2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00047

Rebooting Virtual Memory with Midgard

Siddharth Gupta
EcoCloud, EPFL
siddharth.gupta@epfl.ch

Atri Bhattacharyya
EcoCloud, EPFL
atri.bhattacharyya@epfl.ch

Yunho Oh*
Sungkyunkwan University
yunho.oh@skku.edu

Abhishek Bhattacharjee
Yale University
abhishek@cs.yale.edu

Babak Falsafi
EcoCloud, EPFL
babak.falsafi@epfl.ch

Mathias Payer
EcoCloud, EPFL
mathias.payer@epfl.ch

***Abstract*—Computer systems designers are building cache hierarchies with higher capacity to capture the ever-increasing working sets of modern workloads. Cache hierarchies with higher capacity improve system performance but shift the performance bottleneck to address translation. We propose Midgard, an intermediate address space between the virtual and the physical address spaces, to mitigate address translation overheads without program-level changes.**

Midgard leverages the operating system concept of virtual memory areas (VMAs) to realize a single Midgard address space where VMAs of all processes can be uniquely mapped. The Midgard address space serves as the namespace for all data in a coherence domain and the cache hierarchy. Because real-world workloads use far fewer VMAs than pages to represent their virtual address space, virtual to Midgard translation is achieved with hardware structures that are much smaller than TLB hierarchies. Costlier Midgard to physical address translations are needed only on LLC misses, which become much less frequent with larger caches. As a consequence, Midgard shows that instead of amplifying address translation overheads, memory hierarchies with large caches can reduce address translation overheads.

Our evaluation shows that Midgard achieves only 5% higher address translation overhead as compared to traditional TLB hierarchies for 4KB pages when using a 16MB aggregate LLC. Midgard also breaks even with traditional TLB hierarchies for 2MB pages when using a 256MB aggregate LLC. For cache hierarchies with higher capacity, Midgard's address translation overhead drops to near zero as secondary and tertiary data working sets fit in the LLC, while traditional TLBs suffer even higher degrees of address translation overhead.

***Index Terms*—virtual memory, address translation, memory hierarchy, virtual caches, datacenters, servers**

I. Introduction

We propose, design, and evaluate Midgard,[1] an experiment in future-proofing the virtual memory (VM) abstraction from performance and implementation complexity challenges in emerging big-memory systems.

VM simplifies the programming model by obviating the need for programmer-orchestrated data movement between memory devices and persistent storage [9], offers "a pointer is a pointer everywhere" semantics across multiple CPU cores [50] and accelerators (e.g., GPUs [49], FPGAs [33], NICs [41], ASICs [31], [51]). Also, VM is the foundation of access control and memory protection mechanisms ubiquitous to modern computer systems security.

Unfortunately, VM is today plagued with crippling performance and complexity challenges that undermine its programmability benefits. The central problem is that computer architects are designing systems with increasingly higher-capacity cache hierarchies and memory. The latter improves system performance in the face of big-data workloads [4], [27], [28], [50], as evidenced by recent work on die-stacking, chiplets, DRAM caches [24], [25], [61], and non-volatile byte-addressable memories [13], [19]. However, they also shift the performance bottleneck to virtual-to-physical address translation, which can consume as much as 10-30% of overall system performance [4], [28], [50], [51].

Systems architects are consequently designing complex address translation hardware and Operating System (OS) support that requires significant on-chip area and sophisticated heuristics. The complex hardware and OS support pose verification burdens despite which design bugs still abound [36]. Individual CPU cores (and recent accelerators) integrate large two-level TLB hierarchies with thousands of entries, separate TLBs at the first level for multiple page sizes, and skew/hash-rehash TLBs at the second level to cache multiple page sizes concurrently [14], [43], [54]. These in turn necessitate a staggering amount of OS logic to defragment memory to create 'huge pages' [47], [48], [67] and heuristics to determine when to create, break, and migrate them [35], [40], [59], [60]. Because huge page heuristics can lead to performance pathologies and hence are not a panacea, processor vendors also integrate specialized MMU cache per core to accelerate the page table walk process [3], [7]. Specialized per-core TLBs and MMU cache in turn necessitate sophisticated coherence protocols in the OS (i.e., shootdowns) that are slow and buggy, especially with the adoption of asynchronous approaches to hide shootdown overheads at higher core and socket counts in modern servers [2], [34], [37].

We circumvent these problems by asking the following questions: Larger cache hierarchies (i.e., L1-LLCs) traditionally amplify VM overheads, but could they actually mitigate VM overheads if we redirect most translation activity to them rather than to specialized translation hardware? This question is inspired in part by prior work on virtual cache hierar-

*This work was done while the author was at EPFL.
[1]The middle realm between Asgard and Helheim in Norse mythology.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00047

chies [10], [11], [16] and in-cache address translation [65], which reduce address translation pressure by deferring the need for physical addresses until a system memory access. Unfortunately, they also compromise programmability because of problems with synonyms/homonyms, and reliance on inflexible fixed-size segments. While approaches like single address space OSes [32] tackle some of these problems (i.e., removal of synonyms/homonyms), they require recompilation of binaries to map all data shared among processes into a unified virtual address space. What is needed is a programmer-transparent intermediate address space for cache hierarchies – without the shortcomings of homonyms, synonyms, or fixed-size segments – that requires a lightweight conversion from virtual addresses and a heavier translation to physical addresses only when accessing the system memory.

We propose the Midgard abstraction as this intermediate address space by fusing the OS concept of virtual memory areas (VMAs) into hardware for the first time. Applications view memory as a collection of a few flexibly sized VMAs with certain permissions. We show that it is possible to create a single intermediate Midgard address space where VMAs of various processes can be uniquely mapped. This unique address space serves as a namespace for all data in a coherence domain and cache hierarchies. All accesses to the cache hierarchy must be translated from the program's virtual address to a Midgard address. However, the latter can be accomplished using translation structures that are much smaller than TLBs because there are far fewer VMAs ($\sim$10) than pages in real-world workloads. Translation from Midgard to physical addresses can be filtered to only situations where an access misses in the coherent cache hierarchy. Therefore, instead of amplifying address translation overheads, larger cache hierarchies can now be leveraged to *reduce* them.

We quantify Midgard's performance characteristics over cache hierarchies ranging in size from tens of MBs to tens of GBs and show that even modest MB-scale SRAM cache hierarchies filter the majority of memory accesses, leaving only a small fraction of memory references for translation from Midgard to physical addresses. We characterize VMA counts as a function of dataset size and thread count and confirm that low VMA counts mean a seamless translation from virtual to Midgard addresses. Using average memory access time (AMAT) analysis, we show that LLC capacities in the tens of MBs comfortably outperform traditional address translation and that at hundreds of MBs, they even outperform huge pages. In our evaluation, Midgard reaches within 5% of address translation overhead of conventional 4KB-page TLB hierarchies for a 16MB LLC and breaks even with 2MB-page TLB hierarchies for a 256MB LLC. Unlike TLB hierarchies exhibiting higher overhead with larger cache hierarchies, Midgard's overhead drops to near zero as secondary and tertiary data working sets fit in the cache hierarchies. Finally, we show how, even for pessimistic scenarios with small LLCs, Midgard can be augmented with modest hardware assistance to achieve competitive performance with traditional address translation.

This paper is the first of several steps needed to demonstrate a fully working system with Midgard. In this paper, we focus on a proof-of-concept software-modeled prototype of key architectural components. Future work will address the wide spectrum of topics needed to realize Midgard in real systems, including (but not restricted to) OS support, verification of implementation on a range of architectures, and detailed circuit-level studies of key hardware structures.

II. Virtual Memory

A. The Virtual Memory Abstraction

Modern OSes and toolchains organize the program virtual address space into VMAs, which are large contiguous regions representing various logical data sections (e.g., code, heap, stack, bss, mapped files). Each VMA consists of a base and bound address, a set of permissions, and other optional properties. VMAs adequately represent the program's logical properties without actually tying them to the underlying physical resources. Modern programs regularly use a few hundred VMAs, but only a handful (i.e., $\sim$10) are frequently accessed [20], [38], [67], and thus constitute the working set.

