Low-Latency Transaction Scheduling via Userspace Interrupts

Why Wait or Yield When You Can Preempt? [To Appear at SIGMOD '25]

Kaisong Huang
Simon Fraser University
kha85@sfu.ca

Jiatang Zhou
Simon Fraser University
jiatangz@sfu.ca

Zhuoyue Zhao
University at Buffalo
zzhao35@buffalo.edu

Dong Xie
The Pennsylvania State University
dongx@psu.edu

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

ABSTRACT

Traditional non-preemptive scheduling can lead to long latency under workloads that mix long-running and short transactions with varying priorities. This occurs because worker threads tend to monopolize CPU cores until they finish processing long-running transactions. Thus, short transactions must wait for the CPU, leading to long latency. As an alternative, cooperative scheduling allows for transaction yielding, but it is difficult to tune for diverse workloads. Although preemption could potentially alleviate this issue, it has seen limited adoption in DBMSs due to the high delivery latency of software interrupts and concerns on wasting useful work induced by read-write lock conflicts in traditional lock-based DBMSs.

In this paper, we propose PreemptDB, a new database engine that leverages recent userspace interrupts available in modern CPUs to enable efficient preemptive scheduling. We present an efficient transaction context switching mechanism purely in userspace and scheduling policies that prioritize short, high-priority transactions without significantly affecting long-running queries. Our evaluation demonstrates that PreemptDB significantly reduces end-to-end latency for high-priority transactions compared to non-preemptive FIFO and cooperative scheduling methods.

CCS CONCEPTS

• **Information systems** → **Main memory engines**; **Database transaction processing**; **Online analytical processing engines**.

KEYWORDS

Preemptive Scheduling, User Interrupts, Database Systems, Low-Latency Transactions

ACM Reference Format:
Kaisong Huang, Jiatang Zhou, Zhuoyue Zhao, Dong Xie, and Tianzheng Wang. 2025. Low-Latency Transaction Scheduling via Userspace Interrupts: Why Wait or Yield When You Can Preempt? [To Appear at SIGMOD '25]. In *Proceedings of International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

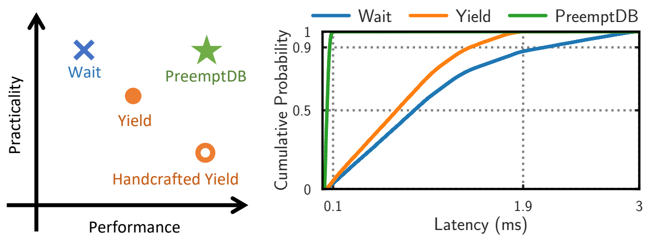


Figure 1: Design space (left) and scheduling latency distribution (right) of high-priority short transactions in a workload mixed with long-running transactions.

1 INTRODUCTION

Modern database applications increasingly mix various types of transactions. For example, heavy-weight, long operational reporting jobs [7] could run alongside short, latency-sensitive sales transactions for e-commerce businesses to identify sales trends and adjust advertising strategies with maximum data freshness [39]. In essence, such workloads consist of both (1) low-priority but long and (2) high-priority but short transactions. The former often monopolize the CPU, leaving little to no resources to run the latter. This can lead to long transaction scheduling latency and high tail latency overall, which we aim to reduce in this work.

1.1 The Practicality and Latency Tradeoff

It is intuitive to allow high-priority transactions to interrupt low-priority long-running transactions for CPU resources, but conventional wisdom [6] has long discouraged preemption in database engines. The rationale was that using preemption would introduce excessive overhead, since a low-priority but long transaction may be holding many locks, and so a high-priority transaction may conflict with the long transaction, causing it to abort anyway. This can waste useful work or even lead to deadlocks, defeating the purpose of preemption. As a result, many systems are forced to make tradeoffs between performance (transaction scheduling latency) and practicality. Figure 1(left) summarizes the design space. Many systems schedule transactions in a first-in-first-out (FIFO) manner, cooperatively, or leave it to the OS.

FIFO-based scheduling (denoted as `Wait`) mandates high-priority transactions to wait until existing low-priority transactions finish. Long-running low-priority transactions such as operational reporting can monopolize CPU resources. As shown in Figure 1 (right;

detailed setup in Section 6), this essentially forgoes priority, leading to long scheduling delays for high-priority transactions (e.g., by waiting in a scheduling queue). A stop-gap solution is cooperative scheduling [11, 15, 16], which allows transactions to give up CPU time voluntarily. This can be implemented by defining "yield points" in the database engine at appropriate locations without involving the application. For example, one could yield once after reading a certain number of records. Such approaches (`Yield` in Figure 1) could lower scheduling latency and is usually not hard to implement, but can still fall short if the choice of yield points does not exactly match the workload. Potentially, one could handcraft a yield strategy according to a target workload to achieve very high performance (i.e., low scheduling latency), as shown in Figure 1(left). But this is difficult to maintain and not cost-effective as the database engine is customized for a particular workload. Overall, it is still a fundamental challenge for database engines to schedule mixed workloads with low latency.

1.2 PreemptDB: Preemption Made Practical

We propose PreemptDB, a database engine that delivers low scheduling latency for mixed workloads with both low-priority long-running transactions and high-priority short transactions. Contrary to the conventional wisdom, PreemptDB does so based on two observations that make preemption viable in modern DBMSs.

First, the tendency of avoiding preemption centered around the use of pessimistic locking (e.g., two-phase locking) across the board for all transactions, including long-running, low-latency ones. However, modern database engines are increasingly memory-optimized, adopting optimistic [25, 48] and multi-versioned [3, 4, 13, 24, 28, 30, 37, 51, 52] approaches to concurrency control that allow read operations to proceed without holding locks. This means interrupting a long read may not lead to severe waste of work or cause transactions to abort.

Second, advances in modern CPUs, particularly user interrupt (uintr) [33] that is available now in mainstream x86 chips can deliver interrupts directly into userspace, providing lightweight mechanisms for preemption. This allows database engines to facilitate interrupt delivery quickly purely in the userspace without crossing kernel and userspace boundaries.

PreemptDB leverages these two facts to design a lightweight transaction scheduling mechanism and associated policies. We base on user interrupts [33] to allow each worker thread to switch between multiple contexts that time-share the underlying CPU core. This way, a low-priority transaction running on a worker thread can be timely interrupted (with its states stored in a transaction control block, or TCB), so that the high-priority transaction can be served by another context on the same thread. Contrary to traditional process context switching in the OS, our context switch mechanism is purely in userspace and eliminates costly kernel transitions. This significantly reduces latency for high-priority transactions and allows PreemptDB to maintain high performance in case no preemption is needed (e.g., if the system is serving uniform OLTP workloads with short transactions only).

Unlike traditional preemptive policies that abort transactions and thus may waste valuable work [32] done for long-running transactions, PreemptDB leverages the fact that modern systems issue optimistic reads to pause preempted transactions. Once the high-priority transaction is finished, PreemptDB resumes the previously paused low-priority transaction. PreemptDB also addresses several challenges inherent in preemptive scheduling, such as potential concurrency problems and transaction starvation, which we discuss in later sections.

As shown in Figure 1(right), PreemptDB outperforms `Wait` and `Yield`-based solutions by orders of magnitude in reducing latency. Our evaluation in Section 6 shows that PreemptDB lowers latency of high-priority transactions by 88–96% over non-preemptive scheduling policies at different latency percentiles. Importantly, PreemptDB does so without sacrificing overall transaction throughput. It is also robust by not requiring fine-tuning based on workloads.

1.3 Contributions and Paper Organization

This paper makes the following contributions:

- We revisit transaction scheduling and make the case for preemptive scheduling in modern database engines.
- We propose to leverage userspace interrupt primitives available in modern CPUs to design a low-overhead, fine-grained preemptive transaction scheduling mechanism.
- We present PreemptDB, a database engine that uses user interrupts to support preemptive scheduling policies to reduce transaction scheduling latency, and discuss the broader implications of preemptive scheduling on future database system designs.
- Through extensive evaluation, we show that PreemptDB improves throughput, latency, and fairness, proving its practicality.

In the rest of the paper, we give the necessary background in Section 2. Section 3 gives the desiderata of preemptive scheduling which serves as our design principles. Sections 4–5 present the detailed design of PreemptDB. Section 6 empirically evaluates PreemptDB to demonstrate the benefits of preemptive scheduling. We cover related work in Section 7 and conclude in Section 8.

2 BACKGROUND

As database applications evolve, the need to handle both long-running analytical and short transactions within a single database engine has become increasingly prevalent. In particular, the long-running analytical transactions are often of low priority, while the operational operations are traditional OLTP that require low-latency and high-priority. Many real-world applications (e.g., financial systems and e-commerce) [10] present such mixed workloads which are our target in this paper.

In the rest of this section, we discuss existing scheduling approaches and their mismatch with our target workloads to motivate work. We then set the stage for our solution by introducing user interrupt primitives available in modern x86 CPUs, followed by our assumptions on system model.

