CoroBase: Coroutine-Oriented Main-Memory Database Engine

Yongjun He
Simon Fraser University
yongjunh@sfu.ca

Jiacheng Lu
Simon Fraser University
jiacheng_lu@sfu.ca

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

ABSTRACT

Data stalls are a major overhead in main-memory database engines due to the use of pointer-rich data structures. Lightweight coroutines ease the implementation of software prefetching to hide data stalls by overlapping computation and asynchronous data prefetching. Prior solutions, however, mainly focused on (1) individual components and operations and (2) intra-transaction batching that requires interface changes, breaking backward compatibility. It was not clear how they apply to a full database engine and how much end-to-end benefit they bring under various workloads.

This paper presents CoroBase, a main-memory database engine that tackles these challenges with a new *coroutine-to-transaction* paradigm. Coroutine-to-transaction models transactions as coroutines and thus enables inter-transaction batching, avoiding application changes but retaining the benefits of prefetching. We show that on a 48-core server, CoroBase can perform close to 2× better for read-intensive workloads and remain competitive for workloads that inherently do not benefit from software prefetching.

PVLDB Reference Format:
Yongjun He, Jiacheng Lu, and Tianzheng Wang. CoroBase: Coroutine-Oriented Main-Memory Database Engine. PVLDB, 14(3): 431-444, 2021.
doi:10.14778/3430915.3430932

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfu-dis/corobase/tree/v1.0.

1 INTRODUCTION

Modern main-memory database engines [11, 22, 24, 25, 30, 33, 56, 59] use memory-optimized data structures [2, 29, 31, 36] to offer high performance on multicore CPUs. Many such data structures rely on pointer chasing [34] which can stall the CPU upon cache misses. For example, in Figure 1(a), to execute two SELECT (get) queries, the engine may traverse a tree, and if a needed tree node is not cache-resident, dereferencing a pointer to it stalls the CPU (dotted box in the figure) to fetch the node from memory. Computation (solid box) would not resume until data is in the cache. With the wide speed gap between CPU and memory, memory accesses have become a major overhead [4, 35]. The emergence of capacious but slower persistent memory [10] is further widening this gap.

Modern processors allow multiple outstanding cache misses and provide `prefetch` instructions [18] for software to explicitly bring data from memory to CPU caches. This gave rise to software prefetching techniques [5, 21, 26, 34, 39, 44, 45] that *hide* memory access latency by overlapping data fetching and computation, alleviating pointer chasing overhead. Most of these techniques, however, require hand-crafting asynchronous/pipelined algorithms or state machines to be able to suspend/resume execution as needed. This is a difficult and error-prone process; the resulted code often deviates a lot from the original code, making it hard to maintain [21].

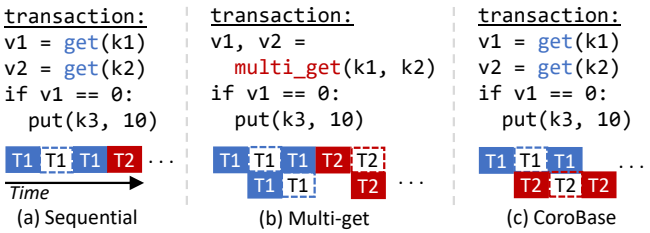

Figure 1: Data access interfaces and execution under (a) sequential execution (no interleaving), (b) prior approaches that require multi-key interfaces, (c) CoroBase which hides data stalls and maintains backward compatibility.

1.1 Software Prefetching via Coroutines

With the recent standardization in C++20 [19], coroutines greatly ease the implementation of software prefetching. Coroutines [38] are functions that can suspend voluntarily and be resumed later. Functions that involve pointer chasing can be written as coroutines which are executed (interleaved) in batches. Before dereferencing a pointer in coroutine $t1$, the thread issues a `prefetch` followed by a `suspend` to pause $t1$ and switches to another coroutine $t2$, overlapping data fetching in $t1$ and computation in $t2$.

Compared to earlier approaches [5, 26], coroutines only require `prefetch`/`suspend` be inserted into sequential code, greatly simplifying implementation while delivering high performance, as the switching overhead can be cheaper than a last-level cache miss [21]. However, adopting software prefetching remains challenging.

First, existing approaches typically use intra-transaction batching which mandates multi-key interfaces that can break backward compatibility. For example, in Figure 1(b) an application[1] uses `multi_get` to retrieve a batch of records at once in a transaction. Cache misses caused by probing `k1` (`k2`) in a tree are hidden behind the computation part of probing `k2` (`k1`). While intra-transaction batching is a natural fit for some operators (e.g., IN-predicate queries [44, 45]), it is not always directly applicable. Changing the application is not always feasible and may not achieve the desired improvement as depending requests need to be issued in

[1]The "application" may be another database system component or an end-user application that uses the record access interfaces provided by the database engine.

Proceedings of the VLDB Endowment, Vol. 14, No. 3 ISSN 2150-8097.
doi:10.14778/3430915.3430932

separate batches, limiting interleaving opportunities. Short (or even single-record) transactions also cannot benefit much due to the lack of interleaving opportunity. It would be desirable to allow batching operations across transactions, i.e., inter-transaction batching.

Second, prior work provided only piece-wise solutions, focusing on optimizing individual database operations (e.g., index traversal [21] and hash join [5, 44]). Despite the significant improvement (e.g., up to 3× faster for tree probing [21]), it was not clear how much overall improvement one can expect when these techniques are applied in a full database engine that involves various components.

Overall, these issues lead to two key questions:

- How should a database engine adopt coroutine-based software prefetching, preferably without requiring application changes?
- How much end-to-end benefit can software prefetching bring to a database engine under realistic workloads?

1.2 CoroBase

To answer these questions, we propose and evaluate CoroBase, a multi-version, main-memory database engine that uses coroutines to hide data stalls. The crux of CoroBase is a simple but effective *coroutine-to-transaction* paradigm that models transactions as coroutines, to enable *inter*-transaction batching and maintain backward compatibility. Worker threads receive transaction requests and switch among transactions (rather than requests within a transaction) without requiring intra-transaction batching or multi-key interfaces. As Figure 1(c) shows, the application remains unchanged as batching and interleaving happen at the transaction level.

Coroutine-to-transaction can be easily adopted to hide data stalls in different database engine components and can even work together with multi-key based approaches. In particular, in multi-version systems versions of data records are typically chained using linked lists [62], traversing which constitutes another main source of data stalls, in addition to index traversals. CoroBase transparently suspends and resumes transactions upon pointer dereferences during version chain traversals. This way, CoroBase "coroutinizes" the full data access paths to provide an end-to-end solution.

To explore how coroutine-to-transaction impacts common design principles of main-memory database systems, instead of building CoroBase from scratch, we base it on ERMIA [24], an open-source, multi-version main-memory database engine. This allows us to devise an end-to-end solution and explore how easy (or hard) it is to adopt coroutine-to-transaction in an existing engine, which we expect to be a common starting point for most practitioners. In this context, we discuss solutions to issues brought by coroutine-to-transaction, such as (nested) coroutine switching overhead, higher latency and more complex resource management in later sections.

On a 48-core server, our evaluation results corroborate with prior work and show that software prefetching is mainly (unsurprisingly) beneficial to read-dominant workloads, with close to 2× improvement over highly-optimized baselines. For write-intensive workloads, we find mixed results with up to 45% improvement depending on access patterns. Importantly, CoroBase retains competitive performance for workloads that inherently do not benefit from prefetching, thanks to its low-overhead coroutine design.

Note that our goal is *not* to outperform prior work, but to (1) effectively adopt software prefetching in a database engine without necessitating new interfaces, and (2) understand its end-to-end benefits. Hand-crafted techniques usually present the performance upper bound; CoroBase strikes a balance between performance, programmability and backward compatibility.

1.3 Contributions and Paper Organization

We make four contributions. ❶ We highlight the challenges for adopting software prefetching in main-memory database engines. ❷ We propose a new execution model, coroutine-to-transaction, to enable inter-transaction batching and avoid interface changes while retaining the benefits of prefetching. ❸ We build CoroBase, a main-memory multi-version database engine that uses coroutine-to-transaction to hide data stalls during index and version chain traversals. We explore the design tradeoffs by describing our experience of transforming an existing engine to use coroutine-to-transaction. ❹ We conduct a comprehensive evaluation of CoroBase to quantify the end-to-end effect of prefetching under various workloads. CoroBase is open-source at https://github.com/sfu-dis/corobase.

Next, we give the necessary background in Section 2. Sections 3–4 then present the design principles and details of CoroBase. Section 5 quantifies the end-to-end benefits of software prefetching. We cover related work in Section 6 and conclude in Section 7.

2 BACKGROUND

This section gives the necessary background on software prefetching techniques and coroutines to motivate our work.

