Analysis and Evaluation of Using Microsecond-Latency Memory for In-Memory Indices and Caches in SSD-Based Key-Value Stores

YOSUKE BANDO, Kioxia Corporation, Japan
AKINOBU MITA, Fixstars Corporation, Japan
KAZUHIRO HIWADA, Kioxia Corporation, Japan
SHINTARO SANO, Kioxia Corporation, Japan
TOMOYA SUZUKI, Kioxia Corporation, Japan
YU NAKANISHI, Kioxia Corporation, Japan
KAZUTAKA TOMIDA, Kioxia Corporation, Japan
HIROTSUGU KAJIHARA, Kioxia Corporation, Japan
AKIYUKI KANEKO, Kioxia Corporation, Japan
DAISUKE TAKI, Kioxia Corporation, Japan
YUKIMASA MIYAMOTO, Kioxia Corporation, Japan
TOMOKAZU YOSHIDA, Fixstars Corporation, Japan
TATSUO SHIOZAWA, Kioxia Corporation, Japan

When key-value (KV) stores use SSDs for storing a large number of items, oftentimes they also require large in-memory data structures including indices and caches to be traversed to reduce IOs. This paper considers offloading most of such data structures from the costly host DRAM to secondary memory whose latency is in the microsecond range, an order of magnitude longer than those of DIMM-mounted persistent memory and currently available CXL memory devices. While emerging microsecond-latency memory, such as one based on flash memory, is likely to cost much less than DRAM, it can significantly slow down pointer-chasing on those in-memory data structures of SSD-based KV stores if naively employed, although its impact has not been well studied. This paper analyzes and evaluates the impact of microsecond-level memory latency on the throughput of SSD-based KV operations. Our analysis finds that a well-known latency-hiding technique of software prefetching for long-latency memory from user-level threads is effective for SSD-based KV stores. The novelty of our analysis lies in modeling how the interplay between prefetching and IO affects performance, from which we derive an equation that well explains the throughput degradation due to long memory latency. The model tells us that the presence of IO in KV operations significantly enhances their tolerance to memory latency, and the throughput degradation is expected to be small even if the memory latency extends to a few microseconds, leading to a finding that SSD-based KV stores can be made latency-tolerant without devising new techniques for microsecond-latency memory. To confirm this through experiments, we design a microbenchmark as well as modify existing SSD-based KV stores so that they issue prefetches for long-latency

memory from user-level threads, and run them while placing most of in-memory data structures on FPGA-based memory with adjustable microsecond latency. The results demonstrate that their KV operation throughputs for varying memory latency can be well explained by our model, and the modified KV stores achieve near-DRAM throughputs for up to a memory latency of around 5 microseconds. This suggests the possibility that SSD-based KV stores involving latency-sensitive in-memory data traversal can use microsecond-latency memory as a cost-effective alternative to the host DRAM.

CCS Concepts: • **Hardware** → *Emerging interfaces*; *Memory and dense storage*; *Analysis and design of emerging devices and systems*; • **Information systems** → *Database performance evaluation.*

Additional Key Words and Phrases: CXL, long-latency memory, prefetch, key-value stores, SSD

1 Introduction

Key-value (KV) stores play a vital role in various data center services [1, 9, 16, 20]. As shown in Figure 1(a), some KV stores use SSDs for storing a large number of items beyond the host DRAM capacity, and use in-memory data structures including indices and caches to minimize the number of time-consuming SSD accesses (i.e., IOs) while searching for items [16]. The size of these in-memory data structures often grows as the database size increases, which can dominate the host DRAM usage [29, 65] (80–96% of the memory footprint in our evaluation). Since having large DRAM is costly, it would be beneficial to offload these large in-memory data structures to lower-cost secondary memory such as DIMM-mounted persistent memory (e.g., Intel Optane DCPMM) and memory based on Compute Express Link (CXL) [12]. While the bandwidth of secondary memory may be made to meet application requirements by using multiple memory devices, their longer latency can have a negative performance impact, as has been reported for memory devices with a latency of hundreds of nanoseconds [26, 33]. The issue is likely to be further exacerbated as emerging CXL devices as well as future memory devices may introduce even longer, microsecond-level latency if they employ lower-cost memory media including flash memory [36, 59]. Naive adoption of such long-latency memory as a replacement for DRAM will significantly degrade the KV operation throughput (measured in operations per second, or ops/sec), given that those in-memory data structures typically require pointer-chasing (e.g., tree/linked-list traversal), which is sensitive to memory latency [22–24, 44].

This paper analyzes and evaluates the performance impact of offloading in-memory data structures of SSD-based KV stores from the host DRAM to much slower, microsecond-latency secondary memory. Note that the memory latency we study in this paper is an order of magnitude longer than those of currently available devices as shown in Figure 1(b). There is a spectrum of how secondary memory may be utilized, and one could consider more in-memory-oriented solutions of replacing SSDs by secondary memory [7]. But in this paper, we are interested in lower-cost solutions of replacing the host DRAM, which can account for half of the server cost [33]. In particular, this paper focuses on offloading of entire indices and caches, as this is a latency-sensitive base-case scenario where, in contrast to partial offloading, every pointer dereference for indices and caches is performed on secondary memory. Through analysis and evaluation, we show that SSD-based KV stores can be made tolerant to microsecond-level memory latency using well-known techniques for hiding memory latency: software prefetching of data on microsecond-latency memory from user-level threads [11, 39, 45]. That is, multiple lightweight threads running in the user space
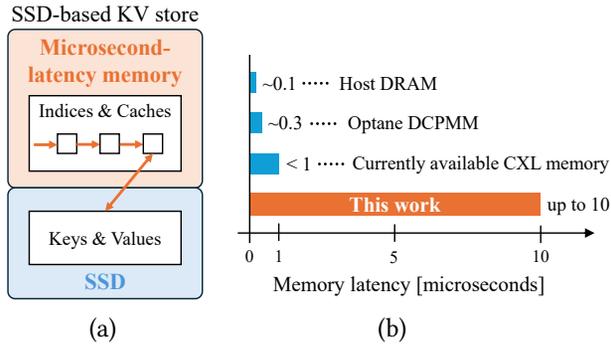
Fig. 1. (a) SSD-based KV stores often perform pointer-chasing on large in-memory indices and caches in addition to SSD access. We offload these in-memory data structures from the host DRAM to much slower, microsecond-latency memory. (b) The memory latency we study is in the microsecond range, an order of magnitude longer than those of currently available memory devices.

process independent KV operations where each thread issues a prefetch to move the desired data to the CPU cache before it actually uses it. To the best of our knowledge, it has not been reported in the literature that SSD-based KV stores with their in-memory indices and caches placed on microsecond-latency memory can still achieve KV throughputs that are close to those when the entire in-memory data is placed on the host DRAM. Our analysis and evaluation show that this is possible without having to devise new techniques.

The novelty of our analysis lies in modeling how the presence of IOs makes SSD-based KV stores more latency-tolerant than they would be without IOs. Prior work has experimentally shown the performance impact of microsecond-latency memory in in-memory settings (i.e., operations consist of memory accesses only), one of whose important observations is that it is generally difficult for latency-sensitive applications to approach DRAM performance if the latency extends to a microsecond level [11]. We extend this prior work by considering operations consisting of memory accesses *as well as IOs*, and we study them experimentally *as well as theoretically*. We introduce a generic operation model of SSD-based KV stores, and derive an equation describing the expected KV throughput as a function of memory latency, leading to a key observation that the presence of IO significantly enhances latency-tolerance. We show that simply adding the IO processing times to the throughput equation only partially explains this latency-tolerance enhancement. Our theoretical analysis reveals how the interplay between memory prefetching and IO leads to further gain in latency-tolerance.

We validate our analysis through experiments. As currently-available memory devices do not have microsecond-level latency, we use FPGA-based CXL memory devices designed to introduce user-specified latency. We first run a microbenchmark that repeats memory accesses and IOs, and show that our throughput equation aligns well with microbenchmark performance. We then show our analysis also applies to some existing SSD-based KV stores if they incorporate the above-mentioned latency-hiding techniques. We take Aerospike [53], RocksDB [17], and CacheLib [8] as representative SSD-based KV stores, and modify them so that they run user-level threads and issue software prefetches before they access microsecond-latency memory. We demonstrate that our model still well explains how the throughputs of these modified KV stores behave as memory latency increases.

In summary, we make the following contributions.

- Analysis: We model how the interplay between prefetching and IO affects the throughput of SSD-based KV stores, leading to a finding that SSD-based KV stores can be made latency-tolerant without devising new techniques for microsecond-latency memory.
- Implementation: We design a microbenchmark[1] as well as modify some existing SSD-based KV stores (Aerospike[2], RocksDB[3], and CacheLib[4]), so they run user-level threads and issue prefetches before they access microsecond-latency memory.
- Evaluation: By running the microbenchmark and the modified KV stores on FPGA-based memory with adjustable microsecond latency, we demonstrate that their operation throughputs can be well explained by our model, confirming that SSD-based KV stores can more readily be made latency-tolerant thanks to the presence of IO.

We believe these contributions provide novel insights into the performance of SSD-based KV stores on microsecond-latency memory that existing memory-only analysis does not provide. Our evaluation shows that the modified KV stores achieve near-DRAM throughputs, suggesting the possibility that SSD-based KV stores involving latency-sensitive in-memory traversal can use microsecond-latency memory as a cost-effective alternative to the host DRAM.

2 Objective and Observations

Our goal is to show that it is possible for SSD-based KV stores to achieve near-DRAM performance even if most of their in-memory data structures are offloaded to microsecond-latency memory. We make the following observations to reach our goal.

O1: Even with prefetching, data traversal slows down on microsecond-latency memory (Section 3.1, reconfirmation of [11])
O2: IO significantly reduces this slowdown and makes prefetching more effective in theory (Section 3.2)
O3: The throughput model based on O2 well explains microbenchmark performance, validating O2 in practice (Section 4.1).
O4: The throughput model also agrees with some SSD-based KV stores in single-core, read-dominant cases, suggesting that IO makes it easier for SSD-based KV stores to become latency-tolerant (Section 4.2.3).
O5: Latency-tolerance does not deteriorate by having other factors that slow down throughputs, including cache and lock contentions in multicore execution, write operations, and background workers (Section 4.2.4).

Putting these observations together, our analysis and experiments show that SSD-based KV stores can be made tolerant to microsecond memory latency.

3 Analysis

In this section, operation throughputs of SSD-based KV stores involving in-memory data traversal are analyzed using a simplified model of a KV operation: a sequence of (long-latency) memory accesses followed by an IO. Based on the model, we derive equations describing how throughputs depend on memory latency.

