Data-Oriented Transaction Execution

Ippokratis Pandis[1] **Ryan Johnson**[1] **Nikos Hardavellas**[2] **Anastasia Ailamaki**[2,3]

January 2010
CMU-CS-10-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1] Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, USA
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[3] School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne, VD, Switzerland

This work was partially supported by a Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds.

Keywords: Data-oriented transaction execution, DORA, Storage Manager Scalability, Multi-threaded Storage Engine, Database Server, Online Transaction Processing (OLTP), Multicore transaction processing, Latch contention, Lock manager.

Abstract

While hardware technology has undergone major advancements over the past decade, transaction processing systems have remained largely unchanged. The number of cores on a chip grows exponentially, following Moore's Law, allowing for an ever-increasing number of transactions to execute in parallel. As the number of concurrently-executing transactions increases, contended critical sections become scalability burdens. In typical transaction processing systems the centralized lock manager is often the first contended component and scalability bottleneck.

In this paper, we identify the conventional thread-to-transaction assignment policy as the primary cause of contention. Then, we design DORA, a system that decomposes each transaction to smaller actions and assigns actions to threads based on which data each action is about to access. This allows each thread to mostly access thread-local data structures, minimizing interaction with the contention-prone centralized lock manager. Built on top of a conventional storage engine, DORA's design maintains all the ACID properties. Evaluation of a prototype implementation of DORA on a multicore system demonstrates that DORA attains up to 4.6x higher throughput than the state-of-the-art storage engine when running a variety of OLTP workloads, such as TPC-C, TPC-B, and Nokia's TM1.

1 Introduction

The diminishing returns of increasing on-chip clock frequency coupled with power and thermal limitations have led hardware vendors to seek performance improvements in thread-level parallelism by placing multiple cores on a single die. Today's multicore processors feature 64 hardware contexts on a single chip equipped with 8 cores[1], while multicores targeting specialized domains find market viability at even larger scales. With experts in both industry and academia forecasting that the number of cores on a chip will follow Moore's Law, an exponentially-growing number of cores will be available with each new process generation.

As the number of hardware contexts on a chip increases exponentially, an unprecedented number of threads concurrently execute, contending for access to shared resources. Thread-parallel applications running on multicores exhibit an increasing amount of delays in heavily-contended critical sections, with detrimental performance effects [14]. To tap the increasing computational power of multicore processors, software systems must alleviate such contention bottlenecks and allow performance to scale commensurately with the number of cores.

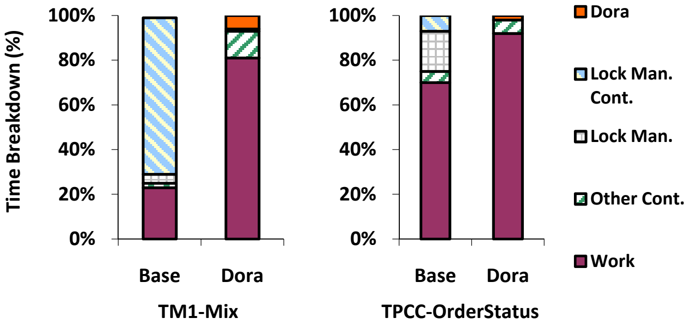

Figure 1. Time breakdown for a conventional transaction processing system and a DORA prototype when all the 64 hardware contexts of a Sun Niagara II chip are fully utilized running (a) the TM1 benchmark, and (b) TPC-C OrderStatus transactions.

Online transaction processing (OLTP) is an indispensable operation in any enterprise. In the past decades, transaction processing systems have evolved into sophisticated software systems with codebases measuring in the millions of lines. Several fundamental design principles, however, have remained largely unchanged since their inception. The execution of transaction processing systems is full of critical sections [14]. Consequently, these systems face significant performance and scalability problems on highly-parallel hardware, as the left bars in Figure 1 show. In order to cope with the scalability problems of transaction processing systems, researchers have suggested employing shared-nothing configurations [6] on a single chip, dropping some of the ACID properties [5], or doing both [24].

1.1 Thread-to-transaction vs. thread-to-data

In this paper, we argue that the primary cause of the contention problem is the uncoordinated data accesses that is characteristic of conventional transaction processing systems. These systems assign each transaction to a worker thread, a mechanism we refer to as thread-to-transaction assignment. Because each transaction runs on a separate thread, threads contend with each other during shared data accesses.

[1] Modern cores contain multiple hardware contexts, allowing concurrent execution of multiple instruction streams.

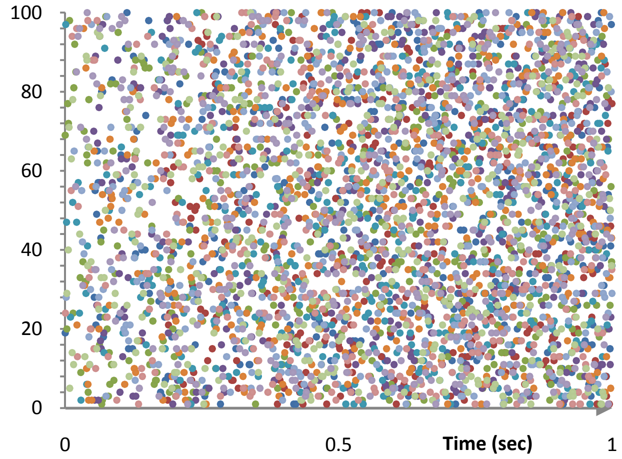

Figure 2 (a). Trace of the record accesses by the threads of a conventional system; data accesses are uncoordinated and complex.

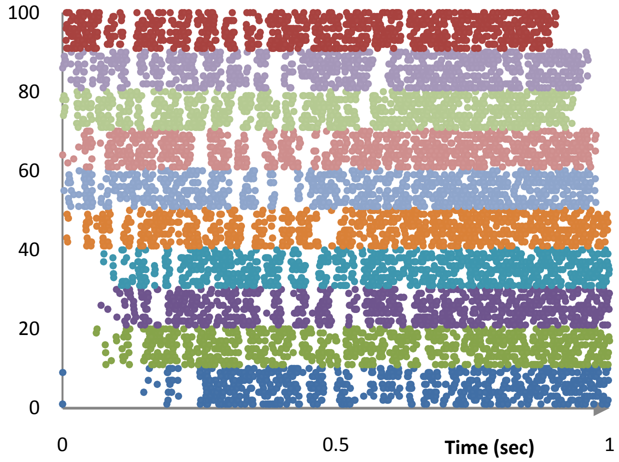

Figure 2 (b). Trace of the record accesses by the threads of a DORA system; data accesses are coordinated and show regularity.

The access patterns of each transaction, and consequently of each thread, are arbitrary and uncoordinated. Figure 2(a) depicts the accesses issued by each worker thread of a transaction processing system, to each one of the records of the District table in a TPC-C database with 10 Warehouses.[2] To ensure data integrity, each thread enters a large number of critical sections in the short lifetime of each transaction it executes. Critical section, however, incur latch acquisitions and releases, whose overhead increases with the number of parallel threads.