2.1 Software Prefetching

Although modern CPUs use sophisticated hardware prefetching mechanisms, they are not effective on reducing pointer-chasing overheads, due to the irregular access patterns in pointer-intensive data structures. For instance, when traversing a tree, it is difficult for hardware to predict and prefetch correctly the node which is going to be accessed next, until the node is needed right away.

Basic Idea. Software prefetching techniques [5, 21, 26, 44] use workload semantics to issue `prefetch` instructions [18] to explicitly bring data into CPU caches. Worker threads handle requests (e.g., tree search) in batches. To access data (e.g., a tree node) in request $t1$ which may incur a cache miss, the thread issues a `prefetch` and switches to another request $t2$, and repeats this process. While the data needed by $t1$ is being fetched from memory to CPU cache, the worker thread handles $t2$, which may further cause the thread to issue `prefetch` and switch to another request. By the time the worker thread switches back to $t1$, the hope is that the needed data is (already and still) cache-resident. The thread then picks up at where it left for $t1$, dereferences the pointer to the prefetched data and continues executing $t1$ until the next possible cache miss upon which a `prefetch` will be issued. It is important that the switching mechanism and representation of requests are cheap and lightweight enough to achieve a net gain.

Hand-Crafted Approaches. The mechanism we just described fits naturally with many loop-based operations. Group prefetching and software pipelined prefetching [5] overlap multiple hash table lookups to accelerate hash joins. After a `prefetch` is issued, the control flow switches to execute the computation stage of another operation. Asynchronous memory access chaining (AMAC) [26] is

a general approach that allows one to transform a heterogeneous set of operations into state machines to facilitate switching between operations upon cache misses. A notable drawback of these approaches is they require developers hand-craft algorithms. The resulted code is typically not intuitive to understand and hard to maintain. This limits their application to simple or individual data structures and operations (e.g., tree traversal). Recent approaches tackle this challenge using lightweight coroutines, described next.

2.2 Coroutines

Coroutines [8] are generalizations of functions with two special characteristics: (1) During the execution between invoke and return, a coroutine can suspend and be resumed at manually defined points. (2) Each coroutine preserves local variables until it is destroyed. Traditional stackful coroutines [38] use separate runtime stacks to keep track of local variables and function calls. They have been available as third-party libraries [37, 51] and are convenient to use, but exhibit high overhead that is greater than the cost of a cache miss [21], defeating the purpose of hiding memory stalls.

Stackless Coroutines. Recent stackless coroutines standardized in C++20 [19] (which is our focus) exhibit low overhead in construction and context switching[2] (cheaper than a last-level cache miss). They do not own stacks and run on the call stack of the underlying thread. Invoking a coroutine is similar to invoking a normal function, but its states (e.g., local variables that live across suspension points) are kept in dynamically allocated memory (coroutine frames) that survive `suspend`/`resume` cycles. Figure 2 shows an example in C++20: any function that uses coroutine keywords (e.g., `co_await`, `co_return`) is a coroutine. A coroutine returns a `promise_type` structure that allows querying the coroutine's states, such as whether it is completed and its return value. The `co_await` keyword operates on a `promise_type` and is translated by the compiler into a code block that can save the states in a coroutine frame and pop the call stack frame. The `suspend_always` object is an instance of `promise_type` that has no logic and suspends unconditionally. The `co_return` keyword matches the syntax of `return`, but instead of returning an `rvalue`, it stores the returned value into a coroutine frame. As Figure 2 shows, upon starting (step ❶) or resuming (step ❸) a coroutine, a frame is created and pushed onto the stack. At unconditional suspension points (steps ❷ and ❹), the frame is popped and control is returned to the caller. Since the coroutine frame lives on the heap, coroutine states are still retained after the stack frame is popped. Coroutine frames need to be explicitly destroyed after the coroutine finishes execution.

Scheduling. Each worker thread essentially runs a scheduler that keeps switching between coroutines, such as the one below:

```
1. // construct coroutines
2. for i = 0 to batch_size - 1:
3.   coroutine_promises[i] = foo(...);
4. // switch between active coroutines
5. while any(coroutine_promises, x: not x.done()):
6.   for i = 0 to batch_size - 1:
7.     if not coroutine_promises[i].done():
8.       coroutine_promises[i].resume()
```

[2]Not to be confused with context switches at the OS level. In main-memory systems, threads are typically pinned to mitigate the impact of OS scheduling. Throughout this paper "contexts" refers to coroutines/transactions that are pure user-space constructs.

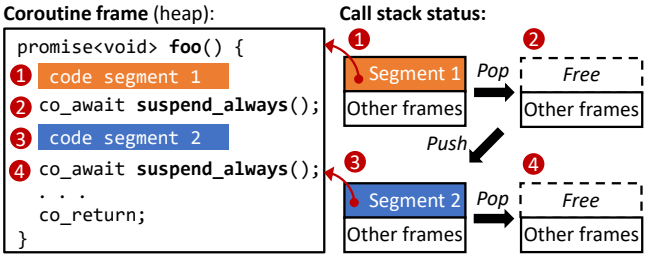

Figure 2: Stackless coroutine that directly uses the underlying thread's stack. States (e.g., local variables and return value) are maintained in dynamically allocated memory.

After creating a batch of operations (coroutines) at lines 1–2, it invokes and switches among coroutines (lines 4–8). The `batch_size` parameter determines the number of inflight memory fetches and how effectively memory stalls can be hidden: once a coroutine suspends, it is not resumed before the other `batch_size-1` coroutines are examined. Prior work has shown that the optimal `batch_size` is roughly the number of outstanding memory accesses that can be supported by the CPU (10 in current Intel x86 processors) [44].

Nested Stackless Coroutines. Similar to "normal" functions, a coroutine may invoke, suspend and resume another coroutine using `co_await`, forming a chain of nested coroutines. By default, when a stackless coroutine suspends, control is returned to its caller. Real-world systems often employ deep function calls for high-level operations to modularize their implementation. To support interleaving at the operation level in database engines (e.g., search), a mechanism that allows control to be returned to the top-level (i.e., the scheduler) is necessary. This is typically done by returning control level-by-level, from the lowest-level suspending coroutine to the scheduler, through every stack frame. Note that the number of frames preceding the suspending coroutine on the stack may not be the same as its position in the call chain. For the first `suspend` in the coroutine chain, the entire chain is on the stack. For subsequent `suspends`, however, the stack frames start from the last suspended coroutine instead of the top level one, since it is directly resumed by the scheduler. When a coroutine c finishes execution, the scheduler resumes execution of c's parent coroutine. As a result, a sequential program with nested function calls can be easily transformed into nested coroutine calls by adding `prefetch` and `suspend` statements.

Astute reader may have noticed that this approach can be easily used to realize coroutine-to-transaction. However, doing so can bring non-trivial overhead associated with scheduling and maintaining coroutine states; we discuss details in later sections.

2.3 Coroutine-based Software Prefetching in Main-Memory Database Engines

Modern main-memory database systems eliminate I/O operations from the critical path. This allows worker threads to execute each transaction without any interruptions.

Execution Model. Recent studies have shown that data stalls are a major overhead in both OLTP and OLAP workloads [53, 54]. In this paper, we mainly focus on OLTP workloads. With I/O off the critical path, thread-to-transaction has been the dominating

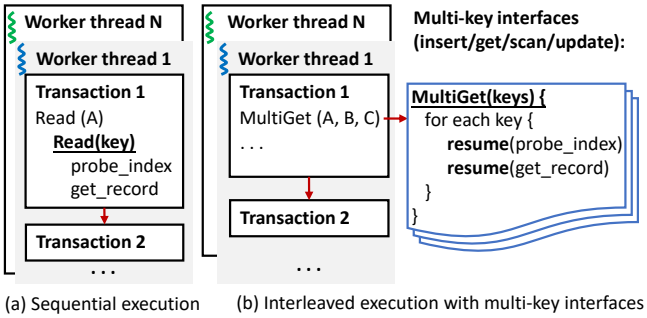

Figure 3: Execution models under thread-to-transaction.

