

WHITE PAPER · DATABASE CACHING

Non-Volatile Memory, NVMe SSDs and Hybrid Tiered Caching

Caches that span DRAM, CXL-attached memory and flash

White Paper 4 · Database Caching series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

Non-Volatile Memory, NVMe SSDs and Hybrid Tiered Caching

White Paper 4 · Database Caching series · Bassirah Technology — Innovation & R&D · 26 October 2026

Document control

Field	Value
Series	Database Caching, paper 4 of 6
Author	Bassirah Technology — Innovation & R&D
Version	1.0 · 26 October 2026
Scope	A technology survey. It describes published research and publicly documented systems; it does not report Bassirah benchmarks or customer results.

Contents

1. Abstract	4
2. Introduction	4
2.1 Why caches span more than DRAM	4
2.2 Non-volatile memory is not NVMe	4
2.3 How this paper builds on the series.....	4
2.4 Scope.....	5
2.5 Terms used in this paper	5
3. The hierarchy and the economics of placement	5
3.1 The tiers	5
3.2 The five-minute rule	5
4. Persistent memory: what Optane taught	6
4.1 The hardware.....	6
4.2 Databases and key-value stores.....	6
4.3 Crash consistency	6
4.4 The end of the product line	6
5. CXL-attached memory	7
5.1 What CXL provides.....	7
5.2 Expansion, pooling and their economics.....	7
5.3 Databases on CXL memory	7
6. Placing hot and cold data across memory tiers	7
7. NVMe flash as a cache tier	8
7.1 How flash behaves.....	8
7.2 Hybrid DRAM and flash caches	8
7.3 Caching tiny objects on flash.....	8
7.4 Admission for endurance	9
7.5 Host placement: Flexible Data Placement and Zoned Namespaces.....	9

8. Database buffer pools over NVMe	9
9. Tiering in caching products	10
10. Persistence and warm restart	10
11. Tiered KV caches for language-model serving	11
12. Comparative analysis.....	11
13. Open challenges and future opportunities.....	11
13.1 Endurance versus hit ratio	11
13.2 Application-level write amplification	12
13.3 Placement across three or more tiers.....	12
13.4 CXL maturity and economics	12
13.5 Warm restart and cold start	12
13.6 Cost models that keep moving.....	12
13.7 Crash consistency on persistent tiers	12
13.8 Tiered caches for AI workloads	12
14. What this means for platform decisions	13
References	13

1. Abstract

DRAM is fast and expensive, and caches built only from DRAM become costly as working sets grow. Hybrid tiered caches keep the hottest data in DRAM and place warmer data on cheaper tiers: memory attached over Compute Express Link (CXL), byte-addressable persistent memory where it exists, and NVMe flash. This paper examines how such caches are built and what each tier demands. It explains the difference between non-volatile memory and NVMe, uses the five-minute rule to frame when data belongs in each tier, and summarizes what a generation of research on Intel Optane persistent memory taught before that product was wound down. It describes CXL memory expansion and pooling, the operating-system and database techniques that place hot and cold pages across memory tiers, and the behavior of flash that shapes flash caches: write amplification, endurance, and the host-placement interfaces Flexible Data Placement and Zoned Namespaces. It then covers hybrid DRAM and flash cache designs, database buffer pools over NVMe, product approaches to tiering, warm restart, and tiered KV caches in language-model serving. It closes with a qualitative comparison, open challenges and notes for platform decisions.

2. Introduction

2.1 Why caches span more than DRAM

Memory is a major driver of both performance and hardware cost in large platforms; cloud providers describe DRAM as a large share of server cost and report that much provisioned memory goes unused [1], [2]. Per-gigabyte DRAM prices have also stopped falling as quickly as they once did [3]. Meanwhile, NVMe SSDs have become inexpensive and offer high throughput, especially with several drives per server [4], [5]. A cache that keeps its hottest data in DRAM and its warm data on a cheaper tier can hold a much larger working set at a given cost. The engineering problem is to decide what lives where, to move data between tiers without stalling requests, and to respect the physical limits of each tier, above all the finite write endurance of flash.

2.2 Non-volatile memory is not NVMe

The two terms are often confused. **Non-volatile memory (NVM)**, in the sense used here, is byte-addressable memory that keeps its contents without power and is accessed with ordinary load and store instructions; Intel Optane persistent memory was the main commercial example [6], [7]. **NVMe** is a family of specifications for accessing storage devices, most often flash SSDs, over PCIe and network transports [8]. Flash behind NVMe is accessed in blocks, erased in larger units and worn by writes [9]. This paper covers both, together with CXL-attached memory, which is volatile DRAM reached over a different interconnect [10].

2.3 How this paper builds on the series

Paper 1 described caching strategies, eviction and invalidation; paper 2 described learned admission, including admission for flash caches. Paper 3 described KV-cache reuse in language-model serving, whose tiered storage returns briefly in Section 11. Paper 2 of the Data Storage Engines series described LSM-tree engines, whose write amplification is closely related to the flash behavior in Section 7.

2.4 Scope

This paper covers tiered caching across DRAM, CXL-attached memory, persistent memory and NVMe flash, in application caches, key-value caches and database buffer managers. It concentrates on peer-reviewed work from 2017 to 2026 and on official documentation, and cites foundational work where it introduced an idea still in use. Preprints are identified as such. Statements about products reflect documentation accessed on 26 October 2026. It does not report latency, bandwidth, throughput, capacity, endurance or cost figures; where a paper reports results, we describe them qualitatively and attribute them to the authors.

