The Art of Latency Hiding in Modern Database Engines

Kaisong Huang
Simon Fraser University
kha85@sfu.ca

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

Qingqing Zhou
DB365000
sendtoqq@gmail.com

Qingzhong Meng
Tencent Inc.
freemeng@tencent.com

ABSTRACT

Modern database engines must well use multicore CPUs, large main memory and fast storage devices to achieve high performance. A common theme is hiding latencies such that more CPU cycles can be dedicated to "real" work, improving overall throughput. Yet existing systems are only able to mitigate the impact of individual latencies, e.g., by interleaving memory accesses with computation to hide CPU cache misses. They still lack the joint optimization of hiding the impact of multiple latency sources.

This paper presents MosaicDB, a set of latency-hiding techniques to solve this problem. With stackless coroutines and carefully crafted scheduling policies, we explore how I/O and synchronization latencies can be hidden in a well-crafted OLTP engine that already hides memory access latency, without hurting the performance of memory-resident workloads. MosaicDB also avoids oversubscription and reduces contention using the coroutine-to-transaction paradigm. Our evaluation shows MosaicDB can achieve these goals and up to 33× speedup over prior state-of-the-art.

PVLDB Reference Format:
Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. The Art of Latency Hiding in Modern Database Engines. PVLDB, 17(3): 577-590, 2023.
doi:10.14778/3632093.3632117

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfu-dis/mosaicdb.

1 INTRODUCTION

Various kinds of latency exist in modern database engines [13, 30, 41, 46, 55, 57] that target machines with large main memory, fast SSDs and multicore CPUs. When the working set fits in memory, memory is the primary home of data and various important data structures (e.g., indexes and version chains [62]) where memory blocks at random locations are chained using pointers. As a result, data stalls caused by pointer chasing in turn become a major bottleneck [27].

Since hardware prefetchers are ineffective for pointer chasing [5, 32, 54], modern CPUs offer prefetching instructions [24] that allow software to proactively move data from memory to CPU caches in advance. Importantly, such instructions are asynchronous, allowing software prefetching approaches to overlap computation and data fetching [5, 27, 32, 48]. Coupled with lightweight coroutines [25], *latency-optimized OLTP engines* [21] use software prefetching to

Proceedings of the VLDB Endowment, Vol. 17, No. 3 ISSN 2150-8097.
doi:10.14778/3632093.3632117

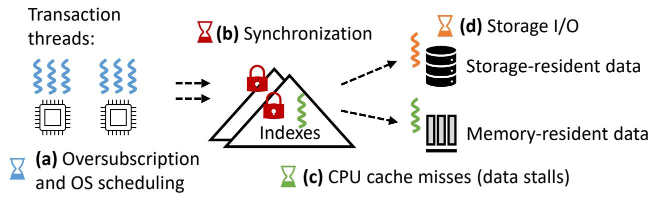


Figure 1: Main sources of latency in memory-optimized OLTP engines. Software prefetching can effectively hide data stalls (c). MosaicDB further mitigates the impact of other latencies (a, b, d) while preserving the benefits of software prefetching.

hide data stalls with the *coroutine-to-transaction* execution model in an end-to-end manner. In these systems, transactions are modeled as coroutines that are suspendable and resumable at various code locations. Each worker thread runs a scheduler that switches between coroutines (transactions) upon potential last-level cache misses, improving overall throughput by overlapping data fetching of one transaction with another's on-CPU computation. Notably, the cost of suspending and switching between stackless coroutines is very low (similar to calling a function), allowing the engine to gain net performance increase, sometimes by up to 2× [21].

1.1 Just Hiding Memory Latency Is Not Enough

Despite the significant performance improvement observed by end-to-end latency-optimized engines [21], these systems are still inadequate and leave many unexplored opportunities in further hiding more types of latency, as shown in Figure 1.

First, as data size grows, it becomes necessary to support larger-than-memory databases. As a result, in Figure 1(c–d) a transaction may access both memory- and storage-resident data. Yet existing latency-optimized OLTP engines based on software-prefetching are mostly designed to hide data stalls caused by memory accesses (0.1μs level) shown in Figure 1(c), without considering different levels of data movement latency, especially storage accesses at the 10μs to ms level. As we elaborate later, a direct addition of storage accesses to an end-to-end OLTP engine optimized for hiding memory latency would cancel out the benefit of software prefetching or even yield worse throughput than without using prefetching at all.

Second, in addition to the complexity caused by a mix of different data access latencies, the software architecture of a database engine can also induce latency during forward processing. As shown in Figure 1(a), CPU cores may be oversubscribed to run more threads than the degree of hardware multiprogramming, causing OS scheduling activities. The use of synchronization primitives can also lead to additional delays. For example, in Figure 1(b) the number of (re)tries (consequently, latency) to acquire a contended spinlock [53] could be arbitrarily long depending on the workload, making system behavior highly unpredictable.

These issues limit the applicability of latency-optimized database engines. Future engines should further address other sources of latency, and more importantly, do so while preserving the benefits of existing latency-hiding techniques, such as software prefetching.

1.2 MosaicDB: Latency Hiding at the Next Level

This paper presents MosaicDB, a multi-versioned, latency-optimized OLTP engine that hides latency from multiple sources, including memory, I/O, synchronization and scheduling as identified earlier. To reach this goal, MosaicDB consists of a set of techniques that could also be applied separately in existing systems.

MosaicDB bases on the coroutine-to-transaction paradigm [21] to hide memory access latency. On top of that, we observe that the key to efficiently hiding I/O latency without hurting the performance of memory-resident transactions is carefully scheduling transactions such that the CPU can be kept busy while I/O is in progress. To this end, MosaicDB proposes a pipelined scheduling policy for coroutine-oriented database engines. The basic ideas are (1) keeping admitting new requests in a pipelined fashion such that each worker thread always works with a full batch of requests, and (2) admitting more I/O operations to the system only when there is enough I/O capacity (measured by bandwidth consumption or IOPS). This way, once the storage devices are saturated, MosaicDB only accepts memory-resident requests, which can benefit from software prefetching. By carefully examining alternatives in later sections, we show how these seemingly simple ideas turned out to be effective and became our eventual design decision.

To avoid latency caused by synchronization primitives and OS scheduling, MosaicDB leverages the coroutine-to-transaction paradigm to regulate contention and eliminate the need for background threads. Specifically, each worker thread can work on multiple transactions concurrently, but only one transaction per thread will be active at any given time. This avoids oversubscribing the system by limiting the degree of multiprogramming to the amount of hardware parallelism (e.g., number of hardware threads). Consequently, the OS scheduler will largely be kept out of the critical path of the OLTP engine because context switching only happens in the user space as transactions are suspended and resumed by the worker threads. Using this architecture, MosaicDB also further removes the need for dedicated background threads (e.g., log flushers) which were required by pipelined commit [26] that is necessary to achieve high transaction throughput without sacrificing durability: cleanup work such as log flushes can be done using asynchronous I/O upon transaction commit, which will then suspend and be resumed and fully committed only when the I/O request is finished.

We implemented MosaicDB on top of CoroBase [21], a latency-optimized OLTP engine that hides memory latency using software prefetching. Compared to baselines, on a 48-core server, MosaicDB maintains high throughput for memory-resident transactions, while allowing additional storage-resident transactions to fully leverage the storage device. Overall, MosaicDB achieves up to 33× higher throughput for larger-than-memory workloads; with given CPU cores, MosaicDB is free of oversubscription and outperforms CoroBase by 1.7× under TPC-C; MosaicDB has better scalability under high contention workloads, with up to 18% less contention and 2.38× throughput compared to state-of-the-art.

Although MosaicDB is implemented and evaluated on top of CoroBase, the techniques can be separately applied in other systems. For example, contention regulation can be adopted by systems that use event-driven connection handling where the total number of worker threads will never exceed the number of hardware threads. We leave it as future work to explore how MosaicDB techniques can be applied in other systems.

1.3 Contributions

This paper makes four contributions. ❶ We quantify the impact of various sources of latency identified in memory-optimized OLTP engines, beyond memory access latency which received most attention in the past. ❷ We propose design principles that preserve the benefits of software prefetching to hide memory latency *and* hide storage access latency at the same time. ❸ In addition to memory and storage I/O, we show how software database engine architecture could be modified to avoid the latency impact of synchronization and OS scheduling. ❹ We build and evaluate MosaicDB on top of an existing latency-optimized OLTP engine to showcase the effectiveness of MosaicDB techniques in practice. MosaicDB is open-source at https://github.com/sfu-dis/mosaicdb.

2 BACKGROUND

We begin by clarifying the type of systems that our work is based upon and assumptions. We then elaborate the main sources of latencies that MosaicDB attempts to hide, followed by existing latency-optimized designs that motivated our work.

2.1 System Architectures and Assumptions

We target memory-optimized OLTP engines that both (1) leverage large DRAM when data fits in memory *and* (2) support larger-than-memory databases when the working set goes beyond memory.