2.1 Existing Wait/Yield based Scheduling

We have briefly introduced the current state-of-the-art wait/yield-based scheduling in Section 1. Now we spell out why they fall short in handling mixed workloads with varying priorities; readers already familiar with this topic may fast forward to Section 2.2.

Non-Preemptive and Cooperative Scheduling. Long analytical queries can dominate CPU cycles due to their highly parallel

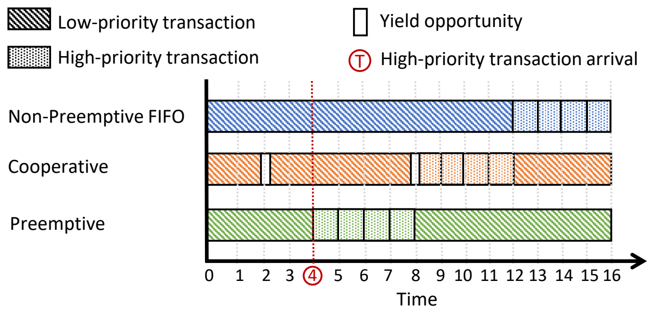
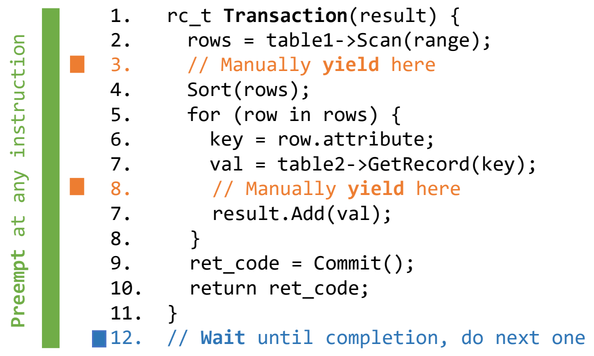


Figure 2: Traditional wait/yield-based scheduling vs. preemption. Preemption could potentially lower transaction scheduling latency by responding to new requests quickly.

nature and long execution times [12, 26, 40]. For example, many systems execute large scan, join and aggregation operations across multiple threads. However, operational transactions often require low-latency responses. As a result, in a database engine where transactions with footprints on different scales share the same hardware resource, high-priority transactions are very likely to be delayed.

Many recent database engines focus on hybrid transactional and analytical processing (HTAP) architectures [8, 23, 34, 36, 38, 42, 44, 50, 53] where the database must support a wide range of scenarios concurrently. However, when CPU resources are already fully utilized by long-running transactions, traditional scheduling mechanisms like non-preemptive FIFO scheduling and cooperative scheduling become insufficient. As the example in Figure 2 shows, if a short, high-priority transaction arrives at $time = 4$, using non-preemptive FIFO scheduling, it must wait for the long-running transaction to complete. Therefore, the high-priority transaction is delayed until $time = 12$. While the execution time of the high-priority transaction is only 1 time unit, the scheduling latency is 8 times of the execution time. Such FIFO methods can be optimized, e.g., by relaxing strict FIFO ordering to organize transactions into a high-priority queue and a low-priority queue, so that the thread always exhausts the former first. As Section 6 shows, they still lead to high tail latency.

In contrast, cooperative scheduling can reduce scheduling latency by allowing transactions to yield the CPU voluntarily. This can be done in two ways. (1) The application could be modified to yield at specific points, e.g., at lines 3 and 8 of Figure 3. This reduces the wait time for the example in Figure 2 to 4 time units, but is still far from responsive. If the high-priority transaction missed the first yield point, it would have to wait for the second one, which can lead to unpredictable latency, depending on how much work remains to be done before the worker thread hits the next yield point. This approach adds additional overheads to application developers who are often not well trained about database system internals. (2) The more realistic practice is to let DBMS engine developers insert yield points in areas like table access methods (not shown in the figure), making yield decisions transparent to the application. However, this is highly impractical by requiring co-design of database engine kernels and the particular targeted workload. The DBMS engine developers must carefully profile the workload and engine via trial-and-error approaches to determine the correct



```
1.   rc_t Transaction(result) {
2.     rows = table1->Scan(range);
3.     // Manually yield here
4.     Sort(rows);
5.     for (row in rows) {
6.       key = row.attribute;
7.       val = table2->GetRecord(key);
8.       // Manually yield here
7.       result.Add(val);
8.     }
9.     ret_code = Commit();
10.    return ret_code;
11.  }
12.  // Wait until completion, do next one
```

Figure 3: Scheduling decision possibilities. Non-preemptive FIFO-based approaches (🟦) could only schedule new work at transaction commit boundaries. Cooperative scheduling (🟧) requires manually added yield points based on the target workload. Preemption-based scheduling (🟩) could allow scheduling new work on demand at almost any time.

yield points. If the workload changes, previous yield points that are already hardcoded in the engine code now become "wrong" and lead to lower performance.

Preemption Potential and Motivation. As shown in Figure 2, an ideal scheduling policy should be able to preempt long-running transactions, which then can easily meet the service level agreements (SLAs) of short, high-priority transactions. Like in Figure 3, preemption should be able to happen at any point, not just at predefined yield points, to promise low latency for high-priority transactions. Systems like MySQL and PostgreSQL do not natively implement preemption. PostgreSQL implements cooperative interrupts [41]. These are not true preemptive interrupts, but rather a mechanism that allows the database engine to handle interrupts gracefully by periodically checking for pending signals at safe points during execution, preventing corruption and ensuring graceful termination of queries and processes. In general, the evolution of preemption in database engines has been slow without the support of userspace interrupt mechanism. Even though preemption offers a promising path for managing mixed workloads, it has seen limited adoption and the conventional wisdom is to avoid it [6] as Section 1 mentioned. However, the availability of lightweight user interrupt mechanisms and recent advances in lightweight concurrency control protocols motivate us to revisit preemption in database systems, as we discuss next.

2.2 Memory-Optimized Database Engines

Traditionally, storage-centric DBMSs assume data is disk-resident and default to pessimistic concurrency control methods such as two-phase locking. As mentioned earlier, this can prohibit preemptive scheduling. However, advances in hardware—in particular large DRAM that can fit entire working sets and multicore CPUs—have allowed modern database engines to assume data to be memory-resident and become memory-optimized. Subsequently, they usually favor optimistic approaches to concurrency control methods which have much lower overhead compared to pessimistic approaches when data is in memory. In particular, many memory-optimized database engines [3, 4, 13, 24, 28, 30, 37, 51, 52] adopt multi-versioning and snapshot isolation which avoid readers and writers from blocking each other [3].

There are various designs available to realize optimistic and multi-versioned concurrency. Without losing generality, we describe ERMIA [24], a representative memory-optimized database engine and use it as as our baseline throughout the paper. Following the multi-versioned database model by Adya et al. [2], each record is represented by an ordered chain of versions, each of which is tagged with a global commit timestamp, which in turn denotes the commit time of the transaction that generate the version. To update a record, the transaction inserts a new version to the head of the record's version chain, forming a new-to-old order among all versions of the same record. Each transaction is also tagged with a begin and a commit timestamp, which is drawn from a centralized counter. To read a record, the transaction traverses the target record's version chain and reads desired version as dictated by the isolation level. For example, the widely used snapshot isolation (SI) [3] can be implemented by reading the first version that was generated before the transaction's commit time. Inserts and deletes can be modeled as special cases of updates. Read committed can be implemented by always reading the latest version, and serializability can be supported using certifiers [5, 51] or by combining with OCC [25]. Regardless of the isolation level used, however, during forward processing no pessimistic lock is taken for reads, which is our key assumption for preemptive scheduling to be viable throughout the paper.

2.3 User Interrupt

Userspace interrupts can enable direct delivery of interrupts to user space, bypassing the kernel. They recently became available in mainstream x86 processors [33]. Our measurement (Section 6.1) shows that user interrupt delivery latency between two POSIX threads is consistently lower than 1μs. This opens up new opportunities for database engines to implement preemptive scheduling purely in userspace, without all the kernel-crossing overhead of traditional software interrupts. To use user interrupt, the DBMS can register user interrupt handlers (similar to traditional interrupt handlers functionality-wise) that implement DBMS-specific interrupt handling logic. Different from traditional interrupts, however, user interrupt does not have advanced programmable interrupt controller [20] support in hardware for sending interrupts. Thus, a typical approach is to employ a dedicated scheduling thread which can use the `senduipi` instruction to send user interrupts. Once a user interrupt is delivered, the receiving thread immediately pauses the execution at the current instruction and jumps to the user interrupt handler. After returning from the handler, the thread resumes execution at the instruction where it was interrupted. Like in the traditional software-based interrupt, the CPU disables user interrupt so that the handler can execute to completion without interruption.