The key to (and the novelty of) our analysis is the presence of IOs. With absence of IOs (i.e., in the memory-only case), performance impacts of microsecond-level memory latency have been studied in the past. Cho et al. showed that (1) software prefetching could potentially hide microsecond

---

[1]Available at https://github.com/ybandy/cxlkvs
[2]Available at https://github.com/ybandy/aerospike-server
[3]Available at https://github.com/ybandy/rocksdb
[4]Available at https://github.com/ybandy/CacheLib

Table 1. Definition of Symbols

Symbol	Definition	Value range	Example value
Θ	KV operation throughput	-	-
L_{mem}	Memory latency	$1 - 10$ μsec	-
T_{mem}	Memory suboperation time	$O(0.1)$ μsec	0.1 μsec
$T_{\mathrm{IO}}^{\mathrm{pre}}$	Pre-IO suboperation time	$O(1)$ μsec	4 μsec
$T_{\mathrm{IO}}^{\mathrm{post}}$	Post-IO suboperation time	$O(1)$ μsec	3 μsec
T_{sw}	Context switch time	$O(0.1)$ μsec	0.05 μsec
N	# of threads	-	-
P	Prefetch queue depth	$O(10)$	10
M	# of memory accesses	$O(10)$	10

latency and achieve throughputs that matched those of DRAM, but that (2) the full potential could not be attained due to a CPU hardware limitation on prefetching [11]. We will first review these two observations in the memory-only case and model these behaviors with equations in Section 3.1. We then move on to the memory-and-IO case in Section 3.2 and show that the presence of IOs ease the prefetch limitation, thereby boosting latency-tolerance. In other words, our analysis shows that the issue observed by Cho et al. (their finding (2) above) can be mitigated by IOs without requiring to relax the CPU hardware limitation.

For tractable analysis, as well as for abstracting other factors away in order to concentrate on the impact of microsecond memory latency, we make the following assumptions. We deal with execution on a single CPU core: multiple threads divide time and no two threads run at the same time. We do not model overheads of threads: using more threads will not slow down each thread. We do not model specifics or inner workings of secondary memory and SSD devices, nor do we consider bandwidth limits of secondary memory and SSDs: we simply look at the average memory access latency and IO processing times that a CPU core experiences. We assume that the CPU has unlimited cache capacity: prefetching will always lead to a cache hit. These simplifications keep the model generic and easy to interpret (some of them will be relaxed later in Section 3.2.3). The model still well explains actual performance of SSD-based KV stores.

In this section, we mainly consider the reciprocal of a throughput, that is, the time a CPU core spends (either by doing meaningful processing or by waiting) per operation, as this will simplify equations. Table 1 summarizes the symbols used in this paper, along with the ranges of their typical values as well as their example values used for illustration.

3.1 Memory-Only Model

Here we derive throughput equations for memory-only model without IOs, which theoretically explain the empirical observations made in [11]. We consider the situation where a CPU core keeps executing memory accesses and some associated computations as shown in Figure 2. We let L_{mem} denote the memory latency, and have T_{mem} absorb all the time a CPU core spends before it requests next data from memory. In the memory-only model, one operation consists of one pair of computation and memory access for simplicity, but a sequence of these pairs will model in-memory data traversal in the memory-and-IO model in Section 3.2.
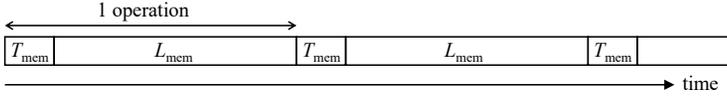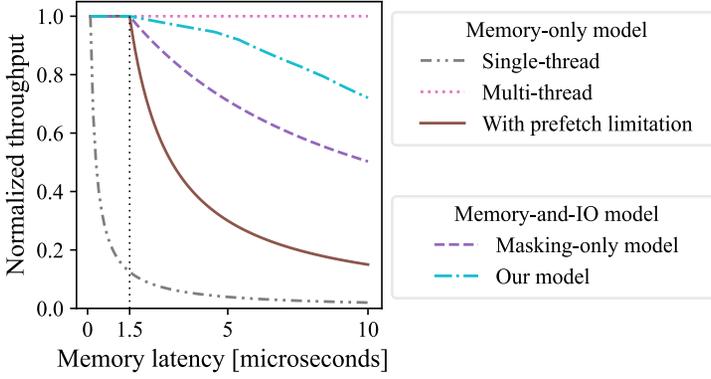
Fig. 2. Memory-only operation model

Fig. 3. Normalized throughput calculated from various models using example values in Table 1

3.1.1 Single-Threaded Case. If we naively process these operations with a single thread, each operation takes $T_{\text{mem}} + L_{\text{mem}}$ seconds, and the reciprocal throughput is given as

$$\Theta_{\text{single}}^{-1} = T_{\text{mem}} + L_{\text{mem}}. \tag{1}$$

Clearly, a longer latency L_{mem} will lead to a lower throughput, as shown in Figure 3 ($-\cdot\cdot-$).

3.1.2 Multi-threaded Case. Memory latency can be hidden by using multiple threads. Figure 4 shows $N = 6$ threads responsible for independent operations involving repeated memory accesses. When a thread (say Thread 1) needs to fetch data from long-latency memory, it issues a prefetch (dashed arrow) and yields to another thread (context switch to Thread 2, solid arrow). While the data is being prefetched to the CPU cache, another thread can resume its own operation, which also performs a prefetch and context switch. If the number of threads is large enough, the data will be on the CPU cache by the time the control returns to the thread that issued a prefetch for the data (dotted arrow). When the thread performs a memory load, it does not see the memory latency.

The reciprocal throughput in this case is given as

$$\Theta_{\text{multi}}^{-1} = \max\left\{T_{\text{mem}} + T_{\text{sw}}, \ \frac{T_{\text{mem}} + L_{\text{mem}}}{N}\right\}. \tag{2}$$

The first term comes from the situation that has just been described above, where the time each thread spends on one operation becomes $T_{\text{mem}} + T_{\text{sw}}$. The second term is due to the Little's Law [35], stating that the reciprocal throughput is the length of one operation, which is $T_{\text{mem}} + L_{\text{mem}}$ as shown in Figure 2, divided by the number N of threads. No matter how many threads we have to decrease the second term, the CPU core has to spend $T_{\text{mem}} + T_{\text{sw}}$ per operation as in the first term, and thus the maximum of the two.

Equation 2 indicates that any arbitrarily long latency can be tolerated, at least in theory, by using a large number N of threads to diminish the second term. Then the throughput becomes $\Theta_{\text{multi}} = 1/(T_{\text{mem}} + T_{\text{sw}})$, which is constant as shown in Figure 3 ($\cdots\cdots$).
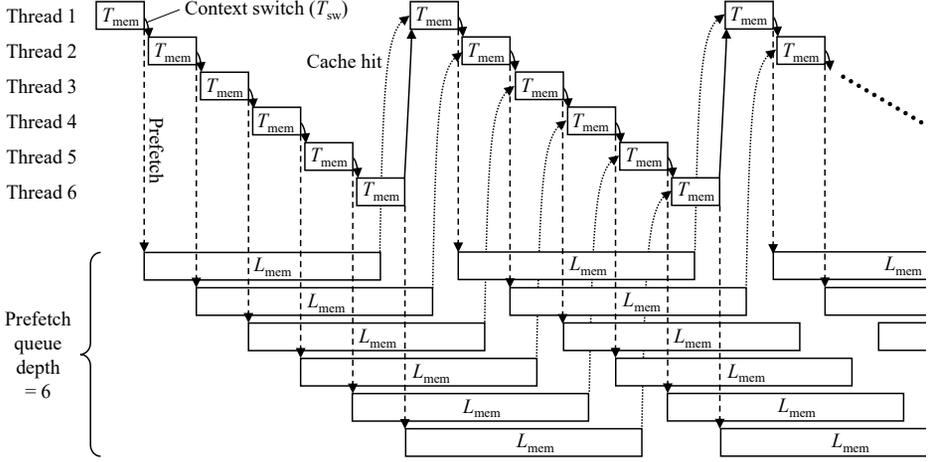
Fig. 4. Memory-only operation model using multiple threads with a large prefetch queue depth
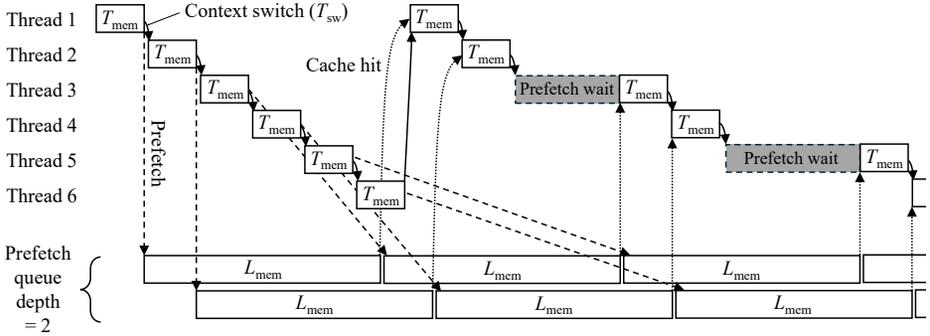

Fig. 5. Memory-only operation model using multiple threads with a small prefetch queue depth

In order for this throughput to be better than the single-threaded case in Equation 1, we need to have a short context switch time $T_{sw} \ll L_{mem}$. Conventional kernel-level threads are not effective as they have $T_{sw} \approx 1$ μsec or more. Thus, *user-level threads* [39], able to switch contexts much more quickly ($T_{sw} \approx 0.1$ μsec) by working in the user space, are used. Since user-level threads are our default choice, we simply call them "threads" unless otherwise noted.

3.1.3 Multi-Threaded Case with Prefetch Limitation. Cho et al. [11] also identified an issue that could keep the approach described above from being effective: the depth P of the prefetch queue per CPU core was limited to around $P = 10$, meaning that having more than P threads did not improve performance. For ease of illustration, let us assume we have only $P = 2$ in Figure 5. Since prefetches cannot start immediately (oblique dashed arrows), threads have to wait for the prefetches they issued to complete, wasting the CPU time (gray bars). The throughput is determined by the Little's Law applied to the memory latency L_{mem} and parallelism P. As this imposes an additional limit to Equation 2, we have a full expression of the reciprocal throughput as

$$\Theta_{mem}^{-1} = \max \left\{ T_{mem} + T_{sw}, \ \frac{T_{mem} + L_{mem}}{N}, \ \frac{L_{mem}}{P} \right\}. \tag{3}$$
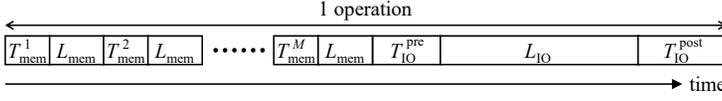
Fig. 6. Memory-and-IO operation model

Depending on the CPU hardware implementation, prefetch wait times may occur at different timings than depicted in Figure 5, or prefetches can even be dropped [37]. In any case, when the prefetch queue is full, the subsequent load will incur a cache miss. This limits the throughput, and Equation 3 still holds. No matter how many threads N are used to minimize the second term of Equation 3, the third term deteriorates the throughput, resulting in the curve shown in Figure 3 (——). By equating the first and the third terms, the latency L_{mem}^* beyond which the throughput deteriorates is given as

$$L_{mem}^* = P(T_{mem} + T_{sw}).\qquad(4)$$

With the example values in Table 1, $L_{mem}^* = 10 \times (0.1 + 0.05) = 1.5\ \mu sec$, which shows why microsecond-latency memory deteriorates throughputs. The issue does not usually manifest itself when the memory latency is in the sub-microsecond range.

Observation O1: Even with prefetching, data traversal slows down on microsecond-latency memory.

3.2 Memory-and-IO Model

Now we extend our model to include IOs, and show that IO processing times hide memory latency and boost latency-tolerance.