To assess the performance overhead of critical section contention, Figure 3(a) depicts the throughput per hardware context attained by a state-of-the-art storage manager [16] as the number of hardware contexts increases. The workload consists of clients repeatedly submitting GetSubscriberData transactions from the TM1 benchmark (methodology detailed in Section 5.1). As the machine utilization increases, the performance of each context drops. At 64 hardware contexts, each hardware context attains less than a fifth of the performance that could be achieved with a single hardware context in the system. Figure 3(b) shows that critical sections within the lock manager quickly dominate performance. With 64 hardware contexts, the system spends more than 85% of its execution time on threads waiting to execute critical sections inside the lock manager.

Based on the observation that uncoordinated accesses to data lead to high levels of contention, we propose a data-oriented architecture (DORA) to alleviate contention. Rather than coupling each thread with a transaction, DORA couples each thread with a disjoint subset of the database. Transactions flow from one thread to the other as they access different data, a mechanism we call thread-to-data assignment. Transactions are decomposed to smaller actions according to the data they access, and are routed to the corresponding threads for execution. In essence, instead of pulling data (database records) to the computation (transaction), a system adopting thread-to-data assignment distributes the computation to wherever the data is mapped. Figure 2(b) illustrates the effect of the data-oriented assignment of work on data accesses. It plots the data access patterns issued by a prototype system which employs the thread-to-data assignment against the same workload as Figure 2(a).

A system adopting thread-to-data assignment can exploit the regular pattern of data accesses, reducing the pressure on contended components. In DORA, each thread coordinates accesses to its subset of data using a private locking mechanism, isolated from the other threads. By limiting thread interactions with the centralized lock manager, DORA eliminates the contention in this component (Figure 3(c)) and provides better scalability (Figure 3(a)).

[2] The system and workload configuration are kept small to enhance the graph's visibility. The system is configured with 10 worker threads and the workload consists of 20 clients repeatedly submitting Payment transactions from the TPC-C benchmark.

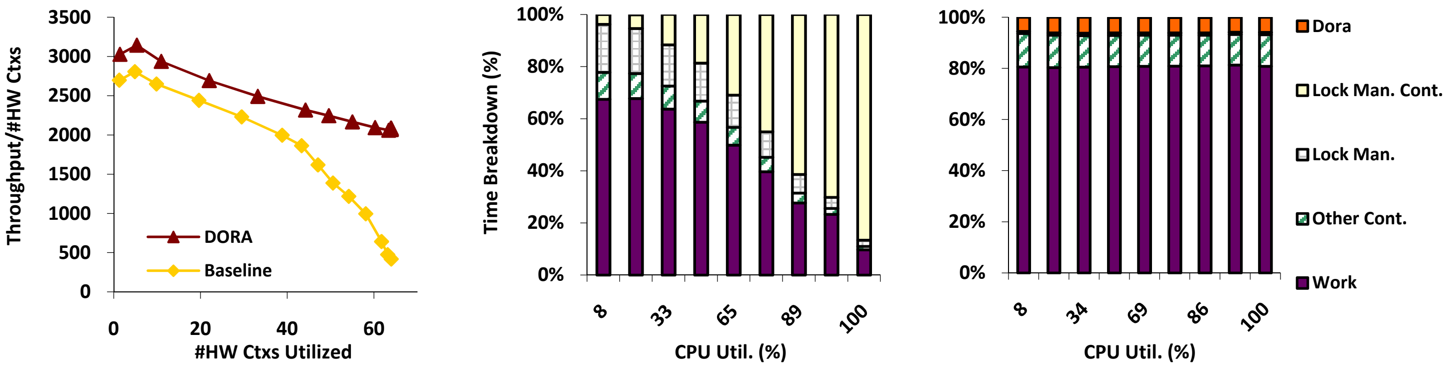

Figure 3. DORA compared to Baseline when the workload consists of TM1-GetSubscriberData transactions. (a) The throughput per number of hardware contexts, as an increasing number of hardware contexts are utilized. (b) The time breakdown for the Baseline system. (c) The time breakdown for a DORA prototype.

DORA is enabled by the low-latency, high-bandwidth inter-core communication of multicore systems. This allows transactions to flow from one thread to the other with minimal overhead, as each thread accesses different parts of the database. Figure 1 compares the time breakdown of a conventional transaction processing system and a prototype DORA implementation, when all the 64 hardware contexts of a Sun Niagara II chip are utilized, running Nokia's TM1 benchmark [22] and TPC-C Order-Status transactions [23]. The DORA prototype eliminates the contention on the lock manager (e.g., in TM1). Also, it substitutes the centralized lock management with much lighter-weight thread-local locking mechanism (e.g., in TPC-C OrderStatus.)

1.2 Contributions and document organization

The contribution of this paper is three-fold:

- We demonstrate that the conventional thread-to-transaction assignment results in contention at the lock manager that severely limits the performance and scalability on multicores.
- We propose a data-oriented architecture that exhibits predictable access patterns, enabling the substitution of the heavyweight centralized lock manager with a lightweight thread-local locking mechanism. This allows shared-everything systems to scale to high core counts without weakening the ACID properties.
- We evaluate a prototype DORA transaction execution engine and show that it attains up to 82% higher peak throughput against a state-of-the-art storage manager. If no admission control is applied, the performance benefits for DORA can reach as large as 4.8x. Additionally, when unsaturated, DORA achieves up to 2.5x lower response times because it exploits the intra-transaction parallelism inherent in many transactions.

The rest of the document is organized as follows. Section 2 discusses related work. Section 3 explain why a typical storage manager may suffer from contention in its lock manager. Section 4 presents DORA, an architecture based on the thread-to-data assignment. Section 5 evaluates the performance of a prototype DORA OLTP engine, and Section 6 concludes.

2 Related Work

Locking overhead is a known problem even for single-threaded systems. Harizopoulos et al. [10] analyze the behavior of the single-threaded Shore storage manager [3] running two TPC-C transactions. When executing the Payment, the system spends 25% of its time on code related to logical locking, while with NewOrder it spends 16%. We corroborate this result and by using a multi-threaded storage manager and increasing the hardware parallelism, we reveal the lurking problem of contention, making the lock manager the system bottleneck.

Advancements in virtual machine technology [2] enable the deployment of shared-nothing configurations on multicores. In shared-nothing configurations, the dataset is physically distributed and there is replication of both instructions and data. For transactions that span multiple partitions, a distributed consensus protocol needs to be applied. H-Store [24] takes the shared-nothing approach to the extreme by deploying a set of single-threaded storage managers that serially execute requests, avoiding concurrency control. However, the complexity of coordinating distributed transactions [12][5] and the imbalances caused by skewed data or requests are problems that reduce the performance of shared-nothing systems.

