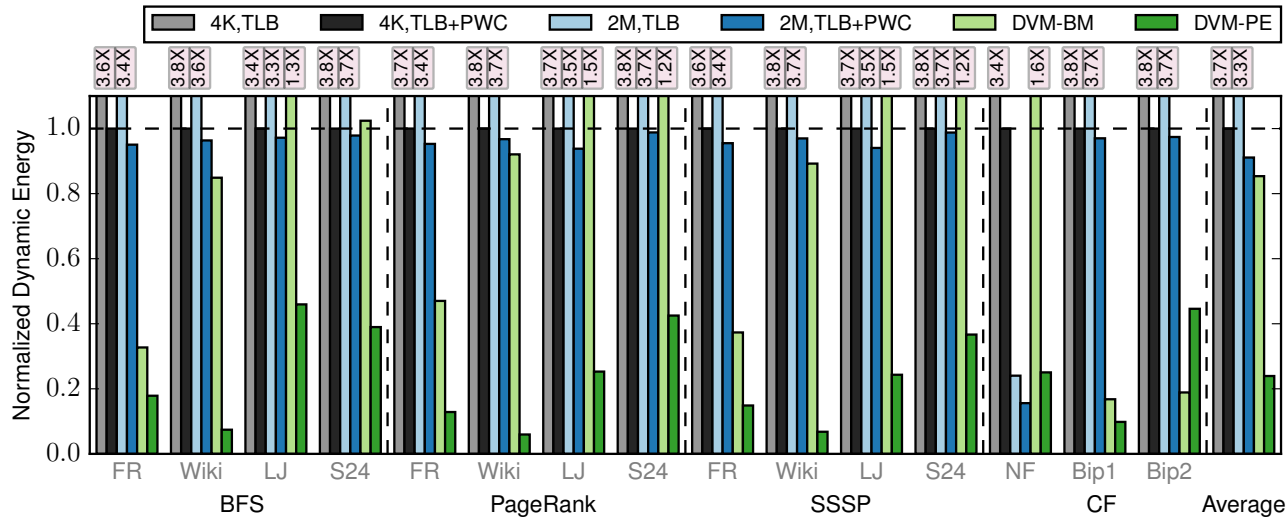

Figure 9. Dynamic energy spent in address translation/access validation, normalized to the 4KB, TLB+PWC implementation.

In systems without PWCs, each TLB miss results in four main-memory accesses with 4KB pages and three for 2MB pages. This coupled with the high miss rates results in extremely high energy consumption, 6X and 5.7X the baseline respectively. Even with PWCs, PWC misses incur the energy penalty of main memory accesses. Most importantly, each of these systems has a power-hungry 128-entry TLB, which is accessed on each memory access.

6.3.3 Identity Mapping.

To evaluate the risk of fragmentation with eager paging and identity mapping, we use the shbench benchmark from MicroQuill, Inc [30]. We configure this benchmark to continuously allocate memory of variable sizes until identity mapping fails to hold for an allocation (VA!=PA). Experiment 1 allocated small chunks of memory, sized between 100 and 10,000 bytes. Experiment 2 allocated larger chunks, sized between 100,000 and 10,000,000 bytes. Finally, we ran four concurrent instances of shbench, all allocating large chunks as in experiment 2. For each of these, we report the percentage of memory that could be allocated before identify mapping failed for systems with 16 GB, 32 GB and 64 GB of total memory capacity. We observe that 95 to 97% of memory can be allocated with identity mapping, even in memory-constrained systems with 16 GBs of memory. Our complete results are shown in Table 4.

	% Memory Allocated (PA==VA)		
System Memory	**Expt 1**	**Expt 2**	**Expt 3**
16 GB	96%	95%	96%
32 GB	97%	97%	96%
64 GB	97%	97%	97%

Table 4. Percentage of total system memory successfully allocated with identity mapping.

7. Towards DVM across Heterogeneous Systems

DVM can provide similar benefits for CPUs. The end of Dennard Scaling and the slowing of Moore's Law implies that future performance growth must look elsewhere. One alternative is specialization through heterogeneous systems, while another is to reduce waste everywhere. Towards the latter purpose, we discuss the use of DVM at CPU cores to reduce waste due to VM. This opportunity comes with CPU hardware and OS changes that are real, but more modest than we initially expected.