The OS is responsible for allocating physical memory to the program and creating mappings from virtual to physical addresses. While a trivial approach might be to map a VMA to an identically sized contiguous region of physical memory, the latter results in external fragmentation. Moreover, as physical memory is a constrained resource, the OS targets optimizing towards efficient memory capacity management. Therefore, current systems divide virtual and physical memory into small, fixed-size regions called pages and frames, respectively. And then, the systems map virtual pages to physical frames, effectively utilizing physical memory capacity. A page becomes the smallest unit of memory allocation, and each VMA's capacity is forced to be a page-size multiple by the OS. Typically, programs can continue using the VMA abstraction without directly encountering the page-based management of physical memory by the OS.

Current VM subsystems combine access control with translation. Each memory access needs to undergo a permission check or access control. Additionally, locating a page in the physical memory device requires virtual to physical address translation. A page table stores the virtual to physical page mappings for a process. While VMAs dictate the permission management granularity, the permission bits are duplicated for each page and stored in the page tables. TLBs are hardware structures that cache the page table entries to provide fast access and perform both access control and address translation.

B. The Address Translation Bottleneck

With the growth in modern data-oriented services, programs are becoming more memory intensive. As online services hosted in the datacenter generate data rapidly, the memory capacity of servers operating on these datasets is growing commensurately already reaching terabytes [23], [56].

Unfortunately, scaling physical memory to terabytes results in linear growth in translation metadata. With 4KB pages, 1TB

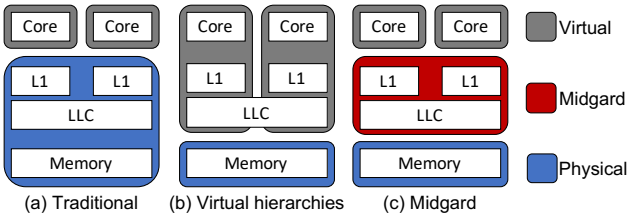

Figure 1: Various arrangements of address spaces. The legend refers to various address spaces.

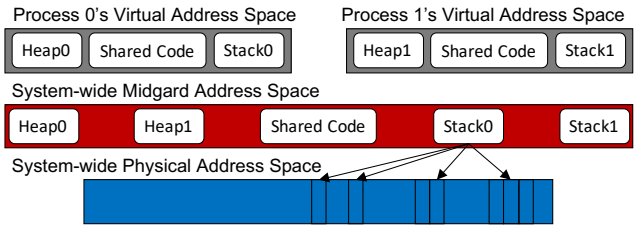

Figure 2: Mapping from the virtual to the Midgard address space in units of VMAs and mapping from Migard address spaces to physical address spaces in units of pages. The cache hierarchy is placed in the Midgard address space.

of physical memory requires 256M mappings. Even with high locality (e.g., 3% of the data captures most of the memory accesses in server workloads [61]), the system requires frequent lookups to 8M mappings. While modern systems already provide thousands of TLB entries per core [26], we still *fall short by three orders of magnitude*. Moreover, we cannot provision thousands of TLB entries for every heterogeneous logic that comes in various sizes (e.g., small cores, GPUs, and accelerators). This mismatch in requirements and availability makes virtual memory one of the most critical bottlenecks in memory scaling.

Each TLB miss incurs a long-latency page table walk. Larger memory capacity often means larger page tables and walk latency, thus further increasing the TLB miss overhead. Intel is already shifting to 57-bit virtual address spaces, which require 5-level page table walks, increasing both page table walk latency and traffic [22]. While there have been proposals to parallelize page table walks for reducing latency [38], [55], the incurred latency remains a performance bottleneck. Overall, core-side long-latency operations are a problem and undermine the hierarchical structure of memory as complex operations should be delegated to the bottom of the hierarchy.

Huge pages were introduced to increase TLB reach. Unfortunately, huge pages are difficult to use transparently in programs without causing internal fragmentation [14]. More importantly, letting programs guide page allocation further erodes the VM abstraction. Ideally, programs using large VMAs should experience iso-performance as using huge pages without explicitly requesting them. Finally, huge pages add alignment constraints to a VMA's base address.

Furthermore, as a result of the end of DRAM scaling [53], the memory is becoming increasingly heterogeneous. Migrating pages dynamically among heterogeneous devices (e.g., DRAM, Persistent Memory, High-Bandwidth Memory, and Flash) for performance gains requires page mapping modifications, which results in expensive, global TLB shootdowns to invalidate the corresponding TLB entries from all cores.

C. Virtual Cache Hierarchies and Single Address Space OSes

Prior work observes that access control is required for every memory access, while translation to physical addresses is only required for physical memory accesses. However, traditional systems translate a virtual address to a physical address and use it as an index in the cache hierarchy because physical addresses form a unique namespace (Figure 1a). This early translation forces the core-side TLBs to operate at page granularity, while pages are only required for efficient capacity management on the memory side.

Virtual cache hierarchies [10], [11], [16] aim to delay this translation to the memory side, thus removing the complex TLB hierarchy from the critical path of every memory access and away from the core. The process-private virtual addresses are used as a namespace to index the cache hierarchies (Figure 1b). Virtual hierarchies are difficult to incorporate in a modern system because of the complexity of resolving synonyms and homonyms across virtual addresses and implementing OS-level access control. While there have been numerous proposals for mechanisms and optimizations to implement virtual cache hierarchies, their implementation complexity remains high, thus preventing mainstream adoption.

Single address space operating systems [32] avoid synonyms and homonyms by design, thus making virtual cache hierarchies easier to adopt. Instead of process-private virtual address spaces, these OSes divide a single virtual address space among all the processes, identify shared data among processes and represent them with unique virtual addresses. Unfortunately, implementing a single address operating system requires significant modifications to programming abstractions, which is a key obstacle to its adoption.

III. The Midgard Abstraction

Midgard is based on three conceptual pillars. First, Midgard enables the placement of the coherent cache hierarchy in a namespace (Figure 1c) that offers the programmability of traditional VM. Second, Midgard quickly translates from virtual addresses to this namespace, permitting access control checks along the way and requiring significantly fewer hardware resources than modern per-core TLBs and MMU cache hierarchies. Third, translating between Midgard addresses and physical addresses requires only modest augmentation of modern OSes. Midgard filters heavyweight translation to physical addresses to only those memory references that miss in the LLC. Sizing LLCs to capture workload working sets also naturally enables better performance than traditional address translation.

A. Building Midgard atop Virtual Memory Areas

Midgard injects VMAs directly into hardware to realize the address space decoupling shown in Figure 2. Instead of a single layer of mapping from per-process virtual pages to a system-wide set of physical frames, logic between CPUs and the cache hierarchy, which we call the "front side", maps per-process virtual pages to a system-wide Midgard address space in the unit of VMAs. Logic after the cache hierarchy, which we call the "back side", maps VMA-size units from the Midgard address space to page-size physical frames.

Translating at the granularity of VMAs is beneficial because real-world workloads use orders of magnitude fewer VMAs of unrestricted size than pages of fixed size. Whereas traditional VM relies on large specialized TLB and MMU cache hierarchies and multi-level page tables, Midgard incorporates much smaller hardware and OS-level translation tables.

Midgard's ability to defer translation until LLC misses means that relatively costly translations at the page-level granularity can be eliminated except for LLC misses. Our evaluation shows that modestly sized LLCs in the 10s of MBs eliminate the need for heavyweight translations to physical frames for more than 90% of the memory references of modern graph processing workloads. The integration of large multi-GB eDRAM and DRAM caches [26], [57], [61] means that, unlike traditional TLB-based translation, Midgard's LLC filtering of translation requests future-proofs VM's performance.

Midgard obviates the need to compromise programmability. Thanks to the indirection to a unique namespace, Midgard mitigates the homonym and synonym problems and access control limitations of virtual cache hierarchies [10], [11], [16] and in-cache address translation [65]. Midgard avoids modifying existing programming abstractions or recompiling binaries as in single address space OSes [32] or virtual block interfaces [18]. Unlike prior work restricting VM abstractions to fixed-size segments [69], Midgard uses the existing OS-level variable VMA sizes in a flexible manner.

B. Operating System Support for Midgard

Midgard requires that the OS is augmented to map VMAs in per-process virtual address spaces to Midgard memory areas (MMAs) in a single system-wide Midgard address space. The OS must also maintain the tables for VMA to MMA mappings and for MMA mappings to physical frame numbers.

