Design Principles for Scaling Multi-core OLTP Under High Contention

Kun Ren
Yale University
kun.ren@yale.edu

Jose M. Faleiro
Yale University
jose.faleiro@yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

Although significant recent progress has been made in improving the multi-core scalability of high throughput transactional database systems, modern systems still fail to achieve scalable throughput for workloads involving frequent access to highly contended data. Most of this inability to achieve high throughput is explained by the fundamental constraints involved in guaranteeing ACID — the addition of cores results in more concurrent transactions accessing the same contended data for which access must be serialized in order to guarantee isolation. Thus, linear scalability for contended workloads is impossible. However, there exist flaws in many modern architectures that exacerbate their poor scalability, and result in throughput that is much worse than fundamentally required by the workload.

In this paper we identify two prevalent design principles that limit the multi-core scalability of many (but not all) transactional database systems on contended workloads: the multi-purpose nature of execution threads in these systems, and the lack of advanced planning of data access. We demonstrate the deleterious results of these design principles by implementing a prototype system, ORTHRUS, that is motivated by the principles of separation of database component functionality and advanced planning of transactions. We find that these two principles alone result in significantly improved scalability on high-contention workloads, and an order of magnitude increase in throughput for a non-trivial subset of these contended workloads.

1. INTRODUCTION

The maximum throughput that a database system is able to achieve is dependent on many factors, from the hardware on which it runs to the particular implementation details of the database software. While most of these factors can be overcome by spending more money, on better hardware or better software developers, throughput will always be fundamentally limited by the presence of *contended operations* in a workload.

The definition of a "contended operation" may vary depending on the user's requested isolation level of transactions, the ability of the database to prevent reads from conflicting with writes via multi-versioning and the semantic commutativity of operations. Nonetheless, unless no isolation whatsoever is required, there will always be certain operations that cannot be executed concurrently and the presence of many of these contended operations in a workload will necessarily limit throughput. Thus, adding more processors to a system, which enables more transactions to be processed in parallel, only increases throughput if the additional transactions that can be run do not conflict with the existing transactions that are currently running.

For decades, database systems had been designed for single processor machines. These conventional database architectures were ill suited for the abundant parallelism in multi-core hardware. As a consequence, they could not achieve scalable throughput across the entire spectrum of transactional workloads. Particularly problematic was the fact that conventional database architectures were not able to scale throughput on *low contention* workloads, despite the fact that low contention workloads have no fundamental limit to scalability. To address this gap between hardware and database software, most work on multi-core database systems has focused on eliminating fundamental scalability bottlenecks in these systems's design [21,23,48]. As a result of this research, today's state-of-the-art systems can achieve close to linear scalability in transactional throughput on low contention workloads.

Unfortunately, as recently demonstrated by Yu et al., multi-core database systems continue to be plagued by performance problems on *high contention* workloads [53]. For the fundamental reason described above, it is impossible to achieve linear scalability in high contention workloads; the throughput of a database system should taper as cores are added under high contention. However, the *actual* shortfall relative to linear scalability on high contention workloads is much worse than what is theoretically required by the nature of the contention in the workload. In some cases, in fact, throughput actually *decreases* as more cores are added. The problem is that the overhead of managing transactions, particularly that of concurrency control, is proportional to the amount of workload contention. At high levels of contention, concurrency control overhead takes up a non-trivial fraction of each transaction's total execution time. As a consequence, modern multi-core database systems achieve nowhere near the theoretical performance determined by the achievable concurrency in high contention workloads.

We attribute this poor performance under high contention to two design decisions that, to the best of our knowledge, are present in every widely-used transactional database system available today. First, despite compelling proposals to the contrary [17,37,38], database systems tend to assign responsibility for a particular transaction to a single thread [18]. Assigning a transaction to a single thread *conflates database functionality*, which leads to poor instruction and data cache locality [17]. Worse, this conflated functionality causes workload contention to directly impact physical contention in the database system (§2.1).

Second, database systems allow dynamic access of data without advanced planning of transactions' data access patterns. The negative side-effects of this design decision are most clearly present in database systems that use pessimistic concurrency control pro-

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882958

tocols based on logical locking. Such systems dynamically acquire each transaction's logical locks in an arbitrary order, which makes transactions susceptible to deadlocks. Any system which employs dynamic lock acquisition must therefore include a mechanism for handling deadlocks. Under high contention workloads, deadlock handling mechanisms are a significant source of overhead and can lead to wasted work due to transaction aborts (§2.2).

This paper proposes two design principles to address these overheads. First, we propose that database systems *partition functionality* across the cores of a single machine. Instead of using a single thread to perform both a transaction's logic *and* concurrency control on behalf of the transaction, we dedicate a set of threads to perform *only* concurrency control, and another set of threads to execute transaction logic. Concurrency control and execution threads cooperatively process transactions using explicit message-passing.

Second, we propose that database systems analyze transactions prior to their execution in order to predict their access patterns. These access patterns are used to coordinate access to data. Since almost all widely-used database systems use pessimistic locking to protect at least some types of data access, our focus in this paper is particularly within the context of pessimistic locking protocols. In this context, planning data access enables the implementation of a deadlock avoidance protocol, which circumvents the overhead of deadlock detection and resolution.

We built a prototype database system, ORTHRUS, based on the design principles of partitioned functionality and deadlock freedom. A notable aspect of ORTHRUS's design is the use of message passing between cores devoted to different functionalities. Although the use of explicit message-passing among a system's cores has been used in the context of multi-core operating systems [2,51] and programming models [5,33], to the best of our knowledge, ORTHRUS is the first multi-core database system to successfully use explicit message-passing to avoid physical contention on shared concurrency control data structures in the database system.

To summarize, the contributions of this paper are as follows:

- We identify two sources of overhead in state-of-the-art multi-core database systems that severely limit throughput under high contention workloads: conflated functionality (§2.1) and dynamic data access (§2.2).
- We propose two design principles to address this overhead; partitioning database functionality (§3.1) and planned data access (§3.2).
- Based on these design principles, we implement a prototype database system, ORTHRUS. We discuss techniques to make our design principles practical to implement (§3.3 and §3.4).
- We perform an extensive set of experiments in order to evaluate the multi-core scalability of ORTHRUS relative to an archetypal modern multi-core database system (§4).

2. PROBLEMS WITH EXISTING DESIGNS

2.1 Conflated functionality

As mentioned above, in most database systems, a single thread processes an individual transaction [18]. This single thread manages both the execution of the transaction's logic and the necessary interactions with the concurrency control module of the database system (e.g. a lock manager or the shared data structures needed for OCC validation). Several such threads concurrently execute on a single multi-core system. These concurrently executing threads make requests of the same concurrency control module[1]. This design pattern of multiple threads globally sharing a single concurrency control module can lead to severe scalability bottlenecks. We discuss these bottlenecks in this section.

Synchronization overhead. We first discuss the overhead associated with synchronization on concurrency control meta-data. In order to control the interleaving of concurrent transactions, any concurrency control protocol must associate some meta-data with the database's logical entities. The meta-data used is protocol dependent. For example, pessimistic locking protocols use a hash-table of lock requests on records [15], while optimistic and multi-version protocols associate timestamps with records [29, 48]. As part of the concurrency control protocol, several concurrent threads may need to read or write the meta-data associated with a particular database object. Concurrent threads must therefore synchronize their access to concurrency control meta-data. Note that synchronization is *not* implementation dependent; instead, it is intrinsic to any concurrency control protocol whose meta-data can be read or written by any database thread. Since meta-data is associated with database objects (such as records), contention for concurrency control meta-data is directly affected by workload contention; if a particular database record is popular, then threads will need to frequently synchronize on that record's meta-data. Unfortunately, on modern multi-core hardware, contention significantly degrades the performance of *atomic instructions* [4, 12, 13, 48]. These atomic instructions – such as *fetch-and-increment* and *compare-and-swap* – are the basic building blocks of both latch based and latch-free algorithms. Thus, under contention, both classes of algorithms are susceptible to severe performance degradation.

Data movement overhead. In addition to synchronization on concurrency control meta-data, another source of overhead in conventional database architectures is the *movement* of this meta-data across multiple cores. In order to access an object's meta-data, a thread must move the memory words corresponding to the meta-data into its CPU core's local cache. As multiple threads request access to a particular object's meta-data, the memory words corresponding to the meta-data are moved between cores. The movement of data between a machine's cores occurs because multiple threads are allowed to read or write the data. If a thread requires access to a latch-protected data-structure, the thread must first acquire the latch and then move the data-structure to its core. Only when these two steps complete can the thread actually access the data-structure. Since the latch is acquired first, it is held for the time it takes to move the data-structure. As a consequence, data-movement extends the duration for which latches are held. In the presence of contention, the increase in latch hold times can contribute to a decrease in concurrency.

In order to validate the deleterious effects of synchronization and data movement overhead, we ran a simple experiment to measure the scalability of short read-only transactions under two-phase locking on a high contention workload (see Appendix A.1 for a detailed description of the experimental setup). Since transactions are read-only, the workload is conflict free (despite the presence of contention). Figure 1 shows the results of the experiment. Two-phase

[1]The discussion that follows assumes a shared-memory architecture where all threads have access to the same shared memory where the concurrency control data structures sit. It should be noted that some database systems, such as H-Store [44] and Hyper [24], use a shared-nothing architecture across threads. Such systems do not suffer from the overheads associated with conflated functionality discussed in this section. However, they introduce other performance problems — namely agreement protocols across threads necessary to handle transactions that access data in multiple separate partitions. We explore this further in §4.

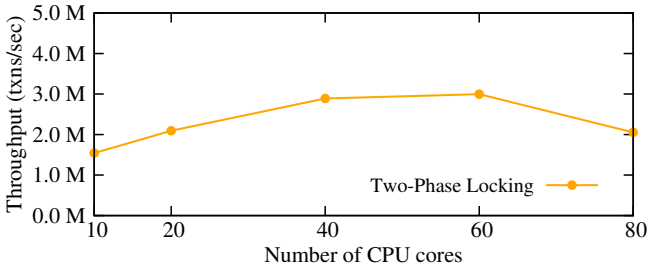

Figure 1: Scalability of read-only transactions under two-phase locking on a high contention workload.

locking is unable to scale beyond 40 cores despite the absence of conflicts and surprisingly *decreases* in performance. The inability to scale is due to synchronization and data movement overhead. The source of synchronization overhead is two-phase locking's use of atomic instructions on contended memory words to manipulate the table of lock requests. Data movement overhead is a consequence of multiple cores manipulating the same list of lock requests (due to multiple cores requesting read locks on the same records).

Instruction and data cache pollution. If a single thread executes both concurrency control and transaction logic, then the thread's CPU core must cache data and instructions corresponding to each of these two functions. The data and instructions corresponding to concurrency control, and transaction logic thus compete for a single core's cache. Concurrency control cache-lines therefore evicts transaction logic cache-lines, and vice-versa. This has the overall effect of increasing the duration of each transaction, which in turn decreases overall throughput.

All three reasons for performance degradation — synchronization overhead, data movement overhead, and cache pollution — have the effect of increasing a transaction's execution time. Not only does this increase in execution time per transaction necessarily reduce throughput by occupying system resources for longer periods of time, but the throughput reduction is compounded by the fact that increasing transaction time is particularly harmful in high contention settings. This is because conflicting transactions either have to block (in pessimistic schemes) or abort (in optimistic schemes). The longer it takes to execute a transaction, the higher the probability that a later conflicting transaction will abort or block behind the original transaction.

