The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework

Nastaran Hajinazar[⋆†] Pratyush Patel[⋈] Minesh Patel[⋆] Konstantinos Kanellopoulos[⋆] Saugata Ghose[‡]
Rachata Ausavarungnirun[⊙] Geraldo F. Oliveira[⋆] Jonathan Appavoo[◇] Vivek Seshadri[▽] Onur Mutlu[⋆‡]

[⋆]*ETH Zürich* [†]*Simon Fraser University* [⋈]*University of Washington* [‡]*Carnegie Mellon University*
[⊙]*King Mongkut's University of Technology North Bangkok* [◇]*Boston University* [▽]*Microsoft Research India*

Computers continue to diversify with respect to system designs, emerging memory technologies, and application memory demands. Unfortunately, continually adapting the conventional virtual memory framework to each possible system configuration is challenging, and often results in performance loss or requires non-trivial workarounds.

To address these challenges, we propose a new virtual memory framework, the Virtual Block Interface (VBI). We design VBI based on the key idea that delegating memory management duties to hardware can reduce the overheads and software complexity associated with virtual memory. VBI introduces a set of variable-sized virtual blocks (VBs) to applications. Each VB is a contiguous region of the globally-visible VBI address space, *and an application can allocate each semantically meaningful unit of information (e.g., a data structure) in a separate VB. VBI decouples access protection from memory allocation and address translation. While the OS controls which programs have access to which VBs, dedicated hardware in the memory controller manages the physical memory allocation and address translation of the VBs. This approach enables several architectural optimizations to (1) efficiently and flexibly cater to different and increasingly diverse system configurations, and (2) eliminate key inefficiencies of conventional virtual memory.*

We demonstrate the benefits of VBI with two important use cases: (1) reducing the overheads of address translation (for both native execution and virtual machine environments), as VBI reduces the number of translation requests and associated memory accesses; and (2) two heterogeneous main memory architectures, where VBI increases the effectiveness of managing fast memory regions. For both cases, VBI significantly improves performance over conventional virtual memory.

1. Introduction

Virtual memory is a core component of modern computing systems [28, 38, 63]. Virtual memory was originally designed for systems whose memory hierarchy fit a simple two-level model of small-but-fast main memory that can be directly accessed via CPU instructions and large-but-slow external storage accessed with the help of the operating system (OS). In such a configuration, the OS can easily abstract away the underlying memory architecture details and present applications with a unified view of memory.

However, continuing to efficiently support the conventional virtual memory framework requires significant effort due to (1) emerging memory technologies (e.g., DRAM–NVM hybrid memories), (2) diverse system architectures, and (3) diverse memory requirements of modern applications. The OS must now efficiently meet the wide range of application memory requirements that leverage the advantages offered by emerging memory architectures and new system designs while simultaneously hiding the complexity of the underlying memory and system architecture from the applications. Unfortunately, this is a difficult problem to tackle in a generalized manner. We describe three examples of challenges that arise when adapting conventional virtual memory frameworks to today's diverse system configurations.

Virtualized Environments. In a virtual machine, the guest OS performs virtual memory management on the emulated "physical memory" while the host OS performs a second round of memory management to map the emulated physical memory to the actual physical memory. This extra level of indirection results in three problems: (1) two-dimensional page walks [14, 39, 40, 85, 99, 112], where the number of memory accesses required to serve a TLB miss increases dramatically (e.g., up to 24 accesses in x86-64 with 4-level page tables); (2) performance loss in case of miscoordination between the guest and host OS mapping and allocation mechanisms (e.g., when the guest supports superpages, but the host does not); and (3) inefficiency in virtualizing increasingly complex physical memory architectures (e.g., hybrid memory systems) for the guest OS. These problems worsen with more page table levels [53], and in systems that support nested virtualization (i.e., a virtual machine running inside another) [36, 43].

Address Translation. In existing virtual memory frameworks, the OS manages virtual-to-physical address mapping. However, the hardware must be able to traverse these mappings to handle memory access operations (e.g., TLB lookups). This arrangement requires using *rigid* address-translation structures that are shared between and understood by both the hardware and the OS. Prior works show that many applications can benefit from flexible page tables, which cater to the application's actual memory footprint and access patterns [4, 11, 33, 58]. Unfortunately, enabling such flexibility in conventional virtual memory frameworks requires more complex address translation structures *every time* a new address translation approach is proposed. For example, a recent work [11] proposes using direct segments to accelerate big-memory applications. However, in order to support direct segments, the virtual memory contract needs to change to enable the OS to specify which regions of memory are directly mapped to physical memory. Despite the potential performance benefits, this approach is not easily scalable to today's increasingly diverse system architectures.

Memory Heterogeneity. Prior works propose many performance-enhancing techniques that require (1) dynamically *mapping* data to different physical memory regions according to application requirements (e.g., mapping frequently-accessed data to fast memory), and (2) *migrating* data when those requirements change (e.g., [22, 23, 29, 57, 64, 73, 74, 78, 80, 82, 103, 106–108, 124, 134, 138]). Efficiently implementing such functionality faces two challenges. First, a customized data mapping requires the OS to be aware of microarchitectural properties of the underlying memory. Second, even if this can

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00089

be achieved, the OS has low visibility into rich fine-grained runtime memory behavior information (e.g., access pattern, memory bandwidth availability), especially at the main memory level. While hardware has access to such fine-grained information, informing the OS *frequently enough* such that it can react to changes in the memory behavior of an application in a *timely* manner is challenging [86, 107, 122, 128, 135].

A wide body of research (e.g., [1, 2, 4, 7–11, 14–17, 20, 21, 26, 30, 39–42, 45, 47, 51, 52, 59–61, 68, 70, 71, 76, 77, 86, 87, 95, 97–99, 101, 102, 104, 105, 110–112, 121–123, 127, 133, 139]) proposes mechanisms to alleviate the overheads of conventional memory allocation and address translation by exploiting specific trends observed in modern systems (e.g., the behavior of emerging applications). Despite notable improvements, these solutions have two major shortcomings. First, these solutions mainly exploit specific system or workload characteristics and, thus, are applicable to a limited set of problems or applications. Second, each solution requires specialized and not necessarily compatible changes to both the OS and hardware. Therefore, implementing all of these proposals at the same time in a system is a daunting prospect.

Our goal in this work is to design *a general-purpose alternative virtual memory framework that naturally supports and better extracts performance from a wide variety of new system configurations, while still providing the key features of conventional virtual memory frameworks.* To this end, we propose the Virtual Block Interface (VBI), an alternative approach to memory virtualization that is inspired by the logical block abstraction used by solid-state drives to hide the underlying device details from the rest of the system. In a similar way, we envision the memory controller as the primary provider of an abstract interface that hides the details of the underlying physical memory architecture, including the physical addresses of the memory locations.

VBI is based on three guiding principles. First, *programs should be allowed to choose the* size *of their virtual address space*, to mitigate translation overheads associated with very large virtual address spaces. Second, *address translation should be decoupled from memory protection*, since they are logically separate and need not be managed at the same granularity by the same structures. Third, *software should be allowed to communicate semantic information about application data to the hardware*, so that the hardware can more intelligently manage the underlying hardware resources.

VBI introduces a *globally-visible* address space called the *VBI Address Space*, that consists of a large set of *virtual blocks (VBs)* of different sizes. For any semantically meaningful unit of information (e.g., a data structure, a shared library), the program can choose a VB of appropriate size, and tag the VB with properties that describe the contents of the VB. **The key idea** of VBI is to delegate physical memory allocation and address translation to a hardware-based Memory Translation Layer (MTL) at the memory controller. This idea is enabled by the fact that the globally-visible VBI address space provides VBI with system-wide unique *VBI addresses* that can be *directly* used by on-chip caches without requiring address translation. In VBI, the OS no longer needs to manage address translation and memory allocation for the physical memory devices. Instead, the OS (1) retains full control over access protection by controlling which programs have access to which virtual blocks, and (2) uses VB properties to communicate the data's memory requirements (e.g., latency sensitivity) and characteristics (e.g., access pattern) to the memory controller.

Figure 1 illustrates the differences between virtual memory management in state-of-the-art production Intel x86-64 systems [54] and in VBI. In x86-64 (Figure 1a), the OS manages a single private virtual address space (VAS) for each process (①), providing each process with a fixed-size 256 TB VAS irrespective of the actual memory requirements of the process (②). The OS uses a set of page tables, one per process, to define how each VAS maps to physical memory (③). In contrast, VBI (Figure 1b) makes *all* virtual blocks (VBs) visible to *all* processes, and the OS controls which processes can access which VBs (❶). Therefore, a process' total virtual address space is defined by which VBs are attached to it, i.e., by the process' actual memory needs (❷). In VBI, the MTL has full control over mapping of data from each VB to physical memory, invisibly to the system software (❸).

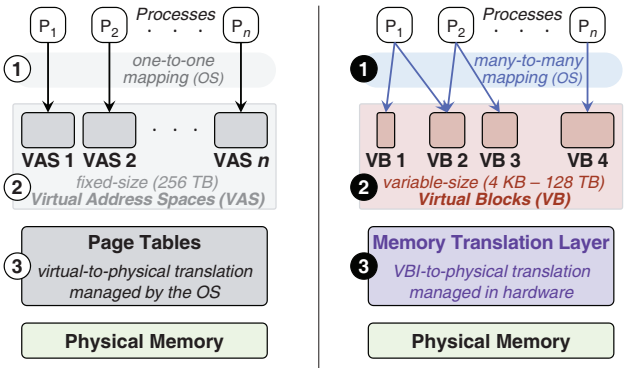

Figure 1: Virtual memory management in x86-64 and in VBI.

VBI seamlessly and efficiently supports important optimizations that improve overall system performance, including: (1) enabling benefits akin to using virtually-indexed virtually-tagged (VIVT) caches (e.g., reduced address translation overhead), (2) eliminating two-dimensional page table walks in virtual machine environments, (3) delaying physical memory allocation until the first dirty last-level cache line eviction, and (4) flexibly supporting different virtual-to-physical address translation structures for different memory regions. §3.5 describes these optimizations in detail.

We evaluate VBI for two important and emerging use-cases. First, we demonstrate that VBI significantly reduces the address translation overhead both for *natively-running programs* and for programs running inside a virtual machine (*VM programs*). Quantitative evaluations using workloads from SPEC CPU 2006 [125], SPEC CPU 2017 [126], TailBench [48], and Graph 500 [44] show that a simplified version of VBI that maps VBs using 4 KB granularity only improves the performance of native programs by 2.18× and VM programs by 3.8×. Even when enabling support for large pages for *all data*, which significantly lowers translation overheads, VBI improves performance by 77% for native programs and 89% for VM programs. Second, we demonstrate that VBI significantly improves the performance of heterogeneous memory architectures by evaluating two heterogeneous memory systems (PCM–DRAM [107] and Tiered-Latency-DRAM [74]). We show that VBI, by intelligently mapping frequently-accessed data to the low-latency region of memory, improves overall performance of these two systems by 33% and 21% respectively, compared to systems that employ a heterogeneity-unaware data mapping scheme. §7 describes our methodology, results, and insights from these evaluations.