Rdb/VMS [17] is a parallel database system design optimized for the inter-node communication bottleneck. In order to reduce the cost of nodes exchanging lock requests over the network, Rdb/VMS keeps a logical lock at the node which last used it until that node returns it to the owning node or a request from another node arrives. Cache Fusion [19], used by Oracle RAC, is designed to allow shared-disk clusters to combine their buffer pools and reduce accesses to the shared disk. Like DORA, Cache Fusion does not physically partition the data but distributes the logical locks. However, neither Rdb/VMS nor Cache Fusion handle the problem of contention. Any large number of threads may access the same resource at the same time leading to poor scalability. DORA ensures that the majority of the resources will be accessed by a single thread.

The scheme employed in DORA is similar as if in a conventional system each transaction-executing thread holds an exclusive lock on a region of records. The exclusive lock is associated with the executing thread, rather than any transaction, and it is held across multiple transactions. Locks on separator keys [8] could be used to implement such behavior.

Speculative lock inheritance (SLI) [15] detects “hot” locks during a transactional workload and those locks may be held by the transaction-executing threads across transactions. SLI similar to DORA manages to reduce the contention on the lock manager. However, it still spends a significant amount of its time inside the lock manager.

DORA shares similarities with staged database systems[11]. A staged system splits queries into multiple requests which may proceed in parallel. The splitting is operator-centric and designed for pipeline parallelism. Usually OLTP involves the invocation of a single GetNext() call per request and pipeline parallelism has little to offer. On the other hand, similar to staged systems, DORA exposes work-sharing opportunities by sending related requests to the same queue. We leave it as future work to try to exploit possible cross-transaction optimization opportunities.

DORA employs intra-transaction parallelism to reduce contention. Intra-transaction parallelism has been a topic of research for more than two decades (e.g., [7]). Colohan et al. [4] use thread-level speculation to execute transactions in parallel. They show the potential of intra-transaction parallelism, achieving up to 4x lower response times than a conventional system. Thread-level speculation, however, is an hardware-based technique not available in today’s hardware. DORA’s mechanism requires only fast inter-core communication that is already available in multicore hardware.

3 Source of Contention

A typical OLTP workload consists of a large number of concurrent, short-lived transactions, each accessing a small fraction (ones to tens of records) of a large dataset. Each transaction independently executes on a separate thread. To guarantee data integrity, transactions enter a large number of critical sections to coordinate accesses to shared resources. One of the shared resources is the logical locks[3] of the lock manager. The lock manager is responsible for maintaining isolation between concurrently-executing transactions. The lock manager provides an interface for transactions to request, upgrade, and release locks. Behind the scenes it also ensures that transactions acquire proper intention locks, and performs deadlock prevention and detection. As hardware parallelism increases, the lock manager becomes contended and the obstacle to scalability.

Next, we describe the lock manager of the Shore-MT storage engine [16]. Although the implementation details of commercial system’s lock managers are largely unknown, we expect their implementations to be similar. A possible varying aspect is that of latches. Shore-MT uses a preemption-resistant variation of the MCS queue-based spinlock [20]. In the Sun Niagara II hardware, our testbed, and for the CPU load we are using in this study (<130%), spinning-based implementations outperform any solution involving blocking [13].

In Shore-MT, every logical lock is a data structure that contains the lock’s mode, the head of a linked list of lock requests (granted or pending), and a latch, as depicted in Figure 5 (labeled System-wide Lock Manager). When a transaction attempts to acquire a lock, the lock manager first ensures the transaction holds higher-level intention locks, requesting them automatically if needed. If an appropriate coarser-grain lock is found, the request is granted immediately. Otherwise, the manager probes a hash table to find the desired lock. Once the lock is located, it is latched and the new request is appended to the request list. If the request is incompatible with the lock’s current mode the transaction must block. Finally, the lock is unlatched and the request returns. Each transaction maintains an ordered list of all the lock requests it is granted, in the order that it acquired them. At transaction completion, the

[3] We use the term “logical locking” instead of the more popular “locking” to emphasize its difference with latching. Latching protects the physical consistency of main memory data structures; logical locking protects the logical consistency of database resources, such as records and tables.

transaction releases the locks one by one starting from the youngest. To release a lock, the lock manager latches the lock and unlinks the corresponding request from the list. Before unlatching the lock, it traverses the request list in order to decide the new lock mode and to find any pending requests which may now be granted.

Due to longer lists, of lock requests the effort required to grant or release a lock grows with the number of active transactions. Frequently-accessed locks, such as table locks, will have many requests in progress at any given point. Deadlock detection imposes additional lock request list traversals. The combination of the longer lists of lock requests, with the increased number of threads executing transactions and contending for locks leads to detrimental performance results.

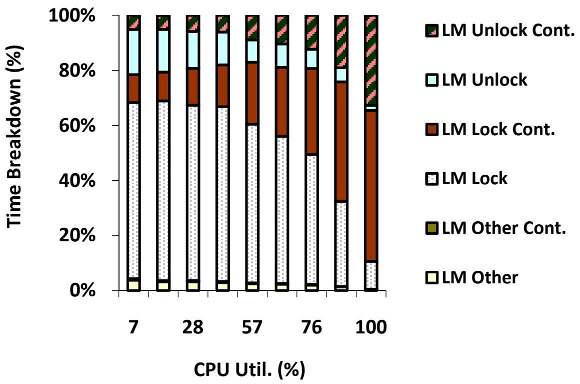

Figure 4. Time breakdown inside the lock manager of Shore-MT when it runs the TPC-B benchark and the load increases.

Figure 4 provides a breakdown of where time is spent inside the lock manager of Shore-MT when it runs the TPC-B benchmark and the system load increases on the x-axis. The time is broken on the time it takes to acquire (lock) the locks, to release them, and the corresponding contention of each operation. When the system is lightly-loaded, it spends more than 85% of the time on useful work inside the lock manager. As the load increases, however, the contention dominates. At 100% CPU utilization, more than 85% of the time inside the lock manager is contention (spinning on latches).

4 A Data-Oriented Architecture

In this section, we present the design of an OLTP system which employs a thread-to-data assignment policy. We exploit the coordinated access patterns of this assignment policy to eliminate interactions with the contention-prone centralized lock manager. At the same time, we maintain the ACID properties and do not physically partition the data. We call the architecture data-oriented architecture, or DORA.

4.1 Design overview

DORA is a layer over a conventional storage manager. Its functionality includes three basic operations. First, it binds worker threads to disjoint subsets of the database. Second, it distributes the work of each transaction across transaction-executing threads according to the data accessed by the transaction. Third, it avoids interactions with the centralized lock manager as much as possible during request execution. This section describes each operation in detail. We use the execution of the Payment transaction of the TPC-C benchmark as our running example. The Payment transaction updates a Customer’s balance, reflects the payment on the District and Warehouse sales statistics, and records it in a History log [23].