2.2 Dynamic concurrency control

Most database systems allow transactions to dynamically request records to read and write as they execute. This lack of advanced planning precludes opportunities to coordinate access to contended items in order to maximize concurrency. The clearest example of this are in systems that use two-phase locking (2PL) for concurrency control, where locks are acquired for a transaction as each data access request is processed. Allowing transactions to dynamically request access to records necessitates dynamic lock acquisition, which can lead to deadlocks. These systems must therefore implement mechanisms to deal with deadlocks.

Under high contention workloads, deadlock handling logic is a significant source of overhead. There are two reasons for this overhead: first, deadlock handling logic extends the duration for which locks are held; second, deadlocks waste useful work performed by transactions that must be aborted.

- Since a transaction cannot deadlock unless it has already acquired locks, any deadlock handling logic must be executed while locks are held. Therefore, deadlock handling logic extends the duration for which locks are held. The increased lock hold time means that conflicting transactions must wait longer to execute. Therefore, deadlock handling logic imposes a performance penalty regardless of whether a deadlock has actually occurred.
- If deadlock handling logic aborts a transaction, then any work performed by the transaction is wasted. Furthermore, if a transaction is allowed to directly write records (i.e., if transactions do not buffer their writes), then the database must also undo the aborted transaction's writes. In addition to wasted work, aborted transactions induce unnecessary waiting on conflicting transactions; if a conflicting transaction is made to wait on a transaction that is eventually aborted, then in retrospect the conflicting transaction could have been allowed to make progress without delay. Finally, under high contention workloads, deadlocks are simply more prevalent. Therefore, the wasted work and unnecessary waiting due to transaction aborts occur more often under such workloads.

3. ARCHITECTURE

In this section, we describe the architecture of a proof-of-concept system that we built, ORTHRUS. ORTHRUS is designed to ameliorate the scalability bottlenecks described in §2. ORTHRUS is not a complete database system — we did not build a relational query processor, a client communications manager, or many of the shared utilities that are present in most database systems. Instead, we just built the transaction management component of the system, with a particular focus on concurrency control, since the main impediment to scalability under high contention is concurrency control.

ORTHRUS implements locking-based pessimistic concurrency control. We choose a pessimistic scheme because ORTHRUS is targeted at workloads with high contention, and optimistic schemes are well-known to perform poorly under high contention due to excessive aborts — even recent proposals for multi-core optimized optimistic schemes (such as Silo [48] and Hekaton [29]) perform poorly under high contention [13].

Like most recently proposed architectures for transactional database systems, ORTHRUS assumes that the working set of data accessed by transactions can be held in main memory, since memory sizes are growing faster than transactional working sets [44]. As a consequence, ORTHRUS does not incur stalls to access data from disk. ORTHRUS therefore creates exactly the same number of threads as physical CPU cores and pins each thread to a single core, as is done in several other main-memory database systems [13, 14, 24, 34, 37, 38, 41, 44, 48, 53].

ORTHRUS's first key design feature is that it *partitions functionality* across the cores of a single machine. The thread pinned to each core on the server is devoted to a single narrow component of transaction processing. Since our focus in this paper is on concurrency control, ORTHRUS assigns cores one of two possible roles; concurrency control or transaction execution. Note, however, that this philosophy can extend to other roles within a database system. For instance, StagedDB successfully separated functionality across cores in the query processor component of the database system [17]. In ORTHRUS, concurrency control and transaction execution cores do not share any data-structures; concurrency control cores cannot read or write any data-structures on execution cores, and vice-versa. Instead of implicitly communicating through shared-memory, ORTHRUS uses explicit message-passing between concurrency control and execution threads.

The second key design feature of ORTHRUS is data access planning for the purpose of deadlock avoidance. In the rest of this section, we discuss these two key features of ORTHRUS's architecture in more detail.

3.1 Partitioned functionality

Logically, concurrency control threads perform the same function as that of a centralized lock manager. ORTHRUS partitions responsibility for database objects across concurrency control threads such that each database object is controlled by single thread. Thus, each concurrency control thread maintains meta-data for a disjoint subset of the database's objects. The meta-data on each concurrency control thread is exactly the same as in a centralized lock manager; each thread maintains a hash-table which maps keys to a list of transaction lock requests.

Execution threads do not contain instructions nor data pertaining to concurrency control; they are only responsible for performing each transaction's logic. Each transaction is assigned to a single execution thread, which is responsible for processing the transaction's logic in its entirety. Execution threads request locks by sending messages to the concurrency control threads responsible for those locks[2]. Message passing between execution threads and concurrency control threads is mediated via queues. Space is allocated in shared memory for an input queue to each concurrency control thread and an output queue from each concurrency control thread. Messages are passed via reads and writes to these queues.

Note that a naïve implementation of these queues leads to the same type of synchronization bottlenecks observed in traditional systems. If every transaction execution thread is allowed to write to a single input queue (for a particular concurrency control thread), then synchronization overhead will prevent a scalable implementation of message passing. To overcome this pitfall, our implementation assigns each concurrency control thread a *separate* queue for each execution thread. Thus, while we mention a single *logical* input queue to each concurrency control thread, its implementation consists of N physical queues, where N is the number of execution threads.

With the above implementation, each queue has only one writer (the associated execution thread) and one reader (the associated concurrency control thread). The queue can be therefore implemented using a standard latch-free circular buffer [33] to avoid synchronization between the reader and writer except in the rare case where the queue fills up. Consequently, our message passing implementation does not suffer from the synchronization costs that contended writes to shared-memory usually encounter.

Concurrency control threads serially process requests from their logical input queues. On dequeuing a request for locks from its input queue, a concurrency control thread checks the requested set of locks, and determines which of these locks are requested on objects in its local partition. For each lock on an object in its logical partition, the concurrency control inserts a lock request into its local hash-table. The concurrency control thread responds to an acquisition request only after it has granted all locks from the request of an individual transaction. Note that the response may take a while; the lock acquisition request may have to wait for prior conflicting requests to release locks (just as in conventional locking protocols). Instead of waiting for responses from concurrency control threads, execution threads begin working on other transactions. The interaction between execution and concurrency control threads during lock release is similar; the only difference is that concurrency control threads respond immediately because lock release requests are satisfied immediately.

Figure 2 shows an example of the interaction between a concurrency control thread and an execution thread. Transaction T_1's

[2]Since responsibility for objects is disjointedly partitioned across concurrency control threads, the set of locks required by a single transaction may reside on multiple concurrency control threads.

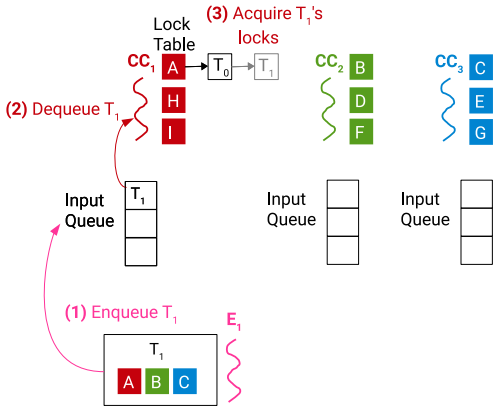

Figure 2: Concurrency control thread acquiring locks on behalf of an execution thread.

execution thread, E_1, requests locks on records A, B, and C (see §3.2 for why multiple requests are made together; for simplicity, we omit the details of the *lock mode* required on each of these records). E_1 writes these lock requests in a message that we label with name of the transaction (T_1). In order to acquire a lock on A, E_1 enqueues message T_1 in CC_1's input queue (**Step 1**). CC_1 dequeues T_1 from its input queue (**Step 2**) and checks the locks requested by T_1. In this particular example, of the locks requested by T_1 (A, B, and C), only A resides on CC_1's partition. CC_1 thus inserts a lock request on record A into its local lock table (**Step 3**). The details of how locks B and C are acquired, and how these requests end up in the input queues of the other concurrency control threads will be explained in §3.3.

The communication pattern between execution and concurrency control threads is much like client-server communication in distributed systems; execution threads behave as clients, and concurrency control threads behave as servers. An execution thread's "request" is an explicit message asking the concurrency control thread to acquire or release a lock on behalf of a particular transaction. While centralized lock managers and ORTHRUS's concurrency control threads perform the same logical function (acquire and release logical locks), they each use two fundamentally different communication mechanisms. In conventional database systems, threads use the abstraction of a shared-address space to implicitly share data. Intuitively, data is shared *by default*. Since data is globally shared, each database thread can directly manipulate the data. In contrast, in ORTHRUS, the default mode of operation is to *eliminate* shared data among threads of different types; execution and concurrency control threads do not share any data. In order to acquire and release locks on database objects, an execution thread must request concurrency control threads to acquire locks on its behalf.

As a result of its physical separation of concurrency control and transaction execution concerns across threads, ORTHRUS guarantees that data is *never* implicitly shared across concurrency control and execution threads; the only way to share data is via message-passing. The consequence of this design principle is that ORTHRUS completely eliminates all overheads associated with conflated functionality (§2.1):

- **Synchronization overhead.** ORTHRUS partitions database objects across concurrency control threads. As a consequence, every lock acquisition and release request for a particular object is serviced by a single concurrency control thread; reads and writes of an object's meta-data are restricted to one thread. Therefore, this thread does not need to synchronize its access to any of the

objects' meta-data it is responsible for. If multiple execution threads concurrently request locks on the same record, then the requests are handled by the same concurrency control thread.

- **Data movement overhead.** Since the locking operations on a particular database object are performed by a single thread, concurrency control meta-data (linked-lists of lock requests in the lock table) never need to move between threads. As a result, ORTHRUS does not suffer from any data movement overhead (for concurrency control).
- **Instruction and data cache pollution.** In ORTHRUS, execution and concurrency control threads perform two different functions. Execution threads are completely isolated from concurrency control threads; these two types of threads do not share any instructions nor data. As a consequence, ORTHRUS avoids the cache pollution that is inherent in conventional database implementations.

3.2 Deadlock avoidance

§2.2 discussed the various overheads associated with handling deadlocks in pessimistic locking-based concurrency control protocols. The fundamental problem behind the cost of deadlocks is the dynamic nature of data access, which in turn stems from a lack of advanced planning. Motivated by this observation, ORTHRUS plans data access prior to transaction execution, and leverages this advanced planning to perform a deadlock avoidance protocol instead of deadlock detection and resolution.

In more detail, ORTHRUS guarantees that deadlocks never occur by enforcing a locking discipline on execution threads. Execution threads must acquire locks on behalf of transactions in some well-defined order. To enforce this locking discipline, an execution thread cannot start to make lock requests (by sending messages to concurrency control threads) until it knows the complete set of locks requests that it will make for a particular transaction. Once an execution thread knows the complete set of locks requests, it can request locks from the appropriate concurrency control threads in order of the threads' unique identifiers. In order to avoid deadlock, these requests to concurrency control threads cannot occur concurrently — locks are requested from the next concurrency control thread after the locks from the previous thread have been granted.

In some cases, the set of locks that a transaction will request cannot be determined via a simple inspection of the transaction logic. Rather, there is a data-dependent access such that a part of the transaction needs to be executed before it can be determined what locks will be requested. In such a situation ORTHRUS uses the OLLP technique proposed by Thomson et al. in the context of Calvin [46]. In OLLP, a transaction is partially executed in "reconnaissance" mode in order to generate an estimate of the access footprint of the transaction. No locks are acquired during this reconnaissance, so no writes to the database state occur, and all reads are not assumed to be consistent (this is why the resulting access footprint is just an estimate and not a guarantee). This access estimate is then annotated as part of the transaction, which is submitted to the system to be actually run. Before starting to process a transaction, ORTHRUS acquires the locks that are indicated by the access estimate annotation. If, over the course of processing the transaction, an execution thread notices that it needs to access a record that it did not acquire a lock for at the beginning of the transaction (that is, the access estimate it received was incorrect), ORTHRUS updates the annotation, and then aborts and restarts the transaction with a new estimate. Ren et al. have shown that the extra overhead of OLLP (the reconnaissance phase) is generally a small percentage of actual transaction processing and that aborts due to incorrect access estimates are rare in practice [42].