One operation of an SSD-based KV store is modeled as a sequence of memory accesses followed by one IO as shown in Figure 6 (this configuration will be relaxed in Section 3.2.3). The number of memory accesses per operation is denoted by M and there are M associated computations each spending T_{mem} seconds (the superscripts in the figure are there to make it clear there are M computations). Since IOs also have long latency, we process them asynchronously. Namely, one IO has three parts, (1) a pre-IO suboperation spending T_{IO}^{pre} seconds including the time to compute a storage address and to submit an IO request in a non-blocking way, (2) IO latency L_{IO}, and (3) post-IO suboperation spending T_{IO}^{post} seconds including the time to check IO completion, to copy the data to a buffer, and to use the data. Note that the figure is not to scale: IO suboperations and latency may be much longer than those related to memory.

As we are interested in hiding both memory and IO latency, we skip the single-threaded case and discuss multi-threaded scenarios.

3.2.1 Masking-Only Model. A simple explanation of the performance impact of IO would be to add an extra processing time E coming from IO to M instances of the memory-only model as

$$\Theta_{mask}^{-1} = M\Theta_{mem}^{-1} + E,\qquad(5)$$

where E is the total time a CPU core spends in processing IO as

$$E = T_{IO}^{pre} + T_{IO}^{post} + 2T_{sw},\qquad(6)$$

which is typically a few microseconds. Here we assume that we have a sufficiently large number of threads to fully hide the IO latency L_{IO} as SSDs have deep queues.

According to Equation 5, the offset E makes the degradation in the overall throughput Θ_{mask} smaller than that in the memory-only throughput Θ_{mem} alone, as shown in Figure 3 (——— over ——). For example, consider an extreme case where E is very large. Then, small change in Θ_{mem}
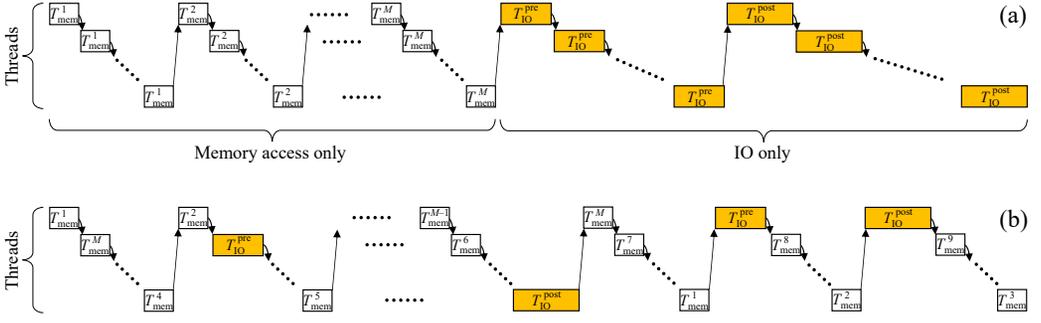
Fig. 7. Memory-and-IO operation model with threads switching context both upon memory access and IO, where suboperations are (a) aligned and (b) misaligned across threads. IO suboperations are highlighted to clarify the difference between (a) and (b).

will not affect Θ_{mask} much. In other words, the IO-related processing times E mask the performance impact of long-latency memory $M\Theta_{\text{mem}}^{-1}$, and we call this the *masking-only model*.

The masking effect increases latency-tolerance, but not to the point where we approach DRAM performance. In our example of Figure 3 (−−−), the masking-only model predicts 29% throughput degradation at a memory latency of 5 μsec, which is considerable. The reason of this degradation is because the few-microsecond offset E coming from IO processing times is not large enough to make the memory access time $M\Theta_{\text{mem}}^{-1}$ negligible. When the memory latency is in the microsecond range, we have $M\Theta_{\text{mem}}^{-1} = ML_{\text{mem}}/P$ from Equation 3. Continuing with the same example of $P = M = 10$ as above, this reduces to $M\Theta_{\text{mem}}^{-1} = L_{\text{mem}}$. Hence, $M\Theta_{\text{mem}}^{-1}$ and E in Equation 5 are both a few microseconds and are comparable to each other. While it may be tempting to think that the impact of memory latency is small in the face of IO processing times, that is not the case in general if the memory latency is in the microsecond range.

3.2.2 Our Model. Now we identify the issue in the masking-only model, and show that IO can further enhance latency-tolerance by deriving a better model. The key is whether the available prefetch depth is used efficiently. To understand this, Figure 7(a) shows the case where suboperations are aligned, meaning that all the threads perform T_{mem}^1 successively, then move on to T_{mem}^2, and so on. Since the left-hand side of this figure consists of memory accesses alone, and the right-hand side IOs alone, the reciprocal throughput is the sum of those for the both sides, as in Equation 5. Therefore, now we know that the masking-only model represents the scenario where IO does not serve to hide memory latency.

The observation above indicates that if we mix memory and IO suboperations by "misaligning" threads as shown in Figure 7(b), we have a better utilization of the available prefetch depth. In the best-case scenario, the throughput cap due to prefetch depth only applies to the entire sequence, resulting in a reciprocal throughput as follows.

$$\Theta_{\text{best}}^{-1} = \max\left\{M(T_{\text{mem}} + T_{\text{sw}}) + E, \ \frac{ML_{\text{mem}}}{P}\right\}. \tag{7}$$

Then, by comparing the two terms in Equation 7, the maximum latency L_{mem}^* that does not deteriorate the throughput is

$$L_{\text{mem}}^* = P(T_{\text{mem}} + T_{\text{sw}}) + \frac{PE}{M}. \tag{8}$$
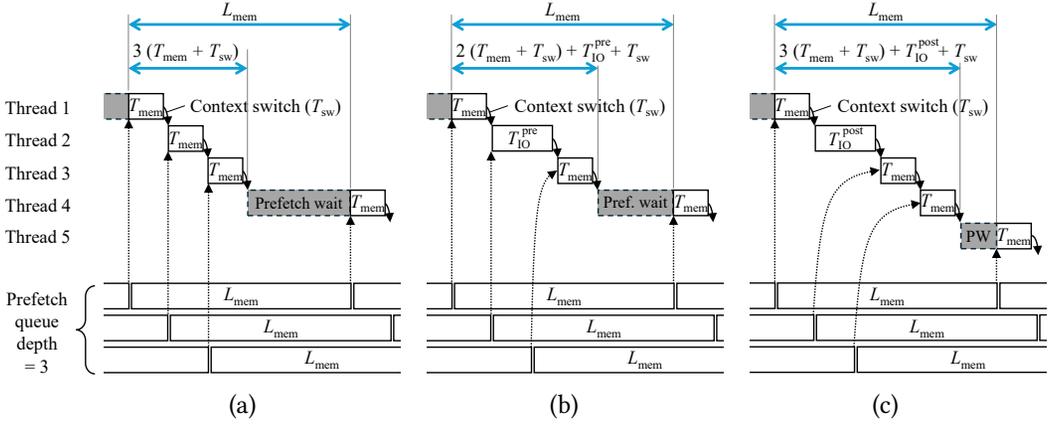
Fig. 8. Prefetch wait time (a) with no IO suboperation, (b) with a pre-IO, and (c) with a post-IO suboperation

This is longer than that of the memory-only case of Equation 4 by PE/M. With the example values in Table 1, we have $PE/M = 7.1$ μsec and thus $L_{\text{mem}}^* = 8.6$ μsec, which is much longer than without IO (1.5 μsec).

In reality, it is not practical to control execution timing of threads relative to each other. Fortunately, timing will not be aligned as in Figure 7(a) either, but will be mostly random. So far, we have considered the number M of memory accesses a constant, but in practice, it varies from operation to operation because traversal ends whenever the searched item is found. While we will still consider the *average* number of memory accesses to be M, its variance naturally misaligns thread timing, albeit not optimally in terms of latency-tolerance. We are interested in the throughput in this practical scenario. Our analysis strategy is to model prefetch wait times probabilistically and compute the expected wait time per operation.

We first see how one pre- and post-IO suboperations reduce prefetch wait times in Figure 8. As shown in Figure 8(a), in prefetch depth-limited scenarios, a prefetch wait time happens every P memory accesses ($P = 3$ in the figure), and the wait time will be $L_{\text{mem}} - P(T_{\text{mem}} + T_{\text{sw}})$. However, as shown in Figure 8(b), if one pre-IO suboperation replaces one of P memory accesses, the wait time will be reduced by $T_{\text{IO}}^{\text{pre}} - T_{\text{mem}}$, as a T_{mem}-second suboperation is replaced by a $T_{\text{IO}}^{\text{pre}}$-second one. Figure 8(c) shows the case of a post-IO suboperation. As a post-IO suboperation waits for an IO but not for a prefetch, it will defer the wait time by one thread (Thread 5 experiences the wait time rather than Thread 4), and the wait time will be reduced by $T_{\text{IO}}^{\text{post}} + T_{\text{sw}}$.

We generalize these observations to cases with more IO suboperations. We consider a sequence of $P + k$ suboperations consisting of j pre-IO, k post-IO, and $P - j$ memory suboperations. In other words, we have P memory suboperations in the beginning, but j out of P are replaced by pre-IOs, and additionally k post-IOs are inserted, making a sequence of length $P + k$. Extending the wait time reduction arguments described above to multiple IO suboperations, the wait time the $(P + k)$-th thread will experience is given as

$$T_{\text{wait}}(j, k) = \max\{0, L_{\text{mem}} - P(T_{\text{mem}} + T_{\text{sw}}) - j(T_{\text{IO}}^{\text{pre}} - T_{\text{mem}}) - k(T_{\text{IO}}^{\text{post}} + T_{\text{sw}})\}, \quad (9)$$

where $\max\{0, \cdot\}$ is because the wait time is non-negative.

Next, we compute the probability that this suboperation sequence happens. To make the analysis tractable, we make the following simplification: we assume memory and IO suboperations to occur

independently and identically. Namely, a suboperation that a CPU core processes next will be one of the following.

- Memory suboperation T_{mem} with a probability of $M/(M+2)$,
- Pre-IO suboperation $T_{\mathrm{IO}}^{\mathrm{pre}}$ with a probability of $1/(M+2)$, or
- Post-IO suboperation $T_{\mathrm{IO}}^{\mathrm{post}}$ with a probability of $1/(M+2)$.

Then, the probability $p(j,k)$ that the above-mentioned sequence with j pre-IO and k post-IO suboperations happens is

$$p(j,k) = \frac{(P+k)!}{(P-j)!\,j!\,k!} \left(\frac{M}{M+2}\right)^{P-j} \left(\frac{1}{M+2}\right)^{j+k}. \tag{10}$$

Finally, we compute the expected prefetch wait time. Suppose we have many $(i = 1, 2, \cdots, n)$ of these sequences each of which is of length $P + k_i$ and contains (j_i, k_i) IO suboperations. In the long run, the average wait time per suboperation will be

$$\frac{T_{\mathrm{wait}}(j_1, k_1) + T_{\mathrm{wait}}(j_2, k_2) + \cdots + T_{\mathrm{wait}}(j_n, k_n)}{(P+k_1) + (P+k_2) + \cdots + (P+k_n)}. \tag{11}$$

When n is large, the numerator and denominator behave as independent Gaussian random variables (Central Limit Theorem), and hence the expected per-suboperation wait time $T_{\mathrm{wait}}^{\mathrm{subop}}$ can be approximated by the ratio of the expectations (denoted by $\mathbb{E}[\cdot]$) of the numerator and denominator [21] as

$$T_{\mathrm{wait}}^{\mathrm{subop}} \approx \frac{\mathbb{E}_{j,k}[T_{\mathrm{wait}}(j,k)]}{\mathbb{E}_{j,k}[P+k]} = \frac{\sum_{j=0}^{P} \sum_{k=0}^{\infty} p(j,k)\,T_{\mathrm{wait}}(j,k)}{\sum_{j=0}^{P} \sum_{k=0}^{\infty} p(j,k)\,(P+k)}. \tag{12}$$

To compute these expectations, one does not have to deal with infinite summation ($k \to \infty$) in practice, since $p(j,k)$ quickly vanishes for large k. As one operation has $M + 2$ suboperations, the expected reciprocal throughput according to our probabilistic formulation is

$$\Theta_{\mathrm{prob}}^{-1} = M(T_{\mathrm{mem}} + T_{\mathrm{sw}}) + E + (M+2)\,T_{\mathrm{wait}}^{\mathrm{subop}}. \tag{13}$$

The throughput degradation predicted by this model is plotted in Figure 3 ($-\cdot-$). The degradation is much smaller, 7% at a memory latency of 5 μsec, compared with 29% of the masking-only model ($---$). This is because IOs interleave with prefetches and alleviate the slowdown coming from the prefetch limitation.