2.5 Terms used in this paper

A **tier** is a class of memory or storage with its own cost, speed and capacity. **Promotion** moves data to a faster tier and **demotion** to a slower one. **Write amplification** is the ratio of bytes written to flash media to bytes the host asked to write; it is caused by garbage collection inside the device and by rewriting in the application [11], [12]. **Endurance** is the amount of data that can be written to a flash device before it can no longer store data reliably [9].

Overprovisioning is spare flash capacity reserved to reduce write amplification. A **warm restart** brings a cache back with its contents after a process restart rather than empty.

3. The hierarchy and the economics of placement

3.1 The tiers

Table 1 summarizes the tiers a cache can use. The descriptions are qualitative and relative; actual behavior depends on the device, platform and workload.

Tier	Access model	Persistence	Main strengths	Main constraints
Local DRAM	Load and store	Volatile	Very low access cost; no wear	Cost per byte; capacity per socket
CXL-attached memory	Load and store over CXL	Volatile (as typically deployed)	Adds capacity and bandwidth beyond the CPU's own memory channels	Slower than local DRAM; software and platform maturity
Persistent memory (Optane PMem)	Load and store; persistent mode	Persistent	Byte-addressable persistence	Product line wound down; crash-consistency burden on software
NVMe flash	Block I/O	Persistent	Low cost per byte; high throughput with parallel I/O	Finite endurance; write amplification; block granularity
Remote and object storage	Network I/O	Persistent	Very low cost per byte at scale	Highest access cost; the tier the cache protects

3.2 The five-minute rule

Gray and Putzolu framed the trade-off between memory and slower storage as a break-even interval: an item re-referenced more often than that interval is cheaper to keep in memory than to fetch each time, and the interval follows from the ratio of memory cost to access cost [13]. Because prices and device speeds change, the interval must be recomputed as technology changes, and revisits have traced how it reshaped the storage hierarchy [14]. Two recent revisits extend the reasoning. Duwe et al. recompute it for the cloud, where resources are elastic and billed by use, and find that the right caching decision for object storage differs

sharply between latency-sensitive and latency-tolerant workloads [15]. A recent preprint argues that for hosts with very fast SSDs, flash becomes an active data tier rather than passive storage [16]. For cache design, the lesson is to size each tier from re-reference intervals and current tier economics, for each adjacent pair of tiers, rather than from a rule of keeping everything in DRAM.

4. Persistent memory: what Optane taught

4.1 The hardware

Intel Optane DC Persistent Memory modules sat on the memory bus and could run in a volatile mode, with DRAM acting as a cache in front of them, or in a persistent mode accessed directly by software [6]. Measurements on real hardware showed byte-addressable, persistent memory that was slower than DRAM but far faster than flash for small accesses, and whose performance depended on access size, write pattern and thread count in ways that earlier emulations had not captured [6], [7].

4.2 Databases and key-value stores

Database research explored how to use the new tier. Van Renen et al. showed that using persistent memory alone, or as a page store behind a DRAM cache, each wastes something, and argued for a three-tier DRAM, persistent memory and SSD buffer manager that loads data from persistent memory at fine granularity [17]. Spitfire found that persistent memory breaks two classic buffer-manager assumptions, that data must be moved into DRAM to be used and that storage is far slower than DRAM, and used tunable, probabilistic migration between tiers whose best setting depends on workload and hardware [18]. Viper showed that a key-value store designed for the medium, with a hash index in DRAM and records in a persistent-memory-aware layout, works better than replacing an SSD with persistent memory under a block-oriented design [19].

4.3 Crash consistency

Persistence in memory moves the burden of crash consistency from the storage stack into the application. A store is not durable until it is flushed from CPU caches, caches can write back in any order, and a crash can leave data partly written; the PMDK libraries provide transactional object stores so that applications need not manage flushing by hand [20]. Systematic testing of persistent-memory key-value stores found many previously unknown correctness and performance bugs, including in PMDK itself [21].

4.4 The end of the product line

In the second quarter of 2022 Intel initiated the wind-down of its Optane memory business, stating that it was not aligned with its strategic priorities and that it expected to meet existing customer commitments [22]. PMDK is now maintained on a best-effort basis, with many libraries discontinued [20]. The research remains useful: its lessons about tier-aware buffer management [17], [18], layouts designed for the medium [19] and crash consistency [20], [21] apply to any slower, larger memory tier, including the CXL memory in Section 5.

5. CXL-attached memory

5.1 What CXL provides

Compute Express Link is an open, cache-coherent interconnect for processors, memory expansion and accelerators [23]. It carries three protocols over the PCIe physical layer: one for I/O, one for device caching and one for memory access [10]. A Type 3 device is a memory expander that the host reaches with ordinary loads and stores, coherent with host memory [10]; to the operating system it usually appears as a NUMA node without CPUs, slower than local DRAM [24], [25]. CXL 2.0 added switching, memory pooling and persistent-memory support; CXL 3.0 added multi-level fabrics and memory sharing, in which one region is coherently visible to several hosts [10]. The consortium presents its current generation as backward compatible with all earlier versions [23].