A disadvantage of ORTHRUS's approach relative to dynamic lock acquisition is that there is a risk of locks being held for a longer period of time. This is because ORTHRUS immediately acquires all locks at the beginning of a transaction, while dynamic locking can acquire one lock at a time, interleaving lock acquisition with transaction execution. For example, if a lock on a highly contended record is only needed for the last operation of a transaction, dynamic lock acquisition only needs to hold the lock for a very short period of time (just the end of the transaction). Meanwhile ORTHRUS must hold the lock for the entire transaction.

Note that increased lock hold time is not a disadvantage under low contention. Increased lock hold time only hurts performance under high contention workloads. However, it is precisely under high contention that deadlocks are more frequent, and the cost of deadlock detection increases (§2.2). We hypothesize that ORTHRUS's increased lock hold time is more than offset by avoiding the decrease in concurrency and wasted work due to deadlock detection and aborts, respectively, in dynamic locking. We experiment with this hypothesis in §4.

3.3 Optimizations

ORTHRUS's design philosophy of partitioning concurrency control and execution functionality across the threads of a database server addresses several sources of overhead in conventional database systems. However, since concurrency control and execution threads communicate with each other via explicit message-passing, ORTHRUS introduces new sources of overhead. The single biggest source of overhead we had to overcome was that of *asynchrony* in the interactions between execution and concurrency control threads.

In ORTHRUS, concurrency control threads run a tight loop which sequentially processes requests for lock acquisition or release. At any given point in time, a concurrency control thread may have multiple outstanding lock acquisition or release requests. Thus, execution threads' requests may experience queuing delay before being processed. To prevent these queuing delays from wasting CPU cycles, execution threads do not synchronously wait on responses from concurrency control threads. After sending a request to a concurrency control thread, an execution thread checks whether any older requests have received responses. If yes, the execution thread resumes the execution of the corresponding transaction. If not, the execution thread begins executing a new transaction. Consequently, when a concurrency control thread does eventually respond to a lock request, the execution thread will likely be in the middle of working on a different transaction, and does not resume the original transaction immediately. Execution threads' lock acquisition and release requests are therefore *asynchronous*.

On its own, asynchrony is not a source of overhead. However, the interaction of asynchrony with the higher level concept of logical locking can hamper concurrency. The queuing delays experienced by requests and responses extend the duration for which logical locks are held. For instance, if an execution thread requests a lock on record A for transaction T, the time between when the lock is acquired by the appropriate concurrency control thread, and the time the execution thread resumes the execution of T represents time for which the lock on A is needlessly held. Furthermore, the overhead due to queuing delay is compounded when an execution thread requests locks from multiple concurrency control threads. This is because our deadlock avoidance mechanism requires that lock requests to these concurrency control threads occur sequentially (as explained in §3.2). For instance, if the execution thread from the example above subsequently requests locks on B and C, the lock on A is held while requests and responses on B and C experience queuing delays.

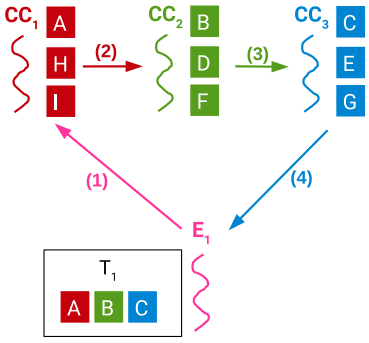

Figure 3: Concurrency control thread acquiring locks on behalf of an execution thread.

In order to reduce the impact of queuing delay, ORTHRUS minimizes the number of messages sent between threads. The basic idea is to allow concurrency control threads to forward lock request messages to other concurrency control threads on behalf of an execution thread. In other words, instead of paying two message delays for all concurrency control threads involved in a transaction, each concurrency control thread forwards the request directly to the next thread involved in that transaction (which reduces the number of message delays per concurrency control thread to one).

Figure 3 shows an example of how multiple locks are acquired. Transaction T_1 requires locks on records A, B, and C. As mentioned in §3.2, to avoid deadlocks, ORTHRUS requires all transactions to request locks from concurrency control threads in a well-defined order. In Figure 3, this order is CC_1, followed by CC_2, followed by CC_3[3].

Execution threads request the first of these concurrency control threads to acquire locks on its partition. E_1 therefore requests CC_1 to acquire T_1's locks (**Step 1**). For every lock required on later concurrency control threads, the concurrency control thread itself forwards the request. For instance, in order to acquire a lock on record B, CC_1 forwards T_1 to CC_2 (**Step 2**). Similarly, after acquiring a lock on record B, CC_2 forwards T_1 to CC_3 (**Step 3**). The last concurrency control thread that needs to acquire a lock on a transaction returns the transaction back to the execution thread. In this case, CC_3 returns T_1 to E_1 (**Step 4**).

If this optimization were not in place, E_1 would have to request locks to each concurrency control thread directly. First it would send a message to CC_1 to acquire a lock on A. CC_1 then responds to E_1 once the lock on A has been acquired by T_1. E_1 then repeats this process for each concurrency control thread. The number of messages per execution-concurrency control thread interaction would thus be two. Therefore, the total number of messages sent in the case of execution thread mediated lock acquisition is $2(N_{cc})$ (where N_{cc} is the number of concurrency control threads from which the execution thread requires locks). On the other hand, ORTHRUS's optimized lock acquisition procedure requires $N_{cc}+1$ messages (one message to each concurrency control thread, and one last message to the execution thread). The reduced number of messages is directly correlated with decreased waiting due to asynchrony.

Note that concurrency control threads may be subject to over- and under-utilization due to workload skew [39]. ORTHRUS can reuse prior techniques for addressing utilization imbalance in shared-nothing systems [6, 39, 45] in order to partition data among concurrency control threads.

[3]In Figure 3, each thread is assumed to have its own input queue (as in Figure 2). However, we do not show them in the figure.

3.4 Alternative architectures

In ORTHRUS, the set of database objects is partitioned across concurrency control threads. This design ensures that concurrency control threads do not share any data, thereby avoiding data movement and synchronization overhead (§2.1). Note, however, that partitioning objects across concurrency control threads is orthogonal to the design principle of separating functionality. ORTHRUS's use of partitioning is just one possible implementation of locking-based concurrency control.

A plausible alternative implementation would be to share a single lock table across all concurrency control threads. A single concurrency control thread could then obtain all the logical locks needed by a particular transaction. Execution threads could request any one of several concurrency control threads to acquire locks on its behalf. Although such an implementation would be subject to synchronization and data movement overhead, this synchronization is only across the concurrency control threads — a much smaller number of threads than the total number of threads in the system. Furthermore, the database system has the flexibility to limit the impact of these synchronization overheads. For example, the system could choose to assign concurrency control threads to execute on cores within a single NUMA socket.

Note that this flexibility to choose the number of threads to dedicate to each function is a direct consequence of the design principle of separating functionality. Conventional database systems do not have this flexibility because a single thread performs all the work entailed in executing a transaction. Furthermore, the separation of functionality enables a single system to support more than one implementation of a particular sub-system (such as the partitioned and non-partitioned lock table). Since components interact through narrow message-passing interfaces, the actual deployed implementation of a sub-system, such as concurrency control, can vary depending on system parameters, such as number of cores, number of NUMA sockets, and so forth.

4. EVALUATION

We run our experiments on a single 80-core machine, consisting of eight 10-core Intel E7-8850 processors and 128GB of memory. The operating system used is Ubuntu 14.04. All our experiments are performed in memory (none of our implementations utilize secondary storage). In all experiments, the number of threads used is equal to the number of cores; we pin a single long running thread to each CPU core (see §3.1).

We compare ORTHRUS against an implementation of two-phase locking (2PL) within the same ORTHRUS transaction management codebase. Our 2PL implementation uses a lock-table to store information about the locks acquired and requested by transactions. The lock-table is implemented as a hash-table. We implemented two important multi-core specific optimizations to improve the scalability of our 2PL implementation. First, the lock manager hash-table uses per-bucket latches instead of a single latch to protect the entire table. Per-bucket latches allow our 2PL implementation to avoid contention and overly conservative serialization on a single global latch. In addition to per-bucket latches, our 2PL implementation does not acquire high-level intention locks; transactions only acquire fine-grained logical locks on individual records. As a consequence, latch contention occurs only when multiple threads try to acquire or release logical locks on the same record. Second, our 2PL implementation *never* interacts with a memory allocator. Each database thread manually manages a pre-allocated thread-local pool of memory. Avoiding interaction with a memory allocator (such as malloc) removes superfluous synchronization in

the operating system's memory management logic and the memory allocator's logic. This allows us to isolate the sources of synchronization overhead to those in our own implementation.

To evaluate the overhead of deadlock handling in 2PL, we implement three different deadlock detection/avoidance mechanisms:

Wait-for graph. We use a graph to track the dependencies between transactions waiting to acquire logical locks, and the current holders of the lock. We only add edges to the wait-for graph if a transaction requests a lock, but finds that the lock is currently held in a conflicting mode by another transaction. The presence of a cycle in the wait-for graph implies that the transactions that constitute the cycle have deadlocked. In order to scale across multiple cores, our implementation avoids the use of a global latch to protect the entire graph. Instead, each database thread maintains a local partition of the wait-for graph, as is done by Yu et al. [53].

Wait die. Unlike the wait-for graph deadlock detection technique, which allows transactions to deadlock, and then detects and resolves deadlocks after the fact, wait die proactively avoids deadlocks by aborting transactions if they are suspected to be involved in a deadlock. In wait die, each transaction is assigned a timestamp prior to its execution, and the timestamp is used to determine whether or not the transaction is allowed to wait for a logical lock. If a transaction fails to immediately acquire a lock, then wait die only allows the transaction to wait on prior transactions if its timestamp is smaller than that of the current lock holder. If not, the transaction is aborted and restarted. Thus, wait die prioritizes older transactions (transactions with smaller timestamps), over younger transactions (transactions with larger timestamps). Each database thread uses the local timestamp counter on its CPU core to assign transactions their timestamps. Reading from the the core-local timestamp counter is low-overhead and contention-free. Core-local timestamp counters are therefore a cheap scalable source of monotonically increasing timestamps.

Dreadlocks. This state-of-the-art deadlock detection technique was proposed by Koskinen et al. in the context of mutual exclusion spin locks [26] and is used in the multi-core optimized version of the Shore database system (Shore-MT) [22]. Each transaction, T, maintains a digest, a data-structure which indicates the set of *other* transactions that T waits on for locks. Intuitively, a transaction's digest is a compact representation of its localized wait-for graph; T's digest contains the *transitive closure* of the transactions it waits for. If T fails to acquire a lock, T performs a set-union of its digest with the digest of the current lock holder. If T ever finds itself in its own digest, then it means that T's transitive closure contains a cycle, and a deadlock has occurred. Note that digests are amenable to a simple bitmap representation, and that a particular transaction's digest is always updated by the thread responsible for running the transaction. However, other threads can *read* the transaction's digest. As a consequence, updates to a transaction's digest can be performed without the use of latches [26].

We also include a version of 2PL that uses the deadlock avoidance protocol described in §3.2. We call this baseline *Deadlock-free locking*. This deadlock-free implementation analyzes each transaction prior to its execution in order to obtain its read- and write-sets and acquires locks in the lexicographical order in advance of transaction execution (as described in §3.2). Thus, we can compare our version of deadlock avoidance with three other widely-used techniques for handling deadlocks in 2PL systems.

