Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads

Raja Appuswamy* raja.appuswamy@epfl.ch
Angelos C. Anadiotis* angelos.anadiotis@epfl.ch
Danica Porobic‡* danica.porobic@oracle.com
Mustafa K. Iman* mustafa.iman@epfl.ch
Anastasia Ailamaki*† anastasia.ailamaki@epfl.ch

*École Polytechnique Fédérale de Lausanne †RAW Labs SA ‡Oracle

ABSTRACT

Main-memory OLTP engines are being increasingly deployed on multicore servers that provide abundant thread-level parallelism. However, recent research has shown that even the state-of-the-art OLTP engines are unable to exploit available parallelism for high contention workloads. While previous studies have shown the lack of scalability of all popular concurrency control protocols, they consider only one system architecture—a non-partitioned, shared everything one where transactions can be scheduled to run on any core and can access any data or metadata stored in shared memory.

In this paper, we perform a thorough analysis of the impact of other architectural alternatives (Data-oriented transaction execution, Partitioned Serial Execution, and Delegation) on scalability under high contention scenarios. In doing so, we present Trireme, a main-memory OLTP engine testbed that implements four system architectures and several popular concurrency control protocols in a single code base. Using Trireme, we present an extensive experimental study to understand i) the impact of each system architecture on overall scalability, ii) the interaction between system architecture and concurrency control protocols, and iii) the pros and cons of new architectures that have been proposed recently to explicitly deal with high-contention workloads.

PVLDB Reference Format:
Raja Appuswamy, Angelos C. Anadiotis, Danica Porobic, Mustafa K. Iman, Anastasia Ailamaki. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads. *PVLDB*, 11(2): 121 - 134, 2017.
DOI: 10.14778/3149193.3149194

1. INTRODUCTION

Modern OLTP engines are deployed on the state-of-the-art servers with hundreds of processing cores and Terabytes of memory. In order to exploit the abundant thread-level parallelism present in modern multi-core servers, OLTP engines should execute multiple transactions concurrently so that cores can make progress in parallel instead of waiting for one another. In *low-contention* workloads, transactions access different records, and hence do not conflict with each other. In such cases, modern main-memory OLTP engines can effectively scale with increasing number of cores.

However, recent research has shown that even the state-of-the-art main-memory OLTP engines suffer from scalability limitations when faced with *high contention* [47]. Under these workloads, several transactions simultaneously perform conflicting operations on a few popular records. As these conflicting operations cannot be executed concurrently while preserving serializability, OLTP engines cannot execute the corresponding conflicting transactions in parallel on different cores. Thus, it is impossible to achieve linear scalability under high-contention workloads. In the ideal case, adding more cores to the system results in throughput scaling up to a threshold permitted by the workload and plateauing beyond that. However, the observed scalability of main-memory OLTP engines is far from the ideal. In fact, adding more cores in such a scenario results in a throughput drop proportional to the level of contention.

Recent studies have shown that synchronization overheads imposed by concurrency control protocols are responsible for this lack of scalability [36, 43, 47]. However, these studies consider only a single system architecture—a non-partitioned, shared-everything one where any transaction can be scheduled to run on any core and can access any data or metadata stored in shared memory. However, there are several alternative architectures that have been proposed in prior research. For instance, the *Data oriented transaction execution (DORA)* [31] architecture eschews thread-to-transaction assignment used by the conventional shared-everything architecture in favor thread-to-data assignment by logically partitioning the data across threads. Transactions flow from one thread to another as they access different data. *Partitioned Serial Execution* [23] is another architecture that physically partitions the data across cores such that each core owns a portion of data and serially executes transactions one at a time by using whole-partition locking instead of fine-grained record-level concurrency control. *Delegation* [2, 36] architecture treats a multi-core server like a distributed system. Data and metadata are physically partitioned across threads. Each thread uses fine-grained concurrency control for coordinating transactions and threads communicate explicitly using message passing despite the existence of shared memory.

While the prior studies have demonstrated the lack of scalability of concurrency control protocols in the conventional shared-everything architecture, there has been no systematic analysis comparing the scalability of other architectures under high-contention workloads. Such an analysis is important due to three reasons. First, some architectures, such as DORA, were proposed in the context of disk-based OLTP engines where high disk-access latency was the dominating source of overhead. Thus, an analysis is re-

*Work done while at EPFL

Proceedings of the VLDB Endowment, Vol. 11, No. 2

DOI: 10.14778/3149193.3149194

Table 1: Types and properties of architectures and example systems

Architecture	PM		TxSM		Example systems
	Parallelism	Thread assignment	Logical synch.	Physical synch.	
SE	Shared memory	thread-to-txn	CC protocols	latch/atomics	Silo [42],Hekaton [11]
PSE	Shared nothing	thread-to-txn	partition lock	partition lock	H-store [23],Hyper [24]
Delegation	Message passing	thread-to-txn	CC protocols	message passing	Caldera [2],Orthrus [36]
Data-oriented	Shared memory	thread-to-data	CC protocols	txn migration	DORA,PLP [31, 32]

quired to determine the applicability of these architectures in the context of main-memory OLTP engines.

Second, no study till date has investigated the interaction between concurrency control protocols and system architectures. In theory, the choice of concurrency control protocol should be independent of the choice of system architecture. However, in practice, we show that this is not the case as the architecture can play a crucial role in determining the effectiveness of a concurrency control protocol under high contention.

Third, researchers have recently proposed new architectures that mix existing synchronization techniques in innovative ways to tackle various problems. For instance, Caldera [2] uses message passing to build OLTP engines on non-cache-coherent hardware. Thread migration is being used to build virtualization-friendly OLTP engines [12]. Orthrus [36] combines message passing with functional partitioning and transaction pre-execution to improve scalability under high-contention workloads. However, it is unclear how these new architectures compare to the state-of-the-art, or what is the impact of each individual technique on scalability. Thus, an in-depth analysis is required to answer these questions.

In this paper, we present the first analysis of the impact of system architecture on scalability of OLTP engines under high-contention workloads. In doing so, we make the following contributions:

- We present the design and implementation of Trireme, a main-memory OLTP engine testbed that implements four system architectures (traditional shared everything, DORA, PSE, Delegation), and several concurrency control protocols in a single code base. In doing so, we identify scalability challenges in implementing each architecture in a multi-core, main memory setting and show how to overcome those challenges.

- We conduct a comprehensive analysis to i) identify the pros and cons of each individual architecture in isolation, and ii) characterize the interaction between system architectures and concurrency control protocols.

- We perform a comparison with other OLTP engine testbeds (DBX1000 [47]) to corroborate our results, and a comparison with the state-of-the-art solutions targeted at high contention (MOCC [43], Orthrus [36], VLL [37]) to tease apart the contribution of each design aspect to overall scalability.

2. BACKGROUND

All modern main-memory OLTP engines are multi-threaded applications that are capable of executing many transactions concurrently using thread-level parallelism provided by multi-cores. However, they can differ significantly in their *system architecture*. Typically, the system architecture of a DBMS describes the functioning and interaction between five major components [14], namely, the Client Communications Manager (CCM), the Process Manager (PM), the Relational Query Processor (QP), the Transactional Storage Manager (TxSM), and other shared components and utilities.

In this work, we focus on OLTP workloads and use the term system architecture to refer to just the PM and the TxSM. The PM is responsible for implementing the process model which dictates how concurrent transactions are scheduled and executed. Traditionally, database systems used the notion of *Multi-programming Levels* (MPLs) to explicitly control the maximum allowable number of concurrent requests. Database engines implement MPLs by mapping each request to a *DBMS worker* which keeps track of the execution context for a given request. Workers are then mapped to *operating system threads*. Finally, these threads are scheduled to run by the operating system on *processing cores* [14]. In contrast, most modern main-memory OLTP engines use a one-to-one correspondence between the number of DBMS workers, operating system threads, and processing cores [11, 23, 42]. Thus, in modern main-memory OLTP engines, the process model implemented by PM dictates two architectural aspects: i) the type of parallelism used, ii) the mapping of transactions to threads.

The TxSM in traditional disk-based OLTP engines is typically a collection of four modules [14], the *concurrency control* module that provides Atomicity and Isolation, the *log management* module that provides Durability, the *buffer pool* module that is responsible for staging I/O to and from storage devices, and the *access* module which is responsible for organizing data on storage. As our work focuses on studying the interaction between process models and concurrency control protocols in the main-memory OLTP context, similarly to recent studies [36, 45, 47], we restrict our focus to the concurrency control module to avoid confounding effects.

In the rest of this section, we present an overview of four popular system architectures that have been used for building OLTP engines. We highlight how they differ in their choice of PM and TxM. Table 1 provides a summary of various architectures together with the examples of database systems that use each architecture. Figure 1 shows a pictorial depiction of these architectures.

2.1 Shared Everything

PM. OLTP engines based on the non-partitioned, shared everything architecture (SE) use the *shared-memory* model of parallelism where they store all data in shared memory that is globally accessible to threads. Most SE engines use a *thread-to-transaction* assignment with each transaction being assigned to a database thread. Once assigned, a thread is responsible for executing the entire transaction and all supporting operations.

TxSM. As two threads can concurrently run transactions that perform conflicting operations on the same record, SE engines need synchronization at two levels: logical and physical. At the *logical level*, *concurrency control (CC) protocols* ensure that conflicting transactions are ordered to guarantee serializability. For instance, pessimistic protocols order transactions by making one transaction wait for another using locks, while optimistic protocols use aborts and retries [4]. In order to enforce such ordering, *all* CC protocols, irrespective of their type, associate additional metadata with databases' logical entities, which are typically records in main-memory OLTP engines. For instance, pessimistic locking protocols might maintain lists of transactions to track lock owners and waiters, while optimistic protocols might maintain per-record timestamp values to decide whether a given transaction must be committed or aborted. As multiple threads can simultaneously update this metadata, CC protocols also synchronize at the *physical level* using *latches* or *atomic instructions* to maintain metadata consistency.

