Learning to Walk: Architecting Learned Virtual Memory Translation

Kaiyang Zhao
Carnegie Mellon University
Pittsburgh, USA
kaiyang2@cs.cmu.edu

Yuang Chen
Carnegie Mellon University
Pittsburgh, USA
hilbertc@cmu.edu

Xenia Xu
Carnegie Mellon University
Pittsburgh, USA
qiaoyinx@andrew.cmu.edu

Dan Schatzberg
Meta
New York, USA
dschatzberg@meta.com

Nastaran Hajinaza
Intel
Hillsboro, USA
nastaran.hajinazar@intel.com

Rupin Vakharwala
Intel
Hillsboro, USA
rupin.h.vakharwala@intel.com

Andy Anderson
Intel
Hillsboro, USA
andy.anderson@intel.com

Dimitrios Skarlatos
Carnegie Mellon University
Pittsburgh, USA
dskarlat@cs.cmu.edu

Abstract

The rise in memory demands of emerging datacenter applications has placed virtual memory translation in the spotlight, exposing it as a significant performance bottleneck. To address this problem, this paper introduces *Learned Virtual Memory (LVM)*, a page table structure that effectively provides optimal *single-access address translation*. LVM is founded on a novel learned index model that dynamically adapts the address translation procedure based on the characteristics of an application's virtual address space. Furthermore, LVM's learned index requires minimal memory space, does not impose stringent physical contiguity requirements, enjoys high cacheability in the MMU, efficiently supports insertions, and relies on simple fixed-point arithmetic. Finally, LVM supports all features of virtual memory, including multiple page sizes. We evaluate LVM with a set of operating system (OS) extensions in Linux, RTL synthesis, and full-system simulations across a wide range of workloads. LVM reduces the address translation overhead by an average of 44% over radix page tables, while reducing the page walk cache area required by 1.5×. Overall, LVM achieves a 2-27% speedup in application execution time and is within 1% of an ideal page table.

CCS Concepts

• **Software and its engineering** → **Operating systems**; **Virtual memory**; • **Computer systems organization** → **Architectures**; • **Computing methodologies** → *Machine learning*.

Keywords

Virtual memory, address translation, learned indexes

ACM Reference Format:
Kaiyang Zhao, Yuang Chen, Xenia Xu, Dan Schatzberg, Nastaran Hajinaza, Rupin Vakharwala, Andy Anderson, and Dimitrios Skarlatos. 2025. Learning to Walk: Architecting Learned Virtual Memory Translation. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3725843.3756093

1 Introduction

The increase in memory capacity in datacenters, coupled with the proliferation of memory-intensive applications, has made virtual memory translation a major performance bottleneck. This overhead is rooted in the status quo of virtual memory translation—*radix page tables*. By design, radix page tables incur substantial overhead during page walks, requiring up to five *sequential* memory accesses.

Despite their inherent drawback of having multiple indirections, the longevity of radix page tables is justified. Organizing page tables in a tree structure (Figure 1(a)) provides high coverage at upper levels (e.g., 512GB of coverage at the 2nd level), with coverage diminishing only closer to the leaf nodes that map actual pages, such as 2MB at the 4th level. This approach is amenable to caching upper levels of the tree in the MMU page walk caches because applications exhibit high locality at coarse address space granularities.

Unfortunately, the meteoric rise in memory capacity has begun to pierce the veil of cacheability of radix page tables. In response, over the past decade, processor architectures have spent the last decade in an arms race to increase TLB capacity and page walk cache sizes. Despite these efforts, studies at Google and Meta reveal that roughly 20% of cycles are spent on page walks [38, 95] and this issue is about to worsen due to several factors: (i) the inherent hardware limits of TLB scaling which have pushed latencies beyond that of L2 cache [91], (ii) the advent of terabyte-scale memory capacity through technologies like CXL [39, 76], and (iii) the increasing prevalence of memory-intensive applications.

Prior research has extensively studied methods to reduce address translation overhead [1–3, 8–10, 12–16, 18, 20, 27, 35, 36, 38, 41, 42, 45–47, 49, 59, 62, 64, 66, 67, 69, 74, 77–80, 82, 87, 88, 93–95].

One line of work has focused on virtual and physical memory contiguity to form larger translation blocks backing application datasets [5, 10, 27, 28, 47, 48, 66, 69, 94, 95]. The goal of these works is to map large, contiguous virtual memory regions to equally large contiguous physical memory regions and form large translations such as huge pages or even larger range translations. However, the limitation of these approaches is their reliance on abundant physical memory contiguity—a scarce resource in modern datacenters [95].

Another line of research has explored an alternative data structure to radix page tables based on hashed page tables [22, 37, 40, 43, 77, 80, 83, 93] (Figure 1(b)). Despite early shortcomings, recent work has shown that hashed page tables are viable [77, 79, 80, 93]. These designs replace sequential page walks with parallel accesses that aim to resolve hash collisions. However, state-of-the-art hashed page table designs [77, 79, 80] do not eliminate the core issue of multiple memory accesses for an address translation. Instead they trade sequential accesses for parallel ones, leading to increased memory traffic and cache pollution.

Recent work in databases has introduced learned indexes [21, 23, 50, 51, 54, 85, 90]. The core idea behind these structures is that the distribution of input keys can be learned, enabling models to replace hash functions and reduce collisions. Typically, learned indexes are organized as a hierarchy of models that incrementally capture finer-grained details of the key distribution. However, their effectiveness hinges on the assumption of a regular key distribution—a condition that is often challenging to satisfy in real-world database workloads [75].

In this paper, we explore the design of a learned index page table structure that enables *single-access translations*. However, building page tables with existing learned indexes is challenging as they are plagued by several limitations that make them ill-suited for virtual memory translation. At a high level, learned indexes have been designed for database software and fail to meet the requirements of address translation at the processor front-end. Specifically, existing learned indexes are hierarchical and can expand significantly in both width and depth, incurring more sequential indirections than radix page tables. To achieve high accuracy, these indexes often grow to tens of megabytes and rely on floating-point arithmetic, which makes them hard to cache and unsuitable for the page walk pipeline. Additionally, existing learned indexes require large, contiguous memory allocations—a poor fit for page tables, which must function within a fragmented physical address space.

Compounding these challenges, most learned index structures are designed for static datasets and assume one-time construction. Although recent efforts have explored mechanisms to support dynamic updates [21, 23, 54, 90], these approaches either incur significant retraining overhead or suffer from accuracy degradation, ultimately resulting in performance inferior to conventional hash tables. These limitations are misaligned with the requirements of address translation, which inherently involves dynamic virtual-to-physical mappings. Moreover, supporting multiple page sizes further complicates the design, as existing techniques often require maintaining separate learned indexes per page size, undermining the efficiency gains of single-access lookups.

In this paper, we holistically address these challenges through a novel learned index design tailored for virtual memory. We call our design *Learned Virtual Memory (LVM)*. At a high level, as shown in Figure 1(c), LVM replaces the hash functions of the hashed page table with a learned index. The learned index itself is organized as a hierarchy of models, each responsible for different parts of the virtual address space. Intuitively, the learned index learns the location of page table entries.

We start by validating our hypothesis that virtual address spaces of applications exhibit significant regularity through an in-depth study spanning a diverse range of applications, programming languages, and memory allocation behaviors, including graph analytics, key-value stores, bioinformatics, databases, datacenter workloads, HPC, and real-world production workloads at Meta. Our analysis reveals that the virtual address spaces of applications are highly structured, making them well-suited for learned indexes.

To resolve the depth and width challenge of learned indexes, we introduce a cost model tailored for virtual memory. The cost model balances prediction accuracy with the cacheability of the index, accounting for the expected latency of address translation based on depth, width, and collision rates. The cost model guides LVM to build a highly optimized learned index based on space-efficient linear models on a per-process basis. The learned index of LVM requires minimal memory to provide page table entry lookup coverage for the complete address space, so it enjoys high cacheability. Next, to support efficient updates, we leverage the tendency of applications to expand their virtual address space in a contiguous manner. Specifically, in LVM, we introduce new scaling techniques for the linear models that allow new translations to be added to the learned index while avoiding retraining or rebuilding. To resolve the physical contiguity requirements, we study Meta's production datacenters and identify that while physical memory contiguity in the range of hundreds of MBs does not exist [95], it remains abundant in the range of hundreds of KBs, even in highly fragmented servers. To this end, we design adaptive learned indexes that can dynamically allocate small leaf page tables based on the available physical contiguity. Finally, to support multiple page sizes, we develop a novel encoding scheme that leverages the linearity of our models. Different page sizes are represented via varying slopes within the same structure, enabling efficient multi-page-size support without separate indexing logic.

We evaluate LVM with OS extensions in Linux, RTL synthesis, and full-system simulations across a wide range of workloads. LVM reduces the address translation overhead by an average of 44% over radix page tables, while reducing the page walk cache area required by 1.5×. Overall, LVM achieves a 2–27% speedup in application execution time and is within 1% in terms of performance and memory traffic of an ideal page table that always performs single-access translations.

2 Background

In this section, we provide background on radix page tables, hashed page tables, and learned indexes.

2.1 Radix Page Tables

Multi-level radix page tables typically use four to five levels with 4KB leaf pages. Figure 1(a) shows a five-level radix page table. In the worst case, translation requires pointer-chasing through all levels, incurring five sequential memory accesses. Modern processors use

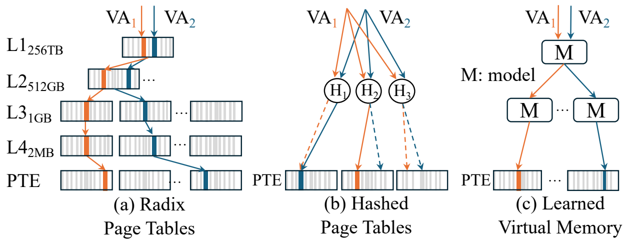

Figure 1: Address translations schemes. LVM avoids both the long sequential page walks of radix page tables and the high collision rates and parallel accesses of hashed page tables.

two techniques to reduce translation performance overhead: large pages (e.g., 2MB) for higher TLB hit rates and shorter walks, and page walk caches (PWC) to store upper-level entries. While upper levels in radix page tables offer high early coverage, PWC reach quickly diminishes—e.g., L4 tables cover only 2MB per entry. As prior work shows [45], even with more entries, MPKI remains high due to random access patterns in memory-intensive workloads.