Observation O2: IO significantly reduces the slowdown due to long memory latency and makes prefetching more effective.

3.2.3 Model Extension. Our memory-and-IO model, which has assumed one IO to follow M memory accesses, can be extended to cases where multiple IOs appear anywhere in the sequence. If one operation has S IOs on average, we can split it into S smaller operations each with M/S memory accesses and one IO, and use Equation 13 to calculate the expected throughput. As this is S-fold overcounting, we divide it by S. Since scale does not matter in discussing throughput degradation, we consider M to be a per-IO value in what follows. Our model applies to any order of suboperations as our formulation only assumes probabilities of suboperations.

For completeness, we further extend our model by relaxing some of the simplifications described in the beginning of Section 3. Table 2 lists additional symbols that will come into play. The right column of the table shows example values for these system parameters, which we use in our evaluation in Section 4. The reciprocal throughput according to our extended model is given as

Table 2. System Parameters

Symbol	Definition	Example value
A_{mem}	Memory access (cacheline) size	64 bytes
B_{mem}	Maximum memory bandwidth	10 GB/sec
A_{IO}	SSD access (IO) size	O(1) kB
B_{IO}	Maximum SSD bandwidth	10 GB/sec
R_{IO}	Maximum SSD random access	2.2 MIOPS
ρ	Offloading ratio of indices and caches	1
ε	Premature CPU cache eviction ratio	≈ 0

follows.

$$\Theta^{-1}_{\mathrm{extended}} = \max\left\{\Theta^{-1}_{\mathrm{rev}},\ \frac{A_{\mathrm{IO}}}{B_{\mathrm{IO}}},\ \frac{1}{R_{\mathrm{IO}}}\right\}, \tag{14}$$

where A_{IO} is the average IO access size (in bytes), and B_{IO} and R_{IO} are the maximum bandwidth (bytes/sec) and random access performance (IOPS) of the SSD, respectively. Θ_{rev} is a revised version of the probabilistic throughput model of Equation 13 taking into account the memory bandwidth and CPU cache capacity limits as well as memory tiering of DRAM and secondary memory. Θ_{rev} can be obtained through a similar derivation to that in Section 3.2.2 with the following two modifications. First, we replace the memory latency L_{mem} in Equation 9 as

$$L_{\mathrm{mem}} \leftarrow \max\left\{\rho\, L_{\mathrm{mem}} + (1 - \rho)\, L_{\mathrm{DRAM}},\ (P - j)\,\frac{A_{\mathrm{mem}}}{B_{\mathrm{mem}}}\right\}, \tag{15}$$

where ρ is the offloading ratio to the secondary memory, L_{DRAM} is the DRAM latency, A_{mem} is the memory access size (in bytes), and B_{mem} is the maximum memory bandwidth (bytes/sec). The offloading ratio ρ is in terms of access frequency, such that the average memory latency in the long run becomes a linear interpolation of the two types of memory latencies as in the first term of Equation 15. For example, $\rho = 0.7$ means that 70% of accesses to indices and caches go to the secondary memory and the rest to the DRAM. The second term of Equation 15 is because a suboperation sequence containing $P - j$ memory suboperations takes at least $(P - j)A_{\mathrm{mem}}/B_{\mathrm{mem}}$ seconds due to the memory bandwidth limit.

The second modification in the derivation relates to the CPU cache capacity limit. If the capacity is small, some of prefetched data may be evicted from the cache before it is referenced. We call this probability a premature CPU cache eviction ratio ε. In the original derivation, we assume one memory suboperation to occur with a probability of $M/(M + 2)$. We split this into two cases depending on whether it is executed before or after the prefetched data is evicted.

- Pre-eviction memory suboperation with a probability of $(1 - \varepsilon)M/(M + 2)$, and
- Post-eviction memory suboperation with a probability of $\varepsilon M/(M + 2)$,

where the former can be treated in the same way as the original memory suboperation, while the latter behaves in the same way as a post-IO suboperation except that it takes L_{mem} seconds instead of $T^{\mathrm{post}}_{\mathrm{IO}}$. Thus, we can follow the same derivation as in Equations 10–12 by using different probabilities and suboperation times.

The simplification we do not address here is the overhead of threads. In practice, using more threads leads to a slowdown in throughputs, mainly caused by the increased cost of context switching and CPU cache contention. However, both of them are difficult to theoretically model, as they depend on many other factors such as the thread implementation, CPU cache configuration,

Table 3. Computational Environment

Part	Specifications
CPU	2 of Intel Xeon Gold 6430 (32 cores/CPU, 2.10 GHz)
DRAM	DDR5 4800 MHz 512 GB (32 GB × 8 ch./CPU)
CXL	2 of Intel Agilex 7 FPGA I-Series Dev. Kit (128 GB in total)
CXL	1 of LR-LINK CXL Memory Expander (64 GB)
SSD	4 of Intel Optane 900P 480 GB NVMe (1.92 TB in total)
OS	Ubuntu 22.04.5 LTS, Linux kernel 6.8.0

memory footprint, memory address assignment, and memory access patterns. Fortunately, the impact of the thread overhead seems limited as we will see later.

4 Evaluation

We conduct performance evaluation to support our analysis. We first run a microbenchmark to validate our throughput model (Section 4.1). We then demonstrate that SSD-based KV stores can be made tolerant to microsecond-level memory latency thanks to the presence of IOs (Section 4.2).

Our computational environment is summarized in Table 3. As currently-available memory devices do not have microsecond-level latency, we use FPGA-based CXL memory. Two FPGA boards are each equipped with 64-GB DRAM, and we implement an FPGA circuit so that the memory latency can be adjusted to an arbitrary length [50]. Additionally, we use a commercially-available CXL memory expander equipped with DRAM. While the measured latency of this device is around 300 nanoseconds, it allows us to check the validity of our model at that specific, sub-microsecond latency. All of these CXL devices are attached via PCIe links, and appear as CPU-less NUMA nodes. Hyper-Threading and hardware prefetching are disabled so that performance is more predictable and interpretable, as is also done in other works [11, 24, 54].

The system parameters of our computational environment are shown in the right column of Table 2. They are chosen so as not to obscure the latency-dependence of KV stores. We use multiple devices so that the combined bandwidths B_{mem} and B_{IO} and random access performance R_{IO} do not limit the KV throughputs. We offload the entire KV indices and caches to secondary memory ($\rho = 1$) so that the KV throughputs always depend on the predetermined microsecond-level latency. The CPU cache size is large enough (60 MB L3) so as not to limit the KV throughputs. As we will see later, the premature CPU cache eviction ratio ϵ is close to zero. Therefore, our simplified model of Equation 13 suffices to describe our experimental conditions. Nonetheless, we validate our extended model of Equation 14 by changing the system parameters to deliberately violate our simplifying assumptions in Section 4.1.4. To this end, our FPGA-based memory is designed to be able to throttle the memory bandwidth.

4.1 Microbenchmark

We design a microbenchmark that performs exactly the operation modeled in Section 3, and confirm that observed throughputs align well with the throughput equation of Equation 13.

4.1.1 Implementation. In our microbenchmark, each operation consists of M successive memory accesses followed by pre-IO and post-IO suboperations as shown in Figure 6, and multiple threads keep processing independent operations as in Figure 7.

Figure 9 illustrates one operation in more detail. Memory suboperations perform pointer chasing, so that a next access depends on its previous one, simulating latency-sensitive data traversal. We
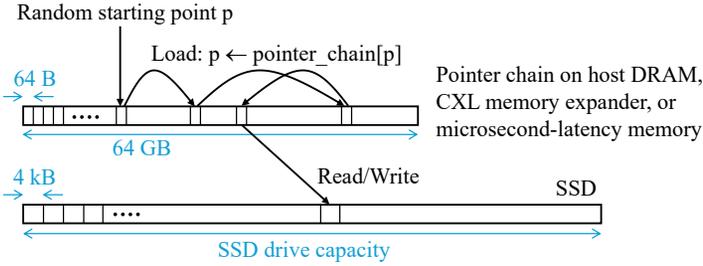
Fig. 9. One operation of the microbenchmark

use Argobots [51] for user-level threads, and each time a thread performs a memory suboperation, it issues a prefetch for the next pointer address and yields to another thread. We place the entire pointer chain on any one of the host DRAM, CXL memory expander, or microsecond-latency memory. The chain elements are permuted to minimize spatial locality. The size of the pointer chain is 64 GB (1 billion of 64-byte cacheline-sized pointers). Each operation picks a random starting point and performs pointer chasing M times. After pointer chasing, an SSD is accessed based on the final pointer (i.e., random read or write). We access an SSD as a block device to avoid performance irregularity due to a file system. Asynchronous IO is implemented with io_uring [5].

4.1.2 Experimental Conditions. We run the microbenchmark on a single core for 1,404 (= $4 \times 3 \times 3 \times 3 \times 13$) combinations of different parameter values and memory latencies as follows.

- The number of memory accesses $M = \{1, 5, 10, 15\}$
- Memory suboperation time $T_{\text{mem}} = \{0.10, 0.12, 0.14\}$ μsec
- Pre-IO suboperation time $T_{\text{IO}}^{\text{pre}} = \{1.5, 2.5, 3.5\}$ μsec
- Post-IO suboperation time $T_{\text{IO}}^{\text{post}} = \{0.2, 1.2, 2.2\}$ μsec
- Memory latency $L_{\text{mem}} = \{0.1, 0.3, 0.5, 1, 2, 3, \cdots, 10\}$ μsec

The variations in T_{mem} are created with a spin loop by calling pause 1, 3, and 5 times. From the host DRAM execution, we estimate one call of pause consumes 10 nsec with an offset of 90 nsec, thus $T_{\text{mem}} = 90 + 10 \times 5 = 140$ nsec if we call pause 5 times, for instance.

The variations in $T_{\text{IO}}^{\text{pre}}$ and $T_{\text{IO}}^{\text{post}}$ are created by adding extra 1 or 2 μsec to the IO submission and completion checking times. The IO submission and completion checking times are estimated to be 1.5 and 0.2 μsec, respectively, by running an IO-only benchmark (i.e., $M = 0$).

For each parameter combination, we evaluate the throughput dependence on memory latency. When the pointer chain is placed on the host DRAM, the memory latency is about 0.1 μsec. When on the CXL memory expander, it is about 0.3 μsec. When on the microsecond-latency memory, we change its latency from 0.5 to 10 μsec (0.5 μsec is the minimum latency of our FPGA-based memory). For each latency, we optimize the number of threads: namely, we try different numbers of threads (on a single core) and report the highest throughput. Since we observe similar throughputs for read IOs and write IOs, we report read IO results.

