Predictive Translation: High-Performance Buffer Management Without the Trade-Offs

Michael Zinsmeister
michael.zinsmeister@tum.de
Technische Universität München
Germany

Lam-Duy Nguyen
lamduy.nguyen@tum.de
Technische Universität München
Germany

Viktor Leis
leis@in.tum.de
Technische Universität München
Germany

Thomas Neumann
thomas.neumann@in.tum.de
Technische Universität München
Germany

Abstract

To efficiently manage larger-than-memory datasets, storage-based database management systems (DBMSs) rely on buffer managers. These are traditionally implemented using hash tables to translate page identifiers (PIDs) to memory pointers. While this design offers many practical advantages, prior studies have shown its performance limitations and proposed alternative designs to close the gap with optimized in-memory DBMSs. However, these modern designs introduce systematic issues, such as intrusive implementations or reliance on kernel modules, which ultimately hinder their adoption.

This paper challenges the notion that hash-table-based buffer pools cannot deliver high performance. We introduce *predictive translation*, a novel approach that combines the high performance of modern approaches with the qualitative benefits of traditional designs. Predictive translation achieves this by exploiting the capabilities of commodity CPUs – particularly their superscalar execution – through deterministic placement of pages to hide the excessive latency of software-level hash table lookups. Our evaluation demonstrates that our approach meets all practical requirements while delivering performance at least on par with state-of-the-art alternatives. We show that our design is a compelling solution for buffer management in modern DBMSs running on fast storage devices.

Keywords

Database Management Systems; Caching; Buffer Management

ACM Reference Format:
Michael Zinsmeister, Lam-Duy Nguyen, Viktor Leis, and Thomas Neumann. 2026. Predictive Translation: High-Performance Buffer Management Without the Trade-Offs. In *Proceedings of Proceedings of the 2026 International Conference on Management of Data (SIGMOD '26).* ACM, New York, NY, USA, Article 64, 16 pages. https://doi.org/10.1145/3786678

1 Introduction

Caching for disk-based DBMSs. Historically, database management systems (DBMSs) relied on mechanical disks as their primary non-volatile storage medium, necessitating a *buffer manager* for efficient data access. Traditional buffer pool designs commonly use a *hash table* to map logical page identifiers (PIDs) to *buffer frames*, i.e., the in-memory mirrors of the physical page [34]. This hash-table-based approach offers numerous advantages, including ease of use, implementation simplicity, robustness, and many more, making it the *de facto standard* for DBMS caching.

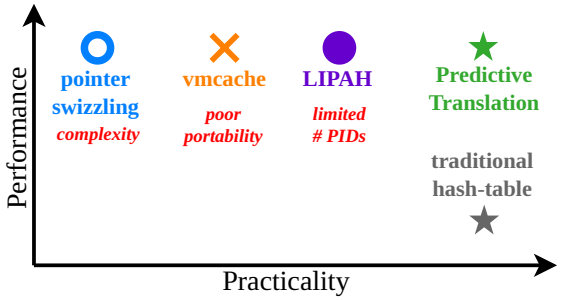


Figure 1: Modern buffer pools offer high performance but have practical limitations that hinder adoption. Predictive translation combines the robustness of traditional designs with modern efficiency.

From in-memory to SSD-optimized caching. Over the last decade, hardware advancements have significantly reshaped the DBMS landscape. Initially, falling DRAM costs resulted in the transition from disk-based DBMSs to high-performance in-memory DBMSs, which completely *discards* the buffer pool to achieve optimal performance [30, 35, 37, 48, 52, 57, 62, 73, 86, 92, 93]. However, this trend was short-lived as DRAM prices stagnated; simultaneously, flash-based solid-state drives (SSDs) experienced dramatic price drops [66]. Altogether, these hardware shifts have initiated the development of SSD-optimized DBMSs that offer performance *comparable* to in-memory systems when the working dataset fits in memory [16, 51, 54, 66, 69, 80, 85, 98]. Achieving this efficiency hinges on a lightweight buffer manager. To this end, techniques like pointer swizzling [44, 60, 61, 69], Logical ID with Physical Address Hinting (LIPAH) [24], and virtual-memory-assisted caching (vmcache) [67] have proven to be particularly effective.

Qualitative problems. While these designs offer performance advantages, they come with significant qualitative trade-offs that limit their practicality. For example, pointer swizzling [44] restricts each

SIGMOD '26, May 31–June 5, 2026, Bengaluru, India

page to a single reference, making it unsuitable for workloads requiring cyclic references, such as graph-like data structures. Another design, vmcache [67], requires a kernel module to achieve good performance, which has several practical issues (cf., Section 2.1). These systematic issues stem from the fundamental aspects of their designs, leaving limited room for viable workarounds and thereby hindering widespread adoption. Given these challenges, it is worth reconsidering the practical advantages of traditional hash-table-based designs and asking whether a hash-table-based buffer pool can achieve performance comparable to in-memory DBMSs.

Contribution: Predictive translation. In this work, we challenge the prevailing belief that hash tables represent a major bottleneck in DBMS buffer management. We tailor a novel buffer management approach, *predictive translation*, that employs a hash table to map PIDs to in-memory buffer frames, thereby avoiding the qualitative shortcomings of modern buffer caches, as illustrated in Figure 1. Nevertheless, predictive translation delivers high performance like other state-of-the-art approaches, as demonstrated in Section 6. The key idea is to *exploit* the superscalar execution capabilities of modern CPUs by placing hot pages in predictable locations. This enables the CPU to speculatively access their contents while the hash table translation is still in progress, effectively masking its overhead. We demonstrate that the proposed approach offers a compelling solution for DBMS buffer management by achieving both high performance and practical usability.

Outline. The rest of this paper is organized as follows: In Section 2, we review existing buffer pool implementations and highlight their limitations. Next, Section 3, explains the conceptual ideas of *predictive translation*, specifically how it leverages the *superscalar execution* of CPUs to improve performance. A lightweight hash table design is presented in Section 4 that, despite its simplicity, is highly efficient and serves as a crucial building block for our buffer pool. Some implementation details follow in Section 5. Finally, we evaluate our design in Section 6, discuss additional related work in Section 7, and conclude in Section 8.

2 Buffer Management and Its Design Goals

In this section, we examine modern buffer management designs to highlight their practical limitations compared to traditional hash-table-based buffer pools. Based on this in-depth discussion, we outline the key *design goals* of a modern buffer cache – which explain why recent proposals have struggled to gain widespread adoption. Finally, we revisit traditional hash-table-based buffer pools to identify their performance bottlenecks, laying the foundation for our proposed approach.

2.1 Buffer Management

Storage-based DBMSs typically structure data into fixed-size pages ranging from 4 to 64 KB. These systems often employ a *buffer manager* to cache frequently accessed pages and manipulate them [34]. The primary operations of these buffer managers involve efficient *page requests*, where the DBMS translates from a PID into an in-memory *buffer frame* that stores the content of the page. In the following, we describe existing designs for DBMS buffer management and discuss their problems in detail.

Table 1: Comparative analysis of buffer pool designs.

	impl.	# pages	meta.[1]	perf.	graphs
traditional [56]	easy	high	low	low	yes
ptr swiz. [44, 69]	hard	high	low	high	no
LIPAH [24]	med	low	low	high	yes
vmcache [67]	med[2]	med	high	high	yes
Predictive Transl.	easy	high	low	high	yes

[1] Per-page metadata overhead
[2] Require OS modification; Hard to support sampling-based eviction

Traditional: General-purpose hash table cache. The most common design for buffer management is to use a general-purpose hash table. In this design, every page request involves a hash-table lookup: if found, the corresponding buffer frame is returned; otherwise, the buffer manager allocates a new frame for the corresponding page and reads its content from storage. It may also need to evict a currently cached page to free space. The typical API consists of *fix*, which pins and latches a page – loading it from storage if necessary – and *unfix*, which unpins and unlatches the page [34]. This design is robust and incurs minimal memory overhead, hence it is the standard for traditional DBMSs such as MySQL [3], PostgreSQL [4], SQLite [5], and even high-performance research systems like Shore-MT [56].

Traditional: Performance problems. However, despite its many qualitative advantages, previous works argued that such a hash-table-based design results in sub-optimal in-memory performance [30, 44, 52, 66, 67, 69], leading to the introduction of the three modern buffer management approaches: pointer swizzling [44, 69], LIPAH [24], and vmcache [67].

Pointer swizzling: Invasive caching. The introduction of in-memory DBMSs has driven the demand for a low-overhead buffer cache, with pointer swizzling emerging as a promising solution [12, 44, 69]. The key idea is to convert every page request into a *single* pointer dereference, where the pointer directly references the memory address of the associated buffer frame. This is achieved by *invasively* modifying the data structure to replace PIDs with the memory addresses of the corresponding frames. Pointer swizzling has demonstrated exceptional performance, even identical to in-memory systems when the working set fits in memory [44, 69, 80].

Pointer swizzling: Practical issues. The key problem of pointer swizzling is its invasive design, which complicates data structure implementation and synchronization. For example, page eviction in pointer swizzling involves exclusively latching both the to-be-evicted page and its parent node [43, 66, 69]. Another significant limitation is that pointer swizzling typically does not support multiple references to pages in real-world implementations, a feature that is crucial for graph-like data structures and B+tree scan operations involving sibling pointers [24, 64].

LIPAH: Addressing problems of pointer swizzling. Logical ID with Physical Address Hinting (LIPAH) [24] is a buffer pool design inspired by pointer swizzling. That is, instead of storing a swizzled pointer, LIPAH maintains a *fat pointer* – a pair consisting of both the PID and the memory address of the buffer frame. Doing so allows the frame address to be incorrect, as the buffer pool can detect mismatches and fall back to a hash-table lookup using the PID. This

design also helps LIPAH avoid the single-reference limitation of traditional pointer swizzling.

LIPAH: Limited PID space. A major limitation of LIPAH is its restricted database size. Since LIPAH represents PIDs with only 4 bytes, it can address at most 2^{32} pages. With a 4KB page size, this limits the database to just 17TB – far smaller than other approaches and insufficient for datacenter-grade NVMe SSDs [8, 10, 47–49]. Additionally, LIPAH still necessitates invasive modifications to index data structures like pointer swizzling to achieve optimal performance, leading to increased complexity.