Larger-Than-Memory Database Engines. There are mainly two approaches to realizing this. One is to craft better buffer pool designs [37, 46] which use techniques like pointer swizzling [18, 28] and low-overhead page eviction algorithms [37, 58] to approach in-memory performance when data fits in DRAM, while otherwise providing graceful degradation and fully utilizing storage bandwidth. The other approach employs a "hot-cold" architecture [14, 31] that does not use a buffer pool, and separates hot and cold data whose primary homes are respectively main memory and secondary storage (e.g., SSDs). In essence, a hot-cold database engine consists of a "hot store" that is memory-resident (although persistence is still guaranteed) and an add-on "cold store" in storage. A transaction then could access data from both stores. However, note that both "stores" use the same mechanisms for such functionality as concurrency control, indexing and checkpointing, inside a single database engine without requiring cross-engine capabilities [64]. In this paper, we focus on the hot-cold architecture and leave it as future work to explore the buffer pool based approach.

Hot-Cold Storage Engines. Figure 2 shows the design of ERMIA [30], a typical hot-cold system that employs multi-versioned concurrency, in-memory indexing and indirection arrays [50] to support in-memory data (hot store) and storage-resident data (cold store). Many other systems [13, 14, 39, 40] follow similar designs. An update to the database will append a new in-memory record version to the indirection array and generate a log record in the log buffer which will later be flushed to the storage device. That is, data records are permanently stored in secondary storage in the form of logs, which are periodically compacted and consolidated, i.e., the log is the database. Within a table, each record is uniquely identified by a record ID (RID). Each table is represented by an indirection array which is a resizable array where each entry carries the address (in memory or storage) of a unique data record. An index then maps keys to RIDs, which in turn are indexes into the indirection array. Note that different from RIDs in traditional systems [49], RIDs here are *logical* without record address information, which must be retrieved through the indirection array.

With such architecture, a transaction can access a record in three steps: (1) traverses the index to obtain the RID *i*, (2) examines the table's indirection array using *i* as the index, and (3) accesses the desired record data in memory or from storage. Many such designs are also multi-versioned [30, 36, 41, 47, 61, 62] to extract more concurrency. As a result, in Figure 2, an indirection array entry can point to a version chain (a linked list of versions) or an address in storage. In the former case, in step (3) the transaction traverses the version chain to retrieve the desired record version as dictated by the concurrency control protocol, such as snapshot isolation. In the latter case, the transaction must issue an I/O to access the cold record, which can be cached using various strategies. Some systems [14] use per-thread caching of cold data. Such design decisions are orthogonal to our work.

Since the hot store assumes DRAM is large enough to hold at least the working set, recent memory-optimized OLTP engines [21, 30, 31, 57] also often employ redo-only logging without using a buffer pool. This way, updates (log records) by aborted transactions are discarded without ever reaching storage. The logs then store actual data generated by committed transactions. The system can recover by replaying the logs after applying a previous checkpoint (if any). For cold records, the recovery process mainly needs to fill the indirection arrays with record addresses in storage (the log) without having to materialize any data record. During forward processing, accesses to cold records can be done on demand by reading the log at addresses carried by the corresponding indirection array entries.

2.2 Where Can Latencies Come From?

We identify and analyze four main sources of latency, given the hot-cold architecture and assumptions described in Section 2.1. We focus on hiding these latencies in current mainstream database servers and discuss latencies that may arise in other environments.

Pointer Chasing. Modern database engines use various in-memory data structures that are directly addressed by virtual memory pointers. Memory blocks used by these data structures are usually allocated from the heap and chained together using pointers. For example, the nodes of an in-memory B+-tree are allocated and deallocated as the tree grows and shrinks. To traverse a tree from its root node to the target leaf node, the accessing thread must dereference multiple pointers from the root to the leaf node, forming a random access pattern. Unfortunately, such patterns are very difficult (if not impossible) for hardware prefetchers to predict accurately, leading to very high likelihood of last-level cache

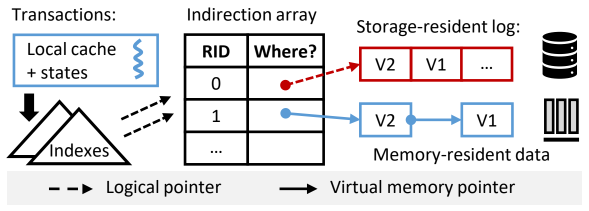


Figure 2: The "hot-cold" store architecture of MosaicDB. Indexes map keys to RIDs. Per-table indirection arrays are indexed by RIDs and store record locations.

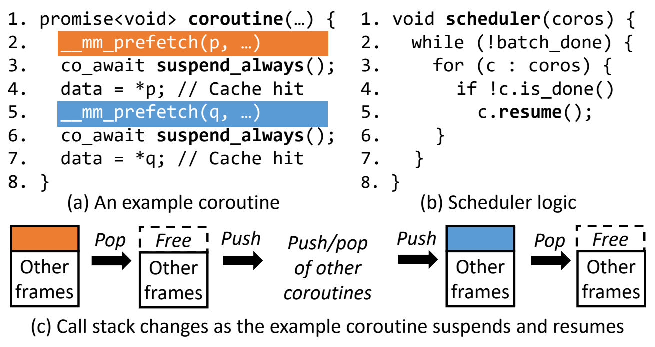
misses[1] upon pointer dereference. As a result, in main-memory systems data stalls often dominate total CPU execution time. Beyond indexes, version chains in multi-versioned systems as described in Section 2.1 can also cause a significant amount of data stalls when the accessing thread searches for a particular version. In some systems, over 50% of the total CPU cycles can be spent on waiting for data to be loaded from memory to CPU caches [21]. Hiding such stalls can potentially lead to much higher overall performance.

Synchronization. In-memory data structures require proper synchronization for correctness. For example, shared index node states are often protected by latches (spinlocks or mutexes). Under contention, a small number of latches may by accessed by a large number of threads. Compared to low contention or uncontended cases, acquiring a latch that is under contention costs more CPU cycles by retrying atomic instructions such as compare-and-swap (CAS) [24]. Such retries can be unbounded without guaranteed progress or success in a finite number of steps, causing long delays and unfairness among transactions.

Storage I/O. As data size grows, it is necessary to allow transactions to access cold or a mix of hot and cold records. Naturally, this would require the accessing transaction to issue I/O requests to load data from storage (if not already cached). Despite of recent advances in fast storage, such as NVMe SSDs [22], storage I/O is still orders of magnitude slower than memory accesses. Worse, I/O requests are often done *on the critical path* via synchronous I/O primitives (e.g., `pread`) as the data is needed right away by the requesting transaction. Some systems, such as MySQL, alleviate this issue by prefetching data in the background using dedicated I/O threads and asynchronous I/O primitives (e.g., AIO [45]). On the one hand, it is difficult to accurately predict the workload and prefetch exactly the needed data from storage.[2] On the other hand, using dedicated threads can also cause (1) inter-thread communication overhead because the transaction worker thread must notify and be notified by the I/O threads to initiate and complete I/O requests, and (2) oversubscription overhead which we elaborate next.

Oversubscription/OS Scheduling. CPU cycles are precious resources that must be well used. In systems that must handle larger-than-memory databases, it is desirable to overlap on-CPU compute (e.g., in-memory transactional logic) with I/O operations in the background. A straightforward and widely adopted approach is to oversubscribe the hardware and leverage the OS for scheduling,

[1] Unless otherwise specified, throughout the paper *cache misses* refers to last-level misses that mandate accessing memory.
[2] Not to be confused with prefetching from memory to CPU caches.



```
1. promise<void> coroutine(…) {
2.   __mm_prefetch(p, …)
3.   co_await suspend_always();
4.   data = *p; // Cache hit
5.   __mm_prefetch(q, …)
6.   co_await suspend_always();
7.   data = *q; // Cache hit
8. }
```

(a) An example coroutine

```
1. void scheduler(coros) {
2.   while (!batch_done) {
3.     for (c : coros) {
4.       if !c.is_done()
5.         c.resume();
6.     }
7.   }
8. }
```

(b) Scheduler logic

(c) Call stack changes as the example coroutine suspends and resumes

Figure 3: Software prefetching using C++20 coroutines. A coroutine (a) can suspend and be resumed by the scheduler (b). Coroutine frames are popped from/pushed onto the caller thread's stack as they are suspended/resumed (c).

i.e., spawning more software threads than the number of hardware threads (hyperthreads) or physical cores. The OS then transparently schedules worker threads depending on whether they are handling I/O. While a thread is asleep waiting for (synchronous) I/O to finish, another thread can be scheduled to run on the CPU, improving overall CPU utilization. The downside is that the OS scheduler can require a non-trivial amount of CPU cycles for itself, canceling out the benefits of overlapping compute and I/O of different threads. This is especially true for modern fast NVMe SSDs that exhibit a narrowing gap between memory performance [22]. Systems that use dedicated I/O threads have similar issues. Although these threads run "in the background," they still consume actual CPU cycles (especially on fast SSDs which often prefer spinning rather than interrupts [22]) and may compete with transaction worker threads in environments with a fixed budget (e.g., in the cloud), again triggering OS scheduler activities.

Other Latencies and Environments. We find the above latencies are dominant in memory-optimized hot-cold engines. Beyond such engines, other important latencies can arise. For example, a system that uses pessimistic concurrency control may suffer from latency caused by logical lock contention, in addition to synchronization latency caused by latches. In contrast, memory-optimized systems typically use optimistic approaches to concurrency control, mitigating the impact of lock-induced latency. Networking delays can also lead to extra latency in distributed database engines.