7.1 Hardware Changes for Processors

With DVM, CPUs can speculatively launch memory requests assuming PA==VA, in parallel with the actual address translation. For address translation, cDVM first checks conventional TLBs. Page table walks needed on TLB misses are expedited using the AVC.

Beyond the optimizations described for accelerators, the cDVM scheme can also optimize stores by exploiting the write-allocate policy of write-back caches. Under the write-allocate policy, a cacheline is first fetched from memory on a store missing in the cache. Subsequently, the store updates the cached location. cDVM speculatively performs the long-latency cacheline fetch in parallel with address translation, thus decreasing the latency of store operations.

7.2 OS Support for DVM in CPUs

Extending DVM system-wide involves more OS changes. For instance, a process running on a CPU fetches instructions from the code segment, and accesses data on its stack. Hence, we must modify the OS to identity map the stack and code segments as well.

Affected Feature	LOC changed
Code Segment	39
Heap Segment	1*
Memory-mapped Segment	56
Stack Segment	63
Page Tables	78
Miscellaneous	15

Table 5. Lines of code changed in Linux v4.10 split up by functionality. *Changes to memory-mapped segment affect heap segment, so we only count them once.

Beyond flexible location of the heap and memory-mapped regions for accelerator DVM, cDVM also requires flexibility in placing stack and code segments. We have implemented a prototype providing this flexibility in Linux kernel v4.10. The lines of code changed is shown in Table 5.

Stack. The base addresses for stacks are already randomized by ASLR. Except the main thread, every other thread in a multi-threaded process gets its own stack, which is allocated as a memory-mapped segment, which can leverage our previously described modifications.

The stack of the first thread is allocated within the kernel, and is used to setup initial arguments to launch the application. To minimize OS changes, we do not identity map this stack initially. Once the arguments are setup, but before control passes to the application, we move the stack to the VA matching its PA, which is not visible to the application.

Dynamically growing a region is difficult with identity mapping, as adjacent physical pages may not be available. Instead, we eagerly allocate an 8MB stack for all threads. This wastes some memory, but this can be adjusted. Stacks can also be grown above this size using through gcc's Split Stacks [55] when possible.

Code and globals. In unmodified Linux, the text segment (i.e., code) is located at a fixed offset near the bottom of the process address space, followed immediately by the data (initialized global variables) and the bss (uninitialized global variables) segments. To protect against return-oriented programming (ROP) attacks [49], OSes have begun to support position independent executables (PIE) which allow binaries to be loaded at random offsets from the base of the address space [48]. PIE incurs a small cost on function calls due to an added level of indirection.

PIE randomizes the base position of the text segment and keeps data and bss segments adjacent. We consider these segments as one logical entity in our prototype and allocate an identity-mapped segment equal to the combined size of these three segments. The permissions for the code region are then set to be Read-Execute, while the other two segments are to Read-Write.

7.3 Preliminary Experiments.

We perform preliminary evaluation of cDVM using a small set of memory intensive CPU-only applications like mcf from SPEC CPU 2006 [29], BT, CG from NAS Parallel Benchmarks [3], canneal from PARSEC [11] and xsbench [56].

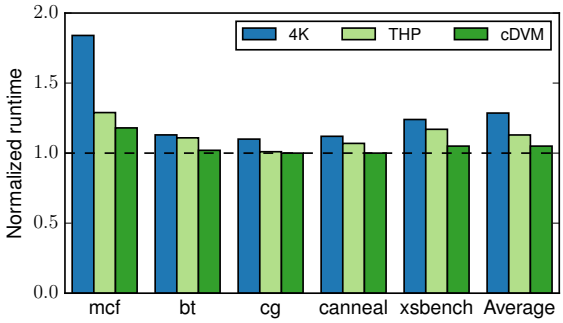

Figure 10. Runtime of CPU-only workloads, normalized to the ideal case.