User interrupt handler respects the Linux x86-64 application binary interface (ABI). Customizing the interrupt handler to enable flexibility in designing scheduling policies, which will be detailed in Section 4, requires register preservation and restoration following the ABI. As shown in Figure 4, when a user interrupt is delivered, the receiving thread automatically skips the *red zone* [31] and pushes the user interrupt frame onto the stack. The red zone is a 128-byte area atop the stack top (RSP) that a function (as dictated by the compiler) can use for local data without adjusting the stack pointer, and it should remain valid until the function returns. Therefore, user interrupt handler should not use it. All the caller-saved registers and callee-saved registers must be preserved in the user interrupt (uintr) frame. More importantly, uintr frame will always bookkeep the instruction pointer `RIP`, the flags register `RFLAGS` and stack pointer `RSP` for the paused function (`foo()` in Figure 4). `RIP` points to the next instruction to be executed; `RFLAGS` contains the status of the CPU; `RSP` points to the top of the stack. The user interrupt handler returns via a specialized instruction `uiret`, which pops these registers in uintr frame's order to restore the original state of the program. Moreover, different from interrupts in the kernel space which are typically prohibited from involving complex computations such as floating point operations, userspace programs may well use the extended register sets for these operations. Therefore, a user interrupt handler also must store these registers, which can be done using the `xsave` and `xrstor` instructions [20]. We discuss how PreemptDB manages these register states and uses the aforementioned features and mechanisms in later sections.

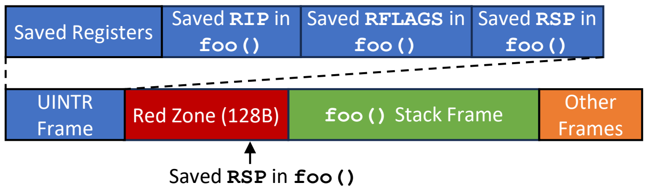


Figure 4: Stack layout during user interrupt handling.

3 DESIDERATA

While it is conceptually straightforward that preemption could lower transaction scheduling latency, we observe for it to be worthwhile, database engines like PreemptDB should provide the following desired properties:

- **Low Latency with High Throughput.** Much prior work has focused on improving transaction throughput. A preemptive scheduling based engine must preserve high throughput, while lowering scheduling latency via preemption.
- **Lightweight Transaction Switching.** The engine should serve preemption requests (i.e., switching between different transactions) quickly without incurring prohibitively high overheads. This is especially important for workloads where the preempted transactions are not very long, compared to scheduling delays.
- **Flexible Scheduling Policies.** Preemption can potentially enable various policies and—perhaps more critically—pitfalls (e.g., starvation as we discuss later) that were not possible. The mechanisms must be amenable to various policies and mitigation.

4 PREEMPTDB DESIGN

This section describes the design of PreemptDB, a new database engine that allows preemptive transaction scheduling via recent hardware-supported userspace interrupts. We first give an overview

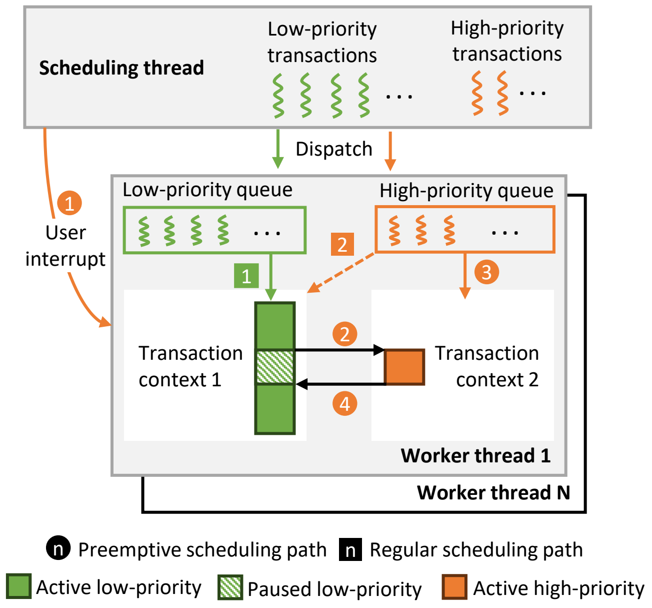


Figure 5: PreemptDB Overview. Upon arrival of a high-priority transaction T, a scheduling thread ❶ issues a user interrupt to preempt and pause the execution of an in-progress low-priority transaction S on a worker thread. The worker thread then ❷ switches to another context to ❸ accommodate T and ❹ resumes S after T finishes.

of PreemptDB, and then describe in detail how it satisfies the aforementioned desired properties.

4.1 Overview

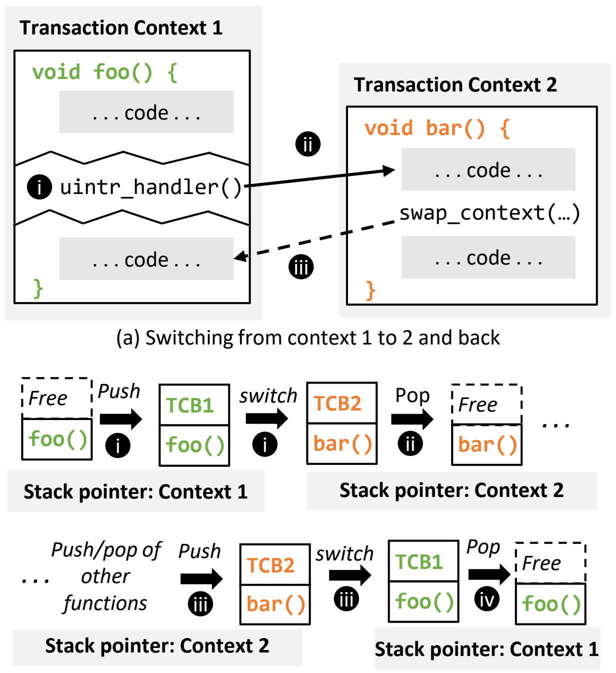
PreemptDB takes transaction requests tagged with priority levels (or service level objectives) as input and schedules them accordingly. Without losing generality, we limit our discussion to two priority levels: "normal" low-priority and more "urgent" high-priority transactions; we discuss how multiple priority levels could be supported later. As Figure 5 shows, PreemptDB employs two kinds of threads: (1) scheduling threads and (2) worker threads. A scheduling thread serves the dual-purpose of (1) dispatching transactions obtained from an admission control component (not shown in the figure) to worker threads for execution, and (2) issuing user interrupts to worker threads to trigger the preemption machinery to serve high-priority transactions. The number of scheduling threads is adjustable, however, our evaluation shows that using a single scheduling thread does not present a bottleneck at least with 32-core CPUs. For brevity, we assume one scheduling thread in the rest of the paper. Each worker thread maintains and continuously polls two scheduling queues (one for high-priority transactions, another for low-priority ones) where the scheduling thread dispatches transactions into. In addition, each worker thread maintains two transaction contexts (akin to OS thread contexts; detailed later) that support preemption caused by user interrupts.

With the above structures, PreemptDB allows two scheduling paths as shown in Figure 5. For normal, low-priority transactions, the worker threads simply follow the regular scheduling path (1) to execute low-priority transactions in a transaction context (e.g., context 1 in the figure) one after another. Without high-priority transactions, the system would stay in the regular scheduling path and execute transactions without interruption. When a high-priority transaction arrives, the scheduler thread first finds a worker thread whose high-priority queue still has at least one free slot (our current implementation does so in a round-robin manner) to accommodate the new request. After enqueuing one (or more as determined by the scheduling policy we discuss later) high-priority transaction(s), the scheduler can immediately issue a user interrupt (❶) to the corresponding worker to preempt (pause) the currently executing low-priority transaction. ❷ Upon receiving the user interrupt, the worker thread switches to context 2 and ❸ fetches a high-priority transaction (e.g., T or another earlier high-priority transaction) to work on. While T is being executed, new high-priority transactions could arrive and queue up. However, for simplicity our current design does not further interrupt an in-progress high-priority transaction, which we leave as future work. ❹ After one or more high-priority transactions executing in context 2 (as determined by the scheduling policy) concluded, the worker thread can switch back to the other context and resume the execution of the previously interrupted transaction. Additionally, a high-priority transaction could optionally get executed on the regular scheduling path (2) when the context becomes free. This can happen if a user interrupt (❶) is dropped due to DBMS-specific reasons (i.e., atomic active switch and critical section mechanisms to be discussed later), or a low-priority transaction happens to finish before the next user interrupt arrives. In other words, on the regular scheduling path, the worker thread may also be configured to prefer taking transactions from the high-priority queue based on the scheduling policy.

Our current implementation assumes memory-optimized environments where the main memory is large enough to hold the entire working set and each thread (scheduling and worker) is pinned to a core. However, PreemptDB's design principles and mechanisms are generally applicable. In the rest of this section, we discuss how PreemptDB realizes the above scheduling steps and tackles related challenges. We start from providing efficient mechanisms for transaction context switching next.