We focus on single-node, memory-optimized engines built for mainstream database servers that are typically dual- or quad-socket. Latencies caused by memory accesses and synchronization can be even more visible at larger scales (e.g., on servers with over 1000 cores across tens of sockets [1]). We leave it as future work to hide other latencies and explore other environments at larger scales.

2.3 State-of-the-Art and Motivation

Many previous efforts have been dedicated to hiding either memory [5–7, 15, 44, 48] or storage [2, 33] access latencies. With software prefetching, state-of-the-art database engines use cooperative multi-tasking enabled by C++ coroutines [25] to hide memory access latency [21, 27, 48]. Figure 3 depicts the approach. Coroutines are functions that can suspend (give up CPU time) voluntarily and be resumed later by a user-space scheduler as needed. This gives the opportunity for OLTP engines to overlap compute and memory accesses of different transactions, leading to the recent coroutine-to-transaction execution paradigm [21]. Each worker thread runs a user-space scheduling logic that takes incoming transactions in batches and each transaction is modeled as a coroutine that will suspend execution upon possible cache misses. For example, in Figure 3(a), a transaction may invoke a coroutine to read records in memory and suspends its execution (line 3) after issuing an asynchronous prefetching instruction [24] (line 5). The scheduler in Figure 3(b) serves transactions in a round-robin manner and switches to and resumes the next transaction T in the batch, hoping to overlap its compute with the just-suspended transaction's prefetching. T may suspend later for the same reason. Importantly, C++20 coroutines are stackless. As Figure 3(c) shows, the coroutine frame directly (re)uses the underlying thread's stack. This allows fast switching with overhead that is cheaper than a last-level cache miss whereas traditional solutions [52] use their own stacks. Switching between them involves high overhead that defeats the purpose of software prefetching.

Compared to the traditional sequential execution model where a thread executes one transaction at a time, coroutine-to-transaction keeps multiple transactions open per thread. This can increase the conflict window and even cause deadlocks using a single thread due to transactions on the same thread conflicting with each other. In practice, however, many memory-optimized systems [30, 31, 41, 42, 57] already use optimistic concurrency [34, 38] that avoids these issues, making coroutine-to-transaction a natural fit. Our work also assumes optimistic concurrency, following prior work [21].

Coroutine-to-transaction has proven to be effective for hiding memory access latency, but still lacks the ability to hide storage I/O latency, which is typically done by using asynchronous I/O primitives (e.g., io_uring [9]). As we discuss later, blindly combining asynchronous I/O and coroutine-to-transaction can lead to poor performance. Some systems leverage asynchronous I/O to overlap storage and compute, but they typically do so using heavyweight mechanisms, such as stackful coroutines [11, 43, 52] and OS threads. Compared to stackless coroutines, these mechanisms bring non-trivial overheads that will cancel out the benefits of software prefetching [27]. These issues call for a new approach that allows asynchronous I/O to co-exist with and not canceling out the effect of memory latency hiding techniques. It was also unclear how synchronization and oversubscription/OS scheduling overheads can be mitigated under the coroutine-to-transaction paradigm. MosaicDB thus aims to enable latency hiding for both memory and storage accesses, and at the same time leverage coroutines to mitigate the impact of oversubscription and synchronization.

3 MOSAICDB OVERVIEW

MosaicDB is a multi-versioned hot-cold OLTP engine that optimizes for both in-memory and out-of-core data accesses, while mitigating the impact of oversubscription and synchronization latencies. At its core is a new *out-of-core coroutine-to-transaction* paradigm that still models transactions as coroutines like prior work [21], but additionally (1) allows transactions that access storage-resident data to use asynchronous I/O without extra inter-thread communication overhead mentioned in Section 2.2, (2) does so without sacrificing the benefits of software prefetching, and (3) naturally avoids oversubscription overhead and regulates contention. As shown in Figure 4, MosaicDB ensures that each core (or hyperthread) runs only one software thread to avoid oversubscription and heavyweight OS scheduler activities. ❶ With the coroutine-to-transaction paradigm, each worker thread—instead of directly running one transaction at a time—takes multiple transactions which are modeled as coroutines and switches between them as needed. ❷ Each transaction coroutine in turn invokes the corresponding coroutines for handling data accesses which may traverse indexes and version chains. Depending on whether the target data is from the hot or cold store, the transaction may only need to suspend its execution upon possible cache misses or ❸ further issue asynchronous I/O requests (e.g., using io_uring or Linux AIO).[3] ❹–❺ After the current transaction is suspended, control is returned to the scheduler which ❻ continues to handle the next request and repeats this process. When the scheduler resumes a previously suspended transaction, it may continue executing the transaction by dereferencing a pointer to the data in memory (if the requested data is in the hot store), or checking whether the asynchronous I/O operation has finished.

MosaicDB inherits the relevant designs in CoroBase [21] to guarantee durability and maintain ACID properties. This is done using redo-only logging as described in Section 2.1 and pipelined/group commit [26]. During forward processing, transactions generate log records in memory and are only committed after the log records are persisted. Upon commit, the underlying thread still decouples the transaction and places it on a (partitioned) commit queue. Different from prior approaches, however, MosaicDB does not use background threads to monitor and release transactions from the commit queue (i.e., truly commit the transaction with results returned to the application). Instead, this is checked by the worker thread itself in between requests lazily: a log flush is issued (using asynchronous I/O) whenever the log buffer is full or times out. The worker thread then checks for log I/O status the next time it accesses the log buffer. If the I/O has finished, it then examines the commit queue to release transactions whose log records are persisted. This way, MosaicDB also avoids oversubscribing the CPU by not using additional background threads like previous work [26, 64].

4 MITIGATING DATA ACCESS LATENCY

Prior coroutine-to-transaction engines focused on hiding stalls caused by in-memory transactions [21] (i.e., those that only access the hot store). The in-memory setup mandated flattening nested function/coroutine calls yet allowed simple batch scheduling policies. However, the presence of cold store transactions requires a departure from these designs. The rest of this section describes how MosaicDB caters such transactions while maintaining the effect of software prefetching for accessing memory-resident data.

[3]We refer to I/O mechanisms that do not block the issuing thread (e.g., io_uring and Linux AIO) as "asynchronous I/O." This is not to be confused with "asynchronous commit" which allows transactions to commit without persisting their log records.

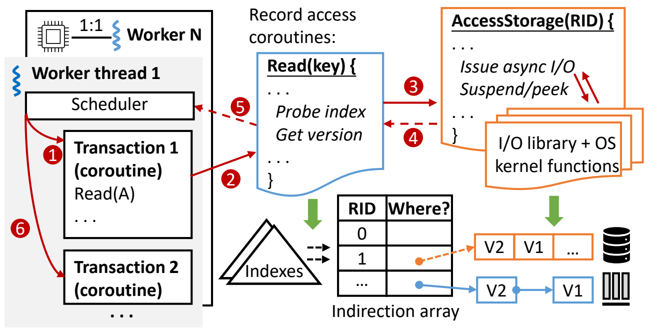


Figure 4: Overview of MosaicDB. Each core/hyperthread runs one thread to schedule transactions in a pipelined manner. Transactions (coroutines) suspend and resume upon cache misses and I/O to overlap compute and data access latency.

4.1 Selective Coroutine Nesting

Database engines use nested function call chains to modularize their implementations. Given an engine implementation, one may simply change the function and return types (e.g., using `co_return` instead of `return`) to convert the necessary functions to coroutines, in addition to adding suspend/resume calls. Nested function call chains then become nested *coroutine* call chains. With proper coroutine library support, any suspend operation inside a coroutine will cause the calling coroutine to be popped out of the stack with control eventually returned to the scheduler which then switches to the next request. Although transforming function call chains into fully-nested coroutines is simple, it brings non-trivial overhead that cancels out the benefits of software prefetching. To reduce coroutine switching overhead, recent approaches [21] suggest a two-level design where function call chains in the storage engine are "flattened" to become a single coroutine, which can be called by transaction coroutines. While such two-level coroutine-to-transaction increases the size of a single coroutine and can increase instruction cache misses, these costs are outweighed by the benefit, allowing coroutine-to-transaction to effectively hide data stalls, since fewer cycles are spent on coroutine switching.

Hiding I/O latency makes the function call chains even deeper by involving additional functions that handle I/O requests (e.g., issuing and checking completion of asynchronous I/O, followed by deserialization). In theory, such additional call chains should be inlined to maintain the aforementioned two-level coroutine-to-transaction structure for the lowest overhead, given it is also relatively straightforward to do so for in-memory data accesses which mostly only involve traversing an index and a version chain to access a record in the hot store. However, we observe that this is neither easy nor necessary. On the one hand, such flattening is usually done manually and the additional storage-related call chain can be complex. Inlining all of them can further bloat code with worse instruction cache utilization and make the code hard to maintain. On the other hand, storage access latency is still orders of magnitude longer than memory latency. This means we need to switch to transaction coroutines that access I/O much less frequently, amortizing the switching overhead.