5.2 Expansion, pooling and their economics

CXL memory can add capacity and bandwidth and decouple memory technology from the CPU's own memory interface [24]. Pond studied pooling in a public cloud: much provisioned memory is stranded or never touched, small CXL pools shared by a few sockets keep access latency low, and machine-learning predictions of how much local memory each virtual machine needs let the platform hide the slower tier from latency-sensitive workloads [1]. The case for pooling is contested: Levis et al. argue against CXL memory pooling in a position paper [26]. Measurements on real CXL devices show that they behave differently from the remote NUMA node used to emulate them in earlier research, so earlier conclusions need revisiting [24].

5.3 Databases on CXL memory

Database researchers have evaluated CXL memory directly. A CXL memory expander inside an in-memory database showed that transactional workloads were barely affected and analytical workloads only modestly, presenting CXL as a lower-cost way to expand in-memory capacity [27]. A later study found that a hardware-managed mode that tiers local DRAM and CXL memory at cache-line granularity can exploit locality in analytical workloads, with results depending on access patterns and data placement [28]. Lerner and Alonso argue that CXL's larger impact may be architectural, enabling databases to scale up by adding individual resources rather than whole servers [29]. Hao et al. compare ways a buffer manager and index can use a far-memory tier between DRAM and SSD and find that no single design wins on every metric [3].

6. Placing hot and cold data across memory tiers

The operating system and runtime research on tiered memory is, in effect, research on two-level caches whose unit is a page. Its lessons apply to any tiered cache.

TPP, developed for CXL-attached memory, found that many applications in Meta's fleet have a sizeable share of cold pages; it demotes cold pages proactively, keeps headroom in local DRAM for new allocations, and promotes hot pages back with low overhead, and most of its changes were merged into the mainline Linux kernel [2]. Linux now groups memory into tiers, demotes pages from a fast tier to a slower one under memory pressure before swapping, promotes frequently used pages through NUMA balancing, and allows promotion to be rate-

limited because migration traffic itself hurts latency [25]. HeMem tracks hot pages by sampling hardware performance counters rather than scanning page tables and migrates pages asynchronously, so applications do not stall [30]. Memtis sizes the hot set from the access distribution of all pages rather than fixed thresholds and splits huge pages whose access is skewed [31]. Nomad keeps a copy of recently promoted pages in the slow tier, so demotion is cheap and pages do not thrash under memory pressure, the same trade-off as inclusive versus exclusive cache hierarchies [32]. Colloid shows that placing the hottest pages in the nominally fastest tier can be wrong under load, because queueing raises that tier's real latency, and balances measured access latencies across tiers instead [33].

For cache designers the lessons are direct: measure hotness cheaply, move data off the request path, size the fast tier from the access distribution, consider keeping copies to make demotion cheap, rate-limit migration, and place data by loaded latency rather than nominal device speed.

7. NVMe flash as a cache tier

7.1 How flash behaves

Flash is written in pages and erased in larger blocks, so a device cannot overwrite data in place; it writes new data elsewhere and later reclaims blocks through garbage collection, copying still-valid pages as it goes [9]. That copying causes write amplification, which consumes endurance and performance; it depends on overprovisioning and on how the device chooses blocks to reclaim, and separating data with different lifetimes reduces it [12]. He et al. describe the implicit contract that SSD clients must follow for good performance, including issuing large or parallel requests and grouping data that will be discarded together, so that garbage collection need not copy it [34]. Log-structured designs, which write sequentially and clean segments later, descend from the log-structured file system [35] and underlie most flash caches. TRIM and deallocation tell the device which data is no longer needed, which helps keep write amplification down [9].

7.2 Hybrid DRAM and flash caches

A hybrid cache keeps a DRAM tier and spills items evicted from it to flash, promoting them back on access. CacheLib, the caching engine behind many services at Meta, supports such hybrid caches to hold large working sets [36]. Its flash engine routes items by size: a small-object cache uses a set-associative hash on the device with a Bloom filter to avoid needless reads and a minimal DRAM index, while a large-object cache appends to large regions in a log-structured layout with a compact DRAM index [37]. Earlier designs followed the same principles; Twitter's fatcache kept an index in DRAM, stored keys and values on SSD, and appended writes in a log so that a hit needed one SSD read [38].

7.3 Caching tiny objects on flash

Tiny objects are hard to cache on flash. A log-structured layout needs a DRAM index entry per object, which becomes expensive with billions of objects, while a set-associative layout rewrites a whole flash page for each small insert, which wears the device. Kangaroo combines a small log, which batches tiny objects, with a large set-associative store, so that each set rewrite carries many objects and both DRAM use and flash writes stay low [39]. Nemo, a recent preprint, argues that even on devices with low device-level write amplification, rewriting whole

sets for small inserts still dominates, and proposes ways to raise set fill and reduce index memory [40].

7.4 Admission for endurance

Because every write consumes endurance, deciding what enters the flash tier matters as much as what leaves it. Flashield uses DRAM as a filter and admits to flash only objects predicted to be read often and rarely updated, writing them in large batches [41]. Baleen learns admission and prefetching for flash caches in front of disk storage, under a flash write budget, optimizing backend disk time rather than miss rate [42]. Paper 2 discusses learned admission in more detail.

