

# HeapLENS: Heap Layout Evaluation & Navigation Suite

Trevor Brown  
University of Waterloo  
Waterloo, Canada

Sean Ovens  
University of Waterloo  
Waterloo, Canada

Gautam Pathak  
University of Waterloo  
Waterloo, Canada

## Abstract

We present HeapLENS, a visualization and profiling tool for spotting memory-layout anti-patterns in multithreaded C/C++ applications. It is built to investigate questions such as: Which object types routinely lie next to tree nodes? Which fields cross cache lines? How do layouts change over time? HeapLENS automatically collects allocation histories with best-effort type information and support for custom user allocators. To make histories with billions of events inspectable, it uses sampling and a novel page fingerprinting technique to cluster and visualize a small set of diverse layouts.

We use HeapLENS to identify layout problems in concurrent data structures and database applications, contributing an accepted RocksDB fix, and improving concurrent data structure performance by up to 30.6%. We also give HeapLENS's text exports to an LLM, which independently investigates layout problems and tests optimizations, improving mean throughput by 4.2% in the Valkey KV-store and approximate nearest-neighbour throughput by up to 9.8% in the HNSWLib vector database.

**CCS Concepts:** • **Software and its engineering** → **Software performance**; • **Human-centered computing** → **Visualization systems and tools**.

**Keywords:** memory layout, heap visualization, memory profiling, cache locality, performance debugging, concurrency

## ACM Reference Format:

Trevor Brown, Sean Ovens, and Gautam Pathak. 2026. HeapLENS: Heap Layout Evaluation & Navigation Suite. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026)*, November 15-18, 2026, Hong Kong, China. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3828523.3857046>

## 1 Introduction

Reasoning about the performance of multithreaded applications requires more than understanding their algorithms. Threads interact through hardware caches and shared memory, and seemingly minor implementation choices can have substantial effects on execution time. This helps explain the gap between a data structure's theoretical and observed

performance. Memory layout—where an application places objects in its address space—is one such influence. A programmer chooses an object's fields and allocation size, but its eventual neighbours and alignment also depend on the allocator, other allocations, and the interleaving of threads.

Some basic layout effects are well known. Scattering a data structure's nodes across more pages than necessary can increase TLB misses and lead to expensive page-table walks. Accessing data sequentially can also make better use of prefetching than following pointers between scattered objects. A more subtle issue is *cache set underutilization*: regular patterns in object addresses can prevent important data types from occupying some cache sets, leaving part of the cache unavailable to those types. This has been shown to cause substantial performance degradation in practice. For example, Afek et al. [1] developed a cache index-aware memory allocator that tackled this problem directly, *doubling performance* on key benchmarks compared to popular memory allocators like jemalloc [17] and tcmalloc [19].

Arbel-Raviv et al. [4] also noted the effects of cache set underutilization on concurrent binary search tree implementations. They additionally studied many other memory layout effects such as field scattering, where search-critical fields of an object (like the left and right child pointers in a binary search tree node) are split across multiple cache lines. Fixing such memory layout issues was shown to nearly double the performance of an in-memory database index. Their investigation required highly manual application-specific collection of allocation types, reconstruction of page and cache-line layouts, and computation of cache-set statistics. It is not realistic to expect practitioners and researchers to repeat this effort to discover memory layout problems.

Existing tools can help with some small parts of this. Allocation profiles can identify allocation locations in the source code responsible for high memory consumption [17, 28]. Object-reference views help explain how objects are connected [2, 37]. Allocation visualizers such as MemoView [27] and AllocRay [33] show addresses and lifetimes, but do not identify C/C++ object types. To investigate a layout problem, a developer may need to know which types share a cache line, where fields and padding fall within it, and whether the same layout recurs across many pages. Obtaining a manageable view of those relationships remains difficult.

We present *HeapLENS*, a profiling tool for spotting memory layout anti-patterns in multithreaded C/C++ applications. It automatically collects typed allocation logs and provides browser-based linked views of pages, cache lines, object



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11

<https://doi.org/10.1145/3828523.3857046>

fields, and cache-set occupancy. Users can move forwards and backwards through execution time to examine how a layout develops. These observations help them discover unexpected layouts, relate them to application behaviour, and choose optimizations to test. HeapLENS is allocator agnostic, which is important because a layout visualization tool that requires a custom allocator will generally change memory layouts and conceal problems under investigation.

Making memory layouts inspectable requires more than displaying logged allocations. A log can contain billions of events, and a person cannot review every page. Sampling can help, but the details matter. Sampling individual objects can remove the neighbours that explain false sharing or wasted cache-line space. HeapLENS therefore samples whole pages along with their complete (de)allocation histories.

Even this page sample can be dominated by repeated layouts. HeapLENS thus groups similar pages and selects representatives from the resulting clusters. The key to page grouping is a temporal type-and-alignment fingerprint that distinguishes, for example, pages with different object type profiles, and pages with identical type profiles but different alignments. Fingerprints summarize each page’s history and the objects live at several execution times, allowing grouping to reflect which objects coexist as well as which objects use a page over the execution. Representative selection then balances coverage of common layouts with layouts different from those already selected. §4.3 explains these techniques. §4.4 evaluates layout preservation.

Another complication is that `malloc` does not capture objects of interest in applications such as RocksDB that request a large buffer, then divide that memory among smaller objects using *placement new*. Such nested allocations are invisible to traditional tools, but HeapLENS tracks their containment relationships and lifetimes, making nested allocations visible. §4.2 explains the (substantial) subtleties of tracking these lifetimes using RocksDB’s arena allocations and a concrete storage-reuse example.

Perhaps surprisingly, the challenges of data scale do not end once a small set of representative pages is selected. With frequent memory reuse, pages can have extremely long histories. For example, in a TPC-C database benchmark, just 55 representative pages had 2.3 million events over 3.1 seconds. Recomputing every view as the user scrubs the timeline would be slow, and precomputation creates other problems. §4.6 explains the engineering needed to make HeapLENS’s data pipeline and UI interactions fast.

We use HeapLENS to identify and explain memory-layout problems in existing applications and benchmarks, then implement and measure changes prompted by those observations. Table 1 summarizes our results. We improve concurrent data structures by up to 30.6%, a TPC-C in-memory database workload by up to 21.1% and RocksDB by up to 8.5%, resulting in an accepted pull request [30]. Although the causes of these problems can be subtle, many fixes require

only small changes to placement, alignment, or field order. §5 explains the investigations in detail.

HeapLENS also produces text exports suitable for LLM ingestion. We had GPT 5.5 independently investigate layout problems, propose changes, and test optimizations in Valkey, a Redis-compatible key-value store, and HNSWLib, a library for approximate nearest-neighbour search. Its proposed optimizations improve mean Valkey throughput by 4.2% and HNSWLib query throughput by 9.8% and 6.1% at 128 and 1536 dimensions, respectively. Given similar prompts and freedom, but no HeapLENS data, the LLM’s search for improvements was less focused and outcomes were worse.

## 2 Background

We first explain the memory and cache organization used in our examples. Memory is divided into pages, usually 4096 bytes on x86/64, with huge pages of 2 MiB or 1 GiB also available. Data moves between memory and caches in aligned cache lines. We assume 64-byte cache lines, which are common. Accessing a byte that is not cached brings in its entire line, and sometimes an adjacent line (to form a 128 byte aligned pair of lines) via prefetching [22].

A cache is divided into *sets*, and hardware determines which sets can hold which lines. For the virtually indexed L1 caches considered here, a line at address  $a$  maps to set  $\lfloor a/C \rfloor \bmod S$ , with line size  $C$  and  $S$  sets. This mapping makes the addresses of frequently accessed objects important. Last-level caches can have more complex mappings [15]. An  $m$ -way set-associative cache holds at most  $m$  lines in each set. If an access brings another line into a full set, an existing line must be evicted *even if other sets have unused capacity*.

### 2.1 Memory usage and layout issues

**Fragmentation.** Fragmentation refers to unused space between or within allocations: deallocation can leave holes between objects that are too small to satisfy future allocations, and alignment and padding can introduce unused space within objects. Notably, compilers often introduce the latter as they align fields to use more efficient load/store instructions. Consequently, a struct’s *size* can depend on the *order* of its fields. As an example, a struct containing: `char, int, char, int, char, int, char, int`, in that order, will typically occupy 32 bytes: each `int` is 4-byte aligned, resulting in 3-byte padding after each `char`. Reordering the `ints` before (or after) the `chars` reduces the struct to 20 bytes. Of course, this matters even more for arrays of structs.

One might suggest prescriptively packing structs to avoid wasted memory, but if nearby fields are accessed close together in time, reordering them may push related fields into different cache lines, causing additional cache- and even TLB-misses. §5.4 shows how careful field ordering led to both a memory- and runtime-saving change in RocksDB.

| Application/data structure     | Throughput | Cache misses/op <sup>1</sup> | DTLB misses/op <sup>1</sup> | LOC changed <sup>2</sup> |
|--------------------------------|------------|------------------------------|-----------------------------|--------------------------|
| ASCYLIB/EFRB tree (§5.2)       | +30.6%     | -79.7%                       | -30.6%                      | +13, -8                  |
| ASCYLIB/DVY tree (§5.2)        | +19.7%     | -36.1%                       | +0.00013 <sup>3</sup>       | +1, -1                   |
| ASCYLIB/HJ tree (§5.2)         | +6.5%      | -24.4%                       | -0.1%                       | —                        |
| TPC-C/BCCO tree (§5.3)         | +15.5%     | -12.2%                       | -18.3%                      | +8, -12                  |
| TPC-C/EFRB tree (§5.3)         | +21.1%     | -23.3%                       | -66.7%                      | +11, -6                  |
| RocksDB/prefix hash (§5.4)     | +8.4%      | +1.4%                        | -1.4%                       | +1, -1                   |
| RocksDB/inline skiplist (§5.4) | +8.5%      | -2.9%                        | -2.2%                       | +27, -1                  |
| Valkey/string cache (§5.5)     | +4.2%      | -3.2%                        | +1.6%                       | +39, -3                  |
| HNSWLib, 128 D (§5.5)          | +9.8%      | -5.4%                        | -88.9%                      | +128, -0                 |
| HNSWLib, 1536 D (§5.5)         | +6.1%      | -5.3%                        | -47.3%                      | +128, -0                 |

**Table 1.** Mean performance improvements obtained using HeapLENS. Ten trials each. <sup>1</sup>Per operation for ASCYLIB, RocksDB, Valkey and HNSWLib; per transaction for TPC-C. <sup>2</sup>Lines of code (added, deleted). <sup>3</sup>Absolute increase from 0.00015.