These observations lead to a simple but useful *selective coroutine nesting* approach that only allows nested coroutines for storage-related functions: Based on an existing two-level coroutine-to-transaction design, we instrument the storage-related functions to become nested coroutines and keep the two-level flattened coroutines for memory accesses. The result is that a transaction will work exactly the same as before until it accesses storage, by invoking inlined record access coroutines in Figure 4. If the transaction accesses storage-resident data, as shown in Figure 4, with the corresponding indirection array entry which points to a location in storage, the record access coroutine will issue asynchronous I/O and suspend. Control is then returned to the scheduler which will later check whether the I/O has finished and resume the transaction coroutine if so. After that, the data read from storage is converted into a node in the in-memory version chain for future accesses.

Our implementation uses io_uring [9]; other asynchronous I/O libraries can also work with MosaicDB as long as they present an asynchronous interface that allows the application to separately issue and check for I/O completion. With io_uring, each worker thread has a thread-local I/O module where there is a "ring" for I/O operations. I/O requests are processed using submission/completion queues. The submission queue in each ring has a fixed number of slots for submission queue entries (SQEs), defined when the ring is initialized. Therefore, all transactions on the same worker thread share this ring to access storage. An I/O request is issued by creating an SQE and completions are indicated by completion queue entries (CQEs). The latter can complete out of order respective to the former. We therefore tag each SQE with the issuing transaction's ID to distinguish different transactions' CQEs.

Now we walk through the process of reading a record using the example shown in Figure 4. (a) Worker thread 1 first starts Transaction 1 (a coroutine) and begins to read the record matching key A (step ❶ in the figure). (b) It then probes the index, during which process we issue `prefetch` instructions followed by `suspend` statements (❷). (c) Control then returns to the scheduler (❺). (d) Depending on the scheduler's logic, it resumes Transaction 2 to continue from where it left off (❻), (e) until it encounters the next `suspend` or concludes. At some point during the execution of Transaction 1, Thread 1 gets the pointer corresponding to RID N and finds that it points to a a location in storage, (f) so the thread issues an asynchronous I/O (❸), immediately suspends the transaction after submitting an SQE, and returns to the scheduler (❹–❺). Each time the scheduler wants to resume Transaction 1, it first peeks at the completion queue to look for the next completed I/O request, which occurs in the user space, and resumes the transaction the completed request belongs to, which is not necessarily Transaction 1. (g) The worker thread gets the I/O request belonging to Transaction 1, retrieves the request, and marks the CQE consumed. Next, we discuss the scheduling policy used in the above process.

4.2 Basic Storage-Aware Batch Scheduling

Traditional coroutine-to-transaction was designed for in-memory OLTP workloads that exhibit similar transaction profiles, without considering the fact that modern OLTP workloads are increasingly heterogeneous with varying transaction types in addition to short, memory-only transactions (e.g., operational reporting [4] and storage accesses). As a result, the worker thread simply gathers a batch of requests at a time, and then switches between them upon pre-defined `suspend` calls. A new batch cannot be admitted until all the transactions in the current batch are concluded (committed or aborted). Such traditional batch scheduling [21, 27, 48] has been widely adopted by existing coroutine-based systems, but exhibits two fundamental issues for mixed memory/storage workloads, which we address in the rest of this section. We start by proposing storage-aware batch scheduling, a simple adaptation of traditional batch scheduling, and then discuss how we evolve it towards the desirable scheduling policies.

At a first glance, it may seem trivial to extend traditional batch scheduling without any change by simply suspending the transaction coroutine once an asynchronous I/O is issued. The first issue with this approach is that as I/O request status is part of the transaction state, the scheduler has to resume the coroutine to find out whether a previous in-progress I/O associated with the transaction has finished. If the I/O is still in-progress, the scheduler would have to switch to the next transactions, in essence wasting the work to resume a transaction. This was not a significant problem in pure-memory environments because typically modern x86 processors allow only a smaller amount of outstanding asynchronous memory loads, which makes it possible to make very accurate "educated guesses" about when the data will be fetched into the cache and batch size. Thus, in most cases, after a transaction is resumed, its requested data is indeed in the CPU cache, making the (already low) switching overhead worthwhile. Yet for the I/O case, recall that it is impractical to inline the I/O stack and I/O latency is still orders of magnitude higher than memory access latency. The former increases the cost of coroutine switching. Coupled with the latter, the two properties mean that when I/O and memory accesses are mixed in a batch of transactions, most of the cycles spent on checking asynchronous I/O completion state are wasted.

Our first scheduling policy (storage-aware batch scheduling) resolves this issue by decoupling I/O status tracking and transaction context. As shown in Figure 5(a), for each worker thread, we separately allocate an array of I/O status tracking structures,[4] each of which corresponds to a transaction in the current batch being processed by the thread. The scheduler then still works in the same way as before, but before resuming a transaction, it first checks whether the transaction was suspended because of an asynchronous I/O request. If so, it then directly checks the thread-local I/O status without resuming the transaction and only resumes the transaction if the I/O has completed; otherwise it directly proceeds to the next transaction. If the transaction was suspended due to a potential CPU last-level cache miss, the scheduler still directly resumes it because currently software cannot query CPU cache status.

4.3 Pipelined Scheduling

Storage-aware batch scheduling avoids wasting CPU cycles on resuming/suspending storage-bound transactions. However, since it uses a fixed-size batch and only processes transactions by full batches, new (memory-resident) transactions may not be scheduled to run timely while there are in fact enough CPU cycles. Consider

[4]io_uring contexts in our current implementation. Each I/O request is tagged with the issuing transaction ID to distinguish I/O completions for different transactions.

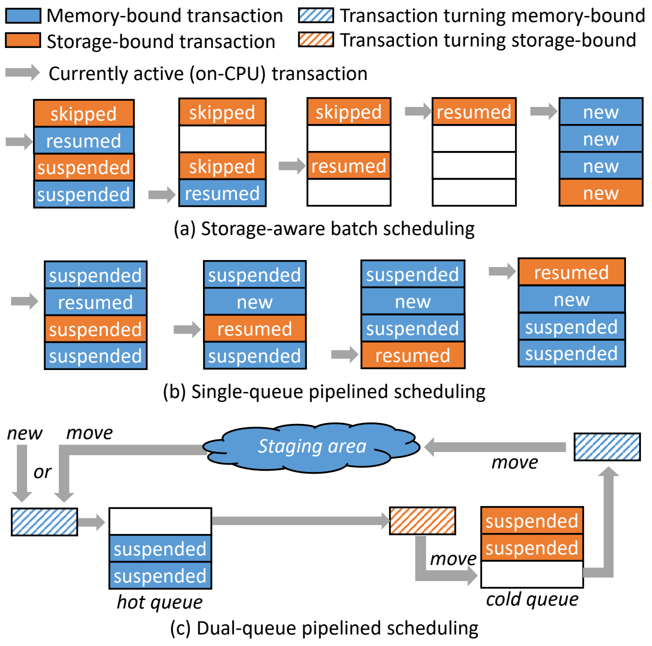


Figure 5: Scheduling policies. (a) Storage-aware batch can selectively resume storage-bound transactions but can lead to low CPU utilization and throughput. (b) Vanilla pipelined improves CPU utilization but can starve in-memory transactions. (c) Dual-queue pipelined isolates memory- and storage-bound transactions to fully utilize storage bandwidth while maintaining high in-memory transaction performance.

a batch of ten (which is typical) where four of the transactions are I/O bound and six are memory-bound. Since I/O exhibits much higher latency, it is likely that the more lightweight memory-bound transactions will conclude earlier than the storage-bound ones. With batch scheduling, six out of ten slots in the batch will be vacant until the storage-bound transactions conclude much later; memory-bound transactions would have to wait to be admitted in the next batch, lowering CPU utilization and overall throughput and affecting the performance of memory-resident transactions.

Vanilla Transaction Pipelining. To improve CPU utilization, MosaicDB instead processes transactions in a pipelined fashion that allows individual new transactions to be admitted as long as there is free slot in the batch. This allows more memory-bound transactions to be processed as I/O is in-progress for the rest of the batch. As Figure 5(b) shows, whenever a transaction is concluded, a new request can be admitted to the queue. Such vanilla transaction pipelining was first discussed in CoroBase [21], but was deemed unnecessary for pure in-memory workloads. Under mixed memory/storage workloads, however, more in-memory transactions can slip through to maintain high throughput. The downside of this policy is that there is no admission control for transactions and the system can be easily dominated by storage-bound transactions, starving memory-resident data accesses. For example, as shown in Figure 5(b), storage-bound transactions can occupy the slots for longer time, and as they keep getting admitted, they will eventually

Algorithm 1 Dual-Queue Pipelined Scheduling.

```
def DualQueuePipeline():
  [hot] = get_transaction_requests()
  while not shutdown:
    if hot[i].is_done():
      hot_txn_done++
      if interval.is_up() or hot.empty():
        hot_txn_done = 0
        for j = 0 to cold_queue_size - 1:
          if cold[j].is_done():
            idle_slots_in_cold_queue.push(j)
          else if cold[j].is_hot():
            staging.push(cold[j])
            idle_slots_in_cold_queue.push(j)
          else:
            tid, read_size = peek()
            if valid_tid(tid) and
                read_size = expected_size:
              cold[tid].resume()
      fetch_transaction()
    else if hot[i].is_cold():
      cold[idle_slots_in_cold_queue.pop()] = hot[i]
      fetch_transaction()
    else:
      hot[i].resume()
    i = i % hot_queue_size
```

dominate all the slots even though they only constitute a small portion of the workload mix. As a stop-gap solution, the system can prioritize memory-bound transactions by reserving a certain number of slots. This way, if the number of storage-bound transactions exceeds a threshold, they will not be admitted and must wait until a slot is freed up.[5] However, modern SSDs exhibit much higher bandwidth and often requires high queue depth (i.e., number of parallel requests) for software to extract their full potential. As a result, the queue can become very long, requiring the scheduler logic to scan through a long (sub)array of I/O states. This essentially increases memory prefetching distance, eventually canceling out the benefits of software prefetching for memory-bound transactions. Thus, it is desirable to keep the queue short, while still allowing saturating the storage device with enough many I/O slots.

