Data-Oriented Transaction Execution

Ippokratis Pandis[1,2] ipandis@ece.cmu.edu
Ryan Johnson[1,2] ryanjohn@ece.cmu.edu
Nikos Hardavellas[3] nikos@northwestern.edu
Anastasia Ailamaki[2,1] natassa@epfl.ch

[1]Carnegie Mellon University Pittsburgh, PA, USA
[2]École Polytechnique Fédérale de Lausanne Lausanne, VD, Switzerland
[3]Northwestern University Evanston, IL, USA

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. DORA's design 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 maintains all the ACID properties. Evaluation of a prototype implementation of DORA on a multicore system demonstrates that DORA attains up to 4.8x higher throughput than a state-of-the-art storage engine when running a variety of synthetic and real-world OLTP workloads.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - transaction processing, concurrency.

General Terms

Design, Performance, Experimentation, Measurement.

Keywords

Data-oriented transaction execution, DORA, Multicore transaction processing, Latch contention, Lock manager.

1. INTRODUCTION

The diminishing returns of increasing on-chip clock frequency coupled with power and thermal limitations have led hardware vendors to place multiple cores on a single die and rely on thread-level parallelism for improved performance. 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 execute concurrently, contending for access to shared resources. Thread-parallel applications, such as online transaction processing (OLTP), running on multicores suffer of increasing delays in heavily-contended critical sections, with detrimental performance effects [14]. To tap the increasing computational power of multicores, the software systems must alleviate such contention bottlenecks and allow performance to scale commensurately with the number of cores.

OLTP is an indispensable operation in most enterprises. 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 is full of critical sections [14]. Consequently, these systems face significant performance and scalability problems on highly-parallel hardware. To cope with the scalability problems of transaction processing systems, researchers have suggested employing shared-nothing configurations [6] on a single chip [21] and/or dropping some of the ACID properties [5].

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.

The access patterns of each transaction, and consequently of each thread, are arbitrary and uncoordinated. To ensure data integrity, each thread enters a large number of critical sections in the short lifetime of each transaction it executes. Critical sections, 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 1(a) depicts the throughput per CPU utilization attained by a state-of-the-art storage manager as the CPU utilization increases. The workload consists of clients repeatedly submitting *GetSubscriberData* transactions from the TM1 benchmark (methodology detailed in Section 5). As

[1] Modern cores contain multiple hardware contexts, allowing them to interleave multiple instruction streams.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

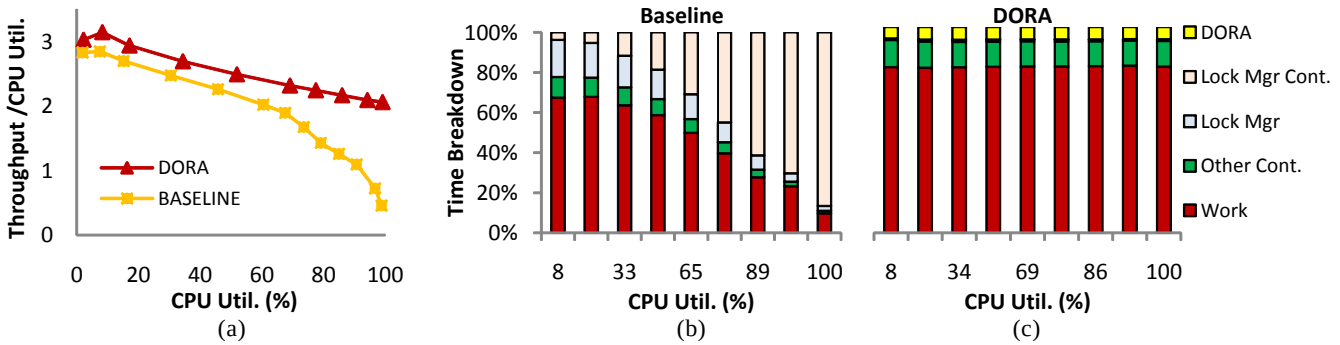

Figure 1. *DORA compared to Baseline when the workload consists of TM1-GetSubscriberData transactions. (a) The throughput per CPU utilization, as CPU utilization increases. (b) The time breakdown for the Baseline system. (c) The time breakdown for a DORA prototype.*

the machine utilization increases, the performance per CPU utilization drops. When utilizing all 64 hardware contexts the per hardware context performance drops by more than 80%. Figure 1(b) shows the contention within the lock manager quickly dominates. At 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. DORA decomposes the transactions to smaller actions according to the data they access, and routs them to the corresponding threads for execution. In essence, instead of pulling data (database records) to the computation (transaction), DORA distributes the computation to wherever the data is mapped.