2.2 Hashed Page Tables

Hashed page tables (HPTs) [22, 32, 37, 40, 43, 77, 80, 83, 93] are an alternative to radix page tables. In HPTs (Figure 1(b)), a hash function maps VPNs to page table entries, enabling single-access translation in the absence of collisions. In practice, however, HPTs face several challenges [8, 93]: hash collisions can degrade performance during resolution, early designs lacked dynamic resizing and support for multiple page sizes, and scattered entries led to poor cacheability. Although clustering improved cacheability, other issues remained [93].

Elastic cuckoo page tables (ECPTs) [77, 79, 80] improve on HPTs using a dynamically resizable d-ary cuckoo hash table [26, 61] with parallel accesses. However, they still require multiple memory accesses per translation, trading sequential for parallel lookups. As shown in Figure 1(b), a 3-way cuckoo table probes three candidate locations, incurring two unnecessary fetches per translation [77], increasing memory traffic and cache pollution. Supporting multiple page sizes adds further overhead, as ECPT must maintain separate tables per size. While ECPTs outperform radix trees via memory-level parallelism, they still stress the memory hierarchy.

2.3 Learned Indexes

Recent work in the database community has introduced learned indexes [51], which can outperform traditional structures like hash tables. Learned indexes replace the hash function with a hierarchy of models that approximate the cumulative distribution function (CDF) of keys, with each model responsible for part of the key space. They predict the position of the value in the table corresponding to the key, and on a misprediction, a bounded search is performed within the model's *min* and *max* error range, leveraging sorted keys [51]. Several approaches [50, 51] have laid the foundation for learned indexes. RMI [51] uses a hierarchical structure where each node holds a simple model, reducing collisions compared to traditional hash functions [75]. A core challenge is supporting inserts: new entries reduce model accuracy and may require slow full rebuilds. Recent work [21, 23, 54, 90] enables incremental updates,

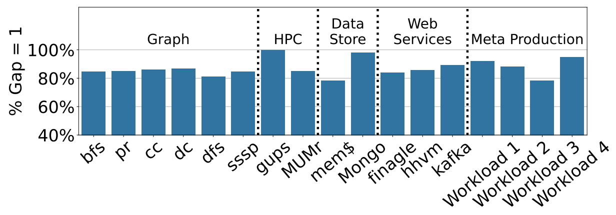

Figure 2: Virtual memory gap coverage of gap = 1.

but often reduces accuracy or adds complexity, increasing memory accesses during lookup.

Key Space Regularity. In practice, the primary requirement for a learned index to perform effectively is a regular key space. In this context, a regular key space is characterized by a key distribution that follows a predictable structure or pattern, e.g., sequential or evenly spaced. This regularity allows machine learning models to easily learn the distribution with high prediction accuracy.

3 Learned Indexes for Page Tables

In this section, we explore the feasibility of learned indexes for page tables.

3.1 Virtual Address Space Regularity

Since learned indexes rely on a regular key space to achieve high accuracy, we aim to determine if the application's virtual address space exhibits a consistent pattern between its keys, i.e., virtual page numbers (VPNs). To quantify regularity, we define a metric called the *virtual memory gap coverage*. Conceptually, a gap = 1 indicates that two virtual pages are sequential, meaning $\mathrm{VPN}_{\mathrm{next}} - \mathrm{VPN}_{\mathrm{current}} = 1$. The coverage of gap = 1 represents the proportion of all mappings in the virtual address space that are sequential. In other words, it measures how much of the virtual address space follows this specific pattern of contiguity, providing insight into the regularity of the virtual address space layout. For example, if 100% of the gaps are equal to 1, it means that all virtual pages are sequential. As a result, in that scenario, a single linear function can represent the whole virtual address space.

We use the *virtual memory gap coverage* to study the virtual address space of a broad set of representative memory-intensive applications, including graph analytics, bioinformatics, caching, HPC, database (MongoDB), web serving (Finagle RPC from Twitter [68] and hhvm from DCPerf [71]), streaming (Apache Kafka [17]) and four production workloads at Meta (Workload 1-4). They represent applications that are written in C, C++, PHP and Java, use file-backed mappings (MongoDB) and anonymous mappings. We see that across the workloads, a minimum of 78% of gaps are equal to 1, showing significant regularity in the distribution of virtual address space of applications. Importantly, both benchmarks and real-world production workloads show similar characteristics, validating our hypothesis that the virtual address space is highly regular. To quantify the effects of different userspace allocators, we further evaluated two broadly used allocators, jemalloc [44] and tcmalloc [86]. We find that across workloads, the regularity remains practically the same.

There are several reasons why application virtual address spaces exhibit regularity. First, userspace allocators aim to minimize holes

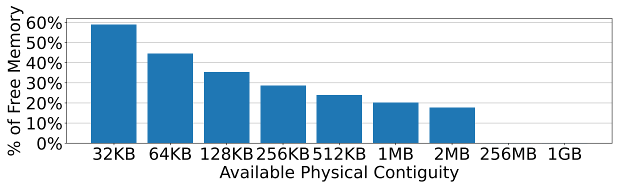

Figure 3: Median percentage of free memory in a Meta's datacenter that can be allocated contiguously at various sizes.

in the address space and pack allocations closely together [89]. Furthermore, allocators tend to buffer irregular allocation and free patterns of applications, hiding potential fragmentation from the OS that allocates pages for userspace [89]. Such buffering behavior is further present in virtualized environments where the guest OS handles the majority of memory allocation and free operations.

In addition, large object allocations, e.g., large arrays, need to be contiguous due to addressing requirements. Moreover, virtual address spaces are per-process and are not susceptible to fragmentation unlike the physical address space. Finally, the OS aims to reduce the space needed for page tables by coalescing mappings because lower-level page tables are only needed for the mapped virtual address space. Based on these observations, we believe that in the future, userspace allocators and the OS can make additional efforts to further bias the virtual address space towards regularity if needed, without modifications to other parts of the stack. As we show in later sections, LVM introduces cost model parameters that bound the depth and width of the learned index, further relaxing the regularity requirements of the virtual address space, even under pathological cases.

3.2 Virtual Memory Challenges

Having established that virtual memory is sufficiently regular to be a good candidate for learned indexes, we now discuss the requirements and challenges of an efficient virtual memory design.

Learned Index Structure and Size. Learned indexes are structured as hierarchical models that can expand in both width and depth. As a result, existing learned indexes often involve more sequential indirections than radix page tables. Additionally, they tend to be large, frequently reaching sizes in the tens of megabytes. One potential solution is to design an MMU caching structure large enough to store these models that is loaded during context switch. However, this approach is impractical due to the significant chip area required and the added context switch overhead. Instead, to effectively use learned indexes for address translation, we need models that are compact and easily cacheable within the MMU.

Insertions and Dynamic Virtual Address Spaces. Most learned indexes are typically constructed once over a static dataset. While recent research attempts to introduce support for updates, these methods often become prohibitively expensive due to retraining or a rapid degradation in accuracy. Address translation requires learned indexes to efficiently adapt to address space changes.

Multiple Page Sizes. Virtual memory supports different page sizes, such as 4KB, 2MB, and 1GB. The page size determines the page offset and hence the size of the VPN. Prior solutions in hashed page tables [77] handle this by maintaining separate tables for each page size and auxiliary data structures providing page size information, leading to additional memory requests. Maintaining separate learned indexes for every page size negates the goal of single-access address translation. Instead, efficient translation requires a single learned index that operates with variable page sizes without loss of accuracy or increase in training time.

Compute. Learned indexes rely on floating point operations. Implementing floating point support in the kernel, where page tables are maintained, would be challenging [24]. In hardware, while building a floating point page walk pipeline is possible, such an approach would (i) require a significant amount of resources, (ii) waste energy, and (iii) add unnecessary overhead to the page walk latency. Importantly, current MMUs support multiple outstanding page walks to satisfy the wide processor front-end; replicating such an expensive pipeline would significantly increase the area overhead. Instead, a practical page walk pipeline would require a learned index that relies on highly efficient integer computations.

Physical Memory Fragmentation. Physical memory fragmentation is a major challenge in datacenters. Recent work [95] showed obtaining large physically contiguous memory regions is infeasible in Meta's production environments. To investigate, we conducted a large-scale study across tens of thousands of servers in Meta's datacenters. Figure 3 shows the percentage of free memory immediately allocatable as a contiguous block. Our findings confirm [95]: contiguous regions of hundreds of megabytes are practically nonexistent. This is a key obstacle for existing learned indexes, which typically run in software and rely on large contiguous memory for the table storing values corresponding to keys. Notably, however, we find that small contiguous regions—on the order of a few hundred kilobytes—remain widely available, even in fragmented systems. Practical learned indexes must adapt to the granularity of available physical memory to allocate their leaf page tables.

4 Learned Virtual Memory Design

The goal of *Learned Virtual Memory (LVM)* is to build an optimal page table structure that performs single-access address translations. To this end, LVM introduces a novel learned index design tailored to the needs of virtual memory translation.

4.1 Overview

Conceptually, LVM aims to replace the fixed hash functions used in hashed page tables with a learned index. LVM organizes this index as a hierarchy of models, where each model is responsible for a subset of the virtual address space. Intuitively, the learned index predicts the location of page table entries (PTEs). Figure 4 provides a high-level overview of how LVM learns the structure of the virtual address space and maps it to page table entries.

Starting with Figure 4(a), each application has its own virtual address space, containing segments such as text, data, and heap. As in radix page tables, each mapped virtual page number (VPN) has a corresponding PTE, which specifies the physical page number (PPN). In radix page tables, the VPN-to-PTE mapping is performed through a fixed number of intermediate page tables that are indexed with a subset of the VPN bits. In hashed page tables, the mapping is identified by passing the VPN through a fixed hash function.

