Devirtualizing Memory in Heterogeneous Systems

Swapnil Haria
University of Wisconsin-Madison
swapnilh@cs.wisc.edu

Mark D. Hill
University of Wisconsin-Madison
markhill@cs.wisc.edu

Michael M. Swift
University of Wisconsin-Madison
swift@cs.wisc.edu

Abstract

Accelerators are increasingly recognized as one of the major drivers of future computational growth. For accelerators, shared 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 more difficult to program.

We propose Devirtualized Memory (DVM) to combine the protection of VM with direct access to PM. By allocating memory such that physical and virtual addresses are almost always identical (VA==PA), DVM mostly replaces page-level address translation with faster region-level Devirtualized Access Validation (DAV). Optionally on read accesses, DAV can be overlapped with data fetch to hide VM overheads. DVM requires modest OS and IOMMU changes, and is transparent to the application.

Implemented in Linux 4.10, DVM reduces VM overheads in a graph-processing accelerator to just 1.6% on average. DVM also improves performance by 2.1X over an 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.

CCS Concepts • **Hardware → Hardware accelerators**; • **Software and its engineering → Virtual memory**;

Keywords accelerators; virtual memory.

ACM Reference Format:
Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3173194

ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173194

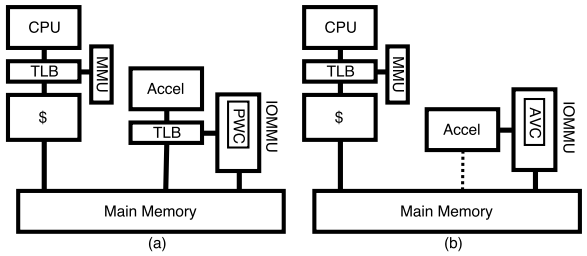

Figure 1. Heterogeneous systems with (a) conventional VM with translation on critical path and (b) DVM with Devirtualized Access Validation alongside direct access on reads.

1 Introduction

The end of Dennard Scaling and slowing of Moore's Law has weakened the future potential of general-purpose computing. To satiate the ever-increasing computational demands of society, research focus has intensified on heterogeneous systems having multiple special-purpose accelerators and conventional CPUs. In such systems, computations are offloaded by general-purpose cores to these accelerators.

Beyond existing accelerators like GPUs, accelerators for big-memory workloads with irregular access patterns are steadily gaining prominence [19]. In recent years, proposals for customized accelerators for graph processing [1, 25], data analytics [61, 62], and neural computing [15, 26] 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) [31] 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.

Ideally, accelerators want direct access to host physical memory to avoid address translation overheads, eliminate expensive data copying and facilitate fine-grained data sharing. This approach is simple to implement as it does not need large, power-hungry structures such as translation lookaside buffers (TLBs). Moreover, the low power and area consumption 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' semantics. This allows pointers to be dereferenced on both the CPU and the accelerator which 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, 32]. These overheads occur in processors with massive two-level TLBs and could be accentuated in accelerators with simpler translation hardware.

Fortunately, conditions that required VM in the past are changing. Previously, 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 [21, 29].

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 (VA==PA). As the PA for most accesses is identical to the VA, DVM replaces slow page-level address translation with faster region-level Devirtualized Access Validation (DAV). For DAV, the IO memory management unit (IOMMU) verifies that the process holds valid permissions for the access and that the access is to an identity-mapped page. Conventional address translation is still needed for accesses to non identity-mapped pages. Thus DVM also preserves the VM abstraction.

DAV can be optimized by exploiting 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. While DAV is still performed via hardware page walks, we introduce the Permission Entry (PE), which is a new page table entry format for storing coarse-grained permissions. PEs reduce DAV overheads in two ways. First, depending on the available contiguity, page walks can be shorter. Second, PEs significantly reduce the size of the overall page table thus improving the performance of page walk caches. DVM for accelerators is completely transparent to applications, and requires small OS changes to identity map memory allocations on the heap and construct PEs.

Furthermore, devirtualized memory can optionally 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.

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

- We propose DVM to minimize VM overheads, 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 format 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 2.1X faster 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 unified memory management throughout the heterogeneous system. cDVM lowers the overheads of VM in big-memory workloads to 5% for CPUs.

However, DVM does have some limitations. Identity Mapping allocates memory eagerly and contiguously (Section 4.3.1) which aggravates the problem of memory fragmentation, although we do not study this effect in this paper. Additionally, while copy-on-write (COW) and fork are supported by DVM, on the first write to a page, a copy is created which cannot be identity mapped, eschewing the benefits of DVM for that mapping. Thus, DVM is not as flexible as VM, but avoids most of the VM overheads. Finally, the Meltdown [37] and Spectre [34] design flaws became broadly known just as this paper was being finalized. One consequence is that future implementations of virtual memory, including DVM, may need to be careful about leaving detectable changes to micro-architecture state made during misspeculation, as these changes may be used as timing channels [35].