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. By limiting thread interactions with the centralized lock manager, DORA eliminates the contention in it (Figure 1(c)) and provides better scalability (Figure 1(a)).

DORA exploits the low-latency, high-bandwidth inter-core communication of multicore systems. Transactions flow from one thread to the other with minimal overhead, as each thread accesses different parts of the database. Figure 2 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 [19] and TPC-C Order-Status transactions [20]. The DORA prototype eliminates the contention on the lock manager (Figure 2(a)). Also, it substitutes the centralized lock management with much lighter-weight thread-local locking mechanism (Figure 2(b)).

1.2 Contributions and document organization

This paper makes three contributions.

- We demonstrate that the conventional thread-to-transaction assignment results in contention at the lock manager that severely limits the performance and scalability of OLTP on multicores.
- We propose a data-oriented architecture for OLTP that exhibits predictable access patterns and allows to substitute the heavyweight centralized lock manager with a lightweight thread-local locking mechanism. The result is a shared-everything system that scales to high core counts without weakening the ACID properties.
- We evaluate a prototype DORA transaction processing engine and show that it attains up to 82% higher peak throughput against a state-of-the-art storage manager. Without admission control the performance benefits for DORA can be up to 4.8x. Additionally, when unsaturated DORA achieves up to 60% 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 explains why a conventional transaction processing system 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. [9] analyze the behavior of the single-threaded SHORE storage manager [3] running two transactions from the TPC-C benchmark. When executing the *Payment* transaction, the system spends 25% of its time on code related to logical locking, while with the *NewOrder* transaction it spends 16%. We corroborate the results and reveal the lurking problem of latch contention that makes the lock manager the system bottleneck when increasing the hardware parallelism.

Rdb/VMS [16] 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 [17], 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. A large number of threads may access the same resource at the same time leading to poor scalability. DORA ensures that the majority of resources are accessed by a single thread.

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

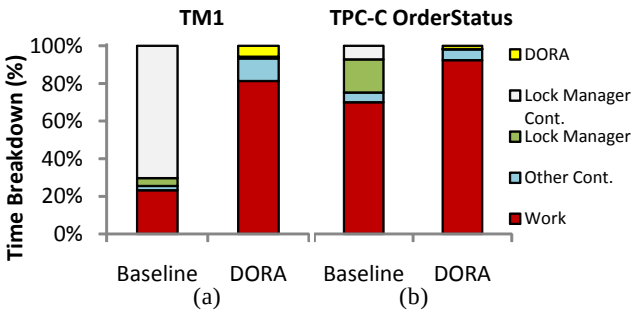

Figure 2. *Time breakdown for a conventional transaction processing system and a DORA prototype when all the 64 hardware contexts of a Sun Niagara II chip are fully utilized running (a) the TM1 benchmark, and (b) TPC-C OrderStatus transactions.*

Speculative lock inheritance (SLI) [13] detects "hot" locks at run-time and those locks may be held by the transaction-executing threads across transactions. SLI, similar to DORA, reduces the contention on the lock manager. However, it does not significantly reduce the other overheads inside the lock manager.

Advancements in virtual machine technology [2] enable the deployment of shared-nothing systems on multicores. In shared-nothing configurations, the database 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 [21] takes the shared-nothing approach to the extreme by deploying a set of single-threaded engines that serially execute requests, avoiding concurrency control. While Jones et al. [15] study a "speculative" locking scheme for H-Store for workloads with few multi-partition transactions. The complexity of coordinating distributed transactions [11][5] and the imbalances caused by skewed data or requests are significant problems for shared-nothing systems. DORA, by being shared-everything, is less sensitive to such problems and can adapt to load changes more readily. In the Appendix we discuss the benefits of DORA from not being shared-nothing.

Staged database systems [10] share similarities with DORA. A staged system splits queries into multiple requests which may proceed in parallel. The splitting is operator-centric and designed for pipeline parallelism. Pipeline parallelism, however, has little to offer to typical OLTP workloads. 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 uses 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 75% 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. THE LOCK MANAGER AS SOURCE OF CONTENTION

In this section, we explain why in typical OLTP workloads the lock manager of conventional systems is often the first contended component and the obstacle to scalability.

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.[1] The lock manager is responsible for maintaining isolation between concurrently-executing transactions, providing 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.

Next, we describe the lock manager of the Shore-MT storage engine [14]. 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. In the Sun Niagara II hardware, our testbed, and for the CPU loads we are using in this study (<120%) spinning-based implementations outperform any known solution involving blocking [12].