7.5 Host placement: Flexible Data Placement and Zoned Namespaces

New NVMe interfaces let the host tell the device which data belongs together, so that data with different lifetimes does not share erase blocks. Zoned Namespaces (ZNS) expose zones that must be written sequentially and reset explicitly; the host gains control but takes on sequential-write discipline and its own garbage collection [43]. A zone-optimized version of CacheLib, described in a preprint, shows the engineering involved [44], and a study of cache workloads finds that the benefit of ZNS depends on the workload [45]. Flexible Data Placement (FDP), ratified by NVM Express, is a lighter model: writes carry a placement handle that steers them to a reclaim unit, the device still performs garbage collection, and software unaware of FDP runs unchanged [11]. Allison et al. used FDP to separate CacheLib's small-object and large-object writes, reducing device write amplification and easing the trade-off between overprovisioning and endurance [46], and a recent preprint examines FDP for databases such as MySQL and RocksDB [47]. FairyWREN goes further, co-designing a cache's admission and eviction with flash garbage collection on such interfaces to cut total writes, extend device life and reduce cost and embodied carbon [48]. Host placement fixes write amplification at the device level; a cache must still avoid its own rewrite overhead [40].

8. Database buffer pools over NVMe

Database buffer pools are DRAM caches over storage, and fast NVMe changes how they should be built. LeanStore showed that a lightweight buffer manager can stay close to in-memory performance when data fits in DRAM while paging larger data sets to flash [49]. Umbra applied the idea to evolve an in-memory database into an SSD-based one, combining a large in-memory buffer with fast SSDs [5]. Haas and Leis found that existing out-of-memory systems use only a fraction of what modern NVMe drives can do, and that asynchronous, highly parallel I/O with lightweight buffer management closes the gap [4]. Virtual-memory-assisted buffer management uses hardware address translation while keeping the database in control of faulting and eviction [50]. Established engines adapt their settings to flash: MySQL's documentation advises turning off neighbor-page flushing, an optimization for spinning disks, and enabling TRIM for SSD storage [51]. Storage engines also tier internally. RocksDB can extend its DRAM block cache with a secondary cache on non-volatile media, including a CacheLib-based implementation [52], and can place its coldest data on a cheaper medium by temperature [53]; both features are labeled experimental.

9. Tiering in caching products

Products take different approaches to what stays in DRAM, and the difference matters for sizing. Table 2 summarizes four documented approaches.

Product feature	What stays in DRAM	What goes to flash	Notes from the documentation
Redis Flex (Redis Software and Redis Cloud)	Key indexes, dictionaries and the hot working set	Warm keys and values	Replaces Auto Tiering; not a durable store, so persistence still needs RDB or AOF; locally attached flash required
Redis Auto Tiering (earlier versions)	All keys and hot values	Warm values	Supported up to Redis 7.4 databases
Amazon ElastiCache data tiering (Valkey or Redis OSS)	All keys and hot values	Warm values	A managed-service feature; values move back to memory on access
Memcached extstore	Hash table, keys, metadata and a pointer per item	Values above a size threshold	A cache, not a store; a restart effectively empties flash
Aerospike hybrid memory	Primary index	Record data	A database rather than a cache; other storage modes available

Redis Flex is generally available in Redis Software from version 8.0.2 and is the tiering option for Redis 8.0 databases; it keeps hot data in RAM, moves warm data to locally attached flash, promotes data read from flash back to RAM, and suits workloads whose working set is much smaller than the data set [54]. Its documentation notes that Redis Search on Flex is in beta and that some capabilities, such as Active-Active, are not supported [54]. Memcached's extstore keeps keys in RAM and flushes larger values from the cold end of its LRU through write buffers into sequentially written pages [55]. ElastiCache data tiering moves infrequently accessed values to local SSD when DRAM fills and back on access, while keys stay in memory [56]. Aerospike's default architecture keeps the primary index in memory and record data on SSD [57]. When keys stay in DRAM, key size and key count bound how much data a node can hold; when keys move too, as in Flex, the DRAM budget shifts to indexes and the hot set.

10. Persistence and warm restart

A large cache that restarts empty can take a long time to refill, and the backend must absorb the misses in the meantime; paper 1 described how this can trigger overload. CacheLib keeps its DRAM cache in shared memory so that a restarted process on the same machine reattaches to a full cache, falling back to a cold start after an unclean shutdown; its authors report that hit ratios suffer badly without warm restart and that large hybrid caches can take days to warm again [36], [58]. Redis reloads data from snapshots or an append-only log at startup, so its restart is a reload from disk whose time grows with the data set [59]. Memcached's extstore, by contrast, treats flash as cache space that a restart effectively empties [55]. Warm-restart behavior is therefore a property to check before relying on a tiered cache.

11. Tiered KV caches for language-model serving

The same tiering pattern now appears in language-model serving, where attention KV caches are large and expensive to recompute. CachedAttention keeps the KV caches of conversation history in a hierarchy of memory and storage and overlaps slow-tier access with computation [60]. Mooncake pools the CPU memory, DRAM and SSD of a serving cluster into a disaggregated KV cache, trading storage for recomputation [61]. LMCache moves KV caches from accelerator memory to CPU memory, local SSD and remote stores, including Redis and Valkey [62]. Paper 3 of this series describes these systems in the context of caching for LLM applications; here they show that DRAM, flash and remote tiers serve AI workloads by the same principles as database caches.

12. Comparative analysis

Table 3 compares the design approaches in this paper. The ratings are qualitative and relative to each other; actual behavior depends on the hardware, software versions and workload.