4.2 Lightweight Transaction Context Switch

PreemptDB needs to handle context switching in two directions: (1) A worker thread may be preempted to pause its currently in-progress transaction and switch to another context to perform another transaction with a higher priority. (2) After finishing processing the high-priority transaction, the worker thread should switch back to the previously interrupted transaction and continue from where it was interrupted. From the perspective of a worker thread, the first direction is passive, triggered by a user interrupt, while the latter is in fact voluntary. Both require storage and resumption of on-going transaction states. To facilitate this, we create a *transaction control block* (TCB) that saves transaction states, including register values, local variables, and so on. The notion of TCB is similar to process control blocks (PCBs) [45] in OS context switching, but is purely userspace. With TCBs, now we describe mechanisms in PreemptDB for handling each direction.



(a) Switching from context 1 to 2 and back

(b) Corresponding stack operations and status

Figure 6: Context switches within a thread in PreemptDB.

Uintr-based Passive Switch. As mentioned, a passive context switch is triggered by a user interrupt. It is then completed by a user interrupt handler (similar to traditional interrupt handlers, but in userspace). Figure 6(a) shows the process with an example of switching between two contexts on a worker thread. In the figure, `foo` is currently running in `Transaction context 1` before the thread received a user interrupt, sent by the scheduling thread (Section 4.1). The thread is then trapped in the user interrupt handler (`uintr_handler` in the figure) which in turn (i) saves the states of the current transaction (`foo`) into `TCB1` by pushing the states of the registers onto the stack. This process is depicted in Figure 6(b). (ii) Once the states of the preempted transaction are saved, the user interrupt handler continues to perform the actual "switch" by moving the current stack pointer to point to the stack top of the preemptive context (i.e., `Transaction context 2`). Recall that each transaction context is associated with its own stack, which is prepared with the necessary information in its TCB (`TCB2`) to begin a new incoming transaction. As Figure 6(b) shows, after the stack pointer is moved to point to `TCB2`, we pop its values into registers. At this point, `uintr_handler` has finished its work, and control is handed over to the high-priority context which can resume execution from the instruction pointer saved in `TCB2`, which could be starting to handle a high-priority transaction.

Interrupt handler code is often written in low-level assembly code as they are short and requires high performance. However, our use case—switching transaction contexts using user interrupts—necessitates extra work to be performed by the handler. In particular, it needs to (1) retrieve the destination stack pointer from the `rsp` register value saved in the preemptive `TCB2`, (2) save or

Algorithm 1 User interrupt handler routine.

```
interrupt_handler:
.check_rip: # check if rip is in swap_context
  cmpq # if rip > .swap_context_end, continue
  jg .continue_uintr
  cmpq # if rip > .swap_context_start, exit uintr
  jg .exit

.continue_uintr:
  push registers # save the states of registers
  call uintr_handler_helper # invoke C++ helper
  mov returned_rsp, %rsp # move the stack pointer
  pop registers # restore the states of registers
  uiret # return and re-enable user interrupts

.exit:
  uiret
```

restore various floating point and SIMD registers, and (3) perform necessary operations for correctness involving context-local storage and non-preemptible regions which we describe in detail later. Among these operations, (1) and (2) are typical and straightforward, yet (3) is a new requirement by user interrupts and transaction scheduling. As we will describe later in this section, it can become more involved and thus significantly complicate the handler logic, especially so in assembly code. Therefore, rather than following conventional wisdom to write interrupt handler code in assembly, we wrap complex work (3) in a C function which is invoked by `uintr_handler` between steps (i) and (ii).[1] The handler then needs to issue inlined assembly only to perform the stack pointer switch. As Section 6 shows, switching transaction contexts using our approach is very lightweight. It incurs little overhead while improving programmability and lowering maintenance efforts.

Atomic Active Switch. A worker thread may voluntarily switch context, e.g., after concluding a high-priority transaction so that it can resume the processing of the previously preempted low-priority transaction. We enable this by introducing a new userspace context switching interface `swap_context` which performs similar operations as (steps (iii) and (iv) in Figure 6) as the user interrupt handler. Between steps (iii) and (iv), `swap_context` can also invoke a helper function `swap_context_helper` to perform additional operations similar to the user interrupt handler helper function. However, the major difference between the aforementioned uintr-based passive switch is that active context switches are not atomic by default. Specifically, during a passive context switch, the thread is inside the user interrupt handler and cannot be interrupted, as guaranteed by the hardware [19]. Therefore, the states of registers will remain consistent during the passive context switch. However, an active switch does not have such protection: A preemption could happen at any time in an active switch, causing dirty states saved/restored silently and subsequently cause undefined behaviors.

To solve this problem, we design an atomic context switch mechanism (Algorithm 2) that leverages a combination of *temporary disabling of user interrupt delivery* and *instruction pointer check*.

[1]Details in `/uintr.{h,cc}` of our code repo.

Algorithm 2 Atomic active context switch routine.

```
 1 swap_handler_func:
 2 .swap_context_start:
 3   clui # temporarily disable user interrupts
 4   push registers
 5   call swap_context_helper
 6   mov returned_rsp, %rsp
 7   pop registers
 8   mov saved_rip, -0x80(%rsp) # bypass red zone
 9   stui # re-enable user interrupts
10   jmp -0x80(%rsp) # indirect jump to saved rip
11 .swap_context_end:
12   nop
```

`swap_context` is a regular function that should use `ret` instead of `uiret`, but `ret` expects a different stack layout than uintr frame, which is shown in Figure 4. If we call `ret` naïvely, we will prematurely set the instruction pointer without restoring the stored `RSP` pointing to the stack top of the paused context, which can lead to stack corruption and undefined behavior. Instead, we first restore `RSP`, and then perform an indirect jump to the saved `RIP`, which we copied to a temporary location at a fixed offset from the paused context's `RSP`. This fixed offset bypasses the 128-byte red zone of the stack to avoid corrupting the red zone. To guarantee that the entire context switch is atomic, an intuitive idea is to temporarily disable user interrupt delivery via `clui`. However, we still leave a small window for the user interrupt to happen between Algorithm 2, line 9 and line 10, because the indirect jump must be the last instruction before returning to the paused context. To solve this problem, we introduce *instruction pointer check* (Algorithm 1, lines 2–6) to force user interrupt handler to return early without performing any stack operations if the interrupted instruction pointer is between `.swap_context_start` and `.swap_context_end`.

4.3 Transparent Context-Local Storage Support

A database engine and its dependent runtime libraries (e.g., glibc, glog, boost) often utilize thread-local storage (TLS) to allow each thread to have its own independent copy of a variable, which is crucial for the correctness of concurrent execution of transactions. For instance, some systems [24] has a per-thread log buffer implemented as a thread-local variable, in which each worker thread may concurrently write its own redo logs. However, when we enable user interrupt in PreemptDB, each thread in the OS only owns its one copy of the TLS. Consequently, multiple contexts in the same worker thread will read and write to the same TLS variables. In this case, they may be overwriting each other's logs. We first note that this is not a unique problem to PreemptDB. For instance, coroutine-based cooperative scheduling systems [15, 16] can also voluntarily yield and maintain multiple transactions on the same thread. However, these systems easily sidestep the problem by storing TLS objects (e.g., per-thread log buffers, timestamps, etc.) as transaction-local objects, or in a preallocated array (one element per transaction). However, this approach does not work for PreemptDB which can preempt and pause the execution of the transaction logic almost anywhere, which includes not only the database engine code but also the dependent runtime libraries. If a context is interrupted within the runtime library code that uses TLS, we may introduce intra-thread data races that could lead to data corruption or crashes. While we can modify the database engine code to turn all existing TLS variables in transaction-local, it is impractical to do so for every existing runtime libraries which often need to be kept standard and are provided by the deployment environment (e.g., the OS).

To address this challenge, we design a *transparent* context-local storage (CLS) mechanism so that existing database engine code and dependent runtime libraries can continue to use it as regular TLS (e.g., as defined by the `thread_local` modifier in C++) without being modified or recompiled. To achieve this, we create a separate CLS area for each context with the same layout of the TLS area of the same program. In our current implementation, we create a redundant `pthread` [17] for each worker thread that never runs and effectively "steals" its TLS as the CLS of its second context.[2] When a context is actively running in a thread, its CLS area is exposed to the thread as its current TLS.[3] Note that no two contexts within the same thread can be executing at the same time, so the code running within a context will transparently read or write the TLS variables in its own CLS area. During a context switch, we automatically swap CLS areas of the two contexts, which we conveniently perform in the `uintr_handler_helper` function.