Sizing the Midgard address space: All per-process VMAs are mapped to a single Midgard address space (Figure 2) without synonyms or homonyms; the OS must deduplicate shared VMAs across various processes. VMAs occupy a contiguous region in virtual and Midgard address spaces, and therefore growing a VMA in the virtual address space requires growing its corresponding MMA. To grow, MMAs must maintain adequate free space between one another to maximize their chances of growth without collision and their ensuing (costly) relocation. Fortunately, in practice, many MMAs are partly backed by physical memory and only some are fully backed. In other words, as long as the Midgard address space is adequately larger than the physical address space (e.g., 10-15 bits in our study), MMAs from thousands of processes can be practically accommodated. In the less common case where growing MMAs collide, the OS can either remap the MMA to another Midgard address, which may require cache flushes or splitting the MMA at the cost of tracking additional MMAs.

Tracking VMA to MMA mappings: We add a data structure in the OS, which we call a VMA Table, for VMA to MMA mappings to achieve V2M translation in the front side. Many data structures can be used to implement the VMA Table. Possibilities include per-process VMA Tables, like traditional per-process radix page tables, or system-wide VMA Tables like traditional inverted page tables.

In our Midgard prototype, we use compact per-process data structures that realize arbitrarily sized VMAs spanning *ranges* of virtual addresses. Akin to recent work on range tables [28], we implement VMA Tables with B-tree structures. Each VMA mapping requires a base and a bound virtual address that captures the size of the VMA. VMA mappings also need an offset field which indicates the relative offset between the position of the VMA in the virtual address space and the position of the MMA that it points to in the Midgard address space. Since the base, bound, and offset are all page-aligned, they require 52 bits of storage each in VMA Table entries for 64-bit virtual and Midgard address spaces. VMA Table entries also need permission bits for access control. Each VMA mapping is therefore roughly 24 bytes. Even relatively high VMA counts that range in the $\sim$100s can be accommodated in just a 4KB page, making the VMA Table far smaller than traditional page tables. We leave a detailed study of VMA Table implementations for future work [55].

Tracking MMA to physical frame number mappings: We also add a data structure, which we call a Midgard Page Table, in the OS to map from pages in MMAs to physical frame numbers to support M2P translation in the backside. Unlike the VMA Table, Midgard Page Table needs to map at the granularity of pages in the VMAs to physical frame numbers. Although alternative designs are possible, we use a radix page table [38]. These mappings can be created either at memory allocation time or lazily akin to demand paging. When a page from the MMA is not mapped to a physical frame, lookups in the cache hierarchy will miss. The latter will prompt an M2P translation, which will in turn prompt a page fault, at which point control will be vectored to the OS, which will either lazily create an MMA to physical frame number mapping, initiate demand paging, or signal a segmentation fault.

Although not necessary for correctness, Midgard's performance also benefits from optimizations where the pages holding the Midgard Page Table are allocated contiguously. As explained subsequently in Section III-C, contiguous allocation allows M2P translations to short-circuit lookups of multiple levels of the radix tree used to implement the Midgard Page Table, similar in spirit to the operation of modern per-core hardware paging structure caches [3], [8]. Contiguous allocation is based on the insight that radix page tables are

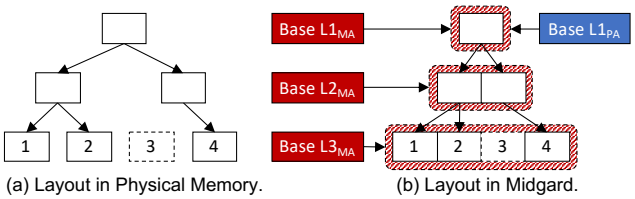

Figure 3: Midgard page table layout using a hypothetical three-level page table with a radix-degree of two.

only sparsely utilized. We fully expand the radix tree and lay out even unmapped pages in a contiguous manner as shown in Figure 3b, enabling short-circuited M2P walks.

C. Hardware Support for Midgard

Midgard requires hardware support within the CPU and memory hierarchy for correct and efficient operation.

Registers to the root of VMA and Midgard Page Table: The VMA Table is mapped into the Midgard address space and exposed to each CPU using per-core *VMA Table Base Registers* which store the Midgard address of the root node of the B-tree. In contrast, a pointer to the physical address of the root of the Midgard Page Table is maintained in dedicated *Midgard Page Table Base Registers* at the memory controllers.

Hardware support for page faults: Page faults due to M2P translation failures signal an exception to the core from which the memory request originated. To maintain precise exceptions, the faulting instruction needs to be rolled back before the exception handler can be executed. Such rollbacks are easier to implement for loads as they are synchronous operations and block until data arrives, than for stores that are asynchronous operations. Modern out-of-order processors retire stores from the reorder buffer once their values and addresses are confirmed. In other words, stores wait for completion in the store buffer while the out-of-order execution moves ahead. If the M2P translation for such a store fails, current speculation techniques cannot roll back the store, leading to imprecise exceptions. Midgard necessitates hardware extensions to buffer speculative state in the pipeline to cover the store buffer [15], [62]. For each store in the store buffer, we need to record the previous mappings to the physical register file, permitting rollback to those register mappings in case of an M2P translation failure. We leave details of such mechanisms for future work.

Updating the Midgard Page Table: The Midgard Page Table entries (like conventional page tables) track access and dirty bits to identify recently used or modified pages. While access bits in TLB-based systems can be updated on a memory access, modern platforms opt for setting the access bit only upon a page walk and a TLB entry fill. Midgard updates the access bit upon an LLC cache block fill and the corresponding page walk. Because access bits present approximate access recency information and are reset periodically by the OS, coarse-grained updates to the access bits may be acceptable for large memory systems [66] as page evictions are infrequent. In contrast, dirty bits require precise information and are updated upon an LLC writeback and the corresponding page walk.

Accelerating V2M translation: We use per-core Virtual Lookaside Buffers (VLBs) akin to traditional TLBs to accelerate V2M translation. Like VMA Tables, VLBs offer access control and fast translation from virtual to Midgard addresses at a VMA granularity. Because real-world programs use orders of magnitude fewer VMAs than pages, it suffices to build VLBs with tens of entries rather than TLBs with thousands of entries. VLBs are significantly smaller than TLBs, but they do rely on range comparisons to determine the VMA corresponding to a virtual address. As we discuss in Section IV, care must be taken to ensure that these comparisons do not unduly slow down the VLB access.

Accelerating M2P translation: Because the majority of memory references are satisfied from the cache hierarchy, M2P translations are far less frequent than V2M translations. We nevertheless propose two optimizations to accelerate M2P translation cost when necessary.

First and foremost, the contiguous layout of the pages used to realize the Midgard Page Table permits short-circuiting of the radix tree walk. A contiguous layout enables the identification of the desired required entry in each level based purely on the Midgard address. The backside logic that performs the M2P translation can calculate the Midgard address of the required entry using the target Midgard address of the data in tandem with the base address of the last-level contiguous region. Contiguous allocation permits walk short-circuiting similar to the way in which paging structure caches are looked up using just the data's target virtual address [3], [8]. If this entry is present in the cache hierarchies, it can be looked up directly using the calculated Midgard address, thus skipping all the other levels in the Midgard Page Table. If the requested entry is not present in the cache hierarchies, the backside Midgard Page Table traversal logic looks up the level above to find the Midgard Page Table's last-level entry's physical address and fetch it from memory. For each subsequent miss, this algorithm traverses the Midgard Page Table tree towards the root, as shown in Figure 4. The physical address of the root node is used when none of the levels are present in the cache hierarchies.

We also explore a second optimization, a system-wide Midgard Lookaside Buffer (MLB), to cache frequently used entries from the Midgard Page Table with a mapping, access control information and access/dirty bits. MLBs are optional and useful really only for power/area-constrained settings where the capacity of the LLC is relatively small (i.e., <32MB). MLBs are similar to traditional set-associative TLBs and are accessed with a Midgard address upon an LLC miss. An MLB miss leads to a Midgard Page Table walk. Because the LLC already absorbs most of the temporal locality, MLB lookups are expected to be primarily spatial in nature and fundamentally different from those of traditional TLBs, requiring typically only a few entries per thread for

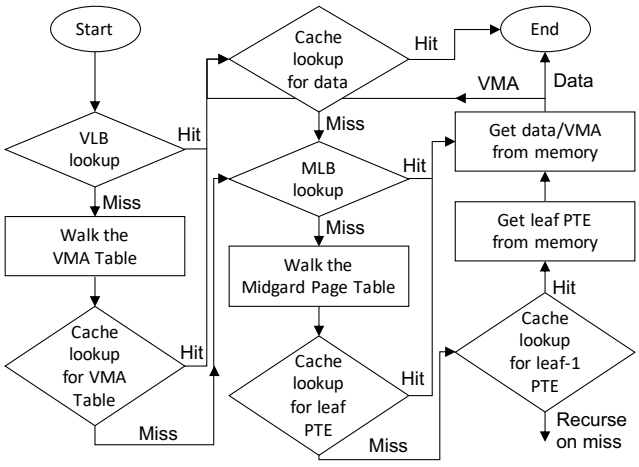

Figure 4: Logical flow of a memory reference from V2M translation through M2P translation. This diagram assumes the use of the optional MLB.

streaming accesses to pages. Much as in TLBs, MLB entries also maintain status bits to implement a replacement policy.