**Cache set underutilization.** Regular patterns in memory allocations can lead to undesirable patterns in cache occupancy. For example, consider allocation of cache line sized objects of two types, *A* and *B*, in alternation. Assuming a typical virtually indexed cache with an even number of sets, and that the first *A*’s address is a multiple of 128, *A* objects can occupy only the even sets and *B* objects can occupy only the odd sets. This may be fine if *A* and *B* objects are accessed equally often. However, if only *A* objects are accessed on the hot path, then the application effectively only uses half of the cache. Researchers have proposed avoiding this problem by occasionally inserting padding between allocations to avoid regular patterns [1], or segregating performance critical types in separate allocation regions [4].

**Crossing cache lines.** Objects that cross cache line boundaries can cause more cache misses. Alignment solves this problem for 64 byte objects, but larger objects can also benefit from alignment and field ordering. If only some of the fields are regularly accessed, collocating them in one cache line permits access with only one cache miss. Crossings can also be caused by an object’s size. For example, while allocators like jemalloc avoid crossings for 32 and 64 byte objects, typically half of 48 byte objects will cross cache lines. A trade-off arises: if accesses to 48 byte objects rarely miss in the cache, there is little harm in these crossings, but if misses are common, it may be better to align them to 64 bytes so loading an object incurs one miss, instead of two.

**False sharing.** It is well known that alignment can prevent false sharing between unrelated data in a cache line, trading contention for cache fragmentation. It is less known that false sharing can occur *across neighbouring cache lines* on systems with adjacent-line prefetching [4]. Aligning to 128 can help, but when the number of cache sets is a power of two, this can limit objects to appear only in even numbered sets, since address  $128c$  appears in  $\text{set } \lfloor 128c/64 \rfloor \bmod 2^k$ , which must be even. In §5.2, we show that aligning to 192 bytes can be better, since  $\lfloor 192c/64 \rfloor \bmod 2^k$  can be even or odd (uniformly for uniform  $c$ ).

### 3 Related Work

Related tools tackle a variety of questions: how much is allocated, which accesses miss the cache(s), how objects reference one another, and how they are arranged in memory. Table 2 compares HeapLENS with its closest related work.

#### 3.1 Memory layout optimization

Several approaches improve locality by placing frequently co-accessed objects or fields together. These include array regrouping [13], structure splitting [42], and field reordering [8]. Profilers by Yu et al. [39] and Ye et al. [38] estimate which fields or objects are accessed together to guide such optimizations. STAFF [29] additionally considers alignment. Allocators also influence locality through their placement policies [3, 8, 20, 34]. For example, mimalloc [23] allocates from per-chunk free lists, while Afek et al. [1] break up regular allocation patterns to reduce cache-set underutilization.

Whereas the aforementioned techniques are restricted to specific layout problems and solutions, HeapLENS is a general profiling and visualization tool that supports diagnosis of a broad variety of problems. It naturally complements access-oriented tools such as Linux perf [12], which provide information about hot types, fields and addresses. For example, HeapLENS can ingest perf c2c cache behaviour reports, and visualize relevant pages, allowing the user to understand how memory layout contributes to the observed behaviour.

#### 3.2 Memory profiling and visualization

Massif [28], VTune [10], and allocator profilers such as jepprof [17] track memory use or allocation sites. AntTracks and TrendViz [24, 36] plot Java memory use over time. Such views reveal growing memory consumption, but offer no insight into questions like “Which typed objects regularly share a cache line or occupy particular cache sets?”

Heapviz [2] and Memory Cities [37] organize objects by references, types, or ownership. These views explain how objects are connected, but omit the byte-level layout needed to inspect alignment, padding, or sharing of cache lines.

| System             | Target       | Heap view              | Object types | Page selection | Collection             |
|--------------------|--------------|------------------------|--------------|----------------|------------------------|
| MemoView [27]      | C/C++        | Address/time           | No           | No             | Allocator hooks        |
| AllocRay [33]      | C/C++ (Win.) | Address/time           | No           | No             | Allocation tracing     |
| Heapviz [2]        | Java         | Object-reference graph | Yes          | No             | Heap snapshots         |
| Memory Cities [37] | Java         | Type/ownership views   | Yes          | No             | Runtime events         |
| HeapLENS           | C/C++        | Pages, lines, fields   | Best effort  | Clustering     | Source instrumentation |

**Table 2.** Closest heap-visualization comparisons. HeapLENS fingerprints pages, clusters similar page layouts, and selects representative pages to reduce the volume users must inspect. HeapLENS infers C/C++ allocation types.

CVT [35] and YACO [31] visualize **simulated** cache behaviour. MTV [9] animates reads and writes and shows their simulated cache hits and misses. Its detailed views require users to specify memory regions of interest. Accuracy depends on how closely the simulated cache model matches the target hardware, and detailed simulation and reference tracing can make realistic workload sizes and durations impractical [28]. Reducing the workload to make it feasible can remove the conditions that trigger a layout anomaly. Moreover, for a component of a distributed service, a representative simulated execution must also reproduce (simulate) its interactions with the rest of the deployment! HeapLENS, in contrast, allows it to interact with its real peers.

MemoView [27] and AllocRay [33] are the closest comparisons. MemoView plots each allocation as a rectangle spanning its address range and lifetime, with colour representing attributes such as the allocating process or function. Its data collection requires allocator-specific instrumentation, and replacing an application’s allocator with MemoView’s instrumented one can destroy the layouts we want to understand. AllocRay visualizes how allocated blocks change over time, with zooming, filters, and colouring by age, size and thread. Neither tool provides C/C++ object-type information, nor support for the nested allocation scenario in §4.2, nor a way to group similar page layouts and select representatives.

**Technical distinction.** Among the systems in Table 2, only HeapLENS records C/C++ types. The other type-aware systems obtain their types from managed-language runtimes. Obtaining type information in C/C++ is far more difficult: malloc specifies a size but no type. Some applications use new/malloc to obtain a large buffer, then logically reserve regions for *inner* objects, accessed through type-cast pointers. HeapLENS must infer types from the source and track inner objects as well as their buffers. Sections 4.1–4.2 explain how we track types and inner object lifetimes.

HeapLENS alone selects a manageable set of diverse layouts for human inspection. Page sampling, fingerprinting, clustering, and representative page selection reduce data while reliably preserving the layout patterns observed in our experiments. Sections 4.3–4.6 explain these techniques and the engineering needed for responsive interaction.

## 4 HeapLENS

To collect an allocation history, users run HeapLENS’s Clang-based instrumentation on the application’s source and link it with the logging library. The instrumentation automatically locates allocation calls and adds logging, avoiding manual edits to individual call sites. For custom allocation interfaces, users provide function names and short wrappers (§4.1).

Figure 1 shows HeapLENS’s components. The **allocation-log processor** tracks object lifetimes, including those of inner objects, and builds page layouts, splitting objects that cross page boundaries into records for each page they occupy. It then *samples* whole pages and stores their histories in SQLite. The **visualization backend** reads these histories from SQLite and prepares the data for the interactive views. It fingerprints each page’s history and its live objects at selected times, clusters the fingerprints, and chooses representative pages within display budgets (§4.3). To support timeline navigation, the allocation-log processor precomputes memory-usage time series, while the visualization backend prepares cache-set occupancy time series (§4.6).

### 4.1 Instrumentation and logging

HeapLENS collects allocation histories by instrumenting the application’s source code. A Clang tooling pass automatically rewrites allocation and deallocation calls to record their arguments and source locations through HeapLENS’s logging library, which is linked with the application. This covers C functions such as malloc and free, and C++ new and delete, including placement new.

The instrumentation also identifies the allocated object’s type where possible. C++ new provides the constructed type directly. For C-style allocation, types are inferred on a best-effort basis. An allocation is assumed to have type  $T$  if any of the following holds: (1) its result is stored in a variable of type  $T^*$ , (2) its result is explicitly cast to type  $T^*$ , or (3) its size argument contains a `sizeof(T)` expression. The rewritten call includes identifiers for the inferred type and source file, along with the source line number.