We make the following key contributions:

- To our knowledge, this is the first work to propose a virtual memory framework that relieves the OS of explicit physical memory management and delegates this duty to the hardware, i.e., the memory controller.
- We propose VBI, a new virtual memory framework that efficiently enables memory-controller-based memory management by exposing a purely virtual memory interface to applications, the OS, and the hardware caches. VBI naturally and seamlessly supports several optimizations (e.g., low-cost page walks in virtual machines, purely virtual caches, delayed physical memory allocation), and integrates well with a wide range of system designs.
- We provide a detailed reference implementation of VBI, including required changes to the user applications, system software, ISA, and hardware.
- We quantitatively evaluate VBI using two concrete use cases: (1) address translation improvements for native execution and virtual machines, and (2) two different heterogeneous memory architectures. Our evaluations show that VBI significantly improves performance in both use cases.

2. Design Principles

To minimize performance and complexity overheads of memory virtualization, our virtual memory framework is grounded on three key design principles.

Appropriately-Sized Virtual Address Spaces. The virtual memory framework should *allow each application to have control over the size of its virtual address space.* The majority of applications far underutilize the large virtual address space offered by modern architectures (e.g., 256 TB in Intel x86-64). Even demanding applications such as databases [27, 35, 89, 91, 93, 114] and caching servers [37, 92] are cognizant of the amount of available physical memory and of the size of virtual memory they need. Unfortunately, a larger virtual address space results in larger or deeper page tables (i.e., page tables with more levels). A larger page table increases TLB contention, while a deeper page table requires a greater number of page table accesses to retrieve the physical address for each TLB miss. In both cases, the address translation overhead increases. Therefore, allowing applications to choose an appropriately-sized virtual address space based on their actual needs, avoids the higher translation overheads associated with a larger address space.

Decoupling Address Translation from Access Protection. The virtual memory framework should *decouple address translation from access protection checks*, as the two have inherently different characteristics. While address translation is typically performed at page granularity, protection information is typically the same for an entire data structure, which can span multiple pages. Moreover, protection information is purely a function of the virtual address, and does not require address translation. However, existing systems store both translation and protection information for each virtual page as part of the page table. Decoupling address translation from protection checking can enable opportunities to remove address translation from the critical path of an access protection check, deferring the translation until physical memory *must* be accessed, thereby lowering the performance overheads of virtual memory.

Better Partitioning of Duties Between Software and Hardware. The virtual memory framework should *allow software to easily communicate semantic information about application data to hardware and allow hardware to manage the physical memory resources.* Different pieces of program data have different performance characteristics (latency, bandwidth, and parallelism), and have other inherent properties (e.g., compressibility, persistence) at the software level. As highlighted by recent work [129,131], while software is aware of this semantic information, the hardware is privy to fine-grained dynamic runtime information (e.g., memory access behavior, phase changes, memory bandwidth availability) that can enable vastly more intelligent management of the underlying hardware resources (e.g., better data mapping, migration, and scheduling decisions). Therefore, conveying semantic information to the hardware (i.e., memory controller) that manages the physical memory resources can enable a host of new optimization opportunities.

3. Virtual Block Interface: Overview

Figure 2 shows an overview of VBI. There are three major aspects of the VBI design: (1) the VBI address space, (2) VBI access permissions, and (3) the Memory Translation Layer. We first describe these aspects in detail (§3.1–§3.3). Next, we explain the implementation of key OS functionalities in VBI (§3.4). Finally, we discuss some of the key optimizations that VBI enables (§3.5).

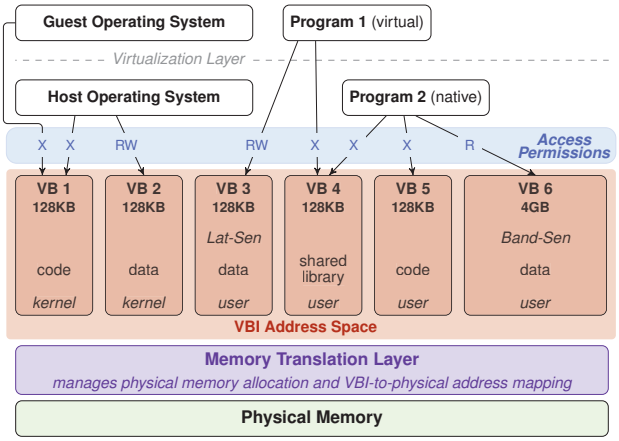

Figure 2: Overview of VBI. *Lat-Sen* and *Band-Sen* represent latency-sensitive and bandwidth-sensitive, respectively.

3.1. VBI Address Space

Unlike most existing architectures wherein each process has its own virtual address space, virtual memory in VBI is a single, globally-visible address space called the *VBI Address Space.* As shown in Figure 2, the VBI Address Space consists of a finite set of *Virtual Blocks* (VBs). Each VB is a contiguous region of VBI address space that does not overlap with any other VB. Each VB contains a semantically meaningful unit of information (e.g., a data structure, a shared library) and is associated with (1) a system-wide unique ID, (2) a specific size (chosen from a set of pre-determined size classes), and (3) a set of properties that specify the semantics of the content of the VB and its desired characteristics. For example, in the figure, `VB 1` indicates the VB with ID 1; its size is 128 KB, and it contains code that is accessible only to the kernel. On the other hand, `VB 6` is the VB with ID 6; its size is 4 GB, and it contains data that is bandwidth-sensitive. In contrast to conventional systems, where the mapping from the process' virtual-to-physical address space is stored in a per-process page table [54], VBI maintains the VBI-to-physical address

mapping information of each VB in a *separate* translation structure. This approach enables VBI to flexibly tune the type of translation structure for each VB to the characteristics of the VB (as described in §5.2). VBI stores the above information and a pointer to the translation structure of each VB in a set of *VB Info Tables* (VITs; described in §4.5.1).

3.2. VBI Access Permissions

As the VBI Address Space is global, all VBs in the system are *visible* to all processes. However, a program can *access* data within a VB *only if* it is attached to the VB with appropriate permissions. In Figure 2, `Program 2` can only execute from `VB 4` or `VB 5`, only read from `VB 6`, and cannot access `VB 3` at all; `Program 1` and `Program 2` both share `VB 4`. For each process, VBI maintains information about the set of VBs attached to the process in an OS-managed per-process table called the *Client–VB Table* (CVT) (described in §4.1.2). VBI provides the OS with a set of instructions with which the OS can control which processes have what type of access permissions to which VBs. On each memory access, the processor checks the CVT to ensure that the program has the necessary permission to perform the access. With this approach, VBI *decouples protection checks from address translation*, which allows it to defer the address translation to the memory controller where the physical address is required to access main memory.

3.3. Memory Translation Layer

In VBI, to access a piece of data, a program must specify the ID of the VB that contains the data and the offset of the data within the VB. Since the ID of the VB is unique system-wide, the combination of the ID and offset points to the address of a specific byte of data in the VBI address space. We call this address the *VBI address*. As the VBI address space is globally visible, similar to the physical address in existing architectures, the VBI address points to a unique piece of data in the system. As a result, VBI uses the VBI address *directly* (i.e., without requiring address translation) to locate data within the on-chip caches without worrying about the complexity of homonyms and synonyms [18, 19, 56], which cannot exist in VBI (see §3.5). Address translation is required only when an access misses in all levels of on-chip caches.

To perform address translation, VBI uses the Memory Translation Layer (MTL). The MTL, implemented in the memory controller with an interface to the system software, manages both allocation of physical memory to VBs and VBI-to-physical address translation (relieving the OS of these duties). Memory-controller-based memory management enables a number of performance optimizations (e.g., avoiding 2D page walks in virtual machines, flexible address translation structures), which we describe in §3.5.

3.4. Implementing Key OS Functionalities

VBI allows the system to efficiently implement existing OS functionalities. In this section, we describe five key functionalities and how VBI enables them.

Physical Memory Capacity Management. In VBI, the MTL allocates physical memory for VBs as and when required. To handle situations when the MTL runs out of physical memory, VBI provides two system calls that allow the MTL to move data from physical memory to the backing store and vice versa. The MTL maintains information about swapped-out data as part of the VB's translation structures.

Data Protection. The goal of data protection is to prevent a malicious program from accessing kernel data or private data of other programs. In VBI, the OS ensures such protection by appropriately setting the permissions with which each process can access different VBs. Before each memory access, the CPU checks if the executing thread has appropriate access permissions to the corresponding VB (§4.2.3).

Inter-Process Data Sharing (True Sharing). When two processes share data (e.g., via pipes), both processes have a coherent view of the shared memory, i.e., modifications made by one process should be visible to the other process. In VBI, the OS supports such *true* sharing by granting both processes permission to access the VB containing the shared data.

Data Deduplication (Copy-on-Write Sharing). In most modern systems, the OS reduces redundancy in physical memory by mapping virtual pages containing the *same* data to the same physical page. On a write to one of the virtual pages, the OS copies the data to a new physical page, and remaps the written virtual page to the new physical page before performing the write. In VBI, the MTL performs data deduplication when a VB is cloned by sharing both translation structures and data pages between the two VBs (§4.4), and using the copy-on-write mechanism to ensure consistency.

Memory-Mapped Files. To support memory-mapped files, existing systems map a region of the virtual address space to a file in storage, and loads/stores to that region are used to access/update the file content. VBI naturally supports memory-mapped files as the OS simply associates the file to a VB of appropriate size. An offset within the VB maps to the same offset within the file. The MTL uses the same system calls used to manage physical memory capacity (described under *Physical Memory Capacity Management* above) to move data between the VB in memory and the file in storage.

3.5. Optimizations Supported by VBI

In this section, we describe four key optimizations that the VBI design enables.

Virtually-Indexed Virtually-Tagged Caches. Using fully-virtual (i.e., VIVT) caches enables the system to delay address translation and reduce accesses to translation structures such as the TLBs. However, most modern architectures do not support VIVT caches due to two main reasons. First, handling homonyms (i.e., where the same virtual address maps to multiple physical addresses) and synonyms (i.e., where multiple virtual addresses map to the same physical address) introduces complexity to the system [18, 19, 56]. Second, although address translation is not required to access VIVT caches, the access permission check required prior to the cache access still necessitates accessing the TLB and can induce a page table walk on a TLB miss. This is due to the fact that the protection bits are stored as part of the page table entry for each page in current systems. VBI avoids both of these problems.

First, VBI addresses are unique system-wide, eliminating the possibility of homonyms. Furthermore, since VBs do not overlap, each VBI address appears in *at most one* VB, avoiding the possibility of synonyms. In case of true sharing (§3.4), different processes are attached to the same VB. Therefore, the VBI address that each process uses to access the shared region refers to the *same* VB. In case of copy-on-write sharing, where the MTL may map two VBI addresses to the same physical memory for deduplication, the MTL creates a new copy of the data before any write to either address. Thus,

neither form of sharing can lead to synonyms. As a result, by using VBI addresses directly to access on-chip caches, VBI achieves benefits akin to VIVT caches without the complexity of dealing with synonyms and homonyms. Additionally, since the VBI address acts as a system-wide single point of reference for the data that it refers to, all coherence-related requests can use VBI addresses without introducing any ambiguity.