D. Summary of the Two-Step Address Translation

Figure 4 summarizes the overall flow of Midgard for the case where all hardware and software features – even optional ones like the MLB – are included. The sequence of steps shows V2M translation, which begins with a VLB lookup and falls back on a lookup of the VMA Table on a VLB miss. The VMA Table may in turn be absent from the cache hierarchy, in which case an M2P translation for the VMA Table is performed. Once that is satisfied (or in the event of a VLB hit or VMA Table hit in the cache hierarchy), the memory reference is replayed and data lookup proceeds. Only if lookup misses in all cache hierarchies does an M2P translation begin. At this point, the MLB can be consulted. In the event of an MLB miss, a walk commences for the Midgard Page Table. This walk can be optimized via short-circuited paging structure cache-style lookups of the LLC to accelerate the Midgard Page Table walk. We assume that traversals of the VMA Table and the Midgard Page Table are managed entirely in hardware without OS intervention, mirroring hardware page table walks in traditional systems.

E. Other Conceptual Benefits

So far in this section, we have focused on Midgard's ability to reduce the frequency of heavyweight virtual to physical address translations in traditional systems via a level of indirection. There are, however, a number of additional benefits that are possible because of Midgard.

Mitigation of shootdown complexity: VMAs are much coarser-grained than pages, allocated/deallocated less frequently than pages, and suffer far fewer permission changes than pages. For these reasons, switching the front side of the system to a VMA-based VLB from a page-based TLB means that expensive and bug-prone OS-initiated shootdowns become

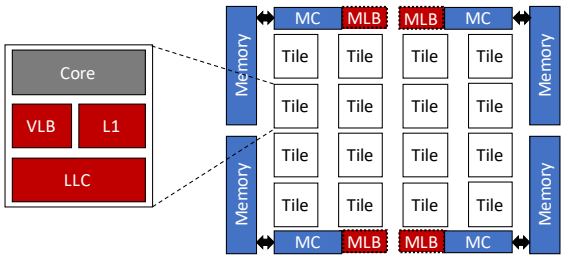

Figure 5: The anatomy of a multicore system with Midgard. Dashed lines indicate that MLB is optional.

far less common in Midgard. Moreover, the fact that there is no need for dedicated translation hardware on the back side for M2P translation for most reasonably sized LLCs means that OS-initiated shootdowns can be entirely elided at that level. Even for conservative scenarios, where only a small LLC is permitted and a single system-wide MLB is required, we find shootdowns to be far less expensive versus the broadcast-based mechanisms that need to be implemented to maintain coherence across multiple TLBs and MMU cache hierarchies private to individual CPUs. The relative ease with which permissions can now be changed opens up optimization opportunities in data sharing and security outside of the scope of this paper.

Flexible page/frame allocations: Midgard allows independent allocation of V2M and M2P translations. Independent allocations at the two levels enable different allocation granularities at different levels. For example, virtual memory might be allocated at 2MB chunks, while physical memory is allocated in 4KB frames. With a larger V2M translation granularity, virtual and Midgard addresses share more bits thereby increasing the L1 set-index bits known prior to translation, ameliorating a limitation of modern VIPT (and our VIMT) cache hierarchies and allowing the L1 cache to scale in capacity [45]. Implementations of guard pages [21], [52] can also be improved with Midgard. Logically united VMAs traditionally separated by a guard page can be merged as one in a Midgard system, and the guard page can be left as unmapped in the M2P translation.

IV. Implementing Midgard

In this section, we discuss implementation details of per-core VLBs and mechanisms to accelerate M2P translation. Without loss of generality, we assume a system with 64-bit virtual addresses (mapping 16 exabytes), 52-bit physical addresses (mapping 4 petabytes), 64-bit Midgard addresses and that the OS allocates memory at a 4KB granularity. Figure 5 depicts the anatomy of a Midgard-based cache-coherent 4×4 multicore with various system components following the same color coding as Figure 5 to indicate which are affected by Midgard. The specific parameters for the system size and configuration are only selected for discussion purposes.

A. Virtual Lookaside Buffers (VLBs)

VLBs benefit from abundant temporal and spatial locality in memory accesses. Real-world applications use few active VMAs requiring a few VLB entries to cache them. Nevertheless, unlike TLBs which require matching a page number, VLBs perform range lookups which are fundamentally slower in hardware. Because each memory access requires a V2M translation, VLB lookups must also match the core's memory access rate.

To understand the impact of the range lookup on VLB access time, we build and analyze VLB hardware in a 22nm CMOS library. Each VLB entry has a base and bound register which require range comparisons with the virtual address. The range comparison latency is largely determined by comparison bit-width and the number of VMA entries concurrently compared. For 64-bit virtual address spaces, the base and bound registers are 52 bits as the VMA capacity is a multiple of 4KB. A 16-entry VLB has an access time of 0.47ns, consuming the entire clock cycle of our 2GHz clock. We prefer leaving greater slack between VLB access time and clock cycle time so that we can accommodate optimizations like adding more ports to the VLB or supporting higher clock rates.

We therefore design a two-level VLB hierarchy, similar to recently proposed range TLBs [28]. The L1 VLB is a traditional fixed-size page-based TLB while the L2 VLB is a fully associative VMA-based Range TLB, as shown in Figure 6. As the L1 VLB requires equality comparison, it can be sized to meet the core's timing constraints. The L1 VLB filters most of the translation requests while maintaining the traditional translation datapath of the core. The L2 VLB performs a range comparison but only on L1 VLB misses and can therefore tolerate a higher (up to 9 cycles [17]) access latency with more entries to capture all active VMAs.

Once VLB translation succeeds, the cache hierarchy is accessed using Midgard addresses. Recall that in our implementation, Midgard addresses are 12 bits wider than physical addresses. Therefore, a cache (or directory) in the Midgard address space integrates tags that are 12 bits longer as compared to a cache in the physical address space. Assuming 64KB L1 instruction and data caches, a 1MB LLC cache per tile, 64-byte blocks and full-map directories (with a copy of the L1 tags), our 16-core Midgard example system maintains tags for ∼320K blocks. With 12 bits extra per tag, the system requires an additional 480KB of SRAM to support Midgard.

Finally, the dimensions and structure of the VMA Table determine the number of memory references for its traversal on a VLB miss. Because each VMA Table entry is 24 bytes, two 64-byte cache lines can store roughly five VMA entries which can accommodate a VMA Table as a balanced three-level B-Tree [28] with 125 VMA mappings. The non-leaf entries in the B-Tree contain a Midgard pointer to their children instead of an offset value. A VMA Table walk starts from the root node, compares against the base and bound registers, and follows the child pointer on a match until arriving at the final leaf entry.

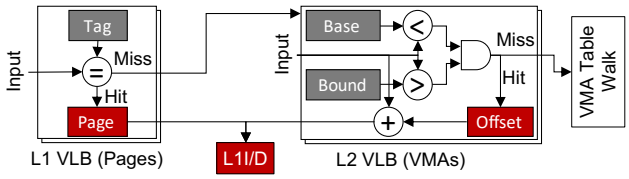

Figure 6: Two-level VLB design.

B. Walking the Midgard Page Table

The dimensions and structure of the Midgard Page Table determine the number of memory references for its traversal. We use a single traditional radix page table with a degree of 512 to store all the Midgard to physical address mappings at the granularity of pages. As our Midgard address space is 64 bits, we need a 6-level radix table. Traversing the Midgard Page Table therefore requires two sequential memory references beyond what is needed for four-level page tables in traditional VM. The contiguous layout of the Midgard Page Table enables short-circuited walks and that effectively hides the latency of the deeper Midgard Page Table.