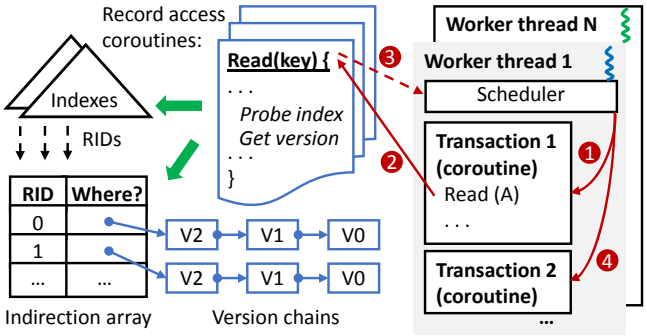

Figure 4: CoroBase overview. Indexes map keys to unique record IDs (RIDs). Versions are maintained by version chains. Each worker thread runs a scheduler that ❶ starts/resumes transactions (coroutines). ❷ A transaction may invoke other coroutines that implement specific operations, which may suspend but ❸ return control directly to the scheduler, which can ❹ resume a different transaction.

execution model in main-memory environments for transaction execution. Each worker thread executes transactions one by one without context switching to handle additional transactions unless the current transaction concluded (i.e., committed or aborted). Figure 3(a) shows an example of worker threads executing transactions under this model. To read a record, the worker thread sequentially executes the corresponding functions that implement the needed functionality, including (1) probing an index to learn about the physical address of the target record, and (2) fetching the record from the address. After all the operations of the current transaction (`Transaction 1` in the figure) are finished, the worker thread continues to serve the next transaction.

Software Prefetching under Thread-to-Transaction. With nested coroutines, it is straightforward to transform individual operations to use software prefetching, by adding suspend points into existing sequential code. However, under thread-to-transaction, once a thread starts to work on a transaction, it cannot switch to another. As a result, the caller of these operations now essentially runs a scheduler that switches between individual operations, i.e., using intra-transaction batching. In the case of a transaction reading records, for example, in Figure 3(b), the transaction calls a `multi_get` function that accepts a set of keys as its parameter and runs a scheduler that switches between coroutines that do the heavylifting of record access and may suspend upon cache misses. All these actions happen in the context of a transaction; another transaction can only be started after the current transaction being handled concludes, limiting inter-transaction batching and necessitating interface changes that may break backward compatibility.

3 DESIGN PRINCIPLES

We summarize four desirable properties and principles that should be followed when designing coroutine-based database engines:

- **Maintain Backward Compatibility.** The engine should allow applications to continue to use single-key interfaces. Interleaving should be enabled within the engine without user intervention.
- **Low Context Switching Overhead.** It should be at least lower than the cost of a last-level cache miss to warrant any end-to-end performance improvement in most cases. For workloads that do not have enough data stalls to benefit from prefetching, having low switching overhead can help retain competitive performance.
- **Maximize Batching Opportunities.** The batching mechanism should allow both intra- and inter-transaction interleaving. This would allow arbitrary query to benefit from prefetching, in addition to operators that naturally fits the batching paradigm.
- **Easy Implementation.** A salient feature of coroutine is it only needs simple changes to sequential code base; a new design must retain this property for maintainability and programmability.

4 COROBASE DESIGN

Now we describe the design of CoroBase, a multi-version main-memory database engine based on the coroutine-to-transaction execution model. We do so by taking an existing memory-optimized database engine (ERMIA [24]) and transforming it to use coroutine-to-transaction. As we mentioned in Section 1, this allows us to contrast and highlight the feasibility and potential of coroutine-to-transaction, and reason about the programming effort required to adopt coroutine-to-transaction. However, CoroBase and coroutine-to-transaction can be applied to other systems.

4.1 Overview

CoroBase organizes data and controls data version visibility in ways similar to other main-memory multi-version systems [11, 24, 30, 33, 62] (in our specific case, ERMIA [24]). Figure 4 gives the overall design of CoroBase. For each record, CoroBase maintains multiple versions that are chained together in a linked list, with the latest version (ordered by logical timestamps) as the list head. This is a common design in multi-version systems [62]. Each record is uniquely identified by a logical record ID (RID) that never changes throughout the lifetime of the record, in contrast to physical RIDs in traditional disk-based systems [47]. For each table, we maintain an indirection array [24, 50] that maps RID to the virtual memory pointer to the record's latest version which further points to the next older version, and so on. Indexes map keys to RIDs, instead of pointers to record versions. A main benefit of this approach is that record updates (i.e., creation of new versions) or movement (e.g., from memory to storage) will not always mandate secondary index updates. Same as ERMIA, CoroBase uses Masstree [36] for indexing

and all data accesses are done through indexes, however, the choice of index types is orthogonal to the techniques being proposed here.

With the indirection and version chain design, accessing a record is a two-step process: the worker thread first traverses an index of the underlying table, and then consults the indirection array (using the RID found in the leaf node as index) to find the actual record and suitable version by traversing the version chain of that record. We defer details on determining version visibility to later sections when we discuss concurrency control. Under thread-to-transaction, this two-step process is done synchronously by the worker thread, with record access procedures implemented as multiple levels of functions. Under coroutine-to-transaction, record access procedures are implemented as coroutines, instead of "normal" functions. As Figure 4 shows, each worker thread independently runs a scheduler that switches between a batch of transactions (coroutines). The key to realize this model is transforming nested functions on the data access path into coroutines, which we elaborate next.

4.2 Fully-Nested Coroutine-to-Transaction

To support common data access operations (insert/read/update/scan/delete), a straightforward way is to transform function call chains that may cause cache misses into nested stackless coroutines outlined in Section 2.2. Functions that will not incur cache misses may be kept as "normal" functions. For index, we follow prior work [21] to add `suspend` statements (`suspend_always` in C++20) as needed after each `prefetch`, which can be identified easily as Masstree already prefetches nodes.[3] We use `co_await` to invoke other coroutines and replace `return` with `co_return`. For version chain traversal, we issue `prefetch` and `suspend` before dereferencing a linked list node pointer. These changes are straightforward and only require inserting `prefetch`/`suspend` statements and replacing keywords. Our implementation defines macros to automatically convert between function and coroutine versions of the code, easing code maintainability.[4] Thus, a call chain of N functions is replaced by an (up to) N-level coroutine chain. Coroutines at any level may voluntarily suspend, after which control goes back to the scheduler which resumes the next transaction that is not done yet; control then goes to the newly resumed transaction.

Figure 4 shows how control flows end-to-end. When a new transaction is created, CoroBase starts to execute it in a coroutine (step ❶ in the figure). Subsequent operations (e.g., read/write/scan) are further handled in the context of their corresponding transaction coroutine (step ❷). All the operations are also coroutines that may suspend and get resumed. For example, in Figure 4, the `Read` coroutine may further use other coroutines to traverse a tree index structure to find the requested record's RID, followed by invoking yet another coroutine that traverses the version chain to retrieve the desirable version. Upon a possible cache miss, the executing coroutine (e.g., index traversal as part of a `Read` call) issues a `prefetch` to bring the needed memory to CPU caches asynchronously, followed by a `suspend` which returns control directly to the scheduler (step ❸). This allows the scheduler to further resume another transaction (step ❹), hoping to overlap computation and data fetching. After

[3]Based on https://github.com/kohler/masstree-beta.

[4]For example, the record read function/coroutine can be found at: https://github.com/sfu-dis/corobase/blob/v1.0/ermia.cc#L145. The `AWAIT` and `PROMISE` macros transparently convert between coroutine and function versions.

Algorithm 1 Scheduler for coroutine-to-transaction.

```
1  def scheduler(batch_size):
     while not shutdown:
3      [T] = get_transaction_requests()
       enter_epoch()
5      while done < batch_size:
         done = 0
7        for i = 0 to batch_size - 1:
           if T[i].is_done:
9            ++done
           else
11           T[i].resume()
       exit_epoch()
```

a transaction commits or aborts, its coroutine structures are destroyed and control is returned to the scheduler which may resume another active transaction. Finally, after every transaction in the batch is concluded, the scheduler starts a new batch of transactions.

Coroutine-to-transaction moves the responsibility of batching from the user API level to the engine level, by grouping transactions. Each worker thread runs a coroutine scheduler which accepts and handles transaction requests. In CoroBase we use a round-robin scheduler shown in Algorithm 1. The scheduler function keeps batching and switching between incoming transactions (lines 7–11). It loops over each batch to execute transactions. When a query in a transaction suspends, control returns to the scheduler which then resumes the next in-progress transaction (line 11). Note that each time the scheduler takes a fixed number (denoted as `batch_size`) of transactions, and when a transaction finishes, we do not start a new one until the whole batch is processed. The rationale behind is to preserve locality and avoid overheads associated with initializing transaction contexts. Although it may reduce the possible window of overlapping, we observe the impact is negligible. Avoiding irregular, ad hoc transaction context initialization helps maintain competitive performance for workloads that inherently do not benefit from prefetching where the scheduler activities and switching are pure overheads that should be minimized. Processing transactions in strict batches also eases the adoption of epoch-based resource management in coroutine environments, as Section 4.4 describes. The downside is that individual transaction latency may become higher. Our focus is OLTP where transactions are often similar and short, so we anticipate the impact to be modest. For workloads that may mix short transactions and long queries in a batch, other approaches, e.g., a scheduler that takes transaction priority into account when choosing the next transaction to resume may be more attractive for reducing system response time.

While easy to implement, fully-nested coroutine-to-transaction and coroutine-to-transaction in general bring three main challenges, which we elaborate next.