We confirm it is rare that prefetched data gets evicted from the CPU cache before it is referenced during pointer chasing. Using PEBS [13] via perf mem command, we measure latencies experienced by load instructions for accessing the secondary memory. As shown in Figure 10(a), most (note the log scale) of the load latencies are close to zero, indicating cache hits. Some loads wait for a few microseconds due to late prefetches caused by the prefetch queue limit. Only a small fraction ($\varepsilon < 0.0005$) of loads suffer premature cache eviction and wait for as long as the set memory latency
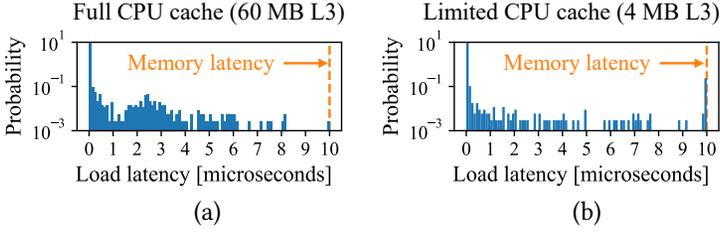
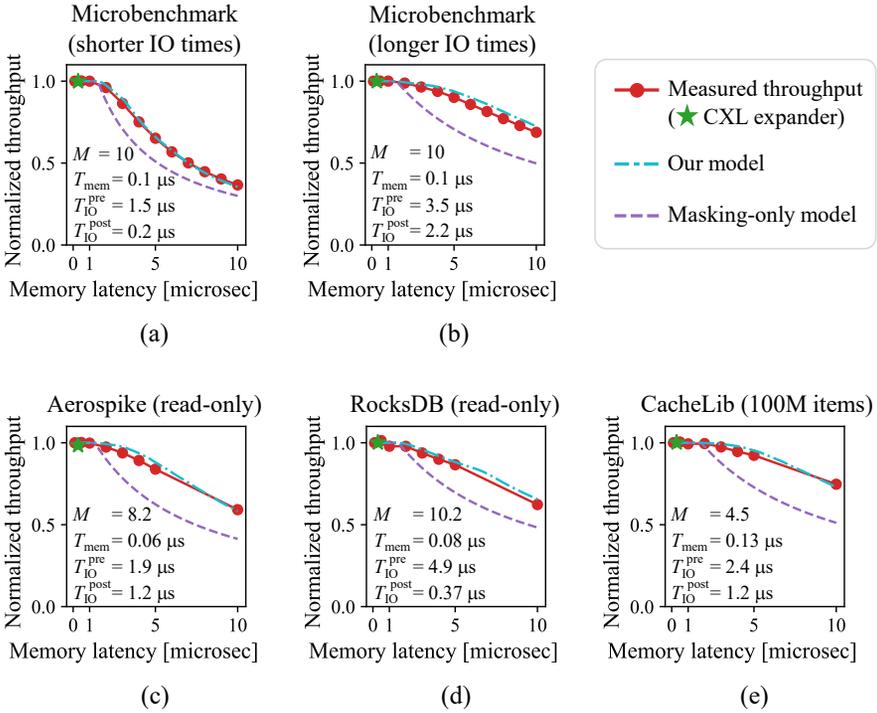Fig. 10. Probability distribution of load latency

Fig. 11. Normalized throughputs of (a)(b) the microbenchmark and of (c)(d)(e) the modified SSD-based KV stores on a single core, along with those predicted by the models

(10 μsec in the figure). By drastically reducing the L3 cache size from 60 MB to 4 MB using `resctrl` [63], we observe a non-negligible eviction ratio of $\varepsilon \approx 0.05$ as shown in Figure 10(b).

4.1.3 Results. Figure 11(a)(b) shows throughputs for some representative parameter combinations. The throughputs are normalized by that on DRAM to show how much throughput degradation occurs as the memory latency increases. Along with the actual measurements, throughput degradations predicted by the masking-only model (Equation 5) and by our probabilistic model (Equation 13) are shown. The model throughputs are calculated from the same parameter values $(M, T_{\mathrm{mem}}, T_{\mathrm{IO}}^{\mathrm{pre}}, T_{\mathrm{IO}}^{\mathrm{post}})$ used to run the microbenchmark. The other model parameters are estimated as $T_{\mathrm{sw}} = 50$ nsec and $P = 12$ via Equation 3 by running the microbenchmark with no IOs.

While the masking-only model underestimates performance for long memory latency, our model better approximates it. In all the 1,404 parameter and latency combinations, the masking-only model
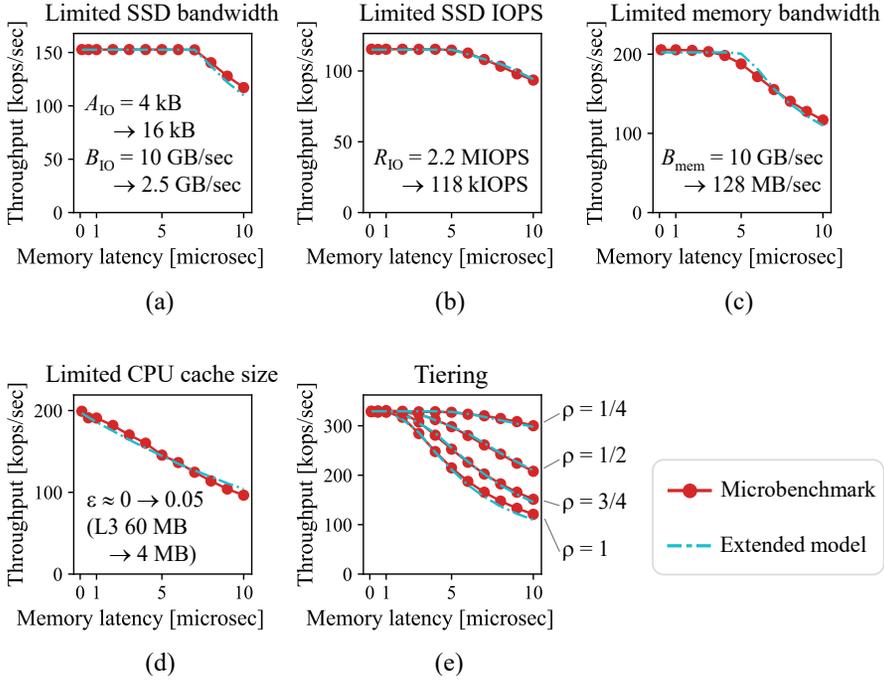
Fig. 12. Throughputs of the microbenchmark and those predicted by the extended model under the conditions where other factors than memory latency affect performance

underestimates the actual performance by up to 32.7%, while our model is within $[-5.0\%, +6.8\%]$ of the actual measurements. This indicates that the latency-tolerance gained from IO cannot be solely explained by masking. IO eases the prefetch limitation and allows prefetching to better hide latency. By comparing the two plots in Figure 11(a)(b), we can also see that the latency-tolerance is better in (b) than in (a) because (b) has longer IO suboperations. The improvement over the masking-only explanation is also larger in (b). Additionally, these microbenchmark results indicate that we can achieve DRAM-equivalent throughputs using the commercial CXL memory expander as predicted by our model.

Observation O3: The throughput model based on Observation O2 (IO enhances latency-tolerance) well explains microbenchmark performance, validating O2.

4.1.4 Validation of Extended Model. The above results are obtained under the condition that the simplifications made in Section 3 hold. However, violation of them can have practical performance impacts, and we illustrate them in Figure 12. Figure 12(a) shows an SSD bandwidth-limited scenario by increasing the access size A_{IO} and by using only one SSD to decrease the bandwidth B_{IO}. Until the memory latency becomes too long to saturate the SSD bandwidth, the throughput is capped by the bandwidth and stays the same. Similar behaviors are observed when the throughput is limited by the SSD random access performance R_{IO} (realized by using a slow SATA SSD) and by the memory bandwidth B_{mem} (throttled by the FPGA) as shown in (b) and (c), respectively. Figure 12(d) shows a CPU cache size-limited scenario. The latency-tolerance deteriorates since prefetching becomes ineffective. Figure 12(e) shows tiering where the pointer chain is split into $(1 - \rho) : \rho$ and
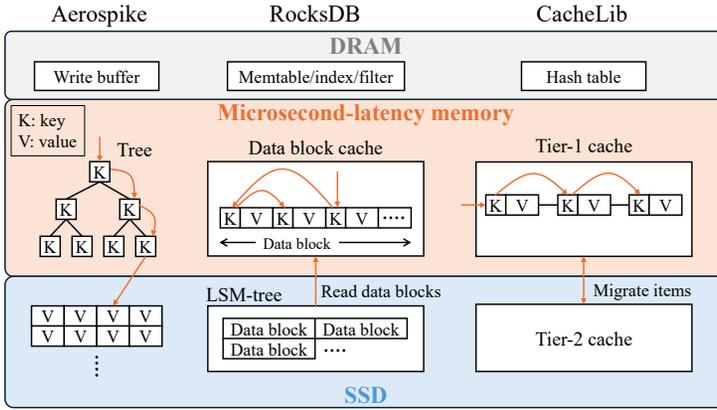
Fig. 13. SSD-based KV stores performing pointer-chasing on large in-memory data structures in addition to accessing SSDs. We place these large data structures on microsecond-latency memory, while leaving other small data structures on the host DRAM.

they are placed on DRAM and the secondary memory, respectively. As expected, we observe better latency-tolerance for smaller offload ratio ρ.

In all of these scenarios, the observed throughputs are well explained by our extended model.

4.2 KV Store Examples

Now we demonstrate that existing SSD-based KV stores can be made tolerant to memory latency with some modifications, and our analysis holds for these modified KV stores, indicating that their latency-tolerance is also enhanced by the presence of IOs.

To test with different SSD-based KV store designs, we take Aerospike [53], RocksDB [17], and CacheLib [8] as examples. As shown in Figure 13, Aerospike traverses in-memory trees before accessing values on SSDs. RocksDB fetches a data block from a log-structured merge-tree (LSM-tree) on SSDs and traverses sorted keys in the data block in an in-memory block cache. CacheLib is a two-tier KV cache where the first, in-memory tier has linked items and LRU lists to be traversed while using SSDs as the second tier. Even though these KV stores are different, they all involve in-memory data traversal and IOs modeled by our analysis. As these in-memory indices and caches are the largest DRAM consumers in the respective SSD-based KV stores, we offload them in their entirety to secondary memory. Offloading other data structures would have much less contribution to cost reduction, and they remain in the host DRAM as depicted at the top of Figure 13.

Our extended model in Section 3.2.3 allows one KV operation to issue a varying number of IOs, covering RocksDB and CacheLib. Beyond that, we do not need the model extension of Equation 14 since our system parameters in Table 2 are chosen so that factors other than memory latency do not limit the KV throughputs.

4.2.1 Implementation. We modify these KV stores so that they

- allocate in-memory indices and caches on microsecond-latency memory,
- replace kernel-level threads with user-level threads,
- issue a prefetch and yield upon accessing indices and caches, and
- use an asynchronous IO interface.

Table 4 exemplifies these modifications in C code. They are written generically, and we refer the reader to the open-sourced modified code for complete modifications specific to each KV store.