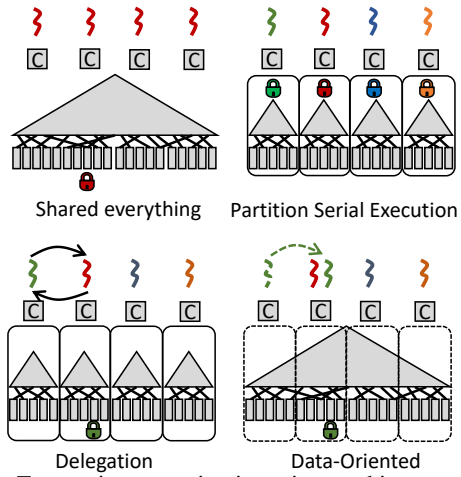

Figure 1: Transaction execution in various architectures. The SE shows data and indices stored in shared memory that is globally accessible to all threads. The PSE shows data and indices partitioned across threads, with each thread using a partition lock. The Delegation shows data partitioned similar to PSE with explicit messaging between threads and fine-grained locking. Finally, the DORA shows logical data partitioning (dotted lines), transaction migration, and fine-grained locking.

2.2 Partitioned Serial Execution

PM. Partitioned Serial Execution(PSE)-based engines *physically partition* data and all associated metadata in such a way that each database thread has exclusive access to its partition. Thus, they adopt the *shared nothing* model of parallelism. Traditionally, shared nothing has been used to refer to a parallel system that is composed of a cluster of independent machines [14]. However, when PSE engines are deployed on a single, scale-up multi-core server [23], all data is stored in shared memory. Despite this, threads in PSE engine do not share any data or metadata. PSE engines use a thread-to-transaction assignment and each thread executes one transaction at a time. However, transactions are assigned to threads in such a way that most transactions are *single-site* in nature, meaning that data accessed by a transaction is local to the thread on which it is scheduled to execute.

TxSM. In the ideal case, all transactions are single-sited. In such a case, PSE guarantees near-linear scalability as each thread executes transactions independently without synchronizing with any other thread. However, if transactions are *multi-sited*, meaning that data accessed by a transaction is stored in more than one partition, it is necessary to coordinate transaction execution across several threads to guarantee serializability. PSE engines ensure that only one transaction can run at any given time on a given set of partitions by locking all relevant partitions before accessing data. Thus, in contrast to SE engines that use fine-grained record-level CC protocols, PSE engines use coarse-grained partition-level locking.

2.3 Delegation

PM. The Delegation architecture treats a multi-socket server as a distributed system where each core represents a node in a cluster. Data is partitioned across cores and a thread-to-transaction assignment is used to execute transactions. Similarly to PSE, Delegation distinguishes between single-site and multi-site transactions. For single-site transactions, Delegation works like PSE; each thread works on its own partition without synchronizing with other threads. However, Delegation differs from PSE in the way multi-site transactions are handled. When a transaction running on one thread wants to perform an operation on a record owned by a different thread, it explicitly sends a message requesting the target thread for the data item even though the data resides in globally accessible shared memory. This is similar to the way nodes in a distributed system communicate via message passing. Thus, Delegation uses *message passing*-based parallelism.

TxSM. As transactions may request data in a random order from any remote thread, Delegation still needs a mechanism for logically synchronizing transactions to enforce serializability. It does so by using fine-grained, record-level CC. Thus, in addition to data and indices, each thread also maintains the additional metadata needed by CC protocols. This metadata is exclusively accessed by the owner thread, and thus, requires no physical synchronization.

2.4 Data-Oriented Architecture

PM. The Data-Oriented Architecture (DORA) was designed in the context of disk-based OLTP engines that used latches for physically synchronizing access to centralized data structures. As multiple threads attempt to update the shared data structure, the ensuing contention on latches resulted in poor scalability on multi-core servers. DORA solves this problem by replacing the thread-to-transaction mapping with a *thread-to-data* mapping. With this approach, each DBMS thread is associated with a disjoint subset of the database. Transactions then flow from thread to thread depending on the data being accessed. It is important to note that the thread-to-data mapping used by DORA is a *logical* partitioning of data as it only affinitizes data to threads instead of physically partitioning data. Thus, changing affinitization is easy as it requires only updating a record-to-thread mapping table instead of physically rearranging records.

TxM. Similar to SE, DORA provides serializability by using fine-grained, record-level CC. However, unlike SE, each thread maintains CC metadata only for the data it manages and has exclusive access to this metadata. Thus, DORA logically partitions data and physically partitions metadata. As a result, DORA does not require physical synchronization to protect CC metadata.

3. TRIREME OLTP ENGINE TESTBED

In this section, we describe Trireme, our prototype OLTP engine that supports all four architectures in a single code base. Trireme is implemented as a multi-threaded application that runs as a single process, with each thread being bound to a different processor core. All data is stored entirely in memory in a row-major format. To avoid memory management delays in the critical path that arise due to the use of `malloc`, we implemented a custom memory allocator that explicitly manages one heap area per thread and meets all memory allocation requests using the thread-local heap. The system also supports a simple primary-key-based hash index. We use a one-latch-per-bucket synchronization strategy for coordinating updates to the hashtable. As our workloads only read or update existing records instead of adding or removing records, index synchronization is not a scalability bottleneck in any of the architectures in our experiments. We leave investigating scalable index synchronization techniques for various architectures to future work.

3.1 CC protocols

In this study, we consider three classes of CC protocols, namely, pessimistic deadlock detection, pessimistic deadlock avoidance, and optimistic. More specifically, we consider `DL_DETECT`, `NO_WAIT`, and `SILO` as representatives from each category. We chose these three protocols as they have shown to be the best in their respective classes by recent studies [47]. In Section 5, we also validate this claim by presenting results that we obtained by running our experiments on several other CC protocols using DBX1000, another prototype OLTP engine that has been used in recent research [47].

We only consider the SERIALIZABLE isolation level and single-versioned storage. Thus, we limit our focus to single-version variants of DL_DETECT, NO_WAIT, and SILO protocols. While we do not consider CC protocols based on multi-versioned storage as they add additional dimensions, like garbage collection, we do consider the two-version variant of the NO_WAIT protocol (2V_NO_WAIT [4]) that exploits undo logging performed by single-version storage to improve concurrency between readers and writers. A recent study has explored other dimensions in implementing MVCC protocols [45]. In the rest of this section, we provide a high-level overview of the CC protocols supported by Trireme.

DL_DETECT is a two-phase locking (2PL) protocol that maintains a *waits-for* graph of transactions, uses it to determine deadlocks (based on cycles in the graph), and aborts transactions to relieve deadlocks. The NO_WAIT 2PL protocol, in contrast, completely avoids deadlocks by aborting transactions that attempt to acquire any lock which is already held in a conflicting lock mode.

2V_NO_WAIT introduces a certify lock in addition to read and write locks used by NO_WAIT. Transactions acquire read/write locks before performing an operation. Write locks conflict with each other. Read locks do not conflict with write locks, and thus, multiple readers are allowed to read the last committed version of a record concurrently while a writer updates a private version. Each writer maintains a private copy of its updates during execution. Creating the private copy incurs no overhead, as the copy created by undo logging mechanism is reused. At commit time, writers convert their write locks into certify locks that conflict with both read and write locks. If a transaction certifies all writes, the commit succeeds and updates are propagated from the private copy to the database. However, even if a single lock cannot be certified, the transaction is aborted and private updates are discarded.

In contrast to these protocols, SILO is an optimistic protocol proposed originally in [42]. SILO tracks record accesses using transaction-private read/write sets during transaction execution and uses a three-step transaction commit protocol. The first step latches all records that need to be updated. The second step verifies the records in the read set. If the first two steps succeed, the third step applies the updates to the actual database.

3.2 SE Implementation

The Trireme SE implementation supports all the aforementioned concurrency control protocols. We add several optimizations that have been proposed in prior research to improve the scalability of two-phase locking protocols. Our NO_WAIT implementation is based on the optimized version proposed by VLL [37] that eschews a centralized lock table. As NO_WAIT aborts any transaction that attempts to acquire a lock in a conflicting mode, no transaction ever waits on a lock. Thus, the only metadata required to implement NO_WAIT protocol is a single counter that keeps track of whether a shared or an exclusive lock has been acquired, and in case of a shared lock, how many transactions currently hold it. The SE NO_WAIT implementation in Trireme associates a semaphore with each record to keep track of this information similar to VLL. The semaphore implementation uses atomic instructions to physically synchronize simultaneous updates from multiple threads.

The SE 2V_NO_WAIT implementation is based on a latch-free variant proposed in prior research [39]. Each record is associated with a read counter and a write flag. Readers proceed by first checking the read counter. A value of -1 indicates that another transaction holds a certify lock on the record, in which case, the reader is aborted. Otherwise, the read counter is incremented using a Compare-and-Swap (CAS) operation to indicate the acquisition of a read lock. A writer proceeds by doing a CAS to set the write flag. A set flag indicates a failure to acquire the write lock, in which case, the writer is aborted. Otherwise, the writer proceeds to update its private copy of the data. Certification is performed by setting the read counter of each record in the write set to -1 and it succeeds if previous read counter values were 0. On successful certification, the transaction is committed by copying the updates from the private copy to the database, decrementing the read counter for all records in the read set, and setting the read counter to 0 for all records in the write set.