Approach	Capacity per cost	Access cost relative to DRAM	Endurance concern	Software effort	Best-fit workloads
DRAM-only cache	Low	Baseline	None	Low	Working sets that fit in memory
DRAM plus CXL memory with OS tiering	Medium	Low to Medium	None	Low to Medium (platform support)	Memory-bound services with many cold pages
DRAM plus persistent memory	Medium	Low to Medium	Low	High (crash consistency)	Historical; lessons apply to slower memory tiers
Hybrid DRAM and flash cache	High	Medium for flash hits	High; needs admission control	Medium to High	Large working sets with skewed access
Flash cache with FDP or ZNS placement	High	Medium for flash hits	Reduced	High (ZNS) to Medium (FDP)	Write-heavy flash caches at scale
Database buffer pool over NVMe	High	Medium for misses	Medium	Built into the engine	Data sets larger than memory
Product tiering (Redis Flex, extstore, data tiering)	High	Medium for flash hits	Managed by the product	Low (configuration)	Working set much smaller than the data set

13. Open challenges and future opportunities

13.1 Endurance versus hit ratio

Challenge. Every write to a flash tier consumes endurance, so admitting more data can raise hit ratio and shorten device life [42], [9]. **Research direction.** Admission policies with write budgets [41], [42] and co-design of caching with flash garbage collection [48]. **Opportunity.** Caches that treat device wear as a first-class objective would lower cost and embodied carbon [48], [46].

13.2 Application-level write amplification

Challenge. Host placement reduces device-level write amplification, but a cache that rewrites whole sets for small inserts still writes much more than it stores [40]. **Research direction.** Log-plus-set designs [39], higher set fill and compact indexes [40]. **Opportunity.** Tiny-object caches with low write amplification and low DRAM use would make flash practical for the smallest items.

13.3 Placement across three or more tiers

Challenge. With DRAM, CXL memory and flash in one system, placement decisions multiply, and placing the hottest data in the nominally fastest tier can be wrong under load [33].

Research direction. Probabilistic and tunable migration [18], workload-specific provisioning of memory tiers [3], and latency-balancing placement [33]. **Opportunity.** Unified placement that spans memory and flash tiers would replace separate, uncoordinated policies.

13.4 CXL maturity and economics

Challenge. Real CXL devices behave differently from emulation [24], and the economics of pooling are contested [1], [26]. **Research direction.** Measurement on real devices [24], database designs for CXL [27], [29], [28], and software support for tiering in the operating system [25]. **Opportunity.** Clear guidance on when CXL expansion or pooling pays would let platform teams adopt it with confidence.

13.5 Warm restart and cold start

Challenge. Large tiered caches can take a long time to refill after a restart, and a cold cache shifts load to the backend [36]. **Research direction.** Shared-memory reattachment [58], persistent flash tiers that survive restarts, and controlled warm-up. **Opportunity.** Tiered caches that restart warm by default would remove a common cause of incidents.

13.6 Cost models that keep moving

Challenge. The break-even point between tiers changes with prices, device speeds and cloud billing [13], [15]. **Research direction.** Revisits of the five-minute rule for the cloud [15] and for very fast SSDs [16]. **Opportunity.** Tools that recompute tier sizing from live prices and access traces would keep caches cost-effective as hardware changes.

13.7 Crash consistency on persistent tiers

Challenge. Byte-addressable persistence moved crash consistency into applications, where bugs were common [20], [21]. **Research direction.** Systematic crash testing [21] and transactional libraries [20]. **Opportunity.** If persistent memory returns, for example over CXL [10], these tools and lessons would shorten the path to reliable software.

13.8 Tiered caches for AI workloads

Challenge. KV caches for language models are large, and moving them between tiers competes with serving [61], [60]. **Research direction.** KV-cache layers that span accelerator memory, CPU memory, SSD and remote stores [62], [61]. **Opportunity.** The tiering techniques of database caching, including admission, endurance-aware placement and warm restart, can make AI serving cheaper.

14. What this means for platform decisions

Start by measuring the working set and the re-reference intervals of the data, and size the DRAM tier from them; use the five-minute reasoning to decide what the next tier should hold. Consider a flash tier when the working set is much smaller than the data set and flash hits are acceptable for the workload, and plan for endurance: control admission, prefer write-friendly layouts, and consider devices with Flexible Data Placement where the software supports it. Treat CXL memory as an emerging option to evaluate on real devices with operating-system tiering enabled. Check how each product keeps keys and values, how it behaves on restart, and whether it is durable, and keep persistence and failover separate from the cache tier.

For tiered in-memory caching, Bassirah works with Redis as a Strategic Technology Partner; Redis Flex keeps hot data in RAM and warm data on locally attached flash, with the limitations its documentation states. Other products and projects named in this paper are ecosystem references. In each case, validate the design on representative workloads, failure scenarios and versions before relying on it.

References

1. Li, H., Berger, D. S., Hsu, L., Ernst, D., Zardoshti, P., Novaković, S., Shah, M., Rajadnya, S., Lee, S., Agarwal, I. et al. (2023). Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), 574-587. (<https://doi.org/10.1145/3575693.3578835>)
2. Maruf, H. A., Wang, H., Dhanotia, A., Weiner, J., Agarwal, N., Bhattacharya, P., Petersen, C., Chowdhury, M., Kanaujia, S. and Chauhan, P. (2023). TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), 742-755. (<https://doi.org/10.1145/3582016.3582063>)
3. Hao, X., Zhou, X., Yu, X. and Stonebraker, M. (2024). Towards Buffer Management with Tiered Main Memory. Proceedings of the ACM on Management of Data, 2(1) (SIGMOD 2024), 1-26. (<https://doi.org/10.1145/3639286>)
4. Haas, G. and Leis, V. (2023). What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. Proceedings of the VLDB Endowment, 16(9), pp. 2090–2102. (<https://doi.org/10.14778/3598581.3598584>)
5. Neumann, T. and Freitag, M. (2020). Umbra: A Disk-Based System with In-Memory Performance. 10th Conference on Innovative Data Systems Research (CIDR 2020). (<https://www.cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf>)
6. Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y. J., Wang, Z., Xu, Y., Dulloor, S. R. et al. (2019). Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv preprint arXiv:1903.05714. (<https://doi.org/10.48550/arXiv.1903.05714>)
7. Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J. and Swanson, S. (2020). An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. 18th USENIX Conference on File and Storage Technologies (FAST '20), 169-182. (<https://www.usenix.org/conference/fast20/presentation/yang>)