Table 4. Code Modifications

Modification	Original code	Modified code
Memory allocation	index_t* indices = malloc(size);	index_t* indices = mmap(···, size, ···); mbind(indices, size, ···, &nodemask, ···);
Use of user-level threads	Kernel-level thread function family • pthread_create(···), • pthread_join(···), etc.	User-level thread equivalents • user_level_thread_create(···), • user_level_thread_join(···), etc.

Insertion of prefetch and yield	index_t val = indices[i];	__builtin_prefetch(indices + i);
		 user_level_thread_yield();
		 index_t val = indices[i];

Asynchronous IO	pread(fd, buf, count, offset);	struct io_uring_sqe *sqe = io_uring_get_sqe(&ring);
		 io_uring_prep_read(sqe, fd, buf, count, offset);
		 io_uring_submit(&ring);
		 user_level_thread_yield();
		 io_uring_peek_cqe(&ring, &cqe);

As CXL memory devices (expander cards as well as our FPGA-based memory) appear as CPU-less NUMA nodes, memory can be allocated on them by specifying nodes in nodemask for mbind.

Kernel-level thread functions are replaced by user-level thread counterparts. As with the microbenchmark, we use Argobots [51] as user-level threads for Aerospike. For RocksDB, we adopt an existing modification [10] that uses PhotonLibOS [34]. Since CacheLib incorporates Folly library [56], we use Folly Fibers.

To insert a prefetch and yield, we need to search the code for all accesses to secondary memory. To this end, we repurpose Valgrind Memcheck memory error detector [14]. By marking the indices and caches on the secondary memory as inaccessible and by running a KV workload, accesses to them are reported as illegal. We insert a prefetch and yield before each detected memory access.

For asynchronous IO, we use io_uring [5]. A standard IO such as pread and pwrite is replaced by an IO submission (the first three lines of code in Table 4) and IO completion checking (last line). We also insert a yield after an IO submission to hide IO latency.

The modified KV stores run faster than the original implementations in our evaluation. When storing all of the in-memory data structures on the host DRAM, they show 1.2x higher throughputs than their original counterparts mainly thanks to the lightweight threads and IO.

4.2.2 Experimental Conditions. We use the benchmark tools accompanying the respective KV stores: Aerospike Benchmark for Aerospike, db_bench for RocksDB, and CacheBench for CacheLib. Table 5 summarizes experimental parameters for the modified KV stores. Our default settings are shown in bold letters, many of which follow those of the benchmark tools. We first run benchmarks using the default settings, and later make variations by changing each parameter as listed in the table. Other parameters are chosen as follows.

Aerospike: We set the default value size to 1.5 kB according to Aerospike Certification Tool [2]. By Aerospike's design, the size of each tree node is always 64 bytes regardless of the key size, thus we do not vary the key size. By placing the trees (32 GB = 64 bytes × 500M) on secondary memory, the host DRAM usage is only 1.3 GB (96% offload), which is mainly a write buffer.

RocksDB: Since the block cache is effective for skewed key distributions, we implement Zipfian distribution in db_bench. With a Zipf exponent of 0.99 and a block cache size of 32 GB, the block

Table 5. KV Store Settings

Parameter	Aerospike	RocksDB	CacheLib
# items	**500M**	**1B**	**100M**, 400M
Key size (bytes)	**20**	10, **20**, 40	[4, 8], **[8, 16]**, [16, 32]
Value size (bytes)	1 k, **1.5 k**, 2–2.5 k	200, **400**, 800	[100, 150], **[200, 300]**, [300, 450]
Distribution	**Uniform**, Zipf 1.1	**Zipf 0.99**, Zipf 0.8	**Gaussian**, graph cache leader*
Read:write	**1:0**, 2:1, 1:1	**1:0**, 2:1, 1:1	**2:1**, 1:1

* One of the workloads in CacheBench. We use its key distribution.

cache hit ratio is 67%. The other in-memory data structures consume 8 GB of the host DRAM, meaning that 80% (32 out of 40 GB) of the memory consumption is offloaded to secondary memory.

CacheLib: We set the value size to a few hundred bytes as observed in many web services [4, 41], and thus Small Object Cache [8] is used as a tier-2 (SSD) cache. We first run a relatively small workload encountering 100 million items with 8-GB tier-1 (host DRAM or secondary memory) and 32-GB tier-2 caches. Since it can take a long time for the throughput of CacheLib to stabilize especially on a single core, running a small workload makes the warm up period short enough (under 6 hours) to allow us to iterate with varying numbers of CPU cores. Then, we run a larger workload on 16 cores using 32-GB tier-1 and 128-GB tier-2 caches to deal with 400 million items. When secondary memory is used, the remaining DRAM usage is 4.3 GB (65% offload) and 7.8 GB (80% offload) for the smaller and larger workloads, respectively, which are mostly attributed to the hash table. Once the throughput is stabilized, the cache hit ratio is 82% (34% at tier-1 and 73% at tier-2 upon tier-1 misses). We further increase the tier-2 cache size to 512 GB later in Section 5.1.

4.2.3 Single-Core Results. We first demonstrate that the throughputs of the modified KV stores as functions of memory latency follow our analysis. Figure 11(c)(d)(e) shows throughputs of the modified SSD-based KV stores for varying memory latency. They are normalized by the throughput observed when the entire in-memory data structures are placed on the host DRAM. As before, normalized throughputs according to the models are also shown. The measured model parameters are shown in each plot. They are measured by recording timestamps before and after prefetch yields and IO yields during host DRAM execution. This is done separately from throughput measurements as the recording incurs some overheads.

As with the microbenchmark results, the actual performance is close to what our probabilistic model predicts, and is higher than the masking-only explanation. Again, this indicates that IO eases the prefetch limitation and allows prefetching to better hide latency.

Observation O4: The throughput model agrees with the modified SSD-based KV stores in single-core, read-dominant cases, suggesting that IO makes it easier for SSD-based KV stores to become latency-tolerant.

4.2.4 Multi-Core Results. We show that the latency-tolerance observed in the single-core execution does not deteriorate when the number of cores increases. We run the same workload as in Section 4.2.3 on 2, 4, 8, and 16 cores. For a given number of core and memory latency, we test different numbers of threads per core and pick the best throughput. The throughputs of the modified KV
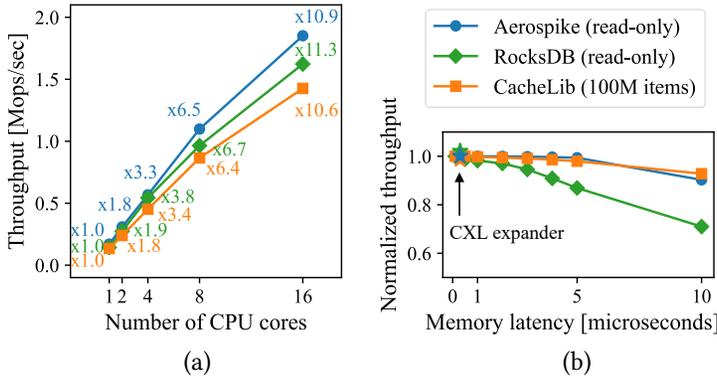
Fig. 14. Multi-core throughputs of the modified SSD-based KV stores (a) for varying numbers of cores at a fixed memory latency of 5 usec, and (b) for varying memory latency at a fixed number (16) of cores.
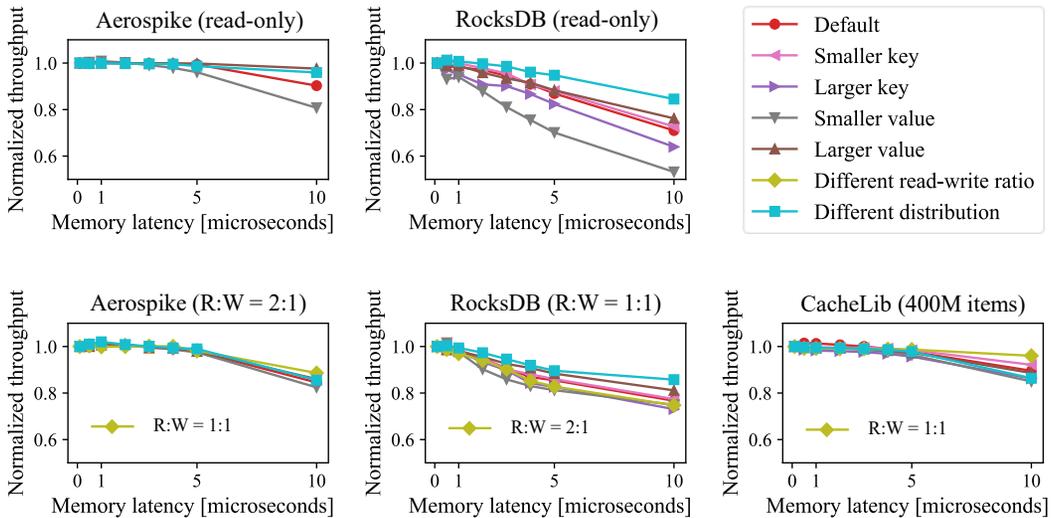

Fig. 15. Normalized throughputs of the modified SSD-based KV stores with various settings

stores with different numbers of cores are plotted in Figure 14(a). Since plots of different memory latencies behave similarly, we only show those for a memory latency of 5 μsec. The vertical axis shows raw throughputs, while the numbers in the plot show the throughputs normalized by that on a single core. The throughput scales nicely as the number of cores increases, although it is sublinear (1.8–1.9x when the number of cores doubles). The sublinearity likely comes from increased lock and CPU cache contentions. This slowdown can favorably affect latency-tolerance as it masks throughput degradation due to memory latency. As shown in Figure 14(b), with 16-core execution, RocksDB maintains the same level of latency-tolerance compared with the single-core case in Figure 11(d), while Aerospike and CacheLib see better tolerance with less than 2% throughput degradation up to a memory latency of 5 μsec.

We also vary KV store settings as shown in Table 5 to see how they influence latency-tolerance. The top and bottom rows of Figure 15 show read-only and write-mix cases, respectively. For each
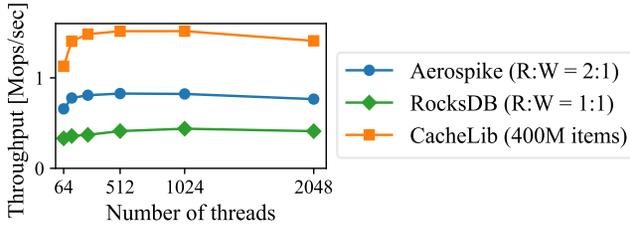
Fig. 16. Throughput dependence of the modified SSD-based KV stores on the number of threads
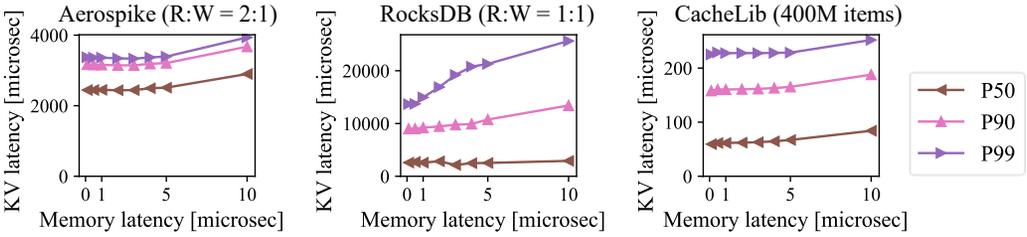

Fig. 17. KV operation latency of the modified SSD-based KV stores

case, we test smaller/larger key/value sizes, different key distributions, and for the write-mix case, different read-write ratios. In most settings, we observe similar levels of latency-tolerance to that with the default settings. Notable exceptions with worse tolerance are when we use smaller values in the read-only case. This is because the slowdown coming from IO becomes smaller, making the impact of memory latency relatively larger. The opposite effect occurs for RocksDB with less skewed distribution. More block cache misses lead to more IOs, resulting in better latency-tolerance. Write-mix settings make latency-tolerance less dependent on other factors as their impacts are masked by the increased burst SSD writes and the overhead of background workers for defragmentation and compaction. The throughput degradation is 8% at a memory latency of 5 μsec when averaged (geomean) over all of these measurements.