Similarly to NO_WAIT, our optimized DL_DETECT implementation also does not maintain a central lock table. Instead, we distribute the lock table metadata (owner and waiter lists) by storing it on a per-record basis. In addition to the lock table, maintaining a central waits-for graph has also been shown to be a scalability bottleneck in DL_DETECT [47]. To avoid centralized graph maintenance from becoming a bottleneck, we also partition the graph across threads by maintaining thread-local dependency lists. Thus, when a transaction updates the waits-for graph, the associated thread updates only its thread-local dependency without synchronizing with other threads. Cycle detection is performed by having a thread search for cycles in a partial waits-for graph that is constructed from just the relevant partitions. Finally, our DL_DETECT implementation also exploits the static, one-to-one mapping between maximum number of concurrent transactions, worker threads and cores by using array-based data structures in place of slow, dynamic data structures for implementing the thread-local adjacency list. This substantially simplifies memory management and accelerates cycle detection.

3.3 PSE Implementation

The PSE implementation physically partitions data and indices across threads. A transaction locks an entire partition before accessing data belonging to it. An efficient approach of implementing such partition-level locks is to simply associate a latch with each partition. When a transaction accesses data belonging to a partition, the thread that runs the transaction acquires the corresponding partition latch before accessing the data item. Latches are acquired in a deterministic order to avoid deadlocks.

A common way of avoiding deadlocks is to latch all relevant partitions upfront based on the partition-id or thread-id before executing the transaction. While this approach requires upfront knowledge of all data accesses made by a transaction, it is conceptually simple to implement and has been used as a baseline PSE implementation by prior work [23, 42]. Thus, Trireme also uses this approach for implementing the PSE architecture. Due to the use of partition latching, no other CC protocols are needed. Furthermore, as a transaction executes only after acquiring all partition latches, aborts are impossible, which makes PSE agnostic to the presence or absence of updates. We also use the partition-latching based implementation to highlight how easily PSE can exploit upfront knowledge of data accesses to avoid transactional aborts.

3.4 Delegation Implementation

Similarly to PSE, the Delegation implementation also partitions data across threads. Each thread mediates access to partition-local records, indices, and all associated metadata. However, Delegation differs from PSE in the way it handles multi-site transactions. When a transaction running on one thread wants to perform an operation on a record owned by a different thread, it *delegates* that operation to the target thread by sending it a message, requesting the target thread to perform the operation on its behalf. Message passing is implemented as a thin layer over hardware cache coherence by using shared variables to implement synchronization-free single-writer, single-reader queues [2, 28, 36]. Thus, messages are sent and received by simply writing or reading shared variables.

As transactions can delegate operations in any order, Delegation enforces serializable transaction execution by using logical syn-

chronization with CC protocols. As in SE, Trireme implements all three CC protocols in the Delegation architecture as well. The NO_WAIT and 2V_NO_WAIT implementations associate corresponding counters and flags with each record similarly to the SE implementation. The DL_DETECT implementation associates owner/waiter lists with each record like SE. However, unlike SE, this CC metadata is exclusively accessed and updated only by one thread due to partitioning. Thus, it is completely synchronization free.

To illustrate the difference between Delegation and SE, let us consider an example where a transaction on thread C (client) wants to read a record R owned by thread S (server) using the NO_WAIT CC protocol. Under SE, C uses an atomic operation to update the semaphore to indicate that it has acquired a read-lock on R. Assuming the read-lock is acquired successfully, the transaction on C accesses the record. When the transaction commits, thread C performs another atomic operation to update the semaphore releasing the lock. In Delegation, thread C sends an explicit message to thread S. On receiving this message, S looks up the record using the partition-local index, updates the counter corresponding to the record indicating that a read lock has been acquired, and returns back a pointer to the record to C. C then accesses the record directly using its address. At commit time, C sends another message to S. Once S receives the message, it releases the lock. As only S can access the CC metadata, which is the counter in this case, there is no need to synchronize access using atomics or latching.

The SILO implementation in Delegation also follows the corresponding SE implementation. The only change is the use of message passing to acquire write locks on records at validation time instead of using atomics. Thus, with SILO as the CC protocol, read-only transactions are unaffected by Delegation as the validation phase does not acquire any locks. To summarize, in all cases, the Delegation implementation differs from SE only in the *physical synchronization* technique used (message passing versus atomics) and not with respect to logical synchronization (CC protocols).

3.5 DORA Implementation

Unlike the other architectures that use a thread-to-transaction mapping, DORA uses a thread-to-data mapping, where the database is logically partitioned across threads, and transactions migrate from thread-to-thread depending on the data which is accessed.

In order to make transaction migration efficient, we implemented *fibers*, which are user-level cooperative threads [1], in Trireme. In the DORA implementation, each transaction runs in the context of a fiber. Each Trireme thread can host multiple fibers but only one fiber per thread executes at any given time. When a transaction running on one thread tries to access data belonging to another thread, its corresponding fiber is suspended on the source thread and migrated to the target thread where it resumes execution. The transaction logic is itself agnostic to fiber migration and the responsibility for suspending, migrating, and resuming fibers is performed by a *fiber scheduler* that runs on each thread.

Let us consider the previous example where where a transaction on thread C wants to read a record R owned by thread S. In DORA, the transaction runs in a fiber on thread C. When the transaction tries to access the remote record, the fiber scheduler on C suspends the fiber and migrates it to the fiber scheduler on thread S. Upon receiving the new fiber, the fiber scheduler on S schedules it for execution. If the transaction makes no further remote accesses, it runs to completion in thread S. However, if it needs a record from another thread T, then the fiber migrates again to T. At commit time, the fiber migrates back to each thread where it was executed and acquired locks, in order to release them. So in the example, when the transaction commits, it first releases the locks in thread T. Then, the fiber migrates back to thread S and finally to C to release locks acquired on records in those threads. As data is logically partitioned and as each thread schedules fibers for execution one at a time, no given record can be accessed concurrently from two threads. Thus, similar to Delegation, DORA also eliminates the need for physical synchronization with latches or atomics. Instead, DORA uses thread migration as the physical synchronization technique.

In DORA, we have implemented only the pessimistic locking protocols (NO_WAIT, DL_DETECT, and 2V_NO_WAIT) for logical synchronization. The protocols work similar to their Delegation counterparts. We did not implement SILO with DORA due to two reasons. First, unlike NO_WAIT, DL_DETECT, and 2V_NO_WAIT that update CC metadata on both reads and writes, SILO does not update any metadata on reads. Thus, for read-intensive workloads, the thread-to-data mapping would cause unnecessary fiber migrations. Second, we could implement SILO in DORA by using fiber migration to lock records at validation time similar to the Delegation implementation. In such a case, DORA, Delegation, and SE would all perform similarly under read-intensive workloads. However, as we show later (Section 4.3), SILO does not scale well even with Delegation under update-intensive workloads due to longer validation time. Thus, SILO would also not scale well under DORA as we only replace message passing with fiber migration. Thus, we do not implement SILO with DORA.

Fiber implementation. The fiber implementation is based on the Linux ucontext library. This library provides several calls for manipulating machine context (program counter, register set, stack/base pointer, etc.), and is commonly used to implement user-level thread packages. ucontext also enables efficient fiber migration by i) saving the machine context to memory, ii) sending a pointer to the saved context, and iii) resuming the fiber by restoring the machine context on the remote thread. Despite the fast context switching ability provided by the ucontext library, we ran into scalability issues beyond eight cores. On profiling, we identified that the *swapcontext* library call, which is used for context switching from one fiber to another, makes a *sigprocmask* system call to save and restore signal masks, which limited scalability. To solve this problem, we wrote a custom swapcontext that saves and restores registers without making the system call. With this change, context switching between fibers poses no scalability issues as it works entirely in user space.

Difference from Shore-MT DORA. The Shore-MT DORA implementation [31] decomposes a transaction into a collection of actions that are routed to threads according to the data they access similar to our fiber implementation. However, the Shore-MT DORA implementation also exploits intra-transaction parallelism by running actions concurrently on different threads. In order to keep comparison across architectures fair, we do not exploit intra-transaction parallelism and process data requests within each transaction in strict serial order in all architectures. Thus, instead of decomposition a transaction into actions, we simply move the fiber hosting the transaction from one thread to another depending on data accessed. As an additional benefit, our implementation is completely synchronization free compared to the Shore-MT DORA that has to synchronize actions using rendezvous points. Thus, our realization of the DORA architecture is more general purpose in nature and suitable to the main-memory OLTP context.