4.1.1 Binding threads to data

DORA couples worker threads with data by setting a routing rule for each table in the database. A routing rule is a mapping of ranges of records, called datasets, to worker threads, called executors. Each dataset is assigned to one executor and an executor can be assigned multiple datasets from a single table. With the routing rules each table is logically decomposed into disjoint sets of records. All data resides in the same bufferpool. There is no physical separation or data movement involved.

A table's routing rule may use all the fields of the primary key of the table or only a subset of them. The fields used by the routing rule are called the routing fields of the table. For example, the primary key of the Customers table of the TPC-C database consists of the Warehouse id, the District id, and the Customer id. The routing fields may be all the three fields, or only a subset of them. In our example, we assume the Warehouse id is the routing field in each one of the four accessed tables. Any routing rule is valid as long as any possible primary key value maps to a unique executor. The routing rules are maintained at runtime by the DORA resource manager. Periodically, the resource manager updates the routing rules to balance load. The resource manager varies the number of executors per table depending on the size of the table, the number of requests for this table, and the available hardware resources.

4.1.2 Transaction flow graphs

In order to distribute the work of each transaction to the appropriate executors, DORA translates each transaction to a transaction flow graph. A transaction flow graph is a graph of actions to datasets. An action is a subset of a transaction's code which involves a single call to the storage manager. The identifier of an action identifies the region of records this action intends to access. Depending on the type of the access the identifier can be either a primary key, a subset of it, or the empty set. Two consecutive actions can be merged if they have the same identifier (refer to the same dataset).

The more specific the identifier of an action is, the easier is for DORA to route the action to the appropriate executor. For example, actions whose identifier is the primary key are directed to their executor by consulting the routing rule of the table. The most frequent operations in OLTP--the primary index accesses--are translated to primary key actions. On the other hand, the actions whose identifier is a subset of the primary key may map to multiple executors. In that case, the action is broken to a set of smaller actions, each of them resized to correspond to the dataset range of each of the executors. Actions of this category are secondary index accesses, when the secondary index contains a subset of the routing fields. Finally, actions on secondary indexes that do not contain any of the routing fields have the empty set as their identifier. For these secondary actions, the system cannot decide who their responsible executor is. In Section 4.2.2, we discuss how the secondary actions are handled.

DORA uses shared objects across actions of the same transaction in order to control the distributed execution of the transaction, to transfer data between actions with data dependencies, and to detect errors as soon as possible. Those shared objects are called rendezvous points or RVPs. If there is data dependency between two actions, an RVP is placed between them. The RVPs coordinate the distributed execution of each transaction by separating its execution to different phases. The system cannot concurrently execute actions from the same transaction that belong to different phases. Each RVP has a counter initially set to the number of actions that need to report to it. Every executor which finishes the execution of an action decrements the corresponding RVP counter by one. When an RVP's counter becomes zero, the next phase starts. The executor which zeroes a particular RVP initiates the next phase by enqueueing all the actions of that phase to their corresponding executors. The executor which zeroes the last RVP in the transaction flow graph calls for the transaction commit. On the other hand, any executor can abort the transaction and hand it to recovery.

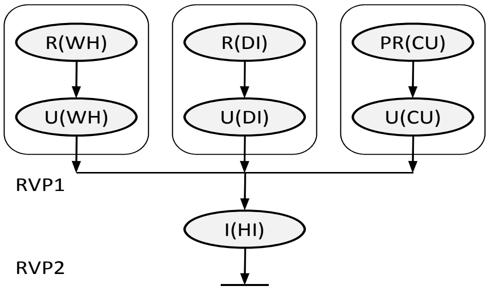

Figure 5. TPC-C Payment transaction flow graph.

The transaction flow graph for the Payment transaction is depicted in Figure 5. Each Payment probes a Warehouse and a District and updates them. In each case, both actions (record retrieval and update) are identified by the primary key. Hence, they can be merged. The Customer, on the other hand, 60% of the time is probed through a secondary index and then updated. This secondary index contains the Warehouse id, the District id, and the Customer's last name. If the routing rule on the Customer table uses only the Warehouse id or the District id fields, then the system knows which executor is responsible for this secondary index access. If the routing rule uses also the

Customer id (field of the primary key of the table), then this secondary index access needs to be broken to smaller actions that cover all the possible values for the Customer id. Finally, if the routing rule uses only the Customer id, then the system cannot decide which executor is responsible for this access and this secondary index access becomes a secondary action. In our example, we assume that the routing field is the Warehouse id. Hence, the secondary index probe and the consequent Customer record update have the same identifier and can be merged. Finally, because of the data dependency between the record insert on the History table and the other three probes an RVP is placed between them, separating the Payment transaction to two different phases.

Payment's specification requires the Customer to be randomly selected from a remote Warehouse 15% of time. A shared-nothing system that partitions the database on the Warehouse has to execute a distributed transaction in that case with all the involved overheads. DORA, on the other hand, handles gracefully such transactions by simply routing the Customer action to a different executor. Hence, its performance is not affected by the percentage of "remote" transactions.

4.1.3 Executing requests

DORA via the routing rules and the transaction flow graphs distributes all the actions that indent to operate on the same dataset to one executor. The executor is responsible to maintain isolation and ordering across conflicting actions.

In order to achieve that each executor has three data structures associated with it: a queue of incoming actions, a queue of completed action identifiers, and a thread-local lock table. The actions are processed in the order they enter the incoming queue. To detect conflicting actions the executor uses the local lock table. The conflict resolution happens at the action-identifier level. That is, the local lock table is a hash table whose input are the action identifiers. The local locks have only two modes, shared and exclusive. There is no need for intention locks since any modification in any dataset happens in a serial fashion by a single executor. Since the action identifiers may be only a subset of the primary key, the locking scheme employed is similar to that of key-prefix locks [8]. Once an action acquires all the local locks, it can be executed without concurrency control, enforced by the system-wide centralized lock manager. Each action holds the local locks it acquired until the overall transaction commits (or aborts). The terminal RVP of each transaction first waits for a response from the underlying storage manager that the commit (or abort) has completed. Then, it enqueues the identifier of all the actions that participated in the transaction to the completed queue of their executors. Each executor removes entries from its local lock table as actions complete, and serially executes any blocked actions which can now proceed.

Each executor implicitly holds an intent exclusive (IX) lock for the whole table, and does not have to interface the centralized lock manager in order to re-acquire it for every transaction. For operations which intend to modify data ranges that span multiple datasets or cover the whole table (e.g. a table or index drop, or a table scan) an action should be enqueued to every executor operating on that particular table. Once all the actions are granted access, the "multi-partition" operation can proceed. In transaction processing workloads such operations already hamper concurrency, and occur rarely in scalable applications.