For custom allocation interfaces, the user registers the function name with the Clang matcher and supplies a wrapper that invokes the allocator and records the returned address, size, and metadata. The tool then rewrites calls to

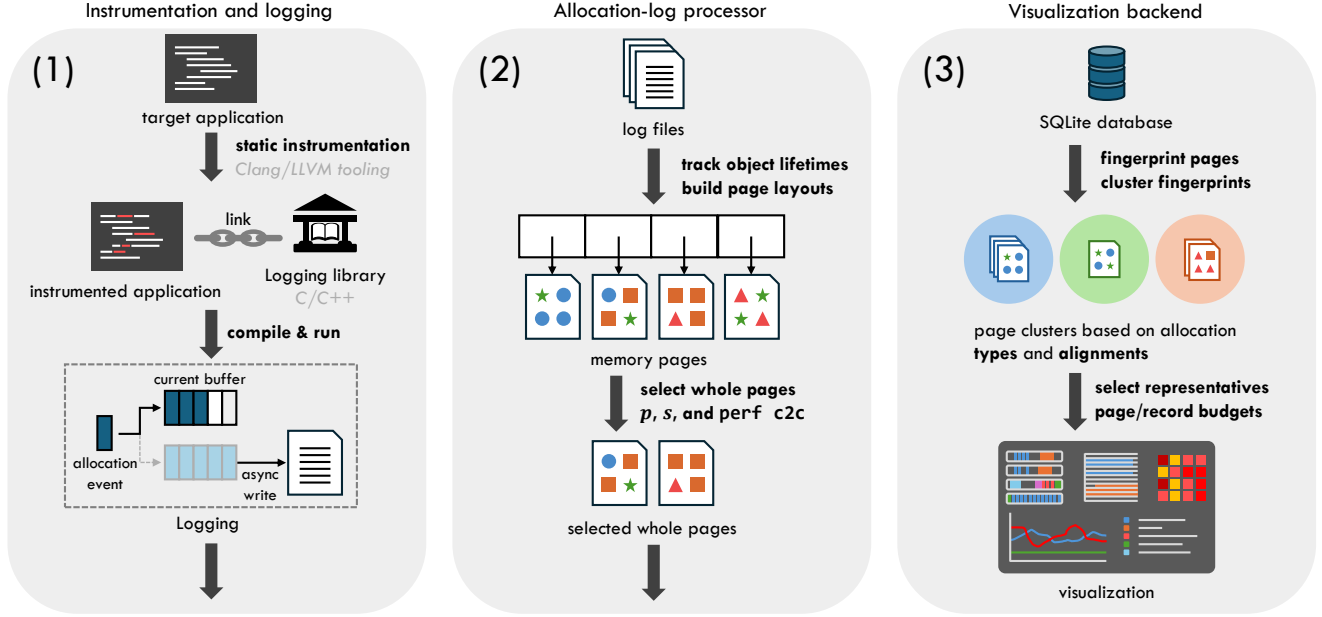


**Figure 1.** HeapLENS’s pipeline, from allocation logging to interactive visualization. Whole-page sampling determines which pages enter the database. Fingerprinting and clustering guide a separate selection of pages for display.

that function automatically: `arena_alloc(sizeof(Node))` becomes `arena_alloc<Node, 51, 7>(sizeof(Node))`, where 51 is the source line and 7 identifies the source file. File names and types are mapped to integers for compact log records. These wrappers and placement-new instrumentation expose objects allocated within larger buffers, as in RocksDB’s arenas.

Logs are accumulated in memory, with each thread initially allocating two buffers of one million 40-byte records each (76.3 MiB per thread). Each event is appended to the thread’s current buffer. When that buffer fills, the thread submits it for asynchronous writing and looks for another buffer whose writes have completed. If none is available, it dynamically allocates an extra buffer, subject to limits of four buffers per thread and 2 GiB of extra buffers process-wide. We call this *adaptive buffering*. If there is no usable buffer, logging waits in `aio_suspend` for a write to complete. This trades additional memory for fewer waits when writes fall behind event collection. On thread exit, HeapLENS preserves any unwritten records and releases the thread’s extra buffers.

**Logging overhead.** We measure logging overhead by comparing throughput with and without logging. Table 3 uses the baseline applications of Table 1, except that ASCYLIB variants were search-only and so would not heavily exercise logging. For those, we perform 10% insertion and 10% deletion, except for ASCYLIB/HJ, which has a memory reclamation bug and crashes. Overhead includes instrumentation, logging, and IO wait measurement, with logs written to local NVMe. The measured results exclude initialization and

| Application             | No logging | Logging | Loss   | Wait |
|-------------------------|------------|---------|--------|------|
| ASCY/EFRB 20% update    | 7010       | 6500    | 7.2%   | 0.0% |
| ASCY/DVY 20% update     | 16700      | 16300   | 1.9%   | 0.0% |
| TPC-C/BCCO              | 969        | 550     | 43%    | 17%  |
| TPC-C/EFRB              | 126        | 120     | 4.6%   | 0.0% |
| RocksDB/prefix hash     | 12600      | 5010    | 60%    | 47%  |
| RocksDB/inline skiplist | 5830       | 5870    | -0.67% | 0.0% |
| Valkey                  | 1090       | 1030    | 5.6%   | 0.0% |
| HNSWLib/128D            | 66.3       | 64.6    | 2.5%   | 0.0% |
| HNSWLib/1536D           | 7.15       | 7.14    | 0.20%  | 0.0% |

**Table 3.** Logging overhead in Table 1 workloads. 10 trials. Performance in Kops/s. Wait = % worker time waiting on IO.

teardown (but include RocksDB’s background writer completion). All data is averaged over 10 trials.

In most cases, overhead is quite low—less than 6%. However, in two cases it is extreme: 43.3% for TPC-C/BCCO and 60.2% for RocksDB’s prefix hash memtable. For each trial, we divide summed buffer-wait time by summed worker elapsed time over the measured phase, then average these fractions. In the extreme cases, workers spent 16.6% and 47.4% of measured worker time waiting, respectively, for buffers to flush. For TPC-C/BCCO, transactions perform 70 (de)allocations on average, and its unlogged throughput would flush 2.7 GiB/s, 2.7x our ideal NVMe bandwidth without considering filesystem overhead. RocksDB/prefix hash is similar (3GiB/s).

Interestingly, TPC-C/BCCO has higher loss relative to its wait time than RocksDB. This is because waits cause

transactional conflicts in TPC-C: 10.5% of attempts abort with logging versus 0.05% without.

We compare adaptive and fixed two-buffer logging for baseline 24-thread TPC-C/BCCO. Adaptive buffering improves throughput by 18.1%, reducing loss from 52.0% to 43.3%, with 1.3 GiB higher mean peak RSS.

## 4.2 Allocation-log processor

**Tracking object lifetimes.** As we have discussed, a large raw allocation can hide many smaller objects. In RocksDB’s HashSkipList, for example, an Arena obtains a slab and its `AllocateAligned` method reserves smaller regions within it. Placement new then constructs nodes and skiplist buckets in those regions. Logging only the slab would display one large allocation with none of these contents visible. Logging the suballocations and object initializations gives several answers to “what is at address  $x$ ”: the enclosing slab, the region reserved in it, and the typed object in that region.

HeapLENS processes allocation and deallocation events in timestamp order. When a new allocation lies within an existing live allocation but does not occupy its entire address range, HeapLENS records it as a child of the smallest enclosing allocation. These parent–child relationships form a forest of allocation trees, whose nodes record address ranges and lifetimes. An address index matches deallocations to live allocations, while an ordered interval index finds enclosing and overlapping allocations.

Figure 2 shows how this interacts with memory reuse. The application uses placement new to construct object  $G$  in already-allocated storage, overlapping live regions  $B$  and  $C$  without being contained in either. This replaces the objects in those regions without calling `free` at their addresses. HeapLENS inserts deallocation events for  $B$ ,  $C$ , and their descendants at  $G$ ’s timestamp. When the enclosing slab  $A$  is freed, the logged free names only its base address. HeapLENS uses the allocation tree to identify all objects still contained in  $A$ , directly or through other objects, and inserts deallocation events for them at the slab’s deallocation timestamp, including objects on other pages. These synthetic deallocations record when objects cease to be live; they do not free application memory. Otherwise replaced objects would appear to coexist and objects would survive slab reclamation in the visualization.

**Building page layouts.** Once an object’s lifetime has been determined, the allocation-log processor identifies the pages it occupies. For each page, it records the address and size of the part of the object on that page, together with the original allocation’s address, size, and lifetime. Each page’s history thus records which allocations occupied its bytes and when they were live. When describing page contents below, we refer to a split object on a page as simply an “object.”

**Selecting whole pages for the database.** Once page layouts are built, we select pages with configurable probability

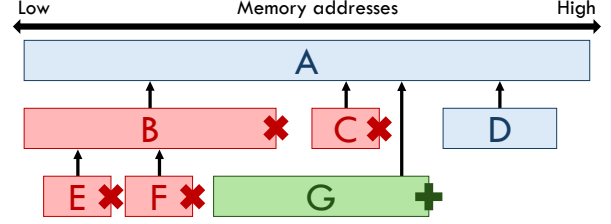


**Figure 2.** Nested allocation lifetimes. Arrows mean “contained by.” Allocating  $G$  ends the lifetimes of overlapping regions  $B$ ,  $C$  and their children  $E$ ,  $F$ . During offline processing, HeapLENS inserts synthetic deallocation events for  $B$ ,  $C$ ,  $E$ , and  $F$  at  $G$ ’s allocation timestamp.

$p$ . This sample can miss types occupying few pages. For each logged type, we supplement the sample with pages containing that type until at least  $s$  such pages are selected.<sup>1</sup> If the type occupies fewer than  $s$  pages, all pages containing that type are selected. Pages already selected via  $p$  count toward this minimum. With  $s \geq 1$ , every logged type is represented in the database. Pages implicated by imported `perf c2c` measurements are always included so that users can inspect the layouts associated with those measurements. The allocation-log processor stores each selected page’s allocation history, including synthetic deallocations, in SQLite. Allocation logs remain available to process with different sampling settings.

## 4.3 Visualization backend

The sample prepared by the allocation-log processor can contain many repeated layouts. The backend fingerprints and clusters these pages, then selects representatives so that common layouts do not dominate the display.

**Page-history fingerprints.** For each page in SQLite, we build a histogram counting objects in its recorded history by type and starting-address alignment. With 64-byte cache lines, each bin corresponds to a type and one of the 128 possible byte offsets within a pair of cache lines. In general, bins are indexed by  $(T, a \bmod 2C)$ , where  $T$  is a logged type,  $a$  is the object’s starting address on the page, and  $C$  is the cache-line size. Each count is scaled relative to the largest count in the same bin across all database pages, then quantized to an integer from 0 to 3, with positive counts kept nonzero. The resulting histogram is the page’s *history fingerprint*.