Intuitively, LVM replaces the fixed hash function with a learned function based on the underlying structure of the virtual address space. To showcase this point, we next visualize the mapping of

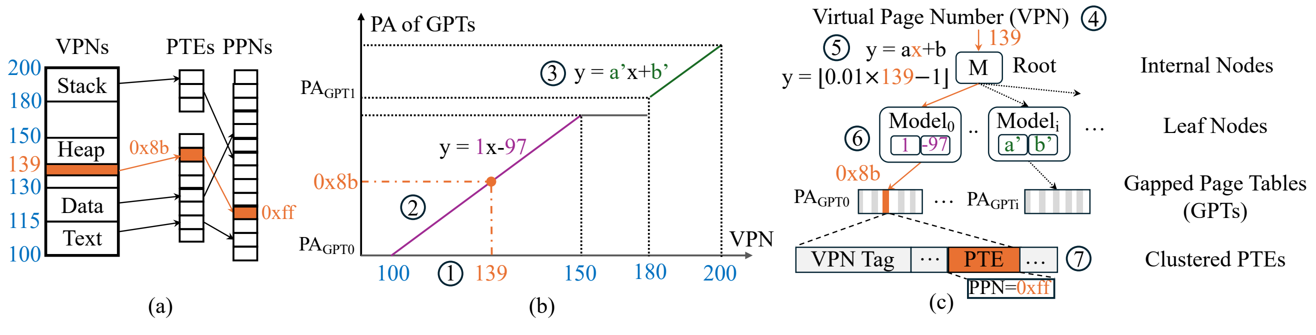

Figure 4: LVM learns the distribution of virtual addresses and maps them to page tables.

VPNs to PTEs in Figure 4(b). In this figure, the x-axis represents the sorted VPNs, while the y-axis shows the corresponding physical addresses of PTEs. The goal of LVM's learned index is to discover a set of functions that, given a specific VPN as input x, compute an output y that accurately predicts the physical address of its corresponding PTE. This concept is central to how LVM operates—it learns the distribution of VPNs and stores the PTEs at locations defined by the learned function.

LVM realizes this idea through a hierarchy of models, as shown in Figure 4(c), which partitions the virtual address space into smaller subsets that are described by models in *internal* and *leaf* nodes. Intuitively, internal nodes zoom in on a particular subset of the address space. They take as input the VPN and output the location of the child node responsible for an even smaller piece of the address space. Leaf nodes are responsible for the final mapping to page table entries and take as input the VPN and output the physical address of the PTE. Internally, the operation of both internal and leaf nodes is similar. Specifically, since the virtual address space is highly regular (Section 3.1), LVM leverages simple linear functions that effectively describe the distribution of the address space shown in Figure 4(b). LVM stores PTEs in gapped page tables (GPTs), which are conceptually small arrays of page table entries. Each GPT can be allocated at different physical address ranges, and hence, LVM does not impose strict physical contiguity requirements.

Under LVM, the OS maintains the learned index and the gapped page tables, and the hardware, on a TLB miss, performs lookups to locate the PTE. As we will discuss later, LVM aims to optimize the learned index's structure—including depth, width, and other parameters—to improve address translation efficiency.

An Address Translation Example with LVM. Next, we describe an end-to-end translation example with LVM. Consider a page in the application's heap with a $VPN = 139$, as shown in Figure 4(a). This VPN is mapped by a PTE that is located in the PA `0x8b`. The PTE maps that page to PPN `0xff`.

Figure 4(b) shows the VPNs of the application in the x-axis, and ① shows VPN 139. Based on the distribution of VPNs shown in the figure, LVM learns a representation of the mapped pages using the linear function $y = 1 \cdot x - 97$ as shown in ②. Specifically, to identify the location of the PTE, we set x as the VPN under translation, and the output y is the physical location of the gapped page table. Similarly, the pages in the application's stack are represented by another linear function (③ in Figure 4(b)).

Putting everything together, Figure 4(c) illustrates a page walk for $VPN = 139$. The walk begins at the root node (④), which spans the entire virtual address space of the application. The root contains a linear model $y = ax + b$ that uses the input $x = 139$ to select one of its child nodes, as shown in step ⑤. In this case, the root has a learned function $y = 0.01x - 1$ to select between its children nodes. The round-down result of this function with $x = 139$ directs us to $Model_0$, which is responsible for the VPN range $[100, 150)$. At step ⑥, the selected model computes $y = 1x - 97$ for $x = 139$, returning the physical address $0x8b$. This address corresponds to the location of the target page table entry (PTE) within the per-leaf-node gapped page table starting at base address PA_{GPT0}. Next, the relevant PTE cluster is fetched from memory (step ⑦), and the cluster's VPN tag is checked to validate the entry. Finally, the retrieved PTE contains the physical page number (PPN) = `0xff`.

4.2 A Learned Index for Virtual Memory

In this section, we describe the learned index structure, the underlying gapped page table, and the cost model of LVM.

4.2.1 The Learned Index Structure. LVM introduces a novel learned index design specialized to serve the needs of address translation. The purpose of the learned index is to identify the location of the PTEs. The learned index is organized as a hierarchy of models, each residing logically within a node. This is because a single model is ineffective at capturing the complete distribution of the keys. Figure 4(c) shows the overview of LVM's learned index structure. There are two types of nodes: internal nodes subdivide the key space into finer granularities to be handled by their children, and leaf nodes predict the position of the value associated with a key in the table. The children nodes evenly divide the address space of a parent node. The number of children per node is flexible and depends on the complexity of the key space. This approach enables a pliable index that can adapt to the needs of different applications. As we will discuss in Section 4.2.3, the training process is responsible for defining the depth and width of the hierarchy.

The learned model is the building block of the index. There is an inherent relationship between the complexity of the model and the address space it aims to represent. As we saw in Section 3.1, the virtual address space of applications is highly regular. Consequently, LVM opts for a linear function of the form $y = ax + b$. Hence, each node only needs to store the slope a and the intercept b. This design decision offers ease of computation (one multiplication and one addition) and provides sufficient accuracy when used in a hierarchy. Furthermore, it greatly enhances the explainability of the model.

Each model is responsible for approximating the cumulative distribution function (CDF) of its key space and mapping its key space to its output range, which is either the number of child nodes (for an internal node) or the size of the table (for a leaf node). Essentially, each model outputs $range \cdot CDF(x)$, where x is the key.

LVM uses this learned index design for address translation. Specifically, the key space is the virtual address space of the application. The input key is the virtual page number (VPN) under translation. The output of the leaf node is used to access the gapped page table holding the PTE, as shown at the bottom of Figure 4(c).

LVM stores all internal nodes at a particular depth consecutively in physical memory. Hence, each internal node can be uniquely identified with an offset, obviating the need to output the physical addresses of the internal child nodes and significantly reducing the size of each node. This layout does not impose any meaningful physical memory contiguity requirements as the learned index models of LVM are tiny, which we will see in Section 7.

Scalability. Because a single function can describe an arbitrarily large, virtually contiguous space, the learned index remains efficient even as memory usage grows. In contrast, radix and hashed page table designs require linearly larger MMU caches. For example, radix page tables need to populate additional entries in the upper levels of the radix tree as the memory footprint increases. This leads to cacheability issues since the hardware page walk caches (PWCs) must also expand—lest additional sequential memory accesses be needed to traverse the page tables. In contrast, LVM provides long-term scalability and cacheability in hardware irrespective of the memory footprint, effectively future-proofing its design. We empirically demonstrate this property of LVM in Section 7.

4.2.2 *Gapped Page Tables.*

Each leaf node in LVM is associated with its own page table. The model within the leaf node maps the VPNs to PTEs. To support insertions after the model is built, the page table is organized as a *gapped array* [21], resulting in *gapped page tables* (Figure 4(c)). Intuitively, the gapped page table contains additional empty slots, called gaps, that are unoccupied at build time. This design allows new keys to be inserted and further reduces collision rates when combined with LVM's insertion procedure, which we discuss in Section 4.3.2.

Relaxing Physical Contiguity Requirements. An important design decision for the gapped page tables is their size. This is because gapped page tables, similar to regular page tables in radix or hashed page tables, are allocated in the physical address space. As we showed in Figure 3, large blocks of physical contiguity are scarce in real-world datacenters due to fragmentation. Instead, LVM optimizes the design of the gapped page tables for small allocations that are ample even in highly fragmented environments and eliminates the need for a large physically contiguous page table. To this end, LVM maintains different gapped page tables for each leaf node as shown in Figure 4(c) with different physical addresses. To achieve this, during training of a leaf node, the physical address of the base of the gapped page table is added to the index of the PTE in the table to yield the final output that the leaf model needs to learn. In the example in Figure 4, this means adding PA_{GPT0} to the index of the PTE in the GPT, using it as the output the leaf model learns. Therefore, leaf nodes in LVM output directly the physical addresses of PTEs in its private table.

Moreover, this per-leaf-node gapped page table design enables LVM to dynamically adapt to the physical memory fragmentation status of the system when a leaf node is created. If there is insufficient physical contiguity for a large table, additional leaf nodes—and thus separate gapped page tables—can be created to utilize smaller physical contiguity as low as the base page size, e.g., 4KB in x86 or 64KB for some ARM systems. In the evaluation, we will demonstrate the adaptability of LVM and show that this approach performs well even in highly fragmented environments.

4.2.3 *Cost Model.*

The shape of the learned index is critical for performance and space efficiency. There are three key parameters that influence the lookup performance of learned indexes in the context of virtual memory: (i) the depth of the index, which defines the number of indirections required to traverse the model, (ii) the branching factor of the nodes, which influences the width of the model and affects the translation coverage of each node, and (iii) the collision rate, which determines the number of accesses required to locate an entry in the underlying table. Generally, increasing the depth or branching factor allows individual models to handle smaller segments of the virtual address space, which decreases collision rates and enhances lookup performance. However, as the index becomes wider and deeper, traversal times increase, and the model's cacheability deteriorates due to its larger size.

To address this trade-off, LVM introduces a principled cost model to balance these parameters effectively. The cost model employs a weighted function associated with a specific number of children (n) for each node under consideration. This function incorporates the index depth (d), the size of the index in bytes (s), the collision rate (cr), and the average number of additional memory accesses per collision (ma). By optimizing this function, LVM finds an equilibrium that minimizes overall lookup latency while maintaining efficient space utilization. Specifically, we define the translation cost $C(n)$ of a node with n children as:

$$C(n) = x_1 \cdot d + x_2 \cdot s + x_3 \cdot cr \cdot ma \qquad (1)$$

We discuss the tunable weights x_i in Section 5. During execution, the cost model first computes the number of *spline points* for the keys within a particular node [50]. Spline points partition the input key space into segments where simple linear models can accurately approximate the key distribution. Essentially, the number of spline points reflects the complexity of the key distribution within the node—more spline points indicate greater variability. This number provides a reasonable estimate for the optimal number of child nodes needed to model the key space effectively. To reduce computational overhead, the cost model limits its evaluation to ±2 around the estimated number of spline points, allowing the cost model to quickly assess a few plausible numbers of child nodes, minimizing the effort required to find the most cost-effective option.