Second, VBI decouples protection checks from address translation, by storing protection and address translation information in *separate* sets of tables and delegating access permission management to the OS, avoiding the need to access translation structures for protection purposes (as done in existing systems).

Avoiding 2D Page Walks in Virtual Machines. In VBI, once a process inside a VM attaches itself to a VB (with the help of the host and guest OSes), any memory access from the VM directly uses a VBI address. As described in §3.3, this address is directly used to address the on-chip caches. In case of an LLC miss, the MTL translates the VBI address to physical address. As a result, unlike existing systems, address translation for a VM under VBI is no different from that for a host, enabling significant performance improvements. We expect these benefits to further increase in systems supporting nested virtualization [36, 43]. §6.1 discusses the implementation of VBI in virtualized environments.

Delayed Physical Memory Allocation. As VBI uses VBI addresses to access all on-chip caches, it is no longer necessary for a cache line to be backed by physical memory *before* it can be accessed. This enables the opportunity to delay physical memory allocation for a VB (or a region of a VB) until a dirty cache line from the VB is evicted from the last-level cache. Delayed allocation has three benefits. First, the allocation process is removed from the critical path of execution, as cache line evictions are not on the critical path. Second, for VBs that never leave the cache during the lifetime of the VB (likely more common with growing cache sizes in modern hardware), VBI avoids physical memory allocation altogether. Third, when using delayed physical memory allocation, for an access to a region with no physical memory allocated yet, VBI simply returns a zero cache line, thereby avoiding *both* address translation and a main memory access, which improves performance. §5.1 describes the implementation of delayed physical memory allocation in VBI.

Flexible Address Translation Structures. A recent work [4] shows that different data structures benefit from different types of address translation structures depending on their data layout and access patterns. However, since in conventional virtual memory, the hardware needs to read the OS-managed page tables to perform page table walks, the structure of the page table needs to be understood by both the hardware and OS, thereby limiting the flexibility of the page table structure. In contrast, in VBI, the MTL is the *only* component that manages and accesses translation structures. Therefore, the constraint of sharing address translation structures with the OS is relaxed, providing VBI with more flexibility in employing different types of translation structures in the MTL. Accordingly, VBI maintains a separate translation structure for each VB, and can tune it to suit the properties of the VB (e.g., multi-level tables for large VBs or those with many sparsely-allocated regions, and single-level tables for small VBs or those with many large contiguously-allocated regions). This optimization reduces the number of memory accesses necessary to serve a TLB miss.

4. VBI: Detailed Design

In this section, we present the detailed design and a reference implementation of the Virtual Block Interface. We describe (1) the components architecturally exposed by VBI to the rest of the system (§4.1), (2) the life-cycle of allocated memory (§4.2), (3) the interactions between the processor, OS, and the process in VBI (§4.4), and (4) the operation of the Memory Translation Layer in detail (§4.5).

4.1. Architectural Components

VBI exposes two architectural components to the rest of the system that form the contract between hardware and software: (1) virtual blocks, and (2) *memory clients*.

4.1.1. Virtual Blocks (VBs). The VBI address space in VBI is characterized by three parameters: (1) the size of the address space, which is determined by the bit width of the processor's address bus (64 in our implementation); (2) the number of VB size classes (8 in our implementation); and (3) the list of size classes (4 KB, 128 KB, 4 MB, 128 MB, 4 GB, 128 GB, 4 TB, and 128 TB). Each size class in VBI is associated with an ID (`SizeID`), and each VB is assigned an ID *within its size class* (`VBID`). Every VB is identified system-wide by its *VBI unique ID* (`VBUID`), which is the concatenation of `SizeID` and `VBID`. As shown in Figure 3, VBI constructs a *VBI address* using two components: (1) `VBUID`, and (2) the offset of the addressed data within the VB. In our implementation, `SizeID` uses three bits to represent each of our eight possible size classes. The remaining address bits are split between `VBID` and the offset. The precise number of bits required for the offset is determined by the size of the VB, and the remaining bits are used for `VBID`. For example, the 4 KB size class in our implementation uses 12 bits for the offset, leaving 49 bits for `VBID`, i.e., 2^{49} VBs of size 4 KB. In contrast, the 128 TB size class uses 47 bits for the offset, leaving 14 bits for `VBID`, i.e., 2^{14} VBs of size 128 TB.

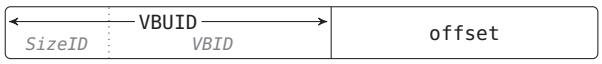

Figure 3: Components of a VBI address.

As §3 describes, VBI associates each VB with a set of flags that characterize the contents of the VB (e.g, `code`, `read-only`, `kernel`, `compressible`, `persistent`). In addition to these flags, software may also provide hints to describe the memory behavior of the data that the VB contains (e.g., latency sensitivity, bandwidth sensitivity, compressibility, error tolerance). Prior work extensively studies a set of useful properties [83, 129, 131, 136]. Software specifies these properties via a bitvector that is defined as part of the ISA specification. VBI maintains the flags and the software-provided hints as a *property bitvector*.

For each VB in the system, VBI stores (1) an *enable* bit to describe whether the VB is currently assigned to any process, (2) the property bitvector, (3) the number of processes attached to the VB (i.e., a reference count), (4) the type of VBI-to-physical address translation structure being used for the VB, and (5) a pointer to the VB's address translation structure. All of this information is stored as an entry in the VB Info Tables (§4.5.1).

4.1.2. Memory Clients. Similar to address space identifiers [3] in existing architectures, VBI introduces the notion of *memory client* to communicate the concept of a process in VBI. A memory client refers to any entity that needs to allocate and use memory, such as the OS itself, and any process

running on the system (natively or inside a virtual machine). In order to track the permissions with which a client can access different VBs, each client in VBI is assigned a unique ID to identify the client system-wide. During execution, VBI tags each core with the client ID of the process currently running on it.

As §3 discusses, the set of VBs that a client can access and their associated permissions are stored in a per-client table called the *Client–VB Table* (CVT). Each entry in the CVT contains (1) a valid bit, (2) `VBUID` of the VB, and (3) a three-bit field representing the read-write-execute permissions (RWX) with which the client can access that VB. For each memory access, the processor checks the CVT to ensure that the client has appropriate access to the VB. The OS implicitly manages the CVTs using the following two new instructions:

```
attach CID, VBUID, RWX
```

```
detach CID, VBUID
```

The `attach` instruction adds an entry for VB `VBUID` in the CVT of client `CID` with the specified `RWX` permissions (either by replacing an invalid entry in the CVT, or being inserted at the end of the CVT). This instruction returns the index of the CVT entry to the OS and increments the reference count of the VB (stored in the VIT entry of the VB; see §4.5.1). The `detach` instruction resets the valid bit of the entry corresponding to VB `VBUID` in the CVT of client `CID` and decrements the reference count of the VB.

The processor maintains the location and size of the CVT for each client in a reserved region of physical memory. As clients are visible to both the hardware and the software, the number of clients is an architectural parameter determined at design time and exposed to the OS. In our implementation, we use 16-bit client IDs (supporting 2^{16} clients).

4.2. Life Cycle of Allocated Memory

In this section, we describe the phases in the life cycle of dynamically-allocated memory: memory allocation, address specification, data access, and deallocation. Figure 4 shows this flow in detail, including the hardware components that aid VBI in efficiently executing memory operations. In §4.4, we discuss how VBI manages code, shared libraries, static data, and the life cycle of an entire process.

When a program needs to allocate memory for a new data structure, it first requests a new VB from the OS. For this purpose, we introduce a new system call, `request_vb`. The program invokes `request_vb` with two parameters: (1) the *expected* size of the data structure, and (2) a bitvector of the desired properties for the data structure (1a in Figure 4).

In response, the OS first scans the VB Info Table to identify the smallest free VB that can accommodate the data structure. The OS then uses the `enable_vb` instruction (1b) to inform the MTL that the VB is now enabled. The `enable_vb` instruction takes the `VBUID` of the VB to be enabled along with the properties bitvector as arguments. Upon executing this instruction, the MTL updates the entry for the VB in the VB Info Table to reflect that it is now enabled with the appropriate properties (1c).

```
enable_vb VBUID, props
```

4.2.1. Dynamic Memory Allocation. After enabling the VB, the OS uses the `attach` instruction (2a) to add the VB to the CVT of the calling process and increment the VB's reference count in its VIT entry (2b; §4.1.2). The OS then returns the index of the newly-added CVT entry as the return value of the `request_vb` system call (stored as `index` in the application code example of Figure 4). This `index` serves as a pointer to the VB. As we discuss in §4.2.2, the program uses this index to specify virtual addresses to the processor.

After the VB is attached to the process, the process can access any location within the VB with the appropriate permissions. It can also dynamically manage memory inside the VB using modified versions of `malloc` and `free` that take the CVT entry `index` as an additional argument (3). During execution, it is possible that the process runs out of memory within a VB (e.g., due to an incorrect estimate of the expected size of the data structure). In such a case, VBI allows automatic promotion of the allocated data to a VB of a larger size class. §4.4 discusses VB promotion in detail.

4.2.2. Address Specification. In order to access data inside a VB, the process generates a two-part virtual address in the format of {`CVT index, offset`}. The `CVT index` specifies the CVT entry that points to the corresponding VB, and the `offset` is the location of the data inside the VB. Accessing the data indirectly through the `CVT index` as opposed to direcly using the VBI address allows VBI to not require relocatable code and maintain the validity of the pointers (i.e., virtual addresses) within a VB when migrating/copying the content of a VB to another VB. With CVT indirection, VBI can seamlessly migrate/copy VBs by just updating the `VBUID` of the corresponding CVT entry with the `VBUID` of the new VB.

4.2.3. Operation of a Memory Load. Figure 4 shows the execution of the memory load instruction triggered by the code `y = (*x)`, where the pointer `x` contains the virtual address consisting of (1) the index of the corresponding VB in the process' CVT, and (2) the offset within the VB (4 in Figure 4). When performing a load operation, the CPU first checks whether `index` is within the range of the client's CVT. Next, the CPU needs to fetch the corresponding CVT entry in order to perform the permissions check. The CPU uses a per-process small direct-mapped CVT cache to speed up accesses to the client's recently-accessed CVT entries (§4.3). Therefore, the CPU looks up the corresponding CVT cache entry using `index` as the key (5), and checks if (1) the client has permission to read from the VB, and (2) `offset` is smaller than the size of the VB. If either of these checks fail, the CPU raises an exception. If the access is allowed, the CPU constructs the VBI address by concatenating the `VBUID` stored in the CVT entry with `offset` (6). The processor directly uses the generated VBI address to access the on-chip caches. If the data is present in any of the on-chip caches, it is returned to the CPU, thereby completing the load operation.