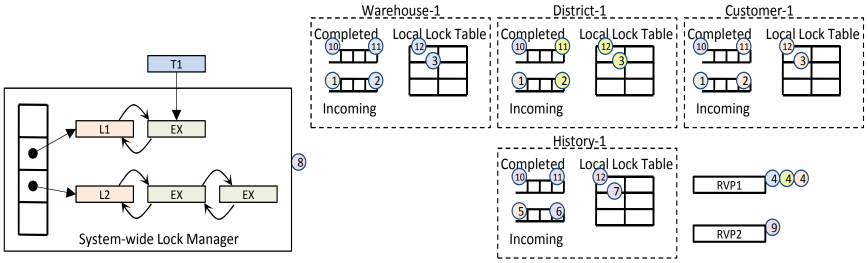

Figure 6. Execution example for the TPC-C Payment.

Figure 6 shows the execution flow for a Payment transaction. Each circle is color-coded to depict the thread which executes that step: The thread that receives the request (e.g., from the network) enqueues the actions of the first phase to the corresponding executors (1). Once the action reaches the head of the incoming queue it is picked by the executor (2). The executor probes its local lock table to determine whether it can process it or not (3). If there is a conflict, the action is added to a list of blocked actions. Otherwise, the executor completes the action without system-wide concurrency control. Once the action is completed, the executor decrements the RVP counter (4). If it is the last action to report, the executor initiates the next phase by enqueueing the action to the History executor (5).

The History table executor does the same routine, picking the action from the head of the incoming queue (6), and probing the local lock table (7). Payment inserts a record to the History table, and for a reason we explain at Section 4.2.1, the execution of that action needs to interface the centralized lock manager (8). Once the action is completed, the History executor updates the terminal RVP and calls for the transaction commit (9). When the underlying storage engine returns, the History executor enqueues the identifiers of all the actions back to their executors (10). The executors pick the committed action identifier (11), remove the entry from their local lock table, and search the list of pending actions which may now proceed (12).

This example shows how DORA converts the execution of each transaction to a collective effort of multiple threads. Also, it shows how at the expense of additional inter-core communication bandwidth, abundant in modern multicores, minimizes the interaction with the contention-prone centralized lock manager.

4.2 Challenges

4.2.1 Record inserts and deletes

Record probes and updates in DORA can be executed using only the local locking mechanism of each executor. However, there is still a need for centralized coordination across concurrent record inserts and deletions (executed by different executors) for their accesses to specific page slots. That is, it is safe to delete a record without centralized concurrency control with respect to any reads to this record, because all the probes will be executed serially by the executor responsible for that dataset. But, there is problem with the record inserts by other executors. The following interleaving of operations by transactions T1 executed by executor E1 and T2 executed by executor E2 can cause a problem: T1 deletes record R1. T2 probes the page where record R1 used to be and finds its slot free. T2 inserts its record. T1 then aborts. The rollback fails because it is unable to reclaim the slot which T2 now uses. This is a physical conflict (T1 and T2 do not intend to access the same data) which row-level locks would normally prevent and which DORA must address. To avoid this problem, the insert and delete record operations must lock the RID (and the accompanying slot) through the centralized lock manager. Although the centralized lock manager can be a source of contention, typically the row-level locks that need to be acquired due to record insertions and deletes are not contended, and they make up only a fraction of the total number of locks a conventional system would lock. For example, the Payment transactions need to acquire only 1 such lock (for inserting the History record), of the 19 it would acquire if conventionally executed.

4.2.2 Secondary Actions

To resolve the difficulty with secondary actions, the secondary indexes whose accesses cannot be mapped to executors, for each entry store the RID and the fields of the primary used by the routing rule. Transactions accessing tuples through that index use this information to determine which executor should perform the operation. Under this scheme, uncommitted record inserts and updates are properly serialized by the executor, but deletes still pose a risk of violating isolation.

Consider the interleaving of operations by transactions T1 and T2 using the primary index Idx1, and a secondary index Idx2 which can be accessed by any thread. T1 deletes Rec1 through Idx1. T1 deletes entry from Idx2. T2 probes Idx2 and returns not-found. T1 rolls back, causing Rec1 to reappear in Idx2. At this point T2 has lost isolation because it saw the uncommitted (and eventually rolled back) delete performed by T1. To overcome this danger, we add a 'deleted' flag to the secondary index entry. When a transaction deletes a record, it does not remove the entry from the secondary index; any transaction which attempts to access the record will go through its owning partition and find that it is or is being deleted.

Once the deleting transaction commits, it goes back and sets the flag for each index entry of a deleted record outside of any transaction. Transactions accessing secondary indexes ignore any entries with a deleted flag, and may safely re-insert a new record with the same primary key.

Because deleted secondary index entries will tend to accumulate over time, we modify the B-Tree's leaf-split algorithm to first garbage collect any deleted records before deciding whether a split is necessary. For growing or update-intensive workloads, this approach will avoid wasting excessive space on deleted records. If updates are very rare, there will be little potential wasted space in the first place.

4.2.3 Deadlock Detection

In DORA transactions can block on local lock tables. Hence, the storage manager must provide an interface for executors to propagate this information to the deadlock detector.

DORA proactively reduces the probability of deadlocks. Whenever a thread is about to submit the actions of a transaction phase, it latches the incoming queues of all the executors it plans to submit to, so that the action submission appears to happen atomically. This ensures that transactions with the same transaction flow graph will never deadlock with each other. Because two transactions with the same transaction flow graph will deadlock only if their conflicting requests are processed by executors in reverse order. But that is impossible, because the submission of the actions appears to happen atomically the executors serve actions in FIFO order and the local locks are held until the transaction commits. The first of the two transactions will enqueue its actions and will finish before the other.

4.2.4 Intra-transaction parallelism

As mentioned, the low-latency and high-bandwidth inter-core communication in modern multicores allows transactions to flow from one thread to the other with minimal overhead, as each transaction accesses different parts of the database. While intra-transaction parallelism in traditional multiprocessors was beneficial only for long queries, the fast on-chip interconnects in today's multicores enable intra-transaction parallelism even for short-lived transactions. DORA is designed around intra-transaction parallelism. One problem with intra-transaction parallelism is transactions with non-negligible abort rates. Different executors may process actions of a transaction that has already aborted. Before starting the execution of any action, the executor checks if that action comes from an aborted transaction. In cases where the abort rates are high, the execution of the parallel actions is serialized.

4.3 Prototype implementation

In order to evaluate the DORA design, we implemented a prototype DORA OLTP engine over the Shore-MT storage manager [16]. Shore-MT is a modified version of the Shore storage manager [3] that supports multiple threads. Shore supports all the major features of modern database engines: full transaction isolation, hierarchical locking, a CLOCK buffer pool with replacement and prefetch hints, B-Tree indexes, and ARIES-style logging and recovery [21]. We use Shore-MT because it has been shown to scale better than any other open-source storage engine [16].