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 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 access to PM). Some accelerators use separate address spaces [31, 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; and (2) this approach makes it difficult to support pointer-is-a-pointer semantics, which reduces

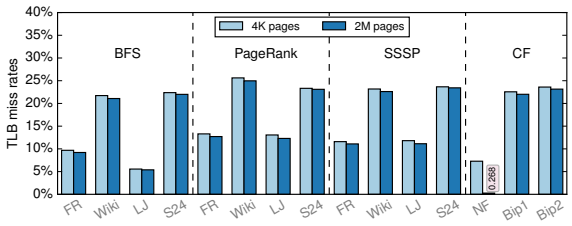

Figure 2. TLB miss rates for Graph Workloads with 128-entry TLB

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 IOMMU to service address translation requests from accelerators [2, 30], as illustrated in Figure 1. We focus on these systems, as address translation overheads severely degrade the performance of these accelerators [16].

For our graph workloads, we observe high TLB miss rates of 21% on average with a 128-entry TLB (Figure 2). There is little spatial locality and hence using larger 2MB pages improves the TLB miss rates only by 1% on average. TLB miss rates of about 30% have also been observed for GPU applications [45, 46]. While optimizations specific to GPU microarchitecture for TLB-awareness (e.g., cache-conscious warp scheduling) have been proposed to mitigate these overheads, these optimizations are not general enough to support efficient memory management in heterogeneous systems with multiple types of accelerators.

Some accelerators (e.g., Tesseract [1]) support simple address translation using a base-plus-offset scheme such as Direct Segments [5]. With this scheme, only memory within a single contiguous PM region can be shared, limiting its flexibility. Complicated address translation schemes such as range translations [32] are more flexible as they support multiple address ranges. However, they require large and power-hungry Range TLBs, 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, efficient, general and performant memory management approach for accelerators.

3 Devirtualizing Memory

In this section, we present the high-level design of our Devirtualized Memory (DVM) approach. Before discussing DVM, we enumerate the goals for a memory management approach 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. Towards this end, solutions should 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].

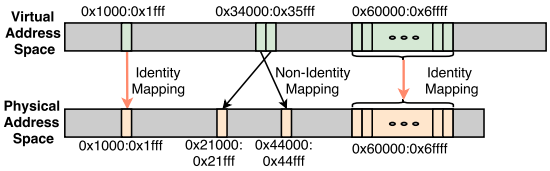

Figure 3. Address Space with Identity Mapped and Demand Paged Allocations.

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 particularly attractive when they offer large speedups under small resource budgets.

Flexibility. Memory management schemes 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 Memory

To minimize VM overheads, DVM introduces *Identity Mapping* and leverages permission validation [36, 60] in the form of *Devirtualized Access Validation*. Identity mapping allocates memory such that all VAs in the allocated region are identical to the backing PAs. DVM uses identity mapping for all heap allocations. Identity mapping can fail if no suitable address range is available in both the virtual and physical address spaces. In this case, DVM falls back to demand paging. Figure 3 illustrates an address space with identity mapping.

As PA==VA for most data on the heap, DVM can avoid address translation on most memory accesses. Instead, it is sufficient to verify that the accessed VA is identity mapped and that the application holds sufficient permissions for the access. We refer to these checks as Devirtualized Access Validation. In rare cases when PA!=VA, DAV fails and DVM resorts to address translation as in conventional VM.

Optionally on read accesses, DAV can be performed off the critical path. By predicting that the accessed VA is identity mapped, a premature load or *preload* is launched for the PA==VA of the access in parallel with DAV. In the common