Dual-Queue Transaction Pipelining. To solve these issues, we advocate a pipelined design that decouples memory- and storage-bound transactions. Each thread is associated with two queues respectively for memory- and storage-bound transactions, in addition to a staging area, shown in Figure 5(c). If a previously memory (storage) bound transaction on the memory (storage) queue turns to access the cold (hot) store, it is then moved the the corresponding queue. In our current implementation, all the new transactions start with a hot-store access because the indexes are in-memory. The worker thread always works on the memory queue unless (1) it finds a cold request, (2) a pre-defined interval is up, or (3) the memory queue is empty. Algorithm 1 shows the details. In case (1), the

[5] Most systems already feature similar admission control features, which can be modified to work with MosaicDB, so we do not repeat here.

worker thread moves the storage-bound transaction to the storage queue and then admits another transaction to fill up the slot. Both cases (2) and (3) happen after a transaction is concluded (line 7 in Algorithm 1). The predefined interval in case (2) records the number of transactions processed from the hot queue (`hot_txn_done` in Algorithm 1) and is configurable. Each time a transaction is completed in the memory queue, `hot_txn_done` is incremented. After that, if the counter reaches a predefined threshold or the memory queue is empty, the worker thread will iterate over the cold queue and process transactions by checking whether their in-progress I/Os have completed (same as the storage-aware batch scheduling policy in Section 4.2). This way, the worker thread can attend to memory-bound transactions more promptly, while storage-bound transactions are checked and resumed less frequently to match storage I/O capabilities.

Since a transaction can transition between memory-bound and storage-bound, a transaction on the memory queue may need to find a slot on the storage queue when the latter is already full. In this case, we abort the transaction so as to ensure there is enough slots left to process memory requests. Conversely, a storage-bound transaction may become memory-bound again after processing I/O. It will then be placed back to the memory queue if there is a vacant slot. Otherwise, we place it in the staging area and prioritize transactions from it (over other requests) to be admitted to the memory queue once it has space. This allows us to keep the benefits of software prefetching while extracting the full potential of the storage device. The sizes of the queues need to be tuned according to the target workload and the underlying storage device's capability (bandwidth and IOPS), but we believe this is not a major limiting factor of MosaicDB as the parameter space is not large: the memory queue should be kept similar to batch sizes used in previous in-memory coroutine-to-transactions (8–10) for the two-level coroutine switching mechanism to work well, and the storage queue only needs to be tuned when storage capability (bandwidth/IOPS) changes (e.g., by upgrading to a faster SSD).

5 MITIGATING OVERSUBSCRIPTION AND SYNCHRONIZATION LATENCY

Without userspace scheduling like MosaicDB, many systems leave it to the OS to schedule transactions and oversubscribe the system to improve CPU utilization. Some memory-optimized database engines already try their best to not oversubscribe the system. They typically use at most one software worker thread per hardware hyperthread (or per physical core). Even so, oversubscription can still happen with various background threads, such as log flushers, group commit threads, and checkpointing threads [26]. With a fixed CPU budget (e.g., in the cloud), these background threads still require a non-trivial amount of CPU cycles and can cause OS scheduler activities that will lower throughput. Note that although some background threads only run periodically (e.g., checkpointing), some will run frequently. We take the widely used parallel logging [60, 63] and pipelined commit [26] mechanisms as an example. With parallel logging, each worker thread accumulates log records in a local buffer and then flushes them to storage upon (group) commit. Further, pipelined commit decouples transactions and threads, such that log flushing does not block the thread from taking the next request. The system then needs to keep pre-committed transactions whose log records have not reached storage, and only notify the client after the log records are persisted. This process is often accomplished using background flusher/committer threads: after the worker thread finishes running the transaction logic, it places the transaction on a commit queue (or a partitioned queue to ease contention) and continues to process the next request. The commit queue is monitored by a committer thread that periodically checks log status and retires transactions from the commit queue as their log records are persisted. Log status is maintained by another flusher thread that invokes I/O primitives (e.g., `pwrite`) to flush log records submitted by worker threads. As we will see in Section 6, this design can significantly lower throughput under high load as the background threads compete with foreground worker threads.

MosaicDB leverages the coroutine-to-transaction architecture to eliminate background threads all together, avoiding oversubscription. All the threads in MosaicDB are worker threads and transaction logging/commit operations are processed in exactly the same way as "normal" I/O operations during forward processing. Upon commit, the worker thread appends the transaction's log records to a thread-local log buffer, and if necessary (e.g., the buffer is full), issues an asynchronous I/O to flush the log. The scheduler logic then switches to the next transaction while I/O is in-progress, and when it resumes the same transaction, it additionally checks whether the transaction's log records have been made persistent (by comparing the current durable and the transaction's commit log sequence numbers). If so, it resumes the transaction to finish post-processing (e.g., finalizing its newly generated versions). This way, all the additional I/O requests associated with transaction commit are handled by worker threads. Moreover, the scheduling queues in Section 4 replaces the commit queue in past systems, saving resources and avoiding a potential source of contention. Other work (e.g., checkpointing) can be performed using system transactions [17] which are then processed in similar ways.

MosaicDB's coroutine-oriented architecture also allows to limit contention while maintaining a high degree of concurrency. Each worker thread can keep multiple transactions open, yet only one of the transactions will be actively running at any time, while others are suspended waiting for data to be fetched to the CPU cache or memory. Compared to systems with oversubscription, MosaicDB limits contention naturally to the degree of multiprogramming, i.e., at most there will be the same number of hyperthreads/cores that contend for a shared resource, keeping OS scheduler out of the critical path. Without oversubscription and excessive contention, individual data structures, e.g., indexes, can then focus on handling contention only up to the physical resources available. Various solutions already exist and MosaicDB can easily adopt them [3].

6 EVALUATION

In this section, we evaluate MosaicDB under various workloads that exhibit latency from memory/storage accesses, CPU core oversubscription and synchronization. Through experiments, we confirm:

- MosaicDB can hide storage access latency without drastically impacting in-memory transaction throughput. Meanwhile, it yields higher throughput for storage-bound transactions than the traditional thread-to-transaction execution model does.

- MosaicDB can avoid CPU core oversubscription by removing background threads altogether, improving performance.
- MosaicDB effectively mitigates synchronization overhead under high contention by limiting contention levels.

6.1 Experimental Setup

Hardware and Software. We use a dual-socket server equipped with two 24-core Intel Xeon Gold 6342 CPUs clocked at 2.80 GHz (up to 3.50 GHz with turbo boost). The CPU has 36MB of caches. In total the server has 256GB of main memory occupying all the six channels per socket to maximize memory bandwidth. We use three SSDs in the server in our evaluation to understand the impact of storage devices: a 500GB Samsung 980 PRO [51], a 375GB Intel Optane SSD DC P4800X [23], and a 480GB Dell SATA SSD [12]. Using `fio`, we observe that for random accesses, the Samsung/Intel/Dell SSDs can deliver up to 860K/490K/125K IOPS, respectively. Unless otherwise specified, we use the Samsung SSD. All the data is persisted on the storage device, with data in the hot store also kept in-memory. For all experiments, hyperthreading is disabled and direct I/O mode (`O_DIRECT`) is enabled to simplify the interpretation of results. The server runs Ubuntu Linux 22.04.3 LTS with kernel 5.15. In our experiments, we set I/O page size to 2KB and scale the number of threads up to the point where the SSD is saturated. For asynchronous I/O, we use io_uring and compiled all the code using GCC 11 with all the optimizations.

Benchmarks. To focus on evaluating the effectiveness of our designs in hiding latencies, we implement benchmarks that directly interface with the engine via C++ APIs, bypassing SQL and networking layers. We run different end-to-end benchmarks where the database engine is dominated by latencies generated from (1) storage accesses, (2) CPU oversubscription and (3) synchronization, respectively. For (1), we use microbenchmarks to stress test MosaicDB. Microbenchmarks allow us to closely control the hot and cold data ratios to evaluate different scheduling policies and understand MosaicDB's performance behavior. For (2), we run TPC-C [56] to evaluate the impact of oversubscription caused by background threads and show the effectiveness of MosaicDB. For (3), we run a high-contention microbenchmark that issues single-step insert-only transactions with monotonically increasing keys to study how well MosaicDB can handle synchronization latency. We give the detailed workload setups in the following corresponding sections.