Type distinguishes pages with similar density but different object types. Alignment distinguishes how objects lie within or across cache lines. For example, consider two pages containing equal numbers of 64-byte Nodes. On one page, nodes begin at offsets 0 or 64 modulo 128 and each fits in one cache line; on the other, they begin at offsets 32 or 96 and cross cache lines. The pages have identical type counts and density, but their history fingerprints differ.

<sup>1</sup>Pages are selected independently: if an object spans multiple pages, selecting one page does not pull in other pages occupied by the same object.

Using two cache lines also distinguishes even- and odd-numbered cache sets: offsets 0 and 64 modulo 128 map to even and odd sets, respectively, under the mapping in §2. Objects concentrated in only even- or only odd-numbered sets can leave half the sets underused. Placement within adjacent-line pairs also matters for prefetch-induced contention [4]. Although the fingerprint does not preserve the exact order of neighbouring objects, users can inspect that order in the representative page’s layout.

**Per-time page fingerprints.** A history fingerprint does not say which objects coexist. Two pages can have identical counts by type and offset even though the objects coexist on one page but are allocated at different times and never coexist on the other. HeapLENS therefore also fingerprints the objects live at sixteen evenly spaced execution times.<sup>2</sup> Duplicate *per-time fingerprints* of the same page are ignored.

Before clustering, the visualization backend scales each varying fingerprint dimension to unit variance so that high-variance dimensions do not dominate the distance calculation. It clusters history fingerprints and per-time fingerprints separately using mini-batch *k*-means, each with up to twice as many clusters as logged types. Each page contributes one history fingerprint and potentially several distinct per-time fingerprints. These fingerprints can belong to different clusters, so one page can be a representative for several clusters.

**Reducing repeated histories.** Repeated memory reuse can create a long history even for a single page. To reduce the history sent to the browser, the visualization backend divides execution time into buckets (2000 by default). For each type and address, it retains only the last object allocated within each bucket, together with that object’s deallocation time. Thus, if two successive objects of the same type are allocated at the same address within one bucket, only the second is kept.<sup>3</sup> Wider buckets reduce the number of retained objects, but can hide objects that existed only briefly. Fingerprints are computed before this reduction. Representative selection uses the reduced histories to determine how many records each selected page would require.

**Selecting representatives with page & record budgets.** After clustering, HeapLENS must choose which pages to display. A page budget keeps the display small enough to inspect. A separate budget on retained object records limits the allocation history sent to the browser. These budgets limit the *displayed* representatives, not the pages in the database. We use budgets of 128 pages and one million records. Each chosen page includes its complete reduced history rather than a history truncated to fit the record budget. Pages flagged by `perf c2c` measurements are also always selected for display. They count toward both display budgets but remain included

even if they exceed either limit. All other selections must fit both budgets.

The visualization backend uses **cluster coverage first**: it seeks a representative from each cluster before selecting additional pages from already represented clusters. When the budgets cannot accommodate every cluster, considering clusters in random order can leave distinctive layouts unrepresented. To choose which cluster to consider next, the backend uses **alternating cluster selection**: it alternates between random order and choosing the cluster most different from those already represented. Here, “most different” means having the greatest distance to the nearest already-represented cluster. Cluster-centre distances are normalized within each fingerprint collection to make history and per-time distances comparable.

For each cluster considered, the backend chooses a representative page that fits the remaining budgets and whose number of object records after history reduction is close to the median for that cluster. This favours a typical history size rather than simply the cheapest page to display. A selected page represents *all* clusters containing its fingerprints. Among equally suitable pages, the backend therefore prefers those that also represent *other* clusters still missing from the display. Already-represented clusters and clusters with no page that fits the remaining record budget are skipped.

After this, if the budgets allow more pages, the backend first selects pages containing types still missing from the display. The earlier *s* minimum ensures type coverage in the database, but not necessarily among the displayed pages. It then tries to show the range of history sizes within each cluster by selecting pages with the fewest and most object records. Finally, it adds short runs of consecutive pages from a cluster so users can inspect layouts across page boundaries.

#### 4.4 Effectiveness of representative pages

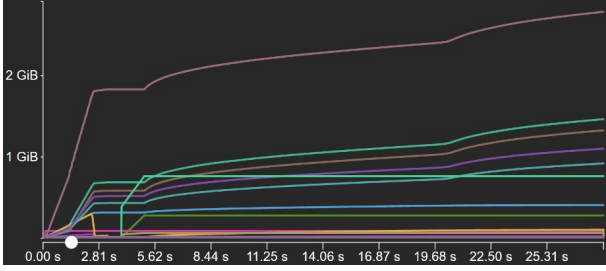
We evaluate whether page sampling and representative selection preserve known layout patterns and for how long the patterns remain visible. Before sampling, for each pattern, we use recorded object lifetimes to find all intervals during which the pattern appears on any page. After sampling, history reduction, and representative selection, we find the intervals during which the displayed pages show each pattern. We compare the displayed intervals with the intervals before sampling. We also check whether each pattern appears on a displayed page at the time-bucket boundaries.

We define *duration coverage* as the total time that selected representatives show a pattern, divided by the total time that pattern occurs anywhere in the complete (unsampled) set of pages. Occurrences on multiple pages at the same time count only once.

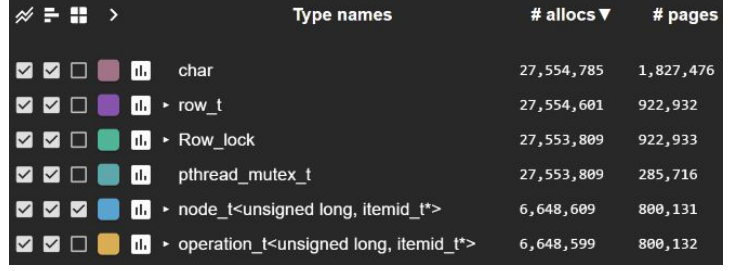
Our twelve test patterns come from the application investigations summarized in Table 1. We define each pattern in terms of object types, sizes, positions, and lifetimes. For example, the EFRB case study (§5.2) examines interleaved

<sup>2</sup>These times are used only to compute fingerprints; users can inspect the selected pages at other times along the timeline.

<sup>3</sup>Different types at the same address are preserved to support nested objects.



(a) Per-type memory consumption and time selection (slider).



(b) Type/field legend, counts, and visibility/filtering controls.

**Figure 3.** Time navigation and filtering for Figures 4 and 5. Allocation/page counts and workload expectations help guide exploration. Time scrolling distinguishes persistent patterns from initialization effects. Arrows expand struct fields.

tree nodes and operation descriptors. The EFRB test looks for a full page with 48 live, cache-line-aligned, 64-byte nodes and 16 live descriptors of the same size, with a descriptor in every fourth cache line (as we observe in §5.2).

Separately, we test whether exceedingly rare layouts survive representative selection. In the optimized RocksDB inline skiplist (§5.4), just **3 out of 22,777 pages** mix tall and short nodes. This layout is not a known problem—just rare.

**Methodology.** For each of the ten configurations in Table 1, and for the optimized inline skiplist used in the rare-layout test, we use the allocation logs from one execution. We repeat page sampling and representative selection with twenty random seeds at each of two settings,  $(p, s) = (1, 1)$  and  $(0.1, 1)$ , giving 440 trials. Each trial compares the *full selection algorithm*, with all techniques enabled, against three variants that separately disable *per-time fingerprints*, and the *cluster coverage first* and *alternating cluster selection* heuristics, respectively. For each of the 440 trials, we run each of the four algorithm variants with both 128- and 32-page budgets, giving 3520 selection runs. All representative selections use a one-million-record budget and 2000 displayed time-buckets. Within a trial, all versions use the same sampled pages and share the clusters for any fingerprint collection they use in common. We average duration coverage over the twenty seeds for each pattern and sampling setting, then report geometric means across these averages.

Table 4 summarizes the full selection algorithm’s preservation of the twelve Table 1 patterns. With a 128-page budget, every pattern remains visible in every trial at both sampling settings, and each pattern’s mean duration coverage is at least 92.8%. Even with a 32-page budget, every pattern remains visible in at least 18 of 20 trials at both sampling settings. Only gaps between database rows in TPC-C/EFRB and nodes crossing cache-line boundaries in RocksDB’s inline skiplist are missed in any trial.

**Ablation.** With a 128-page budget, we separately disable per-time fingerprints, and the cluster coverage first and alternating cluster selection heuristics. The full selection algorithm improves geometric-mean duration coverage by 22.6%,

|     | Page budget | Patterns found in all 20 trials | Duration coverage |
|-----|-------------|---------------------------------|-------------------|
| 1   | 128         | 12/12                           | 99.0%             |
| 0.1 | 128         | 12/12                           | 99.3%             |
| 1   | 32          | 11/12                           | 91.5%             |
| 0.1 | 32          | 10/12                           | 91.3%             |

**Table 4.** Preservation of the twelve Table 1 patterns by the full selection algorithm. The third column counts patterns visible on at least one displayed page at some time-bucket boundary in each of the twenty trials. Duration coverage is the geometric mean across patterns after averaging each pattern’s coverage over twenty trials.

5.3%, and 9.5% relative to these versions, respectively. These geometric means include all thirteen patterns, including the aforementioned rare pattern, at both sampling settings.

**Rare pattern.** With a 128-page budget, the exceedingly rare mixture of tall and short nodes is visible in 17/20 trials at  $p = 1$  and 15/20 at  $p = 0.1$ , compared with 5/20 at each setting without alternating cluster selection. Even with a 32-page budget, alternating cluster selection increases rare-pattern visibility from 1/20 trials to 13/20 at  $p = 1$  and 12/20 at  $p = 0.1$ . This gain trades off some duration coverage of the twelve Table 1 patterns, whose geometric mean falls from 94.5% to 91.4% across both settings.

#### 4.5 Reading and investigating the linked views

Figures 3–5 illustrate a memory layout investigation in a 24-worker/warehouse TPC-C execution using the ASCYLIB/HJ tree as its index.<sup>4</sup> Full system details appear in §5. We begin with the legend in Figure 3, looking for frequently allocated types. From the allocation counts, we might suspect strings, rows, row locks and mutexes are allocated together. For each of these types, we restrict the cache-set heatmap to that type

<sup>4</sup>These linked views appear on one screen, but we separate them for clarity.