Although the cost model inherently penalizes increasing the index depth without significant accuracy gains, LVM further enforces a hard limit on the depth using a parameter `d_limit`, which bounds the number of indirections during a lookup, ensuring efficient traversal. Additionally, LVM imposes a constraint on the worst-case coverage per byte, aiming to match or improve upon the locality provided by a radix page table at the same level. Specifically, if a node does not offer sufficient coverage relative to its size—that is, it doesn't map enough of the address space per byte of index—the cost model prevents the creation of additional child nodes beneath it. This strategy emphasizes the cacheability of the index by ensuring that each node is space-efficient. By bounding the depth and coverage of the learned index, these two constraints also protect against pathological cases when the regularity of the virtual address space is not maintained. Even in a pathological case, LVM's learned index would not grow too deep for hardware page walks nor too large to have good cacheability.

LVM's cost model is carefully tailored to the needs of virtual memory systems. Conceptually, the goal is to create a more efficient hierarchy than the upper layers of a radix tree. In practice, while a radix tree has a rigid structure, LVM's cost model enables a flexible hierarchy that can adjust its depth and width to increase precision for specific regions of the address space. The model dynamically adds depth and width by closely following the regularity of the address space, incorporating additional nodes only where they are necessary. As we will demonstrate in Section 7, this cost model leads to indexes that are minimal in size yet highly efficient.

4.3 Learned Index Operations

In this section, we discuss the main operations of LVM.

4.3.1 Initialization. The learned index is initialized as follows. Given a set of keys, LVM creates a root node and begins the training process. This training invokes a cost model that greedily determines the optimal number of child nodes at each level. When a new branch is created, the key space is evenly divided into k segments, assigning each child node responsibility for $1/k$ of the key space. This recursive process continues for each child node until the cost model decides that a leaf node should not have any further subdivisions. The same procedure is applied when rebuilding the model. In the context of virtual memory, the operating system initializes the index upon mapping the first page of a process.

4.3.2 Training of the Learned Index. The goal of training the linear model in a node is to approximate the cumulative distribution function (CDF) of the keys as closely as possible with minimal computational cost. Internal nodes and leaf nodes are trained using two different methods. For internal nodes of the index, the training process first determines the number of children the node should have based on LVM's cost model. Then, the training continues by allocating memory space for the child nodes. Next, to derive the model of the internal node, LVM learns a linear function between VPNs and indexes of children nodes to evenly divide the parent node's key space. Because the relationship between inputs and outputs can be perfectly represented by a linear function, heavyweight algorithms are unnecessary.

Leaf nodes are trained differently through a specialized process. First, LVM queries the OS allocator for physical contiguity (e.g., the next available allocation order in the buddy allocator in Linux). Then based on the available contiguity, LVM adapts the number of sibling leaf nodes necessary such that a leaf node covers as many translations as allowed by physical contiguity. A linear model for a leaf node is then learned using linear regression over the (VPN, position) pairs, where position denotes the index of the VPN in the sorted list of all VPNs within that node. This model is then scaled by the gapped array scale factor (`ga_scale`) to disperse the existing VPNs across a gapped array, leaving empty slots available for future insertions. After scaling, a table for the node is allocated in memory with size determined by multiplying the total number of VPNs at the node by `ga_scale`, ensuring sufficient space for both current and future VPNs. The page table entries corresponding to the VPNs are then inserted into the gapped page table array at the positions predicted by the scaled model. If a predicted location is already occupied, an exponential search is conducted to find the next available slot for insertion.

4.3.3 Bounding Mispredictions. As learned indexes are based on machine learning models, they can incur rare mispredictions during a lookup. If the predicted location does not result in a hit, then a binary search is performed within the error bound that is enforced during the training of the index [51]. For LVM, this is done by setting the error bound during linear regression such that an upper bound C_{err} of the number of additional memory accesses is enforced to ensure efficient address translation even in the worst case. This approach sacrifices some space efficiency for tighter error bounds by creating more child nodes during training. More specifically, if, at a particular leaf node, the error bound cannot be satisfied, LVM goes back to its parent node and re-evaluates the cost model with a sufficiently boosted weight x_3 for the computational cost of resolving a collision that will satisfy the error bound in all its children nodes. Training then returns to the parent node, and more child nodes are created to subdivide the key space at a finer granularity, which simplifies the key space each child node is responsible for and reduces error.

4.3.4 Supporting Efficient Insertion. Insertions are divided into two categories based on the new key's relation to the existing key range: (i) within-bounds inserts and (ii) out-of-bounds inserts. A within-bounds insert occurs when the new key falls within the existing range of keys, while an out-of-bounds insert happens when the new key lies outside the current key range.

Out of bounds inserts close to the edge. LVM optimizes for the common case of out of bound inserts that happen close to the existing edge. This is because applications tend to exhibit locality in page allocations, so newly added pages tend to expand the virtual address space in a contiguous manner. LVM introduces two new novel techniques for learned indexes to avoid costly retraining.

The first technique defines the *minimum insertion distance* which allows insertions to be processed in batches. Specifically, LVM always expands the address space by at least the minimum insertion distance. The benefits are two-fold. First, this approach absorbs some non-contiguous insertions that are temporary in nature. Second, it allows LVM to amortize the fixed computational cost of insertions over a larger range of keys, further reducing the software management cost. Note that LVM's approach of allocating a minimum distance only creates page tables ahead of time, and the actual physical page allocation only happens later on demand. The second technique relies on rescaling. Specifically, LVM rescales the leaf node by expanding the gapped page table associated with the node without modifying the model and inserting the new keys.

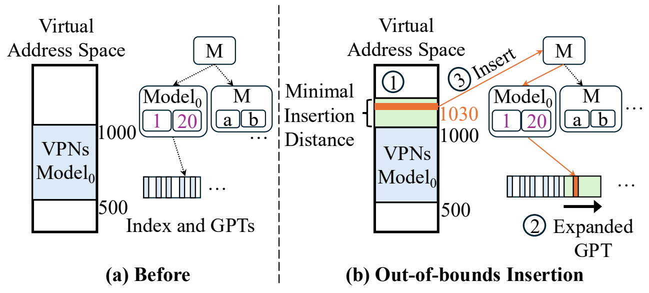

Figure 5: LVM out of bounds inserts close to the edge.

For example, as shown in Figure 5 (a), suppose a leaf node (Model$_0$) is responsible for keys 500 – 1000 and has its gapped page table. During an insertion of a new VPN, 1030, first, LVM records that the key range of this node is expanded to 500 – 1050, taking into account a hypothetical minimum insertion distance of 50 (Step ① in Figure 5 (b)). Then, the gapped page table is expanded by 65 slots (Step ②), taking into account `ga_scale` = 1.3. Finally, the new key for VPN 1030, along with other keys from 1001 – 1050 are inserted into the expanded part of the gapped page table at locations predicted by the unchanged model (Step ③). Note that this avoids retraining the model or re-inserting the existing keys of the node: for existing keys for VPN 500 – 1000, since neither the model nor the location of their PTEs changed, future page walks still yield the correct PTEs; similarly, the expanded keys follow the same model. Overall, these techniques, when combined with the underlying design of gapped page tables, resolve the adaptability limitation of prior learned indexes, making them suitable for virtual memory.

Within bounds inserts and out of bounds inserts away from the edge. If the key to be inserted falls within the current bounds of the key range, LVM proceeds as follows. LVM queries the model and attempts to insert the key into the location indicated by the model. Due to LVM's gapped array design, the table usually has empty slots for insertion. In the common case, the location is empty, and the new key is inserted successfully. Otherwise, in the unlikely scenario that the slot is occupied, LVM proceeds with retraining only the leaf node. In the vast majority of cases, the local retraining is successful, and the new key finds its place within the table.

Finally, if the key is still not inserted, there are two possible approaches we can take. These approaches are also the same for out of bounds inserts away from the edge. One approach is to only rebuild the parent node and try again. This process can continue recursively until the key is inserted. The other approach is to simply rebuild the index. In practice, as we will see in Section 7, the computational cost of rebuilding the model is quite small. Hence, due to the rarity of these events, LVM opts for a full rebuild.

4.4 Supporting Multiple Page Sizes

LVM efficiently supports multiple page sizes without requiring separate learned index structures, thereby overcoming the limitations of hashed page table designs. It leverages the linearity of the learned index to seamlessly accommodate multiple page sizes within a single index. This is achieved by representing different page sizes as lines with varying slopes in the cumulative distribution function

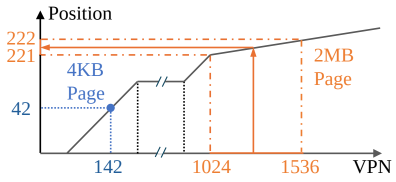

Figure 6: Regular and huge pages as represented by LVM.

(CDF) of the keys. Specifically, smaller pages correspond to steeper slopes, while larger pages correspond to lower slopes. Intuitively, larger page sizes result in fewer PTEs within a given VPN range, leading to lower slopes in the learned index.

Figure 6 shows an example virtual address space distribution with both a 4KB page and a 2MB page. The 4KB page at VPN 142 has only one VPN in its range of virtual addresses, which gets mapped to position 42 in the gapped page tables by the learned index. On the other hand, the 2MB page spans VPNs [1024, 1536). To support huge pages, LVM trains the learned index based on the VPN of the first 4KB sub-page i.e., VPN 1024. During a lookup, the 4KB VPN is derived from the VA to traverse the index. Note that if a huge page is mapped, 4KB sub-pages inside it cannot also be mapped. Therefore, lookups using any addresses inside a huge page will identify a unique translation entry. In the example of Figure 6, any queries using VPNs between 1024 and 1536 will be rounded down to the PTE at position 221, yielding the 2MB translation entry.

To identify the page size, the translation entry in the table uses two bits to encode the size (supporting the 4KB, 2MB, and 1GB pages in x86). Importantly, LVM can support *any* number of page sizes without modifications or additional structures. This is in contrast to existing radix page tables that rely on rigid page sizes and hashed page tables [77] that require separate structures and potentially additional accesses for each page size. We envision that LVM will enable a new direction where applications can leverage arbitrary page sizes without hardware changes, unlocking high performance. We leave this exploration for future work.