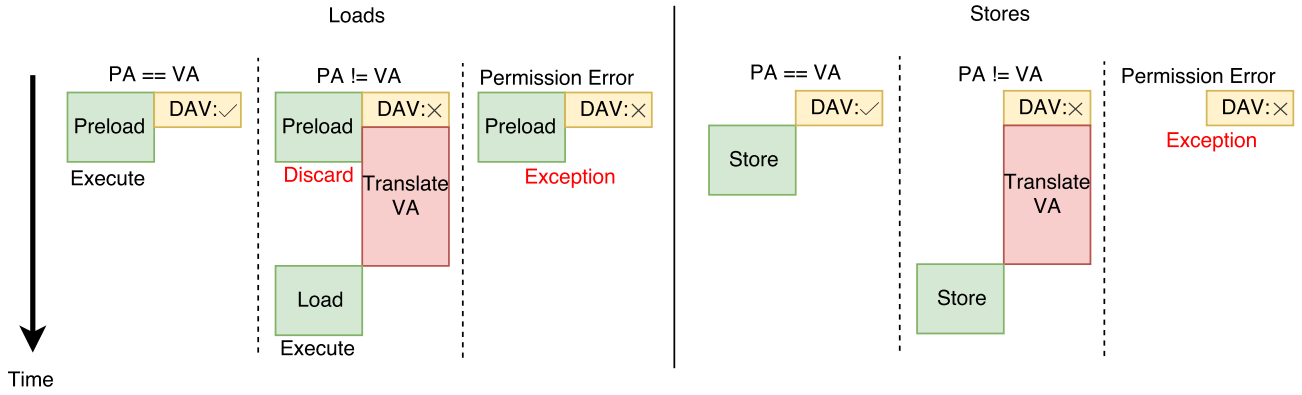

Figure 4. Memory Accesses in DVM

case (PA==VA), DAV succeeds and the preload is treated as the actual read access. If DAV fails, the preloaded value has to be discarded. Address translation is performed and a regular read access is launched to the translated PA. Memory accesses in DVM are illustrated in Figure 4.

DVM is designed to satisfy the goals listed earlier:

Programmability. DVM enables shared address space in heterogeneous systems at minimal cost, thus improving programmability of such systems.

Power/Performance. DVM optimizes for performance and power-efficiency by performing DAV much faster than full address translation. DAV latency is minimized by exploiting the contiguity of permissions for compact storage and efficient caching performance (Section 4.1.1). In case of loads, DVM can perform DAV in parallel with data preload, moving DAV off the critical path and offering immediate access to PM. Even in the rare case of an access to a non-identity mapped page, performance is no worse than conventional VM as DAV reduces the address translation latency, as explained in Section 4. However, additional power is consumed to launch and then squash the preload.

Flexibility. DVM facilitates page-level sharing between the accelerator and the host CPU since regions as small as a single page can be identity mapped independently, as shown in Figure 3. This allows DVM to benefit a variety of applications, including those that do not have a single contiguous heap. Furthermore, DVM is transparent to most applications.

Safety. DVM completely preserves conventional virtual memory protection as all accesses are still checked for valid permissions. If appropriate permissions are not present for an access, an exception is raised on the host CPU.

4 Implementing DVM for Accelerators

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

First, we describe page table improvements and hardware mechanisms for fast DAV. Next, we show how DAV overheads can be minimized further for reads by overlapping it with preload. Finally, we discuss OS modifications to support identity mapping. Here, we use the term *memory region* to mean 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.

4.1 Devirtualized Access Validation

We support DAV with compact page tables and an access validation cache. We assume that the IOMMU uses separate page tables to avoid affecting CPU hardware. We use the following 2-bit encoding for permissions—00:No Permission, 01:Read-Only, 10:Read-Write and 11:Read-Execute.

4.1.1 Compact Page Tables

We leverage available contiguity in permissions to store them at a coarse granularity resulting in a compact page table structure. Figure 5 shows an x86-64 page table. An L2 Page Directory entry (L2PDE) ① maps a contiguous 2MB VA range ③. Physical Page Numbers are stored for each 4K page in this range, needing 512 L1 page table entries (PTEs) ② and 4KB of memory. However, if pages are identity mapped, PAs are already known and only permissions need to be stored. If permissions are the same for the entire 2MB region (or an aligned sub-region), these could be stored at the L2 level. For larger regions, permissions can be stored at the L3 and L4 levels. For new 5-level page tables, permissions can also be stored at the L5 levels.

We introduce a new type of leaf PTE called the Permissions Entry (PE), shown in Figure 6. PEs are direct replacements for regular PTEs at any level, with the same size (8 bytes) and mapping the same VA range as the replaced PTE. PEs contain sixteen permission fields, currently 2-bit each. A permission

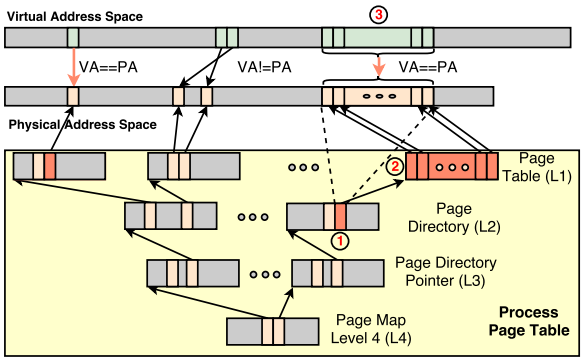

Figure 5. 4-level Address Translation in x86-64

entry bit is added to all PTEs, and is 1 for PEs and 0 for other regular PTEs.