8. NVM Express, Inc. (2026). Specifications (NVM Express Base Specification and command-set specifications, NVMe 2.4 family). Official documentation, accessed 26 October 2026. (<https://nvmexpress.org/specifications/>)
9. Hands, J. (2020). Endurance of NVMe®, SAS, and SATA SSDs. SNIA white paper, Storage Networking Industry Association (Solid State Drive Special Interest Group). (https://www.snia.org/sites/default/files/SSSI/NVMe_SAS_SATA_Endurance_White_Paper.pdf)
10. Das Sharma, D. and Agarwal, I. (2022). Compute Express Link 3.0. CXL Consortium white paper. (https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_3.0_white-paper_FINAL.pdf)
11. Sabol, C., Stenfort, R. and Allison, M. (2023). Flexible Data Placement: State of the Union [and TP4146a Flexible Data Placement (FDP)]. NVM Express presentation, Flash Memory Summit, August 2023. (<https://nvmexpress.org/wp-content/uploads/FMS-2023-Flexible-Data-Placement-FDP-Overview.pdf>)
12. Hu, X.-Y., Eleftheriou, E., Haas, R., Iliadis, I. and Pletka, R. (2009). Write amplification analysis in flash-based solid state drives. Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, pp. 1–9. (<https://doi.org/10.1145/1534530.1534544>)
13. Gray, J. and Putzolu, F. (1987). The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87), 395-398. (<https://doi.org/10.1145/38713.38755>)
14. Appuswamy, R., Graefe, G., Borovica-Gajic, R. and Ailamaki, A. (2019). The five-minute rule 30 years later and its impact on the storage hierarchy. Communications of the ACM, 62(11), 114-120. (<https://doi.org/10.1145/3318163>)
15. Duwe, K., Anadiotis, A., Lamb, A., Lersch, L., Leskes, B., Ritter, D. and Tözün, P. (2025). The Five-Minute Rule for the Cloud: Caching in Analytics Systems. 15th Conference on Innovative Data Systems Research (CIDR 2025). (<https://www.vldb.org/cidrdb/2025/the-five-minute-rule-for-the-cloud-caching-in-analytics-systems.html>)
16. Zhang, T., Mailthody, V. S., Sun, F., Ma, L., Newburn, C. J., Zhang, T., Liu, Y., Li, J., Zhong, H. and Hwu, W.-M. (2025). Five-Minute Rule 40 Years Later: A First-Principles Revisit for Modern Memory Hierarchy. arXiv preprint arXiv:2511.03944 (v2, May 2026). (<https://arxiv.org/abs/2511.03944>)
17. van Renen, A., Leis, V., Kemper, A., Neumann, T., Hashida, T., Oe, K., Doi, Y., Harada, L. and Sato, M. (2018). Managing Non-Volatile Memory in Database Systems. Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18), 1541-1555. (<https://doi.org/10.1145/3183713.3196897>)
18. Zhou, X., Arulraj, J., Pavlo, A. and Cohen, D. (2021). Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), 2195-2207. (<https://doi.org/10.1145/3448016.3452819>)
19. Benson, L., Makait, H. and Rabl, T. (2021). Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. Proceedings of the VLDB Endowment, 14(9), 1544-1556. (<https://doi.org/10.14778/3461535.3461543>)
20. PMDK project (Persistent Memory Development Kit). PMDK overview and libpmem pages. pmem.io. Official documentation, accessed 26 October 2026. (<https://pmem.io/pmdk/> and <https://pmem.io/pmdk/libpmem/>)