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. 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 mode the transaction must block. Finally, the lock is unlatched and the request returns. Each transaction maintains a list of all its lock requests in the order that it acquired them. At transaction completion, the 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 to compute the new lock mode and to find any pending requests which may now be granted.

The effort required to grant or release a lock grows with the number of active transactions, due to longer lists of lock requests. 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 longer lists of lock requests, with the increased number of threads executing transactions and contending for locks leads to detrimental results.

Figure 3 shows where the time is spent inside the lock manager of Shore-MT when it runs the TPC-B benchmark as the system utilization increases on the x-axis. The breakdown is on the time it takes to acquire 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).

[1] 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.

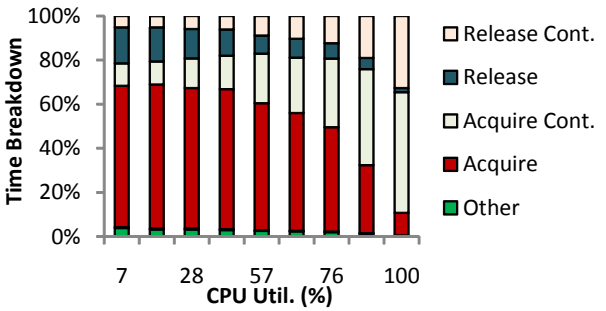

Figure 3. *Time breakdown inside the lock manager of Shore-MT when it runs the TPC-B benchmark and the load increases.*

4. A DATA-ORIENTED ARCHITECTURE FOR OLTP

In this section we present the design of an OLTP system which employs the 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's functionality includes three basic operations: (a) It binds worker threads to disjoint subsets of the database; (b) It distributes the work of each transaction across transaction-executing threads according to the data accessed by the transaction; (c) 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 [20].

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 sets of records, or *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. The only requirement for the routing rule is that each possible record of the table to map to a unique dataset. With the routing rules each table is logically decomposed into disjoint sets of records. All data resides in the same bufferpool and the rules imply no physical separation or data movement.

The columns used by the routing rule are called the *routing fields*. The routing fields can be any combination of the columns of the table. The columns of the primary or candidate key have been shown in practice to work well as routing fields. In the *Payment* transaction example, we assume that the Warehouse id column is the routing field in each of the four accessed tables. 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 that table, and the available hardware resources.

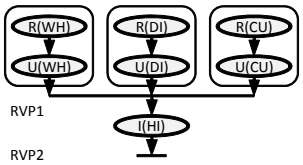

Figure 4. *The transaction flow graph of TPC-C Payment.*

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 access to a single or a small set of records from the same table. The *identifier* of an action identifies the set of records this action intends to access. Depending on the type of the access the identifier can be a set of values for the routing fields or the empty set. Two consecutive actions can be merged if they have the same identifier (refer to the same set).

The more specific the identifier of an action is, the easier is for DORA to route the action to its corresponding executor. That is, actions whose identifier are at least all the routing fields are directed to their executor by consulting the routing rule of the table. Actions whose identifier is a subset of the routing field set may map to multiple datasets. In that case, the action is broken to a set of smaller actions, each of them resized to correspond to a dataset. Secondary index accesses typically fall in this category. Finally, actions 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 DORA handles secondary actions.

DORA uses shared objects across actions of the same transaction in order to control the distributed execution of the transaction and to transfer data between actions with data dependencies. 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 separate the execution of the transaction 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.

The transaction flow graph for the *Payment* transaction is shown in Figure 4. Each *Payment* transaction probes a Warehouse and a District record and updates them. In each case, both actions (record retrieval and update) have the same identifier and they can be merged. The Customer, on the other hand, 60% of the time is probed through a secondary index and then updated. That secondary index contains the Warehouse id, the District id, and the Customer's last name columns. If the routing rule on the Customer table uses only the Warehouse id or the District id columns, then the system knows which executor is responsible for this secondary index access. If the routing rule uses also the Customer id column of the primary

key, then the secondary index access needs to be broken to smaller actions that cover all the possible values for the Customer id. If the routing rule uses only the Customer id, then the system cannot decide which executor is responsible for the execution 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 record update have the same identifier and can be merged. Finally, an RVP separates the *Payment* transaction to two phases, because of the data dependency between the record insert on the History table and the other three record probes.

Payment's specification requires the Customer to be randomly selected from a remote Warehouse 15% of time. In that case, a shared-nothing system that partitions the database on the Warehouse will execute a distributed transaction 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 routes all the actions that indent to operate on the same dataset to one executor. The executor is responsible for maintaining isolation and ordering across conflicting actions. Next, we describe how DORA executes transactions. A detailed example of the execution of one transaction in DORA is presented at the Appendix (Section A.1).