4.1 Quantifying deadlock handling overhead

We begin our evaluation with experiments to show the overhead of the three deadlock handling mechanisms we implemented — wait-for graph, wait die, and dreadlocks — and compare them with

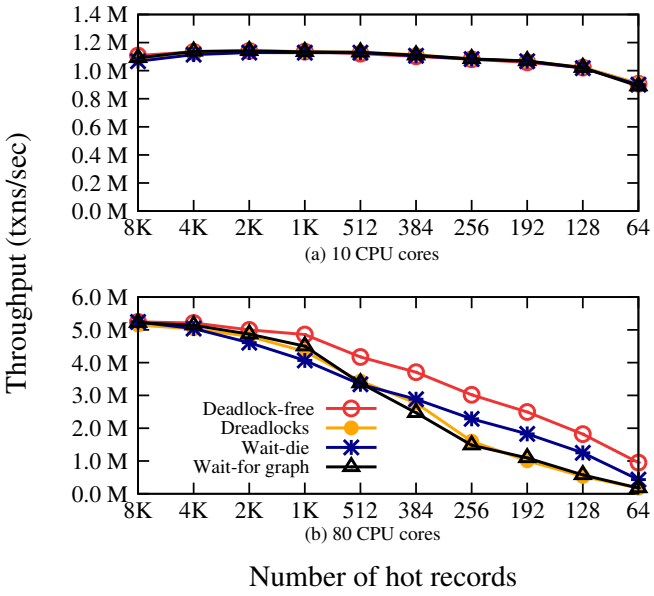

Figure 4: Throughput while varying the number of hot records in the database. (a) Number of database cores = 10. (b) Number of database cores = 80.

the simple, deadlock-free implementation of 2PL discussed above. We measure the throughput of these mechanisms under varying levels of contention.

Our benchmark uses a single table of 10,000,000 records. Each record's size is 1,000 bytes. The workload consists of transactions that each perform read-modify-write operations on ten records. Of the ten records updated by each transaction, we pick two records uniformly at random from a set of "hot" records, while the remaining eight are selected uniformly at random from a set of "cold" records. We vary the level of contention in the workload by varying the number of hot records. Intuitively, if transactions pick two records from a small set of hot records, then the probability that they will conflict is higher than if the set is large. We run two experiments, the first dedicates 10 CPU cores to the database system, while the second uses all 80 CPU cores of our test machine.

Figure 4(a) shows the throughput of each deadlock handling protocol when the database runs on 10 CPU cores. The number of hot records decreases as we move from left to right along the x-axis. As a consequence, contention *increases* from left to right. As expected, each system's throughput decreases with increasing contention, because as contention increases, the likelihood of conflicts between transactions increases. This, in turn, limits the amount of concurrency the database system is able to exploit. However, the relative difference in throughput between each deadlock handling mechanism is small.

Next, we perform the same experiment with 80 CPU cores. Figure 4(b) shows the results. Increasing the core count has two effects. First, it increases overall contention in the system since there are more active threads competing for the same logical data and internal data structures. Note that this is a different type of contention increase than the x-axis in the graphs which are generated via increasing logical contention by reducing the hot-set size. By increasing the number of active transactions, the overhead of the deadlock management schemes all increase as the data structures that they have to access (e.g., linked lists of waiting transactions) all become larger. (2) It causes the system to encounter NUMA effects as different cores are located in different NUMA sockets, increasing the cost of passing active data between cores (the 10 cores in

the previous experiment are located on a single socket). So, for example, the cost to acquire a latch on the data structures maintained by the deadlock handling schemes increases. Combined, these two effects (1) increase the overhead of the deadlock handling schemes, and (2) increase logical contention to the point where deadlock is more frequent, such that avoiding deadlocks a priori is superior to *dynamically* avoiding or detecting them.

The performance of the wait-for graph mirrors that of dreadlocks across the spectrum of contention. Both wait-for graph and dreadlocks effectively use the same algorithm to detect deadlocks — they only use different data-structures to represent the same information (a bitmap vs. an explicit dependency graph). Logically, however, the two algorithms are equivalent: both abort transactions upon detecting a cycle in their respective graph representations. Hence, both have poor performance at the right-hand side of the graph when logical contention is high, and deadlocks are frequent.

Wait-for graph and dreadlocks outperform wait die in the low to medium range of contention (the left-hand side of Figure 4(b)). This is because wait die suffers from false positives — it aborts transactions despite the absence of deadlocks. However, on the right-hand side of the graph, deadlocks are frequent, and the delay inherent in the dreadlocks and wait-for graph schemes, because they first detect deadlocks before resolving them, causes locks to be held by deadlocked transactions for longer periods of time before they abort, further exacerbating the effect of high contention. Thus, the wait-die approach of "giving up" early has the benefit of not allowing deadlocks to increase lock hold times.

Most importantly, Figure 4(b) indicates that *Deadlock-free locking always* outperforms all three deadlock handling mechanisms. There are two reasons for this. First, when deadlocks do not occur too often (in the middle of the graph), *Deadlock-free locking* outperforms the other schemes due to its low-overhead. *Deadlock-free locking* only has to analyze transactions's read- and write-sets in advance, and request locks in the correct order, while the other schemes are burdened by running deadlock handling logic (wait-for graph and dreadlocks) and aborting transactions due to false positives (wait die). Second, by eliminating the possibility of deadlocks, *Deadlock-free locking* does not suffer from aborts, and the subsequent wasted work due to retries. *Deadlock-free locking*'s advantage over deadlock handling techniques grows with increasing contention. At the right-most point in the graph, *Deadlock-free locking*'s throughput is 2.2x, 5.5x and 5.5x times that of wait die, dreadlocks and the wait-for graph, respectively.

Figure 4(b) validates the fact that under high contention workloads, deadlock handling logic is a significant impediment to multi-core throughput; dynamic deadlock handling techniques are *always* outperformed by *Deadlock-free locking*. Furthermore, the fact that throughput drops so drastically from 10 to 80 CPU cores indicates the problem will only get worse with increasing core counts.

Appendix B presents additional experiments on deadlock overhead on alternative workloads.

4.2 Tradeoffs in thread allocation

Until this point, we have experimented with just the benefit of deadlock avoidance in high contention scenarios. We now experiment with the full system design of ORTHRUS including its partitioned functionality.

ORTHRUS's partitioned functionality means that each database thread can be assigned one of two roles; concurrency control or transaction execution. Given a fixed number of threads, ORTHRUS must apportion threads to either concurrency control or execution. This section shows the performance implications of various concurrency control and execution thread allocations. We run one experiment in which 80 threads (corresponding to our test machine's 80 physical CPU cores) are made available to ORTHRUS. We experiment with multiple ORTHRUS configurations. In each configuration, we fix the number of concurrency control threads, and measure throughput while varying the number execution threads.

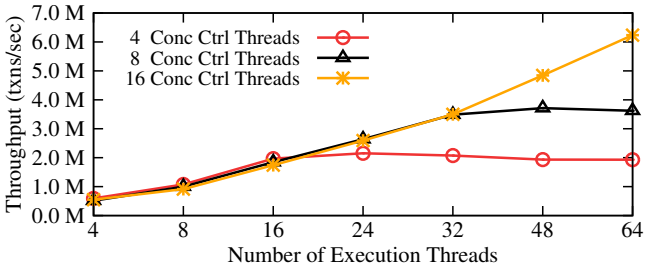

Figure 5: Execution thread scalability in ORTHRUS under various fixed concurrency control thread configurations.

We configure the database with a single table of 10,000,000 rows, each of size 1,000 bytes. The workload consists of transactions performing 10 read-modify-write operations on unique records. The records in transactions' read- and write-sets are selected uniformly at random from the set of 10,000,000 database rows. Each transaction acquires all its locks from a single concurrency control thread (we experiment with transactions that acquire locks on multiple concurrency control threads in §4.3, and §4.4).

Figure 5 shows the results of the experiment. We experiment with three concurrency control configurations, each corresponding to a curve in Figure 5. In each concurrency control configuration, throughput initially increases with increasing execution thread count. This is because on the left-hand-side of Figure 5, there are not enough execution threads to fully utilize the available concurrency control threads. Throughput continues increasing until a sufficient number execution threads saturate the available concurrency control threads. At this point, throughput plateaus. The point at which each curve plateaus is directly proportional to the number of concurrency control threads; more concurrency control threads can sustain a higher aggregate throughput than fewer concurrency control threads.

While ORTHRUS provides the flexibility to configure the number of concurrency control and execution threads, the choice of the optimal division of threads between concurrency control and execution is not obvious. Too few execution threads causes under-utilization of concurrency control threads, and vice-versa. Fortunately, ORTHRUS uses a staged event driven architecture (SEDA) [50]; ORTHRUS's concurrency control and execution modules correspond to SEDA stages communicating via explicit-message passing. Systems based on SEDA are amenable to dynamic allocation of resources (such as threads) based on load. In order to decide on the optimal allocation of threads between concurrency control and execution, therefore, ORTHRUS can use techniques for dynamic resource allocation on SEDA systems.

4.3 Multi-partition transactions

This section explores the cost of multi-partition transactions (transactions that need locks located on multiple concurrency control threads) in ORTHRUS. We compare performance against *Deadlock-free locking* (which has no partitioning whatsoever), and a fully-partitioned, "shared-nothing" system, where no memory is shared between partitions. Our implementation is based on the single-node (not distributed) version of the architecture of H-Store/VoltDB and HyPer [24,44]. This *Partitioned-store* baseline is similar to the corresponding implementation by Tu et al. in Silo [48]. *Partitioned-store* physically partitions data across database worker threads, such

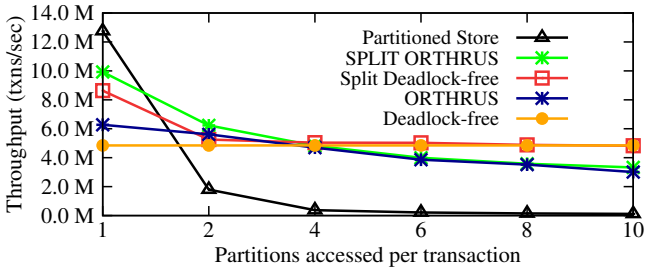

Figure 6: Performance of ORTHRUS and *Partitioned-store* as the number of partitions accessed per transaction is varied.

that each worker has its own local hash-table index. For concurrency control, *Partitioned-store* associates a coarse-grain partition-level spinlock with each worker. In order to execute a transaction, a worker thread obtains partition-level spinlocks on every partition that the transaction needs to access. If every transaction is single-partition, the corresponding workers will only acquire their own partition-level spinlocks. In the single-partition case, therefore, workers' lock acquisitions never conflict with each other. Furthermore, in the single-partition case, spinlock acquisition has minimal overhead because the lock is cached by the corresponding worker.

We run two experiments, both highlighting the effect of multi-partition transactions on performance. The first experiment compares the throughput of each system while varying the number of partitions accessed by each transaction. The second varies the fraction of multi-partition transactions in the workload.

The experiments in this section use a database which consists of a single table with 10,000,000 records of size 1,000 bytes each. These 10,000,000 records are uniformly spread across *Partitioned-store*'s physical partitions. Similarly, 10,000,000 logical locks are uniformly spread across ORTHRUS's concurrency control threads. Both experiments use transactions which perform 10 read-modify-write operations. In *Partitioned-store*, multi-partition transactions span physical partitions. Similarly, multi-partition transactions in ORTHRUS request locks from multiple concurrency control threads.