VBI performs address translation in parallel with the cache lookup in order to minimize the address translation overhead on the critical path of the access. Accordingly, when an access misses in the L2 cache, the processor requests the MTL to perform the VBI-to-physical address translation. To this end, MTL fetches the pointer to the VB's translation structure from the VBI Info Table (VIT) entry associated with the VB. VBI uses a VIT cache to speed up accesses to recently-accessed VIT entries (7). In order to facilitate the VBI-to-physical address translation, MTL employs a translation lookaside buffer (TLB). On a TLB hit, the memory controller accesses the cache line using the physical address in the corresponding TLB entry (8). On a TLB miss, the MTL performs the address translation by traversing the VB's translation structure (9), and inserts the mapping information into the TLB once the

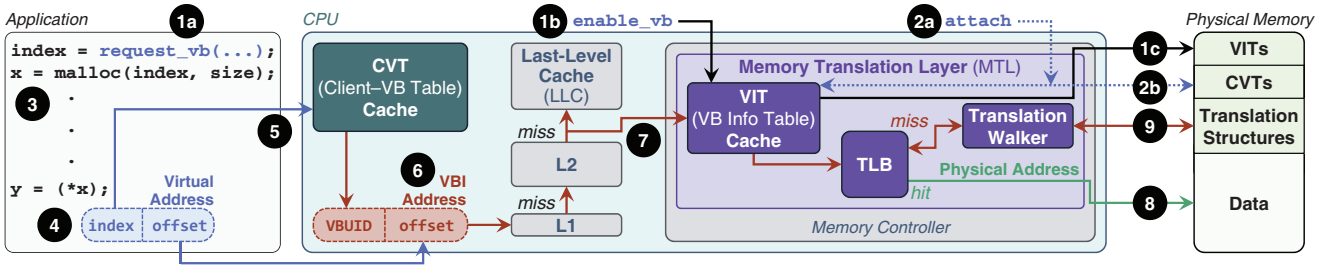

Figure 4: Reference microarchitectural implementation of the Virtual Block Interface.

physical address is obtained. Next, the memory controller fetches the corresponding cache line from main memory and returns it to the processor. The processor inserts the cache line into the on-chip caches using the VBI address, and returns the cache line to the CPU to complete the load. §4.5 describes the operation of the MTL in detail.

4.2.4. Memory Deallocation. The program can deallocate the memory allocated inside a VB using `free` (§4.2.1). When a process terminates, the OS traverses the CVT of the process and detaches all of the VBs attached to the process using the `detach` instruction. For each VB whose reference count (stored as part of VIT entry of the VB; see §4.5.1) drops to zero, the OS informs VBI that the VB is no longer in use via the `disable_vb` instruction.

```
disable_vb VBUID
```

In response to the `disable_vb` instruction, the MTL destroys all state associated with VB `VBUID`. To avoid stale data in the cache, all of the VB's cache lines are invalidated *before* the `VBUID` is reused for another memory allocation. Because there are a large number of VBs in each size class, it is likely that the disabled `VBUID` does not need to be reused immediately, and the cache cleanup can be performed lazily in the background.

4.3. CVT Cache

For every memory operation, the CPU must check if the operation is permitted by accessing the information in the corresponding CVT entry. To exploit locality in the CVT, VBI uses a per-core *CVT cache* to store recently-accessed entries in the client's CVT. The CVT cache is similar to the TLB in existing processors. However, unlike a TLB that caches virtual-to-physical address mappings of page-sized memory regions, the CVT cache maintains information at the VB granularity, and only for VBs that can be accessed by the program. While programs may typically access hundreds or thousands of pages, our evaluations show that most programs only need a few *tens* of VBs to subsume all their data. With the exception of *GemsFDTD* (which allocates 195 VBs),[1] all applications use fewer than 48 VBs. Therefore, the processor can achieve a near-100% hit rate even with a 64-entry *direct-mapped* CVT cache, which is faster and more efficient than the large set-associative TLBs employed by modern processors.

4.4. Processor, OS, and Process Interactions

VBI handles basic process lifetime operations similar to current systems. This section describes in detail how these operations work with VBI.

[1] *GemsFDTD* performs computations in the time domain on 3D grids. It involves multiple execution timesteps, each of which allocates new 3D grids to store the computation output. Multiple allocations are also needed during the post-processing Fourier transformation performed in *GemsFDTD*.

System Booting. When the system is booted, the processor initializes the data structures relevant to VBI (e.g., pointers to VIT tables) with the help of the MTL (discussed in §4.5). An initial ROM program runs as a privileged client, copies the bootloader code from bootable storage to a newly enabled VB, and jumps to the bootloader's entry point. This process initiates the usual sequence of chain loading until the OS is finally loaded into a VB. The OS reads the parameters of VBI, namely, the number of bits of virtual address, the number and sizes of the virtual block size classes, and the maximum number of memory clients supported by the system, to initialize the OS-level memory management subsystem.

Process Creation. When a binary is executed, the OS creates a new process by associating it with one of the available client IDs. For each section of the binary (e.g., code, static data), the OS (1) enables the smallest VB that can fit the contents of the section and associates the VB with the appropriate properties using the `enable_vb` instruction, (2) attaches itself to the VB with write permissions using the `attach` instruction, (3) copies the contents from the application binary into the VB, and (4) detaches itself from the VB using the `detach` instruction. The OS then attaches the client to the newly enabled VBs and jumps to program's entry point.

Shared Libraries. The OS loads the executable code of each shared library into a separate VB. While a shared library can dynamically allocate data using the `request_vb` system call, any static per-process data associated with the library should be loaded into a separate VB for each process that uses the library. In existing systems, access to static data is typically performed using PC-relative addressing. VBI provides an analogous memory addressing mode that we call *CVT-relative addressing*. In this addressing mode, the CVT index of a memory reference is specified relative to the CVT index of the VB containing the reference. Specifically, in shared libraries, all references to static data use +1 CVT-relative addressing, i.e., the CVT index of the data is one more than the CVT index of the code. After process creation, the OS iterates over the list of shared libraries requested by the process. For each shared library, the OS attaches the client to the VB containing the corresponding library code and ensures that the subsequent CVT entry is allocated to the VB containing the static data associated with the shared library. This solution avoids the need to perform load-time relocation for each data reference in the executable code, although VBI can use relocations in the same manner as current systems, if required.

Process Destruction. When a process terminates, the OS deallocates all VBs for the process using the mechanism described in §4.2.4, and then frees the client ID for reuse.

Process Forking. When a process forks, all of its memory state must be replicated for the newly created process. In VBI,

forking entails creating copies of all the private VBs attached to a process. To reduce the overhead of this operation, VBI introduces the following instruction:

```
clone_vb SVBUID, DVBUID
```

`clone_vb` instructs VBI to make the destination VB `DVBUID` a clone of the source VB `SVBUID`. To efficiently implement `clone_vb`, the MTL marks all translation structures and physical pages of the VB as copy-on-write, and lazily copies the relevant regions if they receive a write operation.[2]

When forking a process, the OS first copies all CVT entries of the parent to the CVT of the child so that the child VBs have the same CVT index as the parent VBs. This maintains the validity of the pointers in the child VBs after cloning. Next, for each CVT entry corresponding to a private VB (shared VBs are already enabled), the OS (1) enables a new VB of the same size class and executes the `clone_vb` instruction, and (2) updates the `VBUID` in the CVT entry to point to the newly enabled clone. The fork returns after all the `clone_vb` operations are completed.

VB Promotion. As described in §4.2.1, when a program runs out of memory for a data structure within the assigned VB, the OS can automatically promote the data structure to a VB of higher size class. To perform such a promotion, the OS first suspends the program. It enables a new VB of the higher size class, and executes the `promote_vb` instruction.

```
promote_vb SVBUID, LVBUID
```

In response to this instruction, VBI first flushes all dirty cache lines from the smaller VB with the unique ID of `SVBUID`. This operation can be sped up using structures like the Dirty Block Index [116]. VBI then copies all the translation information from the smaller VB appropriately to the larger VB with the unique ID of `LVBUID`. After this operation, in effect, the early portion of the larger VB is mapped to the same region in the physical memory as the smaller VB. The remaining portions of the larger VB are unallocated and can be used by the program to expand its data structures and allocate more memory using `malloc`. VBI updates the entry in the program's CVT that points to `SVBUID` to now point to `LVBUID`.

4.5. Memory Translation Layer

The Memory Translation Layer (MTL) centers around the VB Info Tables (VITs), which store the metadata associated with each VB. In this section, we discuss (1) the design of the VITs, (2) the two main responsibilities of the MTL; memory allocation and address translation, and (3) the hardware complexity of the MTL.

4.5.1. VB Info Table (VIT). As §4.1.1 briefly describes, MTL uses a set of VB Info Tables (VITs) to maintain information about VBs. Specifically, for each VB in the system, a VB Info Table stores an entry that consists of (1) an *enable* bit, which indicates if the VB is currently assigned to a process; (2) `props`, a bitvector that describes the VB properties; (3) the number of processes attached to the VB (i.e., a reference count); (4) the type of VBI-to-physical address translation structure being used for the VB; and (5) a pointer to the translation structure. For ease of access, the MTL maintains a *separate* VIT for each size class. The ID of a VB within its size class (VBID) is used as an index into the corresponding VIT. When a VB is enabled (using `enable_vb`), the MTL finds the corresponding VIT and entry using the `SizeID` and `VBID`, respectively (both extracted from `VBUID`). MTL then sets the *enabled* bit of the entry and updates `props`. The reference counter of the VB is also set to 0, indicating that no process is attached to this VB. The type and pointer of the translation structure of the VB are updated in its VIT entry at the time of physical memory allocation (as we discuss in §5.2). Since a VIT contains entries for the VBs of only a single size class, the number of entries in each VIT equals the number of VBs that the associated size class supports (§4.1.1). However, VBI limits the size of each VB Info Table by storing entries only up to the currently-enabled VB with the largest `VBID` in the size class associated with that VB Info Table. The OS ensures that the table does not become prohibitively large by reusing previously-disabled VBs for subsequent requests (§4.2.4).

4.5.2. Base Memory Allocation and Address Translation. Our *base* memory allocation algorithm allocates physical memory at 4 KB granularity. Similar to x86-64 [54], Our *base* address translation mechanism stores VBI-to-physical address translation information in multi-level tables. However, unlike the 4-level page tables in x86-64, VBI uses tables with varying number of levels according to the size of the VB. For example, a 4 KB VB does not require a translation structure (i.e., can be direct-mapped) since 4 KB is the minimum granularity of meomry allocation. On the other hand, a 128 KB VB requires a one-level table for translating address to 4 KB regions. As a result, smaller VBs require fewer memory accesses to serve a TLB miss. For each VB, the VIT stores a pointer to the address of the root of the multi-level table (or the base physical address of the directly mapped VBs).

4.5.3. MTL Hardware Complexity. We envision the MTL as software running on a programmable low-power core within the memory controller. While conventional OSes are responsible for memory allocation, virtual-to-physical mapping, and memory protection, the MTL does not need to deal with protection, so we expect the MTL code to be simpler than typical OS memory management software. As a result, the complexity of the MTL hardware is similar to that of prior proposals such as Pinnacle [6] (commercially available) and Page Overlays [118], which perform memory allocation and remapping in the memory controller. While both Pinnacle and Page Overlays are hardware solutions, VBI provides flexibility by making the MTL programmable, thereby allowing software updates for different memory management policies (e.g., address translation, mapping, migration, scheduling). Our goal in this work is to understand the potential of hardware-based memory allocation and address translation.