Each PE records separate permissions for sixteen aligned regions 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, an L2PE maps a 2MB VA range of sixteen 128KB (=2MB/16) regions aligned on 128KB address boundaries. An L3PE maps a 1GB VA range of sixteen 64MB regions aligned on 64MB address boundaries. Other intermediate sizes can be handled simply by replicating permissions. Thus a 1MB region is mapped by storing permissions across 8 permission fields in an L2PE. Region ③ in Figure 5 can be mapped by an L2PE with uniform permissions stored in all 16 fields.

PEs implicitly guarantee that any allocated memory in the mapped VA range is identity-mapped. Unallocated memory i.e., gaps in the mapped VA range can also be handled gracefully, if aligned suitably, by treating them as regions with no permissions (00). This frees the underlying PAs to be re-used for non-identity mapping in the same or other applications or for identity mappings in other applications. 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 DAV 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, the IOMMU may raise an exception on the host CPU.

If a page walk encounters a leaf PTE, the accessed VA may not be identity mapped. In this case, the leaf PTE is used to perform address translation i.e., use the page frame number recorded in the PTE to generate the actual PA. This avoids a separate walk of the page table to translate the address. More importantly, this ensures that even in the fallback case (PA!=VA), the overhead (i.e., full page walk) is no worse than conventional VM.

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 for PageRank and CF. PEs reduce the page table size by eliminating most L1PTEs.

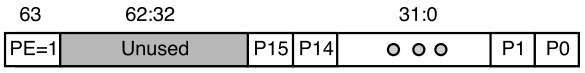

Figure 6. Structure of a Permission Entry. PE: Permission Entry, P15-P0: Permissions.

Incorporating PEs significantly reduces the size of page tables (Table 1) as each higher-level (L2-L4) PE directly replaces an entire sub-tree of the page table. For instance, replacing an L3PTE with a PE eliminates 512 L2PDEs and up to 512×512 L1PTEs, saving as much as 2.05 MB. Most of the benefits come from eliminating L1PTEs as these leaf PTEs comprise about 98% of the size of the page tables. Thus, PEs make page tables more compact.

Alternatives. Page table changes can be minimized by using existing unused bits in PTEs instead of adding PEs. For instance, using 8 out of the 9 unused bits in L2PTEs provides four 512KB regions. Similarly, 16 out of 18 free bits in L3PTEs can support eight 128MB regions. DAV latency can also be traded for space by using flat permission bitmaps for the entire virtual address space as in Border Control [41].

4.1.2 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 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 both TLBs and PWCs for accelerators. The AVC is a standard 4-way set-associative cache with 64B blocks. The AVC caches 128 distinct PTEs, resulting in a total capacity of 1 KB. It is physically-indexed and physically tagged cache, as page table walks use physical addresses. For PEs, this provides 128 sets of permissions. The AVC does not support translation skipping [4].

On every memory reference by an accelerator, the IOMMU walks the page table using the AVC. In the best case, page walks require 2-4 AVC accesses and no main memory access. 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.

Due to the smaller page tables, even a small 128-entry (1KB) AVC has very high hit rates, resulting in fast access validation. As the hardware design is similar to conventional PWCs, the AVC is just as energy-efficient. Moreover, the AVC is more energy-efficient than a comparably sized, fully associative (FA) TLB due to a less associative lookup.

4.2 Preload on Reads

If an accelerator supports the ability to squash and retry an inflight load, DVM allows a preload to occur in parallel with DAV. As a result, the validation latency for loads can be overlapped with the memory access latency. If the access is validated successfully, the preload is treated as the actual memory access. Otherwise, it is discarded, and the access is retried to the correct, translated PA. For stores, this optimization is not possible because the physical address must be validated before the store updates memory.

4.3 Identity Mapping

As accelerators typically only access shared data on the heap, we implement identity mapping only for heap allocations, requiring minor OS changes. The application's heap is actually composed of the heap segment (for smaller allocations) as well as memory-mapped segments (for larger allocations).

To ensure VA==PA for most addresses in memory, firstly, physical frames (and thus PAs) need to be reserved at the time of memory allocation. For this, we use *eager paging* [32]. Next, the allocation is mapped into the virtual address space at VAs equal to the backing PAs. This may result in heap allocations being mapped anywhere in the process address space as opposed to a hardcoded location. To handle this, we add support for a flexible address space. Below, we describe our implementation in Linux 4.10. Figure 7 shows the pseudocode for identity mapping.