Vary partitions per transaction. Figure 6 shows the performance of ORTHRUS, *Partitioned-store* and *Deadlock-free locking* while varying the number of partitions accessed by each transaction. When transactions are restricted to a single partition, *Partitioned-store* outperforms ORTHRUS and *Deadlock-free locking*. There are two primary reasons for this. First, *Partitioned-store* requires no concurrency control when transactions are restricted to a single partition. Second, *Partitioned-store* index structures have better cache locality because indexes are physically partitioned across worker threads. As transactions access two or more partitions, however *Partitioned-store* experiences a sharp drop in throughput. *Partitioned-store*'s drop in throughput is due to its use of coarse-grain concurrency control. *Partitioned-store* isolates transactions at the level of partitions; a pair of transactions conflict if they both access the same partition. In contrast, ORTHRUS and *Deadlock-free locking* isolate transactions at the granularity of read- and write-conflicts on individual records.

We also find that ORTHRUS's throughput decreases as transactions access more partitions. However, unlike *Partitioned-store*, ORTHRUS's drop in throughput is more modest. ORTHRUS's throughput drops because of the increase in the number of messages required to acquire a transaction's locks. In the single-partition case, each transaction acquires its locks from a single concurrency control thread. However, as transactions' locks are distributed over a greater number of concurrency control threads, the number of message hops during lock acquisition increases. In particular, the number of messages required to acquire a transaction's locks is equal to $N_{cc} + 1$, where N_{cc} is the number of concurrency control threads on which a transaction's locks reside (§3.3). Clearly, as N_{cc} increases, the number of messages involved in acquiring a single transaction's locks increases. Figure 6 also shows that *Deadlock-free locking*'s throughput remains unchanged as the number of partitions accessed by a transaction increases. *Deadlock-free locking* is a shared-everything system, and hence is not subject to additional overhead in the presence of multi-partition transactions.

To better understand the performance characteristics of the curves in Figure 6, we physically partitioned indexes across ORTHRUS and *Deadlock-free locking*'s worker threads[4]. These curves are marked SPLIT ORTHRUS and *Split Deadlock-free*, respectively. This optimization puts all three systems on the same level as *Partitioned-store* with respect to cache locality, and any remaining difference between *Partitioned-store* and these two new curves can be attributed to concurrency control.

When analyzing the difference between *Partitioned-store* and SPLIT ORTHRUS/*Split Deadlock-free* and comparing this difference to the difference between *Partitioned-store* and ORTHRUS/*Deadlock-free locking*, we can conclude that when transactions are restricted to a single partition, *Partitioned-store*'s main advantage over ORTHRUS and *Deadlock-free locking* is due to the smaller cache footprint of partitioned indexes. *Partitioned-store*'s throughput is about 2x and 2.6x that of ORTHRUS and *Deadlock-free locking*, respectively, while its advantage over SPLIT ORTHRUS and *Split Deadlock-free* is a more modest 1.3x and 1.5x, respectively. Furthermore, as transactions access more partitions, the performance of the partitioned variants of ORTHRUS and *Deadlock-free locking* converge to their non-partitioned counterparts. This confirms that *Partitioned-store*'s poor performance under multi-partition transactions is due to its use of coarse-grain concurrency control.

It should be noted that a big advantage of ORTHRUS and *Deadlock-free locking* over *Partitioned-store* is that these systems by default do not require a user to be concerned about finding a near-perfect data partitioning such that the vast majority of a transactions in a workload will only access a single-partition. However, if a good or near-perfect partitioning is available for a particular workload, there is no reason why ORTHRUS and *Deadlock-free locking* cannot benefit from it by partitioning their indexes across worker threads accordingly. In other words, although our primary motivation for introducing SPLIT ORTHRUS and *Split Deadlock-free* was in order to break down the performance differences between *Partitioned-store* and ORTHRUS/*Deadlock-free locking*, if a good data partitioning is available for a workload, SPLIT ORTHRUS could be used instead of ORTHRUS, and achieve much closer performance to partitioned stores on single-partitioned transactions, while maintaining its significant advantages over partitioned-stores for multi-partition transactions.

Vary fraction of multi-partition transactions. Although Figure 6 shows the effect of multi-partition transactions on each system's throughput, realistic workloads involve a mix of single-partition and multi-partition transactions. In this experiment, we evaluate a workload consisting of both single- and multi-partition transactions. We vary the percentage of multi-partition transactions in the workload. Multi-partition transactions run on exactly two partitions.

Figure 7 shows the result of the experiment. As in the previous experiment, we find that *Partitioned-store* outperforms ORTHRUS and *Deadlock-free locking* when all transactions are single-partition (0% multi-partition transactions). *Partitioned-store*'s throughput decreases as the fraction of multi-partition transactions increases.

[4]Tu et al. perform a similar analysis in Silo [48]

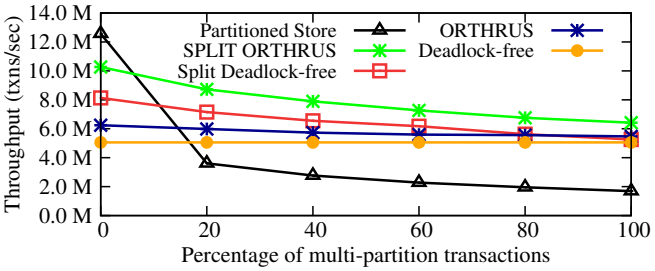

Figure 7: Performance of ORTHRUS and *Partitioned-store* as the percentage of multi-partition transactions is varied.

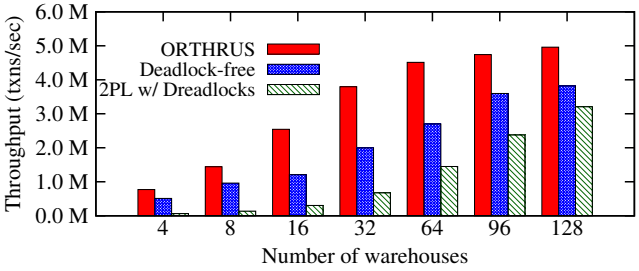

Figure 8: TPC-C NewOrder and Payment throughput, varying number of warehouses. Each system uses 80 CPU cores.

This is expected because we already saw in Figure 6 that *Partitioned-store*'s performance decreases dramatically as soon an transactions access more than one partition.

ORTHRUS's throughput also decreases as the fraction of multi-partition transactions increases. As elaborated in the previous experiment, this is due to the increase in the number of messages required to acquire transactions' locks. However, despite the decrease in throughput, ORTHRUS *always* outperforms *Deadlock-free locking* (even when the percentage of multi-partition transaction is 100%). Although both ORTHRUS and *Deadlock-free locking* use the same underlying locking-based protocol, ORTHRUS's partitioned functionality allows concurrency control and execution threads to better utilize data- and instruction-caches (§3.1).

Finally, Figure 7 also evaluates SPLIT ORTHRUS and *Split Deadlock-free*. These curves reinforce the fact that *Partitioned-store*'s main advantage relative to ORTHRUS and *Deadlock-free locking* is cache locality, and this advantage can be duplicated by ORTHRUS and *Deadlock-free locking* if a good data partitioning is available. Nonetheless, *Partitioned-store*'s total elimination of concurrency control for single-partition transactions prevents SPLIT ORTHRUS and *Split Deadlock-free* from matching its performance at the extreme left-hand side of the graph.

4.4 Performance under contention

This section evaluates the performance of ORTHRUS under contention. We compare ORTHRUS against *Deadlock-free locking*, and 2PL with dreadlocks. The experiments in this section run a subset of the TPC-C benchmark, while we experiment with the YCSB benchmark in Appendix A. As is common practice, our TPC-C implementation does not model client "think" time, and transactions are executed as one-shot stored procedures [1, 41, 44, 48]. We restrict our evaluation to TPC-C's NewOrder and Payment transactions. These two transactions make up the vast majority of the benchmark; approximately 45% and 43% of transactions in the full TPC-C mix are NewOrder and Payment transactions, respectively. Furthermore, NewOrder and Payment are short update transactions[5], and thereby put greater stress on concurrency control. Our evaluation therefore uses an equal mix of NewOrder and Payment transactions; both types of transaction are equally likely to occur.

TPC-C's database conforms to a tree-based schema [44]. Most tables in TPC-C have a foreign key dependency on the "root" Warehouse table. Excluding the Warehouse table itself, out of eight tables in the TPC-C database, only one, Item, does not contain a foreign key dependency on the Warehouse table. TPC-C's Item table is read-only. Hence, none of our baselines perform any concurrency control on reads to Item table's rows. In TPC-C, the $warehouse\_id$ attribute is the primary key of the Warehouse table, and foreign key in all other tables (apart from Item). ORTHRUS partitions database tables across concurrency control threads based on each row's $warehouse\_id$ attribute. Our evaluation adheres to the TPC-C specification of requiring 10% of NewOrder and 15% of Payment transactions to span two warehouses. Therefore, approximately 12.5% of transactions in our evaluation require locks from two concurrency control threads. Some Payment transactions' read- and write-sets are deducible only upon reading the value of a secondary index. In particular, 60% of Payment transactions must find a Customer by a secondary index on customers' last name. For this subset of transactions, ORTHRUS must speculatively read this secondary index in order to obtain the transaction's read- and write-sets using the OLLP protocol described in §3.2.

4.4.1 Throughput under varying contention

Since the TPC-C database schema is tree-based, and rooted at the Warehouse table, contention on all tables (except the read-only Items table) can be controlled by varying the number of records in the Warehouse table; decreasing the number of warehouses increases the level of contention in the workload. Figure 8 shows the throughput of each system while varying the number of warehouses. The number of warehouses increases from left to right along the x-axis, therefore, contention *decreases* from left to right.

When the number of warehouses is small, we find that ORTHRUS significantly outperforms 2PL. When the number of warehouses is small, both transactions update highly contended records. Payment updates two records, one each from the Warehouse and District tables. NewOrder updates a single District record. *Deadlock-free locking* suffers from latching overhead associated with acquiring logical locks on each of these records (synchronization overhead). Furthermore, the linked-list corresponding to a bucket in the lock table must also be moved across cores (data movement overhead).

In addition to suffering from the same sources of overhead as *Deadlock-free locking*, 2PL with dreadlocks must also execute deadlock handling logic, which further reduces throughput (see §4.1). Dreadlocks requires all transactions waiting on a particular logical lock to spin on the lock holder's digest. When the lock holder eventually releases the lock, it updates its digest, which propagates to the threads spinning on the digest. In order to read the new value of the digest, the readers' cached values of the digest must be invalidated and then reloaded with the new value. Note that this cache coherence overhead is *in addition* to the cache coherence overhead associated with latch acquisition and linked-list traversal experienced by *Deadlock-free locking*. Furthermore, we found that the actual occurrence of deadlocks on the TPC-C workload was rare. 2PL with Dreadlocks is thus subject to severe cache coherence overhead despite the fact that the workload itself is mostly deadlock free.

Unlike 2PL, ORTHRUS does not experience cache-coherence induced overhead. In ORTHRUS, a single concurrency control thread is responsible for processing every lock operation on a particular record. Concurrency control threads therefore require no synchronization to process lock acquisition or release requests. Further-

[5]Compared to long running transactions such as StockLevel

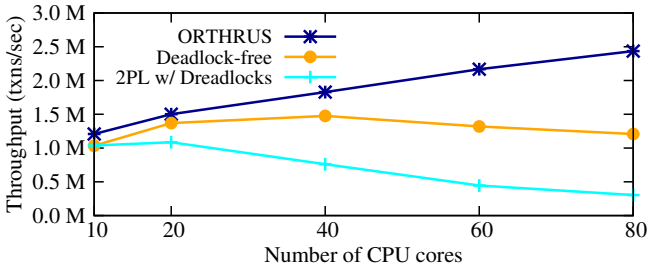

Figure 9: TPC-C throughput while increasing core count. Number of warehouses = 16.

more, the meta-data associated with the locks (the linked list of lock requests) does not need to move across cores. On the contrary, since Warehouse and District records are popular, the linked lists of lock requests on these records experience good cache locality on concurrency control threads. As mentioned previously, 10% of NewOrder and 15% of Payment transactions span two warehouses. Since we assign a single concurrency control thread to a particular warehouse, this subset of NewOrder and Payment transactions must interact with two concurrency control threads. These transactions are therefore subject to greater lock hold times on popular records due to asynchrony (§3.3). Despite the longer duration for which contended locks are held, ORTHRUS is able to significantly outperform both locking implementations.