Virtual-memory assisted cache. Leis et al. [67] proposed a virtual-memory assisted buffer manager (vmcache), another lightweight and high-performance approach for buffer management in storage-based DBMSs. This design leverages the OS page table for translating PIDs to buffer frames, an operation that is inherently fast for in-memory workloads due to hardware acceleration from the Translation Lookaside Buffer (TLB) and CPU support for page table walks. Despite its simplicity, vmcache introduces several significant problems:

- *Dependency on a kernel module*: This approach relies on a kernel module (exmap) to achieve high-performance in out-of-memory workloads. Given that OS kernels evolve rapidly and frequently deprecate internal APIs, maintaining exmap (e.g., for compatibility with different kernel versions, security concerns[1]) becomes burdensome and undesirable. eBPF is a more secure alternative [23, 32]; however, its strict security constraints significantly limit programmability [23]. Because of these restrictions, eBPF does not yet support page table modifications. It remains uncertain whether such functionality will ever be introduced due to security requirements. Finally, eBPF modules must still be loaded with root privileges, which continues to pose a usability issue.
- *Excessive metadata overheads*: vmcache needs to allocate buffer frames (and their associated headers) proportional to the size of the storage media. For instance, with a modern enterprise SSD of 30TB [8, 10], the DBMS manages approximately 8 billion 4KB pages. Assuming the frame header is 64 bytes (used by PostgreSQL [4]), this results in a staggering 480GB of memory overhead solely for buffer pool metadata.
- *Hard to support sampling-based eviction strategy*: The second constraint also rules out the use of efficient sampling-based cache eviction strategies, such as WATT [94] and HyperBolic [21], which are highly relevant for optimizing DBMS workloads as highlighted in these studies. These strategies require per-page metadata to be stored in a quickly accessible place like the buffer frame header.
- *Limited page capacity*: The high metadata overhead highlighted above also directly limits the number of physical pages on disk that vmcache can manage. Notably, even the system used to evaluate vmcache had only 512GB of memory [67], implying that vmcache will struggle to scale with modern storage.

[1]When we evaluate vmcache(+exmap), we have to disable OS secured boot and Supervisor Mode Access Prevention [2]; both pose significant security risks.

2.2 Design Goals

Based on the previous analysis, we can outline five key *design goals* that a modern buffer manager should satisfy:

(1) *High performance*: Achieves performance comparable to pure in-memory DBMSs when the dataset fits in memory. This requires the buffer pool to be highly efficient, introducing minimal overhead. Ideally, a translation should be at most around one CPU cache miss, like in pure in-memory systems [66, 69, 80, 102].
(2) *Low memory overhead*: Metadata size is proportional to the buffer cache size rather than total storage size. This allows for richer per-frame metadata like contention statistics for B-Tree pages, which are used by *contention-split* [13]. It also enables sampling-based eviction strategies [21, 94], improving performance in out-of-memory workloads.
(3) *Portability*: The buffer pool should avoid reliance on OS modifications or kernel modules, and also offer straightforward APIs that don't necessitate changes to other subsystems (e.g., indexing data structures).
(4) *Multiple page references*: Support for cyclic/multiple page references without negatively impacting the implementation of index data structures.
(5) *Unlimited number of pages*: The buffer pool should accommodate a (practically) unlimited number of pages and not be limited by per-page metadata overhead or a 32 bit address space.

Since using a hash table for buffer pool translation satisfies all but the performance-related design goals (i.e., goals 2–5), a promising approach is to address the performance issues directly rather than attempting to fix the qualitative shortcomings of alternative designs, which often stem from their fundamental architecture. In this paper, we introduce a novel buffer management approach, predictive translation, that achieves high performance by exploiting the capabilities of commodity CPUs, especially their *superscalar execution*, to hide the overheads typically associated with general-purpose hash tables. Table 1 summarizes the conceptual differences between our solution and previous buffer managers.

2.3 Superscalar Execution: Best Practices

As hinted at through previous sections, predictive translation takes advantage of *superscalar execution* to improve performance. Here, we provide a brief explanation of how superscalar execution works using the following example:

```
// Dependent accesses
T* ptr2 = *ptr1;
T data = *ptr2;
```

```
// Independent accesses
T data1 = *ptr1;
T data2 = *ptr2;
```

Assuming both code snippets incur two cache misses, the independent version (right) enables modern CPUs, which all have a superscalar architecture, to issue both loads *in parallel*. In contrast, the dependent version (left) introduces a serialization bottleneck: the second load must wait for the first to complete. This dependency can nearly double the execution time. Given that memory accesses are considerably more expensive than arithmetic or logic operations, such dependencies have a substantial negative impact on overall performance.

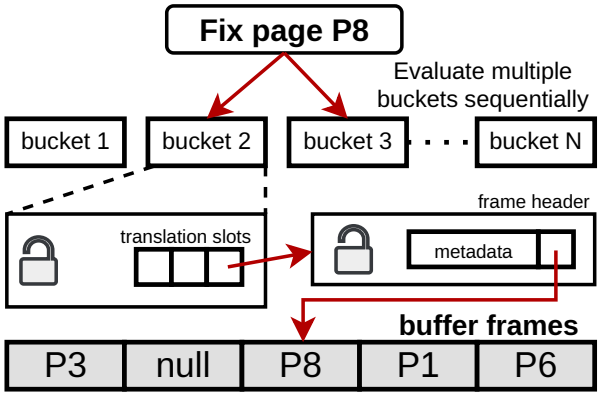


Figure 2: Analysis of a traditional buffer manager (e.g. Shore-MT) reveals two bottlenecks: (1) Bucket latch and (2) Each page request undergoes multiple sequential indirections (red arrows), together causing significant overheads.

2.4 Analysis of Traditional Hash-Table-Based Buffer Managers

Prior research has argued that using a hash table for buffer pool translation introduces significant overheads, making it unsuitable for high-performance in-memory workloads [24, 52, 67, 69]. To understand this conclusion, we must first examine the bottlenecks that hash tables introduce in buffer management. In this section, we analyze the buffer manager of Shore-MT [9, 56] to understand the performance behavior of previous hash-table-based buffer pools, and show it in Figure 2. Note that we will continue using Shore-MT as a representative for traditional buffer management, as its buffer manager design offers high-performance and is very similar to the ones used in systems like PostgreSQL or InnoDB [9, 56].

Problem #1: Hash table translations. One key takeaway from modern buffer pools is that PID-to-buffer-frame translations incur virtually no cost. For example, pointer swizzling [44, 69] embeds the memory address of the buffer frames directly within index structures, hence requiring no translation; vmcache [67] leverages the TLB to deliver efficient translations. In contrast, the hash-table-based translations, as used in Shore-MT [9, 56] and many other DBMSs such as MySQL and PostgreSQL, are significantly more expensive – typically several times costlier than modern alternatives. Specifically, in Shore-MT, the buffer pool is partitioned into multiple buckets, requiring each page request to probe two possible locations following the Cuckoo hashing scheme [38, 56]. As illustrated in Figure 2, these probes must be executed sequentially, each acquiring a latch to safely iterate through a list of *buffer control blocks* (i.e., frame headers) containing metadata, the page latch, and the buffer frame's memory address. This process is inherently more computationally expensive than the translation mechanisms employed by modern buffer pools.

Problem #2: Dependent operations. This issue stems directly from the first problem: each level of indirection – PID to bucket, bucket to translation slot, slot to frame header, and frame header to buffer frame – depends on the result of the previous step. Consequently, every page request triggers a sequence of dependent operations (red arrows in Figure 2), each likely causing at least one cache miss and adding to the overall latency. As noted in Section 2.3, this chain of dependencies directly conflicts with *superscalar execution*, limiting the CPU's ability to parallelize and accelerate the costly buffer pool operations. Furthermore, as shown in Figure 2, an additional latch, the *page latch*, further exaggerates the performance of page request operations beyond the *bucket latch*. The page latch, which is typically placed within the frame header, is essential for synchronizing page accesses and implementing the *latch coupling* mechanism in index concurrency control [13, 56, 70, 79]. Altogether, this dependency chain, combined with the delays introduced by latching at different steps, significantly increases the latency of page request operations.

Summary. The main issue is that the hot path in Shore-MT's and similar systems' buffer pool operations is computationally expensive. Specifically, each page operation involves (1) numerous latches, which can introduce delays due to contention and also trigger CPU cache invalidation, and (2) multiple dependent memory accesses. These factors hinder performance and lead to limited scalability, falling short of our performance goals.

3 Predictive Translation

In a conventional buffer pool, the buffer frame corresponding to an arbitrary PID can reside anywhere in memory, necessitating a hash table lookup to locate its address. Consequently, every page request involves at least *two* dependent operations: a hash table translation followed by buffer frame evaluation (i.e., a page read). This dependency chain introduces latency and makes traditional designs notably slower than modern caching designs (and in-memory DBMSs), as discussed in Section 2.4.

3.1 Decoupling Translation from Access

Traditional memory pool: Problems. In conventional designs, buffer managers rely on a hash table to translate logical PIDs into buffer frame addresses [34, 52]. This design is simple and robust, explaining why it is widely adopted in many DBMSs like ShoreMT, PostgreSQL, and many more [3–5, 9]. However, this design suffers from performance limitations: Each page request incurs multiple cache misses due to at least *two dependent* operations. The primary reason is that buffer frames are assigned to PIDs randomly, so the system must perform a lookup in the translation layer to determine the frame address for every PID [34, 56].

Idea: Making buffer frames predictable. This raises an interesting question: what if we could accurately predict the buffer frame for any given PID? In that scenario, if the hash table translations are also non-blocking (cf., Section 4), the DBMS could leverage *superscalar execution* to interleave two operations: *reading the translated buffer frame* and *PID translation*. As a result, while our buffer pool still incurs those two operations, they are executed in parallel within a single CPU core, effectively reducing the perceived latency to that of a single page read.

Predictive translation. Building on this intuition, we introduce a novel concept: *Predictive translation*, where the buffer frame for a given PID can likely be predicted in advance. This enables us to accurately predict the memory address associated with the requested PID, exploiting superscalar execution to unlock high performance.