Figure 16 shows that throughput degradation due to the thread overhead is small. Unless too many threads are used, the peak throughput is fairly stable across varying numbers of threads.

Figure 17 shows KV operation latency. Longer memory latency leads to longer KV operation latency, but the impact is limited.

Observation O5: Latency-tolerance does not deteriorate by having other factors that slow down throughputs, including cache and lock contentions in multicore execution, write operations, and background workers.

5 Discussion

By making Observations O1-5, our analysis and evaluation have found that SSD-based KV stores involving latency-sensitive in-memory traversal can be made tolerant to microsecond memory latency. This suggests that one could use devices with low-cost memory media including CXL-based secondary memory using compressed DRAM [3] and flash memory [36]. Such devices are still not readily available, and accurately modeling their expected performance is out of the scope of this paper. However, we can make crude estimates of the system cost-performance improvement they would potentially gain, which we discuss below in Section 5.1. After that, we discuss future directions of our work in Section 5.2.

Table 6. Example CPR parameters and values

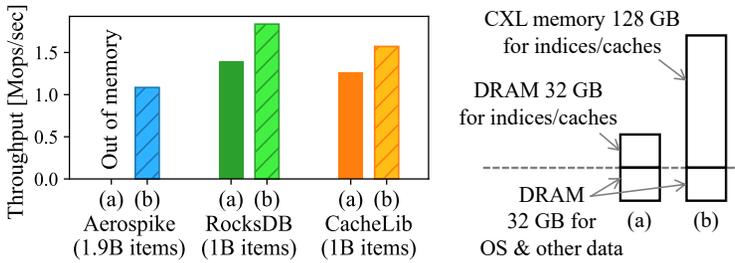| Memory medium	Bit cost b	Degradation d	CPR r
Compressed DRAM	$1/3 - 1/2$	$0 - 0.02$	$1.23 - 1.36$
Low-latency flash	$0.15 - 0.2$	$0.02 - 0.19$	$1.19 - 1.50$

Fig. 18. Throughput comparison between cases where space-consuming in-memory data is stored in (a) 32 GB of DRAM and in (b) 128 GB of microsecond-latency memory.

5.1 Cost-Performance Estimates

When part of the host DRAM is replaced by secondary memory, the cost-performance ratio (CPR) r of the resultant system relative to the original DRAM-only system can be expressed as

$$r = \frac{1 - d}{cb + (1 - c)}, \tag{16}$$

where c (< 1) is the cost of DRAM being replaced by the secondary memory relative to the total server cost, b (< 1) is the relative bit cost (dollars/GB) of the secondary memory to DRAM, and d (< 1) is the throughput degradation caused by the use of the secondary memory. A CPR r exceeding 1 means cost-performance improvement.

Table 6 shows some example values for b and d. The bit cost b of compressed DRAM comes from its typical compression ratio of 2-3x [3], which increases the effective capacity and thereby reduces the bit cost. The bit cost of the flash-based memory device is based on low-latency SLC (single-level cell) flash. The relative bit cost of SLC is 0.15 according to [60]. We add some range to it by conservatively assuming that SLC is up to ten (rather than four) times more expensive than QLC (quad-level cell), where the relative bit cost of QLC is estimated to be 0.02 from retail prices of SSD and DRAM [42].

For throughput degradation d, since the latency of compressed DRAM is under 1 μsec [3], we refer to the 0-2% degradation observed in our evaluations. For flash-based memory, we assume a typical latency of 5 μsec. We also simulate tail latency by configuring our FPGA to occasionally introduce longer latencies of 14 and 48 μsec at probabilities of 9.9% and 0.1%, respectively, which fit the relative latency distribution of a low-latency SSD [49] under our FPGA design constraints. With these settings, we conduct additional experiments and observe 2-19% throughput degradation.

Given these numbers, Table 6 shows CPR values r in a hypothetical scenario where DRAM accounts for half of the server cost [33] and we replace 80% of it with secondary memory (i.e., $c = 1/2 \times 0.8 = 0.4$). These CPR values well exceed one, suggesting that these secondary memory devices will likely bring about cost-performance improvement.

Instead of reducing the server cost, the saved money could be used to purchase more secondary memory. We consider another hypothetical scenario where our server had only 32 GB of DRAM.

We could add 128 GB of flash-based CXL memory at a potentially cheaper price than adding another 32 GB of DRAM. As shown in Figure 18, we keep the original 32 GB DRAM for OS and small in-memory data of an SSD-based KV store, and compare the cases where large indices and caches are stored in (a) the extra 32 GB of DRAM or (b) 128 GB of CXL memory with a latency of 5 μsec along with the tail latency described above. As shown in the figure, Aerospike can handle up to 1.9 billion items with the increased secondary memory capacity, but it runs out of memory with the DRAM-only system. RocksDB running on 32 cores for a less skewed key distribution (a Zipf exponent of 0.7) sees an improved throughput by 32% by having a larger block cache. CacheLib encountering 1 billion items here uses a larger tier-2 (SSD) cache of 512 GB, but it also requires a correspondingly larger tier-1 cache to maintain throughputs, and the system with the secondary memory is 25% faster than the DRAM-only system.

5.2 Future Directions

Our analysis and evaluation stand on a number of assumptions, some of which call for further study and effort as discussed below.

5.2.1 Software Modification. This work has modified existing SSD-based KV stores, and reducing the modification labor is out of the scope of this paper. We believe the extent of the modification has been modest relative to the existing large code bases we have built on, but nonetheless it is desirable to automate the conversion. We have made all the code available, and we hope our findings will encourage communities to develop SSD-based KV stores that natively support longer-latency memory without requiring modification.

5.2.2 Analysis Scope and Model Applicability. Our analysis covers SSD-based KV stores that follow our operation model. While we have shown that our throughput equation holds for three KV stores that are quite different in design, it is likely that there are KV stores and workloads that significantly deviate from our model. It would be valuable to further extend our model, and also to take into account more specifics of prefetching [31] and of interconnects such as CXL [58]. In the meantime, since our operation model only requires there to be memory accesses and IOs, the model may also be applicable to those other than SSD-based KV stores including relational databases and file systems. Expanding the scope of our analysis and applications is one of the interesting future directions.

5.2.3 Tiered Memory for Indices and Caches. This work has studied the scenario where the entire indices and caches are offloaded to secondary memory. While we have demonstrated near-DRAM throughput in many cases, it would be useful to consider using both of the host DRAM and secondary memory for indices and caches to further mitigate performance degradation. Designing tiering and migration techniques specifically for microsecond-latency memory is another promising avenue for future work.

5.2.4 Commercial Availability. For lack of commercial memory devices having microsecond-level latency at the time of this work, we have used FPGA-based memory, and the commercial viability of microsecond-latency memory remains to be seen. However, devices equipped with slower memory media are emerging [52], and we hope our work encourages this trend by demonstrating their potential usefulness based on a research prototype. Even if the current trend does not lead to near-term commercialization, we believe our work can contribute to the informed decision the industry will be making. We also hope that, since memory hierarchy is a recurring theme in computer science, our work will add to that knowledge base for when hardware comes back to this trend in the near future again.

This work has shown that SSD-based KV stores are less susceptible to the slowdown coming from the limited prefetch queue depth P in hiding microsecond-level memory latency. The reason why P is small may be partly because current CPUs expect sub-microsecond memory latency only. If microsecond-latency memory becomes a commodity, future CPUs may support deeper queues, which would further alleviate the slowdown.

Our FPGA-based memory has been designed to introduce a user-specified fixed latency. Commercial microsecond-latency memory devices, if realized, would be different in that they would each have a unique tail latency profile depending on their memory media and its controller, and that they would likely implement an on-device cache such that latency would be much shorter upon cache hits. It would be valuable to study the impact of those features.

6   Related Work

The performance impacts of memory expansion and pooling are being actively studied [30, 32, 33, 40, 54, 57]. They typically deal with a memory latency of a few hundreds of nanoseconds, and tiering the host DRAM and secondary memory is often effective in curbing performance degradation [19, 30, 40, 46, 48]. However, some applications are latency-sensitive and suffer non-negligible performance degradation [33, 57], and mitigating them in general settings still requires a fair amount of host DRAM usage [19].

By focusing on KV stores, a number of hybrid KV store designs utilizing secondary memory have been proposed [6, 7, 15, 25, 27–29, 47, 61, 62, 64], which again assume sub-microsecond latency as they use Optane DCPMM. Notably, a few of them also use SSDs in addition to secondary memory [15, 28, 29, 62], thus studying the same memory-storage configuration as our work. Their goals are to improve one type of SSD-based KV store by proposing new techniques that take advantage of sub-microsecond-latency memory devices. Aside from using secondary memory, some KV stores aim to hide even shorter, host DRAM latency to speed up execution on the host DRAM [22, 24]. Other works explore different storage configurations including two-tier storage [43, 66] and KV-oriented storage devices and use [18, 38].

There are a few prior works that study much longer, microsecond-level memory latency outside of the application domain of SSD-based KV stores [11, 50, 55].

Our work is complementary to these prior works in that it studies the performance impact of microsecond-latency memory on varying types of SSD-based KV stores, with the goal of showing they can be made tolerant to microsecond-level memory latency with existing techniques through new analysis and evaluations.

7   Conclusion

This paper has studied the impact of using microsecond-latency memory for in-memory indices and caches in SSD-based KV stores. Our analysis has revealed that the presence of IO significantly enhances latency-tolerance of SSD-based KV stores, and our evaluation has demonstrated that they can achieve near-DRAM throughputs if they employ standard latency-hiding techniques. This indicates that SSD-based KV stores involving latency-sensitive in-memory traversal can use microsecond-latency memory as a low-cost alternative to DRAM. We believe these findings provide novel insights into the performance of SSD-based KV stores, and hope that they will encourage communities to expand the application scope of microsecond-latency memory.

Acknowledgments

References

[1] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast key-value stores: An idea whose time has come and gone. In *the Workshop on Hot Topics in Operating Systems (HotOS)*. ACM, 113–119.

[2] Aerospike. 2024. *Certifying Flash Devices (SSDs)*. Aerospike. Retrieved July 10, 2024 from https://aerospike.com/docs/server/operations/plan/ssd/ssd_certification

[3] Angelos Arelakis, Nilesh Shah, Yiannis Nikolakopoulos, and Dimitrios Palyvos-Giannas. 2024. Streamlining CXL Adoption for Hyperscale Efficiency. arXiv:2404.03551 [cs.ET] https://arxiv.org/abs/2404.03551

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM, 53–64.

[5] Jens Axboe. 2019. *Efficient IO with io_uring*. Retrieved July 10, 2024 from https://kernel.dk/io_uring.pdf

[6] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2013. Exploring storage class memory with key value stores. In *the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)*. ACM, 1–8.