4. EVALUATION

In this section, we present the results from our experimental analysis. All the experiments were conducted on a server equipped with four 18-core Intel E7-8890 v3 processors (32-KB L1I + 32-KB L1D cache, 256-KB L2 cache, and 45-MB LLC) clocked at 2.5GHz, and 512-GB of DDR4 DRAM. Even though our proces-

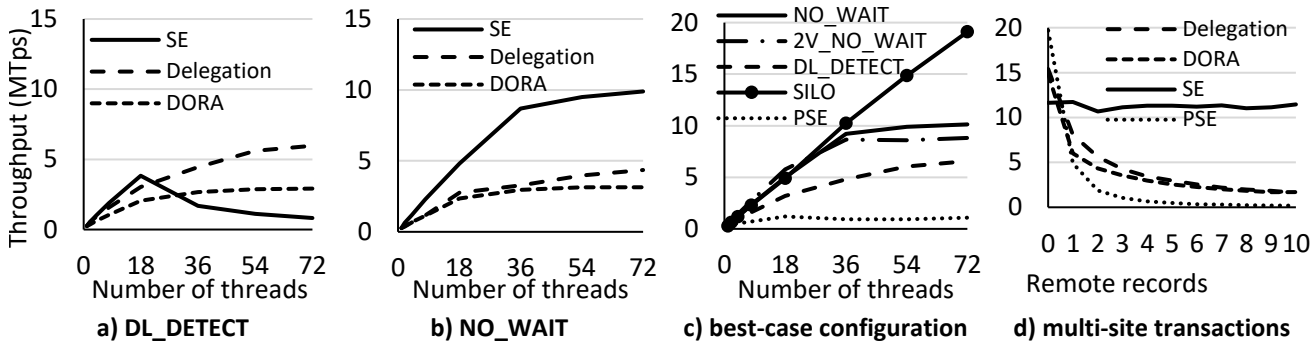

Figure 2: Read-only scalability of various architectures: DL_DETECT , NO_WAIT , and best case configuration as well as sensitivity to multi-site transactions

sor supports two hardware threads per core via HyperThreading, we do not use it to avoid variability in results. Thus, we setup Trireme so that there is a one-to-one mapping between database threads and physical cores. Each thread is pinned to its core to avoid OS scheduling overhead. We also use a socket-fill scheduling strategy, where we pin threads to cores within one socket before using another socket.

4.1 Benchmarks & Methodology

We use Yahoo! Cloud Serving Benchmark (YCSB) [7] as our macrobenchmark. For all YCSB experiments in this paper, we use a 20GB database containing a single table with 20 million records. Each YCSB record has a single primary key field and ten 100-character string fields. Each transaction reads or updates 16 records with record accesses following a zipfian distribution controlled by a parameter *theta*.

YCSB is partitioning unfriendly as records chosen by the zipfian distribution need not be local to any given thread's partition. However, PSE, Delegation, and DORA architectures are based on data partitioning. In order to separate the overhead of multi-site transactions from high contention, we use a microbenchmark that allows us to control the partitionability aspect of the workload. Our microbenchmark uses a 13GB database containing a single table of 100 million records, where each record has one 64-bit primary key and ten 64-bit integer attributes. Each transaction reads or updates ten records from the table. In order to induce contention in the workload, the benchmark divides the records into a hot set and a cold set. Each transaction accesses two records from the hot set and the remaining eight from the cold set. We use this microbenchmark as it has also been used in several studies for evaluating the scalability of OLTP engines under high contention [36, 43, 45, 47].

We use throughput, measured in terms of millions of transactions committed per second (Mtps), as the performance metric. Each experiment is repeated five times and in each run, we execute transactions for 10 seconds before reporting the steady-state throughput. We only report the median throughput value as the relative standard deviation was less than 10% in all cases.

The rest of this section is organized as follows. We start our analysis in this section by first focusing on the interaction between CC protocols and physical synchronization techniques used by various architectures (Section 4.2). Using a read-only microbenchmark, we identify issues that hamper scalability at the physical level as there are no logical conflicts. In Section 4.3, we use an update-intensive microbenchmark to stress synchronization on both levels. We present results in the following order. First, we fix the CC protocol and vary the system architecture to show the interaction between a given CC protocol and various system architectures. Then, for each CC protocol, we pick the best system architecture and show the scalability and throughput behavior of these best-case combinations. Finally, we switch our focus to the YCSB macrobenchmark in Section 4.4. We present the scalability of the best CC–architecture combinations under both read-only and update-only YCSB, evaluate sensitivity to contention by varying the theta parameter, and study scalability under mixed workloads by varying the reader–writer ratio.

4.2 Isolating physical synchronization

To isolate and study the impact of physical synchronization, we use the microbenchmark to generate a high-contention read-only workload by fixing the number of hot records to 16. SE does not partition the data set. Thus, each transaction picks two records randomly from the hot set of 16 records and the remaining eight randomly from the rest of the data set (100M records). Delegation, DORA, and PSE, however, partition the data set across threads. Thus, the location of records plays a crucial role in determining performance. Empirical analysis showed us that storing all hot records in a few partitions creates severe load imbalance as we scale the number of threads. Similarly, the location of cold records also plays an important role in determining the overall throughput due to the overhead associated with multi-site transactions that affect all partitioning-based architectures.

As the problem of identifying the optimal partitioning scheme to avoid load imbalance and minimize multi-site transactions [9, 33] is orthogonal to the problem we consider here, which is interaction between CC protocols and architecture, we configured the system to spread out the 16 hot records across 16 cores for Delegation, DORA, and PSE. Further, we also configured the system so that accesses to cold records are always from the local partition. Thus, for partitioned architectures, each transaction accesses two remote hot records from the hot set and the remaining eight cold records from the local partition. As a result, the workload is 100% multi-site in nature as each transaction performs two remote operations.

4.2.1 DL_DETECT*–architecture interaction*

First, we consider the interaction between DL_DETECT protocol and various architectures. As the read-only microbenchmark workload has no conflicting data accesses at the logical level, it should be possible to schedule transactions concurrently and achieve near-linear increase in throughput. Figure 2 (a) shows the scalability of the DL_DETECT under SE, Delegation, and DORA architectures.

At low thread counts, the SE DL_DETECT implementation outperforms the rest. This is due to the multi-site transaction overhead inherent to the partitioned-nature of Delegation and DORA. When a multi-site transaction tries to access a record from a remote partition, the corresponding client thread has to send a message and receive a reply from the server thread that owns the data before the transaction can access the data in case of Delegation. This synchronous request–response interaction is much slower than directly

accessing a record in SE at low thread counts. Remote data access overhead is even higher for DORA due to the fact that fibers have to be migrated twice during the lifetime of a transaction, once during transaction execution for acquiring locks and once at commit time for releasing back the locks. Due to these overheads, Delegation and DORA lag behind SE at low thread counts.

The second observation from Figure 2 (a) is that the trend of SE outperforming the rest reverses at high thread counts, as Delegation and DORA outperform SE by 7× and 3× respectively with 72 threads. These three architectures differ only in the physical synchronization mechanism used. The DL_DETECT protocol itself remains unchanged. The implementation of DL_DETECT in SE uses latches to protect the lock waiters and owners lists. As concurrent transactions attempt to add themselves to the lock-owner list of a few shared records, threads contend for the latches corresponding to these "hot" records leading to poor scalability at the physical level. These results corroborate recent studies that have shown that pessimistic deadlock detection protocols fail to scale with the SE architecture under high contention [36, 43, 47]. Results from DBX1000, presented in Section 5, show a similar trend.

Delegation and DORA, in contrast, eliminate latching and replace it with explicit thread synchronization based on message passing or fiber migration. Thus, they do not suffer from latch contention and provide better scalability. These results clearly show the benefit of message passing or fiber migration as physical synchronization mechanisms under high contention. These results also contradict prevalent wisdom that DL_DETECT-based locking protocols cannot scale even under read-intensive, high-contention workloads [36, 43], as they show that such lack of scalability is limited only to SE architecture due to its non-scalable physical synchronization technique.

4.2.2 NO_WAIT–architecture interaction

Let us now consider the interaction between NO_WAIT protocol and various architectures. Figure 2 (b) shows the scalability of the NO_WAIT for all three architectures. Comparing Figures 2 (a) and (b), we can make two important observations.

First, NO_WAIT scales much better than DL_DETECT in SE. As the SE NO_WAIT implementation is based on VLL [37], it uses atomic instructions to update a counting semaphore. Thus, it scales better than the latch-based DL_DETECT implementation. But at high thread counts, multiple threads attempt to update a few counters corresponding to the hot records, and the atomic instructions they use target a few shared memory words. As a result, the hardware cache coherence mechanism that provides support for these atomic operations ends up constantly moving these contended memory words from one core-local cache to another resulting in lack of scalability.

Second, while Delegation and DORA scale better than SE for DL_DETECT , this is not the case with NO_WAIT , as SE clearly outperforms Delegation by 1.5× and DORA by 3×. As the overhead of using message passing and fiber migration is greater than the benefit gained by avoiding atomic instructions, Delegation and DORA lag behind SE with NO_WAIT. This shows that no one architecture is optimal for all CC protocols.

4.2.3 Best case comparison

So far, we have presented the interaction between NO_WAIT and DL_DETECT in various system architectures. Figure 2 (c) shows the scalability of five CC protocol–system architecture combinations under the microbenchmark. The system architectures listed here are the best case for each CC protocol. We saw earlier that NO_WAIT works best with SE and DL_DETECT works best with Delegation. Our SILO implementation in Delegation uses message passing only during validation phase to acquire locks on records that are updated. Thus, for the read-only workload, SILO behaves similarly for Delegation and SE, and we include SE results here. We also report only the SE result for 2V_NO_WAIT as the latch-free SE implementation outperforms Delegation and DORA under 2V_NO_WAIT due to the overhead of message passing and fiber migration similarly to NO_WAIT. Finally, PSE uses coarse-grained partition locking instead of record-level CC.

Figure 2 (c) shows that when paired with the right system architecture, all CC protocols scale, albeit at different rates, for the high-contention read-only workload. As SILO is an optimistic protocol, it does not perform any synchronization under this read-only workload and thus does not suffer from any scalability issues. With 72 threads, SE NO_WAIT and 2V_NO_WAIT lag behind by 2× due to contention caused by atomic instructions, and Delegation-DL_DETECT lags behind by 3× due to the overhead of remote data accesses. PSE is the only architecture that fails to scale under this workload. This is due to the well-known multi-site transaction execution overhead that PSE architecture suffers from [23, 42] and we explore this further next.