Each executor has three data structures associated with it: a *queue of incoming actions*, a *queue of completed actions*, 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 input to the local lock table are action identifiers. The local locks have only two modes, shared and exclusive. Since the action identifiers may contain only a subset of the primary key, the locking scheme employed is similar to that of key-prefix locks [8]. Once an action acquires the local lock it can proceed without centralized concurrency control. Each action holds the local lock it acquired until the overall transaction commits (or aborts). At the terminal RVP each transaction first waits for a response from the underlying storage manager that the commit (or abort) has completed. Then, it enqueues all the actions that participated in the transaction to the completion queues 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. Transactions that intend to modify large data ranges which span multiple datasets or cover the entire table (e.g., a table scan, or an index or table drop) enqueue an action to every executor operating on that particular table. Once all the actions are granted access, the "multi-partition" transaction can proceed. In transaction processing workloads such operations already hamper concurrency, and therefore occur rarely in scalable applications.

4.2 Challenges

In this section we describe three challenges in the DORA design. Namely, we describe how DORA executes record inserts and deletes, how it handles secondary actions, and how it avoids deadlocks. In the Appendix (Section A.2.1) we also discuss how DORA efficiently resizes the datasets to balance the load.

4.2.1 Record inserts and deletes

Record probes and updates in DORA require 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 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 lock (for inserting the History record) of the 19 it would acquire if conventionally executed.

4.2.2 Secondary Actions

The problem with secondary actions is that the system does not know which executor is responsible for their execution. To resolve this difficulty, the indexes whose accesses cannot be mapped to executors store the RID as well as all the routing fields at each leaf entry. The RVP-executing thread of the previous phase executes those secondary actions and uses the additional information to determine which executor should perform the access of the record in the heap file.

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 primary index Idx1 and a secondary index Idx2 which is 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 can add a 'deleted' flag to the entries of Idx2. When a transaction deletes a record it does not remove the entry from the index; any transaction which attempts to access the record will go through its owning executor and find that it was, 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, the B-Tree's leaf-split algorithm can be modified 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

The transactions in DORA may block on local lock tables. Hence, the storage manager must provide an interface for DORA 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.[1] This ensures that transactions with the same transaction flow graph will never deadlock with each other. That is, two transactions with the same transaction flow graph will deadlock only if their conflicting requests are processed 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 transaction which will enqueue its actions first it will finish before the other.

4.3 Prototype implementation

In order to evaluate the DORA design, we implemented a prototype DORA OLTP engine over the Shore-MT storage manager [14]. Shore-MT is a modified version of the SHORE storage manager [3] with a multi-threaded kernel. 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 [18]. We use Shore-MT because it has been shown to scale better than any other open-source storage engine [14].

Our prototype does not have a optimizer which transforms regular transaction code to transaction flow graphs, neither Shore-MT has a front-end. Thus, all transactions are partially hard-coded. The database metadata and back-end processing are schema-agnostic and general purpose, but the 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.[2]

The prototype is implemented as a layer over Shore-MT. Shore-MT's sources are linked directly to the code of the prototype. Modifications to Shore-MT were minimal. 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.

5. PERFORMANCE EVALUATION

We use one of the most parallel multicore processors 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. 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 three areas. First, we measure how effectively DORA reduces the interaction with the centralized lock manager and what is the impact on performance (Section 5.2). Then, we quantify how DORA exploits the intra-transaction parallelism of transactions (Section 5.3). Finally, we put everything together and compare the peak performance DORA and Shore-MT achieve, if a perfect admission control mechanism is used (Section 5.4).

5.1 Experimental setup & workloads

Hardware: We perform all our experiments on a Sun T5220 "Niagara II" machine configured with 32GB of RAM and running Sun Solaris 10. The Sun 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. Given the limited budget and that we are interested in the behavior of the systems when a large number of hardware contexts are utilized, we store the database and the log on an in-memory file system. This setup allows us to saturate the CPU, yet it exercises all the codepaths of the storage manager. Preliminary experiments using high performing Flash drives indicate that the relative behavior of the two systems remains the same.

Workloads: We use transactions from three OLTP benchmarks: Nokia's Network Database Benchmark or TM-1 [19], TPC-C [20], and TPC-B [1]. Business intelligence workloads, such as the TPC-H benchmark, 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.