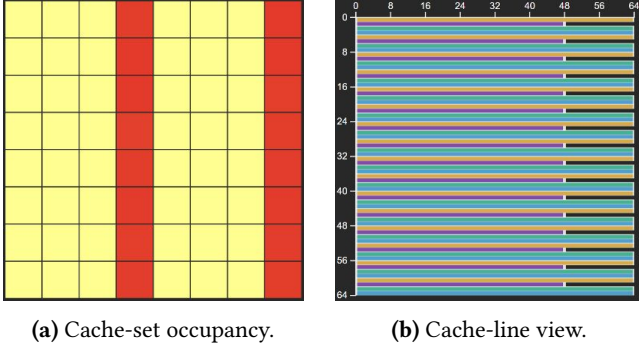


**Figure 4.** (a) Heatmap cells show how evenly nodes occupy L1d cache sets 0–63, in reading order. Yellow to red scale: low to high occupancy. (b) Cluster 98 page with repeating object patterns. Rows are cache lines.

(node\_t in the figure) and move the time slider to examine how their cache set occupancy changes over time.

At the selected time, the heatmap in Figure 4(a) shows nodes suffer imbalance across cache sets. The heatmap summarizes node occupancy across *all pages in the database*.<sup>5</sup> We turn to the page view in Figure 5 to look for layouts that could explain this imbalance, keeping other types visible so we can see what lies between nodes. Colours indicate types, and pages from clusters 98 and 12 show the same repeating allocation pattern.<sup>6</sup> We select the page from cluster 98, opening its cache-line view in Figure 4(b). The pattern consists of an operation descriptor, a database row, a row lock, and an index node. Nodes consequently occupy every fourth cache line on this page. §5.2 shows that this regularity can be an anti-pattern, affecting cache-set use in HJ and other trees.

**Perf C2C integration.** Figure 6 shows a separate HNSWLib example: A HierarchicalNSW<float> index object is expanded to show its fields, and cache lines are marked to show perf c2c hit-modified cache events affecting several statistics counters, a deletion-policy flag and part of a mutex.

#### 4.6 Efficient processing and responsive interaction

Recomputing HeapLENS’s views as the user scrubs through time and toggles type visibility would require processing an enormous volume of data in real time. Not only can there be up to 1 million object records, but the cache-set occupancy heatmap is built using the entire contents of the database. Making the UI responsive thus required a lot of engineering.

##### Computing memory usage and cache-set occupancy.

Before page sampling, the allocation-log processor precomputes and stores the live-byte total for each type at each UI time bucket. A 64-byte allocation contributes +64 to the

<sup>5</sup>Otherwise, it would be vulnerable to representative page selection bias.

<sup>6</sup>These pages may look identical, but their history fingerprints differ, because one of them later contains strings, and the other does not.

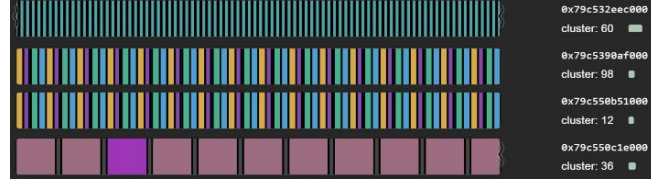


**Figure 5.** Scrollable panel of representative pages from a TPC-C execution. Colour: type. Sort: by address. Clusters 98 & 12 show repeating patterns.

start-time bucket *for its type* and –64 to the end-time bucket. The processor sums these changes within each bucket, then computes per-type running totals across buckets to produce the time series in  $O(\#records + \#types \cdot \#buckets)$  runtime.

The visualization backend computes cache-set occupancy similarly, but using all pages in the database (after page sampling and before selecting representatives). To support filtering, it computes a separate occupancy time bucket series for each combination of cache set, and type or field. For each cache line an object or field overlaps, the backend adds +1 in the object’s allocation-time bucket and –1 in its deallocation-time bucket for the corresponding set. Consecutive cache lines map cyclically to cache sets. The backend therefore aggregates complete cycles and updates the remaining ranges without enumerating individual lines (we call this **modular set counting**). For example, an object spanning 1024 cache lines in a 64-set cache contributes +16 to each set.

Running totals over buckets (**bucket prefix sums**) give each type or field’s contribution to each set over time. This has runtime  $O(\sum_{obj} numFieldsIn(obj) + \#sets \cdot (\#types + \#fields) \cdot \#buckets)$ . The browser receives these tables alongside the per-type memory-usage time series. To draw the heatmap, it sums the contributions of the types and fields enabled in the legend at the selected time. Changing the time or filters therefore combines precomputed values rather than revisiting objects and their lifetimes.

**Processing ablation.** We study how **modular set counting** (MSC) and **bucket prefix sums** (BPS) affect the time to compute cache-set occupancy on Valkey and TPC-C/BCCO traces, testing all four combinations. Without MSC, the backend enumerates individual cache lines to update cache sets. Without BPS, it directly updates every time bucket an object occupies. Each result is the median of three trials. For Valkey, computing cache-set occupancy takes 155.8 seconds with neither technique, 21.3 seconds with MSC alone, 11.8 seconds with BPS alone, and 6.8 seconds with both. For BCCO, MSC improves the computation from 13.5 to 11.5 seconds, but adding BPS worsens that to 14.2 seconds, since objects span few time buckets, so direct updates are already efficient, and the prefix sum computation does not justify its cost.

**Managing UI latency.** When the user changes which types are visible, HeapLENS must sum their contributions to

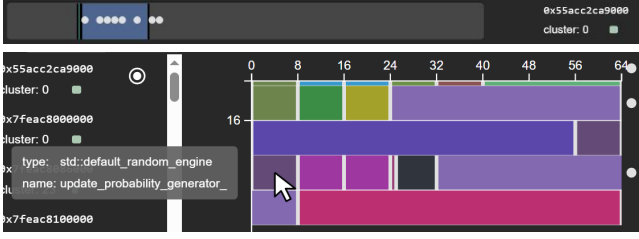


**Figure 6.** HNSWLib with perf c2c. Dots show cache events. Fields are expanded and mouse-over interrogates them.

page occupancy and cache-set occupancy. A straightforward optimization is to compute these sums only for the selected time bucket, and recompute as the time slider is moved. In Edge 154 with an RTX 3080, for a Valkey view of 52.4 KiB pages, this reduced median latency when toggling visible types from **805 to 79 ms**. Dense huge-pages present a different challenge. When a user mouses over a 2 MiB page in HeapLENS, a 64 KiB chunk is highlighted and clicking opens that chunk in a scrollable variant of Figure 4. Opening that view should not require constructing details for the whole 2 MiB page and then truncating. So, HeapLENS first filters histories to objects intersecting the selected chunk, including objects that *enclose it*, and then constructs only their display components. On a synthetic huge page containing 65,536 32-byte objects, this reduced constructed rectangles to 2048 and latency from **19.1 to 0.4 seconds**.

## 5 Case Studies

Our system has two 24-core Intel Xeon Gold 5220R CPUs and 186GiB DRAM running Ubuntu 24.04.4 LTS with kernel 6.8.0-137. Each core’s L1d cache is 32KiB, 8-way with 64 sets.

The results in Table 1 are averages over ten trials per variant. All variants share compiler, host, CPU pinning and NUMA policy, and allocator, except where explicitly varied. Allocators are dynamically linked. HeapLENS is used only to discover optimization opportunities. Performance results are not instrumented and do not use HeapLENS. We use Linux Perf tools [12] to record cache misses and TLB misses only during the measured workload, excluding initialization. To avoid measurement overhead, these results are gathered from separate runs, meaning performance is measured without Perf tools. RocksDB counts include writer/background work per reader operation. Valkey counts measure the server.

### 5.1 Methodology

As in §4.5, we visualize an application’s memory layout, and explore it interactively to first build understanding. The layout patterns illustrated in §4.5 suggest changes such as adjusting object alignment or separating allocation streams. We inspect the source to understand why a pattern arises and how to change it without affecting correctness. We then

measure the change’s effect on performance. Because occupancy does not reveal how frequently objects are accessed, perf c2c can help focus the search.

### 5.2 ASCYLIB

ASCYLIB [11] is a library of concurrent map data structures, and a timed benchmark with  $n$  threads accessing the same map. We consider 5-second search-only workloads with a lock-free binary search tree (EFRB tree) [16] initially containing  $2^{18}$  keys. These experiments use four workers on cores 0–3 and NUMA-local memory.

**Anomaly: cache set underutilization.** First, following the investigative steps in §4.5, we notice that tree nodes are unevenly distributed across cache sets (Figure 7(a)). In the EFRB tree, an insert operation always allocates three nodes followed by an *operation descriptor*, an object used to facilitate helping (a technique commonly used to achieve lock-freedom). Both nodes and operation descriptors occupy exactly one cache line. Prefilling in ASCYLIB performs insert operations until the desired initial data structure size is reached. The page in Figure 8 shows the resulting repeating sequence of three nodes followed by a descriptor. In the early-prefill snapshot at the top of Figure 7(a), 15 cache sets contain no nodes, and one contains only a single node. As prefilling continues, allocations on other pages shift the positions of nodes and descriptors within this four-line pattern, placing nodes in the previously underused sets. After prefill (bottom of Figure 7(a)), all sets contain nodes, but occupancy remains uneven, ranging from 1,967 to 14,400 node mappings per set. This uneven distribution persists throughout the search-only workload.