Insight: *Contention at the physical level can be alleviated using scalable physical synchronization techniques.* Optimistic schemes have been shown to outperform pessimistic protocols on multi-cores by 100× under high-contention workloads due to unscalable nature of physical synchronization used in the SE architecture [43, 47]. Our analysis shows that combining pessimistic protocols with architectures that use scalable physical synchronization techniques, such as message passing or fiber migration, can bridge the gap between pessimistic and optimistic schemes. For example, SE-DL_DETECT lags behind SILO by a factor of 20, but Delegation-DL_DETECT lags only by a factor of 3.

4.2.4 Sensitivity to multi-site transactions

Figure 2 (a)-(c) shows that the throughput of partitioning-based architectures is sensitive to the presence of multi-site transactions in the workload. To quantify the overhead of multi-site transactions, we perform a sensitivity analysis using 72 threads. We modify microbenchmark so that each transaction reads ten random records from the whole dataset of 100M records. Thus, there is no contention in the workload. We vary the number of operations that access records in remote partitions to isolate the impact of multi-site transactions. We pick a random record from a random remote partition. Thus, as we increase the number of remote operations, we also access data from multiple remote partitions.

Figure 2 (d) shows the throughput of various architectures using the NO_WAIT protocol under this low-contention workload as we increase the number of remote accesses from zero to ten. The results for other protocols are similar as the workload is contention free. There are several important observations to note. First, let us consider the zero-remote-operations case. This is the best scenario for partitioned systems as all transactions access records from a single partition and are hence single-site in nature. All partitioning-based architectures outperform the SE in this scenario. This is due to the well-known cache-locality benefits of physical partitioning [23, 42]. PSE outperforms Delegation and DORA due to additional benefits provided by better code locality as PSE does not contain the logic for fine-grained concurrency control.

Second, we see that simply adding one remote operation to each transaction results in a drop in throughput for all partitioned architectures. Note here that all transactions are multi-site although each transaction performs only one remote operation. The reason for performance drop is different for PSE compared to Delegation and DORA. PSE executes transactions serially by having each thread lock all relevant partitions before executing a transaction. Thus, in

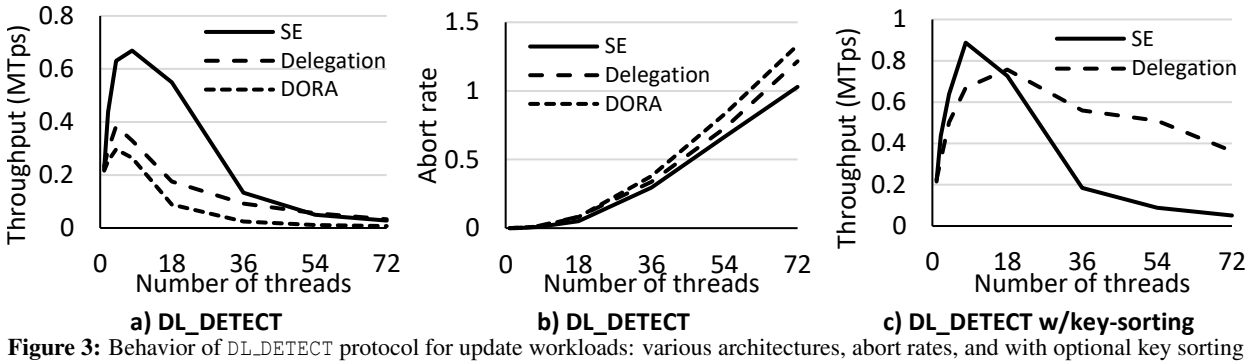

Figure 3: Behavior of DL_DETECT protocol for update workloads: various architectures, abort rates, and with optional key sorting

the presence of one remote operation, each thread locks two partitions before executing a transaction. Thus, in the best case, only 36 threads can execute concurrently. With 72 threads and one remote operation per thread, half of the threads end up waiting for locks held by another thread resulting in throughput drop. As the number of remote operations increases, concurrency plummets resulting in throughput collapse. Delegation and DORA, on the other hand, do not use partition locking. Instead, they use concurrency control protocols, in this case NO_WAIT, to perform fine-grained record locking. When a transaction accesses a record stored in a remote partition, the client thread that hosts the transaction sends a message to the server thread that owns record in case of Delegation, or the client fiber is migrated to the server thread in case of DORA. Thus, performance drops due to the overhead of physical synchronization (message passing or fiber migration). Despite this overhead, Delegation and DORA provide more concurrency than PSE due to the use of fine-grained locking. This explains why they outperform PSE under multi-site workloads.

Third, SE is insensitive to multi-site transactions. When all transactions are multi-site with all operations being remote, SE provides a 7× improvement over Delegation and DORA, and a nearly 60× improvement over PSE. This advantage of SE over PSE is well known [35] and highlights the partitionability-agnostic nature of SE. What our analysis shows here additionally is how using partitioning with fine-grained concurrency control, like Delegation or DORA, can provide 10× improvement in throughput compared to coarse-grained locking under such extreme workloads.

4.3 Impact of updates

In this section, we examine the scalability of various architectures under an update-intensive workload. We generate the workload using the same microbenchmark as before with the exception that each transaction updates 10 records instead of reading them.

4.3.1 DL_DETECT*–architecture interaction*

Figure 3 (a) shows the scalability of Delegation, DORA, and SE architectures when DL_DETECT is used as the CC protocol. Comparing Figures 2 (a) and 3 (a), we see that the throughput under the SE architecture is at least an order of magnitude lower in the update-intensive workload compared to its read-only counterpart. This is because DL_DETECT suffers from three bottlenecks under the update-intensive workload, namely, aborts due to deadlocks, lock thrashing even in the absence of deadlocks, and latch contention.

Deadlocks and aborts. As we increase the number of threads, we also increase the level of logical contention in the workload as transactions attempt to perform conflicting updates on a few hot records, leading to deadlocks. In order to preserve serializability, DL_DETECT aborts deadlocked transactions. Figure 3 (b) shows the abort rate (#aborts/#commits) for DL_DETECT at various thread counts. At 72 threads, almost all transactions get aborted at least once irrespective of the architecture. These aborted transactions result in wasted work and negatively impact scalability.

Latch contention. In addition to the overhead due to aborts, DL_DETECTalso suffers from physical synchronization overhead due to latch contention (Figure 2 (a)). However, unlike in the read-only case, Delegation and DORA do little to improve scalability for the update-intensive workload. At low thread counts, SE outperforms Delegation and DORA due to the overhead of message passing (in Delegation) or fiber migration (in DORA). At high thread counts, DL_DETECT has very high abort rates under all architectures. Thus, even through Delegation and DORA eliminate the physical synchronization bottleneck of SE, they do little to solve the logical synchronization issues created by DL_DETECT.

Lock thrashing. Lock thrashing occurs due to the loss of concurrency caused by transactions holding locks until commit time, thereby preventing other transactions from running [4, 47]. This leads to deterioration in throughput even in the absence of deadlocks. In order to isolate the impact of lock thrashing, we modify the update-intensive workload so that transactions acquire locks in primary key order. Figure 3 (c) shows the scalability of SE and Delegation under this workload. Comparing Figure 3 (c) and Figure 3 (a), we see that Delegation helps in improving throughput under the key-ordering workload. In the absence of aborts due to deadlocks, the benefit of message passing directly translates into higher throughput. However, Delegation does little to solve the decrease in concurrency caused by lock thrashing.

4.3.2 NO_WAIT*,* 2V_NO_WAIT *and* SILO *interaction*

Figures 4 (a) and (b) show the scalability of NO_WAIT and SILO protocols under various architectures. We do not show the results for 2V_NO_WAIT as it behaves similarly to NO_WAIT. Clearly, both Delegation and DORA do not scale beyond a single thread as just adding a second thread results in throughput collapsing. This behavior is different from the read-only case, where Delegation and DORA did scale well despite lagging behind SE (Figures 2 (b)). The reason behind this behavior can be seen in Figure 4 (c) which shows the abort count of NO_WAIT CC protocol under various architectures for the update-intensive workload. As can be seen, NO_WAIT has at least an order of magnitude lower abort rate under SE at high thread counts. Comparing this with with DL_DETECT (Figure 3 (b)), we see that NO_WAIT, 2V_NO_WAIT, and SILO suffer from inflated abort rates for Delegation and DORA.

Profiling the system revealed that this rapid increase in the abort rate is due to longer lock hold time under Delegation and DORA. For instance, at 72 cores, with the NO_WAIT CC protocol, the average commit latency of transactions is 5us under SE, 70us under Delegation, and 600us under DORA. In contrast, the average latencies under an equivalent low-contention workload, where each transaction accesses two randomly chosen remote records and eight local records, was 15us for Delegation and DORA.