4.3 Two-Level Coroutine-to-Transaction

Since currently there is no way for software to tell whether dereferencing a pointer would cause a cache miss [21, 44], software has to "guess" which memory accesses may cause a cache miss. CoroBase issues `prefetch` and `suspend` upon dereferencing pointers to index

nodes and record objects in version chains based on profiling results. To reduce the cost of wrong guesses, it is crucial to reduce switching overheads. Database engine code typically uses nested, deep call chains of multiple functions to modularize implementation. As Section 5 shows, blindly transforming deep function call chains into coroutine chains using fully-nested coroutine-to-transaction incurs non-trivial overhead that overshadows the benefits brought by prefetching. Yet completely flattening (inlining) the entire call chain to become a single coroutine mixes application logic and database engine code, defeating the purpose of coroutine-to-transaction.

CoroBase takes a middle ground to flatten only nested calls within the database engine, forming a two-level structure that balances performance and programmability. This allows the application to still use the conventional interfaces; under the hood, transaction coroutines may invoke other coroutines for individual operations (e.g., `get`), which are single-level coroutines with all the nested coroutines that may suspend inlined. Sequential functions that do not suspend are not inlined unless the compiler does so transparently. At a first glance, it may seem tedious or infeasible to inline the whole read/write paths in a database engine manually. However, this is largely mechanical and straightforward. For instance, it took us shorter than three hours to flatten the search operation in Masstree [36], the index structure used by CoroBase. The flattened code occupies <100 lines and still largely maintains the original logic.[5] This shows that flattening functions is in fact feasible. Moreover, there is a rich body of work in compilers about function inlining and flattening [1, 40, 43, 49, 63] that can help automate this process. For example, developers can still write small, modularized functions, but a source-to-source transformation pass can be performed to flatten the code before compilation.

The downside of flattening is that it may cause more instruction cache misses because the same code segment (previously short functions) may appear repeatedly in different coroutines. For example, the same tree traversal code is required by both update and read operations. Individual coroutines may become larger, causing more instruction cache misses. However, as Section 5 shows, the benefits outweigh this drawback. Code reordering [3] can also be used as an optimization in compilers to reduce the instruction fetch overhead. Discussion of code transformation techniques is beyond the scope of this paper; we leave it as promising future work.

4.4 Resource Management

Resource management in the physical layer is tightly coupled with transaction execution model. Under thread-to-transaction, "transaction" is almost a synonymy of thread, allowing transparent application of parallel programming techniques, in particular epoch-based memory reclamation and thread-local storage to improve performance. Most of these techniques are implicitly thread-centric, sequential algorithms that do not consider the possibility of coroutine switching. Although the OS scheduler may deschedule a thread working on a task t, the thread does not switch to another task. When the thread resumes, it picks up from where it left to continue executing t. This implicit assumption brings extra challenges for coroutine-based asynchronous programming, which we describe and tackle next. Note that these issues are not unique to database systems, and our solutions are generally applicable to other systems employing coroutines and parallel programming techniques.

Epoch-based Reclamation. Many main-memory database engines rely on lock-free data structures, e.g., lock-free lists [15] and trees [31]. Threads may access memory that is simultaneously being removed from the data structure. Although no new accesses are possible once the memory block is unlinked, existing accesses must be allowed to finish before the memory can be recycled.

Epoch-based memory reclamation [16] is a popular approach to implementing this. The basic idea is for each thread to register (enter an epoch) upon accessing memory, and ensure the unlinked memory block is not recycled until all threads in the epoch have deregistered (exited). The epoch is advanced periodically depending pre-defined conditions, e.g., when the amount of allocated memory passes a threshold. An assumption is that data access are coordinated by thread boundaries. Under thread-to-transaction, a transaction exclusively uses all the resources associated with a thread, so thread boundaries are also transaction boundaries. Transactions can transparently use the epoch enter/exit machinery. However, under coroutine-to-transaction this may lead to memory corruption: Suppose transactions $T1$ and $T2$ run on the same thread, and $T1$ has entered epoch e before it suspends. Now the scheduler switches to $T2$ which is already in epoch e and issued epoch exit, allowing the memory to be freed, although it is still needed by $T1$ later.

CoroBase solves this problem by decoupling epoch enter/exit from transactions for the scheduler to issue them. Upon starting/finishing the processing of a batch of transactions, the worker thread enters/exits the epoch (lines 4 and 12 in Algorithm 1). This fits nicely with our scheduling policy which only handles whole batches. It also reduces the overhead associated with epoch-based reclamation as each thread registers/deregisters itself much less frequently. Another potential approach is to implement nested enter/exit machinery that allows a thread to register multiple times as needed. Though flexible, this approach is error-prone to implement, and brings much higher bookkeeping overhead.

Thread-Local Storage (TLS). TLS is widely used to reduce initialization and allocation overheads. In particular, ERMIA uses thread-local read/write sets and log buffers, as well as thread-local scratch areas for storing temporaries such as records that were read and new data to be added to the database. Logically, these structures are transaction-local. Although making these structures TLS greatly reduces memory allocation overheads, it conflicts with the coroutine-to-transaction paradigm, which must decouple threads and transactions. In other words, they need to be transaction-local to provide proper isolation among transactions. To reduce the performance impact, we expand each individual TLS variable to become an array of variables, one per transaction in the batch, and store them again as TLS variables. Upon system initialization, each worker thread creates the TLS array before starting to handle requests. When a transaction starts (e.g., the i-th in a batch), it takes the corresponding TLS array entry for use. This way, we avoid allocation/initialization overhead similar to how it was done under thread-to-transaction, but provide proper isolation among transactions. The tradeoff is that we consume (`batch_size` times) more memory space. As we show in Section 5, our approach makes a practical tradeoff as the optimal batch size does not exceed ten.

[5] Details in our code repo at lines 375–469 at https://github.com/sfu-dis/corobase/blob/v1.0/corobase.cc#L375.

4.5 Concurrency Control and Synchronization

CoroBase inherits the shared-everything architecture, synchronization and concurrency control protocols from ERMIA (snapshot isolation with the serial safety net [60] for serializability). A worker thread is free to access any part of the database. Version visibility is controlled using timestamps drawn from a global, monotonically increasing counter maintained by the engine. Upon transaction start, the worker thread reads the timestamp counter to obtain a begin timestamp b. When reading a record, the version with the latest timestamp that is smaller than b is visible. To update a record, the transaction must be able to see the latest version of the record. To commit, the worker thread atomically increments the timestamp counter (e.g., using atomic `fetch-and-add` [18]) to obtain a commit timestamp which is written on the record versions created by it.

We observe that adopting coroutine-to-transaction in shared-everything systems required no change for snapshot isolation to work. Similar to ERMIA, CoroBase adopts the serial safety net (SSN) [60], a certifier that can be applied on top of snapshot isolation to achieve serializability. It tracks dependencies among transactions and aborts transactions that may lead to non-serializable execution. Adapting SSN to CoroBase mainly requires turning TLS bitmaps used for tracking readers in tuple headers [60] into transaction-local. This adds `batch_size` bits per thread (compared to one in ERMIA). The impact is minimal because of the small ($\leq$10) batch size, and stalls caused by bitmap accesses can be easily hidden by prefetching. Devising a potentially more efficient approach under very high core count (e.g., 1000) is interesting future work. Since CoroBase allows multiple open transactions per thread, the additional overhead on supporting serializability may widen the conflict window and increase abort rate. Our experiments show that the impact is very small, again because the desirable batch sizes are not big (4–8).

For physical-level data structures such as indexes and version chains, coroutines bring extra challenges if they use latches for synchronization (e.g., higher chance to deadlock with multiple transactions open on a thread). However, in main-memory database engines these data structures mainly use optimistic concurrency without much (if not none of) locking. In CoroBase and ERMIA, index (Masstree [36]) traversals proceed without acquiring any latches and rely on versioning for correctness. This makes it straightforward to coroutinize the index structure for read/scan and part of update/insert/delete operations (they share the same traversal code to reach the leaf level). Locks are only held when a tree node is being updated. Hand-over-hand locking is used during structural modification operations (SMOs) such as splits. Our profiling results show that cache misses on code paths that use hand-over-hand locking make up less than 6% of overall misses under an insert-only workload. This is not high enough to benefit much from prefetching. Atomic instructions such as compare-and-swap used by most latch implementations also do not benefit much from prefetching. Therefore, we do not issue `suspend` on SMO code paths that use hand-over-hand locking.

More general, coroutine-centric synchronization primitives such as asynchronous mutex[6] are also being devised. How these primitives would apply to database systems remains to be explored.

[6]Such as the `async_mutex` in CppCoro: https://github.com/lewissbaker/cppcoro/blob/1140628b6e9e6048234d404cc393d855ae80d3e7/include/cppcoro/async_mutex.hpp.