TM1 consists of seven transactions, operating on four tables, implementing various operations executed by mobile networks. Three of the transactions are read-only while the other four perform updates. The transactions are extremely short, yet exercise all the codepaths in typical transaction processing. Each transaction accesses only 1-4 records, and must execute with low latency even under heavy load. 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 creation to final delivery and payment. We use a 150 warehouse 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 they 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

[1] There is a strict ordering between executors. The threads acquire the latches in that order, avoiding deadlocks on the latches of the incoming queues of executors.

[2] http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp

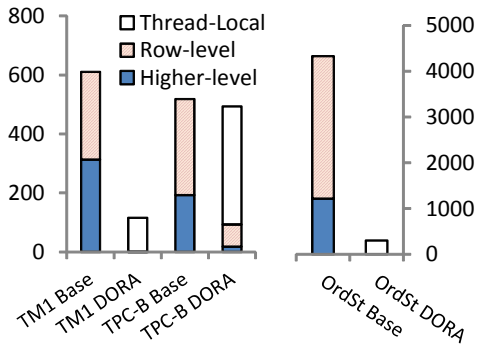

Figure 5. *Locks acquired per 100 transactions by Baseline and DORA in three workloads.*

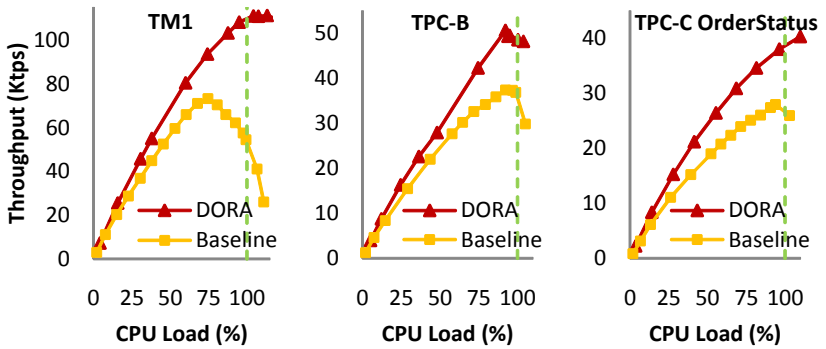

Figure 6. *Performance of Baseline and DORA as the load in the system increases, executing transactions from the TM1 and TPC-B benchmarks, and TPC-C OrderStatus.*

multiple times, and the measured relative standard deviation is less than 5%. We use the highest level of optimization options using the Sun CC v5.10 compiler. For measurements that needed profiling, we used tools from Sun Studio 12 suite. The profiling tools impose a certain overhead (~15%) but the relative behavior between the two systems remains the same.

5.2 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 *GetSubscriberData* transactions of the TM1 benchmark.

The results are shown in Figure 1. The left graph on the y-axis shows the throughput per CPU utilization of the two systems as the CPU utilization increases. The other two graphs show the time 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 centralized lock manager operations it eliminates even when those are uncontended. It is worth mentioning that the *GetSubscriberData* is a read-only transaction. Yet the Baseline system suffers from contention in the lock manager. That is because the threads will contend even if they want to acquire the same lock in compatible mode.

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 to report the number and the type of the acquired locks. Figure 5 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. The locks are categorized in three types. The row-level locks, the locks of the centralized lock manager that are not row-level (labeled higher level), and the DORA local locks.

In typical OLTP workloads the contention for the row-level locks is limited, because there is a very large number of randomly accessed records. But, 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. Figure 5 shows that DORA has only minimal interaction with the centralized lock manager. The non record-level lock acquired by DORA at TPC-B is due to space management (allocation of a new extend of pages).

Figure 5 gives an idea on how those three workloads behave. 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 6 confirms our expectations. We plot the performance of both systems in the three workloads. The x-axis is the offered CPU load. We calculate the offered 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 Baseline system experiences scalability problems, more profound in the case of TM1. DORA, on the other hand, scales its performance as much as the hardware resources allow.

When the offered 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; another proof that DORA reduces the number of contended critical sections.

Figure 2 shows the detailed time breakdown for the two systems at 100% CPU utilization for the TM1 and the TPC-C *OrderStatus* workloads. DORA outperforms the Baseline system in OLTP workloads independently of whether the lock manager of the Baseline system is contended or not.