[7] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proceedings of the VLDB Endowment* 14, 9 (May 2021), 1544–1556.

[8] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 769–786.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[10] Bob Chen. 2023. *200 Lines of Code to Rewrite the 600'000 Lines RocksDB into a Coroutine Program*. Alibaba Cloud. Retrieved November 11, 2024 from https://www.alibabacloud.com/blog/200-lines-of-code-to-rewrite-the-600000-lines-rocksdb-into-a-coroutine-program_599622

[11] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. 2018. Taming the Killer Microsecond. In *The 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Fukuoka, Japan.

[12] The CXL Consortium. 2024. *Compute Express Link*. The CXL Consortium. Retrieved July 10, 2024 from https://www.computeexpresslink.org/

[13] Intel Corporation. 2025. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*. Intel Corporation. Retrieved July 24, 2025 from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[14] Valgrind Developers. 2024. *4. Memcheck: a memory error detector*. Valgrind Developers. Retrieved July 10, 2024 from https://valgrind.org/docs/manual/mc-manual.html

[15] Chen Ding, Ting Yao, Hong Jiang, Qiu Cui, Liu Tang, Yiwen Zhang, Jiguang Wan, and Zhihu Tan. 2022. TriangleKV: Reducing Write Stalls and Write Amplification in LSM-Tree Based KV Stores With Triangle Container in NVM. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4339–4352. doi:10.1109/TPDS.2022.3188268

[16] Krijn Doekemeijer and Animesh Trivedi. 2022. *Key-Value Stores on Flash Storage Devices: A Survey*. Technical Report 2205.07975v1. Vrije Universiteit Amsterdam.

[17] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *the 19th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 33–49.

[18] Carl Duffy, Jaehoon Shim, Sang-Hoon Kim, and Jin-Soo Kim. 2023. Dotori: A Key-Value SSD Based KV Store. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1560–1572. doi:10.14778/3583140.3583167

[19] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 727–741.

[20] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 1037–1048. https://www.usenix.org/conference/atc22/presentation/elhemali

[21] Jack Hayya, Donald Armstrong, and Nicolas Gressis. 1975. A Note on the Ratio of Two Normally Distributed Variables. *Management Science* 21, 11 (1975), 1338–1341.

[22] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: coroutine-oriented main-memory database engine. *Proceedings of the VLDB Endowment* 14, 3 (Nov. 2020), 431–444.

[23] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 25–32. doi:10.1109/ICCD.2016.7753257

[24] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *Proceedings of the VLDB Endowment* 17, 3 (nov 2023), 577–590.

[25] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *2018 USENIX Annual Technical Conference (ATC)*. USENIX Association, 967–979.

[26] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 [cs.DC] https://arxiv.org/abs/1903.05714

[27] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *the 17th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 191–204.

[28] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 993–1005.

[29] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *2021 USENIX Annual Technical Conference*. USENIX Association.

[30] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. 2023. SMT: software-defined memory tiering for heterogeneous computing systems with CXL memory expander. *IEEE Micro* 43, 2 (March 2023), 20–29.

[31] Roland Kühn, Jan Mühlig, and Jens Teubner. 2024. How to Be Fast and Not Furious: Looking Under the Hood of CPU Cache Prefetching. In *Proceedings of the 20th International Workshop on Data Management on New Hardware (DaMoN)*. ACM, 1–9.

[32] Donghun Lee, Thomas Willhalm, Minseon Ahn, Suprasad Mutalik Desai, Daniel Booss, Navneet Singh, Daniel Ritter, Jungmin Kim, and Oliver Rebholz. 2023. Elastic Use of Far Memory for In-Memory Database Management Systems. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN)*. 35–43.

[33] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vol. 2. Vancouver, BC Canada, 574–587.

[34] Huiba Li, Rui Du, Sinan Lin, and Windsor Hsu. 2024. *Stackful Coroutine Made Fast*. Alibaba Cloud. Retrieved November 11, 2024 from https://photonlibos.github.io/blog-20241014/Stackful_Coroutine_Made_Fast.pdf

[35] John D. C. Little. 2011. OR FORUM—Little's Law as Viewed on Its 50th Anniversary. *Oper. Res.* 59, 3 (May 2011), 536–549. doi:10.1287/opre.1110.0940

[36] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S. Berger, and Huaicheng Li. 2024. Dissecting CXL Memory Performance at Scale: Analysis, Modeling, and Optimization. arXiv:2409.14317 [cs.OS] https://arxiv.org/abs/2409.14317

[37] Fabian Mahling, Marcel Weisgut, and Tilmann Rabl. 2025. Fetch Me If You Can: Evaluating CPU Cache Prefetching and Its Reliability on High Latency Memory. In *Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25)*. Association for Computing Machinery, New York, NY, USA, Article 2, 9 pages. doi:10.1145/3736227.3736231

[38] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. 2014. NVMKV: a scalable and lightweight flash aware key-value store. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems* (Philadelphia, PA) *(HotStorage'14)*. USENIX Association, USA, 8.

[39] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. 1991. First-class user-level threads. In *Proceedings of the thirteenth ACM symposium on Operating systems principles (SOSP)*. ACM, 110–121.

[40] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 742–755.

[41] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. ACM, 243–262.

[42] John C. McCallum. 2024. *Price and Performance Changes of Computer Technology with Time.* Retrieved November 11, 2024 from https://jcmit.net/index.htm

[43] Prashanth Menon, Thamir M. Qadah, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2022. LogStore: A Workload-Aware, Adaptable Key-Value Store on Hybrid Storage Systems. *IEEE Transactions on Knowledge and Data Engineering* 34, 8 (2022), 3867–3882. doi:10.1109/TKDE.2020.3027191

[44] Ethan L. Miller, George Neville-Neil, Achilles Benetopoulos, Pankaj Mehra, and Daniel Bittman. 2023. Pointers in Far Memory. *Commun. ACM* 66, 12 (Nov. 2023), 40–45. doi:10.1145/3617581

[45] Todd Mowry and Anoop Gupta. 1991. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel and Distrib. Comput.* 12, 2 (June 1991), 87–106.

[46] Yuanjiang Ni, Pankaj Mehra, Ethan Miller, and Heiner Litz. 2023. TMC: Near-Optimal Resource Allocation for Tiered-Memory Systems. In *the 2023 ACM Symposium on Cloud Computing (SoCC)*. ACM, 376–393.

[47] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *the 2016 International Conference on Management of Data (SIGMOD)*. ACM, 371–386.

[48] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *the Nineteenth European Conference on Computer Systems (EuroSys)*. 803–817.

[49] Ltd. Samsung Electronics Co. 2018. *Samsung Z-SSD SZ985.* Samsung Electronics Co., Ltd. Retrieved February 4, 2025 from https://download.semiconductor.samsung.com/resources/brochure/Brochure_Samsung_S-ZZD_SZ985_1804.pdf

[50] Shintaro Sano, Yosuke Bando, Kazuhiro Hiwada, Hirotsugu Kajihara, Tomoya Suzuki, Yu Nakanishi, Daisuke Taki, Akiyuki Kaneko, and Tatsuo Shiozawa. 2023. GPU Graph Processing on CXL-Based Microsecond-Latency External Memory. In *The SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. ACM, 962–972.

[51] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Adrián Castelló, Damien Genet, Thomas Herault, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Luk, Esteban Meneses, Marc Snir, Yanhua Sun, and Pete Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (March 2018), 512–526.

[52] Reza Soltaniyeh, Gongjin Sun, Caroline Kahn, Hingkwan Huen, Senthil Murugesapandian, Amir Beygi, Andrew Chang, Xuebin Yao, and Ramdas Kachare. 2025. *CMM-H: CXL-Enabled Large Scale Memory Expansion with a Hybrid NAND/DRAM Solution.* Samsung Electronics Co., Ltd. Retrieved July 15, 2025 from https://download.semiconductor.samsung.com/resources/white-paper/CMM-H_Whitepaper_10149503034923.pdf

[53] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. 2016. Aerospike: Architecture of a Real-Time Operational DBMS. *Proceedings of the VLDB Endowment* 9, 13 (Sept. 2016), 1389–1400.

[54] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *The 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 105–121.

[55] Tomoya Suzuki, Kazuhiro Hiwada, Hirotsugu Kajihara, Shintaro Sano, Shuou Nomura, and Tatsuo Shiozawa. 2021. Approaching DRAM performance by using microsecond-latency flash memory for small-sized random read accesses: a new access method and its graph applications. *Proceedings of the VLDB Endowment* 14, 8 (April 2021), 1311–1324.

[56] Dmitry Vinnik. 2021. *ELI5: Folly - Battle-Tested C++ Library.* Meta Platforms, Inc. Retrieved July 10, 2024 from https://developers.facebook.com/blog/post/2021/05/24/eli5-folly-battle-tested-c-plus-plus-library/

[57] Jacob Wahlgren, Maya Gokhale, and Ivy B. Peng. 2022. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. In *IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 11–20.

[58] Zixuan Wang, Suyash Mahar, Luyi Li, Jangseon Park, Jinpyo Kim, Theodore Michailidis, Yue Pan, Tajana Rosing, Dean Tullsen, Steven Swanson, Kyung Chang Ryoo, Sungjoo Park, and Jishen Zhao. 2024. The Hitchhiker's Guide to Programming and Optimizing CXL-Based Heterogeneous Systems. arXiv:2411.02814 [cs.PF] https://arxiv.org/abs/2411.02814

[59] Fred Ware, Javier Bueno, Liji Gopalakrishnan, Brent Haukness, Chris Haywood, Toni Juan, Eric Linstadt, Sally A. McKee, Steven C. Woo, Kenneth L. Wright, Craig Hampel, and Gary Bronner. 2018. Architecting a hardware-managed hybrid DIMM optimized for cost/performance. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '18)*. 327–340.

[60] Mark Webb. 2018. *Overview of Persistent Memory*. MKW Ventures Consulting, LLC. Retrieved November 11, 2024 from https://files.futurememorystorage.com/proceedings/2018/20180806_PreConC_Webb.pdf

[61] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (ATC)*. USENIX Association, 349–362.

[62] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: reducing write stalls and write amplification in LSM-tree based KV stores with a matrix container in NVM. In *the 2020 USENIX Conference on Usenix Annual Technical Conference (ATC)*. USENIX Association, 17–31.

[63] Fenghua Yu, Tony Luck, and Vikas Shivappa. 2016. *User Interface for Resource Control feature*. Intel Corporation. Retrieved July 24, 2025 from https://www.kernel.org/doc/html/v5.11/x86/resctrl.html

[64] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: a key-value store for Optane persistent memory. In *the Sixteenth European Conference on Computer Systems (EuroSys)*. 194–209.

[65] Chen Zhong, Qingqing Zhou, Yuxing Chen, Xingsheng Zhao, Kuang He, Anqun Pan, and Song Jiang. 2024. IndeXY: A Framework for Constructing Indexes Larger than Memory. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 516–529. doi:10.1109/ICDE60146.2024.00046

[66] Ruisong Zhou, Yuzhan Zhang, Chunhua Li, Ke Zhou, Peng Wang, Gong Zhang, Ji Zhang, and Guangyu Zhang. 2024. HyperDB: a Novel Key Value Store for Reducing Background Traffic in Heterogeneous SSD Storage. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 453–463. doi:10.1145/3673038.3673153