Midgard Page Table entries must be cacheable for fast lookup. On M2P translation, the back-side logic responsible for walking the Midgard Page Table generates a request to the cache hierarchy. LLC slices are closer to the back-side walker logic than L1 or intermediate cache levels. Therefore, memory references from the back-side walker are routed to the LLC. In response, the coherence fabric retrieves the most recently updated copy of the desired Midgard Page Table entry, which may be in any of the cache hierarchies. In other words, the coherence fabric satisfies M2P walks in the same way that it satisfies traditional page table walks from IOMMUs [42].

The latency of the M2P walk is determined by the level of the cache hierarchy that the coherence fabric finds the desired Midgard Page Table entry in. If, for example, the OS has recently accessed the Midgard Page Table entry, it may be in the L1 or intermediate levels of cache. But since Midgard Page Table walks are more frequent than OS changes to the Midgard Page Table entry, lookups for the Midgard Page Table entry usually hit in the LLC.

Since cache hierarchies operate on Midgard addresses, Midgard Page Table must also be mapped into the Midgard address space to be cacheable. We reserve a memory chunk within the Midgard address space for the Midgard Page Table. To calculate the size of this memory chunk, consider that the last level of the page table can contain 2^{52} pages and can thus occupy 2^{55} bytes. Because the Midgard Page Table is organized as a radix tree with degree 512, its total size must therefore be no larger than 2^{56} bytes. We reserve a memory chunk of 2^{56} bytes in the Midgard address space for the Midgard Page Table, and use the *Midgard Base Register* to mark the start of this address chunk (e.g., Base $L3_{MA}$ in Figure 3b).

Recall that the contiguous layout of the Midgard Page Table permits short-circuited lookups of the radix tree as well as parallel lookups of each level of the radix tree. Short-circuited

lookups reduce the latency of the Midgard Page Table walk and are a uniformly useful performance optimization. Parallel lookups of the Midgard Page Table, on the other hand, can potentially reduce walk latency (especially if LLC misses for lookups of deeper levels of the Midgard Page Table are frequent) but can also amplify LLC lookup traffic. We studied the potential utility of parallel lookups for multiple levels of the Midgard Page Table and found that the average page walk latency difference is small for the system configurations that we evaluate.

C. The Midgard Lookaside Buffer (MLB)

In the uncommon case where LLCs are relatively small, we can integrate a single central MLB shared among cores in the back-side. Centralized MLBs are better than per-core MLBs in many ways. Centralized MLBs offer the same utilization benefits versus private MLBs that shared TLBs enjoy versus private TLBs; by allocating hardware resources to match the needs of the individual cores rather than statically partitioning the resources into a fixed number of entries. Centralized MLBs eliminate replication of mappings that would exist in per-core MLBs. Centralized MLBs also simplify shootdown logic in the OS, by eliding the need for invalidation broadcasts across multiple MLBs.

The centralized MLB can be sliced to improve access latency and bandwidth, similar to LLCs. As modern memory controllers use page-interleaved policies, we can divide the MLB into slices and colocate them with the memory controllers as shown in Figure 5. In this manner, colocating MLB slices with the memory controller can benefit the overall memory access latency as for an MLB hit, the local memory controller can be directly accessed to retrieve the required data from memory.

Finally, MLBs can be designed to concurrently cache mappings corresponding to different page sizes, similar to modern L2 TLBs [43]. Traditional L2 TLBs tolerate longer access latencies and can therefore sequentially apply multiple hash functions, one for each supported page size, until the desired entry is found or a TLB miss is detected [26], [43] by masking the input address according to the page size and then comparing against the tag. Since the MLB resides at the bottom of the memory hierarchy and receives low traffic, it has even more relaxed latency constraints compared to L2 TLBs. A relaxed-latency MLB is therefore ripe for support of huge pages concurrently with traditional 4K pages if necessary.

V. Methodology

We implement Midgard on top of QFlex [44], a family of full-system instrumentation tools built on top of QEMU. Table I shows the detailed parameters used for evaluation. We model a server containing 16 ARM Cortex-A76 [64] cores operating at 2GHz clock frequency, where each core has a 64KB L1 cache and 1MB LLC tile, along with an aggregate 256GB of memory. Each core has a 48-entry L1 TLB and a 1024-entry L2 TLB that can hold translations for 4KB or 2MB pages. For Midgard, we conservatively model an L1 VLB with the same capacity as the traditional L1 TLB along with a 16-entry L2 VLB. All the cores are arranged in a 4x4 mesh architecture with four memory controllers at the corners.

Core	16× ARM Cortex-A76 [64]
Traditional TLB	L1(I,D): 48 entries, fully associative, 1 cycle Shared L2: 1024 entries, 4-way, 3 cycles
L1 Caches	64KB 4-way L1(I,D), 64-byte blocks, 4 cycles (tag+data)
LLC	1MB/tile, 16-way, 30 cycles, non-inclusive
Memory	256GB capacity (16GB per core) 4 memory controllers at mesh corners
Midgard	VLB: L1(I,D): 48 entries, fully associative, 1 cycle L2 (VMA-based VLB): 16 VMA entries, 3 cycles

Table I: System parameters for simulation on QFlex [44].

As Midgard directly relies on the cache hierarchy for address translation, its performance is susceptible to the cache hierarchy capacity and latency. We evaluate cache hierarchies ranging from MB-scale SRAM caches to GB-scale DRAM caches and use AMAT to estimate the overall address translation overhead in various systems.

To approximate the impact of latency across a wide range of cache hierarchy capacities, we assume three ranges of cache hierarchy configurations modeled based on AMD's Zen2 Rome systems [63]: 1) a single chiplet system with an LLC scaling from 16MB to 64MB and latencies increasing linearly from 30 to 40 cycles, 2) a multi-chiplet system with an aggregate LLC capacity ranging from 64MB to 256MB for up to four chiplets with remote chiplets providing a 50-cycle remote LLC access latency backing up the 64MB local LLC, and 3) and a single chiplet system with a 64MB LLC backed by a DRAM cache [57] using HBM with capacities varying from 512MB to 16GB with an 80-cycle access latency. Our baseline Midgard system directly relies on Midgard Page Table walks for performing M2P translations. We also evaluate Midgard with optional architectural support for M2P translation to filter requests for Midgard Page Table walk for systems with conservative cache sizes.

We use Average Memory Access Time (AMAT) as a metric to compare the impact of address translation on memory access time. We use full-system trace-driven simulation models to extract miss rates for cache and TLB hierarchy components, assume constant (average) latency based on LLC configuration (as described above) at various hierarchy levels, and measure memory-level parallelism [12] in benchmarks to account for latency overlap.

To evaluate the full potential of Midgard, we use graph processing workloads including the GAP benchmark suite [6] and Graph500 [39] with highly irregular access patterns and a high reliance on address translation performance.

The GAP benchmark suite contains six different graph algorithm benchmarks: Breadth-First Search (BFS), Betweenness Centrality (BC), PageRank (PR), Single-Source Shortest Path (SSSP), Connected Components (CC), and Triangle Counting (TC). We evaluate two graph types for each of these algorithms: uniform-random (Uni) and Kronecker (Kron). Graph500 is a single benchmark with behavior similar to BFS

	Dataset Size (GB)				Thread Count					
	0.2	0.5	1	2	4	8	12	16	24	32
BFS SSSP	51	51	52	52	52	60	68	76	84	108

Table II: VMA count against dataset size and thread count.

in the GAP suite. The Kronecker graph type uses the Graph500 specifications in all benchmarks. All graphs evaluated contain 128M vertices each for 16 cores [5].

VI. Evaluation

In this section, we first evaluate Midgard's opportunity for future-proofing virtual memory with minimal support for the VMA abstraction. We then present Midgard's performance sensitivity to cache hierarchy capacity with a comparison to both conventional TLB hierarchies and huge pages. We finally present an evaluation of architectural support to enhance Midgard's performance when the aggregate cache hierarchy capacity is limited.