Table 1: Changes needed to adopt coroutine-to-transaction in systems based on thread-to-transaction. The key is to ensure isolation between transactions on the same thread.

Component	Modifications
Concurrency Control (CC)	*Shared-everything:* transparent, but need careful deadlock handling if pessimistic locking used. *Shared-nothing:* re-introduce CC.
Synchronization	1. Thread-local to transaction-local. 2. Avoid holding latches upon suspension.
Resource Management	1. Thread-local to transaction-local. 2. Piggyback on batching to reduce overhead.
Durability	Transparent, with transaction-local log buffers.

4.6 Discussions

Coroutine-to-transaction only dictates how queries and transactions are interleaved. It does not require fundamental changes to components in ERMIA. Table 1 summarizes the necessary changes in ERMIA and engines that may make different assumptions than ERMIA's. Beyond concurrency control, synchronization and resource management, we find that the durability mechanism (logging, recovery and checkpointing) is mostly orthogonal to the execution model. The only change is to transform the thread-local log buffer to become transaction-local, which is straightforward.

Coroutine-to-transaction fits naturally with shared-everything, multi-versioned systems that use optimistic flavored concurrency control protocols. For pessimistic locking, coroutine-to-transaction may increase deadlock rates if transactions suspend while holding a lock. Optimistic and multi-version approaches alleviate this issue as reads and writes do not block each other, although adding serializability in general widens the conflict window.

Different from shared-everything systems, shared-nothing systems [56] partition data and restrict threads to only access its own data partition. This allowed vastly simplified synchronization and concurrency control protocols: in most cases if a transaction only accesses data in one partition, no synchronization or concurrency control is needed at all, as there is at most one active transaction at any time working on a partition. To adopt coroutine-to-transaction in shared-nothing systems, concurrency control and synchronization needs to be (re-)introduced to provide proper isolation between transactions running on the same thread.

Some systems [12, 13, 41, 52] explore intra-transaction parallelism to improve performance: a transaction is decomposed into pieces, each of which is executed by a thread dedicated to a partition of data, allowing non-conflicting data accesses in the same transaction to proceed in parallel. Data stalls may still occur as threads use pointer-intensive data structures (e.g., indexes) to access data. Coroutine-to-transaction can be adapted to model the individual pieces as coroutines to hide stalls. This would require changes such as a scheduler described in Algorithm 1 in the transaction executors in Bohm [12] and ReactDB [52]); we leave these for future work.

Finally, CoroBase removes the need for multi-key operations, but still supports them. A transaction can call a multi-key operation which interleaves operations within a transaction and does not

return control to the scheduler until completion. A transaction can also use operations in both interfaces to combine inter- and intra-transaction interleaving. For example, it can invoke a `get`, followed by an AMAC-based join to reduce latency and coroutine switching overhead. This hybrid approach can be attractive when the efforts for changing interfaces is not high. Section 5 quantifies the potential of this approach.

5 EVALUATION

Now we evaluate CoroBase to understand the end-to-end effect of software prefetching under various workloads. Through experiments, we confirm the following:

- CoroBase enables inter-transaction batching to effectively batch arbitrary queries to benefit from software prefetching.
- In addition to read-dominant workloads, CoroBase also improves on read-write workloads while remaining competitive for workloads that inherently do not benefit from software prefetching.
- CoroBase can improve performance with and without hyperthreading, on top of hardware prefetching.

5.1 Experimental Setup

We use a dual-socket server equipped with two 24-core Intel Xeon Gold 6252 CPUs clocked at 2.1GHz (up to 3.7GHz with turbo boost). The CPU has 35.75M last-level cache. In total the server has 48 cores (96 hyperthreads) and 384GB main memory occupying all the six channels per socket to maximize memory bandwidth. We compile all the code using Clang 10 with coroutine support on Arch Linux with Linux kernel 5.6.5. All the data is kept in memory using `tmpfs`. We report the average throughput and latency numbers of three 30-second runs of each experiment.

System Model. Similar to prior work [24, 25, 59], we implement benchmarks in C++ directly using APIs exposed by the database engine, without SQL or networking layers. Using coroutines to alleviate overheads in these layers is promising but orthogonal work. The database engine is compiled as a shared library, which is then linked with the benchmark code to perform tests.

Variants. We conduct experiments using the following variants which are all implemented based on ERMIA [24].[7]

- `Naïve`: Baseline that uses thread-to-transaction and executes transactions sequentially without interleaving or prefetching.
- `ERMIA`: Same as `Naïve` but with `prefetch` instructions carefully added to index and version chain traversal code.
- `AMAC-MK`: Same as `ERMIA` but applications use hand-crafted multi-key interfaces based on AMAC.
- `CORO-MK`: Same as `ERMIA` but applications use multi-key interfaces based on flattened coroutines.
- `CORO-FN-MK`: Same as `CORO-MK` but with fully-nested coroutines described in Section 2.2.
- `CoroBase-FN`: CoroBase that uses the fully-nested coroutine-to-transaction design. No changes in applications.
- `CoroBase`: Same as `CoroBase-FN` but uses the optimized 2-level coroutine-to-transaction design described in Section 4.3.
- `Hybrid`: Same as `CoroBase` but selectively leverages multi-key interfaces in TPC-C transactions (details in Section 5.8).

[7] ERMIA code downloaded from https://github.com/sfu-dis/ermia.

We use a customized allocator to avoid coroutine frame allocation/deallocation bottlenecks. Hardware prefetching is enabled for all runs. For interleaved executions, we experimented with different batch sizes and use the optimal setting (eight) unless specified otherwise. We use snapshot isolation[8] for all runs except for pure index-probing workloads which do not involve transactions (described later). We also perform experiments with and without hyperthreading to explore its impact.

5.2 Benchmarks

We use both microbenchmarks and standard benchmarks to stress test and understand the end-to-end potential of CoroBase.

Microbenchmarks. We use YCSB [9] to compare in detail the impact of different design decisions. The workload models point read, read-modify-write (RMW), update and scan transactions on a single table with specified access patterns. We use a ~15GB database of one billion records with 8-byte keys and 8-byte values.

Standard Benchmarks. We use TPC-C [57] to quantify the end-to-end benefits of software prefetching under CoroBase. To show comprehensively how CoroBase performs under a realistic and varying set of workloads with different read/write ratios, we run both the original TPC-C and two variants, TPC-CR [61] and TPC-CH [24]. TPC-CR is a simplified read-only version of TPC-C that comprises 50% of StockLevel and 50% OrderStatus transactions. TPC-CH adds a modified version of the Query2 transaction (Q2*) in TPC-H [58] to TPC-C's transaction mix. This makes TPC-CH a heterogeneous workload that resembles hybrid transaction-analytical processing (HTAP) scenarios.[9] We use the same implementation in ERMIA [24] where the transaction picks a random region and updates records in the stock table whose quantity is lower than a pre-defined threshold. The size of Q2* is determined mainly by the portion of the suppliers table it needs to access. We modify the transaction mix to be 10% of Q2*, 40% of NewOrder, 38% of Payment, plus 4% of StockLevel, Delivery and OrderStatus each. For all TPC-C benchmarks, we set scale factor to 1000 (warehouses). Each transaction uniform-randomly chooses and works on their home warehouse, but 1% of New-Order and 15% of Payment transactions respectively access remote warehouses.

5.3 Sequential vs. Interleaved Execution

As mentioned in earlier sections, our goal in this paper is to adopt coroutine-based interleaving in a database engine and understand its end-to-end benefits. Therefore, it is important to set the proper expectation on the possible gain that could be achieved by CoroBase. To do this, we run a simple non-transactional index probing workload where worker threads keep issuing `multi_get` requests. Each `multi_get` issues 10 requests against the index but does not access the actual database record. The workload is similar to what prior work [21] used to evaluate index performance. As Figure 5 shows, on average `AMAC-MK` outperforms `Naïve`/`ERMIA` by up to 2.3×/2.96×

[8] We also ran experiments under the serializable isolation level (using SSN on top of snapshot isolation). The results show that SSN adds a fixed amount of overhead (~10–15%, similar to the numbers reported earlier for thread-to-transaction systems [60]). We therefore focus on experiments under SI for clarity.

[9] Our focus is on OLTP (Section 2.3). We run TPC-CH to explore how CoroBase performs under various read-intensive workloads. Hiding data stalls in OLAP workloads requires further investigation that considers various access patterns and constraints.