6.2 Larger-Than-Memory Performance

Our first experiment evaluates the performance of memory- and storage-bound transactions.

Workloads. To stress test MosaicDB, we separately load two tables, one as the hot store with all the data in memory, and the other as the cold store with all the data only in secondary storage (the log in our design). Accesses to both stores are done by (1) traversing the index to obtain an RID and (2) using the RID to access the indirection array. For hot access, in step (2) the transaction traverses an in-memory version chain, while for cold accesses the transaction in step (2) would obtain the permanent address of the record in storage and issue an I/O request to access it. Note that we do not cache cold records in memory, i.e., accessing a storage-resident record will always lead to an I/O. This allows us to reliably interpret experimental results and independently scale the sizes of both stores. Implementing a cache is promising but orthogonal future work. We use 8-byte keys and a 8-byte values for each data record. All experiments start with a freshly loaded database with 300 million hot records and 3 million cold records, in total taking ~16GB of storage space (including padding as required by direct I/O). The amount of cold records does not matter here, because cold records are not cached, so that we can constantly create storage-bound transactions during the experiments. We vary the percentage of storage-bound transactions following a uniform distribution to choose the records to access. We evaluate two types of workloads: read-only and read-write. For read-only, an in-memory transaction reads 10 records in the hot table, and a storage-bound transaction reads 8/2 records in the hot/cold tables. We use 40 threads which can saturate the SSD. For read-write workloads, there are 50% read-only transactions (10 reads) and 50% read-modify-write (RMW) transactions (10 updates); cold read-only transactions have 2/8 cold/hot reads; cold RMW transactions have 2/8 cold/hot updates where a cold update is a cold read followed by an update. Based on the IOPS and bandwidth that our storage device can offer, we use 10 threads to saturate the storage device with I/Os needed by reading cold records and flushing log records.

Variants. We vary the scheduling policy discussed in Section 4.

- `Sequential`: Baseline that uses traditional thread-to-transaction to process transactions sequentially, but still uses `prefetch` instructions as a best-effort optimization for memory accesses. Cold store accesses are done using synchronous io_uring calls: the thread will spin until the I/O completes.
- `Batch`: The storage-aware batch scheduling policy described in Section 4.2. We tuned the batch size and set it to 8.
- `Pipeline`: The vanilla pipelined scheduling policy in Section 4.3. Queue size is set to 16, with memory- and storage-bound transactions each using half the capacity.
- `MosaicDB`: Same as `Pipeline` but uses the dual-queue pipelined scheduling policy. Unless otherwise noted, we set the memory/storage queue sizes to 8/16 and check the storage queue once after eight transactions are concluded.

Read-Only Throughput. Figure 6 shows the throughput and latency of `MosaicDB` and baselines along with SSD IOPS, as the workload mix features more storage-bound transactions. When the workload is purely in-memory, as expected, the three coroutine-to-transaction variants all outperform `Sequential`, corroborating with results obtained by prior work [21]. Such performance improvement comes from the increased interleaving of transactions which allows us to hide memory data stalls by overlapping software prefetching and computation. As we start to increase storage-bound transactions in the mix, the throughput of `Sequential` drops drastically, which is mainly attributed to waiting for I/O completions via spinning. For `Batch`, the storage-bound transactions stall the admission of new transactions, which eventually runs into the same problem as `Sequential` has. That is, the scheduling logic eventually degrades into a `Sequential`-equivalent that waits synchronously for the storage-bound transactions. For `Pipeline` and `MosaicDB`, the hot throughput well sustains, thanks to the reserved slots for in-memory transactions, but the additional CPU cycles needed to

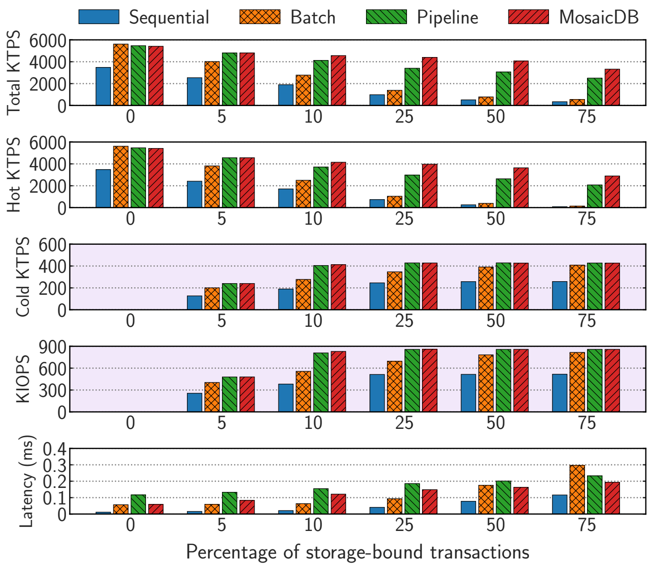


Figure 6: Read-only microbenchmark results under varying percentages of storage-bound transactions (40 threads).

issue and check I/O states still led to reduced interleaving of in-memory transactions. For coroutine variants in general, a single worker thread can handle multiple I/O requests, thus hiding storage latency with interleaving and improving SSD utilization by up to a factor of two (measured in IOPS at 10%). When the workload is still memory-bound (i.e., 5% storage-bound transactions), compared to `Sequential`, the total throughput of `Batch`/`Pipeline`/`MosaicDB` is 1.6×/1.9×/1.9× higher. Across benchmarks, `MosaicDB` commits 1.55×–33× in-memory transactions than `Sequential`; `MosaicDB` achieves up to 33× speedup over `Batch`. When the workload becomes more storage-bound, the hot throughput of `Pipeline` cannot keep up with `MosaicDB`'s, because `Pipeline` always needs to check the storage-bound transactions, which is not necessary when the storage device is saturated. In contrast, `MosaicDB` has two queues, allowing threads to work on the in-memory queue most of the time. The loss of total throughput of `MosaicDB` is sustained at 16%, when the SSD is just fully utilized (i.e., 10% storage-bound transactions).

Read-Only Latency. Interleaving-based approaches like MosaicDB may trade latency for throughput. As shown in Figure 6, `Sequential` consistently exhibits the lowest latency. `Pipeline` uses twice as many slots as `Batch` and `MosaicDB`'s hot queue, leading to ∼2× the latency for in-memory transactions than `Sequential`. The latency of `MosaicDB` always stands in between `Batch` and `Pipeline` because compared to `Batch`, `MosaicDB` is also affected by the storage queue, although the scheduler does not always check the cold queue. With more storage-bound transactions, `Batch`'s latency is bounded by the batch size, which is smaller than `MosaicDB`'s cold queue size. So we observe the growing gap between `Batch` and `MosaicDB`. Compared to `Pipeline`, `MosaicDB` spends more cycles on the hot queue, leading to slightly lower latency than `Pipeline`.

Read-Write Performance. Figure 7 shows the performance of `MosaicDB` under the RMW workload where records are fetched from storage, updated in memory and then persisted in the log. This

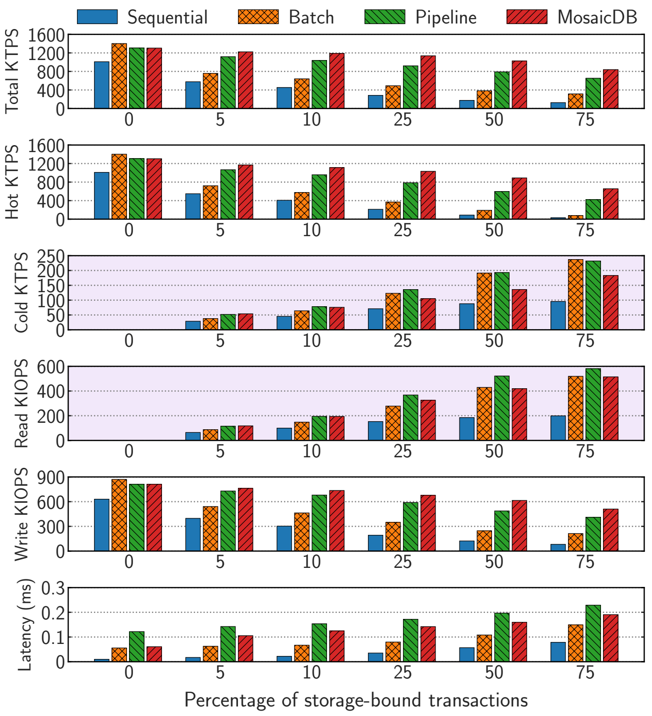


Figure 7: Read-write (50% read-only, 50% update-only) microbenchmark results under 10 threads and a varying percentage of storage-bound transactions.

workload is write-heavy with 50% RMW transactions. Compared to cold reads that use 2KB I/Os, log records are first buffered in memory and persisted in SSD in batches (8MB per thread in our setup), which makes write IOPS an order of magnitude lower than read IOPS. For the in-memory workload, `Batch`/`Pipeline`/`MosaicDB` is 1.39×/1.29×/1.29× faster than `Sequential`. This is because log flushes are asynchronous and we use double buffering to avoid blocking writers while write I/O is in-progress. Therefore, prefetching still plays a role in improving throughput. However, since RMW transactions contribute to more I/O API calls for log flushes, fewer cycles can be dedicated to software prefetching, leading to lower speedups for in-memory transactions. With more storage-bound transactions, the coroutine-oriented variants overall follow a similar trend seen in the read-only benchmarks, with `MosaicDB` maintaining high performance for both in-memory and storage-bound transactions. `MosaicDB` is up to 2.1× faster than `Sequential` in terms of the overall performance with comparable storage-bound transaction processing speed. `Batch` usually has the highest cold throughput as it treats both hot and storage-bound transactions equally, which means more storage-bound transactions get processed while in-memory transactions are starved. From 0% to 50% storage-bound transactions, the total throughput of `MosaicDB` drops by 21%, but is still on par with `Sequential`'s in-memory throughput. In terms of latency, we observed similar results to those from the read-only workloads, except that the cold queue size of `MosaicDB` is doubled in this workload to accommodate more storage-bound transactions, which become slower due to the increased storage access latency.