A. VMA Usage Characterization

We begin by confirming that the number of unique VMAs needed for large-scale real-world workloads, which directly dictates the number of VMA entries required by the L2 VLB, is much lower than the number of unique pages. To evaluate how the VMA count scales with the dataset size and the number of threads, we pick BFS and SSSP from the GAP benchmark suite as they exhibit the worst-case performance with page-based translation. Table II depicts the change in the number of VMAs used by the benchmark as we vary the dataset size from 0.2GB to 2GB. Over this range, the VMA count only increases by one, possibly from the change in algorithm going from malloc to mmap for allocating large spaces. The VMA count plateaus when scaling the dataset from 2GB to the full size of 200GB ($\sim 2^{25}$ pages) because larger datasets use larger VMAs without affecting their count.

Table II also shows the required number of VMAs while increasing the number of threads in our benchmarks using the full 200GB dataset. The table shows that each additional thread adds two VMAs comprising a private stack and an adjoining guard page. Because these VMAs are private per thread, their addition does not imply an increase in the working set of the number of L2 VLB entries for active threads.

Finally, Table III depicts the required L2 VLB size for benchmarks. For each benchmark, the table presents the power-of-two VLB size needed to achieve a 99.5% hit rate. In these benchmarks, >90% accesses are to four VMAs including the code, stack, heap and a memory-mapped VMA storing the graph dataset. TC is the only benchmark that achieves the required hit rate with four VLB entries. All other benchmarks require more than four entries but achieve the hit rate with 8, with BFS and Graph500 being the only benchmarks that require more than eight entries. These results corroborate prior findings [38], [67] showing that ∼10 entries are sufficient even for modern server workloads. We therefore conservatively over-provision the L2 VLB with 16 entries in our evaluation.

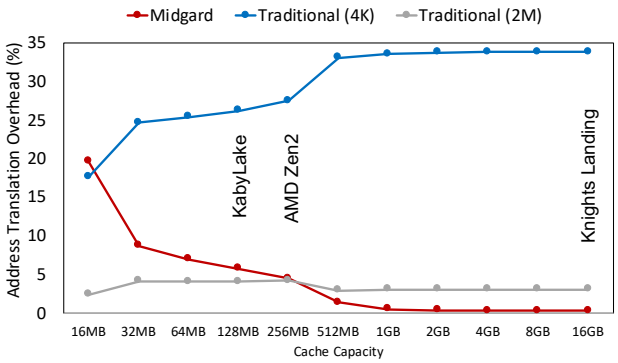

Figure 7: Percent AMAT spent in address translation.

B. Address Translation Overhead

Besides supporting VMA translations directly in hardware, a key opportunity that Midgard exploits is that a well-provisioned cache hierarchy filters the majority of the memory accesses, requiring M2P translation for only a minuscule fraction of the memory requests. Much like in-cache address translation, a baseline Midgard system uses table walks to perform the M2P translation for memory requests that are not filtered by the cache hierarchy. In contrast, a traditional TLB-based system typically requires provisioning more resources (e.g., TLB hierarchies, MMU caches) to extend address translation reach with an increased aggregate cache hierarchy capacity.

Figure 7 compares the overall address translation overhead as a fraction of AMAT between Midgard and TLB-based systems. The figure plots the geometric mean of the address translation overhead across all benchmarks. In this figure, we vary the cache hierarchy configurations (as described in V) in steps to reflect aggregate capacity in recent products – such as Intel Kabylake [26], AMD Zen2 Rome [63], and Intel Knights Landing [57].

The figure shows that address translation overhead in traditional 4KB-page systems running graph workloads with large datasets even for minimally sized 16MB LLCs is quite high at around 17%. As shown in Table III, the L2 TLB misses per thousand instructions (MPKI) in 4KB-page systems is overall quite high in our graph benchmarks (with the exception of BC and TC with Kron graphs). These miss rates are also much higher than those in desktop workloads [29] or scaled down (e.g., 5GB) server workloads [30].

The figure shows that Midgard achieves only 5% higher overall address translation overhead as compared to traditional 4KB-page TLB-based systems for a minimally sized LLC while virtually eliminating the silicon provisioned for per-core 1K-entry L2 TLBs (i.e., ∼16KB SRAM), obviating the need for MMU caches and hardware support for M2P translation and page walks.

As the dataset sizes for server workloads increase, modern servers are now featuring increasingly larger aggregate cache hierarchy capacities [26], [57], [63]. With an increase in aggregate cache capacity, the relative TLB reach in traditional

Benchmark	Traditional L2 TLB MPKI		Required L2 VLB capacity	% Traffic filtered by LLC				Avg. page walk cycles			
				Uni		Kron		Uni		Kron	
	Uni	Kron		32MB	512MB	32MB	512MB	Traditional	Midgard	Traditional	Midgard
BFS	23	29	16	95	99	95	100	51	31	30	30
BC	< 1	< 1	8	100	100	100	100	20	35	20	35
PR	71	68	8	85	100	89	100	45	30	42	30
SSSP	74	70	8	87	98	90	100	47	31	38	30
CC	23	18	8	98	100	97	100	39	34	44	31
TC	62	< 1	4	80	92	100	100	48	30	48	30
Graph500	-	27	16	-	-	96	100	-	-	32	30

Table III: Analysis of miss rate (MPKI) in traditional 4KB-page L2 TLBs, L2 VLB size for 99.5%+ hit rate, M2P traffic filtered, and page walk latency for traditional 4KB-page TLB-based and Midgard systems. Graph500 only uses the Kronecker graph type.

systems decreases while the average time to fetch data decreases due to higher cache hit rates. Unsurprisingly, the figure shows that the address translation overhead for traditional 4KB-page TLB-based systems exhibit an overall increase, thereby justifying the continued increase in TLB-entry counts in modern cores. The figure also shows that our workloads exhibit secondary and tertiary working set capacities at 32MB and 512MB where the traditional 4KB-page system's address translation overhead increases because of limited TLB reach to 25% and 33% of AMAT respectively.

In contrast, Midgard's address translation overhead drops dramatically at both secondary and tertiary working set transitions in the graph thanks to the corresponding fraction of memory requests filtered in the cache hierarchy. Table III also shows the amount of M2P traffic filtered by 32MB and 512MB LLCs for the two working sets. The table shows that 32MB already filters over 90% of the M2P traffic in the majority of benchmarks by serving data directly in the Midgard namespace in the cache hierarchy. With a 512MB LLC, all benchmarks have over 90% of traffic filtered with benchmarks using the Kron graph (virtually) eliminating all translation traffic due to enhanced locality. As a result, with Midgard the resulting address translation overhead (Figure 7) drops to below 10% at 32MB, and below 2% at 512MB of LLC.

Next, we provide a comparison of average page table walk latency between a 4KB-page TLB-based system and Midgard. Because Midgard fetches the leaf page table entries from the caches during a page walk in the common case, on average it requires only 1.2 accesses per walk to an LLC tile which is ($\sim$30 cycles) away (Table III). In contrast, TLB-based systems require four lookups per walk. While these lookups are performed in the cache hierarchy, they typically miss in L1 requiring one or more LLC accesses. As such, Midgard achieves up to 40% reduction in the walk latency as compared to TLB-based systems. BC stands as the outlier with high locality in the four lookups in L1 resulting in a TLB-based average page walk latency being lower than Midgard.

C. Comparison with Huge Pages

To evaluate future-proofing virtual memory with Midgard, we also compare Midgard's performance against an optimistic lower bound for address translation overhead using

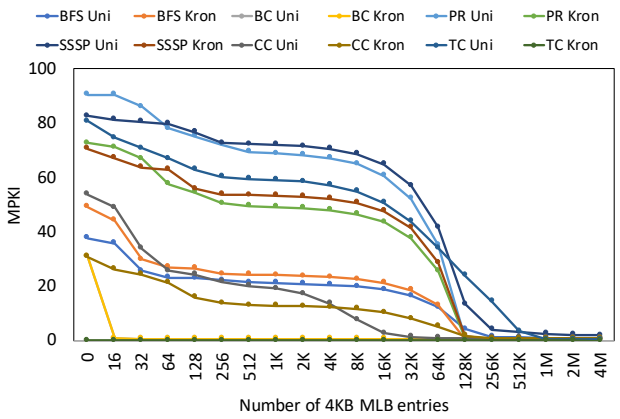

Figure 8: Sensitivity to MLB size for a 16MB LLC.

huge pages. Huge pages provide translation at a larger page granularity (e.g., 2MB or 1GB) and thereby enhance TLB reach and reduce the overall address translation overhead. Prior work [40], [48], [67] indicates that creating and maintaining huge pages throughout program execution requires costly memory defragmentation and frequent TLB shootdowns [1], [2]. Huge pages may also inadvertently cause a performance bottleneck – e.g., when migrating pages in a NUMA system [46]. To evaluate a lower bound, we optimistically assume zero-cost memory defragmentation and TLB shootdown, thus allowing ideal 2MB pages for address translation that do not require migration. We also assume the same number of L1 and L2 2MB TLB entries per core as the 4KB-page system.