4.5 Fixed-Point Arithmetic for Fast Lookups

State-of-the-art learned indexes rely on floating-point operations, which violates the requirements discussed in Section 3.2. To ease the implementation of LVM, we introduce an effective quantization strategy for learned indexes that represents model parameters in fixed-point values. Specifically, each value in the model consists of a 44-bit integer part and a 20-bit fractional part. In this way, each value of a model requires 8 bytes, and each node takes 16 bytes.

4.6 OS and Hardware Support

At a high level, LVM maintains the same abstraction and mechanisms of virtual memory as in radix and hashed page tables.

4.6.1 Operating System. The OS components of LVM build and manage per-process learned indexes. They include the cost model along with logic in the page fault handler and generic page walk logic for the OS to access and modify translations. In terms of operations, the OS is responsible for initiating, training, and manipulating the learned index and the translations, while the hardware performs lookups during address translation. We further discuss our OS prototype of LVM in Section 5.

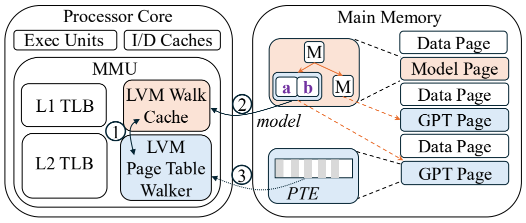

Figure 7: LVM hardware overview.

4.6.2 Hardware. The changes of LVM are isolated in the hardware memory management unit (MMU) as depicted in Figure 7. Only the two shaded components in the MMU are modified compared to existing architectures, while other components, such as the L1/L2 TLBs, remain unchanged. The page table walker and the page walk cache (PWC) of the radix page table are replaced by the *LVM Page Table Walker* and the *LVM Walk Cache (LWC)*. Each LVM walker contains an adder and a multiplier that are used to compute the output of the model. Upon an L2 TLB miss, the LVM page table walker starts to translate a virtual address to a physical address. It performs the LVM page walk by traversing through a hierarchy of learned models, which are cached by the *LVM Walk Cache (LWC)*. If a requested model is present in LWC, it is supplied to the page table walker (Step ①); otherwise, it is fetched from main memory on demand (Step ②). After the page walk reaches the leaf node in the index, the physical address of the page table entry (PTE) is computed, and the walker fetches it from the memory hierarchy (Step ③). Similarly to radix, the PTEs can be cached by the data caches of the processor, but are not cached in the LWC.

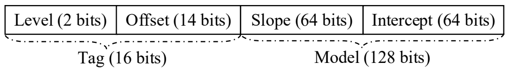

Figure 8: LVM Page Walk Cache Entry.

The LWC is fully associative and stores the slope and intercept of individual models of the learned index. Figure 8 shows an LWC entry in details. This design allows LVM to only cache nodes on demand instead of fetching the whole learned index. The page table walker fetches 64-byte cache lines, each containing four individual models. Each model takes 16 bytes of space (slope and intercept). The page walker stores the learned index node in an LWC entry and attaches the level and the offset of the node. Furthermore, each entry in the LWC stores the address space identifier (ASID). This approach allows LVM to handle context switches without flushes, similar to PWCs in radix page tables. As we will see in Section 7, LVM achieves a 3× area reduction over radix page walk caches while being more scalable.

The OS exposes information required for the hardware page walk using `d_limit` control registers (Section 4). LVM adds these registers that store the base physical address of the internal nodes at each level in the index. This is similar to how the x86-64 architecture stores the base address of the page table in the CR3 register.

Virtualization Support. LVM can support virtualization through nested page tables, similarly to radix page tables. In this environment, the hypervisor maintains LVM page tables for each VM to translate guest physical addresses (GPA) to host physical addresses (HPA). Similarly, each guest OS maintains LVM page tables to translate guest virtual addresses (GVA) to guest physical addresses (GPA). Due to the increased performance cost of nested radix page tables, we expect LVM to provide even higher performance gains.

5 Implementation

5.1 Parameters Used

LVM uses a set of tunable parameters. We empirically set the values of the tunable weights in the cost model to be $x_1 = 10, x_2 = 5, x_3 = 200$. We define the depth limit of the index `d_limit` = 3 to mirror the maximum number of indirections in a traditional radix page table (LVM has another fetch for the translation entry, bringing the max possible memory accesses to 4). We set `ga_scale` = 1.3 to effectively leave space for future inserts in the table. Finally, we set the *minimum insertion distance* to be 64MB, enough to cover most inserts in applications and not unduly expand the size of the table.

5.2 LVM Operations in the OS

Software lookup. When the translation entry of an already mapped page needs to be changed or looked up (e.g., a permission change or checking the *accessed* or *dirty* bits), the OS performs the same page walk operation as the hardware does to obtain the page table entry. These operations do not modify the index.

Free. When a page is freed to the OS, LVM follows the same steps as radix page tables by clearing the PTE in page tables, flushing the TLB, and releasing the physical page. However, LVM opts to not change the index and keeps the empty space for the PTE in the table because, first, the OS prefers to keep the virtual address space contiguous and plugs in holes in the virtual address space with later allocations, so the space for the entry will likely be reused in the future by new allocations. Second, memory usage of applications typically remains relatively stable near their peak usage after their initialization phase. For workloads that exhibit a peak memory usage that is significantly higher than their steady-state memory usage, the OS can rebuild the index and reclaim unused space.

Insert. LVM maps new pages by inserting into the learned page table as described in detail in Section 4. Physical frame allocations are deferred until the first access to a page, as in current OSs.

Kernel Mappings. The Linux kernel has a shared kernel address space mapped to the virtual address space of all processes. LVM keeps one learned page table for the kernel address space, reducing memory consumption and also avoiding repetitively training a different kernel space learned page table for each new process.

ASLR. Address space layout randomization (ASLR) partitions the virtual address space into distinct parts that are far apart. The OS exposes the ASLR base addresses to hardware through registers, removing ASLR effects during LVM training and management. This approach is transparent to applications, maintains ASLR's security, and simplifies the learned index training process of LVM.

TLB Shootdowns. LVM maintains the same TLB shootdown behavior as radix page tables. Specifically, TLB shootdowns take place only when PTEs are modified, e.g., during OS swapping or page permission changes. During any LVM-specific operation, such as retraining, insertions, or rescaling, PTEs are not modified and hence a TLB shootdown is not required.

LWC Flushes. On regular insertions or rescaling, which are the most frequent operations, the LVM learned index is not modified and hence the LWC does not need to be flushed. On the very rare occasion when an LVM model node is retrained, it causes the cached entry in the LWC to become stale. In this case, the OS simply flushes the entry from the LWC. As we will see in the evaluation (Section 7.3), retraining operations, and thus LWC flushes, are very rare, with an average of two and at most three occurring during the entire application runtime, taking place at the program initialization.

Multi-threading LVM naturally supports multi-threaded applications. At a high level, LVM leverages similar locking mechanisms as in radix page tables. For updates to the PTEs, the table in which the PTE resides is locked by the OS so that only one thread can perform the modification at any given time. This locking requirement is consistent with PTEs being modified in radix page tables, when the OS acquires a lock associated with the leaf page table. For retraining, LVM acquires a lock on the parent node during retraining, preventing concurrent modifications. This is similar to when radix page tables allocate or free page tables.

5.3 Linux Prototype

We evaluate LVM with a Linux prototype. Specifically, we modify Linux kernel 5.15 to stream operations that map and unmap pages to a userspace agent that implements LVM. As the hardware features of LVM are simulated, we choose to implement LVM in userspace through an indirection layer for ease of development. Our prototype maintains LVM for a particular process, including all the necessary virtual memory operations, including building and training the learned index, as well as managing the translation tables, including allocation and free operations. During simulation, we can configure our prototype to rely solely on LVM for virtual memory operations, providing a realistic evaluation of its performance. The prototype is composed of 4200 lines of C and C++.

6 Evaluation Methodology

6.1 Architectural Simulation

We use SST [72], a detailed cycle-level simulation backend, integrated with a QEMU [11] frontend that runs the Linux kernel and userspace processes for full-system simulation. SST also incorporates DRAMSim3 [73] for main memory modeling. We model all the hardware components of LVM, including page walkers, LWCs, and flush operations. The architectural parameters are shown in Table 1. The simulation includes all the computational costs of managing LVM, e.g., training, insertions, and rescaling by relying on our Linux prototype described in Section 5.3.

6.2 Workloads

We consider a wide range of workloads in our evaluation. They include six workloads from the graphBIG [60] graph benchmark suite: Breadth First Search (BFS), Depth First Search (DFS), Connected Components (CC), Degree Centrality (DC), Page Rank (PR), and Shortest Path (SSSP). They take a Kronecker graph that produces a runtime memory footprint of 75GB. GUPS is a random access benchmark from HPC Challenge [55]. MUMmer (MUMr) is a DNA sequence alignment program from the BioBench2 [4] bioinformatics suite and has a memory footprint of 20GB. Memcached [25] (mem$) is a widely used in-memory key-value store. It has a memory footprint of 124GB. During simulation, we execute 1 billion instructions in the region of interest of the benchmarks.

Full-System Simulation Parameters	
Core	4-issue out-of-order cores at 2GHz
L1-I and L1-D cache	32KB each, 8-way, 1 cycle RT
L2 cache	1MB, 8-way, 20 cycles RT
L3 cache	2MB per core, 16-way, 56 cycles RT
L1 DTLB/ITLB (4KB pages)	64 entries, 4-way
L1 DTLB/ITLB (2MB pages)	32 entries, 4-way
L2 TLB (4KB pages)	2048 entries, 12-way
L2 TLB (2MB pages)	2048 entries, 12-way
Radix Page Walk Cache	3 levels, 32 entries per level, 2 cycles
LVM Page Walk Cache	16 entries, 2 cycles
Cuckoo Walk Cache	PMD: 16 entries. PUD: 2 entries. 2 cycles
Cuckoo Page Tables	3 ways. 16384 entry initial size.
Main Memory	128GB, DDR4 3200MT/s, 8 banks, 4 channels
OS	Linux Kernel 5.15

Table 1: Architectural Parameters.