5. Allocation and Translation Optimizations

The MTL employs three techniques to optimize the base memory allocation and address translation described in §4.5.2. We explain these techniques in the following subsections.

5.1. Delayed Physical Memory Allocation

As described in §3.5, VBI delays physical memory allocation for a VB (or a region of a VB) until a dirty cache line from that VB (or a region of the VB) is evicted from the last-level cache (LLC). This optimization is enabled by the fact that VBI uses VBI address directly to access *all* on-chip caches. Therefore, a cache line does *not* need to be backed by a physical memory mapping in order to be accessed.

In this approach, when a VB is enabled, VBI does not immediately allocate physical memory to the VB. On an LLC miss

[2]The actual physical copy can be accelerated using in-DRAM copy mechanisms such as RowClone [117], LISA [22], and NoM [119].

to the VB, VBI checks the status of the VB in its corresponding VIT entry. If there is no physical memory backing the data, VBI does one of two things. (1) If the VB corresponds to a memory-mapped file or if the required data was allocated before but swapped out to a backing store, then VBI allocates physical memory for the region, interrupts the OS to copy the relevant data from storage into the allocated memory, and then returns the relevant cache line to the processor. (2) If this is the first time the cache line is being accessed from memory, VBI simply returns a zeroed cache line without allocating physical memory to the VB.

On a dirty cache line writeback from the LLC, if physical memory is yet to be allocated for the region that the cache line maps to, VBI first allocates physical memory for the region, and then performs the writeback. VBI allocates only the region of the VB containing the evicted cache line. As §4.5.2 describes, our base memory allocation mechanism allocates physical memory at a 4 KB granularity. Therefore, the region allocated for the evicted cache line is 4 KB. §5.3 describes an optimization that eagerly *reserves* a larger amount of physical memory for a VB during allocation, to reduce the overall translation overhead.

5.2. Flexible Address Translation Structures

For each VB, VBI chooses one of three types of address translation structures, depending on the needs of the VB and the physical memory availability. The first type *directly* maps the VB to physical memory when enough contiguous memory is available. With this mapping, a single TLB entry is sufficient to maintain the translation for the entire VB. The second type uses a single-level table, where the VB is divided into equal-sized blocks of one of the supported size classes. Each entry in the table maintains the mapping for the corresponding block. This mapping exploits the fact that a majority of the data structures are densely allocated inside their respective VBs. With a single-level table, the mapping for any region of the VB can be retrieved with a single memory access. The third type, suitable for sparsely-allocated VBs, is our base address translation mechanism (described in §4.5), which uses multi-level page tables where the table depth is chosen based on the size of the VB.

In our evaluation, we implement a flexible mechanism that statically chooses a translation structure type based on the size of the VB. Each 4 KB VB is directly mapped. 128 KB and 4 MB VBs use a single-level table. VBs of a larger size class use a multi-level table with as many levels as necessary to map the VB using 4 KB pages.[3] The *early reservation* optimization (described in §5.3) improves upon this static policy by dynamically choosing a translation structure type from the three types mentioned above based on the available contiguous physical memory. While we evaluate table-based translation structures in this work, VBI can be easily extended to support other structures (e.g., customized per-application translation structures as proposed in DVMT [4]).

Similar to x86-64, VBI uses multiple types of TLBs to cache mappings of different granularity. The type of translation structure used for a VB is stored in the VIT and is cached in the on-chip VIT Cache. This information enables VBI to access the right type of TLB. For a fair comparison, our evaluations use the same TLB type and size for all baselines and variants of VBI.

[3]For fair comparison with conventional virtual memory, our evaluations use a 4 KB granularity to map VBs to physical memory. However, VBI can flexibly map VBs at the granularity of any available size class.

5.3. Early Reservation of Physical Memory

VBI can perform early reservation of the physical memory for a VB. To this end, VBI reserves (but does not allocate) physical memory for the entire VB at the time of memory allocation, and treats the VB as *directly mapped* by serving future memory allocation requests for that VB from that contiguous reserved region. This optimization is inspired by prior work on super-page management [90], which reserves a larger contiguous region of memory than the requested size, and upgrades the allocated pages to larger super-pages when enough contiguous pages are allocated in that region.

For VBI's early reservation optimization, at the time of the *first* physical memory allocation request for a VB, the MTL checks if there is enough contiguous free space in physical memory to fit the entire VB. If so, it allocates the requested memory from that contiguous space, and marks the remaining free blocks in that contiguous space as reserved for that specific VB. In order to reduce internal fragmentation when free physical memory is running low, physical blocks reserved for a VB may be used by another VB when no unreserved blocks are available. As a result, the MTL uses a three-level priority when allocating physical blocks: (1) free blocks reserved for the VB that is demanding allocation, (2) unreserved free blocks, and (3) free blocks reserved for other VBs. A VB is considered directly mapped as long as all its allocated memory is mapped to a single contiguous region of memory, thereby requiring just a single TLB entry for the entire VB. If there is not enough contiguous physical memory available to fit the entire VB, the early reservation mechanism allocates the VB sparsely by reserving blocks of the largest size class that can be allocated contiguously.

With the early reservation approach, memory allocation is performed at a different granularity than mapping, which enables VBI to benefit from larger mapping granularities and thereby minimize the address translation latency, while eliminating memory allocation for regions that may never be accessed. To support the early reservation mechanism, VBI uses the Buddy algorithm [67, 120] to manage free and reserved regions of different size classes.

6. VBI in Other System Architectures

VBI is designed to easily and efficiently function in various system designs. We describe the implementation of VBI in two important examples of modern system architectures: virtualized environments and multi-node systems.

6.1. Supporting Virtual Machines

VBI implements address space isolation between virtual machines (VMs) by *partitioning* the global VBI address space among multiple VMs and the host OS. To this end, VBI reserves a few bits in the VBI address for the `VM ID`. Figure 5 shows how VBI implements this for a system supporting 31 virtual machines (ID 0 is reserved for the host). In the VBI address, the 5 bits following the size class bits are used to denote the `VM ID`. For every new virtual machine in the system, the host OS assigns a `VM ID` to be used by the guest OS while assigning virtual blocks to processes inside the virtual machine. VBI partitions client IDs using a similar approach. With address space division between VMs, a guest VM is unaware that it is virtualized, and it can allocate/deallocate/access VBs *without* having to coordinate with the host OS. Sharing VBs across multiple VMs is possible, but requires explicit coordination with the host OS.

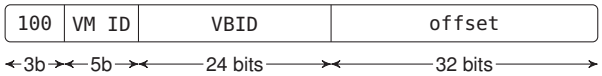

Figure 5: Partitioning the VBI address space among virtual machines, using the 4 GB size class (100) as an example.

6.2. Supporting Multi-Node Systems

There are many ways to implement VBI in multi-node systems. Our initial approach provides each node with its own MTL. VBI equally partitions VBs of each size class among the MTLs, with the higher order bits of VBID indicating the VB's home MTL. The home MTL of a VB is the only MTL that manages the VB's physical memory allocation and address translation. When allocating a VB to a process, the OS attempts to ensure that the VB's home MTL is in the same node as the core executing the process. During phase changes, the OS can seamlessly migrate data from a VB hosted by one MTL to a VB hosted by another MTL. We leave the evaluation of this approach and exploration of other ways of integrating VBI with multi-node systems to future work.

7. Evaluation

We evaluate VBI for two concrete use cases. First, we evaluate how VBI reduces address translation overheads in native and virtualized environments (§7.2.1 and §7.2.2, respectively). Second, we evaluate the benefits that VBI offers in harnessing the full potential of two main memory architectures that are tightly dependent on the data mapping: (1) a hybrid PCM–DRAM memory architecture; and (2) TL-DRAM [74], a heterogeneous-latency DRAM architecture (§7.3).

7.1. Methodology

For our evaluations, we use a heavily-customized version of Ramulator [65] to faithfully model all components of the memory subsystem (including TLBs, page tables, the page table walker, and the page walk cache), as well as the functionality of memory management calls (e.g., malloc, realloc, free). We have released this modified version of Ramulator [113]. Table 1 summarizes the main simulation parameters. Our workloads consist of benchmarks from SPECspeed 2017 [126], SPEC CPU 2006 [125], TailBench [48], and Graph 500 [44]. We identify representative code regions for the SPEC benchmarks using SimPoint [96]. For TailBench applications, we skip the first five billion instructions. For Graph 500, we mark the region of interest directly in the source code. We use an Intel Pintool [81] to collect traces of the representative regions of each of our benchmarks. For our evaluations, we first warm up the system with 100 million instructions, and then run the benchmark for 1 billion instructions.

CPU	4-wide issue, OOO, 128-entry ROB
L1 Cache	32 KB, 8-way associative, 4 cycles
L2 Cache	256 KB, 8-way associative, 8 cycles
L3 Cache	8 MB (2 MB per-core), 16-way associative, 31 cycles
L1 DTLB	4 KB pages: 64-entry, fully associative 2 MB pages: 32-entry, fully associative
L2 DTLB	4 KB and 2 MB pages: 512-entry, 4-way associative
Page Walk Cache	32-entry, fully associative
DRAM	DDR3-1600, 1 channel, 1 rank/channel 8 banks/rank, open-page policy
DRAM Timing [88]	tRCD=5cy, tRP=5cy, tRRDact=3cy, tRRDpre=3cy
PCM	PCM-800, 1 channel, 1 rank/channel, 8 banks/rank
PCM Timing [72]	tRCD=22cy, tRP=60cy, tRRDact=2cy, tRRDpre=11cy

Table 1: Simulation configuration.

7.2. Use Case 1: Address Translation

We evaluate the performance of seven baseline systems to compare with VBI: (1) **Native**: applications run natively on an x86-64 system with only 4 KB pages; (2) **Native-2M**: Native but with only 2 MB pages; (3) **Virtual**: applications run inside a virtual machine with only 4 KB pages; (4) **Virtual-2M**: Virtual but with only 2 MB pages;[4] (5) **Perfect TLB**: an unrealistic version of Native with no L1 TLB misses (i.e., no address translation overhead); (6) **VIVT**: Native with VIVT on-chip caches; and (7) **Enigma-HW-2M**: applications run natively in a system with Enigma [137]. Enigma uses a system-wide unique intermediate address space to defer address translation until data must be retrieved from physical memory. A centralized translation cache (CTC) at the memory controller performs intermediate-to-physical address translation. However, unlike VBI, Enigma asks the OS to perform the translation on a CTC miss, and to explicitly manage address mapping. Therefore, Enigma's benefits do not seamlessly extend to programs running inside a virtual machine. We evaluate Enigma with a 16K-entry centralized translation cache (CTC) that we enhance with hardware-managed page walks and 2 MB pages.