Our prototype does not have a transaction optimizer which can transform normal transaction code to transaction flow graphs. This is left as a challenging future work. Therefore, all transactions are partially hard-coded. The database metadata and back-end processing are schema-agnostic and general- purpose, but the transaction code is schema-aware. This arrangement is similar to the statically compiled stored procedures that commercial engines support, converting annotated C code into a compiled object that is bound to the database and directly executed. For example, for maximum performance, DB2 allows developers to generate compiled “external routines” in a shared library for the engine to dlopen and execute directly within the engine's core.

The prototype is implemented as a layer (of ~6,000 lines of code) over Shore-MT. Shore-MT's sources are linked directly to the code of the prototype. Modifications to Shore-MT were minimal (less than 500 lines of code out of 170,000). We added an additional parameter to the functions which read or update records, and to the index and table scan iterators. This flag instructs Shore-MT to not use concurrency control. Shore-MT already has a built-in option to access some resources without concurrency control. In the case of insert and delete records, another flag instructs Shore-MT to acquire only the row-level lock and avoid acquiring the whole hierarchy. Finally, Shore-MT employs a variant of the “dreadlocks” distributed deadlock detection scheme [18]. This deadlock detection was entirely internal to the lock manager. We extended its API in order executors to inform Shore-MT about transactions blocking other transactions on local lock tables.

5 Performance Evaluation

We use the most parallel multicore available to compare the DORA prototype against Shore-MT (labeled as Baseline). Shore-MT's current performance and scalability make it one of the first systems to face the contention problem on commodity chip multicores. As hardware parallelism increases and transaction processing systems solve other scalability problems, they are expected to similarly face the problem of contention in the lock manager.

Our evaluation covers four areas. First, we measure how effectively DORA reduces the interaction with the centralized lock manager and what is the impact on performance (Section 5.3). Second, we quantify how DORA exploits the intra-transaction parallelism of transactions (Section 5.4). Third, we measure how efficiently DORA handles secondary actions (Section 5.5). Finally, we put everything together and compare the peak performance DORA and Shore-MT achieve, if a perfect admission control mechanism is used (Section 5.6).

5.1 Experimental setup

Hardware. We perform all our experiments on a Sun T5220 “Niagara II” box configured with 32GB of RAM and running Sun Solaris 10. The Niagara II chip contains 8 cores, each capable of supporting 8 hardware contexts, for a total of 64 “OS-visible” CPUs. Each core has two execution pipelines, allowing it to simultaneously process instructions from any two threads. Thus, it can process up to 16 instructions per machine cycle, using the many available contexts to overlap delays in any one thread.

I/O subsystem. When running OLTP workloads on the Sun Niagara II processor, both systems are capable of high performance. The demand on the I/O subsystem scales with throughput due to dirty page flushes and log writes. For the random I/O generated, hundreds or even thousands of disks may be necessary to meet the demand. To emulate “realistic” behavior, provided the limited budget, we decouple the I/O subsystem performance from the measurements by storing the database on an in-memory file system and modifying the buffer pool manager to impose an artificial penalty for each I/O operation. The artificial delay for our experiments is set to 50usec in order to simulate a high-end array of flash drives. That is, all requests proceed in parallel but each must wait at least 50usec for the requested data to arrive. This arrangement ensures that all aspects of the storage manager are exercised.

5.2 Workloads

We use transactions from three OLTP benchmarks: Nokia's Network Database Benchmark [22] (also known as TM-1), TPC-C [23], and TPC-B [1]. DSS workloads, like TPC-H, spend a large fraction of their time on computations outside the storage engine, imposing small pressure on the lock manager. Hence, they are not an interesting workload for this study.

The primary workload for our interest is TM-1. TM1 is a telecommunication workload benchmark originally developed by Nokia. Nokia used TM1 to evaluate equipment it considered purchasing to run its own workloads on. TM1 consists of seven transactions, operating on four large tables. Three of the transactions are read-only while the other four perform updates. The transactions implement various operations executed by mobile networks during cell phone calls and other common events such as call forwarding. The transactions are extremely short, yet exercise all the codepaths in typical transaction processing. Every transaction accesses only 1-4 database rows, and must execute with very low latency even under extreme load. The benchmark is unusual in that a large fraction of transactions (~25%) fail due to invalid inputs. For the experiments we use a database of 5M subscribers (~7.5GB). TPC-C models an OLTP database for a retailer. It consists of five transactions that follow customer orders from initial creation to final delivery and payment. We use a 150 warehouse TPC-C dataset (~20GB) with a 4GB buffer pool. 150 warehouses can support enough concurrent requests to saturate the machine, but the database is still small enough to fit in the in-memory file system. TPC-B models a bank where customers deposit and withdraw from their accounts. We use a 100 branches TPC-B dataset (~2GB).

For each run, the benchmark spawns a certain number clients and allows them to start submitting transactions. Although the clients run on the same machine with the rest of the system, they add small overhead (<3%). We repeat the measurements multiple times, and the measured relative standard deviation is smaller than 5%. We compile the sources with Sun CC v5.10 compiler using the highest level of optimizations. For measurements that needed profiling, we use tools from Sun Studio 12 suite. The profiling tools impose a small overhead (<15%) but the relative behavior between the two systems remains the same.

5.3 Eliminating the lock manager contention

First, we examine the impact of contention on the lock manager for the Baseline system and DORA as they utilize an increasing number of hardware resources. The workload for this experiment consists of clients repeatedly submitting TM1-GetSubscriberData transactions. The results are shown in Figure 3. The left graph on the y-axis shows the throughput per CPU utilization of the two systems as an increasing number of hardware contexts gets utilized on the x-axis. The other two graphs in each figure show the breakdown for each of the two systems. We can see that the contention in lock manager becomes the bottleneck for the Baseline system, growing to more than 85% of the total execution. In contrast, for DORA the contention on the lock manager is eliminated. We can also observe that the overhead of the DORA mechanism is small, much smaller than the lock manager contention, as well as, the lock manager operations it eliminates.

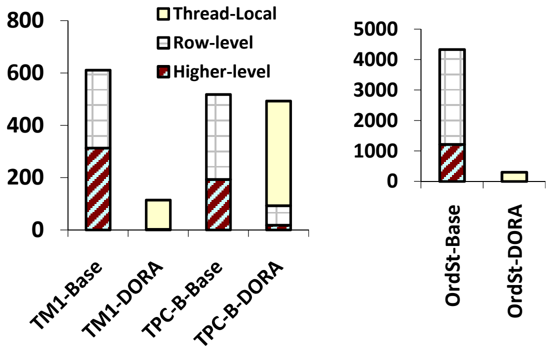

Figure 7. Locks acquired per 100 transactions by Baseline and DORA in three workloads.