6.3 Configurations and Baselines

We compare **LVM** with four-level radix page tables and elastic cuckoo page tables (**ECPT**) [77], a state-of-the-art hashed page table. We model both designs in detail, including a page walk cache (PWC) for radix page tables, and for ECPT, we include the Cuckoo Walk Cache (CWC) and Cuckoo Walk Tables (CWT) that trim the required parallel accesses for huge pages. The configuration parameters are shown in Table 1. Furthermore, to get a better understanding of the upper bound performance benefits, we compare LVM to an **Ideal** page table scheme that always requires only a single memory access to locate the page table entry to complete a page walk. For every configuration, we consider two page sizes: (i) **4KB**, and (ii) 2MB using Linux's Transparent Huge Pages (**THP**).

7 Evaluation

In the evaluation, we first present the end-to-end speedups of workloads in Section 7.1. We also compare the performance of LVM to three related work, Midgard [34], ASAP [59] and FPT [64]. Next, to understand the sources of the speedups, we investigate in Section 7.2 the architectural characteristics of LVM regarding MMU, caches and the memory hierarchy. Then, we characterize the learned page table data structures of LVM in Section 7.3 in terms of size, collision rates, memory fragmentation, memory consumption, etc. Finally, we evaluate the hardware properties of LVM in Section 7.4 using our implementation and synthesis in RTL and a commercial PDK and compare its area and power to those of Radix Page Tables.

7.1 End-to-end Speedups.

In Figure 9, we show the end-to-end performance results. We report speedups relative to the 4KB radix page tables. We see that LVM provides major performance gains. When using 4KB pages, LVM achieves a 5%–26% speedup over radix page tables, with an average speedup of 14%. For THP, LVM achieves a 2%–27% speedup over radix page tables, with the average speedup being 7%. When compared to ECPT with 4KB, the state-of-the-art hashed page table design, LVM achieves up to a 7% speedup, with an average speedup of 5%. These are significant performance gains. Similarly, LVM outperforms ECPT by 3% when using huge pages. As we will see next, ECPT achieves good performance by trading latency for increased

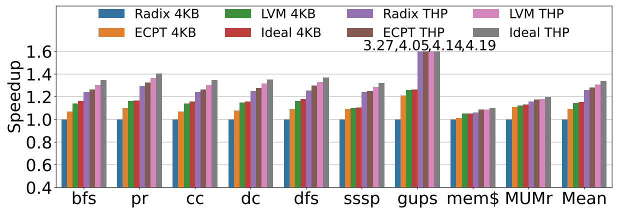

Figure 9: End-to-end speedups.

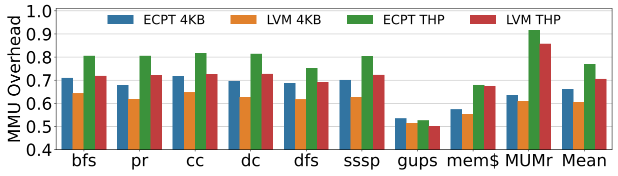

Figure 10: MMU overhead relative to radix. Results are normalized separately to radix 4KB and THP.

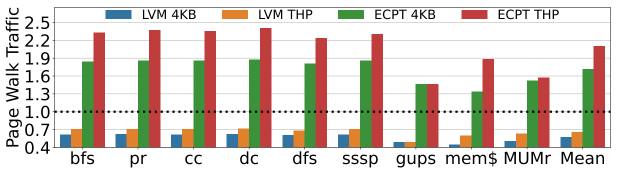

Figure 11: Page walk traffic relative to radix. Results are normalized separately to Radix 4KB and THP.

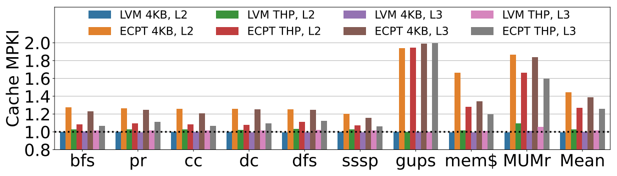

Figure 12: Cache MPKI relative to radix page tables.

memory traffic due to parallel accesses. LVM outperforms ECPT and drastically reduces memory traffic for address translation.

Next, to get a better understanding of the upper bound performance of LVM, we compare it to an ideal design that always requires only a single memory access for locating the page table entry. LVM achieves performance within 1% of an ideal page table design on average. Overall, LVM provides significant speedups over radix page tables and state-of-the-art hashed page table designs, providing near-ideal address translation performance.

Multi-tenancy. To evaluate the efficacy of LVM in multi-tenant environments with stacked workloads, we performed additional simulations with combinations of our workloads on an 8-core setup, and every workload runs on its own core. The speedups of LVM remain the same (within 0.5% difference) across configurations.

Multi-threaded Applications. LVM supports multi-threaded applications as described in Section 5.2. We simulated the graph workloads with eight threads. LVM maintains its performance gains and the results remain similar (within 1% difference) across configurations. This is because LVM efficiently uses fine-grained locking for page table modifications and introduces mechanisms (rescaling and minimum insertion distance) that make retrains exceedingly rare (Section 7.3).

7.2 Architectural Characterizations.

MMU Overhead. To understand the performance results, we look at the MMU overhead relative to radix page tables shown in Figure 10. We measure the total cycles that memory requests spend in the MMU, including in the TLB and page table walker. Note that for the 4KB configurations, we normalize the results to Radix with 4KB pages and the THP configurations to Radix with THP. As we can see from the results, LVM reduces the MMU overhead by an average of 39% compared to Radix for 4KB pages and by 29% when using THP. Furthermore, LVM outperforms ECPT by 3%–10% and 8% on average for 4KB pages and by 1%–11% and 8% on average under THP. In terms of page walk cycles, LVM achieves a 52% reduction when using 4KB pages and 44% reduction when using THP on average compared to radix. In comparison, ECPT achieves only a 25% reduction in page walk cycles when using 4KB pages and 20% reduction when using THP. In other words, *LVM outperforms ECPT in terms of page walk cycles by 2×*. Overall, LVM achieves significant MMU overhead reduction over Radix and ECPT.

Page Walk Traffic. Next, we examine the MMU overhead reduction achieved by LVM through an analysis of page walk traffic. Figure 11 presents the number of memory requests sent to the cache hierarchy due to page walks. Results for 4KB configurations are normalized to Radix with 4KB pages, and THP configurations are normalized to Radix with THP. LVM drastically reduces memory traffic associated with page walks. Compared to Radix with 4KB pages, LVM reduces page walk traffic by an average of 43%. For THP, the reduction is similarly significant at 34%. These reductions represent substantial improvements in MMU efficiency.

When comparing LVM to the state-of-the-art hashed page table, ECPT, the benefits are even more pronounced. ECPT, on average, generates 1.7× and 2.1× more memory accesses for page walks compared to Radix with 4KB and THP configurations, respectively. This is due to ECPT's design, which employs multiple parallel requests, trading off reduced latency for significantly increased bandwidth usage. In contrast, LVM not only surpasses ECPT in terms of end-to-end performance but also achieves a substantial reduction in MMU traffic, with 2.9× and 3.1× fewer memory accesses compared to ECPT 4KB and THP configurations, respectively.

Finally, while not depicted in Figure 11, LVM requires only 1% additional page walk-induced memory traffic compared to the ideal page table design, underscoring its efficiency.

Cache Performance. To quantify the impact of page walk traffic, we examine the misses per kilo instructions (MPKI) for L2 and L3 caches shown in Figure 12. The results demonstrate that LVM achieves high cache efficiency. Specifically, the L2 and L3 MPKI of LVM is, on average, within 1% of that of radix. In contrast, ECPT exhibits significantly higher MPKI, with increases of 44% in L2 and 40% in L3. This behavior stems from ECPT's reliance on parallel accesses to locate a translation, resulting in additional page table entries being fetched into the caches. This effect is particularly pronounced in workloads with large page table entry working sets, such as GUPS, memcached, and MUMmer, where ECPT leads to on average 72% and 59% higher MPKI in L2 and L3, respectively. This is expected: as workload memory footprints grow, ECPT's larger PTE working set exceeds the capacity of the caches, causing cache pollution and contention between data and PTEs. In contrast, LVM efficiently locates translations with a single page table entry lookup in the common case, thereby ensuring low translation latency while maintaining high cache efficiency.

TLB, PWC and LWC Miss Rates. In the simulation, the L2 TLB miss rates (57.5%–99.4%) remain nearly identical across different configurations (Radix, ECPT, and LVM) since they solely affect the page walk process. Radix PWC suffers medium to high miss rates (59.7%–99.6%) at the PMD level due to limited coverage, while the upper levels enjoy high hit rates (96.0%–99.9%) across all workloads. Among the workloads, graphs exhibit moderate TLB and PWC miss rates of about 60% to 70%, whereas MUMmer, memcached and GUPS exhibit very high TLB and PWC miss rates of over 90%. Instead, the LWC of LVM enjoys high hit rates of above 99% across all applications due to the small size of the learned index.

Connecting PTW to L1/L2 cache. We consider an additional configuration that connects the MMU page table walkers to the L1 cache instead of the L2 cache. On average, LVM outperforms radix by 11% by connecting to the L1 cache and by 14% when connecting to the L2. Connecting to the L1 enables faster page walks due to L1 cache hits for page table entries but it significantly increases cache pressure since page table entries contend at the L1 cache with regular data. Specifically, while the MMU overhead reduces in general, radix benefits a bit more as as its sequential multi-access page walks can now potentially hit at the L1 cache, which has lower latency. However, the page walk traffic at the L1 drastically increases the L1 MPKI by 59% for radix, while LVM performs better, increasing L1 MPKI less, by 38%. This is because LVM sends close to 43% less page walk traffic to the caches as we discussed earlier (Section 7.1). Overall, LVM significantly outperforms radix irrespective of configuration.

7.3 Learned Page Table Characterization

Collision Rates. Collisions during page walk occur when multiple VPNs are predicted by the learned index to reside in the same location in the page table. To characterize the effectiveness of LVM, we measure collision rates and compare LVM to a hash table that has a load factor of 0.6 and uses the state-of-the-art hash function Blake2 [6]. On average, LVM achieves a minimal collision rate of 0.2% for 4KB and 0.6% for THP. This is a drastic improvement over regular hash tables, which have a collision rate of 22% for 4KB and 19% for THP. Notably, several graph workloads enjoy near-zero collision rates under LVM. Overall, LVM achieves a very low collision rate by tailoring to applications' virtual address spaces.