As the number of warehouses increases, the level of contention in the workload decreases. As a consequence, both locking systems experiences lower synchronization overhead. However, despite the decrease in contention, we find that ORTHRUS maintains a significant advantage over both locking systems. At 128 warehouses, ORTHRUS's throughput is 1.3x and 1.5x that of *Deadlock-free locking* and 2PL, respectively. We attribute this remaining advantage to the lower instruction- and data-cache footprint entailed by partitioned functionality (§2.1 and §3.1).

4.4.2 Scalability under high contention

Next, we compare the scalability of ORTHRUS, *Deadlock-free locking* and 2PL under high contention. We set the total number of warehouses to 16 and vary the number of cores used by the system. Figure 9 shows the results of the experiment. When each system uses 10 CPU cores, we find that *Deadlock-free locking* and 2PL's throughput is identical (as explained in §4.1). On increasing the number of CPU cores, we find that 2PL's throughput begins to drop because of the overhead of deadlock handling logic. As mentioned above, deadlocks occur with negligible frequency on the TPC-C benchmark. However, despite the negligible frequency of deadlocks, we find a significant difference between 2PL and *Deadlock-free locking*; this difference validates our claim that deadlock handling logic is a significant source of overhead, even on workloads which are devoid of deadlocks. Despite the significant level of contention in the workload ORTHRUS's throughput scales with additional CPU cores. At 80 cores, ORTHRUS outperforms *Deadlock-free locking* and 2PL by 2x and nearly an *order of magnitude*, respectively.

4.4.3 Execution time breakdown

We conclude our experimental evaluation by showing the breakdown of CPU time on database execution threads in ORTHRUS, *Deadlock-free locking*, and 2PL. Figure 10 shows that 2PL spends significantly more time locking records than *Deadlock-free locking*. This is due to the fact the the dreadlocks algorithm spends time spinning on threads' digests in the lock manager. The difference between these two baselines is only that 2PL spends time waiting

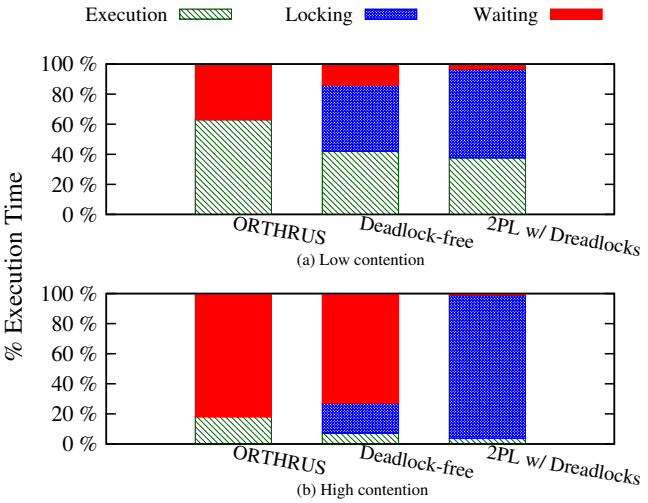

Figure 10: Execution thread CPU time breakdown on TPC-C with 80 threads. (a): 128 Warehouses (low contention). (b): 16 Warehouses (high contention)

within the lock manager, while *Deadlock-free locking* spends time waiting outside the lock manager.

ORTHRUS's execution threads spend more cycles doing useful work than both *Deadlock-free locking* and 2PL under both high contention and low contention. Under high contention all three systems spend a large fraction of their time waiting on locks. However, ORTHRUS's execution threads spend significantly more time executing transactions (in comparison to *Deadlock-free locking* and 2PL). In particular, ORTHRUS's execution threads spend 18% of their time executing transactions. In comparison, *Deadlock-free locking* and 2PL threads spend just 7.2% and 3.7% of their time doing useful work. Thus, ORTHRUS's execution thread utilization is about 2.5x and 5x greater than that *Deadlock-free locking* and 2PL, respectively. It should be noted that as a consequence of its partitioned functionality, ORTHRUS uses fewer execution threads than *Deadlock-free locking* and 2PL. In both experiments, ORTHRUS uses 16 concurrency control threads and 64 execution threads. In contrast, the other systems utilize all 80 threads for execution. However, despite having 20% fewer execution threads at its disposal, ORTHRUS outperforms *Deadlock-free locking* and 2PL by much better utilizing its available execution threads.

5. RELATED WORK

Synchronization. Remote Core Locking (RCL) is a technique for reducing synchronization overhead and increasing data locality in contended critical sections [33]. RCL classifies a subset of a machine's CPU cores as "server" cores, and assigns contended critical sections to server cores. Threads request server cores to execute critical sections on their behalf. Flat combining is a synchronization technique that addresses the same problem as RCL; the impact of contended critical sections on synchronization overhead and data locality [19]. Unlike RCL, flat combining does not dedicate CPU cores for the sole purpose of critical section execution. Instead, a single "combiner" thread is dynamically chosen to execute critical sections on behalf of others. Both RCL and flat combining use the design principle of dedicating a single thread to repeatedly execute critical sections for the purposes of exploiting instruction- and data-cache locality, and reducing synchronization overhead. ORTHRUS uses the same design principle; RCL's server cores, and flat combining's combiner are analogous to ORTHRUS's concurrency control threads. However, both RCL and flat com-

bining address high contention in critical sections without regard to any higher level functionality. In contrast, ORTHRUS's concurrency control threads implement logical locking, and must therefore carefully address harmful interactions between logical locking and asynchronous message-passing (§3.3).

Operating systems. Barrelfish is a multi-core operating system kernel that forbids shared-memory-based inter-core communication altogether [2]. Barrelfish forces cores to communicate using explicit message passing. One of the motivations for Barrelfish's deisgn was shared-memory operating system kernels's inability to reason about contention at scale. Wentzlaff et al. propose a factored operating system (fos), which assigns operating system functions to specific cores of a single machine [51]. fos's design is intended to reduce contention, and improve instruction- and data-cache locality. ORTHRUS shares some of the motivation behind Barrelfish and fos: synchronization overhead, and poor locality due to conflated functionality (§2.1). However, Barrelfish and fos address overheads in operating system kernels, while ORTHRUS addresses overheads in database concurrency control.

Partitioned functionality. Bernstein and Das, and Ding et al. proposed optimistic distributed DBMSs in which validation is performed by a dedicated set of processes, independent from those that execute transactions's logic [3, 11]. Faleiro et al. proposed techniques for lazy transaction evaluation [14], and multi-version concurrency control [13], that separate concurrency control and transaction execution. None of these systems advocated for explicit message-passing as an inter-thread communication mechanism on a single multi-core machine. Moreover, this paper analyzes the broader implications of separating concurrency control from transaction execution, while these prior systems used separate concurrency control and execution threads in the narrower context of distributed optimistic concurrency control, lazy transaction evaluation, and multi-version concurrency control.

Staged DBMSs. Harizopoulos et al. identified several sources of overhead with thread-based query execution architectures [17]. Chief among these was poor instruction- and data-cache locality due to thread context switching. To address this overhead, they proposed StagedDB, a staged event-driven query execution architecture. StagedDB uses long-lived threads to exploit inter-query instruction- and data-cache locality. ORTHRUS uses a similar design; it dedicates long-lived threads to perform concurrency control and transaction execution logic respectively. These threads are pinned to physical CPU cores in order to exploit instruction- and data-cache locality. ORTHRUS differs from StagedDB in that it addresses bottlenecks in transaction processing, not query execution. Furthermore, ORTHRUS addresses transaction processing overheads on modern multi-core machines, such as synchronization, data movement, and deadlock handling.

Several distributed DBMSs have adopted a staged event driven architecture (SEDA) [10, 20, 28, 54]. SEDA is a natural fit for distributed database systems, which must *necessarily* use message-passing as a communication mechanism. In contrast, ORTHRUS uses a staged message-passing among threads on a *single node*.

Multi-core optimized DBMSs. Several researchers have recently proposed techniques to address synchronization overhead and improve cache-locality in multi-core databases [25, 30–32, 35, 40, 47, 49]. Johnson et al. devised a technique to reduce the frequency of contended latch acquisitions in conventional lock managers [21]. Their technique, speculative lock inheritance, passes contended logical locks between transactions without requiring calls to the lock manager (consequently decreasing the frequency of contended latch acquisition within the lock manager). Jung et al. devised scalable latch-free algorithms for conventional lock managers [23]. Larson et al. address several scalability bottlenecks in both pessimistic and optimistic concurrency control protocols [29]. For instance, their optimistic validation protocol does not require the use of a global critical section (as required by conventional protocols [27]). Tu et al. propose Silo, an optimistic main-memory multi-core database system designed to eliminate contended centralized data-structures [48]. All of these systems use a conventional shared-memory design. In contrast, ORTHRUS mediates communication among concurrency control and execution threads using explicit message-passing.

Semantics-aware concurrency control. Several researchers have argued for reasoning about conflicts using the semantics of operations [1,7,8,36,43]. Doppel is a main-memory database system that exploits commutative operations on contended records [34]. Contended records are replicated across a machine's cores, and commutative operations on these records are satisfied by any core. In order to process non-commutative operations, the state on each replica is periodically aggregated. In contrast, ORTHRUS is designed to address high contention in workloads where semantic knowledge about conflicts is not available (beyond reads, and writes to records).

Shared-nothing systems. Prior research found that the cost of two-phase locking was prohibitively expensive on main-memory database systems [16]. Several researchers subsequently recommended doing away with two-phase locking altogether, and advocated for serial transaction execution. H-Store is an example of a database system which employs serial transaction execution, and works best when workloads are perfectly partitionable [44]. HyPer is another example of a single-thread transaction processing system, but its design is meant to simultaneously support OLTP and OLAP workloads [24] (note that the latest version of HyPer abandons a shared-nothing design in favor of shared-everyting optimistic multi-versioning [35]). Since these systems do away with concurrency control altogether, they do not suffer from any of the overheads described in this paper. However, for the same reason, they cannot adequately utilize multi-core systems when a workload contains a non-trivial fraction of distributed transactions.

Pandis et al. propose a shared-nothing transaction processing architecture to avoid contention on locking meta-data (DORA) [37] and indexes (PLP) [38]. Unlike conventional designs where a single thread performs a transaction's logic, their work proposes that a transaction is collectively processed by the partitions on which it must execute. A transaction's logic is broken into smaller sub-transactions such that an entire sub-transaction is restricted to a single partition. Unlike ORTHRUS, DORA assigns a single thread to perform both concurrency control and execution within a partition. Furthermore, this paper advocates for *partitioned functionality*, which does not necessarily preclude shared-data among concurrency control or execution threads (§3.4). PLP is complimentary to our work; we do not address index contention.