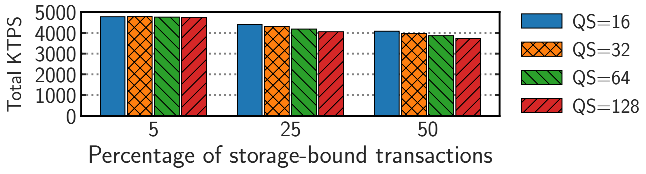


Figure 8: Tuning storage queue size for a read-only workload.

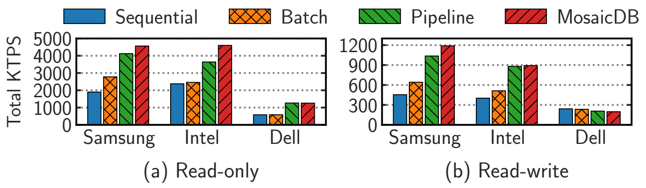


Figure 9: `MosaicDB` running read-only and read-write workloads with 10% storage-bound transactions on different SSDs.

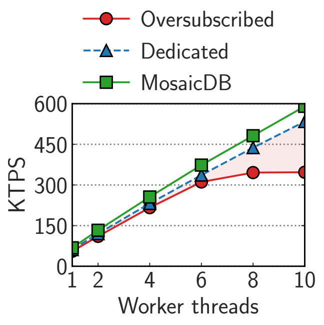


Figure 10: TPC-C throughput varying oversubscription handling policies (up to 10 threads before the SSD is saturated).

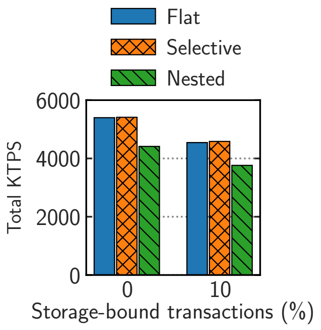


Figure 11: Throughput of read-only workloads using `MosaicDB` under different coroutine flattening/nesting approaches.

Impact of Queue Size. `MosaicDB` favors in-memory transactions over storage-bound transactions. Therefore, in Figure 8, the total throughput of the system is robust to the storage queue size (QS). In practice, we first experimentally find the desirable memory queue size under pure in-memory workloads, which usually is below 10 and is 8 in our case. The tuning for storage-bound transactions is also lightweight. The intuition is that the storage queue size should be large enough to not underutilize the storage device, but not too large to forfeit memory prefetching. In Figure 8, with a small percentage of storage-bound transactions (e.g., 5%), storage access latency can be hidden very well, so QS does not impact total throughput. With more I/O, `MosaicDB` can easily saturate storage with QS=16 (also used in the following experiments); a larger QS will only increases the latency of in-memory transactions.

Impact of Devices and Page Size. The selected devices have very distinctive characteristics. The Samsung SSD delivers the highest IOPS and bandwidth but has a relatively higher latency (usually $> 100us$) than the Intel Optane SSD ($10us$ at its best). The Dell SSD has very limited bandwidth and the highest latency, which is the least powerful among the three devices in every dimension. Figure 9 shows that `MosaicDB` yields similar performance on the Samsung and Intel devices, and outperforms other approaches. Under the hood, all the three devices are fully utilized, but the performance is still bounded by in-memory transaction throughput as `MosaicDB` prioritizes in-memory transactions. The uniqueness of read-write workloads is that the in-memory transactions also generate logs that lead to I/O writes, therefore in-memory transaction throughput is also affected by IOPS of the device. In other words, as `MosaicDB` favors in-memory transactions, which could in turn lead to more I/Os than other approaches, its performance is not always guaranteed to be the best but is still comparable to others. We also evaluated the impact of page size on the performance of MosaicDB. For brevity, the details are left out. Briefly, regardless of page size, the performance with the same settings between runs barely changes, as for read-only workloads, the overall performance is always bounded by in-memory transactions whether or not the device is saturated. For read-write workloads, the in-memory transaction throughput is also affected by logs flushes, but the write I/Os only happen when the log buffer is full. Since the log buffer size is fixed, throughput insensitive to individual page sizes, unlike cold reads.

6.3 Effect of Oversubscription Avoidance

We evaluate the effect of oversubscription avoidance in `MosaicDB` using the TPC-C benchmark [56]. We take two variants of CoroBase as baselines as they are also based on the coroutine-to-transaction paradigm but use background threads.

- `Oversubscribed`: All the threads (worker and background) are limited to run on a given CPU budget of N cores. We set the number of worker threads to be the number of CPU cores so that the background threads will be scheduled by the OS to compete with worker threads.
- `Dedicated`: Uses an additional CPU core for the background log flushing thread. Each thread is pinned to a different core, avoiding OS scheduler activities.

All the three variants here run the batch policy with a batch size of four for fair comparison. We vary the number of threads to understand how performance is affected by oversubscription as background thread work increases. We run experiments before the SSD is saturated to eliminate the impact of storage performance. Figure 10 shows the result. All the variants scaled up to six threads, and `Oversubscribed` is on par with `Dedicated` and `MosaicDB`. However, with more threads, the overhead of oversubscription starts to appear (shaded area in the figure), which increases to 35% at 10 threads. For `Oversubscribed`, using more threads leads to more OS scheduling activities, as the background thread needs to flush logs and group commit transactions more frequently, so the effective CPU cycles that can be used by threads to process transactions become fewer. Compared to `Oversubscribed` and `Dedicated`, `MosaicDB` does not oversubscribe CPU cores or require additional resources, and outperforms `Oversubscribed` by 1.7×.

6.4 Effect of Reduced Contention

Our next experiment evaluates `MosaicDB`'s effect on contention regulation. We use an insert-only workload that issues monotonically increasing new keys to continuously append new records to a

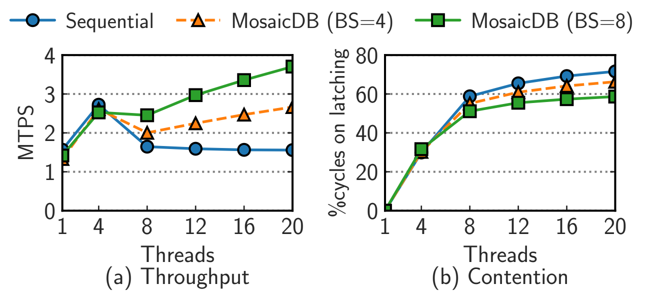


Figure 12: Throughput of a high-contention insert-only workload.

single table. All the data is stored in `tmpfs` to avoid I/O becoming a bottleneck and to stress the system. Each transaction inserts a new record. We start each experiment with a table containing 300 million records. All worker threads are pinned to use one NUMA node. We ran the benchmarks with the three coroutine-based variants in Section 6.2 and obtained the same results because insert-only workloads always happen in-memory first before their log records are flushed. For brevity, we show the performance of `Batch` and compare it with the baseline (`Sequential`). We vary the batch size between 4 and 8 to explore how interleaving plays a role in reducing synchronization overhead.

Figure 12 shows the throughput (a) and contention level as measured by the percentage of cycles spent on acquiring the latch (b). `Sequential` is on par with `MosaicDB` until four threads. Starting at eight threads, `Sequential`'s performance first drops drastically by 40%, then remains at single-threaded performance. Beyond four threads, `MosaicDB`'s performance first drops at eight threads before it starts to scale. Compared to `Sequential`, two factors affect the performance of `MosaicDB`: (1) the number of threads and (2) degree of interleaving (batch size). Similar to `Sequential`, using more threads can lead to higher contention, hence reduced performance. However, `MosaicDB` interleaves transactions on each thread, which helps reduce the chance of all the threads contending on the latch at the same time. The reason for the drop at eight threads is due to the increased number of threads (contention) playing a bigger role, whereas beyond eight threads, the effect of reduced contention because of interleaving starts to show up, leading to better performance than `Sequential`. Overall, at batch size of 4/8, `MosaicDB`'s contention is up to 8%/18% lower than `Sequential`'s, which gives `MosaicDB` up to 1.71×/2.38× higher throughput and generally better scalability under high contention.