4.3.1 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 VA==PA for all the constituent pages. We use the eager paging modifications to Linux's default buddy allocator developed by others [32] to allocate contiguous powers-of-two

```
Memory-Allocation (Size S)
PA ← contiguous-PM-allocation(S)
if PA ≠ NULL then
    VA ← VM-allocation(S)
    Move region to new VA_2 equal to PA
    if Move succeeds then
        return VA_2 // Identity-Mapped
    end
    else
        Free-PM(PA, S)
        return VA // Fallback to Demand-Paging
    end
end
else
    VA ← VM-allocation(S)
    return VA // Fallback to Demand-Paging
end
```

Figure 7. Pseudocode for Identity Mapping

pages. Once contiguous pages are obtained, additional pages obtained due to rounding up are returned immediately. Eager allocation can increase physical memory use if programs allocate much more memory then they actually use.

4.3.2 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. We modify the default behavior of glibc `malloc` to always use the `mmap` system call instead of `brk`. This is because identity mapped regions cannot be grown easily, and `brk` requires dynamically growing a region. 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 noncontiguous 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 allow any possible positions of the heap and memory-mapped 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 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

Here we address potential concerns regarding DVM.

Security implications. While DVM sets PA==VA in the common case, this does not weaken isolation. Just because applications can address all of PM 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. However, with a cache, preloads could be vulnerable to the Meltdown exploit [37], so this optimization could be disabled.

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. DVM gets randomness from physical addresses, which may have fewer bits, such as 12 bits for 2MB-aligned allocation in 8GB physical address space. However even the stronger Linux randomization has already been derandomized by software [23, 52] and hardware-based attacks [24]. 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 for virtualized environments as well. The overheads of conventional virtual memory are exacerbated in such environments [7] 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).

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 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, supporting multiple page sizes is difficult [17, 54], particularly with set associative TLBs which are commonly used due to their power-efficiency. On the other hand, PEs at higher levels of the page table allow DVM to gracefully scale with increasing memory sizes.

Finally, huge page TLB performance still depends on the locality of memory references. TLB performance can be an issue for big-memory workloads with irregular or streaming accesses [43, 47], as shown in Figure 2. In comparison, DVM exploits the locality in permissions which is found in most applications due to how memory is typically allocated.

6 Evaluation

6.1 Methodology

We quantitatively evaluate DVM using a heterogeneous system containing an out-of-order core and the Graphicionado graph-processing accelerator [25]. Graphicionado is 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. Also, 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. Thus a graph algorithm is expressed as operations on a single vertex and its edges. Most graph algorithms can be specified and executed on Graphicionado with three custom functions, namely `processEdge`, `reduce` and `apply`. The graph is stored as a list of edges, each in the form of a 3-tuple (srcid, dstid, weight). A list of vertices is maintained 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 with 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 the shared 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 6.5 [38] and access counts from our gem5 simulation.

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	Heap Size
Flickr (FR) [20]	0.82M	9.84M	288 MB
Wikipedia (Wiki) [20]	3.56M	84.75M	1.26 GB
LiveJournal (LJ) [20]	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

6.2 Workloads

We run four common graph algorithms on the 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

We evaluate seven separate implementations. We evaluate conventional VM implementations using an IOMMU with 128-entry fully associative (FA) TLB and 1KB PWC. We show the performance with three page sizes—4KB, 2MB and 1GB. Next, we evaluate three DVM implementations with different DAV hardware. First, we store permissions for all VAs in a flat 2MB bitmap in memory for 1-step DAV, with a separate 128-entry cache for caching bitmap entries (*DVM-BM*). 2-bit permissions are stored for all identity-mapped pages in the application's heap for fast access validation. If no permissions (i.e., 00) are found, full address translation is performed, expedited by a 128-entry FA TLB. Second, we implement DAV using page tables modified to use PEs and a 128-entry (1KB) AVC (*DVM-PE*). Third, we extend *DVM-PE* by allowing preload on reads (*DVM-PE+*). Finally, we evaluate an ideal implementation in which the accelerator directly accesses physical memory without any translation or protection checks.

6.3.1 Performance

Figure 8 shows the execution time of our graph workloads for different input graphs for the above systems, normalized to the ideal implementation.

DVM-PE outperforms most other VM implementations with only 3.5% overheads. Preload support in *DVM-PE+* further reduces DVM overheads to only 1.7%. The performance improvements come from being able to complete most page walks entirely from the AVC without any memory references. Conventional PWCs typically avoid caching L1PTEs to prevent cache pollution, so page walks for 4K pages require at least one memory reference.