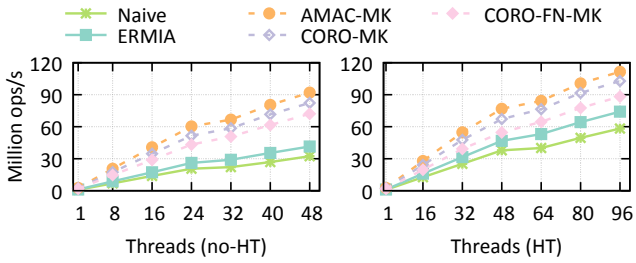

Figure 5: Index probing throughput with hyperthreading disabled (left) and enabled (right). Fully-nested coroutines are ~30% slower than AMAC. Flattened coroutines (CORO-MK) reduce the gap to 8–10% under high concurrency.

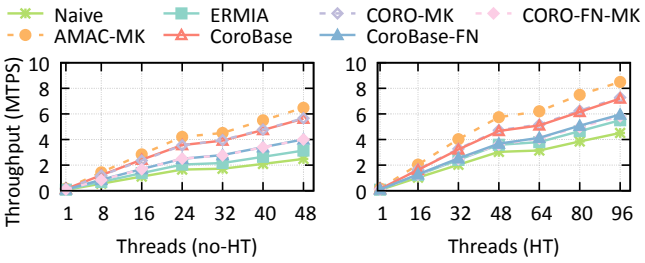

Figure 6: Throughput of a read-only YCSB workload (10 reads per transaction) without (left) and with (right) hyperthreading. CoroBase matches the performance of multi-key variants but without requiring application changes.

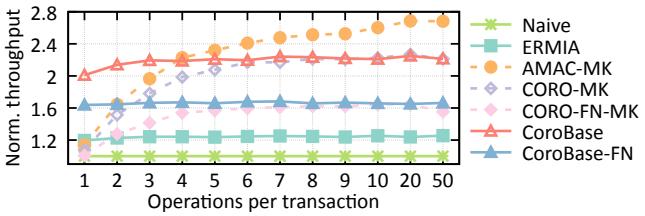

Figure 7: YCSB read-only performance normalized to Naïve under 48 threads. CoroBase also benefits very short transactions (1–4 operations) while multi-key based approaches inherently cannot under thread-to-transaction.

without hyperthreading. Since AMAC-MK does not incur much overhead in managing additional metadata like coroutine frames, these results set the upper bound of the potential gain of interleave execution. However, AMAC-MK uses highly-optimized but complex, hand-crafted code, making it much less practical. Using single-level coroutines, CORO-MK achieves up to 2.56×/1.99× higher throughput than Naïve/ERMIA, which is 17% faster than CORO-FN-MK because of its lower switching overhead. With hyperthreading, the improvement becomes smaller across all variants.

These results match what was reported earlier in the literature, and set the upper bound for CoroBase to be up to ~2× faster than optimized sequential execution that already uses prefetching (i.e., the ERMIA variant), or ~2.5× faster than Naïve under read-only workloads. In the rest of this section, we mark the upper bound and multi-key variants that require interface changes as dashed lines in figures, and explore how closely CoroBase matches the upper bound under various workloads.

5.4 Effect of Coroutine-to-Transaction

Our first end-to-end experiment evaluates the effectiveness of coroutine-to-transaction. We use a read-only YCSB workload where each transaction issues 10 record read operations that are uniform randomly chosen from the database table; we expand on to other operations (write and scan) later. Note that different from the previous probe-only experiment in Section 5.3, from now on we run fully transactional experiments that both probe indexes and access database records. We compare variants that use coroutine-to-transaction (CoroBase and CoroBase-FN) with other variants that use highly-optimized multi-key interfaces and thread-to-transaction. Figure 6 plots result using physical cores (left) and hyperthreads (right). Compared to the baselines (ERMIA and Naïve), all variants show significant improvement. Because of the use of highly-optimized, hand-crafted state machines and multi-key interfaces, AMAC-MK outperforms all the other variants. Without hyperthreading, CoroBase exhibits an average of ~15/~18% slowdown from AMAC-MK, but is still ~1.3/1.8× faster than ERMIA when hyperthreading is enabled/disabled. This is mainly caused by the inherent overhead of the coroutine machinery. CORO-FN-MK and CoroBase-FN are only up ~1.25× faster than ERMIA due to high switching overhead. CORO-MK and CoroBase minimize switching overhead by flattening the entire record access call chain (index probing and version chain traversal). The only difference is CORO-MK uses multi_get, while CoroBase allows the application to remain unchanged using single-key interfaces. Therefore, it is necessary to flatten call chains as much as possible to reduce context switching overhead. As the coroutine infrastructure continues to improve, we expect the gap between CoroBase, CoroBase-FN and AMAC-MK to become smaller.

Coroutine-to-transaction also makes it possible for short transactions to benefit from prefetching. We perform the same YCSB read-only workload but vary the number of reads per transaction. As shown in Figure 7, CoroBase outperforms all multi_get approaches for very short transactions (1–4 record reads) and continues to outperform all approaches except AMAC-MK for larger transactions, as batches are formed across transactions.

These results show that inter-transaction batching enabled by coroutine-to-transaction can match closely the performance of intra-transaction batching while retaining backward compatibility. Hyperthreading helps to a limited extent and CoroBase can extract more performance, partially due to the limited hardware contexts (two per core) available in modern Intel processors. Compared to prior approaches, CoroBase and coroutine-to-transaction further enable short transactions (with little/no intra-transaction batching opportunity) to also benefit from software prefetching.

5.5 Write and Scan Workloads

One of our goals is to use software prefetching to hide data stalls as much as possible. While most prior work focused on read-only or read-dominant operations, we extend our experiments to cover write-intensive scenarios. Write operations also need to traverse

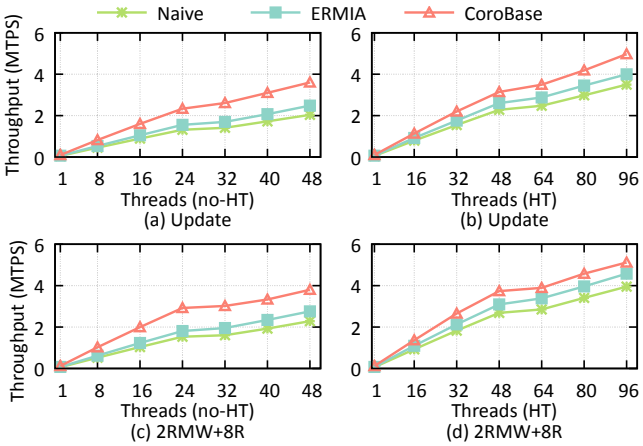

Figure 8: Throughput of update-only YCSB with 10 blind writes per transaction (a–b), and a mixed workload (c–d).

index and version chains to reach the target record where data stalls constitute a significant portion of stalled cycles. Figure 8(a–b) plots the throughput of update-only (blind write) workload, where each transaction updates 10 uniform randomly-chosen records. CoroBase achieves up to 1.77× and 1.45× higher throughput than Naïve and ERMIA, respectively. We observe similar results but with lower improvement for a mixed workload that does two RMWs and eight reads per transaction, shown in Figure 8(c–d); a pure RMW workload showed similar trends (not shown here). The lower improvement comes from the fact that the read operation before each modify-write has already brought necessary data into CPU caches, making subsequent transaction switches for modify-write pure overhead. Moreover, compared to read operations, write operations need to concurrently update the version chain using atomic instructions, which cannot benefit much from prefetching. Nevertheless, the results show that even for write-intensive workloads, prefetching has the potential of improving overall performance as data stalls still constitute a significant portion of total cycles.

Unlike point-read operations, we observe that scan operations do not always benefit as much. Figure 9 shows the throughput of a pure scan workload. As we enlarge the number of scanned records, the performance of Naïve, ERMIA and CoroBase converges. This is because Masstree builds on a structure similar to B-link-trees [28] where border nodes are linked, minimizing the need to traverse internal nodes. It becomes more possible for the border nodes to be cached and with longer scan ranges, more records can be retrieved directly at the leaf level, amortizing the cost of tree traversal.

5.6 Impact of Key Lengths and Data Sizes

Now we examine how key length, value size, and database size affect runtime performance. Figure 10(left) shows the throughput of the YCSB read-only workload under different key lengths. CoroBase retains high performance across different key lengths, but CoroBase-FN performs worse under longer keys which cause more coroutine calls to drill down Masstree's hybrid trie/B-tree structure, causing high switching overhead. As shown in Figure 10(right), value size does not impact the overall relative merits of different

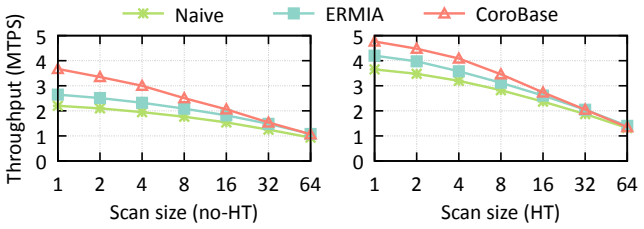

Figure 9: Throughput of a YCSB scan workload. The benefits of prefetching diminish with larger scan sizes: more records can be directly retrieved in leaf nodes using B-link-tree structures.