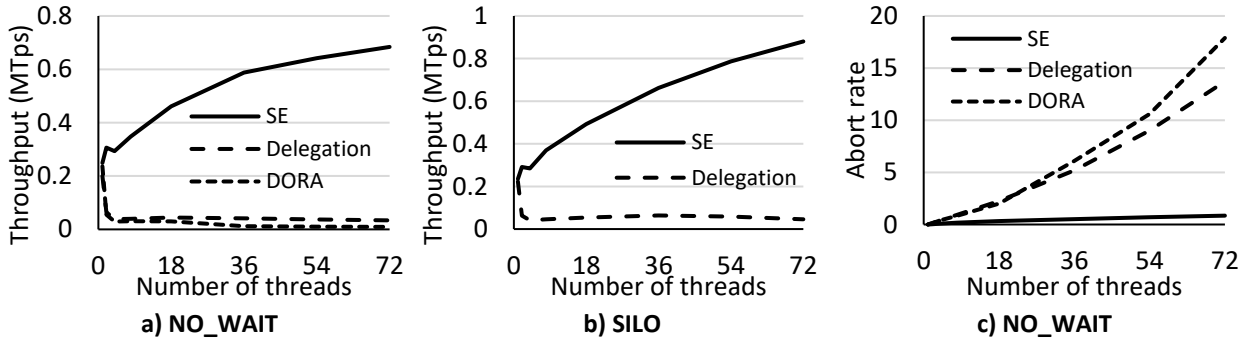

Figure 4: Scalability for update workloads: NO_WAIT protocol, SILO protocol, and NO_WAIT abort rates

Let us consider Delegation to see why this happens. When a client thread wants to access a remote record, it sends a message to the server thread requesting the record in update mode. Upon receiving this request, the server thread updates the record-local metadata to indicate that the record has been locked in exclusive mode and grants access to the client. From this point, all other requests to this record will be denied, and all requesting transactions aborted, until the former client sends a release message unlocking the record. Under high contention, threads containing hot records become overloaded as they have to reply to lock requests from other threads. This load imbalance increases the round-trip time for remote lock requests, which, in turn, increases the duration for which already-acquired locks are held. While this increased lock hold duration results in transactions being blocked for longer duration in DL_DETECT, it increases the number of aborts under NO_WAIT, 2V_NO_WAIT and SILO leading to throughput collapse.

Insight: *Under update-intensive high-contention workloads, contention at the logical level can reverse the benefit of scalable physical synchronization techniques.* While read-intensive workloads create contention at the physical level, update-intensive workloads create contention at both logical and physical level. Our analysis shows that under such workloads, even in the absence of any physical synchronization overhead, CC protocols still do not scale due to lock thrashing (DL_DETECT) or transaction aborts (NO_WAIT, 2V_NO_WAIT, and SILO). Worse yet, physical synchronization that improves scalability under read-intensive workload negatively interacts with some CC protocols under update-intensive workloads leading to poor scalability.

4.3.3 Best case comparison

Figure 5 presents the scalability of the best CC–system architecture combinations under the update-intensive workload. First, note that NO_WAIT and 2V_NO_WAIT behave similarly. This is expected given that the workload is an update-only workload both NO_WAIT and 2V_NO_WAIT do not permit concurrent updates.

Second, DL_DETECT outperforms the rest at thread counts below 18. Figure 5 shows that, in this case, the abort rate of DL_DETECT is much lower than the rest. This is expected given the fact that optimistic and deadlock avoidance protocols abort transactions that can be serialized and committed by DL_DETECT [4]. However, at high thread counts, this trend reverses and DL_DETECT lags behind the rest. Figure 5 shows that at 72 threads, DL_DETECT has a higher abort rate than both SILO and NO_WAIT as i) DL_DETECT already has low throughput due to various overheads, and ii) several transactions that are executed get aborted due to actual deadlocks [36, 43]. Thus, a low commit rate and a high abort rate results in DL_DETECT performing poorly.

Third, comparing Figure 5 with the read-only case (Figure 2 (c)), we see that the trend is reversed again. While all architectures outperform PSE under the read-only workload, they lag behind PSE

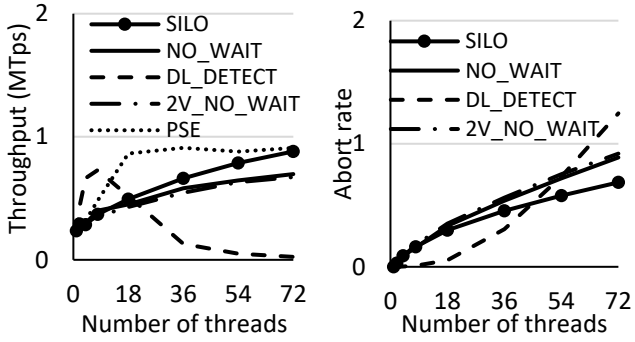

Figure 5: Best-case comparison for update-intensive workload: throughput and abort rates

in the update-intensive workload. Since PSE uses partition locking instead of record locking, it is immune to updates in the workload. Thus, PSE scales similarly under both read-only and update-intensive workloads. PSE stops scaling beyond 18 cores due to contention at both logical level, as threads wait for partition latches, and physical level, as latch acquisition crosses the socket boundary. Our PSE implementation acquires all partition locks upfront in a deterministic order due to which transaction aborts are impossible. This explains why PSE outperforms other architectures which suffer due to transactional aborts. It is important to note that spreading out 16 hot records across 16 cores plays an important role in improving PSE throughput. Without this, PSE would offer an order of magnitude lower throughput due to load imbalance. This suggests that PSE-based skew handling mechanisms should use fine-grained data partitioning for improving throughput under high contention.

Insight: *Coarse-grained locking based on upfront knowledge of data accesses helps in improving throughput under high-contention workloads.* Our analysis reveals that despite all transactions being multi-site in nature, lack of aborts in PSE helps in *improving throughput* under high-contention workloads. However, PSE *does not improve scalability* as coarse-grained partition locking severely limits concurrency.

4.4 YCSB Results

In this section, we use YCSB benchmark to evaluate: i) scalability of various CC–architecture combinations, ii) sensitivity to the degree of contention by varying the theta parameter, and iii) scalability under mixed read–write workloads.

4.4.1 Read-only scalability

Figure 6 (a) presents the scalability of various CC–system architecture combinations under a high-contention, read-only YCSB workload generated by setting theta = 0.8 [47]. We present only SE results for NO_WAIT, 2V_NO_WAIT, and SILO, as SE outperformed

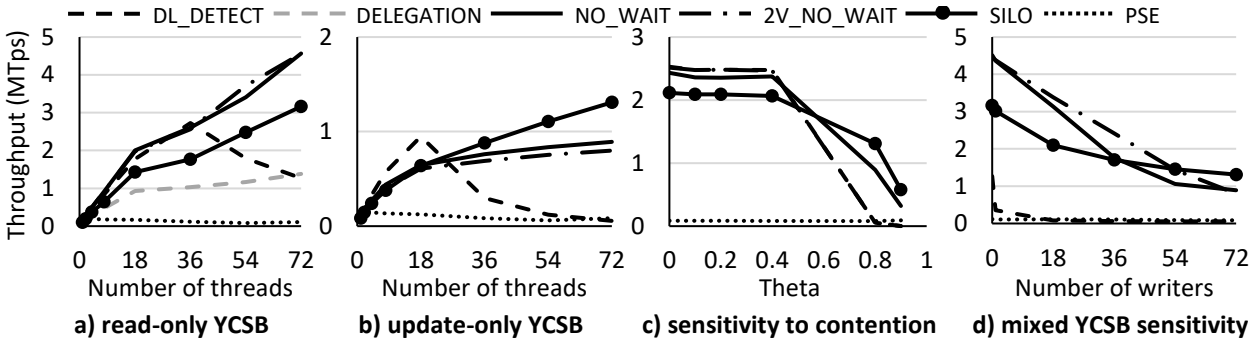

Figure 6: YCSB experiments: read-only scalability, update-only scalability, sensitivity to contention, and mixed workload sensitivity

Delegation and DORA similarly to the microbenchmark case (Section 4.2). For DL_DETECT, we present both Delegation and SE.

Comparing these results with the microbenchmark results (Figure 2) (c), we can observe several differences. First, under the microbenchmark, scalability under NO_WAIT and 2V_NO_WAIT plateaus at 54 cores due to lack of scalability of atomic instructions. However, under YCSB, both scale well. Under this YCSB workload, 10% of the 20M tuples are accessed by 60% of all transactions. Thus, the level of contention is lower than the microbenchmark, where all transactions access two records from a hot set of 16. As a result, atomic instructions used by NO_WAIT and 2V_NO_WAIT do not pose scalability problems.

Second, under the microbenchmark, Delegation DL_DETECT outperformed SE DL_DETECT. However, under YCSB, Delegation only matches SE DL_DETECT at 72 cores and under performs at lower core counts. Unlike the microbenchmark, where only 2 out of 10 operations were remote, nearly all operations are remote under the YCSB workload. Thus, partitioning-based architectures like Delegation, DORA, and PSE substantially under perform SE.

Third, unlike the microbenchmark, where SE SILO clearly outperformed the rest, SILO lags behind NO_WAIT and 2V_NO_WAIT under this YCSB workload. 2PL-based NO_WAIT and 2V_NO_WAIT protocols do not create a private copy of the record for a read operation. SILO, in contrast, always creates a private copy of the record due to its optimistic nature. Under the microbenchmark, the associated memory copies did not add much overhead as each record is composed of a key and ten 64-bit integers. Under YCSB, however, each record is much larger due to the use of ten 100-byte strings. Thus, the memory copy operation adds overhead to SILO.

4.4.2 Update-only scalability

Figure 6 (b) presents the scalability of the best CC–system architecture combinations under an update-only high-contention YCSB workload (theta = 0.8). We only show the SE results for all CC protocols, as Delegation and DORA fail to scale due to reasons mentioned in Section 4.3.

Comparing Figure 6 (b) and 5, we see that the behavior of CC protocols under SE is consistent in both benchmarks; DL_DETECT performs the worst due to both logical and physical synchronization issues, 2V_NO_WAIT and NO_WAIT both lag behind SILO since they abort more transactions. The main difference is that PSE lags behind NO_WAIT, 2V_NO_WAIT, and SILO by at least 10× under YCSB, while it outperformed them all under the microbenchmark. This is because the benefit of avoiding aborts under PSE is overshadowed by the overhead of multi-site transactions for the YCSB workload.