Listing 1: Costly page access in traditional design

```
1 // 1-2 cache misses from translation
2 BufferFrame* bf = hashtable.lookup(pid);
3 // perform read on page; depends on above op
4 read(bf)
5 // ⇒ total of 1 cache miss + page access
```

Listing 2: Predictive Translation: Fast path for in-memory workloads

```
1 BufferFrame* correct_bf = hashtable.lookup(pid) // (1) 1 cache miss
2 // predicting page is in its preferred position
3 BufferFrame* pred_bf = getPreferredBufferFrame(pid)
4 if (isPreferredPosition(correct_bf)): // rely on CPU branch predictor
5   read(pred_bf) // interleave with (1) ⇐ superscalar exec.
6 else:
7   read(correct_bf) // (rare) slow path
```

Our solution, inspired by ideas from vmcache [67], is to logically assign each PID a dedicated *preferred position/frame* in the memory pool. For instance, PID 2 may be assigned the second buffer frame, even if that frame does not necessarily always contain the actual content of page 2. This strategy enables an optimistic assumption that a PID maps directly to its preferred frame. As a result, page operations can proceed in parallel with the PID-to-frame translation, reducing overhead and improving performance.

In-memory benefit. Listings 1 and 2 compare traditional memory pool designs (Listing 1) with predictive translation (Listing 2). In the traditional approach (Listing 1), page reads must occur sequentially after hash table translation, making them costly. In contrast, predictive translation (Listing 2) masks most of the hash table overhead (line 2) with the page read (line 6), assuming these translations do not block. Consequently, a page read becomes significantly cheaper, appearing without hash table operations. As a result, the fast path in our buffer pool design is also the most common execution path when the dataset fits in memory (cf., Section 3.2), achieving performance akin to other modern buffer pools for in-memory workloads.

Misprediction effect. However, our design may occasionally make incorrect predictions, i.e., line 5 in Listing 2 evaluates to `false`. Such a misprediction involves two page reads, which is more expensive than a typical CPU cache miss. Nonetheless, we argue that as long as the *hot* pages (i.e., frequently accessed ones) are kept in their preferred locations, the overall performance benefit remains substantial. Moreover, in case of a misprediction, the thread would typically be stalled waiting for the hash table translation anyway, which helps mask the additional cost. The placement of *cold* pages, by contrast, has little impact and can be handled arbitrarily. Furthermore, in typical transactional workloads, effective buffer frame management can reliably place most hot pages into their expected frames. In Section 3.2, we detail how we organize buffer frames to ensure that hot pages will primarily reside in their preferred frame.

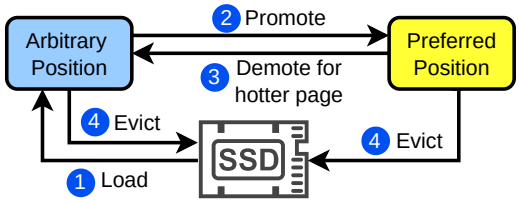
How to determine preferred position? To determine the preferred position, we use a hash function on the PID and apply a modulo operation with the buffer pool size. The calculated value from hashing PID can also be reused to avoid hashing a second time. Since the integer division required for modulo is typically slow, we precompute magic numbers at startup to replace division with faster multiplication operations [45].

3.2 Managing Frame Conflicts

A key challenge in our design is that multiple PIDs inevitably share the same preferred buffer frame. For example, in a buffer pool with 10 pages, PIDs 1, 11, 21, and so on will all share the same preferred frame – assuming we use modulo identity hash function to evaluate the preferred position. As a result, when allocating a frame for such PIDs, the preferred frame is often occupied, requiring a fallback to a random free frame. Without a carefully designed strategy, most PID-to-frame mappings would deviate from their preferred positions, undermining the performance gains outlined in Listings 1 and 2. In the following section, we describe how we manage the page lifecycle to maintain a high degree of determinism within the memory pool with little performance overhead.

Observation. In transactional workloads, data access patterns often exhibit a natural separation between hot and cold data. A significant portion of the cold data consists of *one-hit wonders* – pages that are accessed only once or very infrequently [17, 72]. These pages provide little opportunity for reuse and thus do not benefit from long-term caching. Therefore, we do not need to assign preferred frames to those pages.

Page lifecycle. Building on this observation, we propose the following page lifecycle:



First, (1) the DBMS loads the page from storage into an arbitrary free frame, assuming that the page is a *one-hit wonder*, unless the preferred frame is available for allocation. Second, (2) upon subsequent accesses, the DBMS acknowledges that the page is not a one-hit-wonder; as such, if the requested PID is not already mapped to its preferred position, the DBMS *probabilistically promotes* it to the preferred frame for more efficient accesses later.

Handling demotion. A previously promoted PID may already occupy the preferred buffer frame. In such cases, the DBMS uses a reduced promotion probability to mitigate thrashing. If the current PID is hotter, (3) it will eventually be mapped to the preferred frame by demoting the resident one. We note that the hotness level is not tracked explicitly; it is done based on a probabilistic approach for elasticity – which is detailed in Section 5. Lastly, if the page is later selected for eviction (4), the buffer pool removes the PID and marks the associated frame available for re-allocation.

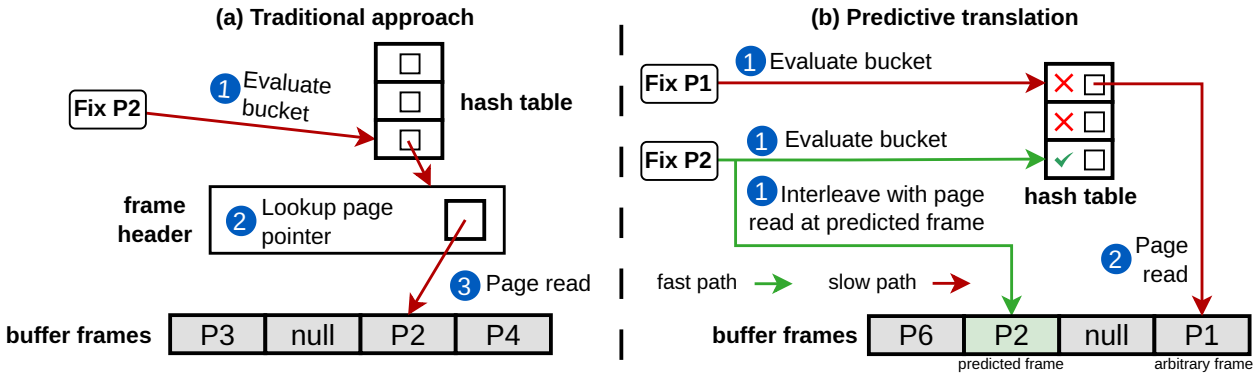


Figure 3: Comparison between a traditional buffer pool design (a) (simplified version of Figure 2) vs. Predictive Translation (b).

4 Hash Table for Predictive Translation

In Section 3, we discuss the key concept of predictive translation, which enables the buffer manager to interleave hash table operations with buffer frame evaluation by leveraging superscalar execution. This design imposes two critical requirements on the underlying hash table implementation. First, hash table operations must be as lightweight as possible; traditional designs often employ complex conflict-resolution mechanisms that make them computationally expensive and possibly hinder the benefits of superscalar execution. Second, these operations must be lock-free; otherwise, the translation step could block the concurrent execution of buffer frame evaluations. It is worth noting that general-purpose lock-free hash tables are notoriously tricky to implement correctly and tend to be computationally costly [59]. To address these challenges, this section introduces a set of simple yet effective techniques that enable fast, scalable, and lock-free hash-table-based translations for the buffer pool.

4.1 Lightweight Translation

In a hash-table-based buffer manager, a hash table maps PIDs to the memory addresses of the corresponding buffer frames. The mapping is determined by the memory pool component of the buffer pool (see Section 3). The hash table consists of multiple translation slots to store this mapping, and these slots may also include collision resolution information, such as linked lists in chaining hash tables.

Background: Traditional design. Traditional designs usually use a general-purpose hash table synchronized by traditional latches. For example, Shore-MT implements a Cuckoo hash table as the translation layer for its buffer pool [9, 56]. The buffer pool is organized into *buckets*, with the number of buckets set to approximately 1.79× of the configured buffer pool size[2]. This design, combined with the Cuckoo scheme, minimizes collisions.

Problems of traditional design. Despite its advantages, this design introduces significant performance bottlenecks. As illustrated in Figure 3(a), fixing page P2 in Shore-MT involves a sequence of multiple, dependent steps: First, (1) it identifies the corresponding bucket, acquires the bucket latch, and looks up the control block for P2. Next, (2) it dereferences the control block to retrieve the memory address of the corresponding buffer frame. Finally, (3) it still has to dereference the buffer frame address to read the page. This multistep translation process, coupled with an exclusive latch that can introduce delays, results in substantial overhead.

Chaining hash table. A key observation is that, with a sufficiently large hash table, most translations will not conflict. Consequently, if the PID translation typically resolves in the first probed position, the expected CPU cache miss cost is reduced to one, rendering closed-addressing comparable in performance to open-addressing schemes. Closed-addressing, particularly chaining, offers advantages in concurrent environments due to its simplicity: only one latch per key is relevant. In contrast, open-addressing approaches such as Cuckoo hashing or linear probing involve entry relocation. This not only increases implementation complexity but also sometimes requires a single reader to hold many latches simultaneously, thereby introducing contention. Given these considerations, we choose a chaining hash table design for its simplicity and its suitability for highly concurrent workloads without compromising performance.

Inlining first slot. Moreover, since most translations are expected to land in the first slot of the chain, we propose inlining this slot within the translation array to improve cache locality, as illustrated in Figure 3(b). As we will show later in Section 6, simply pre-allocating a large enough chaining hash table with an inlined first slot makes PID translation highly lightweight, laying the foundation for a high-performance buffer pool. We also inline the buffer frame header directly into its hash table entry, eliminating an additional indirection.

Discussion: Why separate buffer frame and frame header. Leis et al. [69] advocate placing the buffer frame header directly before the frame in memory. This inlined layout works well with pointer swizzling, as there is no obvious alternative without introducing another indirection. In contrast, our design separates frame headers and buffer frames: headers are stored in the hash table, while buffer frames reside in a dedicated memory pool. Although the inlined layout might seem applicable to our design, it introduces a major drawback: Frames must be 512-byte aligned to support direct I/O. This leads to substantial memory overhead due to padding – a problem also present in LeanStore. By decoupling headers from

[2]Shore-MT targets a maximum hash table fill factor of 56% as defined in src/sm/bf_core.cpp, line 429 (commit #3f28d8f) [9].

frames, we avoid this problem while achieving optimal performance and resource efficiency.