6. CONCLUSIONS

The vast majority of database systems adhere to the design principle of assigning a single thread the responsibility of performing all logic on behalf of a transaction. This design principle leads to severe scalability problems on main-memory multi-core databases due to synchronization overhead, data movement overhead, and cache pollution. Furthermore, these systems allow dynamic access of data which necessitates expensive deadlock handling mechanisms. ORTHRUS addresses these limitations by partitioning functionality across a machine's cores and eliminating the need for handling deadlocks. Our experimental evaluation shows that ORTHRUS's design enables it to outperform conventional database systems by upto an order of magnitude on high contention workloads.

Acknowledgements. This work was sponsored by the NSF under grant IIS-1527118. We thank the anonymous SIGMOD 2016 reviewers for their insightful feedback.

7. REFERENCES

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

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *SOSP*, 2009.

[3] P. A. Bernstein and S. Das. Scaling optimistic concurrency control by approximately partitioning the certifier and log. *DE Bull*, 38(1), 2015.

[4] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Linux OLS*, 2012.

[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*. 2013.

[6] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *PVLDB*, 5(11), 2012.

[7] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *SOSP*, 2013.

[8] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SoCC*, 2012.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[11] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *SoCC*, 2015.

[12] J. M. Faleiro and D. J. Abadi. FIT: A distributed database performance tradeoff. *DE Bull*, 38(1), 2015.

[13] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11), 2015.

[14] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, 2014.

[15] J. Gray and A. Reuter. *Transaction processing*. Morgan Kaufmann Publishers, 1992.

[16] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[17] S. Harizopoulos and A. Ailamaki. A case for staged database systems. In *CIDR*, 2003.

[18] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a database system*. Now Publishers, 2007.

[19] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[20] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of PIER: An internet-scale query processor. In *CIDR*, 2005.

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

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

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

[24] A. Kemper and T. Neumann. Hyper: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[25] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, 2015.

[26] E. Koskinen and M. Herlihy. Dreadlocks: Efficient Deadlock Detection. In *SPAA*, 2008.

[27] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2), 1981.

[28] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS OSR*, 44(2), 2010.

[29] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4), 2011.

[30] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, 2014.

[31] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. Multi-version range concurrency control in deuteronomy. *PVLDB*, 8(13), 2015.

[32] D. B. Lomet, S. Sengupta, and J. J. Levandoski. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, 2013.

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

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

[35] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, 2015.

[36] P. E. O'Neil. The escrow transactional method. *TODS*, 11(4), 1986.

[37] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2), 2010.

[38] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: page latch-free shared-everything oltp. *PVLDB*, 4(10), 2011.

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

[40] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. Oltp on hardware islands. *PVLDB*, 5(11), 2012.

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

[42] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *PVLDB*, 7(10), 2014.

[43] 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*, 2015.

[44] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.

[45] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *PVLDB*, 8(3), 2014.

[46] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[47] P. Tözün, I. Atta, A. Ailamaki, and A. Moshovos. Addict: Advanced instruction chasing for transactions. *PVLDB*, 7(14), 2014.

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

[49] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Eurosys*, 2014.

[50] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[51] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS OSR*, 43(2), 2009.

[52] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. *PVLDB*, 9(5), 2016.

[53] 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), 2014.

[54] L.-Y. Yuan, L. Wu, J.-H. You, and Y. Chi. Rubato DB: A highly scalable staged grid database system for OLTP and big data applications. In *CIKM*, 2014.

APPENDIX

A. YCSB EVALUATION

In this section, we compare ORTHRUS's performance against that of *Deadlock-free locking* and 2PL on the Yahoo! Cloud Serving Benchmark (YCSB) [9].

This set of experiments uses a single table of 10,000,000 records, each 1,000 bytes in size. Since YCSB transactions execute over a single table, the benchmark implicitly assumes a "flat" schema. This is in contrast to the *tree schema* from TPC-C that we experimented with in §4.4. The key difference between flat and tree schemas is that it is easier to partition data in a tree schema such that multi-partition transactions are rare.

To avoid assumptions about the partitionability of the data, we run ORTHRUS under three different configurations; single partition, dual partition, and random. In the single partition configuration all the locks required by a particular transaction are guaranteed to reside on a single concurrency control thread. The single partition configuration represents a perfectly partitionable tree schema. In the dual partition configuration, all the locks required by a transaction are guaranteed to reside on exactly two concurrency control threads. The dual partition configuration forces *every* transaction

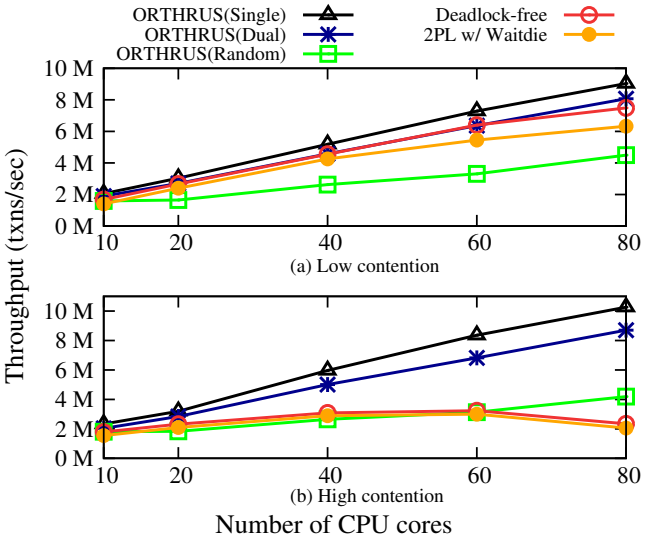

Figure 11: YCSB read-only transaction scalability.

to acquire locks from exactly two concurrency control threads. The dual partition configuration thus represents a workload in which *every* transaction is distributed due to the involvement of an ad-hoc pairing of real-world entities involved in the transaction. Under the random configuration, a transaction's locks are randomly distributed across concurrency control threads and could potentially access many more than two threads. This corresponds to complex transactions involving many real-world entities whose joint-involvement in transactions are not predictable in advance.

A.1 Read-only transactions

This experiment compares ORTHRUS's throughput against that of 2PL with wait-die[6] and *Deadlock-free locking* on a workload consisting of read-only transactions. Each transaction reads 10 records. We run both low and high contention experiments. In the low contention experiment, the 10 records in a transaction's read-set are selected uniformly at random from the set of 10,000,000 records. In the high contention experiment, each transaction picks two records from a set of hot records, and the remaining eight records from a set of cold records (locks on two hot records are acquired before locks on cold records). We set the number of hot records to 64. In both low, and high contention experiments, 2PL's deadlock handling logic is *never* invoked. This is because read-only transactions do not conflict with each other. Hence, this experiment is designed to explore the overhead of conflated functionality in conventional concurrency control protocols in the absence of deadlock handling logic.

Figure 11(a) shows the results of the low contention experiment. All the schemes that we compare use lock-based concurrency control. However, since all transactions are read-only, all locks that are acquired are shared-locks, and no lock request is ever denied. Thus, one may expect that all schemes should perform similarly. Surprisingly, this is not the case. Read-only transactions are extremely short in YCSB. As a consequence, concurrency control overhead — the process of requesting and immediately receiving a shared lock — takes up a non-trivial fraction of each transaction's total execution time. Single partition ORTHRUS outperforms all other baselines, which indicates that its concurrency control overhead is the least among all the other baselines when only one concurrency control thread is involved in a transaction. This is because it doesn't require synchronized access to shared memory concurrency control meta-data.

§3.3 explained that the number of message delays required in order for a transaction to acquire its locks in ORTHRUS is $N_{cc} + 1$, where N_{cc} is the number of concurrency control threads on which a transaction's locks reside. The number of message delays required for lock acquisition directly affects the time spent acquiring locks; more message delays correspond to more time spent acquiring locks. Since each transaction in single partition ORTHRUS acquires *all* its locks from a single concurrency control thread, the number of message delays per transaction in single partition ORTHRUS is 2. The number of message delays per transaction in dual partition ORTHRUS is 3 (because every transaction must acquire its locks from two concurrency control threads, but the threads forward messages to eliminate one extra message). Finally, in random ORTHRUS, the number of concurrency control threads that a single transaction acquires its locks from is, on average, a factor of 3 higher than in dual ORTHRUS. The difference in throughput of each of these systems is therefore directly attributable to the difference in the number of message delays required for lock acquisition.

Figure 11(a) also shows that both *Deadlock-free locking* and 2PL outperform random ORTHRUS. The reason is that messaging overhead in ORTHRUS when there are 8-10 messages per transaction exceeds the overhead of latch acquisition and data movement in the 2PL and *Deadlock-free locking* baselines. However, single and dual partition ORTHRUS (when there are only 2-3 messages per transaction) outperform both 2PL and *Deadlock-free locking*, indicating that the reduction in synchronization and data movement overhead does indeed pay off for many workloads, even under low contention.

Figure 11(b) compares each of these systems under high contention. Since read-only transactions never conflict with one another, the increase in contention does not affect the number of actual conflicts in the workload (which remains zero). In this case, the throughput of each ORTHRUS configuration increases slightly. The increase in throughput is due to better cache locality in concurrency control and execution threads. Concurrency control threads experience better locality because they often update the same meta-data across different transactions (corresponding to contended records). Execution threads experience better locality because they read the same contended records across transactions.

However, unlike ORTHRUS, *Deadlock-free locking* and 2PL do not scale beyond 60 cores. Instead, their throughput *decreases* after 60 cores. This decrease in throughput occurs despite the absence of conflicts among read-only transactions. *Deadlock-free locking* and 2PL do not scale because of contention for concurrency control meta-data. In order to acquire or release a logical lock, both locking implementations must acquire a latch. This latch protects the hash-bucket in which requests for a particular lock reside. Under high contention, database threads will contend for the latches protecting the meta-data corresponding to popular records. Furthermore, as the number of cores allocated to the database increases, latch contention increases because more threads attempt to acquire the same set of latches. As §2.1 explained, contention on memory words – such as those corresponding to latches – leads to significant cache coherence overhead, which in turn inhibits *Deadlock-free locking* and 2PL's scalability[7]. Note that *Deadlock-free locking*'s throughput is nearly identical to that of 2PL because 2PL's deadlock handling logic is never invoked due to the absence of logical conflicts among read-only transactions.

[6]We also evaluated the throughput of 2PL with the wait-for graph, and dreadlocks algorithms, but found that wait-die outperformed both these schemes on this benchmark.

[7]Note that while our implementation uses latches, latch-free algorithms are subject to the same cache-coherence overhead [13, 48].

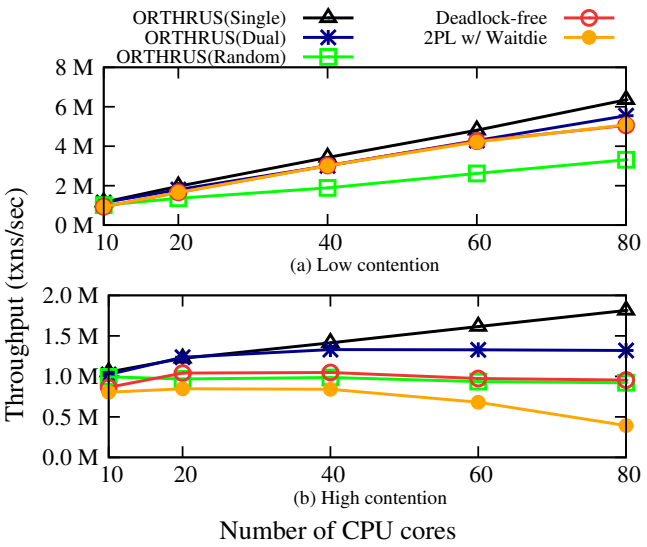

Figure 12: YCSB 10RMW scalability.