We evaluate the performance of three VBI systems: (1) **VBI-1**: inherently virtual caches (§3.5) along with our *flexible translation mechanism* that maps VBs using a 4 KB granularity (§4.5.2) , (2) **VBI-2**: VBI-1 with *delayed physical memory allocation* (allocates the 4 KB region of the VB that the dirty cache line evicted from the last-level cache belongs to). (§5.1), and (3) **VBI-Full**: VBI-2 with *early reservation* (§5.3). VBI-1 and VBI-2 manage memory at 4 KB granularity, while VBI-Full uses early reservation to support all of the size classes listed in §4.1.1 for VB allocation, providing similar benefits to large page support and direct mapping. We first present results comparing VBI-1 and VBI-2 with Native, Virtual, VIVT, and Perfect TLB (§7.2.1). We then present results comparing VBI-Full with Native-2M, Enigma-HW-2M, and Perfect TLB (§7.2.2).

7.2.1. Results with 4 KB Pages. Figure 6 plots the performance of Virtual, VIVT, VBI-1, VBI-2, and Perfect TLB normalized to the performance of Native, for a single-core system. We also show VBI-Full as a reference that shows the full potentials of VBI which VBI-1 and VBI-2 do not enable. *mcf* has an overwhelmingly high number of TLB misses. Consequently, mechanisms that reduce TLB misses greatly improve *mcf*'s performance, to the point of skewing the average significantly. Therefore, the figure also presents the average speedup without *mcf*. We draw five observations from the figure.

First, VBI-1 outperforms Native by 50%, averaged across all benchmarks (25% without *mcf*). This performance gain is a direct result of (1) inherently virtual on-chip caches in VBI that reduce the number of address translation requests, and (2) fewer levels of address translation for smaller VBs, which reduces the number of translation-related memory accesses (i.e., page walks).

Second, Perfect TLB serves as an upper bound for the performance benefits of VBI-1. However, by employing flexible translation structures, VBI-1 bridges the performance gap between Native and Perfect TLB by 52%, on average.

Third, when accessing regions for which no physical memory is allocated yet, VBI-2 avoids *both* the memory requests themselves and any translation-related memory accesses for those requests. Therefore, VBI-2 enables benefits over and beyond solely reducing the number of page walks, as it further improves the overall performance by reducing the number of memory requests accessing the main memory as well. Consequently, for many memory-intensive applications, VBI-2 outperforms Perfect TLB. Compared to Perfect TLB, VBI-2

[4] We augment this system with a 2D page walk cache, which is shown to improve the performance of guest workloads [14].

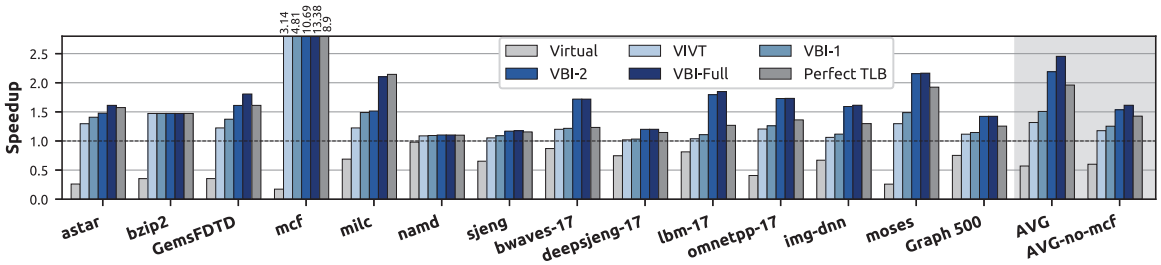

Figure 6: Performance of systems with 4KB pages (normalized to Native).

reduces the total number of DRAM accesses (including the translation-related memory accesses) by 62%, averaged across applications that outperform Perfect TLB, and by 46% across all applications. Overall, VBI-2 outperforms Native by an average of 118% (53% without *mcf*).

Fourth, by performing address translations *only for and in parallel with* LLC accesses, VIVT outperforms Native by 31% on average (17% without *mcf*). This performance gain is due to reducing the number of translation requests and therefore decreasing the number of TLB misses using VIVT caches. However, VBI-1 and VBI-2 gain an extra 19% and 87% performance on average, respectively, over VIVT. These improvements highlight VBI's ability to improve performance beyond only employing VIVT caches.

Finally, our results indicate that due to considerably higher translation overhead, Virtual significantly slows down applications compared to Native (44% on average). As described in §3.5, once an application running inside a virtual machine is attached to its VBs, VBI incurs no additional translation overhead compared to running natively. As a result, in virtualized environments that use only 4K pages, VBI-1 and VBI-2 achieve an average performance of 2.6× and 3.8×, respectively, compared to Virtual.

We conclude that even when mapping and allocating VBs using 4 KB granularity only, both VBI-1 and VBI-2 provide large benefits over a wide range of baseline systems, due to their effective optimizations to reduce address translation and memory allocation overheads. VBI-Full further improves performance by mapping VBs using larger granularities (as we elaborate in §7.2.2).

7.2.2. Results with Large Pages. Figure 7 plots the performance of Virtual-2M, Enigma-HW-2M, VBI-Full, and Perfect TLB normalized to the performance of Native-2M. We enhance the original design of Enigma [137] by replacing the OS system call handler for address translation on a CTC miss with a completely hardware-managed address translation, similar to VBI. For legibility, the figure shows results for only a subset of the applications. However, the chosen applications capture the behavior of all the applications, and the average (and average without *mcf*) is calculated across all evaluated applications. We draw three observations from the figure.

First, managing memory at 2 MB granularity improves the performance of applications compared to managing memory at 4 KB granularity. This is because (1) the larger page size lowers the average TLB miss count (e.g., 66% lower for Native-2M compared to Native), and (2) requires fewer page table accesses on average to serve TLB misses (e.g., 73% fewer for Native-2M compared to Native).

Second, Enigma-HW-2M improves overall performance for programs running *natively* on the system by 34% compared to Native-2M, averaged across all benchmarks (including mcf). The performance gain is a direct result of (1) the very large

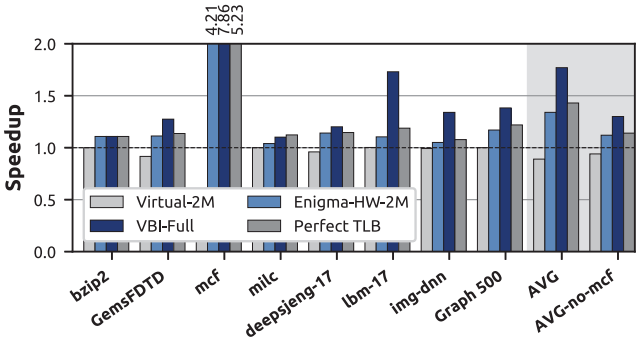

Figure 7: Performance with large pages (norm. to Native-2M).

CTC (16K entries), which reduces the number of translation-related memory accesses by 89% on average compared to Native-2M; and (2) our hardware-managed address translation enhancement, which removes the costly system calls on each page walk request.

Third, VBI-Full, with all three of our optimizations in §5, maps most VBs using direct mapping, thereby significantly reducing the number of TLB misses and translation-related memory accesses compared to Native-2M (on average by 79% and 99%, respectively). In addition, VBI-Full retains the benefits of VBI-2, which reduces the number of *overall* DRAM accesses. VBI-Full reduces the *total* number of DRAM accesses (including translation-related memory accesses) by 56% on average compared to Perfect TLB. Consequently, VBI-Full outperforms all four comparison points including Perfect TLB. Specifically, VBI-Full improves performance by 77% compared to Native-2M, 43% compared to Enigma-HW-2M and 89% compared to Virtual-2M.

We conclude that by employing all of the optimizations that it enables, VBI significantly outperforms all of our baselines in both native and virtualized environments.

7.2.3. Multicore Evaluation. Figure 8 compares the weighted speedup of VBI-Full against four baselines in a quad-core system. We examine six different workload bundles, listed in Table 2, which consist of the applications studied in our single-core evaluations. From the figure, we make two observations. First, averaged across all bundles, VBI-Full improves performance by 38% and 18%, compared to Native and Native-2M, respectively. Second, VBI-Full outperforms Virtual and Virtual-2M by an average 67% and 34%, respectively. We conclude that the benefits of VBI persist even in the presence of higher memory load in multicore systems.

wl1	deepsjeng, omnetpp, bwaves, lbm	**wl4**	milc, namd, GemsFDTD, bzip2
wl2	graph500, astar, img-dnn, moses	**wl5**	bzip2, GemsFDTD, sjeng, mcf
wl3	mcf, GemsFDTD, astar, milc	**wl6**	namd, bzip2, astar, sjeng

Table 2: Multiprogrammed workload bundles.

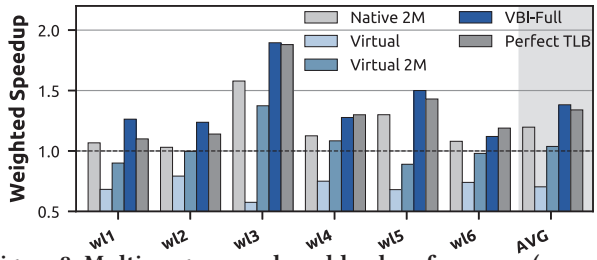

Figure 8: Multiprogrammed workload performance (normalized to Native).

7.3. Use Case 2: Memory Heterogeneity

As mentioned in §1, extracting the best performance from heterogeneous-latency DRAM architectures [22, 23, 64, 73, 74, 80, 82, 117, 124] and hybrid memory architectures [29, 57, 78, 103, 106–108, 134, 135, 138] critically depends on mapping data to the memory that suits the data requirements, and migrating data as its requirements change. We quantitatively show the performance benefits of VBI in exploiting heterogeneity by evaluating (1) a PCM–DRAM hybrid memory [107]; and (2) TL-DRAM [74], a heterogeneous-latency DRAM architecture. We evaluate five systems: (1) VBI PCM–DRAM and (2) VBI TL-DRAM, in which VBI maps and migrates frequently-accessed VBs to the low-latency memory (the fast memory region in the case of TL-DRAM); (3) Hotness-Unaware PCM–DRAM and (4) Hotness-Unaware TL-DRAM, where the mapping mechanism is unaware of the hotness (i.e., the access frequency) of the data and therefore do not necessarily map the frequently-accessed data to the fast region; and (5) IDEAL in each plot refers to an unrealistic perfect mapping mechanism, which uses oracle knowledge to always map frequently-accessed data to the fast portion of memory.

Figures 9 and 10 show the speedup obtained by VBI-enabled mapping over the hotness-unaware mapping in a PCM–DRAM hybrid memory and in TL-DRAM, respectively. We draw three observations from the figures. First, for PCM–DRAM, VBI PCM–DRAM improves performance by 33% on average compared to the Hotness-Unaware PCM–DRAM, by accurately mapping the frequently-accessed data structures to the low-latency DRAM. Second, by mapping frequently-accessed data to the fast DRAM regions, VBI TL-DRAM takes better advantage of the benefits of TL-DRAM, with a performance improvement of 21% on average compared to Hotness-Unaware TL-DRAM. Third, VBI TL-DRAM performs only 5.3% slower than IDEAL, which is the upper bound of performance achieved by mapping hot data to the fast regions of DRAM.