Using hardware performance counters, we measure L2 TLB misses, page walk cycles and total execution cycles of these applications on an Intel Xeon E5-2430 machine with 96 GB memory, 64-entry L1 DTLB and 512-entry DTLB. Then, we use BadgerTrap [24] to instrument TLB misses and estimate the hit rate of the AVC. Finally, we use a simple analytical model to conservatively estimate the VM overheads under cDVM, like past work [5, 44, 35, 8, 18, 9]. For the ideal case, we estimate running time by subtracting page walk cycles for 2MB pages from total execution cycles.

We compare cDVM with conventional VM using 4 KB pages and 2MB pages using Transparent Huge Paging (THP). From our results in Figure 10, we see that conventional VM adds overheads of about 29% on average with 4KB pages and 13% with THP, even with a large two-level TLB hierarchy. THP improves performance by expanding TLB reach and shorter page walks. Due to the limits of our evaluation methodology, we can only estimate performance benefits of the AVC: we do not evaluate speculative data fetch in parallel to access validation nor our proposed cache design. Even so, cDVM reduces VM overheads to from 13% with 2MB pages to within 5% of the ideal implementation without address translation. The performance benefits come from the high hit rate of the AVC, due to our compact page table representation. Thus, we believe that cDVM merits more investigation to optimize systems with high VM overheads.

8. Related Work

Overheads of VM. The increasing overheads of supporting VM have been studied before for CPU workloads (e.g., direct segments [5]), and recently for accelerators (e.g., Cong et al. [16]).

Virtual Memory for Accelerators. Border Control (BC) [41] recognized the need for enforcing memory security in heterogeneous systems. BC provides mechanisms to checking permissions on physical addresses of requests leaving the

accelerator. However, BC does not aim to mitigate virtual memory overheads. Our DVM-BM implementation optimizes BC for fast access validation with DVM.

Most prior proposals have lowered virtual memory overheads for accelerators using changes in TLB location or hierarchy [16, 59]. For instance, two-level TLB structures in the IOMMU with page walks on the host CPU have been shown to reduce VM overheads to within 6.4% of ideal [16]. This design is similar to our evaluated implementations (iv). Our implementation uses large pages to improve TLB reach instead of a level 2 TLB as in the original proposal, and uses the IOMMU PWC. We see that TLBs are not very effective for workloads with irregular access patterns. Moreover, using TLBs greatly reduces the energy-efficiency of accelerators.

For throughput-oriented accelerators such as GPGPUs, memory request coalescers have been utilized to great effect in reducing the address translation traffic seen by TLBs [46, 45]. However, as discussed in Section 2, coalescers and TLBs are ineffective for applications with irregular memory accesses.

Address Translation for CPUs. Several address translation mechanisms have been proposed for CPUs, which could be extended to accelerators. Coalesced Large-Reach TLBs (CoLT) [44] uses eager paging to increase contiguity of memory allocations, and coalesces translation of adjacent pages into each TLB entries. However, address translation remains on the critical path of memory accesses. CoLT can be optimized further with identity mapping and DVM. Cooperative TLB prefetching [10] has been proposed to exploit correlations in translations across multicores. By sharing the AVC among the processing lanes of the accelerator, it already exploits any correlations among them.

Coalescing can also be performed for PTEs to increase PWC reach [8]. This can be applied directly to our proposed AVC design. However, due to our compact page table strucure, benefits will only be seen for workloads with much higher memory footprints. Furthermore, page table walks can be expedited with translation skipping [4]. Translation skipping does not increase the reach of the page table, and is less effective with DVM, as page table walks are not on the critical path for most accesses.

Direct Segments (DS) [5] is efficient but inflexible. It requires a monolithic, eagerly-mapped heap with uniform permissions, whose size is known at startup. On the other hand, DVM individually identity-maps heap allocations as they occur, helping mitigate fragmentation. RMM [35] are more flexible than DS, supporting heaps composed of multiple memory ranges. However, it requires power-hungry hardware (range-TLBs, range-table walkers in addition to TLBs and page-table walkers) thus being infeasible for accelerators, but could also be optimized with DVM.

9. Conclusion

Unified virtual memory is important for increasing the programmability of accelerators. We propose Devirtualized Virtual Memory (DVM) to minimize the performance and energy overheads of VM for accelerators. DVM enables almost direct access to PM while enforcing memory protection. DVM requires modest OS and IOMMU changes, and is transparent to applications. We also discuss ways to extend DVM throughout a heterogeneous system, to support both CPUs and accelerators with a single memory management scheme.