4.2 Lightweight Synchronization

Pre-allocating a large chaining hash table effectively reduces hash table collisions and eliminates multiple dependent translation steps. However, hot pages may still be accessed concurrently by many threads, leading to contention on the bucket latch—whether implemented as a standard `mutex` or a `reader-writer latch`, since both incur a memory write for each latch operation. Therefore, an efficient and lightweight mechanism is required to synchronize hash table operations between multiple threads. In this section, we first examine the problems with traditional designs and then present our approach to address them.

Bucket latch in traditional designs. Collisions are often handled at the *bucket* granularity, with each bucket protected by a *test-and-set* latch. This design requires all page operations, including page read, to acquire the latch in *exclusive* mode, preventing concurrent threads from accessing the conflicting frame, even for read operations. One potential improvement is to use reader-writer latch; however, as we will demonstrate later in Section 6, this approach still suffers from significant scalability issue – a finding consistent with previous studies [70, 89].

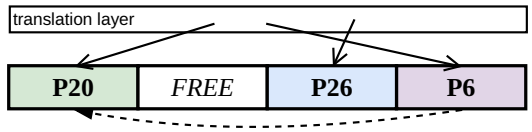
Observation #1: Collision chains are short. As discussed in Section 4.1, by using a large chaining hash table, the fill factor becomes small, leading to a minimal collision ratio. In other words, a majority of chains are short: most of them will comprise only one slot. As shown in Section 6.6, a well-configured setup, e.g., with a maximum fill factor of 75%, ensures that 92% of chain lengths do not exceed two. Therefore, it is enough to have one latch for the entire bucket instead of each element. This is especially true if that latch allows for scalable read operations.

Observation #2: Lock-free enables superscalar execution. Lock-free collision chains, e.g., a lock-free linked list, may seem like the most natural solution. This eliminates strongly serialized instructions like atomic read-modify-write for read operations. As a result, commodity CPUs can leverage *superscalar execution* to overlap the bucket and page accesses, effectively hiding their latency.

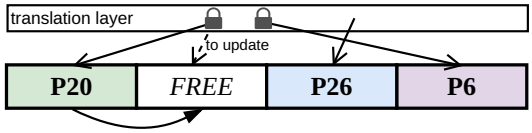
Optimistic latch. Given that collision chains are typically short, conflicts are rare, and modifications are infrequent, we advocate using an *optimistic latch* for efficient chain synchronization. Optimistic latches are often implemented using a version counter that increments whenever the latch transitions out of exclusive mode [22, 71, 77]. Upon latch acquisition, the system (1) retrieves the current latch version, then (2) iterates through the chain to find the corresponding frame, and (3) validates the latch version to detect any concurrent updates or inserts[3].

Summary. By leveraging optimistic latching, the synchronization becomes lightweight and is simple to implement. PID translations are also very efficient as commodity CPUs can internally parallelize two operations – *optimistic latching* and *page access* – using *superscalar execution*. The translation step is then very cheap as chains are short and mainly comprise only one element. When combined with predictive translation described in Section 3, the proposed buffer pool design can interleave hash table translation with page access, effectively achieving high in-memory performance.

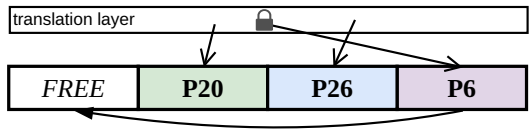
[3]Lock-free accesses also require a memory reclamation strategy, with some lightweight strategies, i.e., trivially affects system operations and performance, including epoch-based reclamation and hazard pointers [53, 65, 74, 78, 95].



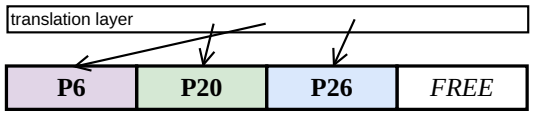
(a) P6 is hot ⇒ DB wants to promote P6 to its preferred position.



(b) Acquire exclusive latches on the P6 and the page currently resides in its preferred frame (i.e., P20). Move P20 to a free frame, update the translation layer, then unlatch P20.



(c) Move P6 to its preferred position, mark previous buffer frame as free, and then unlatch P6.



(d) P6 is in its preferred slot ⇒ fast-path in-mem accesses.

Figure 4: Walk-through example of promotion operation.

5 High-Performance Implementation

We will now summarize how all the conceptual components work together to ensure high performance, using page access operations and synchronization as the means of explanation.

5.1 Buffer Pool Operation

Page caching states. Page caching states – indicating whether a frame is shared/exclusively latched or invalid – are stored alongside the frame header in the optimized hash table described in Section 4. Combined with the version counter required for optimistic latch coupling [70], these page states can be implemented using a 64-bit atomic variable (e.g., `std::atomic<uint64_t>` in C++), enabling efficient state transitions via `compare-and-swap` operations.

Common page operations. Most page operations are similar to those in other buffer pool designs. Specifically, when a page is allocated, the buffer pool determines a free buffer frame from the *free list*, a data structure that maintains all the unused buffer frames of the memory pool [9, 56, 69]. A new frame header entry is then inserted into the translation layer (i.e., hash table Section 4), pointing to the allocated buffer frame. Page accesses, whether read or write, follow the procedure outlined in Listing 2, and the page state in the frame header is updated accordingly (e.g., page updates increase the version counter of the latch). During eviction, the

DBMS first latches the frame in `Exclusive` mode, marks the buffer frame as available for reuse by inserting its pointer back into the *free list*, and then removes the corresponding frame header from the translation layer.

Promotion implementation. The primary implementation difference lies in the promotion mechanism. As explained in Section 3.2, promotion is only considered for pages already resident in the buffer pool. Starting with the second access, the DBMS *probabilistically promotes* a page and potentially demotes the one stored in the associated frame. We adopt a probabilistic approach primarily for its simplicity. Accurately determining the relative *hotness* of pages – such as whether page X is hotter than page Y – would require (1) tracking additional statistics and consuming extra memory and (2) complex coordination with the cache eviction submodule. Instead, probabilistic promotion naturally favors hot pages: since they are accessed more frequently, they are more likely to be promoted into their preferred frames.

Promotion process. The promotion process is illustrated in Figure 4 and proceeds as follows:

(a) Assume `P6` is a hot page eligible for promotion, and its preferred slot is the first buffer frame – currently holding the in-memory content of `P20`.
(b) Transition both `P6` and `P20` to `Exclusive` mode by updating their page states accordingly. After that, a new buffer frame (from the *free list*) is allocated to clone the content of the demoted page `P20`. Update the hash table entry for `P20` in the translation layer to point to this new frame and then release the latch on `P20`.
(c) Copy the content of `P6` into its preferred buffer frame, update the corresponding translation entry in the hash table, and then release `P6`'s exclusive latch.
(d) Finally, subsequent accesses to `P6` can now benefit from *superscalar execution*, as illustrated in Listing 2.

Cost of promotion. Promotion may seem costly because it incurs a full memory copy for a database page. In practice, however, promotion is cheap as the cost of memory copy is amortized over many page accesses: Promotion occurs infrequently – only with a small probability (we chose $\frac{1}{32}$ if the promotion would not lead to a demotion, $\frac{1}{512}$ otherwise). As a result, most hot pages are quickly placed in their preferred positions, after which further copying is rare. For out-of-memory workloads, the overhead of page copies is negligible compared to I/O latency; meanwhile, hot pages (such as B-tree inner nodes) tend to stay in their preferred frames. Although cold pages may incorrectly be promoted, (1) hot pages will soon reclaim those frames, and (2) the occasional cost of incorrect promotion is likewise amortized over time. In short, while full-page copying is inherently expensive, its impact is limited and the performance gains from predictive translation far outweigh the cost. We will further demonstrate this in a microbenchmark in Section 6.7

5.2 Page Operation

Optimistic page access. While the concept of exploiting superscalar execution is straightforward, its actual implementation is more intricate. In Listing 3, we present the pseudocode of how we serve optimistic page reads with predictive translation. First, (1) the predicted frame for the target PID is computed. Then, (2) inside the optimistic read loop [53, 70], (3) we try to execute the *fast path* – assuming PID is in predicted frame to benefit from superscalar execution. This fast path requires three main conditions: the page (1) must currently reside in the buffer pool, (2) not be latched in *exclusive* mode by another thread, and (3) in its predicted frame. If so, the system executes the read callback, which contains the data structure page access logic, on the predicted frame. This is also where the superscalar execution comes into play: The parallel execution of the hash table lookup and the page read is not triggered explicitly anywhere in the code. The introduction of the branch under (3) simply gives the CPU the opportunity to speculatively execute the page read. Note that, as discussed earlier in Listing 2, the CPU's branch predictor also plays a role in this process: most of the hot pages will satisfy the three above conditions; if the prediction is wrong, the page is not hot enough to justify fast-path optimization. In the case that a misprediction happened and the CPU still speculatively executed the branch, there is nothing to be done in the code, as the CPU will transparently flush the pipeline and therefore discard all of the wrongly calculated results. It will then simply wait for the hash table lookup to be ready and use the pointer from there. Finally, if the page is either (4) not yet in the buffer pool and/or (5) not sufficiently hot, we fall back to the slow path by calling the read callback on the translated frame address. In this case, the page may also be probabilistically promoted to its preferred frame.