4.4 Non-Preemptible Regions

Next, we discuss another important mechanism for ensuring correctness: non-preemptible regions, which stop preemption from happening in enclosed code sections. The primary usage is to prevent deadlocks among database latches[4], for which the conventional approach is to take the latches in a predefined order (*consistent lock ordering*). For instance, when OCC in a non-preemptible database engine tries to validate the transactions, it has to latch the records in its read and write set in some consistent order (e.g., increasing address order), such that no wait-for cycles could form across concurrent transactions in *different threads*. However, in PreemptDB, concurrent transactions could be validating over overlapping read/write sets in different contexts of the *same thread*, with only one of them running. As a result, one of them could still be blocked on a latch if the other is preempted while holding a latch on a lower-address record, resulting in a deadlock. To prevent this from happening, we can wrap the OCC validation procedure into a non-preemptible region. Deadlocks could also arise from dependent libraries that use synchronization primitives internally (e.g., glibc `malloc()/free()`), which must also be wrapped into non-preemptible regions. There are many code locations in PreemptDB codebase that need to be wrapped into non-preemptible regions, including index APIs, memory allocator, transaction validation/-commit/abort logics. Since they are often intertwined with each other, simply enabling and disabling user interrupts with `stui/clui`

[2]We steal the initialized TLS as CLS from a redundant `pthread` because different OS kernels/toolchains may have different ways of setting up TLS even for the same x86-64 architecture. Doing so avoids replicating the code in the dynamic loader in Linux and makes our code more portable.

[3]On x86-64, a TLS is set by updating two special registers `fs` and `gs` to the thread-specific memory locations in an OS/toolchain/ABI-specific manner.

[4]Database latches are often implemented using common synchronization primitives such as spin locks, mutex, POSIX rwlocks, or lock-free primitives like CAS, which do not have built-in deadlock detection mechanisms.

does not work well. Instead, we support nested non-preemptible regions through enter (`TCB::lock()`) and exit (`TCB::unlock()`) APIs. To achieve this, we introduce a CLS lock counter to record how many times the current context has entered but not exited non-preemptible region. Thus, `TCB::lock()`/`TCB::unlock()` simply increments/decrements the CLS lock counter without any additional synchronization. When an interrupt arrives but we observe the lock counter is greater than zero, the user interrupt handler will return directly back to its current context, instead of switching to the other context, which is, again, conveniently implemented in `uintr_handler_helper` by directly returning the current `rsp` register value to the interrupt handler.

5 PREEMPTIVE SCHEDULING POLICIES

With all the mechanisms laid out, now we introduce two preemptive scheduling policies in PreemptDB, with the primary goal of lowering end-to-end latency of short high-priority transactions mixed with long low-priority transactions, namely (1) batched on-demand preemption policy, and (2) starvation prevention policy. We discuss support for other possible policies and workloads later.

Batched On-Demand Preemption. Long scheduling delay is the root cause of high end-to-end latency of short high-priority transactions, because they have to wait for long low-priority transactions to finish under non-preemptive policies. On the one hand, PreemptDB can leverage user interrupts to immediately schedule and execute high priority transactions on demand. On the other hand, excessively interrupting workers for every high priority transaction request will reduce the number of CPU cycles spent on useful works. Therefore, we devise a batched on-demand preemption policy that admits a batch of high-priority transactions up to a certain size at fixed intervals. We fix the size of the high-priority transaction queue of each worker to a tunable number. Once a batch is admitted, we select a worker in round-robin manner and push as many high-priority transactions as possible into the high-priority transaction queue until it is filled up or we deplete the available admitted transactions. Then, we send a single user interrupt to the worker for it to execute the batch immediately.

Starvation Prevention Policy. Under the batched on-demand preemption policy, the system may become overloaded when there is a constant stream of high priority transactions. This could starve low-priority transactions: too many CPU cycles could be spent on processing high-priority transactions, leading to significantly lower throughput and higher latency of the low-priority ones.

To solve this problem, we introduce a starvation prevention policy to work alongside the batched on-demand preemption policy. We monitor the ***starvation level*** L of low-priority transactions, defined as the percentage of cycles spent on high-priority transactions, and make sure the starvation level never exceeds a tunable ***starvation threshold*** $L_{\max}$ in scheduling decisions. More formally, when each low-priority transaction starts execution, we record a start timestamp T_0 (e.g., using `rdtscp` [20]) and reset an accumulator T_h to zero. Then, for each high priority transaction executed through preemption on the same worker, we accumulate its number of CPU cycles in T_h. For any time point T_1, we define the *starvation level* L of a low-priority transaction as $\mathbf{L = T_h/(T_1 - T_0)}$. Note that T_0, T_1 and T_h are all stored in a shared memory location across both

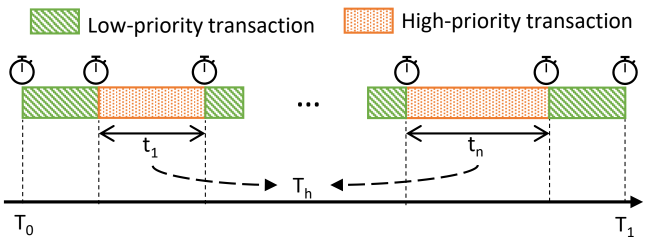


Figure 7: Starvation prevention in PreemptDB by monitoring starvation level. T_h is the accumulated cycles of high-priority transactions between the start (T_0) of the paused low-priority transaction in the same worker and current time (T_1).

contexts for each worker. We check the starvation level against the starvation threshold at two specific policy decision-making locations with different alternative decisions:

- Before a scheduler thread pushes high-priority transactions to the high-priority queue of a worker, we check whether L is above the threshold. If so, we do not push additional high-priority transactions into the worker and skip sending a user interrupt.
- After executing a high-priority transaction , we check whether L is above the threshold. If so, we prematurely switch back to the other context which holds a paused and starved low-priority transaction, without executing the remaining high-priority transactions in the queue.

Discussions. User interrupt opens up many opportunities. PreemptDB explores its potential of allowing timely scheduling of high-priority transactions when low-priority transactions have dominated all the system resources. We have so far only considered two priority levels (low and high) without hard service level agreement (SLA) guarantees. Thanks to PreemptDB's context switching design (instead of executing a high-priority transaction directly in the interrupt handler), one may easily extend PreemptDB to support more fine-grained priority levels by using multiple contexts/TCBs. A high-priority transaction that has already interrupted a previous lower-priority transaction could then be interrupted again, and the scheduler could choose to pause or even move the transaction to be performed elsewhere in another thread. More policies could also be developed, for example, to support real-time scheduling [1] with hard SLA guarantees. Other constraints, such as prioritizing user transactions over background tasks, and dynamic priority adjustment (e.g., increasing the priority for transactions that are already aborted beyond a threshold number of times) [54] could all be combined with PreemptDB. These are interesting future work.

Preemptive scheduling could also be implemented using traditional interrupts, if the entire DBMS were implemented in the kernel space. Some recent work has started initial exploration of this direction [27], but deploying them in today's cloud environment can be challenging. In contrast, PreemptDB is a pure userspace solution, allowing easy adoption in existing cloud environments.

6 EVALUATION

In this section, we perform experiments using various workloads to evaluate PreemptDB and show the following:

- PreemptDB can significantly reduce tail latency for scheduling high-priority transactions with low impact to the overall throughput under the same workload mix.
- PreemptDB's starvation prevention mechanism can effectively balance high- and low-priority transactions.
- Compared to cooperative scheduling, PreemptDB can provide low latency for high-priority transactions without harming low-priority transaction performance, yet is robust without requiring (often unrealistic) tuning.

6.1 Experimental Setup

We use a dual-socket server equipped with two 32-core Intel Xeon Gold 6448H CPUs and 512GB of main memory. The CPU is clocked at 2.4GHz (up to 4.1GHz with turbo boost) and has 60MB of caches. It runs Ubuntu 22.04.4 LTS with our patched uintr-enabled Linux kernel 6.2.0 based on Intel's uintr-linux-kernel repository [18]. We limit our experiments to a single socket without hyperthreading to focus on user interrupts. This allows us to avoid NUMA effect and ease the interpretation of results. One CPU core in our system is dedicated to delivering user interrupts. We place all the data in memory to stress the system without storage I/O.

Implementation. We implemented both PreemptDB and baselines (described later) on top of ERMIA [24] described in Section 2.2. Originally, ERMIA does not allow measuring scheduling latency as workload generation is piggybacked to worker threads: each worker thread repeats the cycle of generating a transaction, processing it, then moving on to the generation and processing of next transaction. To properly measure scheduling latency, we developed a benchmark driver that decouples workload generation and execution. Specifically, we dedicated the scheduling thread to generate transaction requests at fixed intervals (*arrival interval*). At each arrival interval, the scheduling thread first generates and pushes new low-priority transactions into the worker's low-priority transaction queue if it has empty slots. Then, the scheduling thread generates a batch of high-priority transactions with the same start timestamp; the batch size is set to the number of workers multiplied by the high priority transaction queue size. The scheduling thread then attempts to move these transactions into the lock-free high-priority transaction queues of the workers in a round-robin fashion until the the batch is depleted or the next arrival interval passes. For `PreemptDB`, the scheduling thread also sends a user interrupt to the worker immediately after pushing a batch of high-priority transactions to its queue. The engine worker threads process requests found on their queues according to their scheduling policies, which we describe later.