DVM-BM incurs 23% DVM overheads, much lower than most other VM implementations but greater than the other DVM variants. Unfortunately, the hit rate of the BM cache

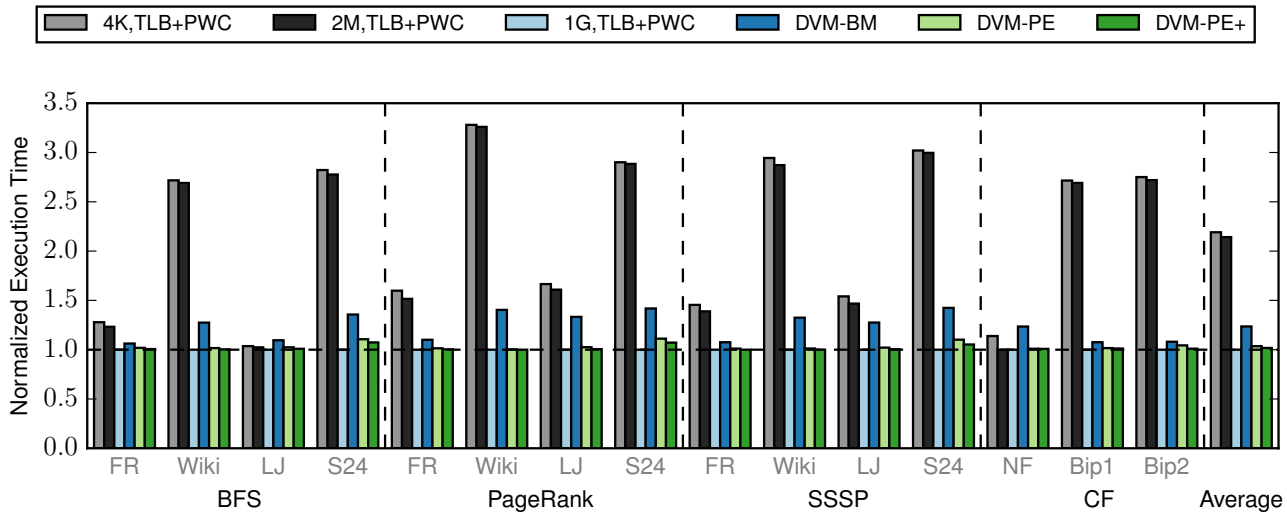

Figure 8. Execution time for accelerator workloads, normalized to runtime of ideal implementation.

is not as high as the AVC, due to the much larger size of standard page tables and use of 4KB pages instead of 128KB or larger regions.

4K,TLB+PWC and *2M,TLB+PWC* have high VM overheads, on average 119% and 114% respectively. As seen in Figure 2, the irregular access patterns of our workloads result in high TLB miss rates. Using 2MB pages does not help much, as the TLB reach is still limited to 256 MB (128*2MB), which is smaller than the working sets 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.

1G,TLB+PWC also performs well—virtually no VM overhead—for the workloads and system that we evaluate, but with three issues. First, <10 1GB pages are sufficient to map these workloads, but not necessarily for future workloads. Second, there are known OS challenges for managing 1GB pages. Third, the 128-entry fully associatve TLB we assume is power-hungry and often avoided in industrial designs (e.g., Intel currently uses four-way set associative).

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 memory accesses by the page table walker [33]. We show the dynamic energy consumption of VM implementations, normalized to *4K,TLB+PWC* in Figure 9.

DVM-PE offers 76% reduction in dynamic translation energy over the baseline. This mainly comes from removing the FA TLB. Also, page walks can be entirely serviced with AVC cache accesses without any main memory accesses, significantly decreasing the energy consumption. Memory accesses for discarded preloads for non-identity mapped pages increase dynamic energy slightly in *DVM-PE+*.

DVM-BM shows a 15% energy reduction over the baseline. Energy consumption is higher than other DVM variants due to memory references on bitmap cache misses.

1G,TLB+PWC shows low energy consumption due to lack of TLB misses.

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 [28]. 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.