Listing 3: Optimistic page read.

```
1  void readOptimistic(u64 pid, lambda readCallback):
2    // (1) determine the predicted frame
3    BufferFrame* predictedFrame = predictFrame(pid);
4    while true: // (2) optimistic read loop
5      [FrameHeader* fh, u64 latchState] = ht.find(pid);
6      // (3) fast path requires 3 conditions:
7      if (fh != nullptr // (1) page is in buffer cache
8          && isUnlatched(latchState) // (2) not being latched
9          && fh->isInPredictedFrame): // (3) in pred. frame
10       // superscalar execution can happen after the return
11       // using the predicted frame
12       readCallback(predictedFrame, latchState)
13       // check if version has changed
14       if (fh.latchState == latchState): return
15       else: continue // retry if value changed during read
16     // (4) load page from storage if not in cache
17     if (fh == nullptr):
18       loadPageFromStorage(pid)
19       continue
20     // (5) slow path - use frame pointer from frame header
21     if (isUnlatched(latchState)):
22       readCallback(fh.framePtr, latchState)
23       // Check if version has changed
24       if (fh.latchState == latchState):
25         maybePromote(pid) // probabilistically promote page
26         return
27       else: continue // retry if value changed during read
```

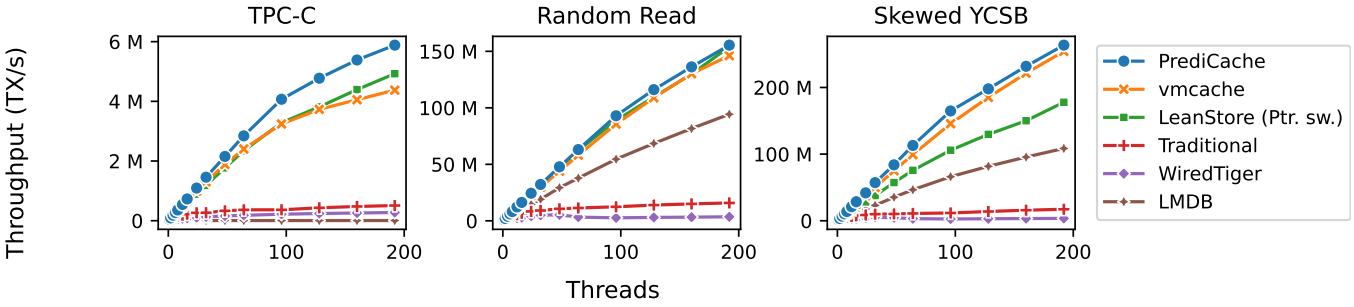


Figure 5: In-memory evaluation.

6 Evaluation

This section presents an empirical evaluation of predictive translation in comparison with several state-of-the-art buffer pools. The evaluation focuses on two primary aspects: (1) showing predictive translation can be at least on par with the best existing approaches under both in-memory and out-of-memory workloads, and (2) providing a qualitative comparison highlighting its design benefits relative to modern alternatives. Through this evaluation, we aim to provide a comprehensive understanding of modern buffer management techniques and demonstrate why predictive translation is superior to all existing designs.

Experimental setup. All performance measurements were performed on a single-socket server with 384GB of RAM and an AMD EPYC 9645P CPU with 96 cores (192 hardware threads). We use Ubuntu with Linux kernel version 6.8. The storage device used in all experiments is a single Kioxia CM7-R PCIe 5.0 NVMe SSD.

Implementation. We implement our buffer pool design from scratch in C++ into a prototype called PrediCache and integrate it with a B-tree that supports variable-sized objects, using optimistic latch coupling to ensure efficient concurrency control. The page size is set to 4KB, which has been discussed as optimal for SSD-optimized DBMSs [48, 49]. The hash function we use, for both the hash table and the predictive translation, is MurmurHash2 [18]. The hash table is sized at twice the buffer pool capacity, resulting in a low expected fill factor for most chains.

Competitors. We compare PrediCache against several state-of-the-art storage engines: (1) LeanStore [12, 69], (2) vmcache [67], integrated with the exmap kernel module for optimal performance, (3) WiredTiger, and (4) LMDB. Both LeanStore and WiredTiger employ the pointer swizzling mechanism for buffer management. The conceptual design of LIPAH is based on pointer swizzling, with a minor extension that allows falling back to hash table translation when the PID is invalid. This fallback mechanism addresses some of the known limitations of pointer swizzling; however, as a result, LIPAH's performance remains upper-bounded by that of pointer swizzling. Hence, we exclude LIPAH from our evaluation.

Baseline implementation. In addition, we implement a baseline hash-table-based buffer pool, denoted as *traditional*, which incorporates several key features of traditional approaches like the Shore-MT buffer manager:

- A read-write spin latch protects each hash table bucket. This offers an upper bound compared to the *test-and-set* latch used in Shore-MT [9], which supports latch acquisition only in *exclusive* mode.
- A large hash table is used to eliminate collisions almost entirely.
- Traditional memory pool, i.e., without predictive translation described in Section 3.

Accessing a buffer frame requires three pointer dereferences (i.e., hash table bucket, buffer frame header, and buffer frame), which is typical of conventional buffer pools. This configuration avoids collisions and ensures that almost every non-empty hash table chain holds exactly one slot. Additionally, it uses optimistic latch coupling for B-tree pages. Together, these features form an upper-bound approximation of traditional buffer pool implementations like Shore-MT [1, 9], which do not utilize optimistic page latches.

Competitor setup. To isolate the conceptual performance differences for a fair comparison of buffer management, we turn off logging and configure all systems to operate under their lowest supported isolation levels. Where applicable, direct I/O is enabled to minimize OS-level caching effects and improve I/O latencies [12, 48, 81]. All systems are configured with their default page size of 4KB, except WiredTiger, which uses its default of 2KB for inner pages and 32KB for leaf pages. For LMDB, we (1) enable the `MDB_NOSYNC` option to mimic the behavior of PrediCache – database pages are only flushed to the storage when necessary, and (2) disable OS-level read-ahead, which helps improve LMDB out-of-memory performance. In the case of LeanStore, we allocate 12 page provider threads for page eviction – the empirically optimal setting in our test environment.

6.1 In-Memory Evaluation

We first evaluate the in-memory performance with three popular workloads: TPC-C, random lookup (i.e., uniform YCSB read-only), and a skewed random lookup (YCSB with $\theta = 0.9$). Note that random lookup is the worst-case workload for our design, as the hit rate of predicted positions will be the lowest possible, since all leaf pages are equally hot/cold. The buffer pool is 256GB in all systems. The number of warehouses used in TPC-C is 200, resulting in a data set of 40GB initially. For random lookup and skewed YCSB, there are 100 M entries, each entry is a pair of 8-byte uniformly-distributed keys and 120-byte values, leading to 20GB of data stored. The experimental results are shown in Figure 5.

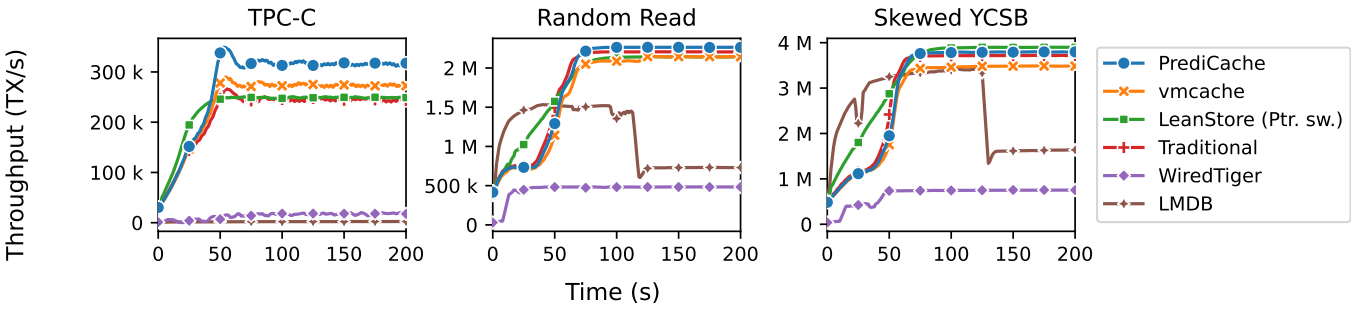


Figure 6: Out-of-memory evaluation.

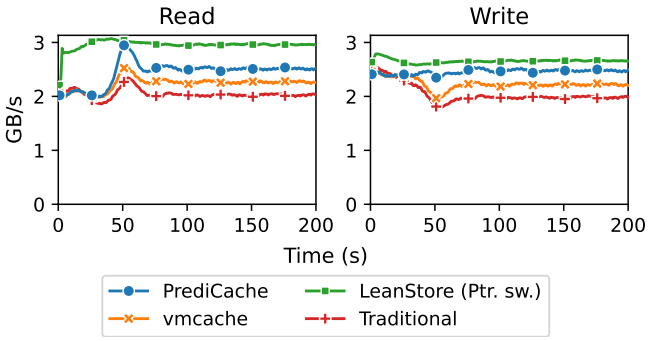


Figure 7: IO rates in the TPC-C out-of-memory workload

Conventional competitors. WiredTiger and *traditional* both perform poorly all three workloads. In the case of the *traditional* competitor, its performance suffers due to the use of a read-write spin latch combined with multiple levels of translation dependencies (see Section 2.4). WiredTiger, despite employing pointer swizzling, also shows weak performance – primarily because of its isolation level enforcement. Specifically, for general read-write workloads, WiredTiger only supports isolation levels not lower than *read committed*; *read uncommitted* can only be applied for read-only transactions, in which scenario WiredTiger will perform comparably with modern designs as illustrated in [12]. LMDB, meanwhile, scales poorly on TPC-C due to its single-writer architecture.

Our design vs. vmcache and pointer swizzling. These three buffer cache designs perform similarly in random lookup. The skewed YCSB workload shows both vmcache and PrediCache outperforming LeanStore (which implements pointer swizzling). For TPC-C, PrediCache substantially outperforms vmcache and LeanStore – 34% and 19% at 192 threads, respectively. The largest performance gaps occur in the TPC-C benchmark, where the closest competitor to PrediCache is vmcache.

Detailed comparison with vmcache. PrediCache outperforms vmcache for three reasons. First, the TPC-C workload frequently allocates new pages, which triggers exmap system calls in vmcache and incur more kernel cycles compared to PrediCache. Second, vmcache faces a scalability problem due to its page state array, implemented as a list of `atomic<u64>`. In this design, eight page states share a single cache line, causing cache-line ping-pong when adjacent page states are updated and/or accessed concurrently – which happens frequently in TPC-C workloads (i.e., the *neworder* problem [14])[4]. In contrast, PrediCache avoids this entirely – every entry in the hash table is 40B, meaning each cache line references at most two page states. It can afford larger entries because it only needs metadata for pages currently in the buffer pool. Lastly, PrediCache's ability to exploit huge pages – which is difficult in vmcache design because DBMS page size in vmcache must align with the page size of the OS virtual memory subsystem – further enable faster page accesses; disabling huge pages in PrediCache reduces TPC-C throughput by about 400k TX/s, roughly 7% throughput.

Detailed comparison with pointer swizzling. Our design surpasses LeanStore by employing a more efficient and compact B-tree structure. This advantage stems from our ability to support multiple page references – a capability that LeanStore lacks, as explained in Section 2.1. Although LeanStore's B-tree could theoretically be optimized to achieve similar performance, its inherent qualitative limitations make such improvements considerably more difficult than with vmcache and our buffer pool [13, 24].