Adjusting queue sizes and arrival intervals will allow us to simulate different maximum scheduling delays and workload mixes. Unless other specified, we use 16 worker threads. For each worker, we set the low-priority transaction queue size to 1, and the high-priority transaction queue size to 4. The high-priority transaction request batch size is $16 \times 4 = 64$ by default. The default arrival interval is 1ms. We explore the impact of different number of workers, high-priority queue/batch sizes and arrival intervals later.

Benchmarks and Metrics. We use a mixed workload based on the TPC-C [46] and TPC-H [47] benchmarks to evaluate how PreemptDB and baselines perform. Specifically, we use Q2 from TPC-H as the long-running low-priority transaction, and NewOrder and Payment from TPC-C as the short-running high-priority transactions. This allows us to simulate scenarios where the system's resource is always dominated by long-running transactions (Q2), but short-running transactions (NewOrder and Payment) that require immediate attention for low scheduling latency may arrive following a certain arrival rate. We set the workload to use an equal number of warehouses as the number of threads, and follow the TPC-C specification to let each transaction to use a home warehouse with 15% of chance of using a remote warehouse. Like prior work [24], our benchmark code directly invokes the storage engine's C++ interfaces, without SQL parsing, networking and optimizer overheads to focus on scheduling mechanisms and policies. These create workloads with low logical contention, putting more pressure on the physical layer (including the scheduling subsystem), therefore allowing us to focus on scheduling overheads. Each experiment runs for 30 seconds and we report both throughput across varying numbers of threads and latency at the 50, 90, 99 and 99.9 percentiles.

Competing Methods. We test three scheduling methods below.

- `Wait`: Baseline scheduling policy with no preemption or cooperative scheduling.
- `Cooperative`: Cooperative scheduling which voluntarily yields at hard-coded locations in the storage engine code.
- `PreemptDB`: Preemptive scheduling using user interrupts to allow high-priority transactions to interrupt low-priority transactions.

For fair comparison, all policies are implemented in PreemptDB codebase. Under `Wait`, transactions are processed from beginning to end without preemptive or cooperative scheduling. Each worker thread starts with the low-priority transaction queue to run Q2. After concluding a transaction, it checks both queues and picks a high-priority transaction if it is present. As a result, high-priority transactions could be generated while Q2 is running and thus get delayed until the current Q2 is concluded. Moreover, `Wait` will attempt to exhaust all the high-priority transactions first, before executing the next Q2 from the low-priority queue.

To implement `Cooperative`, we instrumented `Wait` code so that the worker thread will yield and check the high-priority queue after having performed a predefined number of record read operations. Like `Wait`, the worker thread will process all the high-priority NewOrder and Payment transactions found in the queue without interruption before switching back to continue executing the original transaction (Q2). Such context switching is also implemented using pcontext and the `swap_context` primitive described in Section 4. Unless otherwise specified, we set `Cooperative` to yield after accessing every 10,000 records, denoted as the *yield interval*; we explore the impact of it later. Our implementation `Cooperative` represents a plausible cooperative scheduling settings where the system maintains a counter at the storage engine interfaces and yield regularly at a fixed interval.

`PreemptDB` follows the same benchmarking design, but allows the scheduling thread to issue user interrupts once a high-priority transaction is generated and uses the starvation mechanism to prevent starving Q2. Note that in most of the experiments, we test the system with a light mix of high-priority transactions, which do not impact the amount of CPU cycles on Q2 significantly. Therefore,

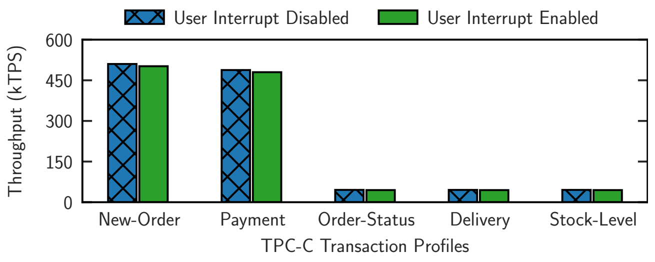


Figure 8: Standard TPC-C throughput with and without user interrupt mechanisms. The slowdown is minuscule, showing the low overhead of user interrupt and PreemptDB's preemptive scheduling mechanisms.

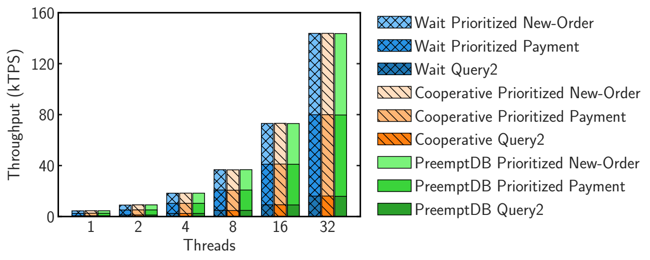


Figure 9: Scalability under different scheduling policies and the mixed workload. All variants scale well and `PreemptDB` maintains the high throughput compared to baselines.

we set the starvation threshold to 100 by default. In Section 6.4, we will showcase the effectiveness of starvation prevention using a workload settings that overload the system.

6.2 Overhead and Scalability of User Interrupts

The main goal of PreemptDB is to reduce the tail latency of high-priority transactions via preemptive scheduling enabled by user interrupts. Compared to classical solutions, using user interrupts to enable preemptive scheduling requires adding additional mechanisms to monitor incoming (high priority) requests and subsequently handle interrupts timely. For this to work well, it is important to ensure that (1) the scheduling mechanism itself is lightweight enough and (2) existing advantages such as throughput and scalability prior systems tried hard to achieve—often at the cost of tail latency—are (almost) unaffected.

We quantify the overhead of such additional mechanisms under short transactional workloads. In particular, we run the original TPC-C benchmark with all transactions begin sent as low-priority transactions. Meanwhile, the scheduling thread will still periodically wake up and interrupt worker threads without sending any high-priority transaction requests. This allows us to highlight user interrupts as pure overhead. As shown in Figure 8, the overhead is still minuscule with only $\sim 1.7\%$ reduction in throughput. The reason is that the user interrupts overload is extremely low and all workers will always immediately switch back to the main context due to not having any high-priority transaction request.

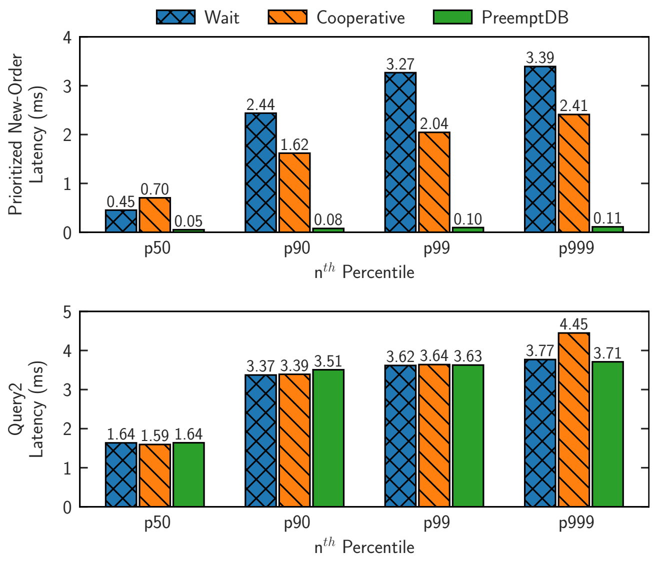


Figure 10: End-to-end latency of NewOrder (top) and Q2 (bottom). Without preemption, `Wait` exhibits up to ~30× higher tail latency than `PreemptDB` which both lowers NewOrder latency and maintains Q2 latency. `Cooperative` can lower tail latency but performs even worse than `Wait` at 50 percentile due to inaccurate tuning.

Under our actual target mixed workload, Figure 9 shows the throughput of the three types of transactions in the mix under varying core counts. As expected, all the variants scale well and maintain high throughput with expected transaction mix breakdown.

These results validate that overall, user interrupts are lightweight and `PreemptDB` does not trade off throughput for latency. Hence, we explore latency profiles of these variants in the rest of this section.

6.3 Effectiveness of Lightweight Preemption

Our first latency experiment compares the end-to-end latency of NewOrder, Payment and Q2 under different scheduling variants, following the experimental setup in Section 6.1. We obtained similar results for NewOrder and Payment showing the same trend, so we focus on NewOrder and Q2 here.