**Root cause: internal node accumulation.** EFRB is an *external* or *leaf-oriented* binary search tree (BST). That means its keys are all stored in leaves. Internal nodes contain dummy *routing* keys that serve only to direct searches to the correct leaf. To insert a key, we first traverse until we reach a leaf. If the leaf contains the key we’re inserting, depending on the desired semantics, we either overwrite the existing value or return immediately. Otherwise, we *replace* the leaf with two new leaves and a new parent, retiring the old leaf. The new leaves contain the key of the now-replaced leaf, and the key we are inserting, respectively. So, prefilling in ASCYLIB, which simply inserts new keys, *retires exactly one leaf* per insert and *does not retire internal nodes*. Consequently, internal nodes accumulate while leaves are gradually retired. This partly explains the red columns in the top of Figure 7(a).

**Anomaly: unusable memory.** As we scrolled through the timeline in HeapLENS, we noticed that memory of objects retired during prefilling was never reused. This manifests as grey gaps appearing in the memory page visualization which are never filled in (see, e.g., Figure 8). As a result, the data structure uses much more memory than necessary.

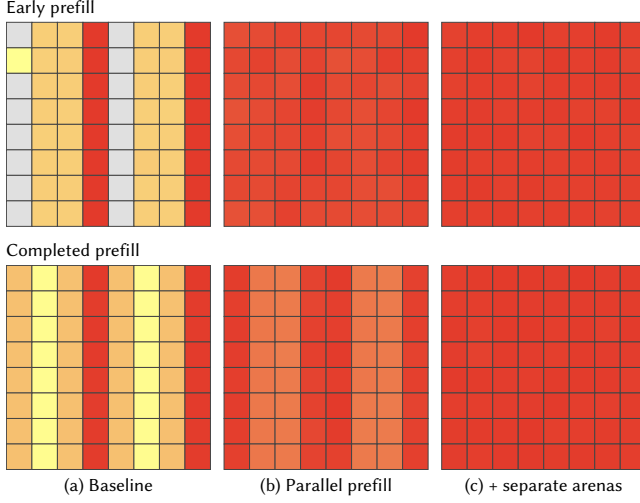


**Figure 7.** EFRB node occupancy across L1 cache sets during early prefill (top) and after prefill (bottom). Columns show (a) baseline, (b) four-thread prefill, and (c) prefill plus separate arenas. Grey means zero; yellow to red indicates increasing occupancy, scaled separately within each panel.

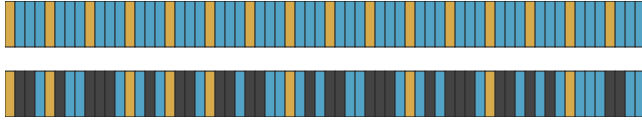


**Figure 8.** One EFRB page early (top) and after prefill. Blue nodes, yellow operation descriptors, dark grey empty space.

**Root cause: an eternal epoch.** By default, prefilling is done by a single thread while the other threads wait. This prevents ASCYLIB’s Epoch-Based Reclamation (EBR) from reclaiming any objects, as the single prefilling thread cannot advance the global epoch unless the other threads take steps to participate in the epoch. This is especially punishing for a search-only workload because that unreclaimed space is never reused, even after prefilling. In a workload with insert operations, the EBR mechanism “kicks in” after prefilling, and that memory can eventually be reused.

A simple fix is to make prefilling multithreaded. This also partially addresses cache set underutilization, because the regularity in the memory allocation pattern is perturbed by the unpredictability of the thread scheduler. Figure 7(b) shows that multithreaded prefilling spreads nodes across all sets early, although final set occupancy still varies.

We can further improve the layout by allocating nodes and operation descriptors from separate memory arenas. Combining this separation with multithreaded prefilling results in balanced sets (Figure 7(c)), on average reducing cache misses/search by 79.7% and increasing throughput by 30.6%.

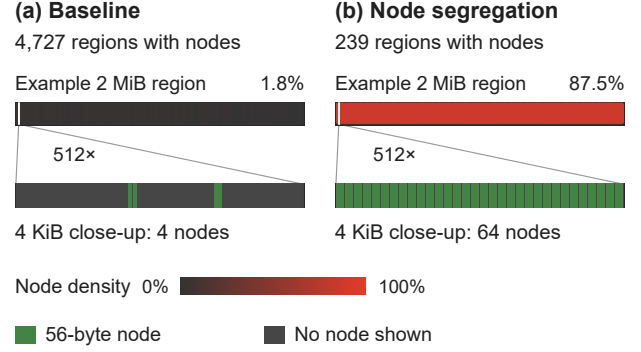


**Figure 9.** BCCO node placement in a 24-thread TPC-C execution. Counts cover all node-containing 2 MiB regions, while percentages give node density in the example regions. The marked 4 KiB portions are enlarged below. Grey in the close-ups means no node, not necessarily free memory.

**Additional trees.** We also investigate the layouts of two other ASCYLIB trees, each starting with  $2^{20}$  keys. In the DVY tree [14], the fields needed for search fit within the first cache line of each 96 byte node. Padding nodes to 128 bytes places these fields two cache lines apart, so they map to only even- or only odd-numbered cache sets, reducing performance. With 192-byte nodes, successive search-critical lines are three cache lines apart and cycle through all 64 sets. With eight search-only threads on physical cores 0–7, the 192-byte variant improves mean throughput by 19.7%. This gain depends on the tree being too large to fit in cache.

In the HJ tree [21], nodes alternate with operation descriptors, concentrating nodes in one parity of cache sets. A 24-thread search-only allocator comparison shows the impact of this layout: glibc shifts the pattern between slabs, distributing nodes across both parities and improving mean throughput by 6.5% over jemalloc.<sup>7</sup> A more targeted fix would be to pad descriptors into the next jemalloc size class.<sup>8</sup>

### 5.3 DBx1000

DBx1000 is a C++ in-memory database benchmark implemented by Yu et al. [40]. Arbel-Raviv et al. [4] studied its performance using several different search trees as database indexes, running the well-known TPC-C OLTP database workload. We reproduce and extend their results using a benchmark called SetBench [6], which derives from [4].

**Database index type: BCCO tree.** We first examine a few memory layout issues that arise in this benchmark when using a binary search tree by Bronson et al. [5] (BCCO tree). One specific memory layout issue noted by Arbel-Raviv et al. [4] occurs when huge pages are used with jemalloc. To allocate memory from a huge page, jemalloc first splits the

<sup>7</sup>Readers familiar with jemalloc might be surprised that glibc is faster here, but in a search-only workload there is no allocation to slow glibc down.

<sup>8</sup>jemalloc allocates objects of different size classes from different pages.

| Variant          | Gain   | Cache/txn | DTLB/txn |
|------------------|--------|-----------|----------|
| Baseline         | —      | 161.5     | 124.4    |
| Node segregation | +3.1%  | 164.1     | 114.4    |
| Packed spin lock | +11.6% | 138.5     | 111.3    |
| Both             | +15.5% | 141.8     | 101.6    |

**Table 5.** BCCO factor analysis: ten runs per variant. Gains relative to baseline 961k txns/s. Counters: misses/transaction.

page into 4KiB **chunks** that are assigned to size classes as needed. The TPC-C benchmark in DBx1000 performs a repeating sequence of allocations of various data types, including items (32B size class), tree nodes, table rows, row locks (64B size class), and strings (128B size class). Chunks containing nodes are thus interleaved with chunks containing other types, spreading nodes across many huge pages.

We run HeapLENS with 24 threads/warehouses, using jemalloc 5.3.0 with huge pages enabled. Filtering to nodes in the huge-page view reveals sparse node allocation across 2 MiB regions (Figure 9). Using a separate instance of the same allocator for nodes prevents interleaving with other database objects, packing nodes into 239 such regions instead of 4,727. The segregated close-up shows 56-byte nodes starting every 64 bytes, accounting for its 87.5% node density.

Examining row locks suggests a separate optimization. Each row lock contains a pointer to a `pthread_mutex_t`. Replacing the pointer and mutex with a four-byte spin lock removes an indirection and reduces storage from 40+8 bytes (mutex+pointer) to 4. It also changes synchronization behaviour, so we measure its performance effect separately from node segregation.

Table 5 isolates these changes over ten runs per variant, using 24 threads on cores 0–23, NUMA-local memory and jemalloc. Node segregation alone improves throughput by 3.1%, spin locks alone by 11.6%, and the two together by 15.5%. Table 1 includes both. In a separate comparison using jemalloc 5.3.0, segregation improves throughput both with huge-page backing disabled and enabled, by 3.1% and 3.9%, respectively.

**Database index type: EFRB tree.** When using the more allocation-heavy EFRB tree as an index, *HeapLENS* reveals an unexpected buildup of *block* objects allocated internally by SetBench’s EBR algorithm. These blocks store pointers to retired objects awaiting reclamation. Although the blocks should also be reclaimed as execution progresses, 8,600+ are allocated during the benchmark and 1,500+ remain at its end. Investigation with *gdb* reveals that each database table has a separate EBR instance. Sharing one instance across all tables accelerates reclamation and reduces block allocations to 174, a reduction of approximately 98%. Beyond the node segregation optimization, sharing the EBR instance improves throughput by a further 20.0%.

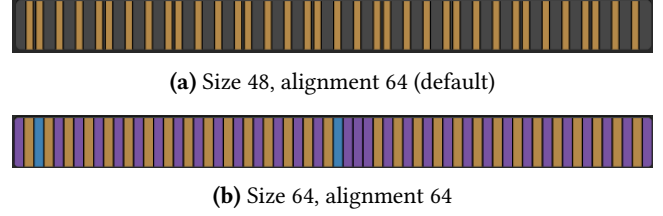


**Figure 10.** mimalloc’s layout for `row_t` objects (orange) in a 24-thread TPC-C execution. Increasing row size eliminates alignment-induced gaps.

**Swapping allocators.** As we saw with the HJ tree, allocator choice can affect both layout and performance. We repeat this experiment with mimalloc instead of jemalloc to see if different layout problems occur. This reveals substantial gaps between database rows (Figure 10a).