6.2 Out-Of-Memory Evaluation

Let us now move the focus to data sets that are larger than the allocated buffer pool. In this benchmark, all systems employ a buffer manager of 128GB and use 192 worker threads[5]. We also use the same three workloads, TPC-C, random lookup, and skewed YCSB, with different configurations to ensure the data is larger than the buffer pool. Specifically, we use 5000 warehouses for TPC-C and 5 billion key-value pairs for random lookup and YCSB; all are roughly 1TB in size. We show the experimental results in Figure 6.

WiredTiger and LMDB. As expected, WiredTiger performs poorly under out-of-memory (OOM) workloads because of the combined overhead of isolation checks and I/O operations. LMDB, on the other hand, shows strong performance on random lookup and skewed YCSB workloads during the first 50 seconds, primarily because (1) OS read-ahead is disabled, improving its OOM behavior, and (2) it relies on the OS page cache, which typically does not strictly limit memory usage. Once the OS page cache is full and the kernel needs to start doing eviction, the performance drops significantly – an observation aligning with previous findings in [27].

Pointer swizzling. LeanStore (with pointer swizzling) slightly outperforms other modern designs during the first 50 seconds of the

[4]While padding could mitigate this issue, doing so would exaggerate the memory overhead issue further (cf., Section 2.1).

[5]LMDB uses the OS page cache and will therefore use more of the available RAM.

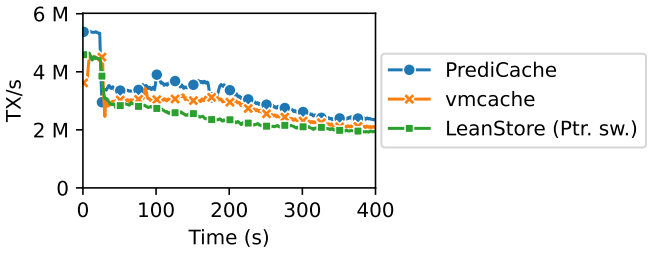


Figure 8: TPC-C Workload: Performance during transition from in-memory to out-of-memory.

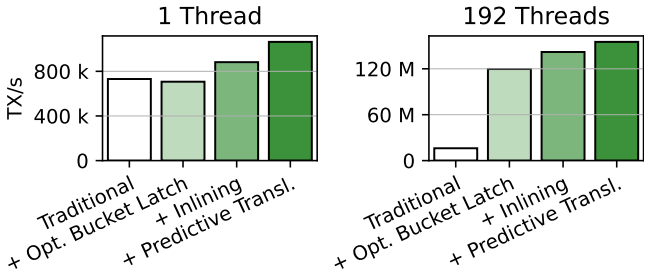


Figure 9: Impact of different techniques on buffer pool performance.

experiment, primarily due to its background page provider threads. Specifically, at the beginning of the experiment, most pages residing in the buffer pool are dirty, so having dedicated background threads for evicting these pages helps boost performance. However, LeanStore eventually trails behind other SSD-optimized designs due to its inefficient B-tree layout – stemming from the qualitative limitations discussed in Section 2.4, even though it invokes I/O operations at a higher rate than the other systems, as shown in Figure 7.

Hash-table-based designs vs. vmcache. Compared to vmcache, our design delivers better performance, mainly because our design is computationally more efficient, as discussed earlier in Section 6.1, allowing it to complete more transactions and deliver superior overall performance. As Figure 6 illustrates, *traditional* achieves performance comparable to SSD-optimized designs in out-of-memory workloads. This is because I/O is the biggest overhead in out-of-memory workloads; hence, the in-memory overheads of the traditional hash table are negligible.

6.3 Changing Workload Evaluation

In addition to evaluating purely in-memory and purely out-of-memory workloads, we also consider scenarios where the dataset gradually grows beyond the buffer pool capacity. To simulate this, we run TPC-C with an initial configuration of 400 warehouses and a fixed buffer pool size of 128GB across all systems. At the beginning of the experiment, the buffer pool is approximately half full and quickly fills up due to the high write rates of the TPC-C workload. We focus our evaluation on our design, vmcache, and LeanStore, as other systems performed significantly worse in either in-memory and/or out-of-memory settings compared to these three modern approaches.

Figure 8 shows that our design outperforms both vmcache and LeanStore during the transition from in-memory to out-of-memory operation. While LeanStore exhibits a smoother transition – thanks to its background page eviction threads – it ultimately falls behind the other two, consistent with the results in Section 6.2. Our buffer manager achieves higher performance throughout the entire experiment; by the end, it delivers 19.7% and 24.6% higher throughput than vmcache and LeanStore, respectively.

6.4 Ablation Study

To better understand the impact of our conceptual building blocks, we now analyze how each optimization affects the performance. We start with the *traditional* buffer cache as the baseline. Step by step, we add additional features:

(1) Leveraging optimistic bucket latches to enable lock-free hash table translations in Section 4.2.
(2) Inlining the first slot directly within the translation array of the hash table, c.f., Section 4.1.
(3) Predictive translation to exploit superscalar execution in Section 3.

For this experiment, we use the same in-memory random lookup as described in Section 6.1. We show the experimental results in Figure 9.

Hash table optimizations. As shown in Figure 9, switching from read-write spin latching to optimistic latching yields a more scalable hash table design for the translation layer. Specifically, this change allows lookup throughput to scale with thread-count again. For 192 threads this means an increase by 7.4×. However, it has very little effect at low thread-counts – actually reducing performance slightly – in which scenario Predictive Translation shows the biggest improvement. The result highlights that optimistic latching, despite its simplicity and robustness, is a necessary foundation for building scalable buffer managers – as also discussed in the current literature [12, 66, 67, 69, 70, 89]. Furthermore, inlining the first hash table slot directly into the translation array (see Section 4.1) enhances system throughput by 24.8% with a single thread and by 18.6% at 192 threads. This improvement stems from the low fill factor, where most hash table chains contain only one translation slot (as will be discussed in Section 6.6); hence, inlining the first slot considerably reduces cache misses and enhances performance.

Predictive translation to hide translation overheads. Predictive translation enables the buffer pool to exploit *superscalar execution*, effectively hiding the translation overheads and improving performance. As being shown, the integration with predictive translation delivers a substantial 20.6% throughput improvement over the variant without predictive translation in the single-threaded benchmark, demonstrating its effectiveness.

6.5 Microarchitectural Analysis

To assess whether PrediCache effectively exploits the *superscalar execution* capabilities of commodity CPUs, we perform a microarchitectural analysis using hardware performance counters from multiple systems under a single-threaded in-memory uniform random lookup workload. LeanStore (pointer swizzling) is excluded

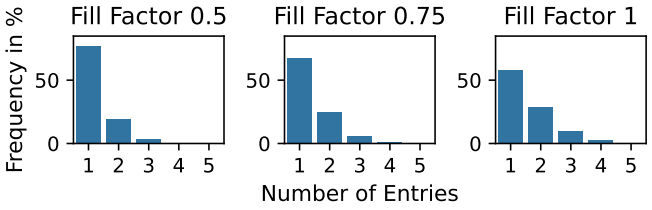


Figure 10: Distribution of hash table chain lengths (excluding empty buckets) at different fill factors.

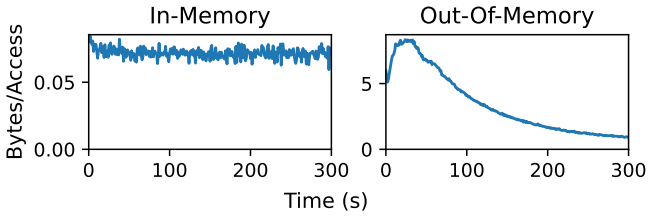


Figure 11: Random Read Workload: Number of copied (promoted) bytes per access

because (1) it uses a fundamentally different B-tree layout, and (2) it integrates additional features like logging and snapshot isolation [12], introducing significant conceptual differences that prevent fair comparison. The most important hardware performance counters are shown in the following table:

system	cycles	inst.	IPC	L2 miss	dTLB miss	br. miss	f.e.[6] stall
PrediCache	3510	1915	0.55	28	3.5	11	131
vmcache	3939	1315	0.33	31	5.1	11	150
Traditional	5090	1574	0.31	34	3.7	11	155

Other potential factors. L2 misses and branch misses are similar across all systems, so they cannot explain the performance differences. L1 misses are similar for vmcache and Traditional (46-48) and lower for PrediCache (35). While the reason for this is unclear it should only have a moderate effect. L3 misses were analyzed but excluded as they can only be collected per CCX on modern AMD CPUs, making them noisy and incomparable to L1/L2 misses, though they occur in similar numbers as L2 misses across all systems. iTLB misses were ~0 for all systems. dTLB misses are higher for vmcache due to its use of 4kB virtual memory pages versus the 2MB huge pages used by the other systems. Kernel cycles are negligible for all variants in this workload.

Effect of superscalar execution. PrediCache achieves significantly higher instructions per cycle (IPC) than vmcache while also having fewer frontend stalls than both vmcache and the traditional variant. While our buffer pool executes more instructions than vmcache due to additional hash table translations, it effectively hides this overhead, resulting in comparable CPU cycles. These metrics, together with the fact that none of the others fully account for the performance difference discussed above, suggest that the improvements stem from better utilization of superscalar execution. In contrast, *traditional* fails to exploit superscalar execution due to data dependencies (discussed in Section 2.4), leading to substantially higher cycle counts.

6.6 Analysis of Hash Table Chain Lengths

As discussed earlier in Section 4.2, one of the key assumptions that our buffer pool design relies on is that the hash table chains are mostly short. In this experiment, we empirically evaluate the chain length distribution by generating random numbers and counting how many go into each bucket with different fill factors, i.e., different hash table sizes, to verify the original assumption.

[6]Frontend stalls occur when a CPU's fetch and decode stages are delayed, often due to cache misses or branch mispredictions, reducing superscalar execution efficiency.

Chain length distribution. As Figure 10 illustrates, even with a fill factor of 1, 97% of the chains are not longer than three. With a reasonable hash table size, e.g., twice the size of the buffer pool, which leads to an expected fill factor of 50%, 77% of the chain lengths are one. These observations align with prior findings that chain lengths follow a zero-truncated binomial distribution [41].

Result. In summary, by inlining the first element and using optimistic latching for hash table synchronization, the translation layer remains both lightweight and lock-free, enabling *superscalar execution* to effectively hide software-level translation overhead.