21. Fu, X., Kim, W.-H., Shreepathi, A. P., Ismail, M., Wadkar, S., Lee, D. and Min, C. (2021). Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), 100-115. (<https://doi.org/10.1145/3477132.3483556>)
22. Intel Corporation (2022). Form 10-Q for the quarterly period ended July 2, 2022. U.S. Securities and Exchange Commission (EDGAR). (<https://www.sec.gov/Archives/edgar/data/50863/000005086322000030/intc-20220702.htm>)
23. CXL Consortium (2026). About CXL. Compute Express Link Consortium. Official documentation, accessed 26 October 2026. (<https://computeexpresslink.org/about-cxl/>)
24. Sun, Y., Yuan, Y., Yu, Z., Kuper, R., Song, C., Huang, J., Ji, H., Agarwal, S., Lou, J., Jeong, I. et al. (2023). Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), 105-121. (<https://doi.org/10.1145/3613424.3614256>)
25. The Linux Kernel documentation (2026). Documentation for kernel.numa_balancing (admin-guide/sysctl/kernel), and sysfs ABI entries /sys/kernel/mm/numa/demotion_enabled and /sys/devices/virtual/memory_tiering/. kernel.org. Official documentation, accessed 26 October 2026. (<https://docs.kernel.org/admin-guide/sysctl/kernel.html>; <https://github.com/torvalds/linux/blob/master/Documentation/ABI/testing/sysfs-kernel-mm-numa>; <https://github.com/torvalds/linux/blob/master/Documentation/ABI/testing/sysfs-kernel-mm-memory-tiers>)
26. Levis, P., Lin, K. and Tai, A. (2023). A Case Against CXL Memory Pooling. Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets '23), 18-24. (<https://doi.org/10.1145/3626111.3628195>)
27. Ahn, M., Chang, A., Lee, D., Gim, J., Kim, J., Jung, J., Rebholz, O., Pham, V. T., Malladi, K. T. and Ki, Y. S. (2022). Enabling CXL Memory Expansion for In-Memory Database Management Systems. Proceedings of the 18th International Workshop on Data Management on New Hardware (DaMoN '22), 1-5. (<https://doi.org/10.1145/3533737.3535090>)
28. Ahn, M., Willhalm, T., Lee, D., May, N., Kim, J., Ritter, D. and Rebholz, O. (2025). Exploiting Locality in Flat Memory with CXL for In-Memory Database Management Systems. Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25), 1-9. (<https://doi.org/10.1145/3736227.3736230>)
29. Lerner, A. and Alonso, G. (2024). CXL and the Return of Scale-Up Database Engines. Proceedings of the VLDB Endowment, 17(10), 2568-2575. (<https://doi.org/10.14778/3675034.3675047>)
30. Raybuck, A., Stamler, T., Zhang, W., Erez, M. and Peter, S. (2021). HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), 392-407. (<https://doi.org/10.1145/3477132.3483550>)
31. Lee, T., Monga, S. K., Min, C. and Eom, Y. I. (2023). MENTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23), 17-34. (<https://doi.org/10.1145/3600006.3613167>)

32. Xiang, L., Lin, Z., Deng, W., Lu, H., Rao, J., Yuan, Y. and Wang, R. (2024). Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24), 19-35.
(<https://www.usenix.org/conference/osdi24/presentation/xiang>)
33. Vuppalapati, M. and Agarwal, R. (2024). Tiered Memory Management: Access Latency is the Key! Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), 79-94. (<https://doi.org/10.1145/3694715.3695968>)
34. He, J., Kannan, S., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. (2017). The Unwritten Contract of Solid State Drives. Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17), pp. 127–144. (<https://doi.org/10.1145/3064176.3064187>)
35. Rosenblum, M. and Ousterhout, J. K. (1992). The design and implementation of a log-structured file system. ACM Transactions on Computer Systems, 10(1), pp. 26–52.
(<https://doi.org/10.1145/146941.146943>)
36. Berg, B., Berger, D. S., McAllister, S., Grosz, I., Gunasekar, S., Lu, J., Uhlar, M., Carrig, J., Beckmann, N., Harchol-Balter, M. and Ganger, G. R. (2020). The CacheLib Caching Engine: Design and Experiences at Scale. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 753–768.
(<https://www.usenix.org/conference/osdi20/presentation/berg>)
37. CacheLib project (Meta). HybridCache; Navy Overview. Official documentation, accessed 26 October 2026.
(https://cachelib.org/docs/Cache_Library_User_Guides/HybridCache ;
https://cachelib.org/docs/Cache_Library_Architecture_Guide/navy_overview)
38. Twitter (2013–2021). fatcache: Memcache on SSD. Official GitHub repository README (archived, read-only), accessed 26 October 2026. (<https://github.com/twitter/fatcache>)
39. McAllister, S., Berg, B., Tutuncu-Macias, J., Yang, J., Gunasekar, S., Lu, J., Berger, D. S., Beckmann, N. and Ganger, G. R. (2021). Kangaroo: Caching Billions of Tiny Objects on Flash. Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), pp. 243–262. (<https://doi.org/10.1145/3477132.3483568>)
40. Yang, X., Tan, T., Hu, J., Gao, C., Liu, M., Jiang, T., Chen, J., Long, L., Lv, Y. and Shu, J. (2026). Nemo: A Low-Write-Amplification Cache for Tiny Objects on Log-Structured Flash Devices. arXiv preprint arXiv:2603.09605. (<https://arxiv.org/abs/2603.09605>)
41. Eisenman, A., Cidon, A., Pergament, E., Haimovich, O., Stutsman, R., Alizadeh, M. and Katti, S. (2019). Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pp. 65–78.
(<https://www.usenix.org/conference/nsdi19/presentation/eisenman>)
42. Wong, D. L.-K., Wu, H., Molder, C., Gunasekar, S., Lu, J., Khandkar, S., Sharma, A., Berger, D. S., Beckmann, N. and Ganger, G. R. (2024). Baleen: ML Admission & Prefetching for Flash Caches. 22nd USENIX Conference on File and Storage Technologies (FAST 24), pp. 347–371.
(<https://www.usenix.org/conference/fast24/presentation/wong>)
43. Bjørling, M., Aghayev, A., Holmberg, H., Ramesh, A., Le Moal, D., Ganger, G. R. and Amvrosiadis, G. (2021). ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. 2021 USENIX Annual Technical Conference (USENIX ATC 21), pp. 689–703.
(<https://www.usenix.org/conference/atc21/presentation/bjorling>)
44. Yang, C., Guo, C., Zhao, M. and Cao, Z. (2024). A Zoned Storage Optimized Flash Cache on ZNS SSDs. arXiv preprint arXiv:2410.11260. (<https://arxiv.org/abs/2410.11260>)