For partitioning-unfriendly workloads, the difference between PSE and SE grows further as contention decreases. Figure 6 (c) shows the throughput of various configurations as we vary the level of contention in a update-only YCSB workload by increasing theta from 0 to 0.9. For this experiment, we fix the number of cores at 72. Given that most accesses are to remote partitions in this workload, the difference between PSE and the best SE configuration increases from 5.5× under severe contention at $theta = 0.9$ to 30× under very low contention at $theta = 0$.

4.4.3 Mixed workload sensitivity

So far, we have used read-only and update-only workloads to analyze the interaction between architecture and CC protocols. Due to the lack of read–write conflicts in these workloads, we saw that 2V_NO_WAIT behaves identically to NO_WAIT in all cases. In this section, we analyze the performance sensitivity of various configurations under mixed read–write workloads. We only present the results for SE architecture as no Delegation or DORA configuration outperforms the best SE configuration due to the partition-unfriendly nature of the workload. Figure 6 (d) shows the throughput of various configurations under a mixed YCSB workload. For this experiment, we fix the number of cores at 72 and YCSB theta value to 0.8. Each thread is affinitized to be either a reader or a writer. The figure shows throughput as we vary the number of writer threads from zero (read-only) to 72 (update-only).

Figure 6 (d) highlights the benefit of multi-versioning. In the best case, 2V_NO_WAIT provides a 1.35× improvement in throughput over NO_WAIT when the ratio of writers to readers is between 50% (36 threads) and 66% (54 threads). This is expected given that 2V_NO_WAIT can schedule multiple readers concurrently with a writer. PSE, in contrast, is insensitive to workload read–write ratio. Thus, its performance does not change as we increase logical contention by adding more writers. However, PSE lags behind SE due to the partitioning-unfriendly nature of YCSB, and multi-versioning has the effect of widening the gap between PSE and SE under mixed workloads.

Insight: *Partitioning-based architectures yield marginal returns.* Our evaluation shows that Delegation and DORA improve scalability over SE only when i) DL_DETECT is used as the CC protocol , ii) degree of contention is severely high, and iii) the workload is read-intensive with few conflicts at the logical level. Given that NO_WAIT and 2V_NO_WAIT with SE can outperform DL_DETECT in all cases, there is little incentive for using Delegation or DORA on multi-core servers with cache-coherent shared memory.

Similarly, PSE can improve performance under high-contention workloads only when i) the workload is partitioning friendly, and ii) upfront knowledge of data access is used to avoid transactional aborts. Given that SE outperforms PSE in all other cases, and that PSE-style coarse-grained locking can also be used with the SE architecture, we believe that PSE also provides marginal benefit over SE with respect to dealing with high-contention workloads.

To summarize, while no architecture–CC protocol combination works best in all cases, SE architecture is still the best option for building contention-tolerant, in-memory OLTP engines for modern multi-socket, multi-core servers.

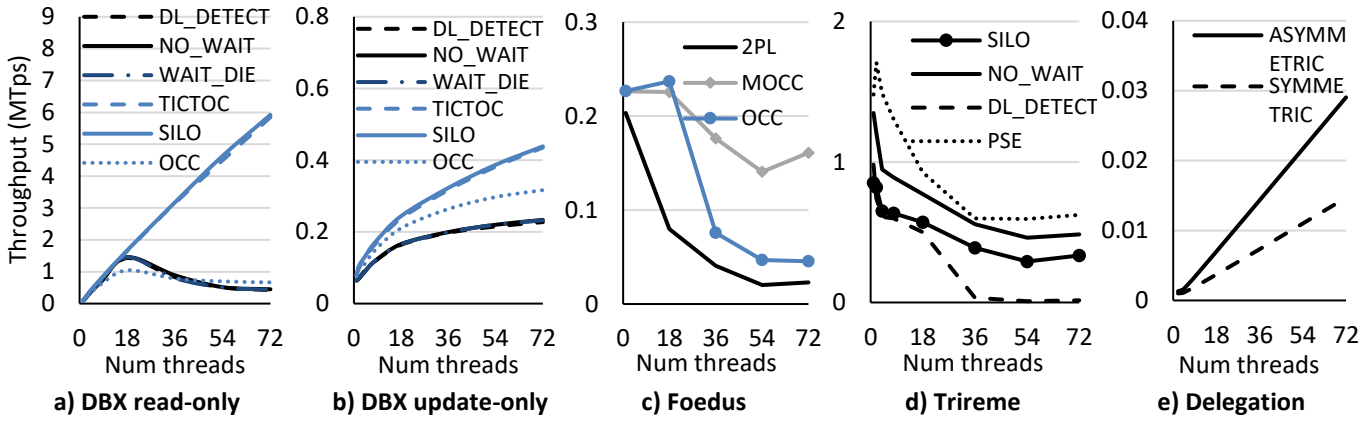

Figure 7: Scalability of CC protocols in DBX1000 for read-only and update microbenchmarks (a)-(b) and the very high contention case: Foedus, Trireme, and symmetric and asymmetric Delegation (c)-(e)

5. DISCUSSION

Scalability of other CC protocols. In our analysis, we used NO_WAIT, DL_DETECT, and SILO as representatives from each of the three classes of CC protocols. However, OLTP engines have also used other CC protocols, and recent analysis has shown that all popular CC protocols suffer under high-contention workloads [47]. In order to corroborate this, and to validate our claim that NO_WAIT, DL_DETECT, and SILO represent the best protocols in their respective classes, we repeated our experiments using DBX1000, a prototype SE OLTP engine that has been used in prior research to evaluate scalability of CC protocols in emerging many-core servers [47]. We changed DBX1000 to support our microbenchmark and deployed it on our server to evaluate the scalability of six protocols–one pessimistic deadlock detection (DL_DETECT), two pessimistic deadlock avoidance (NO_WAIT, WAIT_DIE), and three optimistic protocols (SILO, OCC [26], and TICTOC [48]).

Figure 7 shows the scalability of various CC protocols under the read-only (a) and update-intensive (b) microbenchmarks. First, note that SILO and TICTOC, the two optimistic protocols, have identical performance, and are better than OCC. Similarly, NO_WAIT outperforms WAIT_DIE. This validates our claim that the protocols we implemented in Trireme are the best performing ones in their respective classes. Second, comparing the DBX1000 results with Trireme, we see similar trends. Optimistic protocols scale well under the read-only benchmark while pessimistic protocols fail to scale. All protocols suffer from at least an order of magnitude drop in throughput under the update-intensive benchmark. The only differences are that NO_WAIT and DL_DETECT scale much better under Trireme than DBX1000 due to an optimized implementation.

5.1 Implications of our analysis

Scaling up OLTP systems for high-contention workloads has attracted a lot of attention recently. Research proposals in this area can be broadly classified along three dimensions: i) designing new concurrency control protocols, ii) developing new system architectures, and iii) exploiting application semantics. We will now discuss the implications of our analysis on each of these dimensions and suggest directions for future work.

New CC protocols. Our analysis showed that pessimistic protocols suffer from physical synchronization overhead even under read-only workloads. Even our optimized NO_WAIT implementation does not scale as well as SILO due to contention caused by atomic instructions used in the read-write lock implementation (Figure 2). Designing scalable, NUMA-aware read-write lock is a topic of intense research in the concurrent programming community [6, 10, 16]. Using such locks to further minimize the impact of physical synchronization in both pessimistic and optimistic protocols is a promising direction of future research.

MOCC [43] is a state-of-the-art optimistic protocol proposed recently and used by Foedus [25], a multi-core-optimized main-memory OLTP engine. MOCC exploits the fact that OCC protocols delay serializability verification until commit time by maintaining access statistics for contended records, and using them to determine if a pessimistic lock should be acquired during the read phase. Such a selective use of pessimistic locking results in a reduction in the number of aborts caused by clobbered reads under OCC. MOCC uses MQL, a cancellable read-writer lock optimized for multi-cores, to implement pessimistic locking.

In order to evaluate if the scalability trend changes with such optimized CC protocols, we also experimented with MOCC. We use a simple microbenchmark where each transaction updates ten records from a single table of 72 records to analyze scalability of MOCC on our hardware. We used this microbenchmark as it is supported out of the box in Foedus and has been used in prior research [43]. Figure 7 (c) shows the scalability of 2PL, OCC, and MOCC protocols as implemented in Foedus under the microbenchmark. This result corroborates prior work [43] and shows that MOCC clearly outperforms OCC and 2PL. Figure 7 (d) shows the scalability of PSE and SE architectures in Trireme under the same microbenchmark. While absolute throughput numbers in Figures 7 (c) and (d) are not comparable due to differences in code base, we can make two important observations. First, NO_WAIT 2PL implementation in Trireme scales much better than MOCC 2PL. The MOCC 2PL implementation (referred to as PCC in [43]) delays lock acquisition until commit time compared to NO_WAIT 2PL in Trireme which acquires locks before updates can be performed. As a result, NO_WAIT 2PL will have fewer aborts than PCC. Second, PSE still outperforms other architectures due to lack of aborts. These results suggest that there is still room for optimization.

New system architectures. Recently, several researchers have recently developed scale-out OLTP engines that exploit knowledge of read–write sets to scale distributed transactions under high contention [8, 29, 41]. Thus, an interesting direction of future work would be to examine the applicability of these techniques for avoiding synchronization bottlenecks in the context of a single, scale-up, multi-core server. Orthrus [36] is one such scale-up OLTP engine that combines message passing from Delegation and functional partition from Staged databases [13, 32] with pre-execution-based transaction scheduling. Orthrus divides threads into two types, namely, a set of concurrency control threads that manage the lock table, and a set of transaction executors that run application logic.