5.3 Intra-transaction parallelism

DORA exploits intra-transaction not only as a mechanism for reducing the pressure to the contended centralized lock manager, but also for improving 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 7 we compare the average response time per request the Baseline system and DORA achieve when a single client submits intra-parallel transactions from the three workloads and the log resides in a in-memory file system. DORA exploits the available intra-

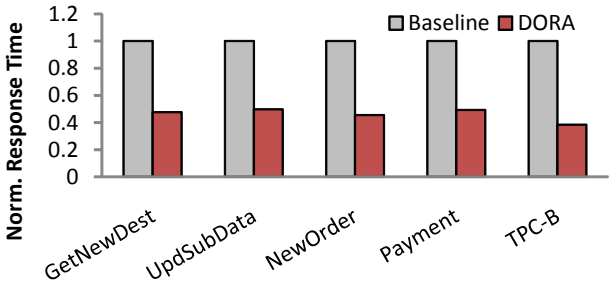

Figure 7. *Response times for single transactions. DORA exploits the intra-transaction parallelism, inherent in many transactions.*

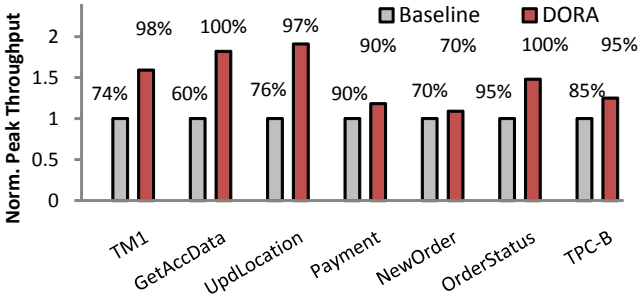

Figure 8. *Comparison of the maximum throughput DORA and Baseline achieve per workload. The peak throughput is achieved at different CPU utilization.*

transaction parallelism of the transactions and achieves lower response times. For example TPC-C NewOrder transactions are executed 60% faster under DORA.

5.4 Maximum throughput

Admission control can limit the number of outstanding transactions, and in turn, limit contention within the lock manager of the system. Properly tuned, admission control allows the system to achieve the highest possible throughput, even if it means leaving the machine underutilized. In Figure 8 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. CONCLUSION

The thread-to-transaction assignment of work of conventional transaction processing systems fail to realize the full potential of the multicores. 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 conventional 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. 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.” Datamation, 1985.

[2] E. Bugnion, et al. “Disco: running commodity operating systems on scalable multiprocessors.” ACM TOCS, 15(4), 1997.

[3] M. Carey, et al. “Shoring Up Persistent Applications.” In Proc. SIGMOD, 1994.

[4] C. Colohan, et al. “Optimistic Intra-Transaction Parallelism on Chip Multiprocessors.” In Proc. VLDB, 2005.

[5] G. DeCandia, et al. “Dynamo: Amazon's Highly Available Key-value Store.” In Proc. SOSP, 2007.

[6] D. J. DeWitt, et al. “The Gamma Database Machine Project.” 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] S. Harizopoulos, et al. “OLTP Through the Looking Glass, and What We Found There.” In Proc. SIGMOD, 2008.

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

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

[12] R. Johnson, et al. “A New Look at the Roles of Spinning and Blocking.” In Proc. DaMoN, 2009.

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

[14] 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.

[15] E. Jones, D. Abadi, and S. Madden. “Low Overhead Concurrency Control for Partitioned Main Memory Databases.” In Proc. SIGMOD, 2010.

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

[17] T. Lahiri, et al. “Cache Fusion: Extending shared-disk clusters with shared caches.” In Proc. VLDB, 2001.

[18] C. Mohan, et al. “ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.” ACM TODS 17(1), 1992.

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

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

[21] M. Stonebraker, et al. “The End of an Architectural Era (It's Time for a Complete Rewrite).” In Proc. VLDB, 2007.

A. APPENDIX

The appendix has four sections. First, for a better understanding of the DORA system, we describe in detail the execution of one transaction in DORA (Section A.1). Then, we discuss two benefits of DORA from being a shared-everything rather than a shared-nothing system (Section A.2). Next, in order to give a better intuition of the differences between conventional execution and DORA, we present graphically the differences of their data access patterns and discuss the potential of exploiting the regularity in DORA's accesses (Section A.3). Finally, we discuss another challenge for DORA which is transactions with non-negligible abort rates and intra-transaction parallelism (Section A.4).

A.1 Detailed transaction execution example

In this section we describe in detail the execution of one *Payment* transaction from the TPC-C benchmark by DORA. As a reminder, 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. The transaction flow graph for the *Payment* transaction is shown in Figure 4.

Figure 9 shows the execution flow in DORA for a *Payment* transaction. Each circle is color-coded to depict the thread which executes that step. In total there are 12 steps for executing this transaction.

Step 1: The execution of the transaction starts from the thread that receives the request (e.g., from the network). That thread, also called dispatcher, enqueues the actions of the first phase of the transaction to the corresponding executors.