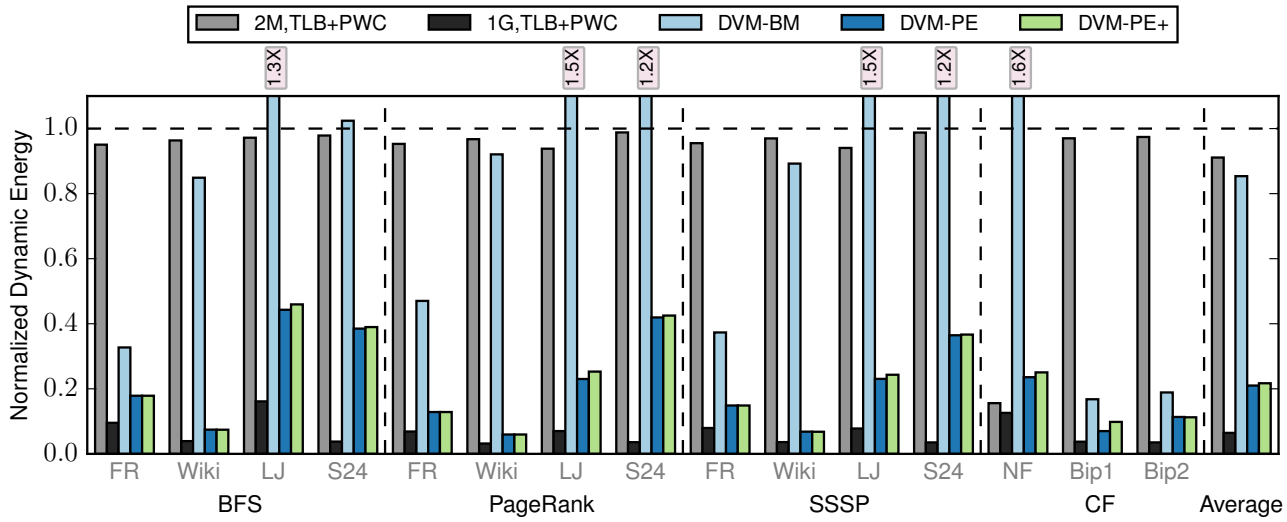

Figure 9. Dynamic energy spent in address translation/access validation, normalized to the 4KB, TLB+PWC implementation.

	% 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 (cDVM). With the end of Dennard Scaling and the slowing of Moore's Law, one avenue for future performance growth is to reduce waste everywhere. Towards this end, we discuss the use of DVM for 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

In addition to the DVM benefits described for accelerators, the cDVM approach 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. If speculation is found incorrect on DAV, implementations may wish to undo micro-architectural changes to mitigate timing channel exploits [34, 35, 37].

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

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

7.2 OS Support for DVM in CPUs

The simplest way to extend to CPUs is to enable the OS VM and CPU page table walkers to handle the new compact page tables with PEs. Next, we can optionally extend to code and/or stack, but typically heap is much larger than other segments. We have implemented a prototype providing this flexibility in Linux v4.10. The lines of code changed is shown in Table 5.

Stack. The base addresses for stacks are already randomized by ASLR. The stack of the main thread is allocated by 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.

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 be grown above this size using gcc's Split Stacks [55].

The stacks of other threads beside the main thread in a multi-threaded process are allocated as memory-mapped segments, and can be handled as discussed previously.

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 Performance Evaluation

We evaluate the performance benefits of cDVM using memory intensive CPU-only applications like mcf from SPEC CPU 2006 [27], BT, CG from NAS Parallel Benchmarks [3], canneal from PARSEC [11] and xsbench [56].

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 [22] 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, 8, 9, 18, 32, 44]. 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 4KB pages and 2MB pages with Transparent Huge Paging (THP). From our results in Figure 10, we see that conventional VM adds about 29% overheads on average with 4KB pages and 13% with THP, even with a two-level TLB hierarchy. THP improves performance by expanding TLB reach and shortening page walks. Due to the limits of our evaluation methodology, we can only estimate performance benefits of the AVC: we do not implement preloads. Even so, cDVM reduces VM overheads from 13% with 2MB pages to within 5% of the ideal implementation without address translation. The performance benefits come from shorter page walks with fewer memory accesses. Thus, we believe that cDVM merits more investigation to optimize systems with high VM overheads.

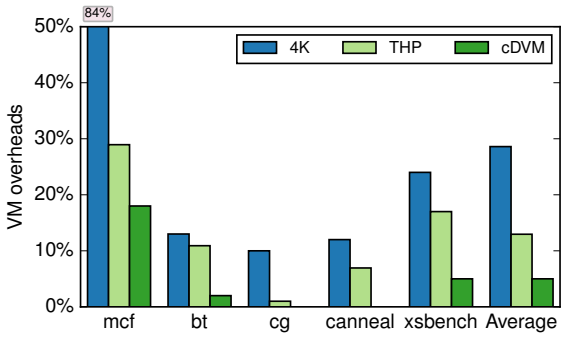

Figure 10. Runtime of CPU-only workloads, normalized to the ideal case.

8 Related Work

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

VM 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 2M,TLB+PWC implementation which 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 increases the energy use of accelerators.

Particularly for GPGPUs, microarchitecture-specific optimizations such as coalescers have been effective in reducing the address translation overheads [45, 46]. However, these techniques cannot be easily extended for other accelerators.

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] use 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. The AVC exploits any correlations among the processing lanes of the accelerator.

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 structure, 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] are efficient but inflexible. Using DS 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 [32] 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

Shared memory is important for increasing the programmability of accelerators. We propose Devirtualized 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 approach.