6.7 Analysis of Deep Copies during Promotion

As discussed in Section 5.1, the cost of page copies during page promotion should be minimal, especially for out-of-memory workloads. And for all three in-memory benchmarks (i.e., random read, skewed YCSB, and TPC-C), we observe an amortized cost of less than 1B copied per page access. For example, Figure 11 illustrates the overhead of deep copies in the random read scenario. In the in-memory case, shown in Figure 11 (left), the copy rate remains relatively constant and low. In the out-of-memory case, illustrated in Figure 11 (right), the copy rate is initially higher as hotter pages (e.g., inner B+Tree nodes) are promoted, but it eventually stabilizes at a low value. Similar patterns and numbers can be observed for both skewed YCSB and TPC-C.

6.8 Analysis of Pages in Preferred Position

In this microbenchmark, we evaluate two metrics: the proportion of pages placed in their preferred frames and the proportion of successful accesses to predicted frames – excluding accesses that cause page faults. The workload used is a uniform random read workload on a 128GB buffer pool, yielding the following results:

data size	pages in preferred pos.	accesses to predicted frames
0.5× buffer pool	75%	96%
3× buffer pool	3%	91%

As depicted in the above table, in in-memory scenarios, e.g., the dataset size is half of the buffer pool, most pages will be in their preferred frames, explaining the high in-memory performance of PrediCache. Even in out-of-memory settings, such as when the dataset is 3× of the buffer pool, 91% of accesses still successfully exploit predictive translation (e.g., for B-Tree inner pages).

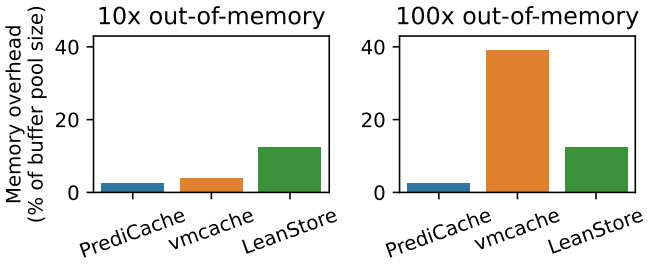


Figure 12: Estimated memory overhead for buffer management metadata.

6.9 Design Goal Validation

As discussed in Section 2.1, previous high-performance buffer caches fail to satisfy the *design goals* discussed in Section 2.2, hindering their widespread adoption. In this discussion, we systematically evaluate how well our proposed design satisfies each *goal* in order to evaluate its practicality:

(1) *High performance*: Throughout Section 6, we have experimentally demonstrated that our proposed solution outperforms modern buffer pools in both in-memory and out-of-memory workloads.
(2) *Low memory overhead*: Figure 12 shows the estimated memory overhead across varying dataset-to-buffer-pool ratios. This figure highlights the problem of vmcache in this regard, with an overhead of around 39% of the buffer pool size at 100x out-of-memory. In contrast, in our approach, per-page metadata is embedded within the frame headers, which are managed by the hash table. As a result, the total metadata size scales proportionally with the buffer pool size. Moreover, this advantage also enables seamless integration with state-of-the-art sampling-based eviction strategies, such as WATT [94] and HyperBolic [21].
(3) *Portability*: Our buffer pool provides APIs similar to those of commercial DBMSs and requires no modifications to the OS kernel or other DBMS subsystems.
(4) *Multiple page references*: The design natively supports multiple and cyclic page references.
(5) *Unlimited number of pages*: Using 8-byte PIDs, our buffer pool supports a (practically) unlimited number of pages.

Thus, our design successfully meets all the design goals defined in Section 2.2. We believe this work serves as a crucial stepping stone for DBMSs seeking high-performance caching solutions.

7 Related Work

In Section 2, we already introduced some of the related work we compare our design against. In this section, we will also include some other relevant prior work.

Data processing on novel hardware. Enhancing the performance of database systems often requires rethinking and adapting traditional techniques to better align with modern hardware – an objective our proposal also embraces. Some research efforts advocate for tight hardware-software co-design, as seen in works such as Alonso et al. [15], Chiosa et al. [26]. A significant body of work focuses on leveraging specialized hardware to boost data processing performance, particularly FPGA-based accelerators [20, 28, 29, 36, 55, 58, 90]. In parallel, numerous studies explore how to better exploit hardware; examples include (but not limited to) vmcache [67], HyPer [62], and Umbra [80], virtual memory-assisted indexing [88], and a recent effort to optimize graph processing on commodity systems [50].

DB/OS co-design. The notorious frenemy relationship between database systems and operating systems has long been recognized by both communities [27, 31, 40, 75, 91], prompting a range of research efforts aimed at addressing the tension. For instance, Giceva et al. [40] argue for "opening up the OS" through new declarative interfaces between the DBMS and the operating system. Building on this idea, Leis and Dietrich [68] propose fusing the database and OS into a single unikernel – an approach they suggest is especially suitable for cloud deployments. Similarly, Zhou et al. [101] present a practical DB/OS co-design that places the database process directly in the kernel space of a lightweight guest OS. In contrast, our design is based on commodity hardware and should be available on most platforms. Therefore, we believe that the ideas presented in this paper are complementary to all the DB/OS co-design projects.

Perfect hashing. Perfect hashing is an interesting approach for completely eliminating hash collisions over a known set of keys, with some prior works having explored its potential in database systems [39, 76]. The main disadvantage of perfect hashing is that it typically requires a static predefined set of keys. In the context of buffer management, PIDs can theoretically span the full 64-bit range, making the key space effectively unbounded and dynamic. Therefore, we argue that perfect hashing is not a good fit for buffer management.

Variable-sized objects. Modern applications increasingly rely on rich, contextual data, driving up the need to store large and complex objects in database systems. Use cases such as surveillance and medical diagnostics demand specialized video databases for efficient video analysis and management [96, 97]. Similarly, AI and search applications rely heavily on vector embeddings, which require database systems to store and index high-dimensional data effectively [25, 83, 84, 100]. Additionally, analytical workloads also benefit from columnar formats [6, 7, 11, 63, 87, 99], which also depend on efficient management of variable-sized objects within the DBMS. These requirements underscore the growing importance of efficient large object management within database systems. Achieving this challenge requires rethinking many aspects of database system design, as analyzed in [82], with support for caching variable-size pages identified as a key optimization. Our design currently lacks native support for variable-size pages; however, since this limitation is orthogonal to transaction processing, we consider variable-size page caching out of scope for the current work. Nevertheless, extending predictive translation to support variable-size pages remains a promising research direction for future work. One possible starting point is to adopt a tiered buffer pool design – each tier supports different page sizes, similar to that of Umbra [80] and Bf-Tree [51].

Cost-optimal cloud-native buffer manager. Buffer management in cloud-native environments poses different requirements than in single-node DBMSs, especially regarding cost-efficiency. One possible starting point is to re-evaluate the classical five-minute

rule [19, 42, 46] and adapt it for cloud-native buffer caches. Duwe et al. [33] provide a preliminary cost model, but how it maps to real designs and the trade-offs between cost and performance remain open questions, which we plan to explore in future work.

8 Summary

In summary, this paper overturns the long-standing belief that hash-table-based buffer pools must trade simplicity for performance. By integrating an optimized chaining hash table with predictive translation, our design successfully exploits superscalar execution to interleave hash-table translation and page access, thereby hiding the expensive translations. At the same time, our buffer pool preserves all the key qualitative advantages associated with hash-table-based designs: It provides a portable, flexible, and self-contained design with the traditional API and low memory overhead. Predictive translation also allows for multiple and cyclic page references, unbounded PIDs, and page-metadata storage. Our performance evaluation demonstrates that predictive translation can consistently perform at least on par with other modern designs. Given its combination of qualitative benefits and strong performance, we consider it to be the best approach for buffer management in modern, storage-optimized DBMSs.

Acknowledgments

We thank the reviewers for their thoughtful feedback. We are also grateful to Lukas Michael Englhauser and Maximilian Kuschewski for their initial support in exploring this research.