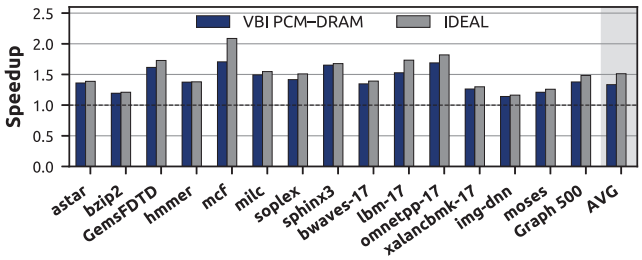

Figure 9: Performance of VBI PCM-DRAM (normalized to data-hotness-unaware mapping).

We conclude that VBI is effective for enabling efficient data mapping and migration in heterogeneous memory systems.

8. Related Work

To our knowledge, VBI is the first virtual memory framework to fully delegate physical memory allocation and address translation to the hardware. This section compares VBI with other virtual memory designs and related works.

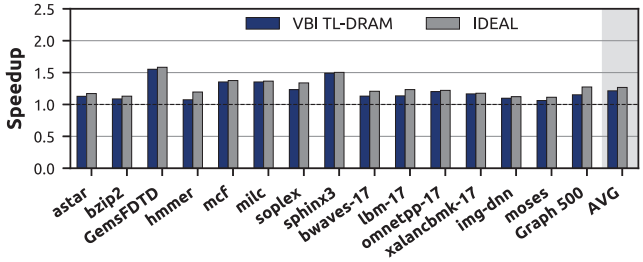

Figure 10: Performance of VBI TL-DRAM (normalized to data-hotness-unaware mapping).

Virtual Memory in Modern Architectures. Modern virtual memory architectures, such as those employed as part of modern instruction set architectures [5, 50, 54, 55], have evolved into sophisticated systems. These architectures have support for features such as large pages, multi-level page tables, hardware-managed TLBs, and variable-size memory segments, but require significant system software support to enable these features and to manage memory. While system software support provides some flexibility to adapt to new ideas, it must communicate with hardware through a rigid contract. Such rigid hardware/software communication introduces costly overheads for many applications (e.g., high overheads with fixed-size per-application virtual address spaces, for applications that only need a small fraction of the space) and prevents the easy adoption of significantly different virtual memory architectures or ideas that depend on large changes to the existing virtual memory framework. VBI is a completely different framework from existing virtual memory architectures. It supports the functionalities of existing virtual memory architectures, but can do much more by reducing translation overheads, inherently and seamlessly supporting virtual caches, and avoiding unnecessary physical memory allocation. These benefits come from enabling completely hardware-managed physical memory allocation and address translation, which no modern virtual memory architecture does.

Several memory management frameworks [7, 8, 75, 101, 104, 130] are designed to minimize the virtual memory overhead in GPUs. Unlike VBI, these works provide optimizations *within the existing virtual memory design*, so their benefits are constrained to the design of conventional virtual memory.

OS Support for Virtual Memory. There has been extensive work on how address spaces should be mapped to execution contexts [79]. Unix-like OSes provide a rigid one-to-one mapping between virtual address spaces and processes [84, 109]. SpaceJMP [31] proposes a design in which processes can jump from one virtual address space to another in order to access larger amounts of physical memory. Single address space OSes rely on system-software-based mechanisms to expose a single global address space to processes, to facilitate efficient data sharing between processes [24, 25, 49]. VBI makes use of a similar concept as single address space OSes with its single globally-visible VBI address space. However, while existing single address space OS designs expose the single address space to processes, VBI does not do so, and instead has processes operate on CVT-relative virtual addresses. This allows VBI to enjoy the same advantages as single address space OSes (e.g., synonym-/homonym-free VIVT caches), while providing further benefits (e.g., non-fixed addresses for shared libraries, hardware-based memory management). Additionally, VBI naturally supports single

address space sharing between the host OS and guest OSes in virtualized environments.

User-Space Memory Management. Several OS designs propose user-space techniques to provide an application with more control over memory management [4, 12, 32, 33, 46, 58, 66, 115, 132]. For example, the exokernel OS architecture [33, 58] allows applications to manage their own memory and provides memory protection via capabilities, thereby minimizing OS involvement. Do-It-Yourself Virtual Memory Translation (DVMT) [4] decouples memory translation from protection in the OS, and allows applications to handle their virtual-to-physical memory translation. These solutions (1) increase application complexity and add non-trivial programmer burden to directly manage hardware resources, and (2) do not expose the rich runtime information available in the hardware to memory managers. In contrast to these works, which continue to rely on software for physical memory management, VBI does not use *any* part of the software stack for physical memory management. By partitioning the duties differently between software and hardware, and, importantly, performing physical memory management in the memory controller, VBI provides similar flexibility benefits as user-space memory management without introducing additional programmer burden.

Reducing Address Translation Overhead. Several studies have characterized the overhead of virtual-to-physical address translation in modern systems, which occurs primarily due to growing physical memory sizes, inflexible memory mappings, and virtualization [11, 17, 51, 54, 61, 85]. Prior works try to ameliorate the address translation issue by: (1) increasing the TLB reach to address a larger physical address space [7, 26, 59, 95, 97, 98, 112, 127]; (2) using TLB speculation to speed up address translation [10, 94, 100]; (3) introducing and optimizing page walk caches to store intermediate page table addresses [9, 14, 15, 34]; (4) adding sharing and coherence between caching structures to share relevant address translation updates [13, 16, 34, 62, 69, 98, 110, 133]; (5) allocating and using large contiguous regions of memory such as superpages [7, 11, 39–41, 99]; (6) improving memory virtualization with large, contiguous memory allocations and better paging structures [7, 39, 40, 99, 100, 112]; and (7) prioritizing page walk data throughout the memory hierarchy [8]. While all of these works can mitigate the translation overhead, they build on top of the existing rigid virtual memory framework and do not address the underlying overheads inherent to the existing rigid framework and to software-based memory management. Unlike these works, VBI is a completely new framework for virtual memory, which eliminates several underlying sources of address translation overhead and enables many other benefits (e.g., efficient memory management in virtual machines, easy extensibility to heterogeneous memory systems). VBI can be combined with some of the above proposals to further optimize address translation.

9. Conclusion

We introduce the Virtual Block Interface (VBI), a new virtual memory framework to address the challenges in adapting conventional virtual memory to increasingly diverse system configurations and workloads. The key idea of VBI is to delegate memory management to hardware in the memory controller. The memory-controller-based memory management in VBI leads to many benefits not easily attainable in existing virtual memory, such as inherently virtual caches, avoiding 2D page walks in virtual machines, and delayed physical memory allocation. We experimentally show that VBI (1) reduces the overheads of address translation by reducing the number of translation requests and associated memory accesses, and (2) increases the effectiveness of managing heterogeneous main memory architectures. We conclude that VBI is a promising new virtual memory framework that can enable several important optimizations and increased design flexibility for virtual memory. We believe and hope that VBI will open up a new direction and many opportunities for future work in novel virtual memory frameworks.

Acknowledgments

We thank the anonymous reviewers of ISCA 2019, MICRO 2019, HPCA 2019, and ISCA 2020 for their valuable comments. We thank our industrial partners, especially Alibaba, Facebook, Google, Huawei, Intel, Microsoft, and VMware, for their generous donations. We thank SAFARI group members for valuable feedback and the stimulating environment.

References

[1] R. Achermann *et al.*, "Separating Translation from Protection in Address Spaces with Dynamic Remapping," in *HotOS*, 2017.
[2] R. Achermann *et al.*, "Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines," in *ASPLOS*, 2020.
[3] T. Ahearn *et al.*, "Virtual Memory System," U.S. Patent 3 781 808, 1973.
[4] H. Alam *et al.*, "Do-It-Yourself Virtual Memory Translation," in *ISCA*, 2017.
[5] Arm Ltd., *Arm® Architecture Reference Manual: ARMv8, for ARMv8-A Architecture Profile*, 2013.
[6] S. Arramreddy *et al.*, "Pinnacle: IBM MXT in a Memory Controller Chip," *IEEE Micro*, 2001.
[7] R. Ausavarungnirun *et al.*, "Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes," in *MICRO*, 2017.
[8] R. Ausavarungnirun *et al.*, "MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency," in *ASPLOS*, 2018.
[9] T. W. Barr *et al.*, "Translation Caching: Skip, Don't Walk (the Page Table)," in *ISCA*, 2010.
[10] T. W. Barr *et al.*, "SpecTLB: A Mechanism for Speculative Address Translation," in *ISCA*, 2011.
[11] A. Basu *et al.*, "Efficient Virtual Memory for Big Memory Servers," in *ISCA*, 2013.
[12] A. Baumann *et al.*, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *SOSP*, 2009.
[13] S. Bharadwaj *et al.*, "Scalable Distributed Shared Last-Level TLBs Using Low-Latency Interconnects," in *MICRO*, 2018.
[14] R. Bhargava *et al.*, "Accelerating Two-Dimensional Page Walks for Virtualized Systems," in *ASPLOS*, 2008.
[15] A. Bhattacharjee, "Large-Reach Memory Management Unit Caches," in *MICRO*, 2013.
[16] A. Bhattacharjee *et al.*, "Shared Last-Level TLBs for Chip Multiprocessors," in *ISCA*, 2011.
[17] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," in *PACT*, 2009.
[18] M. Cekleov and M. Dubois, "Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors," *IEEE Micro*, 1997.
[19] M. Cekleov and M. Dubois, "Virtual-Address Caches Part 2: Multiprocessor Issues," *IEEE Micro*, 1997.
[20] J. M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *TC*, 1996.
[21] J. M. Chang *et al.*, "Architectural Support for Dynamic Memory Management," in *ICCD*, 2000.
[22] K. K. Chang *et al.*, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
[23] K. K. Chang *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
[24] J. S. Chase *et al.*, "Sharing and Protection in a Single-Address-Space Operating System," *TOCS*, 1994.
[25] J. S. Chase *et al.*, "Lightweight Shared Objects in a 64-bit Operating System," in *OOPSLA*, 1992.
[26] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *ASPLOS*, 2017.
[27] G. DeCandia *et al.*, "Dynamo: Amazon's Highly Available Key-Value Store," in *SOSP*, 2007.
[28] P. J. Denning, "Virtual Memory," *CSUR*, 1970.
[29] G. Dhiman *et al.*, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *DAC*, 2009.
[30] Y. Du *et al.*, "Supporting Superpages in Non-Contiguous Physical Memory," in *HPCA*, 2015.
[31] I. El Hajj *et al.*, "SpaceJMP: Programming with Multiple Virtual Address Spaces," in *ASPLOS*, 2016.
[32] D. R. Engler *et al.*, "AVM: Application-Level Virtual Memory," in *HotOS*, 1995.
[33] D. R. Engler *et al.*, "Exokernel: An Operating System Architecture for Application-Level Resource Management," in *SOSP*, 1995.
[34] A. Esteve *et al.*, "Exploiting Parallelization on Address Translation: Shared Page Walk Cache," in *OMHI*, 2014.
[35] Facebook, Inc., "RocksDB: A Persistent Key–Value Store," https://rocksdb.org/.