Acknowledgments

We thank the 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. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*.

[2] AMD. 2016. AMD I/O Virtualization Technology (IOMMU) Specification, Revision 3.00. http://support.amd.com/TechDocs/48882_IOMMU.pdf. (Dec. 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. 1991. The NAS Parallel Benchmarks - Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC)*.

[4] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*.

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*.

[6] James Bennett and Stan Lanning. 2017. The Netflix Prize. In *KDD Cup and Workshop in conjunction with KDD, CA*.

[7] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. 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)*.

[8] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[9] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*.

[10] Abhishek Bhattacharjee and Margaret Martonosi. 2010. 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)*.

[11] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[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. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011).

[13] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SIAM International Conference on Data Mining*. http://www.cs.cmu.edu/~christos/PUBLICATIONS/siam04.pdf

[14] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. 1994. Sharing and Protection in a Single-address-space Operating System. *ACM Trans. Comput. Syst.* 12, 4 (Nov. 1994).

[15] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*.

[16] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2017.19

[17] Guilherme Cox and Abhishek Bhattacharjee. 2017. 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)*.

[18] Guilherme Cox and Abhishek Bhattacharjee. 2017. 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)*.

[19] DARPA. 2017. Extracting Insight from the Data Deluge Is a Hard-to-Do Must-Do. https://www.darpa.mil/news-events/2017-06-02. (2017).

[20] Tim Davis. [n. d.]. The university of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices. ([n. d.]).

[21] Hewlett Packard Enterprise. [n. d.]. Persistent Memory. hpe.com/us/en/servers/persistent-memory.html. ([n. d.]).

[22] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *SIGARCH Comput. Archit. News* 42, 2 (Sept. 2014).

[23] Enes Goktas, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Information Hiding (and What to Do about It). In *USENIX Security Symposium*.

[24] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. *Network and Distributed System Security Symposium (NDSS)* (2017).

[25] Tae J. Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[26] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*.

[27] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006).

[28] MicroQuill Inc. 2011. SmartHeap and SmartHeap MC. http://microquill.com/smartheap/. (2011).

[29] Intel. [n. d.]. Intel Optane technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html. ([n. d.]).

[30] intel. 2016. IntelÂő 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).

[31] 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. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*.

[32] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*.

[33] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. 2016. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 631–643. https://doi.org/10.1109/HPCA.2016.7446100

[34] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2017. Spectre Attacks: Exploiting Speculative Execution. https://spectreattack.com/spectre.pdf. (2017).

[35] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*. http://dl.acm.org/citation.cfm?id=646761.706156

[36] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. 1992. Architecture Support for Single Address Space Operating Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA. https://doi.org/10.1145/143365.143508

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2017. Meltdown. https://meltdownattack.com/meltdown.pdf. (2017).

[38] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. 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)*.

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

[40] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*.

[41] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. 2015. Border control: Sandboxing accelerators. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 470–481. https://doi.org/10.1145/2830772.2830819

[42] Lena E. Olson, Simha Sethumadhavan, and Mark D. Hill. 2016. Security Implications of Third-Party Accelerators. *IEEE Comput. Archit. Lett.* 15, 1 (Jan. 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. 2010. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.* 43, 4 (Jan. 2010).

[44] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[45] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. 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)*.

[46] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 568–578. https://doi.org/10.1109/HPCA.2014.6835965

[47] Parthasarathy Ranganathan. 2011. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *Computer* (Jan 2011). https://doi.org/10.1109/MC.2011.18

[48] RedHat. 2012. Position Independent Executables (PIE). https://access.redhat.com/blogs/766093/posts/1975793. (2012).

[49] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, Article 2 (March 2012).

[50] Phil Rogers. 2011. 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. 2014. 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)*.

[52] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*.

[53] Kirill A. Shutemov. 2005. 5-level paging. https://lwn.net/Articles/708526/. (Jan. 2005).

[54] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*.

[55] Ian Lance Taylor. 2011. Split Stacks in GCC. https://gcc.gnu.org/wiki/SplitStacks. (Feb. 2011).

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

[57] Arjan van de Ven. 2005. Linux patch for virtual address space randomization. https://lwn.net/Articles/120966/. (Jan. 2005).

[58] Oracle Vijay Tatkar. 2016. What Is the SPARC M7 Data Analytics Accelerator? https://community.oracle.com/docs/DOC-994842. (Feb. 2016).

[59] Pirmin Vogel, Andrea Marongiu, and Luca Benini. 2015. 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)*.

[60] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[61] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. 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)*.

[62] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. 2015. Beyond the Wall: Near-Data Processing for Databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware (DAMON)*. Article 2.