Acknowledgments

We thank Wisconsin Multifacet group, Arkaprava Basu and Dan Gibson for their feedback. This work was supported by the National Science Foundation under grants CCF-1533885, CCF-1617824 and John P. Morgridge Chair. Hill and Swift have significant financial interests in Google and Microsoft respectively.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[2] AMD. Amd i/o virtualization technology (iommu) specification, revision 3.00. `http://support.amd.com/TechDocs/48882_IOMMU.pdf`, December 2016.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks - summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, 1991.

[4] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[6] James Bennett and Stan Lanning. The netflix prize. In *KDD Cup and Workshop in conjunction with KDD*.

[7] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 2008.

[8] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM*

International Symposium on Microarchitecture, MICRO-46, 2013.

[9] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level tlbs for chip multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[10] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative tlb for chip multiprocessors. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), August 2011.

[13] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, 2004.

[14] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4), November 1994.

[15] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[16] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. Supporting address translation for accelerator-centric architectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 37–48, Feb 2017.

[17] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[18] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[19] Tim Davis. The university of florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices`.

[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, 1995.

[21] Hewlett Packard Enterprise. Persistent memory. `hpe.com/us/en/servers/persistent-memory.html`.

[22] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 283–295. IEEE, 2015.

[23] Michael Feldman. Darpa taps intel for graph analytics chip project. `DARPATapsIntelforGraphAnalyticsChipProject`, 2017.

[24] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *SIGARCH Comput. Archit. News*, 42(2), September 2014.

[25] Enes Goktas, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining information hiding (and what to do about it). In *USENIX Security Symposium*, 2016.

[26] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. *NDSS (Feb. 2017)*, 2017.

[27] Tae J. Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[28] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[29] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4), September 2006.

[30] MicroQuill Inc. Smartheap and smartheap mc. `http://microquill.com/smartheap/`, 2011.

[31] Intel. Intel optane technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`.

[32] intel. Intel virtualization technology for directed i/o, revision 2.4. `https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf`, 2016.

[33] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana

Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[34] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 631–643, March 2016.

[35] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[36] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, 2014.

[37] Hadi Asghari Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[38] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, 2007.

[39] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barret, and James A. Ang. Introducing the graph 500. In *Cray User's Group (CUG)*, 2010.

[40] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[41] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border control: Sandboxing accelerators. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 470–481, Dec 2015.

[42] Lena E. Olson, Simha Sethumadhavan, and Mark D. Hill. Security implications of third-party accelerators. *IEEE Comput. Archit. Lett.*, 15(1), January 2016.

[43] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4), January 2010.

[44] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, 2012.

[45] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[46] Jason Power, Mark D. Hill, and David A. Wood. Supporting x86-64 address translation for 100s of gpu lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 568–578, Feb 2014.

[47] Parthasarathy Ranganathan. From microprocessors to nanostores: Rethinking data-centric systems. *Computer*, 44(1):39–48, Jan 2011.

[48] RedHat. Position independent executables (pie). `https://access.redhat.com/blogs/766093/posts/1975793`, 2012.

[49] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), March 2012.

[50] Phil Rogers. The programmer's guide to the apu galaxy, 2011.

[51] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 2014.

[52] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, 2004.

[53] Kirill A. Shutemov. 5-level paging. `https://lwn.net/Articles/708526/`, January 2005.

[54] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, 1992.

[55] Ian Lance Taylor. Split stacks in gcc. `https://gcc.gnu.org/wiki/SplitStacks`, February 2011.

[56] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

[57] Arjan van de Ven. Linux patch for virtual address space randomization. `https://lwn.net/Articles/120966/`, January 2005.

[58] Oracle Vijay Tatkar. What is the sparc m7 data analytics accelerator? `https://community.oracle.com/docs/DOC-994842`, February 2016.

[59] Pirmin Vogel, Andrea Marongiu, and Luca Benini. Lightweight virtual memory support for many-core accelerators in heterogeneous embedded socs. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, CODES '15, 2015.

[60] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[61] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, 2015.