Next, we quantify how effectively DORA reduces the interaction with the centralized lock manager and the impact in performance. We measure the number of locks acquired by the Baseline and DORA. We instrument the code of the two systems to report the number and the type of the acquired locks. The locks are categorized in three types. The row-level locks, the locks of the centralized lock manager that are not row-level, and the local locks which only DORA uses. In OLTP we expect the contention for the row-level locks to be limited, because there is a very large number of randomly accessed records. On the other hand, as we go up in the hierarchy of locks, we expect the contention to increase. For example, every transaction needs to acquire intention locks on the tables it is going to access. Figure 7 shows the number of locks acquired per 100 transactions when the two systems execute transactions from the TM1 and TPC-B benchmarks, as well as, TPC-C OrderStatus transactions. DORA has only minimal interaction with the centralized lock manager.

Figure 7 gives an idea on how do those three workloads look like. TM1 consists of extremely short running transactions. For their execution the conventional system acquires as many higher-level locks as row-level. In TPC-B, the ratio between the row-level to higher-level locks acquired is 2:1. Consequently, we expect the contention on the lock manager of the conventional system to be smaller when it executes the TPC-B benchmark than TM1. The conventional system is expected to scale even better when it executes TPC-C OrderStatus transactions, which they have even larger ratio of row-level to higher-level locks.

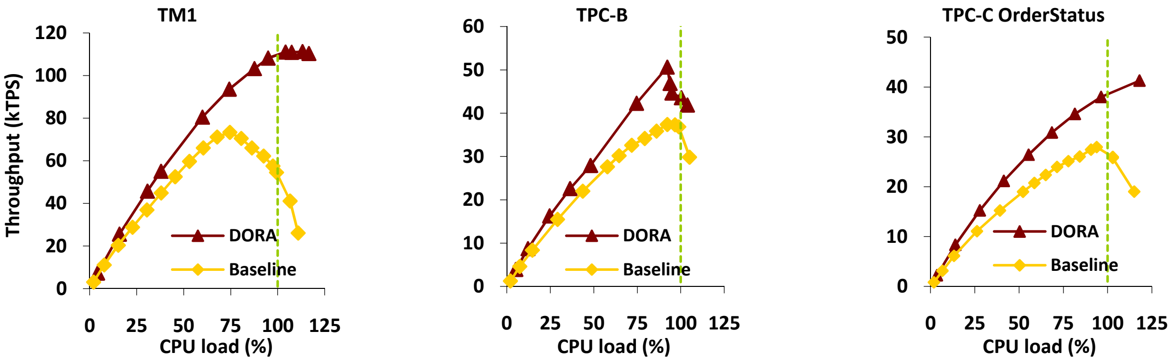

Figure 8. Performance of Baseline and DORA as the load in the system increases, executing transactions from the TM1 and TPC-B benchmarks, and TPC-C OrderStatus.

Figure 8 confirms our expectations. We plot the performance of both systems in the three workloads. The x-axis is the CPU load. We calculate the CPU load by adding to the measured CPU utilization, the time the threads spend in the runnable queue waiting for a processor to run. We see that the convention system experiences scalability problems, more profound in the case of the TM1 workload. DORA, on the other hand, scales its performance as much as the hardware resources allow. In Figure 1 we present the breakdown for the TM1 benchmark and the TPC-C OrderStatus transactions.

When the CPU load exceeds 100%, the performance of the conventional system in all three workloads collapses. This happens because the operating system needs to preempt threads, and in some cases it happens to preempt threads that are in the middle of contended critical sections. The performance of DORA, on the other hand, remains high, even though the CPU load is well beyond 100%; another proof that DORA reduces significantly the number of contended critical sections. In the case of the TPC-B benchmark, DORA is bottlenecked on the log manager, which is still centralized.

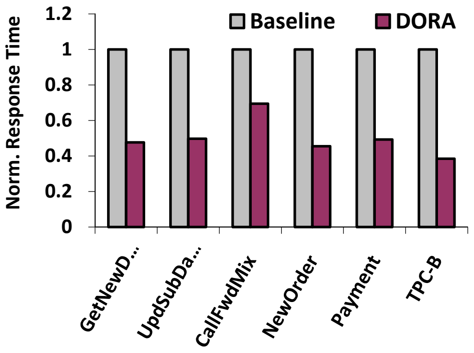

Figure 9. Response times for single transactions. DORA exploits the intra-transaction parallelism, inherent in many transactions.

5.4 Intra-transaction parallelism

DORA exploits intra-transaction not only as a mechanism for reducing the pressure to the contended centralized lock manager, but also as a way to improve response times when the workload does not saturate the available hardware. For example, in applications that exhibit limited concurrency due to heavy contention for logical locks, or for organizations that simply do not utilize their available processing power, intra-transaction parallelism is useful. In the experiment shown in Figure 9 we compare the average response time per request the Baseline system and DORA achieve in intra-parallel transactions from the three workloads. DORA exploits the available intra-transaction parallelism of the transactions and achieves lower response times. For example TPC-C NewOrder transactions are executed 2.4x faster under DORA.

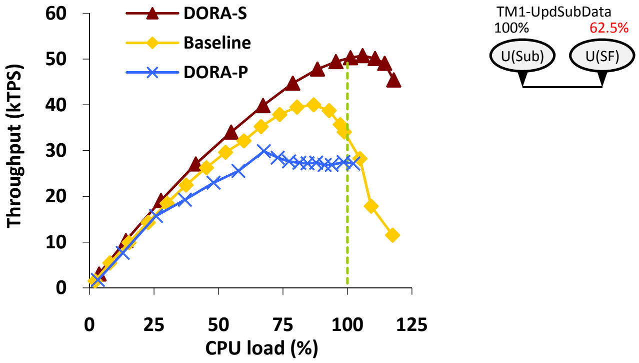

Figure 10. Performance on TM1-UpdateSubscriberData, a transaction with high abort rate.

A challenging workload is intra-parallel transactions that have non-negligible abort rates. In such workloads, DORA may end up serving actions from already aborted transactions. Figure 10, compares the throughput of the Baseline system and two variations of DORA. The first DORA variation (labeled DORA-P) executes the actions in parallel, checking regularly if any other action has aborted. The second (labeled DORA-S) executes the actions serially ensuring that no work will be wasted in case of an abort. The workload consists of TM1-UpdateSubscriberData transactions. The transaction flow graph for the parallel plan (DORA-P) is shown on the right side of the figure. This transaction consists of two actions, one of them failing 37.5% of the time. As we can see, the parallel plan is a bad choice for this workload. The DORA resource manager monitors the abort rates of transactions and switches to serial execution plans (by inserting empty rendezvous points between all actions) when the abort rates are high.

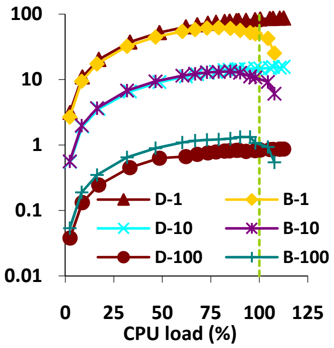

Figure 11. Performance on secondary actions.