Step 2: Once the action reaches the head of the incoming queue it is picked by the corresponding executor.

Step 3: Each executor probes its local lock table to determine whether it can process the action it is currently serving or not. If there is a conflict, the action is added to a list of blocked actions. Its execution will resume once the transaction whose action blocks the particular action is completed, either committed or aborted. Otherwise, the executor executes the action most of the time without system-wide concurrency control.

Step 4: Once the action is completed, the executor decrements the counter of the RVP of the first phase (RVP1).

Step 5: If it is the last action to report to the RVP, the executor of that action that zeroed the RVP initiates the next phase by enqueueing the corresponding action to the History executor.

Step 6: The History table executor does the same routine, picking the action from the head of the incoming queue.

Step 7: The History table executor probes the local lock table.

Step 8: The Payment transaction inserts a record to the History table, and for the reason we explained at Section 4.2.1, the execution of that action needs to interface the centralized lock manager.

Step 9: Once the action is completed, the History executor updates the terminal RVP and calls for the transaction commit.

Step 10: When the underlying storage engine returns from the system-wide commit (with the log flush and the release of any centralized locks), the History executor enqueues the identifiers of all the actions back to their executors.

Step 11: The executors pick the committed action identifier.

Step 12: The executors remove the entry from their local lock table, and search the list of pending actions for action which may now proceed.

The detailed execution example, and especially steps 9-12, show that the commit operation in DORA is similar with 2 phase commit protocol in the sense that the thread that calls the commit (coordinator in 2PC) also sends messages to release the local locks to the various executors (participants in 2PC). The main difference with the traditional 2 phase commit is that the messaging happens asynchronously and that the participants do not have to vote. Since all the modifications are logged under the same transaction identifier there is no need for additional messages and log inserts (separate Prepare and Commit messages and records [A3]). That is, the commit is a one-off operation in terms of logging but still involves the asynchronous exchange of a message from the coordinator to the participants for the thread-local locking.

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

A.2 DORA vs. shared-nothing

One of the main benefits of the DORA design is that the system remains shared-everything, but at the same time the threads most of the time operate on thread-private data structures. That is, even though the datasets partition each table to disjoint sets of records and those datasets are assigned

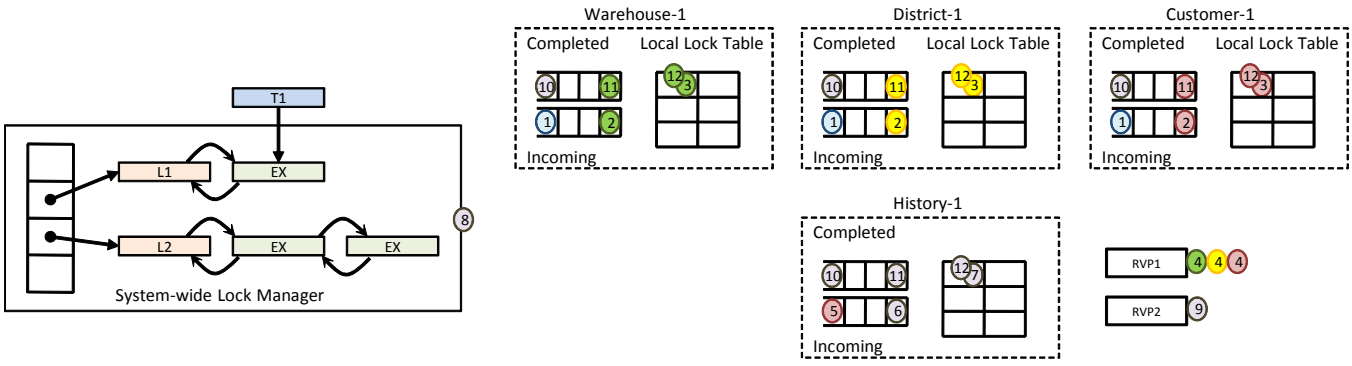

Figure 9. *Execution example of the TPC-C Payment transaction in DORA.*

to different threads, the partitioning is purely logical, not physical.

A shared-nothing system may face significant performance problems due to either imbalances caused by skewed data or requests or due to data accesses which do not align with the partitioning scheme. Hence, the performance of such systems is sensitive to the application design [A2]. Shared-nothing systems need applications that partition their workload in a way that there is a small fraction of multi-partition transactions, the load is balanced across partitions, and the non-partitioning aligned data accesses are minimal (e.g., [A8]).