Figure 7 also depicts a comparison of address translation overhead between Midgard and ideal 2MB-page TLB-based systems. Not surprisingly, a 2MB-page system dramatically outperforms both TLB-based 4KB-page and Midgard systems for a minimally sized 16MB LLC because of the 500x increase in TLB reach. Much like 4KB-page systems, the TLB-based huge page system also exhibits an increase in address translation overhead with an increase in aggregate LLC capacity with a near doubling in overall address translation overhead from 16MB to 32MB cache capacity.

In contrast to 4KB-page systems which exhibit a drastic drop in TLB reach from 256MB to 512MB caches with the

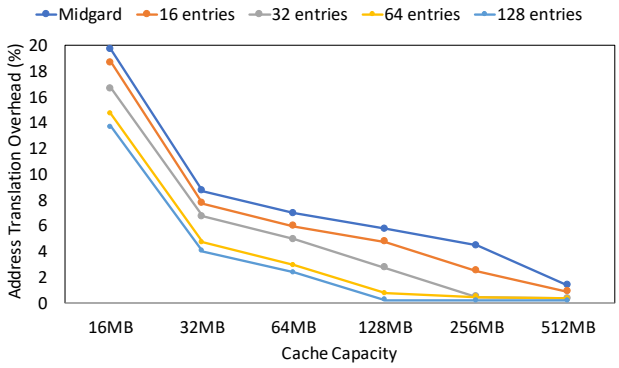

Figure 9: Required MLB size as a function of LLC capacity.

tertiary data working set, huge pages allow for a 32GB overall TLB reach (i.e., 16 cores with 1K-entry L2 TLB) showing little sensitivity to the tertiary working set fitting. Instead, because for the DRAM cache configurations we assume a transition from multiple chiplets to a single chiplet of 64MB backed up by a slower 512MB DRAM cache with a higher access latency, a larger fraction of overall AMAT goes to the slower DRAM cache accesses. As such, the address translation overhead drops a bit for DRAM caches but increases again with an increase in aggregate cache capacity.

In comparison to huge pages, Midgard's performance continuously improves with cache hierarchy capacity until address translation overhead is virtually eliminated. Midgard reaches within 2x of huge pages' performance with 32MB cache capacity, breaks even at 256MB which is the aggregate SRAM-based LLC capacity in AMD's Zen2 Rome [63] products, and drops below 1% beyond 1GB of cache capacity. While Midgard is compatible with and can benefit from huge pages, Midgard does not require huge page support to provide adequate performance as traditional systems do. Overall, Midgard provides high-performance address translation for large memory servers without relying on larger page sizes.

D. Architectural Support for M2P Translation

While we expect future server systems to continue integrating larger cache hierarchies, our memory-access intensive workloads exhibit a non-negligible degree of sensitivity to address translation overhead (i.e., $> 5\%$) with aggregate cache hierarchy capacities of less than 256MB. In this subsection, we evaluate architectural support for M2P translation using MLBs. While MLBs (like TLBs) complicate overall system design, the higher complexity may be justified by the improvements in address translation overhead for a given range of cache hierarchy capacity.

We first analyze the required aggregate MLB size (i.e., the total number of MLB entries across the four memory controllers) for the GAP benchmarks for a minimally sized LLC at 16MB. Figure 8 illustrates the MPKI (i.e., the number of M2P translations per kilo instruction requiring a page walk) as a function of log scale of MLB size. The figure shows that while the MLB size requirements across the benchmarks largely vary, there are approximately two windows of sizes that exhibit M2P translation working sets. The primary working set on average appears to be roughly around 64 aggregate MLB entries with many benchmarks exhibiting a step function in MPKI. The latter would provision only four MLB entries per thread indicating that the M2P translations are the results of spatial streams to 4KB page frames in memory. Beyond the first window, the second and final working set of M2P translations is around 128K MLB entries which is prohibitive, thereby suggesting that practical MLB designs would only require a few entries per memory controller.

Figure 9 illustrates the address translation overhead for cache hierarchies of upto 512MB while varying the number of aggregate MLB entries from 0 to 128 averaged over all the GAP benchmarks. Midgard in the figure refers to the baseline system without an MLB. The figure corroborates the results that on average 64 MLB entries is the proper sweet spot for 16MB caches. Comparing the figure with Figure 7, we see that for a 16MB LLC, Midgard can break even in address translation overhead with traditional 4KB-page systems with only 32 overall MLB entries (i.e., 8 entry per memory controller). In contrast, practical provisioning for MLB will not help Midgard break even with an ideal huge page system for a minimally sized LLC. The figure also shows that with only 32 and 64 MLB entries, Midgard can virtually eliminate address translation overhead in systems with 256MB and 128MB aggregate LLC respectively. Moreover, with 64 MLB entries, Midgard outperforms huge pages for LLC capacity equal or greater than 32MB. Finally, for LLC capacity of 512MB or larger, there is very little benefit from architectural support for M2P translation.

VII. Related Work

While there is a large body of work on virtual memory [9], we only list a few of them because of space constraints.

Virtual caches: Proposals for virtual caches date back to the '80s [16], with recent papers proposing virtual caches in the context of GPUs [68]. Ceckleov et al. [10], [11] summarize a variety of tradeoffs in the context of virtual caches. Single Address Space Operating System [32] eases virtual cache implementations but requires significant software modifications.

Intermediate address spaces: Wood et al. [65] propose a global virtual address space used to address the caches. However, translations from virtual to global virtual address space are done using fixed-size program segments, which now have been replaced with pages. Zhang et al. [69] propose an intermediate address space while requiring virtual addresses to be translated at a 256MB granularity. While this works well for GB-scale memory, it does not scale for TB-scale memory. Hajinazar et al. [18] propose an intermediate address space containing fixed-size virtual blocks, which are then used by the applications. While these virtual blocks are similar to the VMAs in Midgard, using them requires the significant application and toolchain modifications.

Huge pages and Ranges: There is a large body of work targeting adoption and implementation of huge pages [40], [48], [58]–[60] to reduce the address translation overheads. Midgard is compatible with most proposals on huge page integration and would benefit in M2P translation performance. Recent papers [4], [19], [28], [47], [67] have also introduced the notion of ranges, which are contiguous data regions in the virtual address space mapped to contiguous regions in the physical address space. These optimizations are compatible with Midgard M2P translation as well.

VIII. Conclusion

Despite decades of research on building complex TLB and MMU cache hardware as well as hardware/OS support for huge pages, address translation has remained a vexing performance problem for high-performance systems. As computer systems designers integrate cache hierarchies with higher capacity, the cost of address translation has continued to surge.

This paper proposed, realized, and evaluated a proof-of-concept design of Midgard, an intermediate namespace for all data in the coherence domain and cache hierarchy, in order to reduce address translation overheads and future-proof the VM abstraction. Midgard decouples address translation requirements into core-side access control at the granularity of VMAs, and memory-side translation at the granularity of pages for efficient capacity management. Midgard's decoupling enables lightweight core-side virtual to Midgard address translation, using leaner hardware support than traditional TLBs, and filtering of costlier Midgard to physical address translations to only situations where there are LLC misses. As process vendors increase LLC capacities to fit the primary, secondary, and tertiary working sets of modern workloads, Midgard to physical address translation becomes infrequent.

We used AMAT analysis to show that Midgard achieves only 5% higher address translation overhead as compared to traditional TLB hierarchies for 4KB pages when using a 16MB aggregate LLC. Midgard also breaks even with traditional TLB hierarchies for 2MB pages when using a 256MB aggregate LLC. For cache hierarchies with higher capacity, Midgard's address translation overhead drops to near zero as secondary and tertiary data working sets fit in the LLC, while traditional TLBs suffer even higher degrees of address translation overhead.