A.2 10RMW transactions

We now compare ORTHRUS's throughput with that of *Deadlock-free locking* and 2PL when transactions perform 10 read-modify-write (RMW) operations. As in §A.1, we perform both low contention and high contention experiments. Records in transactions' read- and write-sets are picked in the same manner as the read-only experiment, where the hot set for the high contention experiment is 64 records. Note that unlike read-only transactions which never logically conflict, a pair of 10RMW transactions conflict with each other if their read-/write-sets intersect.

Figure 12(a) shows the results of the low contention experiment. Each system's performance trend is similar to that in the low contention read-only experiment (§A.1). However, the absolute throughput of each system is less than its throughput in the low contention read-only experiment. This is because 10RMW transactions take longer than read-only transactions since they both read and write 10 records while read-only transactions only read 10 records.

Figure 12(b) shows the throughput of each system under high contention. Under high contention, 2PL does not scale beyond 20 cores. On the contrary, its throughput begins to *drop* as more cores are added. *Deadlock-free locking*'s throughput also does not scale beyond 20 cores. However, unlike 2PL, its throughput plateaus at 1,000,000 transactions per second. We attribute the difference between *Deadlock-free locking* and 2PL to deadlock handling overhead. 2PL uses the wait-die deadlock avoidance algorithm, which aborts a transaction if it requests a lock held by an older transaction (§4). Therefore, in addition to suffering from synchronization and data-movement overheads associated with lock acquisition, 2PL suffers additional overhead due to deadlock handling (in particular, wasted work due to transaction aborts).

As expected, single and dual partition ORTHRUS outperform random ORTHRUS. This is because transactions in single and dual partition ORTHRUS request locks from fewer concurrency control threads (relative to random ORTHRUS), and therefore hold contended locks for shorter durations (§3.3). Single partition ORTHRUS outperforms dual partition ORTHRUS configuration for the same reason.

At 80 cores, the difference between 2PL and random, dual, and single partition ORTHRUS is 2.3x, 3.35x, and 4.65x, respectively.

B. DEADLOCK EXPERIMENTS

This section analyzes the pros and cons various deadlock handling mechanisms in greater detail. §4 explained that *Deadlock-free locking* eliminates the possibility of deadlock by acquiring locks in lexicographic order. This lexicographic order does not necessarily match the order in which locks are actually required by a transaction's logic. For instance, if the last operation performed by a transaction is an update to a contended record, then a conventional locking protocol would acquire a lock on this record at the end of the transaction. In contrast, *Deadlock-free locking* may need to acquire this contended lock before other locks because the contended lock may occur earlier in the lexicographic order. In the worst case, the lexicographic order may force the contended lock to be acquired at the *beginning* of the transaction. Thus, although lexicographically ordered lock acquisition avoids deadlocks, it may force transactions to hold contended locks for longer. This section analyzes this tradeoff in detail.

The workload used in this section consists of transactions which perform 10 read-modify-write operations (RMW) on 1000-byte sized records (as in §A.2). One or two of the records in each transaction's write-set are chosen from a set of hot records. The remainder – nine or eight, respectively – are chosen from a set of cold records. We vary contention in the workload by decreasing the number of hot records in the database. For each experiment, we plot the throughput of each system against the number of hot records. In each graph, the number of hot records decreases along the x-axis, which corresponds to increasing contention.

Figure 13 shows the results of the experiments. We run experiments corresponding to different points at which transactions update the hot record(s). In Figure 13(a) and Figure 13(d), hot records are updated at the beginning of transactions. In Figure 13(b) and Figure 13(e), hot records are updated at the end of transactions. In Figure 13(c) and Figure 13(f), hot record updates are randomly distributed across transactions' writesets.

When hot records are updated at the beginning of transactions (Figure 13(a) and Figure 13(d)), we find that *Deadlock-free locking* outperforms two-phase locking (2PL) with every variant of deadlock handling algorithm. This is because *Deadlock-free locking* and 2PL are forced to acquire locks on hot records at the beginning of transactions, which mostly eliminates their advantage over *Deadlock-free locking* which is also forced to acquire its locks at the beginning of the transaction. Meanwhile the 2PL variants must additionally execute deadlock handling logic, which was shown to be expensive in §4.1. Furthermore, when transactions access two hot records (Figure 13(a)), deadlocks occur more frequently than in the case when transactions access one hot record. Thus, in Figure 13(a), 2PL-based systems waste additional cycles on transactions that are eventually aborted. As a consequence, the relative difference between *Deadlock-free locking* and the best performing 2PL system is higher when transactions update two hot records (2.2x) than when transactions update one hot record (1.4x).

The higher probability of deadlocks when transactions update two hot records (Figure 13(a)) also explains the relative performance between the wait-die and the dreadlocks/wait-for graph deadlock handling mechanisms. Wait-die conservatively aborts transactions that wait on locks held by older transactions. These conservative aborts minimize the wasted work involved in forcing a transaction to wait for a lock, only to have the same transaction later abort due to a deadlock. As a consequence, wait-die outperforms both the dreadlocks and wait-for graph deadlock detection algorithms in Figure 13(a). However, when transactions update a single hot record (Figure 13(d)), conservatively aborting transactions in anticipation of deadlocks does not pay off because deadlocks are rare.

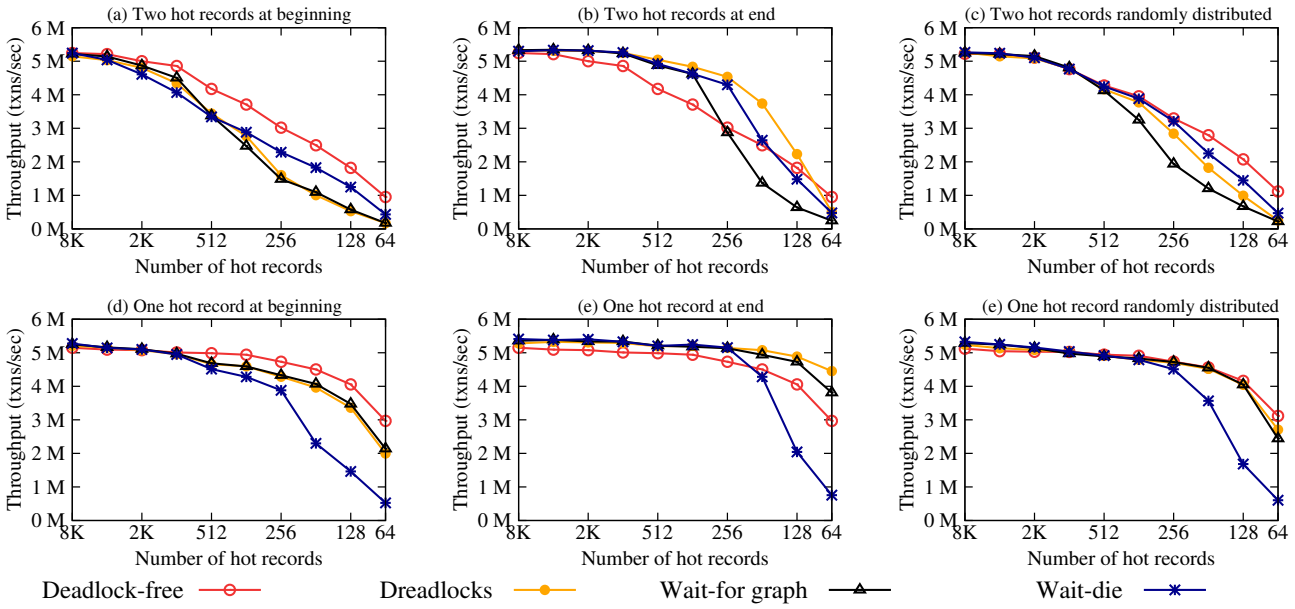

Figure 13: Throughput while varying number of hot records at 80 CPU cores.

Thus, both dreadlocks and wait-for graph outperform wait-die in Figure 13(d).

Figure 13(b) and Figure 13(e) show the results of experiments where updates to two hot records, and one hot record, respectively, occur at the end of each transaction. These experiments represent the best-case behavior for the 2PL systems because they hold locks on contended data for the least possible duration (the time taken to update the data) [52]. Meanwhile, because it acquires locks in lexicographic order, *Deadlock-free locking* cannot wait until the end of the transaction to acquire locks on hot records. Indeed, in Figure 13(b), when transactions update two hot records, we find that 2PL initially outperforms *Deadlock-free locking*. However, at the highest level of contention (at 64 hot records), the 2PL baselines are more susceptible to deadlocks, which in turn causes them to waste cycles on transactions that will abort. At this point, despite the fact it holds contended locks for longer durations, *Deadlock-free locking* outperforms all 2PL implementations.

In Figure 13(e), however, only a single hot record is updated at the end of each transaction. This experiment represents the *best-case* scenario for the 2PL schemes for two reasons. First, 2PL holds contended locks for the shortest possible duration. Second, deadlocks are rare. Accordingly, 2PL with dreadlocks and wait-for graph outperform *Deadlock-free locking* by 1.5x and 1.3x respectively. Despite the fact it holds locks for longer duration, *Deadlock-free locking* outperforms 2PL with wait-die at high levels of contention because wait-die spuriously aborts transactions that are not involved in deadlocks.

Note that when locks for hot records are acquired at the end of transactions, the performance of the wait-for graph and dreadlock approaches to deadlock control are no longer identical. §4.1 explained that both algorithms effectively search for cycles in the transitive closure of transactions waiting for locks. Even though the dreadlocks and wait-for graph algorithms are equivalent, they use very different graph representations — a bitmap to represent a transaction's transitive closure of wait-for dependencies, and an explicit distributed dependency graph of transactions, respectively. Computing the transitive closure of the explicit graph involves traversing the graph of transactions, which is distributed across the cores of a machine. In contrast, dreadlocks only spins on the bitmap of the most recent preceding transaction (the transitive closure computation is effectively memoized in each bitmap) — a much lighter-weight process. The extra overhead of the graph traversal in the wait-for graph approach is hidden when the hot records are accessed at the beginning of the transaction, because the traversal can take place in parallel with the transaction that is holding the lock performing the rest of the transaction. However, when the hot record locks are acquired at the end of the transaction, the original transaction may complete while the graph traversal is ongoing, making this graph traversal overhead more visible.

Finally, Figure 13(c) and Figure 13(f) show the results of experiments where the positions of updates to hot records are randomly distributed across transactions. In both these experiments, *Deadlock-free locking* outperforms all variants of 2PL. As in prior experiments, *Deadlock-free locking* has a greater advantage over 2PL when transactions update two hot records because deadlocks are more likely.

In conclusion, these experiments indicate that *Deadlock-free locking*'s lexicographic lock acquisition can cause locks on hot records to be held for longer, which can translate into poorer performance (relative to 2PL) in cases where hot records are accessed at the end of transactions. However, when deadlocks are actually prevalent (experiments with two hot records), *Deadlock-free locking* outperforms 2PL, even when hot records are updated at the end of transactions. Furthermore, when operations on hot records are not restricted to the end of transactions, *Deadlock-free locking always* outperforms 2PL (Figure 13(a), Figure 13(c), Figure 13(d), and Figure 13(f)). These results make a strong case for eliminating deadlock handling logic from pessimistic locking protocols; for the most part, the benefits of eliminating deadlocks significantly outweigh any concurrency disadvantage due to increased lock hold times.

Furthermore, it should be noted that the lack of flexibility of *Deadlock-free locking* in these experiments, in particular the requirement to acquire locks on hot records before they are actually needed, could be overcome with further improvements to the algorithm. In particular, it is possible to envision an algorithm that *renames* hot records such that they appear at the end of the lexicographic order relative to the other records in the database. This would further increase the advantage of *Deadlock-free locking* relative to the 2PL-based deadlock handling schemes.