These gaps are caused by mimalloc’s implementation of `posix_memalign`. If the next free block already has the requested alignment, mimalloc returns it directly. Otherwise, for an allocation of size  $s$  with alignment  $a$ , it obtains a block of size  $s + a$  and returns an aligned pointer within that block. Thus, aligning a 48B `row_t` object to 64B can require a 112B block. This explains the pattern in Figure 10a: since  $112 \times 4 = 448$  is divisible by 64, every fourth block is already aligned and can be returned directly.

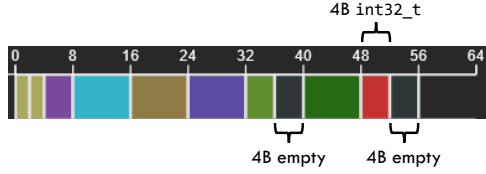
A simple fix is to allocate 64-byte `row_t` objects, eliminating the alignment-induced gaps (Figure 10b). To distinguish its benefit from that of sharing the EBR instance, we test all four combinations with mimalloc and node segregation held fixed. Row padding alone improves mean throughput by 0.5%, sharing the EBR instance by 19.5%, and combining them by 21.1% (as in Table 1). Thus, most of the throughput improvement comes from reclamation, although row padding eliminates the visible fragmentation.

#### 5.4 RocksDB

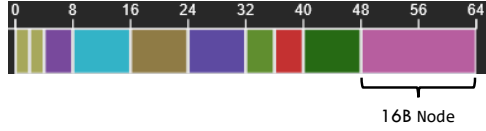
RocksDB [18], Meta’s key-value store, is used by Bing [25], WeChat [41], and TikTok [26]. Its log-structured merge tree accumulates updates in an in-memory *memtable*, then flushes them to sorted files that are periodically merged during compaction. Memtables allocate objects within larger arena buffers. HeapLENS’s lifetime tracking (§4.2) lets us inspect these objects individually.

We use RocksDB’s `db_bench` to investigate **two memtable implementations**. Each test starts with 10 million unique random 32B keys and 128B values. For prefix hash, flushing and compaction are enabled, and we wait for initial compaction before measurement. For the inline skiplist, we keep data in memory to isolate node-layout effects from disk accesses. We measure reader throughput while  $n - 1$  threads perform random lookups and one writes random keys.

The **prefix hash memtable** hashes a fixed-length key prefix to choose a bucket, implemented by a skiplist object.



(a) Before reordering. The grey blocks at 36B and 52B are empty.



(b) After reordering. The pink object at 48B is a node.

**Figure 11.** A bucket object in RocksDB’s HashSkipList memtable (96 threads), expanded to show its fields.

Expanding these bucket objects in *HeapLENS* (Figure 11) exposes compiler-inserted padding between fields. Moving the 4B `int32_t prev_height_` before the 8B `Node** prev_` fills a hole, shrinking the bucket from 56B to 48B. The following node requires 16B alignment, so it begins at byte 64, leaving an 8B gap after the 56B bucket. After reordering, the bucket ends at byte 48, where the node can begin immediately. This saves 8B inside the bucket and another 8B after it. With 95 readers and one writer across both NUMA nodes, interleaved memory, and 256-MiB write buffers, this change improves mean reader throughput by 8.4%. Cache misses per reader operation increase 1.4%, while DTLB misses fall 1.4%. *RocksDB* has already incorporated this change.

Next, we consider the **inline skiplist memtable**, which uses a single skiplist and stores key/value pairs together with their nodes. *HeapLENS* reveals that tall nodes, which contain many forward pointers, cross more cache lines than their size requires. These nodes are frequently traversed during searches, making unnecessary crossings particularly undesirable. **Aligning tall nodes** to cache-line boundaries minimizes the number of lines they span. Tall nodes are also interspersed with much more numerous short nodes, spreading them across many pages. **Segregating tall nodes** into their own memory region concentrates them in fewer pages. With 19 readers and one writer on physical cores 0–19 and local memory, these changes increase mean reader throughput by 8.5%, reduce cache misses per reader operation by 2.9%, and reduce DTLB misses by 2.2%. The prefix-hash and inline-skiplist measurements both use *jemalloc* and last 10 and 60 seconds, respectively.

## 5.5 Using HeapLENS with an LLM

*HeapLENS* was designed primarily for performance experts, but its observations could also help an LLM investigate layout problems and test optimizations independently. To support such investigations, *HeapLENS* exports text descriptions of representative pages at eight equally spaced execution

times. The export includes object types, cache-line crossings, objects constructed inside larger allocations, types sharing cache lines, and cache-set occupancy by type.

**Comparison and procedure.** We compare two LLM investigations: one receives a *HeapLENS* export and a copy of this paper with all LLM material removed; the other receives neither. Both inspect source, report suspected layout problems, recommend a sequence of changes, implement them, and A/B test them. We prepare instrumentation and allocation data in a separate repository and measure performance using uninstrumented application variants. Each investigation uses its own optimization repository and begins in a fresh session without history or memory from the other.

Both use Codex 0.137, GPT-5.5, extra-high reasoning, and priority/fast mode. The analysis prompts differ in whether they refer to *HeapLENS* inputs, while recommendation, implementation, and testing prompts are shared. Written reports carry findings between stages. Occasional follow-up prompts ask the LLM to continue the investigation. When the LLM without *HeapLENS* pursued a sweeping redesign, we asked it to test alternatives requiring fewer changes.

**Valkey string cache.** Valkey is a Redis-compatible C data store (about 420,000 source lines). The workload generator Memtier [32] adds about 60,000 source lines. The workload preloads four million string keys (10–16 bytes) with 128-byte values, then runs 80% GET/20% SET for 30 seconds. Valkey uses *jemalloc* with persistence disabled. Twenty-four server I/O threads use NUMA node 0 (CPUs 0–23), and 24 Memtier threads use node 1 (CPUs 24–47), with four connections per client thread and pipeline depth 16. Requests use RESP over TCP loopback.

*HeapLENS* exposed frequent cache-line crossings in `robj` objects and `rax` radix-tree nodes. Using this information, the LLM produced a 39-line patch aligning small object headers and radix-tree nodes to cache lines, while preserving object semantics and network/persistence formats. Across ten interleaved trials per variant, aligning only `robj` improves mean throughput by 3.6%; also aligning `rax` nodes raises the gain to 4.2% (95% paired-block bootstrap interval 3.7–4.7%). The combined patch increases mean post-run RSS by 9.4%, from 952.9 to 1042.7 MiB, while mean per-run p99 latency falls 4.0% to 3.2 ms. Same-run performance counters show 3.2% fewer cache misses/op but 1.6% more DTLB misses/op.

Without *HeapLENS*, the LLM proposed a nine-stage search ranging across workload boundaries, string-update semantics, embedding thresholds, allocator tuning, hash-table density, and a custom allocator. After the request for a smaller alternative change, it selected same-size string-value reuse and *jemalloc* tuning. Its reuse shortcut failed an APPEND correctness test, so we repaired it to preserve a required embedded-to-raw string conversion. On the same workload, the repaired unguided candidate reduces mean post-run RSS to 851.3 MiB but **lowers** throughput by 1.2%, compared with

the guided patch’s 4.2% gain. The larger string-embedding redesign is out of scope, because it changes ownership, defragmentation, persistence, and module interfaces.

**HNSWLib.** HNSWLib constructs a graph over one million random float32 vectors, with separate 128- and 1536-dimensional workloads. Each trial builds a fresh index, then performs 10,000 warmup queries followed by five repetitions of the same 100,000 queries using vectors sampled from the index. Both construction and queries use 24 threads. The graph’s neighbour-limit parameter is  $M = 16$ , construction search breadth is 200, query search breadth is  $ef = 64$ , and  $k = 10$  neighbours are requested. Trials use the libc allocator (because it was faster than jemalloc), physical cores 0–23, and node-0 memory. Query throughput is 500,000 divided by total query time.

HNSWLib stores each vector together with its graph links in a record. The export showed that the vector begins at byte offset 132 within this record. The vectors therefore do not consistently begin at cache-line boundaries. A 128-dimensional vector occupies 512 bytes and would fit in eight cache lines if aligned, but spans nine when its start is misaligned. Distance calculations read the vector’s coordinates, so these extra crossings can require additional cache-line accesses. The LLM’s patch moves vectors into separate storage and aligns each vector to a cache-line boundary. It also requests huge pages for this storage to reduce address-translation overhead. Table 1 reports mean query-throughput gains of 9.8% at 128 dimensions and 6.1% at 1536 dimensions, with DTLB misses per query falling 88.9% and 47.3%, respectively.

Without HeapLENS, the LLM also identified the vector’s misaligned offset, but prioritized adding padding or changing alignment while keeping vectors and graph links together in the same records. It mentioned separating vector storage, but ranked this behind smaller local changes. The LLM found that some lists of neighbouring nodes were allocated one extra byte that was never used. Removing this byte was its best tested change. Median query throughput improved by 0.4% across three trials at 128 dimensions.

**Separating the HNSW effects.** To distinguish between separation/alignment and huge-page use, we test all four combinations ten times at each dimension, varying their execution order. We request huge pages before first accessing the memory and verify their backing. Table 6 shows that both changes improve throughput individually, but huge pages help more when vectors are separated and aligned.

To measure the benefit of alignment, we place vectors in separate storage and compare two starting positions: aligned to a cache-line boundary or offset by 32 bytes. Huge-page advice is disabled. We run six groups of trials, each testing both variants and the original layout that stores vectors together with graph links. Alignment improves throughput over the 32-byte-offset variant by 9.1% at 128 dimensions and 1.5% at 1536 dimensions (95% intervals 8.9–9.3% and

| Variant                        | 128 D | 1536 D |
|--------------------------------|-------|--------|
| Packed baseline (thousand QPS) | 66.2  | 7.2    |
| Separate + align               | +3.2% | +2.3%  |
| Huge-page advice only          | +2.6% | +1.8%  |
| Combined                       | +9.7% | +6.6%  |

**Table 6.** HNSW factors, measured independently of Table 1. Gains compare arithmetic 10-trial means with the baseline.

1.2–1.8%). We compute these gains as geometric means of throughput ratios within trial groups.