The execution threads communicate with the concurrency control threads via explicit message passing. Transactions are pre-executed to identify the read-write sets. Using this information, Orthrus determines an optimal schedule for transaction execution with the goal of minimizing conflicts.

Given the combination of features in Orthrus, it is unclear as to how much each aspect contributes to overall performance. Our analysis already shows that thread-to-data assignment or message passing in isolation do not necessarily improve scalability for update-intensive workloads, as exemplified by our DORA and Delegation results. In order to understand if functional partitioning plays a crucial role in improving performance, we modified the Trireme code base to approximate Orthrus by dedicating one set of threads as servers which serve data requests without hosting transactions and the remainder as clients which run actual transactions. We refer to this as the *Asymmetric-Delegation*, as the normal Delegation configuration uses a single thread to perform both transaction execution and concurrency control. Figure 7 (e) shows the scalability of both approaches for the 72-record, single-table microbenchmark as before using `NO_WAIT` as the CC protocol.

These results show the benefit of functional partitioning as asymmetric Delegation outperforms symmetric Delegation. However, comparing Figures 7 (d) and (e), we see that both symmetric and asymmetric Delegation lag behind `SILO` and `NO_WAIT` with SE and PSE by at least 10×. This shows that of all the techniques used by Orthrus, pre-execution plays the most important role, as without it, both functional partitioning and message passing scale poorly for high-conflict workloads.

Exploiting application semantics. While pre-execution tries to identify records read and written by transactions without any application semantics, recent work has also shown the benefit of using application semantics explicitly to modify transaction run time with the goal of minimizing, or even avoiding, conflicting operations [3, 30, 38, 44, 46]. Given that all CC protocols suffer under high-conflict workloads, such techniques, if applicable, will definitely assist in improving the scalability of all architectures. Our analysis focuses on the scalability behavior of CC protocols and system architectures when such techniques are not applicable.

6. RELATED WORK

Delegation. Calciu *et al*. [5] present several message passing-based concurrent data structures and show that they can outperform their lock-based counterparts under high contention. CPHash [28] is a concurrent hash map that uses asynchronous message passing to scale throughput under high contention. Remote Core Locking [27] replaces latch acquisition with an explicit message to a dedicated server thread that mediates accesses to critical sections. Our analysis shows that message passing is also useful in OLTP as long as the workload is conflict free. In the presence of conflicts, message passing increases the lock-hold duration and causes an amplification of concurrency-control-enforced transactional aborts.

Non-partitioned OLTP engines. Several studies focus on scaling traditional disk-based OLTP engines on multi-cores by eliminating scalability bottlenecks caused by global latching on centralized data structures [15, 17, 18, 19, 20, 22]. As modern main-memory OLTP engines [11, 23, 24] avoid such overheads by design, they scale much better than their disk-based counterparts.

Recent research has shown that pessimistic locking protocols suffer from scalability issues under high contention due to the use of latching or atomics [43, 47]. In contrast, we show that optimized physical synchronization techniques can help in bridging the gap between pessimistic and optimistic protocols. Similarly to our read-write lock, VLL also proposed replacing lock lists with a counting semaphore for physical synchronization [37]. We showed that, even though such an approach performs better than a vanilla `NO_WAIT` implementation, it still does not scale as well as optimistic protocols (like `SILO`) under read-intensive workloads due to the contention caused by atomic instructions. VLL did not experience this effect due to the use of selective contention analysis which relies on upfront knowledge of transaction read/write sets.

PSE engines. Research on PSE engines has focused on reducing multi-site transactions by using workload-driven static partitioning [9, 33], or adaptive repartitioning [34]. Jones *et al*. [21] showed that introducing concurrency control to a distributed PSE engine helps in improving overall throughput in the presence of multi-site transactions. We showed that Delegation provides similar benefits over PSE within a single server.

Other system architectures. DORA [31] logically partitions data across threads and executes a transaction by assigning actions to threads based on the data they access. PLP [32] extends DORA by physically partitioning the index and even shared buffer pool pages. Both DORA and PLP were implemented in Shore-MT [19], a disk-based OLTP engine. In this paper, we revisit the data-oriented architecture in the main-memory OLTP context.

Multimed [40] treats a multi-core like a distributed system by running multiple database instances on separate cores with replication. Porobic *et al*. [35] investigate the impact of deploying existing SE engines in various granularities on multi-socket multi-cores. Our analysis, in contrast, focuses on understanding architectural aspects that play a key role in designing main-memory OLTP engines.

7. CONCLUSION

In this paper, we perform a thorough analysis of the impact of system architecture on scalability of main-memory OLTP engines under high-contention workloads. We implement four system architectures (SE, DORA, PSE, and Delegation) and three CC protocols in Trireme–a main-memory OLTP engine testbed. Using Trireme, we analyze the scalability of each architecture and the interaction between the CC protocols and the architectures. Our results both corroborate and refute prevalent wisdom.

As expected, optimistic protocols scale better than pessimistic ones under read-intensive workloads. However, unlike prevalent wisdom, we showed that pessimistic protocols are not inherently unscalable. Using the right physical synchronization techniques, it is possible to implement pessimistic protocols that scale well under read-intensive workloads even under high contention. As shown by prior research [2, 5, 28, 36], message passing is a promising technique to avoid physical synchronization bottlenecks in the presence of contention. However, we showed that under update-intensive workloads, message passing amplifies the negative effect of transaction aborts due to increase in lock-hold time, resulting in throughput collapse. The tradeoffs between PSE and SE with respect to multi-site transactions is well known. However, we showed that despite low concurrency due to coarse-grained locking, PSE can outperform all other architectures under partitioning-friendly, update-intensive, high-contention workloads due to the ability to avoid transactional aborts. Finally, we discussed the implications of our analysis on a few recent solutions and highlighted directions for future work.

ACKNOWLEDGMENTS

We would like to thank Eliezer Levy, Shay Goikhman, the anonymous reviewers, and the DIAS lab members for their constructive feedback. This work is funded by the an industrial-academic collaboration between the DIAS laboratory and Huawei Central Software Institute (CSI), and the Swiss National Science Foundation (Grant FNXCore No. 200021 146407/1).

8. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *USENIX ATC*, pages 289–302, 2002.

[2] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki. The Case for Heterogeneous HTAP. In *CIDR*, 2017.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3):185–196, 2014.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[5] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *OPODIS*, pages 83–97, 2013.

[6] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. Numa-aware reader-writer locks. In *PPoPP*, pages 157–166, 2013.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SOCC*, pages 143–154, 2010.

[8] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX ATC*, pages 21–21, 2012.

[9] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2):48–57, 2010.

[10] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48, 2013.

[11] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*, pages 1243–1254, 2013.

[12] V. Gasiunas, D. Dominguez-Sal, R. Acker, A. Avitzur, I. Bronshtein, R. Chen, E. Ginot, N. Martinez-Bazan, M. Muller, A. Nozdrin, W. Ou, N. Pachter, D. Sivov, and E. Levy. Fiber-based architecture for nfv cloud databases. *PVLDB*, 10(12):1682–1693, 2017.

[13] S. Harizopoulos and A. Ailamaki. A case for staged database systems. *IEEE Data Eng. Bull.*, 28(2):11–16, 2005.

[14] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Found. Trends databases*, 1(2):141–259, 2007.

[15] T. Horikawa. Latch-free data structures for dbms: Design, implementation, and evaluation. In *SIGMOD*, pages 409–420, 2013.

[16] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. In *ICPP*, pages 656–659, 1992.

[17] R. Johnson, I. Pandis, and A. Ailamaki. Critical sections: Re-emerging scalability concerns for database storage engines. In *DaMoN*, 2008.

[18] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, 2009.

[19] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[20] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3(1-2):681–692, 2010.

[21] E. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, pages 603–614, 2010.

[22] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, pages 73–84, 2013.

[23] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[24] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[25] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *SIGMOD*, pages 691–706, 2015.

[26] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *TODS*, 6(2):213–226, 1981.

[27] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC*, 2012.

[28] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: A cache-partitioned hash table. In *PPoPP*, pages 319–320, 2012.

[29] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, pages 479–494, 2014.

[30] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *OSDI*, pages 511–524, 2014.

[31] I. Pandis, P. Tözün, M. Branco, D. Karampinas, D. Porobic, R. Johnson, and A. Ailamaki. A data-oriented transaction execution engine and supporting tools. In *SIGMOD*, pages 1237–1240, 2011.

[32] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page Latch-free Shared-everything OLTP. *PVLDB*, 4(10):610–621, 2011.

[33] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *SIGMOD*, pages 61–72, 2012.

[34] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. ATraPos: Adaptive Transaction Processing on Hardware Islands. In *ICDE*, 2014.

[35] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on Hardware Islands. *PVLDB*, 5(11):1447–1458, 2012.

[36] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core oltp under high contention. In *SIGMOD*, pages 1583–1598, 2016.

[37] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *PVLDB*, 6(2):145–156, 2012.

[38] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, pages 1311–1326, 2015.

[39] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross. Reducing database locking contention through multi-version concurrency. *PVLDB*, 7(13):1331–1342, 2014.

[40] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *EuroSys*, pages 17–30, 2011.

[41] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[42] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, pages 18–32, 2013.

[43] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.

[44] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *SIGMOD*, pages 1643–1658, 2016.

[45] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7):781–792, 2017.

[46] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *SIGMOD*, pages 1689–1704, 2016.

[47] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[48] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *SIGMOD*, pages 1629–1642, 2016.