Figure 10(top) shows the latency of NewOrder under 16 threads and different scheduling variants. `PreemptDB` can significantly lower latency by 88–96% at different percentiles over `Wait`, as `Wait` has high queuing delay (consequently, high end-to-end latency) while `PreemptDB` can respond timely within a few microseconds to a high-priority NewOrder transaction. The second baseline, `Cooperative`, which periodically checks the high-priority queue for pending transactions while executing Q2, and yields if a high-priority transaction is present, presents even worse latency at 50 percentile (~55% and 1300% higher than `Wait` and `PreemptDB`). The reason is that the default yield interval (10,000) is too infrequent to allow a worker to start a high-priority transaction timely most of the time. It does, however, allows a reduction in tail latency at 90–99.9 percentile as it may occasionally yield shortly after a high priority transaction becomes available. Nevertheless, PreemptDB consistently has much lower latency than Cooperative at all percentiles.

For Q2 in Figure 10(bottom), PreemptDB maintains similar latency to that of Wait, thanks to the low overhead of user interrupts. In contrast, Cooperative presents higher 99.9 percentile latency for Q2 due to yield operations.

The above results highlight the effectiveness of preemption in PreemptDB without too much tuning requirements. In contrast, Cooperative must be tuned carefully and often in an unrealistic manner: a high yield interval can lead to high latency for high-priority transactions, but a low yield interval may introduce non-trivial overheads for low-priority transactions. Figure 11 quantifies this effect by presenting the throughput (top) and latency (bottom) under different yield intervals. In addition to Cooperative, we evaluated a handcrafted variant of it by carefully inspecting Q2 logic, profiling its performance, inserting yield operations at the "right" locations and tuning for a suitable yield interval. In this specific case, we inserted the yield right outside the nested query block of Q2 and only does so for every 1000 nested query block executed. As shown in the figure, Cooperative (Handcrafted) behaves comparably to PreemptDB under our target workload mix. Nevertheless, this would require DBMS developers to insert yield operations inside the system strategically such that the amount of CPU cycles between every two yields are roughly the same while not being too frequent. In other words, the DBMS must be customized specifically for a particular workload and target tail latency profile. Such strategic location of yields are usually impossible to find, especially in systems with user-defined functions, I/O, or skewed joins. Even if it exists as in our case, it is unrealistic to expect a database system developer to have such knowledge in advance. And thus, the handcrafted Cooperative policy is prohibitively expensive or even impossible to realize in practice.

To summarize, compared to Cooperative and Wait methods, PreemptDB can quickly respond to high-priority requests without negatively affecting the latency of low-priority transactions. It reliably delivers low scheduling latency for high-priority transactions without requiring sophisticated tuning, making it practical.

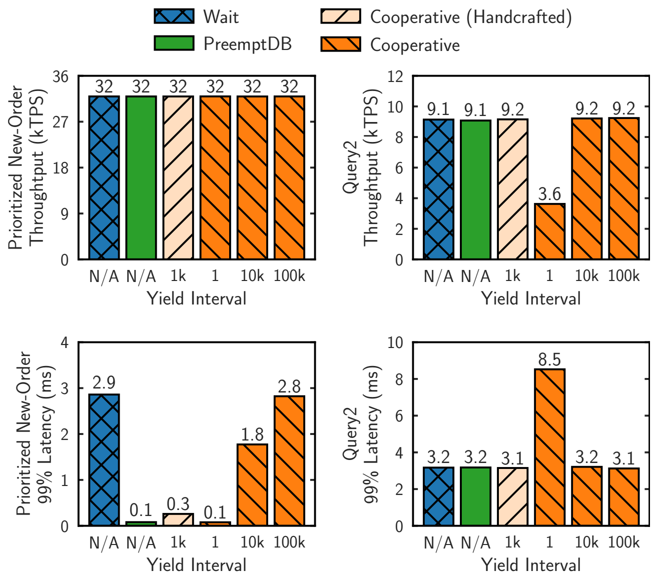


Figure 11: Yield interval vs. throughput (top) and latency (bottom). Yielding too frequently (e.g., once per record access) can improve high-priority transactions (left), but negatively impact low-priority Q2 (right). Handcrafting Cooperative could give desirable behaviors similar to PreemptDB's for both types of transactions but has to be handcrafted for a specific workload, which is unrealistic and not always possible.

6.4 Effectiveness of Starvation Prevention

As previously discussed, the workers may constantly execute high-priority transactions, starving low priority transactions, if we allow PreemptDB to preempt low-priority transactions without restrictions. To test such scenario, we increase the high-priority transaction queue size to 100 and we send 1600 high-priority transactions across all 16 workers every 1 millisecond. We then compare PreemptDB with different starvation threshold and the Wait policy. As shown in Figure 12, the Wait policy suffers from starvation of Q2: its throughput drops from about 9.14 kTPS (Figure 9 where we only send 64 high-priority transactions per 1 millisecond) down to only 0.42 kTPS, with the latency increased from 3.62 ms to 560 ms at 99 percentile. Similarly, when the starvation threshold of PreemptDB is set to 100, which effectively disables the starvation prevention mechanism, PreemptDB also witnesses similar throughput decrease and latency increase of Q2. By lowering the starvation threshold, we can limit the percentage of CPU cycles spent on the high priority transactions and thus cause less starvation of Q2 at the cost of lower throughput and higher latency of high-priority transactions. For instance, we can achieve a more balanced performance with a starvation threshold of 0.75, where the high-priority transaction NewOrder's latency at 99 percentile is 2.59*ms* while Q2's throughput is maintained at 1.49 kTPS. On the other extreme, if we set the starvation threshold to 0, which prevents preemptive context to execute prioritized transactions, PreemptDB can achieve the highest possible throughput of 9.33 kTPS for Q2 at the cost of significantly increased tail latency for NewOrder. We also want to note that, under such a high system load, the p50 latency of PreemptDB and Wait with starvation threshold 1 are similar, indicating a very low overhead of our preemption mechanisms regardless of system load. Usually, the user needs to adjust the starvation threshold based on their need of the transaction latency and throughput profiles, and we leave the automatic tuning of this threshold for future work.

6.5 Robustness under Varying Arrival Intervals

Our experiments so far have used a fixed arrival interval of 1*ms* for high-priority transactions. We further stress test the system with varying arrival intervals to show PreemptDB can reduce scheduling latency under varying load of high-priority transactions. Figure 13 shows the geometric mean of end-to-end latency of the low-priority Q2 and the high-priority NewOrder under arrival rates of 50μs to 50*ms*. Corroborating with the results from previous experiments, Q2 latency under Wait, Cooperative and PreemptDB remain similar across all arrival intervals. As arrival interval decreases (i.e., the

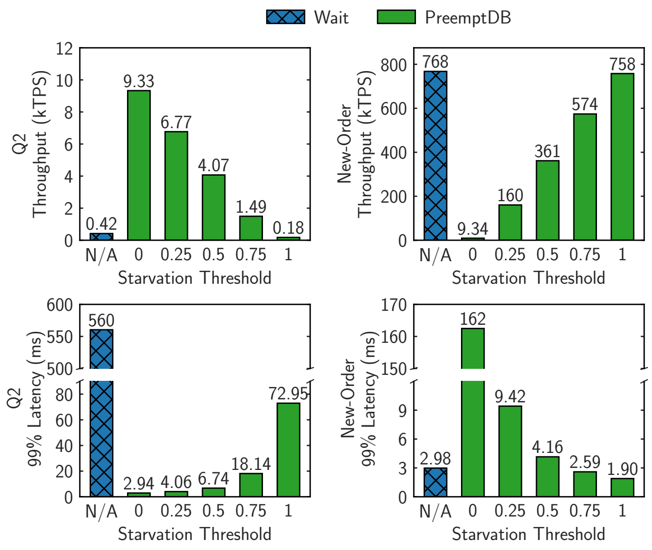


Figure 12: Throughput and 99 percentile latency of New-Order and Q2 at different starvation thresholds.

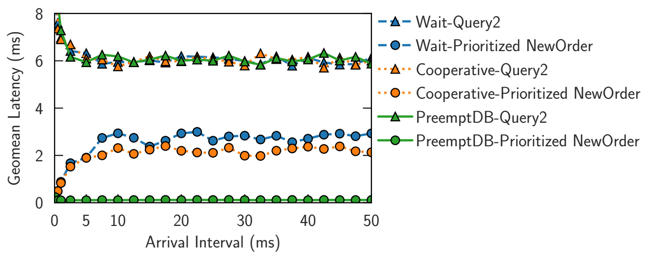


Figure 13: Geomean latency with varying arrival intervals. Higher intervals indicate the system is less overloaded by NewOrder, leading to lower Q2 latency. Compared to baselines, `PreemptDB` is robust and exhibits low latency for high-priority NewOrder regardless of arrival rates.

system becomes more loaded with high-priority transactions), Q2 latency increases as the system spends fewer CPU cycles on it.