This paper is the first of several steps needed to demonstrate a fully working system with Midgard. We focused on a proof-of-concept software-modeled prototype of key architectural components. Future work will address the wide spectrum of topics needed to realize Midgard in real systems.

Acknowledgments

We thank Mark Silberstein, Mario Drumond, Arash Pourhabibi, Mark Sutherland, Ognjen Glamocanin, Yuanlong Li, Ahmet Yuzuguler, and Shanqing Lin for their feedback and support. This work was partially supported by FNS projects "Hardware/Software Co-Design for In-Memory Services" (200020B_188696) and "Memory-Centric Server Architecture for Datacenters" (200021_165749), and an IBM PhD Fellowship (612341).

References

[1] N. Amit, "Optimizing the TLB Shootdown Algorithm with Page Access Tracking." in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017, pp. 27–39.

[2] N. Amit, A. Tai, and M. Wei, "Don't shoot down TLB shootdowns!" in *Proceedings of the 2020 EuroSys Conference*, 2020, pp. 35:1–35:14.

[3] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)." in *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*, 2010, pp. 48–59.

[4] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers." in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 237–248.

[5] S. Beamer, K. Asanovic, and D. A. Patterson, "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server." in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 56–65.

[6] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite." *CoRR*, vol. abs/1508.03619, 2015.

[7] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems." in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*, 2008, pp. 26–35.

[8] A. Bhattacharjee, "Large-reach memory management unit caches." in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 383–394.

[9] A. Bhattacharjee and D. Lustig, *Architectural and Operating System Support for Virtual Memory*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[10] M. Cekleov and M. Dubois, "Virtual-address caches. Part 1: problems and solutions in uniprocessors." *IEEE Micro*, vol. 17, no. 5, pp. 64–71, 1997.

[11] M. Cekleov and M. Dubois, "Virtual-address caches.2. Multiprocessor issues." *IEEE Micro*, vol. 17, no. 6, pp. 69–74, 1997.

[12] Y. Chou, B. Fahs, and S. G. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism." in *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, 2004, pp. 76–89.

[13] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory." in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 133–146.

[14] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes." in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, 2017, pp. 435–448.

[15] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP=RC?" in *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, 1999, pp. 162–171.

[16] J. R. Goodman, "Coherency for Multiprocessor Virtual Address Caches." in *Proceedings of the 1st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-I)*, 1987, pp. 72–81.

[17] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks." in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 955–972.

[18] N. Hajinazar, P. Patel, M. Patel, K. Kanellopoulos, S. Ghose, R. Ausavarungnirun, G. F. Oliveira, J. Appavoo, V. Seshadri, and O. Mutlu, "The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 1050–1063.

[19] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing Memory in Heterogeneous Systems." in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018, pp. 637–650.

[20] P. Hornyack, L. Ceze, S. Gribble, D. Ports, and H. Levy, "A study of virtual memory usage and implications for large memory," *Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[21] IBM, "Guard Pages," https://patents.google.com/patent/US20080034179A1/en, 2020.

[22] Intel, "Intel 5-level Paging and 5-level EPT," https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, 2017.

[23] Jeff Barr, AWS, "EC2 High Memory Update: New 18 TB and 24 TB Instances," https://aws.amazon.com/blogs/aws/ec2-high-memory-update-new-18-tb-and-24-tb-instances/, 2019.

[24] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache." in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 25–37.

[25] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache." in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 404–415.

[26] Kabylake, "Intel Kabylake," https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake, 2020.

[27] A. Kannan, N. D. E. Jerger, and G. H. Loh, "Enabling interposer-based disintegration of multi-core processors." in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 546–558.

[28] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Redundant memory mappings for fast access to large memories." in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 66–78.

[29] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Energy-efficient address translation." in *Proceedings of the 22nd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2016, pp. 631–643.

[30] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. M. Swift, "Performance analysis of the memory management unit under scale-out workloads." in *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 1–12.

[31] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS." in *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, 2018, pp. 107–127.

[32] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architectural Support for Single Address Space Operating Systems." in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, 1992, pp. 175–186.

[33] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?" in *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, 2020, pp. 991–1010.

[34] M. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy Translation Coherence." in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018, pp. 651–664.

[35] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens." in *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, 2016, pp. 705–721.

[36] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying Memory Ordering at the Hardware-OS Interface." in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXI)*, 2016, pp. 233–247.

[37] S. Maass, M. Kumar, T. Kim, T. Krishna, and A. Bhattacharjee, "Ecotlb: Eventually consistent tlbs," *ACM Transactions on Architecture and Code Optimization*, 2020.

[38] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched Address Translation." in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 1023–1036.

[39] R. C. Murphy, K. B. Wheeler, and B. W. Barrett, "Introducing the graph 500," http://www.richardmurphy.net/archive/cug-may2010.pdf, 2010.

[40] J. Navarro, S. Iyer, P. Druschel, and A. L. Cox, "Practical, Transparent Operating System Support for Superpages." in *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[41] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA." in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014, pp. 3–18.

[42] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: sandboxing accelerators." in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 470–481.

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

[44] Parallel Systems Architecture Lab (PARSA) EPFL, "Qflex," https://qflex.epfl.ch, 2020.

[45] M. Parasar, A. Bhattacharjee, and T. Krishna, "SEESAW: Using Superpages to Improve VIPT Caches." in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 193–206.

[46] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated Page: Supporting Fragmented Memory Allocation for Large Pages." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 913–925.

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

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

[49] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces." in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014, pp. 743–758.

[50] J. Picorel, D. Jevdjic, and B. Falsafi, "Near-Memory Address Translation." in *Proceedings of the 26th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2017, pp. 303–317.

[51] J. Picorel, S. A. S. Kohroudi, Z. Yan, A. Bhattacharjee, B. Falsafi, and D. Jevdjic, "SPARTA: A divide and conquer approach to address translation for accelerators," *CoRR*, vol. abs/2001.07045, 2020. [Online]. Available: https://arxiv.org/abs/2001.07045

[52] D. Plakosh, "Guard Pages," https://us-cert.cisa.gov/bsi/articles/knowledge/coding-practices/guard-pages, 2020.

[53] M. K. Qureshi, "Memory Scaling is Dead, Long Live Memory Scaling," https://hps.ece.utexas.edu/yale75/qureshi_slides.pdf, 2014.

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

[55] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism." in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020, pp. 1093–1108.

[56] Snehanshu Shah, Google Cloud, "Announcing the general availability of 6 and 12 TB VMs for SAP HANA instances on Google Cloud Platform," https://cloud.google.com/blog/products/sap-google-cloud/announcing-the-general-availability-of-6-and-12tb-vms-for-sap-hana-instances-on-gcp, 2019.

[57] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product." *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

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

[59] M. Talluri and M. D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support." in *Proceedings of the*

6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), 1994, pp. 171–182.
[60] M. Talluri, S. I. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in Supporting Two Page Sizes." in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, 1992, pp. 415–424.
[61] S. Volos, D. Jevdjic, B. Falsafi, and B. Grot, "Fat Caches for Scale-Out Servers." *IEEE Micro*, vol. 37, no. 2, pp. 90–103, 2017.
[62] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors." in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 266–277.
[63] Wikichip, "AMD Zen2," https://en.wikichip.org/wiki/amd/microarchitectures/zen_2, 2020.
[64] Wikichip, "ARM Cortex A76," https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76, 2020.
[65] D. A. Wood, S. J. Eggers, G. A. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz, and D. A. Patterson, "An In-Cache Address Translation Mechanism." in *Proceedings of the 13th International Symposium on Computer Architecture (ISCA)*, 1986, pp. 358–365.
[66] D. A. Wood and R. H. Katz, "Supporting Reference and Dirty Bits in SPUR's Virtual Address Cache." in *Proceedings of the 16th International Symposium on Computer Architecture (ISCA)*, 1989, pp. 122–130.
[67] Z. Yan, D. Lustig, D. W. Nellans, and A. Bhattacharjee, "Translation ranger: operating system support for contiguity-aware TLBs." in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 698–710.
[68] H. Yoon, J. Lowe-Power, and G. S. Sohi, "Filtering Translation Bandwidth with Virtual Caching." in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018, pp. 113–127.
[69] L. Zhang, E. Speight, R. Rajamony, and J. Lin, "Enigma: architectural and operating system support for reducing the impact of address translation." in *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010, pp. 159–168.