45. Ramsden, J., Cheng, S., Holmberg, H., Björling, M. and Fedorova, A. (2026). When Do Zoned Namespaces SSDs Matter? A Comparative Study of Cache Workloads. Proceedings of the 19th ACM International Systems and Storage Conference (SYSTOR '26), pp. 112–125. (<https://doi.org/10.1145/3793230.3837770>)
46. Allison, M., George, A., Gonzalez, J., Helmick, D., Kumar, V., Nair, R. R. and Shah, V. S. (2025). Towards Efficient Flash Caches with Emerging NVMe Flexible Data Placement SSDs. Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25), pp. 1142–1160. (<https://doi.org/10.1145/3689031.3696091>)
47. Lan, S., Qi, H., He, X., Kandemir, M., González, J. and Shah, V. S. (2026). FDP: The Data Placement Promise of Modern NVMe SSDs. arXiv preprint arXiv:2610.02676. (<https://arxiv.org/abs/2610.02676>)
48. McAllister, S., Wang, Y., Berg, B., Berger, D. S., Amvrosiadis, G., Beckmann, N. and Ganger, G. R. (2024). FairyWREN: A Sustainable Cache for Emerging Write-Read-Erase Flash Interfaces. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pp. 745–764. (<https://www.usenix.org/conference/osdi24/presentation/mcallister>)
49. Leis, V., Haubenschild, M., Kemper, A. and Neumann, T. (2018). LeanStore: In-Memory Data Management beyond Main Memory. 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 185–196. (<https://doi.org/10.1109/ICDE.2018.00026>)
50. Leis, V., Alhomssi, A., Ziegler, T., Loeck, Y. and Dietrich, C. (2023). Virtual-Memory Assisted Buffer Management. Proceedings of the ACM on Management of Data, 1(1) (SIGMOD 2023), pp. 1–25. (<https://doi.org/10.1145/3588687>)
51. Oracle. MySQL 8.4 Reference Manual, section "Optimizing InnoDB Disk I/O". Official documentation, accessed 26 October 2026. (<https://dev.mysql.com/doc/refman/8.4/en/optimizing-innodb-diskio.html>)
52. RocksDB project (Meta). SecondaryCache (Experimental). RocksDB wiki, official documentation, accessed 26 October 2026. ([https://github.com/facebook/rocksdb/wiki/SecondaryCache-\(Experimental\)](https://github.com/facebook/rocksdb/wiki/SecondaryCache-(Experimental)))
53. RocksDB project (Meta). Tiered Storage (Experimental). RocksDB wiki, GitHub. Official documentation, accessed 26 October 2026. ([https://github.com/facebook/rocksdb/wiki/Tiered-Storage-\(Experimental\)](https://github.com/facebook/rocksdb/wiki/Tiered-Storage-(Experimental)))
54. Redis Ltd. Flex databases; Flex and Auto Tiering (Redis Software); Create a Flex database (Redis Cloud). Official documentation, accessed 26 October 2026. (<https://redis.io/docs/latest/operate/rs/flex/> ; <https://redis.io/docs/latest/operate/rs/databases/flash/> ; <https://redis.io/docs/latest/operate/rc/databases/create-database/create-flex-database/>)
55. Memcached project. Flash Storage (extstore). Official documentation, accessed 26 October 2026. (<https://docs.memcached.org/features/flashstorage/> ; the GitHub wiki page <https://github.com/memcached/memcached/wiki/Extstore> now only redirects here)
56. Amazon Web Services. Data tiering in ElastiCache (Amazon ElastiCache User Guide). Official documentation, accessed 26 October 2026. (<https://docs.aws.amazon.com/AmazonElastiCache/latest/dg/data-tiering.html>)
57. Aerospike, Inc. Flexible storage (Database architecture). Official documentation, accessed 26 October 2026. (<https://aerospike.com/docs/database/learn/architecture/hybrid-storage/>)

58. CacheLib project (Meta). Cache persistence. CacheLib user guide. Official documentation, accessed 26 October 2026. (https://cachelib.org/docs/Cache_Library_User_Guides/Cache_persistence/)
59. Redis Ltd. Redis persistence. Redis documentation (Redis Open Source, operate/management). Official documentation, accessed 26 October 2026. (https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/)
60. Gao, B., He, Z., Sharma, P., Kang, Q., Jevdjic, D., Deng, J., Yang, X., Yu, Z. and Zuo, P. (2024). Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. 2024 USENIX Annual Technical Conference (USENIX ATC 24), pp. 111–126. (<https://www.usenix.org/conference/atc24/presentation/gao-bin-cost>)
61. Qin, R., Li, Z., He, W., Cui, J., Ren, F., Zhang, M., Wu, Y., Zheng, W. and Xu, X. (2025). Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. 23rd USENIX Conference on File and Storage Technologies (FAST 25), pp. 155–170. (<https://www.usenix.org/conference/fast25/presentation/qin>)
62. LMCache project. LMCache documentation. Official documentation, accessed 26 October 2026. (<https://docs.lmcache.ai/>)

Product and project names are trademarks of their respective owners. Their mention describes publicly documented technology and does not imply a business relationship, except where Bassirah names a Strategic Technology Partner.