Separating vectors without aligning them does not consistently help. Compared with the original layout, the 32-byte-offset variant reduces throughput by 5.3% at 128 dimensions and improves it by 0.8% at 1536 dimensions. This comparison changes both storage organization and alignment: separated vectors all begin 32 bytes into a cache line, whereas vectors in the original layout begin at varying positions.

## 6 Conclusion

HeapLENS makes memory-layout anti-patterns visible in multithreaded C/C++ applications. It exposes objects hidden within larger allocations and reduces vast allocation histories to representative page layouts that developers can inspect interactively. Whole-page sampling preserves neighbouring-object context, while fingerprints of page histories and live objects distinguish layouts that differ in space and time. Our evaluation shows that these techniques preserve the tested layout patterns within practical display budgets.

Inspecting these layouts led to performance improvements through changes in alignment, field order, allocation, and reclamation, including a fix incorporated into RocksDB. Text exports also helped an LLM independently investigate and optimize Valkey and HNSWLib. Together, these results demonstrate the value of making actual object placement and lifetimes accessible to both developers and automated tools. See our artifact [7] for HeapLENS and reproduction scripts.

## Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant 2019-04227, Discovery Launch Supplement 2019-00048, and the University of Waterloo. Computing infrastructure used in this work was funded by NSERC Collaborative Research and Development Grant 539431-19 and the Canada Foundation for Innovation John R. Evans Leaders Fund (38512), with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund. Sean Ovens was supported by an NSERC Postdoctoral Fellowship (PDF-578266-2023). We thank Abhirup Das for his contributions as an intern, the artifact evaluators for their excellent feedback, and Prof. Demke-Brown for shepherding the paper.

## References

- [1] Yehuda Afek, Dave Dice, and Adam Morrison. 2011. Cache index-aware memory allocation. In *Proceedings of the International Symposium on Memory Management* (San Jose, California, USA) (ISMM '11). Association for Computing Machinery, New York, NY, USA, 55–64. doi:10.1145/1993478.1993486
- [2] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization* (Salt Lake City, Utah, USA) (SOFTVIS '10). Association for Computing Machinery, New York, NY, USA, 53–62. doi:10.1145/1879211.1879222
- [3] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) (OOPSLA 2015). Association for Computing Machinery, New York, NY, USA, 451–469. doi:10.1145/2814270.2814294
- [4] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 295–306. <https://www.usenix.org/conference/atc18/presentation/arbel-raviv>
- [5] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. *SIGPLAN Not.* 45, 5 (Jan. 2010), 257–268. doi:10.1145/1837853.1693488
- [6] Trevor Brown. 2019. SetBench. <https://gitlab.com/trbot86/setbench>. Accessed: 2025-08-20.
- [7] Trevor Brown, Sean Ovens, and Gautam Pathak. 2026. HeapLENS: Heap Layout Evaluation & Navigation Suite—ACM ATC 2026 Artifact. doi:10.5281/zenodo.23206284 <https://github.com/trbot86/heaplens-artifact>
- [8] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. 1999. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) (PLDI '99). Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/301618.301633
- [9] A. N. M. Imroz Choudhury, Kristin C. Potter, and Steven G. Parker. 2008. Interactive Visualization for Memory Reference Traces. *Computer Graphics Forum* 27, 3 (2008), 815–822. doi:10.1111/j.1467-8659.2008.01212.x
- [10] Intel Corporation. [n. d.]. *Intel VTune Profiler*. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>
- [11] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronous Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) (ASPLOS '15). Association for Computing Machinery, New York, NY, USA, 631–644. doi:10.1145/2694344.2694359
- [12] Arnaldo Carvalho de Melo. 2010. The New Linux “perf” Tools. In *Linux Kongress*.
- [13] Chen Ding and Ken Kennedy. 2000. Inter-array Data Regrouping. In *Languages and Compilers for Parallel Computing*, Larry Carter and Jeanne Ferrante (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–163.
- [14] Dana Drachler, Martin Vechev, and Eran Yahav. 2014. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) (PPoPP '14). Association for Computing Machinery, New York, NY, USA, 343–356. doi:10.1145/2555243.2555269
- [15] Ulrich Drepper. 2010. What every programmer should know about memory. <http://people.redhat.com/drepper/cpumemory.pdf> (2010).
- [16] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Zurich, Switzerland) (PODC '10). Association for Computing Machinery, New York, NY, USA, 131–140. doi:10.1145/1835698.1835736
- [17] Jason Evans. 2011. *Scalable memory allocation using jemalloc*. <https://engineering.fb.com/2011/01/03/core-infra/scalable-memory-allocation-using-jemalloc/>
- [18] Facebook. 2013. RocksDB: A Persistent Key-Value Store for Fast Storage Environments. <https://github.com/facebook/rocksdb>.
- [19] Sanjay Ghemawat and Paul Menage. [n. d.]. *TCMalloc: Thread-Caching Malloc*. <https://goog-perftools.sourceforge.net/doc/tcmalloc.html>
- [20] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) (PLDI '93). Association for Computing Machinery, New York, NY, USA, 177–186. doi:10.1145/155090.155107
- [21] Shane V. Howley and Jeremy Jones. 2012. A non-blocking internal binary search tree. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Pittsburgh, Pennsylvania, USA) (SPAA '12). Association for Computing Machinery, New York, NY, USA, 161–171. doi:10.1145/2312005.2312036
- [22] Intel Corporation. 2012. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. <https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf#page=62> Order number 248966-026, Section 2.1.5.4, Data Prefetching: Spatial Prefetcher, p. 2-24.
- [23] Daan Leijen, Ben Zorn, and Leonardo de Moura. 2019. *Mimalloc: Free List Sharding in Action*. Technical Report MSR-TR-2019-18. Microsoft. <https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/>
- [24] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) (ICPE '15). Association for Computing Machinery, New York, NY, USA, 51–62. doi:10.1145/2668930.2688037
- [25] Burton (Pu) Li, Jason (Zengzhong) Li, Max Sigalov, Dafan Liu, and Knut Magne Rísvik. 2021. *RocksDB in Microsoft Bing*. <https://blogs.bing.com/Engineering-Blog/october-2021/RocksDB-in-Microsoft-Bing>
- [26] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, Xudong Wang, Huiming Zhu, Xuwei Fu, Tingwei Wu, Hongfei Tan, Hengtian Ding, Mengjin Liu, Kangcheng Wang, Ting Ye, Lei Li, Xin Li, Yu Wang, Chenguang Zheng, Hao Yang, and James Cheng. 2022. ByteGraph: a high-performance distributed graph database in ByteDance. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3306–3318. doi:10.14778/3554821.3554824
- [27] Sergio Moreta and Alexandru Telea. 2007. Visualizing Dynamic Memory Allocations. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 31–38. doi:10.1109/VISSOF.2007.4290697
- [28] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. doi:10.1145/1273442.1250746
- [29] Zhuorui Ning, Naijie Gu, Junjie Su, and Dongsheng Qi. 2022. STAFF: A Model for Structure Layout Optimization. In *2022 7th International Conference on Computer and Communication Systems (ICCCS)*. 115–122. doi:10.1109/ICCCS55155.2022.9846314
- [30] Sean Ovens. 2025. Shrink size of HashSkipList buckets from 56B to 48B. <https://github.com/facebook/rocksdb/pull/13424>. RocksDB pull request #13424.
- [31] Boris Quaing, Jie Tao, and Wolfgang Karl. 2005. YACO: A User Conducted Visualization Tool for Supporting Cache Optimization.

- In *High Performance Computing and Communications*, Laurence T. Yang, Omer F. Rana, Beniamino Di Martino, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 694–703.
- [32] Redis. 2026. memtier\_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. [https://github.com/redis/memtier\\_benchmark](https://github.com/redis/memtier_benchmark). Accessed: 2026-06-11.
  - [33] George G. Robertson, Trishul Chilimbi, and Bongshin Lee. 2010. Al-locRay: memory allocation visualization for unmanaged languages (*SOFTVIS '10*). Association for Computing Machinery, New York, NY, USA, 43–52. doi:10.1145/1879211.1879221
  - [34] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2006. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th International Symposium on Memory Management* (Ottawa, Ontario, Canada) (*ISMM '06*). Association for Computing Machinery, New York, NY, USA, 84–94. doi:10.1145/1133956.1133968
  - [35] E. van der Deijl, G. Kanbier, O. Temam, and E.D. Granston. 1997. A cache visualization tool. *Computer* 30, 7 (1997), 71–78. doi:10.1109/2.596631
  - [36] Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time (*ICPE '19*). Association for Computing Machinery, New York, NY, USA, 29–32. doi:10.1145/3302541.3313100
  - [37] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *2020 Working Conference on Software Visualization (VISSOFT)*. 110–121. doi:10.1109/VISSOFT51673.2020.00017
  - [38] Louis Ye, Mieszko Lis, and Alexandra Fedorova. 2019. A unifying abstraction for data structure splicing. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) (*MEMSYS '19*). Association for Computing Machinery, New York, NY, USA, 173–183. doi:10.1145/3357526.3357548
  - [39] Chao Yu, Probir Roy, Yuebin Bai, Hailong Yang, and Xu Liu. 2018. LWPTool: A Lightweight Profiler to Guide Data Layout Optimization. *IEEE Transactions on Parallel and Distributed Systems* 29, 11 (2018), 2489–2502. doi:10.1109/TPDS.2018.2840992
  - [40] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: an evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. doi:10.14778/2735508.2735511
  - [41] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. 2017. PaxosStore: high-availability storage made practical in WeChat. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1730–1741. doi:10.14778/3137765.3137778
  - [42] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) (*PLDI '04*). Association for Computing Machinery, New York, NY, USA, 255–266. doi:10.1145/996841.996872