6.5 Effect of Selective Coroutine Nesting

We run the same read-only workload as the one in Section 6.2 but use three different coroutine approaches: (1) flattened which has the least coroutine suspend/resume overhead, (2) selectively nested and (3) fully nested (the most overhead). We show two representative cases based on whether there is storage access. As Figure 11 shows, when the workload is purely in-memory, using flattened coroutines gives the same performance as selectively nested coroutines. This is expected because the two structures are the same on the in-memory codepath. However, using fully nested coroutines leads to 18% slowdown because the memory path is heavily affected by the depth of the coroutine chain; the same effect was also documented elsewhere [21]. With storage access, selectively nested coroutines have deeper coroutine chains than flattened on the storage path, but the performance is still comparable. The reason is storage latency is much higher than memory latency, and storage coroutine call chains are resumed only after I/O is finished, amortizing the cost.

7 RELATED WORK

Memory-Optimized OLTP Engines. Indirection has been studied extensively in previous work [8, 13, 21, 30, 40, 50, 62]. Bw-Tree [40] and Hekaton [13] rely on indirection to provide latch-free index operations. ERMIA and CoroBase, which MosaicDB is based upon, use indirection arrays to reduce index maintenance costs.

Cold Data Management. Some systems [16, 29, 35] identify and compress cold data to reduce its main-memory footprint. The cold data is compressed into a format that is optimized for OLAP queries. These techniques can be adopted by MosaicDB to extend its capabilities for large OLAP queries. Siberia [14] is integrated with Hekaton [13] to manage a hot store and a cold store. In H-Store [55], anti-caching [10] supports larger-than-memory transactions by ensuring first all the data needed by a transaction is in-memory.

Optimizations for Fast Storage Devices. Recent NVMe SSDs come with very high bandwidth and low latency, mandating careful redesigns of storage engines [22]. DANA [19] employs multiple SSDs to achieve memory-like bandwidth. LeanStore [37], a storage engine which was first proposed for optimizing I/Os associated with SSDs, has been upgraded in its recent version [20] for exploiting DANA-based storage backend using stackful coroutines. Merzljak et al. [59] suggest a way to take advantage of NVMe SSDs by leveraging coroutines and asynchronous I/O for OLAP.

8 SUMMARY

We have identified latencies beyond memory access latency from multiple sources, i.e., storage I/O, oversubscription/OS scheduling and synchronization, in memory-optimized OLTP engines. Yet prior work mostly focused on mitigating a single source of latency. With the coroutine-to-transaction execution model providing a new perspective for building memory-optimized database engines and advances in modern hardware, we see unexplored opportunities in jointly hiding these identified latencies. We present MosaicDB, a multi-versioned, latency-optimized OLTP engine that hides different latencies at the same time. Techniques in MosaicDB can be applied independently in other systems. Overall, MosaicDB achieves up to 33× higher throughput under larger-than-memory workloads. With a given CPU budget, MosaicDB avoids oversubscription and improves TPC-C throughput by 1.7×. MosaicDB also scales well under skewed workloads, with up to 18% less contention and 2.38× higher throughput than state-of-the-art.

ACKNOWLEDGMENTS

We thank Baotong Lu, Yongjun He and Ge Shi for their invaluable comments. We also thank the associate editor and the anonymous reviewers for their constructive feedback. This work is partially supported by an NSERC Discovery Grant, Canada Foundation for Innovation John R. Evans Leaders Fund and the B.C. Knowledge Development Fund.

REFERENCES

[1] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. 2022. The Full Story of 1000 Cores: An Examination of Concurrency Control on Real(Ly) Large Multi-Socket Hardware. *The VLDB Journal* 31, 6 (apr 2022), 1185–1213.

[2] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. 387–400.

[3] Hokeun Cha, Xiangpeng Hao, Tianzheng Wang, Huanchen Zhang, Aditya Akella, and Xiangyao Yu. 2023. Blink-Hash: An Adaptive Hybrid Index for In-Memory Time-Series Databases. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1235–1248.

[4] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. 2011. An Overview of Business Intelligence Technology. *Commun. ACM* 54, 8 (aug 2011), 88–98.

[5] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2004. Improving Hash Join Performance through Prefetching. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*. 116.

[6] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving Index Performance through Prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*. 235–246.

[7] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. 2002. Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. 157–168.

[8] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 133–146.

[9] Jonathan Corbet. 2019. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/ Linux Weekly Newsletter.

[10] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1942–1953.

[11] K. Delaney and C. Freeman. 2013. *Microsoft SQL Server 2012 Internals*. Pearson Education.

[12] Dell. 2023. Dell 480GB SSD SATA Mixed Use 6Gbps 512e 2.5in Hot-Plug. (2023). https://www.dell.com/en-ca/shop/dell-480gb-ssd-sata-mixed-use-6gbps-512e-25in-hot-plug/apd/345-befn/

[13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. 1243–1254.

[14] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking through siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment* 7, 11 (2014), 931–942.

[15] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. 2019. Interleaved Multi-Vectorizing. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 226–238.

[16] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *Proc. VLDB Endow.* 5, 11 (jul 2012), 1424–1435.

[17] Goetz Graefe. 2012. A Survey of B-Tree Logging and Recovery Techniques. *ACM Trans. Database Syst.* 37, 1, Article 1 (mar 2012), 35 pages.

[18] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. 2014. In-Memory Performance for Big Data. *PVLDB* 8, 1 (sep 2014), 37–48.

[19] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS.. In *CIDR (Conference on Innovative Data Systems Research)*.

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

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

[22] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. 2022. SSDs Striking Back: The Storage Jungle and Its Implications to Persistent Indexes. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*.

[23] Intel. 2023. Intel® Optane™ SSD DC P4800X Series. (2023). https://www.intel.com/content/www/us/en/products/sku/97161/intel-optane-ssd-dc-p4800x-series-375gb-2-5in-pcie-x4-3d-xpoint/specifications.html

[24] Intel Corporation. 2016. Intel 64 and IA-32 Architectures Software Developer Manuals. (Oct. 2016).

[25] ISO/IEC. 2017. Technical Specification — C++ Extensions for Coroutines. https://www.iso.org/standard/73008.html.

[26] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *PVLDB* 3, 1 (Sept. 2010), 681–692.

[27] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the killer nanoseconds. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714.

[28] Alfons Kemper and Donald Kossmann. 1995. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *The VLDB Journal* 4, 3 (jul 1995), 519–567.

[29] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. 195–206.

[30] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687.

[31] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 691–706.

[32] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment* 9, 4 (2015), 252–263.

[33] Sridhar K.T. and M.A. Sakkeer. 2014. Optimizing Database Load and Extract for Big Data Era. 8422 (04 2014), 503–512.

[34] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226.

[35] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*. 311–326.

[36] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (Dec. 2011), 298–309.

[37] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE) (IEEE ICDE)*. 185–196.

[38] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. Article 3, 8 pages.

[39] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB* 6, 10 (aug 2013), 877–888. https://doi.org/10.14778/2536206.2536215

[40] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-Tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. 302–313.

[41] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 21–35.

[42] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

[43] Microsoft. 2018. *Windows Technical Documentation*. https://docs.microsoft.com/en-us/windows/win32/procthread/fibers?redirectedfrom=MSDN

[44] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*. 62–73.

[45] MySQL 8.0 Reference Manual. 2023. Using Asynchronous I/O on Linux. https://dev.mysql.com/doc/refman/8.0/en/innodb-linux-native-aio.html

[46] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*.

[47] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 677–689.

[48] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *The VLDB Journal* 28, 4 (2019), 451–471.

[49] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems* (3 ed.).

[50] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. 2013. Making Updates Disk-I/O Friendly Using SSDs. *PVLDB* 6, 11 (2013), 997–1008.

[51] Samsung. 2023. 980 PRO PCIe 4.0 NVMe SSD 500GB. (2023). https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-500gb-mz-v8p500b-am/

[52] Boris Schling. 2011. *The Boost C++ Libraries*.

[53] Michael Lee Scott. 2013. *Shared-memory synchronization*.

[54] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-architectural Analysis of In-memory OLTP. In *Proceedings of the 2016 International Conference on Management of Data*. 387–402.

[55] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era: (It's Time for a Complete Rewrite). (2007), 1150–1160.

[56] TPC. 2010. TPC Benchmark C (OLTP) Standard Specification, revision 5.11. http://www.tpc.org/tpcc

[57] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 18–32.

[58] Demian Vöhringer and Viktor Leis. 2023. Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3323–3334. https://doi.org/10.14778/3611479.3611529

[59] Leonard von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. 2022. What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022*.

[60] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *PVLDB* 7, 10 (June 2014), 865–876.

[61] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. 2017. Efficiently Making (Almost) Any Concurrency Control Mechanism Serializable. *The VLDB Journal* 26, 4 (Aug. 2017), 537–562.

[62] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7 (March 2017), 781–792.

[63] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: Lightweight Parallel Logging for in-Memory Database Management Systems. *PVLDB* 14, 2 (Oct. 2020), 189–201.

[64] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2022. Skeena: Efficient and Consistent Cross-Engine Transactions. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22)*.