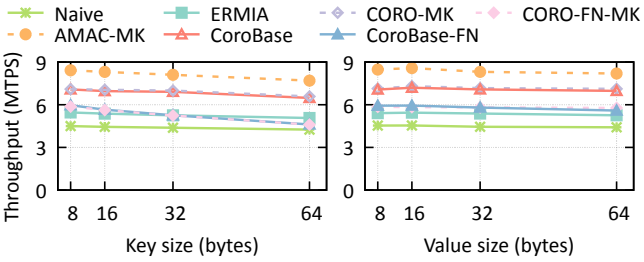

Figure 10: Throughput of read-only YCSB (10 reads per transaction) under varying key/value sizes and 96 hyperthreads.

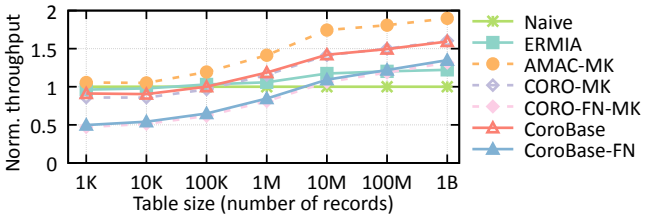

Figure 11: YCSB read-only (10 reads per transaction) performance normalized to Naïve under 96 hyperthreads and varying table sizes.

variants. Figure 11 depicts the impact of data size by plotting the throughput normalized to that of Naïve. As shown, with small table sizes, interleaving does not improve performance. For example, with 10K records, the total data size (including database records, index, etc.) is merely 1.23MB, whereas the CPU has 35MB of last level cache, making all the anticipated cache misses cache hits. Context switching becomes pure overhead and exhibits up to ~12.64% lower throughput (compared to Naïve). Approaches that use fully-nested coroutines (CoroBase-FN and CORO-FN-MK) exhibit even up to 50% slower performance, whereas other approaches including CoroBase keep a very low overhead. In particular, CoroBase follows the trend of AMAC-MK with a fixed amount of overhead.

These results again emphasize the importance of reducing switching overhead. CoroBase's low switching overhead for longer keys and small tables make it a practical approach for systems to adopt. We also expect coroutine support in compilers and runtime libraries to continue to improve and reduce switching overhead in the future.

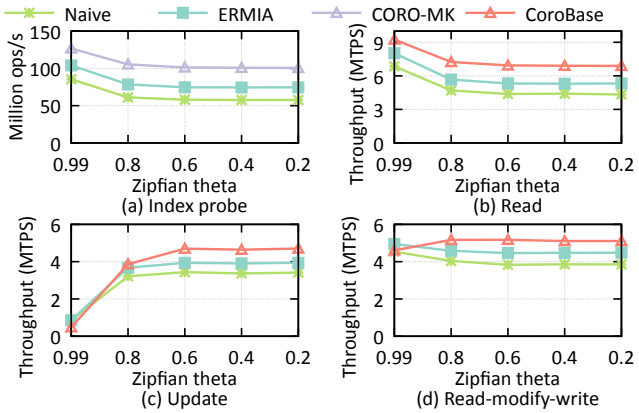

Figure 12: YCSB throughput with 96 hyperthreads and varying skewness (larger theta indicates more skewed access).

5.7 Impact of Skewed Accesses

Our last microbenchmark tests how different approaches perform under varying skewness. Figure 12(a) calibrates the expectation using a multi-get based index probing workload that does not access records. Each transaction here issues 10 operations. With higher skewness, all schemes perform better because of the better locality. The YCSB read-only workload in Figure 12(b) shows a similar trend. For update and RMW operations shown in Figures 12(c–d), highly skewed workloads lead to high contention and low performance across all schemes, and memory stall is no longer the major bottleneck. `CoroBase` therefore shows lower performance compared to `ERMIA` as switching becomes pure overhead.

5.8 End-to-End TPC-C Results

Now we turn to TPC-C benchmarks to see how prefetching works in more complex workloads. We begin with the default TPC-C configuration which is write-intensive. As shown in Table 2, CoroBase manages to perform marginally better than `Naïve` and `ERMIA` without hyperthreading but is 3.4% slower than `ERMIA` with hyperthreading. One reason is that TPC-C is write-intensive and exhibits good data locality, with fewer exposed cache misses as shown by the narrow gap between `Naïve` and `ERMIA`. Our top-down microarchitecture analysis [18] result shown in Figure 13(b) verifies this: TPC-C exhibits less than 50% of memory stall cycles, which as previous work [44] pointed out, do not provide enough room to benefit from prefetching. The high memory stall percentage in Figure 13(a) confirms our YCSB results which showed more improvement. Under TPC-CR which is read-only, `CoroBase` achieves up to 1.55×/1.3× higher throughput with/without hyperthreading (Figure 14). Notably, with hyperthreading `CoroBase` performs similarly (4% better) to `ERMIA`, showing that using two hyperthreads is enough to hide the memory stalls that were exposed on physical cores for TPC-CR. Interleaved execution is inherently not beneficial for such workloads, so the goal is for CoroBase is to match the performance of sequential execution as close as possible.

Under TPC-CH which is read-intensive and exhibits high memory stall cycles, `CoroBase` achieves up to 34% higher throughput than `ERMIA` when the number of suppliers is set to 100 under 48

Table 2: Throughput (TPS) of original TPC-C which is not inherently memory-bound. With two-level coroutines, `CoroBase` outperforms `Naïve` and matches `ERMIA` and `Hybrid`.

Number of Threads	Naïve	ERMIA	CoroBase	Hybrid
48 (no-HT)	1306197	1487147	1489567	1682290
96 (HT)	1985490	2195667	2120033	2197260

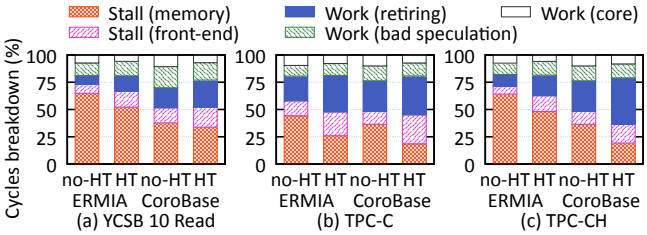

Figure 13: Microarchitecture analysis for YCSB, TPC-C and TPC-CH workloads. CoroBase reduces memory stall cycles under all workloads, especially the read-dominant ones.

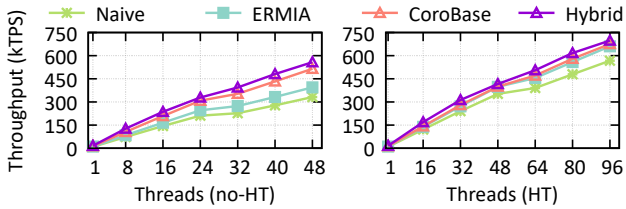

Figure 14: Throughput of the read-only TPC-CR workload.

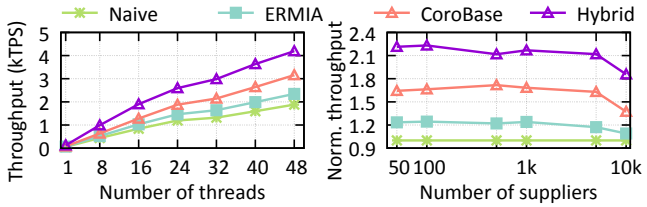

Figure 15: TPC-CH scalability (left) and throughput with 48 threads normalized to `Naïve` (right). CoroBase reaps more benefits from prefetching with more read operations.

threads in Figure 15(left). Figure 15(right) explores how throughput changes as the size of the Q2* transaction changes. `CoroBase` exhibits 25–39% higher throughput than `ERMIA`, and the numbers over `Naïve` are 37–71%. Correspondingly, Figure 13(c) shows fewer memory stall cycles under CoroBase.

We explore the potential of using selective multi-key operations in CoroBase (Section 4.6) with the `Hybrid` variant. We use `multi_get` coroutines for long queries in NewOrder, StockLevel and Query2 to retrieve items and supplying warehouses, recently sold items and items from certain pairs of the Stock and Supplier tables, respectively. Other queries use the same single-key operations as in `CoroBase`. As shown in Table 2 and Figures 14–15,

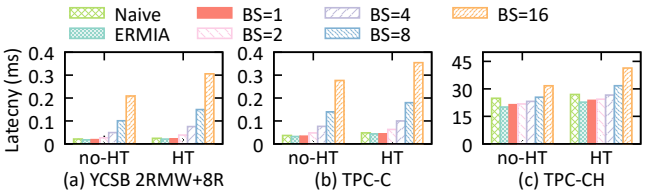

Figure 16: Average transaction latency under varying batch sizes (BS=1–16) with asynchronous commit. The end-to-end impact is expected to be small with group/pipelined commit.