5.5 Secondary Actions

In the following experiment, we measure the performance of DORA when it processes secondary actions. The workload is a transaction that probes the TM1 dataset for Subscribers using their number. The Subscriber number field is a candidate key. Thus, we can control the number of accessed Customer records. We execute transactions similar to the UpdateSubscriberData transaction and we alter the number of probed Customers, from 1 to 100. Figure 11 compares the performance of Baseline and DORA when 1, 10, and 100 Customer records are accessed. The overhead of the mechanism used by DORA to handle secondary actions is small, as long as, the number of probed records is kept low. On the other hand, DORA still does not face the scalability problems of the conventional system.

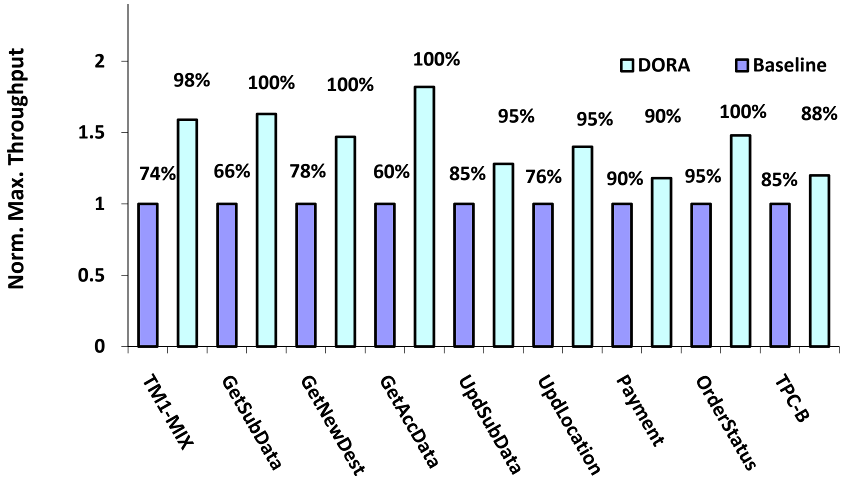

Figure 12. Comparison of the maximum throughput Baseline and DORA achieve per workload. The peak throughput is achieved at different CPU utilization.

5.6 Maximum throughput

With admission control the system can limit the number of outstanding transactions, and in turn, limit contention within the lock manager. Properly tuned, admission control allows the system to achieve the highest possible throughput, even if it means leaving the machine underutilized. In Figure 12 we compare the maximum throughput of Baseline and DORA achieve, if the systems were employing perfect admission control. For each system and workload we report the cpu utilization, when this peak throughput was achieved. DORA achieves higher peak throughput in all the transactions we study, and this peak is achieved closer to the hardware limits.

For the TPC-C and TPC-B transactions, DORA achieves relatively smaller improvements. This happens for two reasons. First, those transactions do not expose the same degree of contention within the lock manager, and leave little room for improvement. Second, some of the transactions (like TPC-C NewOrder and Payment, or TPC-B) impose great pressure on the log manager that becomes the new bottleneck.

6 Conclusions

Conventional transaction processing systems by applying a thread-to-transaction assignment of work fail to realize the full potential of the multicore processors. The resulting contention within the lock manager becomes burden on scalability. This paper shows the potential for thread-to-data assignment to eliminate this bottleneck and improve both performance and scalability. As multicore hardware continues to stress scalability within the storage manager and as DORA matures, the gap with traditional systems will only continue to widen.

7 Acknowledgements

We cordially thank Michael Abd El Malek, Kyriaki Levanti, and the members of the DIAS lab for their feedback and technical support throughout this research effort. We thank the PVLDB reviewers for their valuable feedback in early drafts of this paper. This work was partially supported by a Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds.

8 References

[1] Anon, et al. “A measure of transaction processing power.” In Datamation, 1985.

[2] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. “Disco: running commodity operating systems on scalable multiprocessors.” In ACM TOCS, 15(4), 1997.

[3] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. K. Tan, O. Tsatalos, S. White, and M. Zwilling. “Shoring Up Persistent Applications.” In Proc. SIGMOD, 1994.

[4] C. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. “Optimistic Intra-Transaction Parallelism on Chip Multiprocessors.” In Proc. VLDB, 2005.

[5] 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 Proc. SOSP, 2007.

[6] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. “The Gamma Database Machine Project.” In IEEE TKDE 2(1), 1990.

[7] H. Garcia-Molina, and K. Salem. “Sagas.” In Proc. SIGMOD, 1987.

[8] G. Graefe. “Hierarchical locking in B-tree indexes.” In Proc. BTW, 2007.

[9] J. Gray, and A. Reuter. “Transaction Processing: Concepts and Techniques.” Morgan Kaufmann, 1993.

[10] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. “OLTP Through the Looking Glass, and What We Found There.” In Proc. of SIGMOD, 2008.

[11] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. “QPipe: A Simultaneously Pipelined Relational Query Engine.” In Proc. SIGMOD, 2005.

[12] P. Helland. “Life Beyond Distributed Transactions: an Apostate's Opinion.” In Proc. CIDR, 2007.

[13] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. “A New Look at the Roles of Spinning and Blocking.” In Proc. DAMON, 2009.

[14] R. Johnson, I. Pandis, and A. Ailamaki. “Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines.” In Proc. DaMoN, 2008.

[15] R. Johnson, I. Pandis, and A. Ailamaki. “Improving OLTP Scalability with Speculative Lock Inheritance.” In Proc. VLDB, 2009.

[16] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. “Shore-MT: A Scalable Storage Manager for the Multicore Era.” In Proc. EDBT, 2009.

[17] A. Joshi. “Adaptive Locking Strategies in a Multi-node Data Sharing Environment.” In Proc. VLDB, 1991.

[18] E. Koskinen, and M. Herlihy. “Dreadlocks: Efficient Deadlock Detection.” In Proc. SPAA, 2008.

[19] T. Lahiri, V. Srihari, W. Chan, N. MacNaughton, and S. Chandrasekaran. “Cache Fusion: Extending shared-disk clusters with shared caches.” In Proc. VLDB, 2001.

[20] J. Mellor-Crummey, and M. Scot. “Algorithms for scalable synchronization on shared-memory multiprocessors.” ACM TOCS, 9(1), 1991.

[21] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. “ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.” In ACM TODS, 17(1), 1992.

[22] Nokia. Network Database Benchmark. At http://hoslab.cs.helsinki.fi/homepages/ndbbenchmark/

[23] Transaction Processing Performance Council. TPC - C v5.5: On-Line Transaction Processing (OLTP) Benchmark.

[24] M. Stonebraker, S. 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 Proc. VLDB, 2007.