[36] J. Fan, "Nested Virtualization in Azure," https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/, Microsoft Corp., 2017.
[37] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux J.*, 2004.
[38] J. Fotheringham, "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store," *CACM*, 1961.
[39] J. Gandhi *et al.*, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," in *MICRO*, 2014.
[40] J. Gandhi *et al.*, "Agile Paging: Exceeding the Best of Nested and Shadow Paging," in *ISCA*, 2016.
[41] J. Gandhi *et al.*, "Range Translations for Fast Virtual Memory," *IEEE Micro*, 2016.
[42] S. Gerber *et al.*, "Not Your Parents' Physical Address Space," in *HotOS*, 2015.
[43] Google, Inc., "Compute Engine: Enabling Nested Virtualization for VM Instances," https://cloud.google.com/compute/docs/instances/enable-nested-virtualization-vm-instances.
[44] Graph 500, "Graph 500 Large-Scale Benchmarks," http://www.graph500.org/.
[45] M. Gupta *et al.*, "Reliability-Aware Data Placement for Heterogeneous Memory Architecture," in *HPCA*, 2018.
[46] S. M. Hand, "Self-Paging in the Nemesis Operating System," in *OSDI*, 1999.
[47] S. Haria *et al.*, "Devirtualizing Memory in Heterogeneous Systems," in *ASPLOS*, 2018.
[48] Harshad Kasture and Daniel Sanchez, "TailBench Benchmark Suite," http://tailbench.csail.mit.edu/.
[49] G. Heiser *et al.*, "The Mungi Single-Address-Space Operating System," *SPRE*, 1998.
[50] Hewlett-Packard Company, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual, Third Edition*, 1994.
[51] P. Hornyack *et al.*, "A Study of Virtual Memory Usage and Implications for Large Memory," Univ. of Washington, Tech. Rep., 2013.
[52] J. Huang *et al.*, "Unified Address Translation for Memory-Mapped SSDs with FlashMap," in *ISCA*, 2015.
[53] Intel Corp., "5-Level Paging and 5-Level EPT," white paper, 2017.
[54] Intel Corp., *Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3: System Programming Guide*, 2019.
[55] International Business Machines Corp, *PowerPC® Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors*, 2005.
[56] B. Jacob and T. Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE Micro*, 1998.
[57] X. Jiang *et al.*, "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in *HPCA*, 2010.
[58] M. F. Kaashoek *et al.*, "Application Performance and Flexibility on Exokernel Systems," in *SOSP*, 1997.
[59] V. Karakostas *et al.*, "Redundant Memory Mappings for Fast Access to Large Memories," in *ISCA*, 2015.
[60] V. Karakostas *et al.*, "Energy-Efficient Address Translation," in *HPCA*, 2016.
[61] V. Karakostas *et al.*, "Performance Analysis of the Memory Management Unit Under Scale-Out Workloads," in *IISWC*, 2014.
[62] S. Kaxiras and A. Ros, "A New Perspective for Efficient Virtual-Cache Coherence," in *ISCA*, 2013.
[63] T. Kilburn *et al.*, "One-Level Storage System," *IRE Trans. Electronic Computers*, 1962.
[64] J. S. Kim *et al.*, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.
[65] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
[66] G. Klein *et al.*, "seL4: Formal Verification of an OS Kernel," in *SOSP*, 2009.
[67] K. C. Knowlton, "A Fast Storage Allocator," *CACM*, 1965.
[68] O. Kocberber *et al.*, "Meet the Walkers: Accelerating Index Traversals for In-memory Databases," in *MICRO*, 2013.
[69] M. K. Kumar *et al.*, "Latr: Lazy Translation Coherence," in *ASPLOS*, 2018.
[70] Y. Kwon *et al.*, "Coordinated and Efficient Huge Page Management with Ingens," in *OSDI*, 2016.
[71] Y. Kwon *et al.*, "Ingens: Huge Page Support for the OS and Hypervisor," *OSR*, 2017.
[72] B. C. Lee *et al.*, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
[73] D. Lee *et al.*, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
[74] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[75] C. Li *et al.*, "A Framework for Memory Oversubscription Management in Graphics Processing Units," in *ASPLOS*, 2019.
[76] W. Li *et al.*, "A Page-Based Hybrid (Software-Hardware) Dynamic Memory Allocator," *CAL*, 2006.
[77] W. Li *et al.*, "Feasibility of Decoupling Memory Management from the Execution Pipeline," *J. Syst. Archit.*, 2007.
[78] Y. Li *et al.*, "Utility-Based Hybrid Memory Management," in *CLUSTER*, 2017.
[79] A. Lindstrom *et al.*, "The Grand Unified Theory of Address Spaces," in *HotOS*, 1995.
[80] S.-L. Lu *et al.*, "Improving DRAM Latency with Dynamic Asymmetric Subarray," in *MICRO*, 2015.
[81] C.-K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.
[82] H. Luo *et al.*, "CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off," in *ISCA*, 2020.
[83] Y. Luo *et al.*, "Characterizing Application Memory Error Vulnerability to Optimize Data Center Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
[84] M. K. McKusick *et al.*, *The Design and Implementation of the FreeBSD Operating System.* Addison-Wesley Professional, 2014.
[85] T. Merrifield and H. R. Taheri, "Performance Implications of Extended Page Tables on Virtualized x86 Processors," in *VEE*, 2016.
[86] M. R. Meswani *et al.*, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories," in *HPCA*, 2015.
[87] J. Meza *et al.*, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," in *WEED*, 2013.
[88] Micron Technology, Inc., *2Gb: x4, x8, x16 DDR3 SDRAM Data Sheet*, 2016.
[89] MonetDB B.V., "MonetDB Column Store," https://www.monetdb.org/.
[90] J. Navarro *et al.*, "Practical, Transparent Operating System Support for Superpages," in *OSDI*, 2002.
[91] Neo4j, Inc., "Neo4j Graph Platform," https://neo4j.com/.
[92] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *NSDI*, 2013.
[93] Oracle Corp., "TimesTen In-Memory Database," https://www.oracle.com/database/technologies/related/timesten.html.
[94] M.-M. Papadopoulou *et al.*, "Prediction-Based Superpage-Friendly TLB Designs," in *HPCA*, 2015.
[95] C. H. Park *et al.*, "Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations," in *ISCA*, 2017.
[96] E. Perelman *et al.*, "Using SimPoint for Accurate and Efficient Simulation," in *SIGMETRICS*, 2003.
[97] B. Pham *et al.*, "Increasing TLB Reach by Exploiting Clustering in Page Translations," in *HPCA*, 2014.
[98] B. Pham *et al.*, "CoLT: Coalesced Large-Reach TLBs," in *MICRO*, 2012.
[99] B. Pham *et al.*, "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?" in *MICRO*, 2015.
[100] B. Pham *et al.*, "Using TLB Speculation to Overcome Page Splintering in Virtual Machines," Rutgers Univ., Tech. Rep. DCS-TR-713, 2015.
[101] B. Pichai *et al.*, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *ASPLOS*, 2014.
[102] J. Picorel *et al.*, "Near-Memory Address Translation," in *PACT*, 2017.
[103] B. Pourshirazi and Z. Zhu, "Refree: A Refresh-Free Hybrid DRAM/PCM Main Memory System," in *IPDPS*, 2016.
[104] J. Power *et al.*, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *HPCA*, 2014.
[105] A. Prodromou *et al.*, "MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-Level Memories," in *HPCA*, 2017.
[106] M. K. Qureshi *et al.*, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.
[107] L. Ramos *et al.*, "Page Placement in Hybrid Memory Systems," in *ICS*, 2011.
[108] S. Raoux *et al.*, "Phase-Change Random Access Memory: A Scalable Technology," *IBM JRD*, 2008.
[109] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *The Bell System Technical Journal*, 1978.
[110] B. F. Romanescu *et al.*, "UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All," in *HPCA*, 2010.
[111] J. H. Ryoo *et al.*, "SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization," in *HPCA*, 2017.
[112] J. H. Ryoo *et al.*, "Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB," in *ISCA*, 2017.
[113] SAFARI Research Group, "Ramulator-VBI — GitHub Repository," https://github.com/CMU-SAFARI/Ramulator-VBI.git.
[114] SAP SE, "SAP HANA: In-Memory Data Platform," https://www.sap.com/products/hana.html.
[115] D. Schatzberg *et al.*, "EbbRT: A Framework for Building Per-Application Library Operating Systems," in *OSDI*, 2016.
[116] V. Seshadri *et al.*, "The Dirty-Block Index," in *ISCA*, 2014.
[117] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[118] V. Seshadri *et al.*, "Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-Grained Memory Management," in *ISCA*, 2015.
[119] S. H. SeyyedAghaei Rezaei *et al.*, "NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories," *CAL*, 2020.
[120] K. K. Shen and J. L. Peterson, "A Weighted Buddy Method for Dynamic Storage Allocation," *CACM*, 1974.
[121] S. Shin *et al.*, "Scheduling Page Table Walks for Irregular GPU Applications," in *ISCA*, 2018.
[122] J. Sim *et al.*, "Transparent Hardware Management of Stacked DRAM as Part of Memory," in *MICRO*, 2014.
[123] D. Skarlatos *et al.*, "Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism," in *ASPLOS*, 2020.
[124] Y. H. Son *et al.*, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.
[125] Standard Performance Evaluation Corp., "SPEC CPU® 2006," https://www.spec.org/cpu2006/.
[126] Standard Performance Evaluation Corp., "SPEC CPU® 2017 Benchmark Suite," https://www.spec.org/cpu2017/.
[127] M. Talluri and M. D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," in *ASPLOS*, 1994.
[128] A. Tumanov *et al.*, "Asymmetry-Aware Execution Placement on Manycore Chips," in *SFMA*, 2013.
[129] N. Vijaykumar *et al.*, "The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs," in *ISCA*, 2018.
[130] N. Vijaykumar *et al.*, "Zorua: A Holistic Approach to Resource Virtualization in GPUs," in *MICRO*, 2016.
[131] N. Vijaykumar *et al.*, "A Case for Richer Cross-Layer Abstractions: Bridging the Semantic Gap with Expressive Memory," in *ISCA*, 2018.
[132] D. Wentzlaff and A. Agarwal, "Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores," *OSR*, 2009.
[133] Z. Yan *et al.*, "Hardware Translation Coherence for Virtualized Systems," in *ISCA*, 2017.
[134] H. Yoon *et al.*, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
[135] X. Yu *et al.*, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," in *MICRO*, 2017.
[136] Z. Yu *et al.*, "Labeled RISC-V: A New Perspective on Software-Defined Architecture," in *CARRV*, 2017.
[137] L. Zhang *et al.*, "Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation," in *ICS*, 2010.
[138] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *PACT*, 2009.
[139] T. Zheng *et al.*, "SIPT: Speculatively Indexed, Physically Tagged Caches," in *HPCA*, 2018.