However, the latency of high-priority transactions differs significantly between `PreemptDB` and the two baselines. Under larger arrival intervals (lighter high-priority transaction load), `Wait` and `Cooperative` have 24.7× and 18.7× higher latency for the high-priority transactions compared to `PreemptDB`. The gap becomes smaller with smaller arrival intervals (heavier high-priority transaction load). The reason is that both `Wait` and `Cooperative` will exhaust the high-priority queue when it finishes or yield from a transaction, so they can benefit from having a constant stream of high-priority short-running transactions to process them more timely. Nevertheless, `Wait` and `Cooperative` both still have about 3.8× higher latency than `PreemptDB`, even in the most extreme case with the arrival interval of 50μs. These results show that `PreemptDB` is robust under varying arrival intervals and transaction load.

7 RELATED WORK

Our work is most related to non-preemptive and cooperative scheduling in DBMSs, and scheduling issues in operating systems. Note that we differentiate from concurrency control protocols (which are also often referred to as "scheduling" in database literature).

Non-Preemptive DBMS Scheduling in DBMSs. Since preemption was previously deemed unsuitable [6] and could lead to starvation issues for low-priority transactions [32], most work in this category attempts to optimize the order of issuing transactions based on metrics such as priority. PreemptDB mitigates these issues by leveraging recent user interrupts and optimistic concurrency. SMF [9] greedy selects the next transaction to run based on the one that would lead to the smallest increase in overall execution time. Wagner et al. [49] split each query into morsels, and use a self-tuning stride scheduler to adaptively allocate resources to optimize tail latency. Psaroudakis et al. [43] schedule tasks by making it NUMA-aware to enhance memory locality. It focuses more on improving throughput rather than reducing latency. Polaris [54] augments OCC with a lightweight reservation mechanism to optimize high priority transactions. Compared to preemption-based PreemptDB, a common issue in these approaches is that they cannot respond to incoming high-priority transactions promptly. Nevertheless, some of them could be potentially combined with PreemptDB, e.g., to further optimize data placement in NUMA environment, which we leave as future work.

Cooperative DBMS Scheduling. Much recent work has devised cooperative scheduling to hide latency incurred by memory accesses. MxTasks [35] distributes tasks to task pool based on the data dependencies and serialize tasks in the task pool to reduce contention. CoroBase [15] and MosaicDB [16] use coroutines [21] to implement software prefetching and asynchronous I/O with hard-coded yield points in database engine. Contrary to PreemptDB's goal which is to reduce transaction latency, these proposals are more like to trade latency for throughput by aggressive batching to saturate memory or SSD bandwidth. It is interesting future work to add preemption to such systems for the benefits of both latency hiding and low scheduling latency.

OS Scheduling. The systems community has proposed multiple approaches to lower tail latency via new scheduling mechanisms and policies. Caladan [14] uses a dedicated scheduling core and a custom kernel module to quickly respond to resource contention, such as memory bandwidth or cache pressure. PreemptDB can also implement such features using a monitoring thread to collect and act on the necessary information using user interrupts. Shinjuku [22] leverages hardware virtualization features to allow very fast preemption and prioritization. LibPreemptible[29] is a preemption framework also based on user interrupt. However, like other work in this category, it is designed for general applications, whereas PreemptDB targets database systems and is designed with mechanisms and policies specific to database use cases.

8 CONCLUSION

Modern database applications increasingly mix short, high-priority transactions with long, low-priority analytics. Traditional scheduling approaches such as FIFO-based and cooperative methods fall short on these workloads by inducing long scheduling latency for high-priority transactions. The former allows analytical transactions to monopolize CPU cycles, while the latter can be very hard to tune and maintain. The potential of preemptive scheduling has been left unexplored due to the high cost of software interrupt delivery and assumptions of pessimistic concurrency prevalent in traditional storage-centric systems. However, neither is still the case today with new userspace interrupt primitives in modern x86 CPUs and the wide adoption of optimistic concurrency. Based on these observations, we propose PreemptDB, a database engine that allows efficient pure-userspace preemption for mixed workloads. PreemptDB proposes an efficient transaction context switching mechanism and scheduling policies based on preemption. Compared to prior approaches, PreemptDB is robust against workload patterns, and reduces scheduling latency and end-to-end tail latency by up to 96%, while maintaining high overall throughput.

REFERENCES

[1] Robert K. Abbott and Hector Garcia-Molina. 1992. Scheduling real-time transactions: a performance evaluation. *ACM Trans. Database Syst.* 17, 3 (Sept. 1992), 513–560. https://doi.org/10.1145/132271.132276

[2] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*. 67–78. https://doi.org/10.1109/ICDE.2000.839388

[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[4] Michael J Cahill, Uwe Röhm, and Alan D Fekete. 2009. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–42.

[5] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (Dec. 2009), 42 pages.

[6] M. J. Carey, R. Jauhari, and M. Livny. 1989. Priority in DBMS resource scheduling. In *Proceedings of the 15th International Conference on Very Large Data Bases* (Amsterdam, The Netherlands) *(VLDB '89)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 397–410.

[7] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. 2011. An Overview of Business Intelligence Technology. *Commun. ACM* 54, 8 (aug 2011), 88–98. https://doi.org/10.1145/1978542.1978562

[8] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3411–3424.

[9] Audrey Cheng, Aaron Kabcenell, Jason Chan, Xiao Shi, Peter Bailis, Natacha Crooks, and Ion Stoica. 2024. Towards Optimal Transaction Scheduling. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 2694–2707. https://doi.org/10.14778/3681954.3681956

[10] Data Sleek. 2023. Unlocking the Potential and Power of HTAP Databases. https://data-sleek.com/htap-databases/ Accessed: 2024-10-17.

[11] K. Delaney and C. Freeman. 2013. *Microsoft SQL Server 2012 Internals.* Pearson Education. https://books.google.ca/books?id=wK1CAwAAQBAJ

[12] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. 2019. A morsel-driven query execution engine for heterogeneous multicores. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2218–2229.

[13] Jose M Faleiro and Daniel J Abadi. 2014. Rethinking serializable multiversion concurrency control. *arXiv preprint arXiv:1412.2324* (2014).

[14] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 16, 17 pages.

[15] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: coroutine-oriented main-memory database engine. *arXiv preprint arXiv:2010.15981* (2020).

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

[17] IEEE and The Open Group. 2016. The Open Group Base Specifications Issue 7, IEEE Std 1003.1. (2016).

[18] Intel. 2022. User Interrupt Enabled Linux Kernel. Retrieved 2024 from https://github.com/intel/uintr-linux-kernel

[19] Intel. 2024. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Chapter 7.4.2. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Accessed: 2024-10-17.

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

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

[22] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[23] 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.

[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*. 1675–1687.

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

[26] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

[27] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *Proc. VLDB Endow.* 17, 8 (May 2024), 2115–2122. https://doi.org/10.14778/3659437.3659462

[28] 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)*.

[29] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. 2024. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–936.

[30] 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*. 21–35.

[31] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2021. System V Application Binary Interface: AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models).

[32] D.T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. 2005. Improving preemptive prioritization via statistical characterization of OLTP locking. In *21st International Conference on Data Engineering (ICDE'05)*. 446–457. https://doi.org/10.1109/ICDE.2005.78

[33] Sohil Mehta. 2021. User Interrupts – A faster way to signal. https://lpc.events/event/11/contributions/985/attachments/756/1417/User_Interrupts_LPC_2021.pdf

[34] Henrik Mühe, Alfons Kemper, and Thomas Neumann. 2013. Executing Long-Running Transactions in Synchronization-Free Main Memory Database Systems.. In *CIDR*.

[35] Jan Mühlig and Jens Teubner. 2021. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1331–1344. https://doi.org/10.1145/3448016.3457268

[36] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[37] 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*. 677–689.

[38] Oracle. 2024. HeatWave MySQL Database Service. https://www.oracle.com/mysql/heatwave/ Accessed: 2024-10-17.

[39] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. *Gartner Research* (2014).

[40] 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*. 1493–1508.

[41] PostgreSQL. 2024. PostgreSQL Signals. https://wiki.postgresql.org/wiki/Signals. Accessed: 2024-10-17.

[42] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, et al. 2022. Cloud-native transactions and analytics in singlestore. In *Proceedings of the 2022 International Conference on Management of Data*. 2340–2352.

[43] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling

for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.* 10, 2 (Oct. 2016), 37–48. https://doi.org/10.14778/3015274.3015275

[44] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3290–3303.

[45] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2012. *Operating System Concepts* (9th ed.). Wiley Publishing.

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

[47] TPC. 2011. TPC Benchmark H (Decision Support) Standard Specification, revision 2.14.3. http://www.tpc.org/tpch

[48] 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.

[49] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1879–1891. https://doi.org/10.1145/3448016.3457260

[50] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. 2023. Polardb-imci: A cloud-native htap database system at alibaba. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[51] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal* 26 (2017), 537–562.

[52] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[53] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, et al. 2020. F1 Lightning: HTAP as a Service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3313–3325.

[54] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling Transaction Priority in Optimistic Concurrency Control. *Proc. ACM Manag. Data* 1, 1, Article 44 (May 2023), 24 pages. https://doi.org/10.1145/3588724