`Hybrid` outperforms `CoroBase` by up to 1.29×/1.08×/1.36× under TPC-C/TPC-CR/TPC-CH, taking advantage of data locality and reduced switching overhead. The tradeoff is increased code complexity. For operators that already exhibit or are easily amenable to multi-key interfaces, `Hybrid` can be an attractive option.

5.9 Impact on Transaction Latency

We analyze the impact of interleaved execution on transaction latency using a mixed YCSB workload (2 RMW +8 read operations per transaction), TPC-C and TPC-CH. As shown in Figure 16, with larger batch sizes, transaction latency increases. We find setting batch size to four to be optimal for the tested YCSB and TPC-C workloads: when batch size exceeds four, latency grows proportionally since there is no room for interleaving. TPC-CH exhibits smaller increase in latency, indicating there is much room for overlapping computation and data fetching. Hyperthreading also only slightly increases average latency. Note that in this experiment we use asynchronous commit which excludes I/O cost for persisting log records. Many real systems use group/pipelined commit [20] to hide I/O cost. The result is higher throughput but longer latency for individual transactions (e.g., 1–5ms reported by recent literature [61]). Therefore, we expect the increased latency's impact on end-to-end latency in a real system to be very low.

5.10 Coroutine Frame Management

Flattening coroutines may increase code size and incur more instruction cache misses (Section 4.3). For two-level coroutine-to-transaction, the size of a coroutine frame for the flattened read-/update/insert functions are 232/200/312 bytes, respectively. With fully-nested coroutine-to-transaction, five small functions are involved, and their size range from 112–216 bytes, which are indeed smaller than their flattened counterparts. When handling a request, CoroBase allocates a single coroutine frame. `CoroBase-FN` maintains a chain of at least five coroutine frames (e.g., for reads it adds up to 808 bytes per request) and switches between them. Performance drops with both larger memory footprint (therefore higher allocation/deallocation overhead) and switching. Overall, two-level coroutines ease this problem and achieves better performance.

6 RELATED WORK

Our work is closely related to prior work on coroutine-based systems, cache-aware optimizations and database engine architectures.

Interleaving and Coroutines. We have covered most related work in this category [5, 21, 26, 44, 45] in Section 2, so we do not repeat here. The gap between CPU and the memory subsystem continues to widen with the introduction of more spacious but slower persistent memory [10]. Psaropoulos et al. [46] adapted interleaved execution using coroutine to speed up index joins and tuple construction on persistent memory. Data stalls are also becoming a bottleneck for vectorized queries [23, 42]. IMV [14] interleaves vectorized code to reduce cache misses in SIMD vectorization.

Cache-Aware Optimizations. Many proposals try to improve locality. CSB+-tree [48] stores child nodes contiguously to better utilize the cache but trades off update performance. ART [29] is a trie that uses a set of techniques to improve space efficiency and cache utilization. HOT [2] dynamically adjusts trie node span based on data distributions to achieve a cache-friendly layout. Prefetching B+-trees [6] uses wider nodes to reduce B+-tree height and cache misses during tree traversal. Fractal prefetching B+-trees [7] embed cache-optimized trees within disk-optimized trees to optimize both memory and I/O. Masstree [36] is a trie of B+-trees that uses prefetching. Software prefetching was also studied in the context of compilers [39]. At the hardware level, path prefetching [32] adds a customized prefetcher to record and read-ahead index nodes. Widx [27] is an on-chip accelerator that decouples hashing and list traversal and processes multiple requests in parallel.

Database Engine Architectures. Most main-memory database engines [11, 22, 24, 25, 59] use the shared-everything architecture that is easy to be adapted to coroutine-to-transaction. Some systems [12, 13, 41, 52] allow intra-transaction parallelism with delegation. Techniques in CoroBase are complementary to and can be used to hide data stalls in these systems (Section 4.6). To adopt coroutine-to-transaction in shared-nothing systems [56], concurrency control and synchronization need to be re-introduced to allow context switching. Data stall issues were also identified in column stores [4, 17, 55] and analytical workloads [53]. Exploring ways to hide data stalls in these systems is interesting future work.

7 CONCLUSION

We highlighted the gap between software prefetching and its adoption in database engines. Prior approaches often break backward compatibility using multi-key interfaces and/or are piece-wise solutions that optimize individual database operations. Leveraging recently standardized lightweight coroutines, we propose a new coroutine-to-transaction execution model to fill this gap. Coroutine-to-transaction retains backward compatibility by allowing inter-transaction batching which also enables more potential of software prefetching. Based on coroutine-to-transaction, we build CoroBase, a main-memory database engine that judiciously leverages coroutines to hide memory stalls. CoroBase achieves high performance via a lightweight two-level coroutine design. Evaluation results show that on a 48-core server CoroBase is up to ~2× faster than highly-optimized baselines and remains competitive for workloads that inherently do not benefit from software prefetching.

ACKNOWLEDGMENTS

We thank Qingqing Zhou and Kangnyeon Kim for their valuable discussions. We also thank the anonymous reviewers for their useful feedback. This paper is dedicated to the memory of our wonderful colleague Ryan Shea.

REFERENCES

[1] Frances E. Allen and J. Cocke. 1972. A catalogue of optimizing transformations. In *Design and Optimization of Compilers* (1 ed.). Prentice Hall.

[2] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 521–534.

[3] Omer Boehm, Daniel Citron, Gadi Haber, Moshe Klausner, and Roy Levin. 2006. Aggressive Function Inlining with Global Code Reordering. *IBM Research Report* (2006), 1–26.

[4] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 54–65.

[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] Melvin E Conway. 1963. Design of a Separable Transition-Diagram Compiler. *Commun. ACM* 6, 7 (1963), 396–408.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] Rob Crooke and Mark Durcan. 2015. A Revolutionary Breakthrough in Memory Technology. *3D XPoint Launch Keynote* (2015).

[11] 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.

[12] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (July 2015), 1190–1201.

[13] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 613–624.

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

[15] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

[16] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.* 67, 12 (Dec. 2007), 1270–1285.

[17] Stratos Idreos, F. Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35 (01 2012).

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

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

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

[21] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (July 2018), 1702–1714.

[22] 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.

[23] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222.

[24] 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* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1675–1687.

[25] 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* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 691–706.

[26] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 252–263.

[27] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for in-Memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 468–479.

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

[29] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 38–49.

[30] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)*.

[31] 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.

[32] Shuo Li, Zhiguang Chen, Nong Xiao, and Guangyu Sun. 2018. Path Prefetching: Accelerating Index Searches for In-Memory Databases. In *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*. IEEE Computer Society, 274–277.

[33] 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.

[34] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. 222–233.

[35] Stefan Manegold, Martin L. Kersten, and Peter Boncz. 2009. Database Architecture Evolution: Mammals Flourished Long before Dinosaurs Became Extinct. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1648–1653.

[36] 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.

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

[38] Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 2 (2009), 1–31.

[39] 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.

[40] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, Berlin, Heidelberg, 213–228.

[41] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-Oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 928–939.

[42] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508.

[43] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 164–179.

[44] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 230–242.

[45] 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.

[46] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap between NVM and DRAM for Latency-Bound Operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 13, 8 pages.

[47] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems* (3 ed.). McGraw-Hill, Inc., New York, NY, USA.

[48] Jun Rao and Kenneth A. Ross. 2000. Making B+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference*

on Management of Data (Dallas, Texas, USA) *(SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 475–486.

[49] Silvano Rivoira and Carlo Alberto Ferraris. 2011. Compiler optimizations based on call-graph flattening. *MSc Thesis* (2011).

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

[51] Boris Schling. 2020. *The Boost C++ Libraries.* https://theboostcpplibraries.com/.

[52] Vivek Shah and Marcos Antonio Vaz Salles. 2018. Reactors: A Case for Predictable, Virtualized Actor Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 259–274.

[53] Utku Sirin and Anastasia Ailamaki. 2020. Micro-Architectural Analysis of OLAP: Limitations and Opportunities. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 840–853.

[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] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 553–564.

[56] 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.

[57] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark C (OLTP) Standard Specification, revision 5.11. http://www.tpc.org/tpcc.

[58] Transaction Processing Performance Council (TPC). 2018. TPC Benchmark H (Decision Support) Standard Specification, revision 2.18.0. http://www.tpc.org/tpch.

[59] 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.

[60] 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.

[61] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 406–419.

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

[63] Xuejun Yang, Nathan Cooprider, and John Regehr. 2009. Eliminating the Call Stack to Save RAM. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '09)*. 60–69.