Collision Resolution. When there is a collision during address translation, LVM searches for the actual location of the translation entry corresponding to the VPN. As we discussed in Section 4, LVM bounds the number of memory accesses involved in such a search by enforcing an upper bound of $C_{\text{err}} = 3$ when building the model. To understand how this upper bound behaves in practice, we characterize the number of additional memory accesses across our workloads. We identify that the average number of additional memory accesses per collision across all workloads is 2.36 memory accesses. Since LVM has collision rates of only 0.6%, it requires a single-access 99.4% of the time on average across all workloads.

Index Size	bfs	pr	cc	dc	dfs	sssp	gups	mem$	MUMr
LVM 4KB	112	112	112	112	112	112	96	112	128
LVM THP	192	192	192	192	112	192	112	112	160

Table 2: Index size of LVM in bytes.

Index Size, Cacheability, and Scaling. In Table 2, we present the index size in bytes during the steady state. The results show that LVM requires a minimal index size of 112 bytes and 162 bytes on average for 4KB pages and THP, respectively. To determine the appropriate size for the LVM page walk cache (LWC), we analyze the steady-state and peak index sizes. The peak index size during the initial training phase averages 570 bytes, after which it stabilizes to the steady-state size. Because the peak and steady-state indexes are small, the LWC can be sized to comfortably accommodate the entire index. As discussed in the next section, the LWC required by LVM is 3× smaller than that of radix page tables.

Importantly, the learned index size is not dependent on the memory footprint. For example, memcached uses six times more memory than MUMmer but requires a smaller index. To better showcase this property, we conducted a scaling study based on memcached, where the working set was scaled from 32 GB, to 64 GB, 128 GB, and 240 GB. The corresponding steady-state index sizes were all 112 bytes. This contrasts sharply with radix page tables, which require linearly more entries in the page walk caches as the memory footprint grows, making LVM a future-proof solution.

LVM Overheads in the OS. The operating system is responsible for managing LVM for each application throughout its lifetime. The computational costs are included in the simulation as they occur, including the initialization of the learned index and ongoing management during simulation. For 4KB pages, the management overhead averages 1.17% of the total execution time across all workloads, with a peak of 1.91% for dfs. A primary reason for the low computational cost of LVM is its minimal and lightweight retraining, and beyond the short window of simulation, the overhead will be even more diluted. For THP, the percentage of time spent on management is less than 0.01% for all workloads since the OS has significantly fewer pages to manage. Overall, the management overhead of LVM is negligible.

We further use our OS prototype to run applications end-to-end for their complete runtime, far beyond what is possible in simulation, and measure the frequency and the computational cost of retrainings. Across workloads, the frequency of retraining is very low. Retrains occur at most 3 times, and 2 times on average. Importantly, retrains are very fast, less than 1.7ms on average and with the worst case of 1.9ms. This is because application address spaces remain largely unchanged after initialization, with expansions handled efficiently without retraining (Section 4.3.4). We further perform a tail latency study with memcached. Our results show that LVM computational costs do not affect even the 99th percentile tail latency.

Memory Consumption. Beyond the space of the page tables entries, every page table structure has additional memory overhead—radix page tables require space for upper-level page tables, and hashed page tables require over-provisioning of the hash tables to sit below an occupancy threshold. We compute the overhead of memory usage of LVM by calculating the additional space used in comparison to the absolute minimum required: eight bytes for every translation entry that maps a physical page. The overhead of LVM comes from the gapped array organization, which, in the worst case, uses an additional 1.3 times the space for PTEs than what is absolutely necessary. For example, LVM uses at most an additional 12MB of memory compared to the minimum required page table for MUMmer with 20GB of memory footprint, while ECPT uses an additional 27MB. Note that, as we discussed in Section 4.2.2, gapped page tables are not required to be physically contiguous.

Memory Fragmentation As we discussed in Section 4, LVM adapts to the available physical memory contiguity in the system. To this end, we evaluate a range of memory fragmentation levels. We first limit the available physical contiguity for LVM allocations to at most 256 KB, which represents 30% of free memory even in highly fragmented production datacenters as shown in Figure 3. Furthermore, we study three even more severe levels of fragmentation with a free memory fragmentation index (FMFI) [31, 52] of 0.8, 0.85, and even 0.9. When physical contiguity is limited, LVM dynamically adjusts the number of leaf nodes based on the available contiguity during allocation. Importantly, per-node coverage of LVM remains high, resulting in high LWC hit rates (above 99%). In practice, such severe fragmentation is not maintainable as the OS immediately compacts memory to replenish multi-KB contiguity. Overall, LVM's performance remains the same in production environments and even at more aggressive fragmentation levels due to its ability to dynamically allocate small non-contiguous gapped page tables based on the available physical contiguity.

7.4 Hardware Characterization

We implemented the page walker of LVM in RTL and synthesized it in a commercial 22nm PDK, and used CACTI [7] to generate the SRAMs of the caching structures. Our results show that a page walk model calculation and lookup in the LWC can be completed in 2 cycles at 2GHz and a single LVM page walker requires $0.000637mm^2$ of area. The LWC of LVM requires $0.00364mm^2$ of area and $0.588mW$ of leakage power, both lower than Radix PWC. The hardware structures of LVM achieve 3.0×, 1.5× and 1.9× improvement over radix in terms of size in bytes, area, and power, respectively.

7.5 Additional Comparison to Prior Work

Beyond ECPT, we further compare LVM to additional prior work.

7.5.1 Comparison to ASAP. ASAP [59], which relies on both virtual and physical contiguity to allocate leaf page tables in contiguous physical memory and to prefetch them during page walks. ASAP performs worse than both ECPT and LVM, with average slowdowns of 3% and 8%, respectively. This performance degradation is primarily due to the prefetcher introducing additional memory traffic on top of the standard page walk. Moreover, ASAP requires the operating system to allocate large contiguous regions of physical memory—potentially hundreds of megabytes—at VMA creation time. In practice, such large contiguous allocations are difficult to guarantee [95].

7.5.2 Comparison to Midgard. Midgard [34] reduces address translation overheads by creating an intermediate Midgard address space for data in the cache hierarchy, thus avoiding page walks for cache hits. However, Midgard still relies on radix page tables to access main memory. We evaluated an ideal Midgard that does not include any OS overheads. Midgard achieves, on average, a modest speedup of 3% compared to Radix, and LVM outperforms Midgard by 11%. This is because memory-intensive workloads exhibit a high L3 MPKI, which requires frequent radix page table walks. To further test the scalability of Midgard with larger caches, we further evaluated a Midgard design with twice the LLC slice size per core, resulting in 4MB LLC slices. Both LVM and Midgard achieve higher speedups. However, even with such a large LLC capacity, Midgard achieves only modest speedups over radix of 5%, and LVM still outperforms Midgard by 11%. Overall, we believe that Midgard is orthogonal to LVM as a potential design can rely on Midgard combined with LVM page tables for maximum benefits.

7.5.3 Comparison to Flattened Page Tables. Flattened Page Tables [64] (FPT) reduce the number of memory accesses per page walk by folding adjacent levels of Radix Page Tables, but requires 2MB or 1GB pages for folding to succeed. As we show in Figure 3, there is a limited supply of 2MB or larger physical contiguity in datacenters, which makes flattening difficult in practice. Furthermore, since it is unlikely that the high latency caused by memory reclaim or compaction can be tolerated at page fault time, the success rate of 2MB or larger physical memory allocations will be even lower. Instead, LVM does not require such large contiguity. Another drawback of FPT is that it constrains the concurrent use of multiple page sizes, e.g., 4KB, 2MB, and 1GB in the same virtual region, depending on which levels of the radix page tables are folded. Real-world applications benefit from multiple page sizes [95].

To demonstrate the difficulty of FPT allocations, we ran a simulation of FPT where both L4+L3 and L2+L1 radix page tables are flattened using 2MB pages. Our results show that in lightly fragmented environments, LVM outperforms FPT by 5% while in highly fragmented environments, the performance of FPT degrades to that of radix. The main challenge of FPT is that 2MB page table allocations compete with 2MB page allocations for application data, which, as a result, can lead to FPT allocating regular radix page tables, especially as fragmentation increases. On the other hand, LVM is responsive to the available memory contiguity (Section 4.2.2), resulting in single-access page walks even with limited physical memory contiguity. Finally, LVM flexibly supports all page sizes.

8 Related Work

A large body of work has focused on reducing the computational cost of address translation [10, 27, 29, 30, 33, 34, 36, 38, 47, 52, 56–58, 62–67, 69, 70, 77, 79–81, 84, 87, 88, 92, 94, 95] through alternative page table structures, large and coalesced translation entries, and huge page support in the OS.

Relevant to this work is [56], which leverages machine learning techniques to improve the allocation of huge pages. Other works have focused on the application of learned indexes for storage FTL [81] and persistent memory [19, 53]. The most relevant prior work to LVM is a short paper [58] that applied a learned index [51] using neural networks to predict translation locations in radix page tables. This approach did not support insertions and fell back to traditional page table walks on mispredictions, effectively functioning as a prefetching mechanism. The paper concluded that existing learned indexes were not suitable for address translation due to their large model sizes and high computational requirements resulting in 120 cycles to produce a PTE location. While the idea was forward-looking, we believe the primary pitfall of that work was its adherence to radix page tables for backward compatibility with existing operating systems. It further relied on an off-the-shelf learned index with heavyweight neural network functions originally designed for databases, which introduced significant implementation challenges. In contrast, LVM reimagines the end-to-end page table structure to unlock a highly efficient and practical design.

9 Discussion and Future Work

Learned indexes are a promising new class of data structures that accelerate indexing operations in software. LVM pioneers the first practical learned index for hardware, specifically for address translation. To make learned indexes suitable for fast hardware lookups, LVM employs cost models that bound index depth and size, introduces efficient mechanisms for key updates, fast lookups, and relaxes the need for contiguous memory allocations. These advances are essential to applying learned indexes in hardware.