References

[1] 2010. Shore-MT: Implementation Notes. https://pages.cs.wisc.edu/~nhall/shore-mt/onlinedoc/implnotes.html.
[2] 2012. Supervisor mode access prevention. https://lwn.net/Articles/517475/.
[3] 2023. MySQL source code. https://github.com/mysql/mysql-server/tree/mysql-cluster-8.0.33.
[4] 2023. PostgreSQL source code. https://github.com/postgres/postgres/tree/REL_15_3.
[5] 2024. SQLite source code. https://github.com/sqlite/sqlite.
[6] 2025. Apache Iceberg. https://iceberg.apache.org/.
[7] 2025. Apache Parquet. https://parquet.apache.org/.
[8] 2025. KIOXIA Enterprise SSD. https://europe.kioxia.com/en-europe/business/ssd/enterprise-ssd.html.
[9] 2025. Shore-MT. https://github.com/epfl-dias/shore-mt.
[10] 2025. SOLIDIGM Enterprise SSD. https://www.solidigm.com/products/data-center.html.
[11] Azim Afroozeh and Peter A. Boncz. 2023. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (2023), 2132–2144.
[12] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *BTW (LNI, Vol. P-331)*. Gesellschaft für Informatik e.V., 259–281.
[13] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *CIDR*. www.cidrdb.org.
[14] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 6 (2023), 1426–1438.
[15] Gustavo Alonso, Timothy Roscoe, David A. Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. 2020. Tackling Hardware/Software co-design from a database perspective. In *CIDR*. www.cidrdb.org.
[16] Mijin An, Soojun Im, Dawoon Jung, and Sang Won Lee. 2022. Your Read is Our Priority in Flash Storage. *Proc. VLDB Endow.* 15, 9 (2022), 1911–1923.
[17] Mijin An, Jonghyeok Park, Tianzheng Wang, Beomseok Nam, and Sang-Won Lee. 2023. NV-SQL: Boosting OLTP Performance with Non-Volatile DIMMs. *Proc. VLDB Endow.* 16, 6 (2023), 1453–1465.
[18] Austin Appleby. 2016. *MurmurHash2*. https://github.com/aappleby/smhasher/wiki/MurmurHash2/ee3f8e837425b51e0118f768c6c3df082be0700f
[19] Raja Appuswamy, Goetz Graefe, Renata Borovica-Gajic, and Anastasia Ailamaki. 2019. The five-minute rule 30 years later and its impact on the storage hierarchy. *Commun. ACM* 62, 11 (2019), 114–120.
[20] Maximilian Bandle and Jana Giceva. 2021. Database Technology for the Masses: Sub-Operators as First-Class Entities. *Proc. VLDB Endow.* 14, 11 (2021), 2483–2490.
[21] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *USENIX ATC*. USENIX Association, 499–511.
[22] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *DaMoN*. ACM, 2:1–2:8.
[23] Matthew Butrovich, Samuel Arch, Wan Shen Lim, William Zhang, Jignesh M. Patel, and Andrew Pavlo. 2025. BPF-DB: A Kernel-Embedded Transactional Database Management System For eBPF Applications. *Proc. ACM Manag. Data* 3, 3 (2025), 135:1–135:27.
[24] Riki Otaki Jun Hyuk Chang, Charles Benello, Aaron J. Elmore, and Goetz Graefe. 2025. Resource-Adaptive Query Execution with Paged Memory Management. In *CIDR*. www.cidrdb.org.
[25] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proc. VLDB Endow.* 17, 12 (2024), 3772–3785.
[26] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. 2022. Hardware Acceleration of Compression and Encryption in SAP HANA. *Proc. VLDB Endow.* 15, 12 (2022), 3277–3291.
[27] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR*. www.cidrdb.org.
[28] Jonas Dann, Daniel Ritter, and Holger Fröning. 2023. Non-relational Databases on FPGAs: Survey, Design Decisions, Challenges. *ACM Comput. Surv.* 55, 11 (2023), 225:1–225:37.
[29] Jonas Dann, Royden Wagner, Daniel Ritter, Christian Faerber, and Holger Fröning. 2022. PipeJSON: Parsing JSON at Line Speed on FPGAs. In *DaMoN*. ACM, 3:1–3:7.
[30] Justin A. DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (2013), 1942–1953.
[31] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. 1990. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.* 2, 1 (1990), 44–62.
[32] Jörg Domaschka, Simon Volpert, Kevin Maier, Georg Eisenhart, and Daniel Seybold. 2023. Using eBPF for Database Workload Tracing: An Explorative Study. In *ICPE (Companion)*. ACM, 311–317.
[33] Kira Duwe, Angelos Anadiotis, Andrew Lamb, Lucas Lersch, Boaz Leskes, Daniel Ritter, and Pınar Tözün. 2025. The Five-Minute Rule for the Cloud: Caching in Analytics Systems. In *Conference on Innovative Data Systems Research*. Conference on Innovative Data Systems Research.
[34] Wolfgang Effelsberg and Theo Härder. 1984. Principles of Database Buffer Management. *ACM Trans. Database Syst.* 9, 4 (1984), 560–595.
[35] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Found. Trends Databases* 8, 1-2 (2017), 1–130.
[36] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. 2020. In-memory database acceleration on FPGAs: a survey. *VLDB J.* 29, 1 (2020), 33–59.
[37] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.
[38] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. 2005. Space Efficient Hash Tables with Worst Case Constant Access Time. *Theory Comput. Syst.* 38, 2 (2005), 229–248.
[39] Kevin P. Gaffney and Jignesh M. Patel. 2024. Is Perfect Hashing Practical for OLAP Systems?. In *CIDR*. www.cidrdb.org.
[40] Jana Giceva, Tudor-Ioan Salomie, Adrian Schüpbach, Gustavo Alonso, and Timothy Roscoe. 2013. COD: Database / Operating System Co-Design. In *CIDR*. www.cidrdb.org.
[41] Gaston H. Gonnet. 1981. Expected Length of the Longest Probe Sequence in Hash Code Searching. *J. ACM* 28, 2 (1981), 289–304.
[42] Goetz Graefe. 2008. The Five-Minute Rule 20 Years Later: and How Flash Memory Changes the Rules. *ACM Queue* 6, 4 (2008), 40–52.
[43] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *PVLDB* 8, 1 (2014), 37–48.
[44] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph A. Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *Proc. VLDB Endow.* 8, 1 (2014), 37–48.
[45] Torbjörn Granlund and Peter L. Montgomery. 1994. Division by Invariant Integers using Multiplication. In *PLDI*. ACM, 61–72.
[46] Jim Gray and Gianfranco R. Putzolu. 1987. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *SIGMOD Conference*. ACM Press, 395–398.

[47] Gabriel Haas, Adnan Alhomssi, and Viktor Leis. 2025. Managing Very Large Datasets on Directly Attached NVMe Arrays. In *Scalable Data Management for Future Hardware*. Springer, 223–240.

[48] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*. www.cidrdb.org.

[49] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (2023), 2090–2102.

[50] Alireza Haddadi, David Black-Schaffer, and Chang Hyun Park. 2023. Large-scale Graph Processing on Commodity Systems: Understanding and Mitigating the Impact of Swapping. In *MEMSYS*. ACM, 2:1–2:11.

[51] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. *Proc. VLDB Endow.* 17, 11 (2024), 3442–3455.

[52] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *SIGMOD Conference*. ACM, 981–992.

[53] Michael Haubenschild and Viktor Leis. 2023. Lock-Free Buffer Managers Do Not Require Delayed Memory Reclamation. In *SiMoD@SIGMOD*. ACM, 1:1–1:3.

[54] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. 2022. SSDs Striking Back: The Storage Jungle and Its Implications to Persistent Indexes. In *CIDR*. www.cidrdb.org.

[55] Zsolt István. 2020. Let's add transactions to FPGA-based key-value stores!. In *DaMoN*. ACM, 13:1–13:3.

[56] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. In *EDBT (ACM International Conference Proceeding Series, Vol. 360)*. ACM, 24–35.

[57] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499.

[58] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-based Data Partitioning. In *SIGMOD Conference*. ACM, 433–445.

[59] Antonios Katsarakis, Vasilis Gavrielatos, and Nikos Ntarmos. 2024. DLHT: A Non-blocking Resizable Hashtable with Fast Deletes and Memory-awareness. In *HPDC*. ACM, 186–199.

[60] Alfons Kemper and Donald Kossmann. 1993. Adaptable Pointer Swizzling Strategies in Object Bases. In *ICDE*. IEEE Computer Society, 155–162.

[61] Alfons Kemper and Donald Kossmann. 1995. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB J.* 4, 3 (1995), 519–566.

[62] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[63] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26.

[64] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (1981), 650–670.

[65] Viktor Leis. 2016. *Query Processing and Optimization in Modern Database Systems*. Ph. D. Dissertation. Technical University Munich, Germany.

[66] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proc. VLDB Endow.* 17 (2024), 4536–4545.

[67] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25.

[68] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *Proc. VLDB Endow.* 17, 8 (2024), 2115–2122.

[69] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. IEEE Computer Society, 185–196.

[70] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[71] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*. ACM, 3:1–3:8.

[72] Scott T. Leutenegger and Daniel M. Dias. 1993. A Modeling Study of the TPC-C Benchmark. In *SIGMOD Conference*. ACM Press, 22–31.

[73] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. 2013. Identifying hot and cold data in main-memory databases. In *ICDE*. IEEE Computer Society, 26–37.

[74] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. IEEE Computer Society, 302–313.

[75] Qian Li, Peter Kraft, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Jason Li, Michael J. Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2022. A Progress Report on DBOS: A Database-oriented Operating System. In *CIDR*. www.cidrdb.org.

[76] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. 1996. A Family of Perfect Hashing Methods. *Comput. J.* 39, 6 (1996), 547–554.

[77] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. ACM, 183–196.

[78] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distributed Syst.* 15, 6 (2004), 491–504.

[79] Marcus Müller, Lawrence Benson, and Viktor Leis. 2025. B-Trees Are Back: Engineering Fast and Pageable Node Layouts. *Proc. ACM Manag. Data* (2025).

[80] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[81] Lam-Duy Nguyen, Adnan Alhomssi, Tobias Ziegler, and Viktor Leis. 2025. Moving on From Group Commit: Autonomous Commit Enables High Throughput and Low Latency on NVMe SSDs. *Proc. ACM Manag. Data* 3.

[82] Lam-Duy Nguyen and Viktor Leis. 2024. Why Files If You Have a DBMS?. In *ICDE*. IEEE, 3878–3892.

[83] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *The VLDB Journal* 33, 5 (2024), 1591–1615.

[84] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *SIGMOD Conference Companion*. ACM, 597–604.

[85] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *ICDE*. IEEE, 1326–1339.

[86] Andrew Pavlo. 2014. *On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems*. Ph. D. Dissertation. Brown University, USA.

[87] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. *Proc. VLDB Endow.* 17, 11 (2024), 3290–3303.

[88] Felix Martin Schuhknecht. 2024. Taking the Shortcut: Actively Incorporating the Virtual Memory Index of the OS to Hardware-Accelerate Database Indexing. In *CIDR*. www.cidrdb.org.

[89] Ge Shi, Ziyi Yan, and Tianzheng Wang. 2023. OptiQL: Robust Optimistic Locking for Memory-Optimized Indexes. *Proc. ACM Manag. Data* 1, 3 (2023), 216:1–216:26.

[90] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *SIGMOD Conference*. ACM, 403–415.

[91] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael J. Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2021. DBOS: A DBMS-oriented Operating System. *Proc. VLDB Endow.* 15, 1 (2021), 21–30.

[92] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2019. The end of an architectural era: it's time for a complete rewrite. In *Making Databases Work*. ACM Books, Vol. 22. ACM / Morgan & Claypool, 463–489.

[93] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.

[94] Demian E. Vöhringer and Viktor Leis. 2023. Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware. *Proc. VLDB Endow.* 16, 11 (2023), 3323–3334.

[95] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD Conference*. ACM, 473–488.

[96] Renzhi Wu, Pramod Chunduri, Ali Payani, Xu Chu, Joy Arulraj, and Kexin Rong. 2024. SketchQL: Video Moment Querying with a Visual Query Interface. *Proc. ACM Manag. Data* 2, 4 (2024), 204:1–204:27.

[97] Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *SIGMOD Conference*. ACM, 602–616.

[98] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. *Proc. VLDB Endow.* 16, 1 (2022), 99–112.

[99] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *CoRR* abs/2304.05028 (2023).

[100] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *OSDI*. USENIX Association, 377–395.

[101] Xinjing Zhou, Viktor Leis, Jinming Hu, Xiangyao Yu, and Michael Stonebraker. 2025. Practical DB-OS Co-Design with Privileged Kernel Bypass. *Proc. ACM Manag. Data* 3, 1 (2025), 64:1–64:27.

[102] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD Conference*. ACM, 685–699.