Looking ahead, the LVM framework can also benefit other hardware caching structures indexed by virtual addresses, such as TLBs, branch target buffers (BTBs), and L1 caches. Beyond this, we envision extending LVM to physically indexed structures like last-level caches. Such structures often suffer from hash-table-like collisions that cause conflict misses and reduce hit rates. By leveraging lightweight machine learning, the LVM framework offers a promising direction to mitigate these collisions and improve performance. We leave this exploration to future work.

10 Conclusion

We presented Learned Virtual Memory (LVM), a page table structure that effectively provides optimal single-access address translation. LVM relies on a novel learned index model that tailors address translation to the virtual address space of applications. Our evaluation shows that LVM performs within 1% of an ideal page table.

Acknowledgments

This work was funded in part by NSF grants CNS-2239311, CNS-2107307, CCF-2217016, by Intel Corp under the SRS award, a Meta Faculty Award, an Oracle Faculty Award, and a Linux eBPF Foundation Faculty Award. Kaiyang Zhao and Yuang Chen were partly funded by a Qualcomm Innovation Fellowship. We thank Boris Grot and the anonymous reviewers for all of their valuable feedback.

References

[1] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.

[2] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting Hardware-assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. Portland, Oregon.

[3] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*.

[4] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. 2005. BioBench: A Benchmark Suite of Bioinformatics Applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*.

[5] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA'20)*.

[6] Jean-Philippe Aumasson, Willi Meier, Raphael Phan, and Luca Henzen. 2014. *The Hash Function BLAKE*. Springer.

[7] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization* 14, 2, Article 14 (June 2017), 14:1–14:25 pages.

[8] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*.

[9] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*.

[10] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*.

[11] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. USENIX, 10–5555.

[12] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*.

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

[14] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.

[15] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.

[16] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*.

[17] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Reconsidering Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM. https://doi.org/10.1145/3669940.3707217

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

[19] Lixiao Cui, Yijing Luo, Yusen Li, Gang Wang, and Xiaoguang Liu. 2024. When Learned Indexes Meet Persistent Memory: The Analysis and the Optimization. *IEEE Transactions on Knowledge and Data Engineering* 36, 12 (2024), 9517–9531. https://doi.org/10.1109/TKDE.2023.3342825

[20] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[21] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[22] Cort Dougan, Paul Mackerras, and Victor Yodaiken. 1999. Optimizing the Idle Task and Other MMU Tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*.

[23] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[24] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. 2023. Towards a Machine Learning-Assisted Kernel with LAKE. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 846–861. https://doi.org/10.1145/3575693.3575697

[25] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (2004).

[26] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. 2005. Space Efficient Hash Tables with Worst Case Constant Access Time. *Theory of Computing Systems* 38, 2 (Feb. 2005), 229–248.

[27] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. Cambridge, United Kingdom.

[28] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16).*

[29] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14).*

[30] Mel Gorman and Patrick Healy. 2010. Performance Characteristics of Explicit Superpage Support. In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10).*

[31] Mel Gorman and Andy Whitcroft. [n. d.]. Linux Symposium, title = The what, the why and the where to of anti-fragmentation., year = 2005,.

[32] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. 2005. Itanium — A System Implementor's Tale. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC'05).*

[33] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th ACM International Conference on Virtual Execution Environments (VEE'15).*

[34] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. 2021. Rebooting Virtual Memory with Midgard. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* 512–525. https://doi.org/10.1109/ISCA52012.2021.00047

[35] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. 2020. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* 1050–1063. https://doi.org/10.1109/ISCA45697.2020.00089

[36] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* Williamsburg, VA, USA.

[37] Jerry Huck and Jim Hays. 1993. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93).*

[38] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21).* USENIX Association, 257–273. https://www.usenix.org/conference/osdi21/presentation/hunter

[39] Intel. 2022. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[40] Bruce L. Jacob and Trevor N. Mudge. 1998. A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII).*

[41] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova. 2024. EXTMEM: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC'24).*

[42] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems. *ACM Trans. Archit. Code Optim.* 16, 1, Article 6 (mar 2019), 24 pages. https://doi.org/10.1145/3309710

[43] Joefon Jann, Paul Mackerras, John Ludden, Michael Gschwind, Wade Ouren, Stuart Jacobs, Brian F. Veale, and David Edelsohn. 2018. IBM POWER9 system software. *IBM Journal of Research and Development* 62, 4/5 (June 2018).

[44] jemalloc. 2025. jemalloc. https://jemalloc.net.

[45] Konstantinos Kanellopoulos, Rahul Bera, Kosta Stojiljkovic, F. Nisa Bostanci, Can Firtina, Rachata Ausavarungnirun, Rakesh Kumar, Nastaran Hajinazar, Mohammad Sadrosadati, Nandita Vijaykumar, and Onur Mutlu. 2023. Utopia: Fast and Efficient Address Translation via Hybrid Restrictive & Flexible Virtual-to-Physical Address Mappings. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23).* Association for Computing Machinery, New York, NY, USA, 1196–1212. https://doi.org/10.1145/3613424.3623789

[46] Konstantinos Kanellopoulos, Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini, and Onur Mutlu. 2023. Victima: Drastically Increasing Address Translation Reach by Leveraging Underutilized Cache Resources. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23).* Association for Computing Machinery, New York, NY, USA, 1178–1195. https://doi.org/10.1145/3613424.3614276

[47] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15).*

[48] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. 2016. Energy-Efficient Address Translation. In *Proceedings of 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16).*

[49] Samuel T. King, George W. Dunlap, and Peter M. Chen. 2003. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC'03).*

[50] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management.* 1–5.

[51] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data.* 489–504.

[52] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16).*

[53] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: a high-performance learned index on persistent memory. *Proc. VLDB Endow.* 15, 3 (Nov. 2021), 597–610. https://doi.org/10.14778/3494124.3494141

[54] Yongping Luo, Peiquan Jin, Zhaole Chu, Xiaoliang Wang, Yigui Yuan, Zhou Zhang, Yun Luo, Xufei Wu, and Peng Zou. 2023. Morphtree: a polymorphic main-memory learned index for dynamic workloads. *The VLDB Journal* (2023), 1–20.

[55] Piotr R. Luszczek, David H. Bailey, Jack J. Dongarra, Jeremy Kepner, Robert F. Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06).*

[56] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[57] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S. McKinley, and Paul Turner. 2021. Adaptive Huge-Page Subrelease for Non-Moving Memory Allocators in Warehouse-Scale Computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (Virtual, Canada) *(ISMM 2021).* Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/3459898.3463905

[58] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2018. Virtual address translation via learned page table indexes. In *Workshop on ML for Systems at NeurIPS.*

[59] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52).* Association for Computing Machinery, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[60] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15).*

[61] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* 51, 2 (May 2004), 122–144.

[62] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19).*

[63] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).*

[64] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every walk's a hit: making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22).* Association for Computing Machinery, New York, NY, USA, 128–141. https://doi.org/10.1145/3503222.3507718

[65] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14).*

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

[67] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. 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-48).* Waikiki, Hawaii, USA.

[68] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[69] Stratos Psomadakis, Chloe Alverti, Vasileios Karakostas, Christos Katsakioris, Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2024. Elastic Translations: Fast Virtual Memory with Multiple Translation Sizes . In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 17–35. https://doi.org/10.1109/MICRO61859.2024.00012

[70] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. 2021. Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1106–1120. https://doi.org/10.1145/3466752.3480062

[71] Facebook Research. 2025. DCPerf benchmark suite for hyperscale cloud applications. https://github.com/facebookresearch/DCPerf.

[72] Arun F. Rodrigues, Jeanine Cook, Elliott Cooper-Balis, K. Scott Hemmert, Chad Kersey, Rolf Riesen, Paul Rosenfeld, Ron Oldfield, and Marlow Weston. 2006. The Structural Simulation Toolkit. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'10)*. Tampa, Florida.

[73] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[74] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. Toronto, ON, Canada.

[75] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. 2022. Can Learned Models Replace Hash Functions? *Proceedings of the VLDB Endowment* 16, 3 (2022), 532–545.

[76] Samsung. 2019. Samsung Electronics Introduces Industry's First 512GB CXL Memory Module. https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module.

[77] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*.

[78] J. E. Smith and Ravi Nair. 2005. The Architecture of Virtual Machines. *IEEE Computer* 38, 5 (2005), 32–38. https://doi.org/10.1109/MC.2005.173

[79] Jovan Stojkovic, Namrata Mantri, Dimitrios Skarlatos, Tianyin Xu, and Josep Torrellas. 2023. Memory-Efficient Hashed Page Tables. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1221–1235. https://doi.org/10.1109/HPCA56546.2023.10071061

[80] Jovan Stojkovic, Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2022. Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. https://doi.org/10.1145/3503222.3507720

[81] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. 2023. Leaftl: A learning-based flash translation layer for solid-state drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 442–456.

[82] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VI)*. Association for Computing Machinery, New York, NY, USA, 171–182. https://doi.org/10.1145/195473.195531

[83] Madhusudhan Talluri and Mark D. Hill. 1994. 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)*.

[84] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. In *19th International Symposium on Computer Architecture (ISCA'92)*.

[85] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*. 308–320.

[86] tcmalloc. 2025. tcmalloc. https://github.com/google/tcmalloc.

[87] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. 2021. Morrigan: A Composite Instruction TLB Prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1138–1153. https://doi.org/10.1145/3466752.3480049

[88] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. 2021. Exploiting Page Table Locality for Agile TLB Prefetching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 85–98. https://doi.org/10.1109/ISCA52012.2021.00016

[89] Ziqi Wang, Kaiyang Zhao, Pei Li, Andrew Jacob, Michael Kozuch, Todd Mowry, and Dimitrios Skarlatos. 2023. Memento: Architectural Support for Ephemeral Memory Management in Serverless Environments. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 122–136.

[90] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable learned index with precise positions. In *Proceedings of the VLDB Endowment* (2021-04). 1276–1288. https://doi.org/10.14778/3457390.3457393

[91] www.7-cpu.com. 2023. Intel Skylake Timing. https://www.7-cpu.com/cpu/Skylake.html.

[92] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA'19)*. Phoenix, AZ, USA.

[93] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don't Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS'16)*.

[94] Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. 2024. Direct Memory Translation for Virtualized Clouds. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (, La Jolla, CA, USA,) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 287–304. https://doi.org/10.1145/3620665.3640358

[95] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. 2023. Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. New York, NY, USA, 1–15. https://doi.org/10.1145/3579371.3589079