In this section we argue that DORA, by being shared-everything, is less sensitive than shared-nothing systems to such problems. The following subsections compare how a typical shared-nothing system and DORA adjust at runtime to changes in access patterns and data skew (Section A.2.1) and how they handle non-partitioning aligned data accesses (Section A.2.2).

A.2.1 Load-balancing

Typically a shared-nothing system responds to imbalances at the load by resizing or replicating the tables of each partition. To resize a table a shared-nothing system needs to physically delete a region of records from one partition and insert them to another. During the record transfer, the execution of requests that access the resizing table is difficult. The associated costs for the record transfer are not negligible. For example, all the indexes on the tables which are being resized need to be updated. The larger the number of records that need to be transferred or the larger the number of indexes on the resizing tables, the larger the cost of the repartitioning.

On the other hand, DORA adapts to load imbalances by modifying the routing rule of each table and spawning new executors, if needed. The DORA resource manager monitors the load of each executor and reacts if the average load assigned to an executor is disproportional larger than the rest. The typical reaction is to resize the dataset assigned to each executor in order to balance the load. Resizing the datasets, however, is not for free.

Every routing rule modification involves two executors, one whose area of responsibility shrinks or *shrinking executor* and one who is assigned a larger fraction of the table or *growing executor*. To achieve the resize the resource manager enqueues a system action to the incoming queue of each of the two executors, and updates the routing rule of the table. When the executor whose range gets smaller reaches that system action, it stops serving new actions until all the actions already served by that executor leave the system. That is, the shrinking executor needs to drain all its in-flight actions. Otherwise, it may execute actions which intend to modify the data region which is responsibility of the growing executor after the resize. The growing executor can continue serving actions involving its old dataset normally. Once the in-flight actions of the shrinking executor are drained, the growing executor can start serving actions that fall into the newly assigned region.

There are also some indirect overheads associated with modifying the datasets at runtime. Cache locality is lost for the data that need to be accessed by an executor running on a different core. The data have to move to a different core, and the system experiences increased on-chip coherence traffic to collect the modified data and increased miss rates while warming up the cache. Thus, adapting at runtime must be applied judiciously to balance the trade off. Still, it is less painful than for a shared-nothing system.

A.2.2 Non-partitioning aligned data accesses

Some applications may need to access data in a way that does not align with the partitioning scheme. For example, consider a partitioning of the Customer table of the TPC-C database based on the Warehouse id, and a transaction that tries to access the Customers based on their name. A secondary index needs to be built on the name column of the Customer table, if this transaction is executed frequently enough. In a shared-nothing system, such a secondary index would have been built on every partition.

The execution of this transaction in a shared-nothing system would involve all the N partitions, making N secondary index probes. Out of those N probes only few of them may actually return records.

In DORA such accesses are typically secondary actions and Section 4.2.2 describes how they are handled. A single index is maintained. This index for each leaf entry has not only the RID but also the routing fields for that entry. Whenever such a transaction is executed a single probe takes place. The RIDs are collected and then, based on the routing fields for each entry, each record is accessed by its corresponding executor. For the execution of this transaction only the executors that contain records returned from the secondary index probe will be involved.

A.3 Comparison of access patterns and potential

DORA is not a shared-nothing system. At the same time, it is not a conventional shared-everything one. With the main difference stemming from the different way it assigns work to the various threads.

The difference between the execution of transactions based on the thread-to-transaction (i.e., conventional) assignment policy and one based on the thread-to-data (i.e., DORA) becomes easily apparent with visual inspection. Figure 10(a) depicts the accesses issued by each worker thread of a conventional transaction processing system, to each one of the records of the District table in a TPC-C database with 10 Warehouses. The system is configured with 10 worker threads and the workload consists of 20 clients repeatedly submitting *Payment* transactions from the TPC-C benchmark[1], while we trace only 0.7 seconds of execution. The accesses are totally uncoordinated. To guarantee the consistency of the data from those uncoordinated accesses the system needs to use latches whose cost increases with hardware parallelism.

On the other hand, Figure 10(b) illustrates the effect of the data-oriented assignment of work on data accesses. It plots the data access patterns issued by the prototype DORA system, which employs the thread-to-data assignment, against the same workload as Figure 10(a). The accesses in DORA are coordinated and show regularity.

In this paper, we exploit this regularity to reduce the interaction with the centralized lock manager, and hence reduce the number of expensive latch acquisitions and releases. We exploit it also to improve single thread performance by replacing the execution of the expensive lock manager code with the execution of a much lighter-weight thread-local locking mechanism. But, the potential of the DORA execution does not stop there. Potentially DORA's access pattern can be

[1] The system and workload configuration are kept small to enhance the graph's visibility.

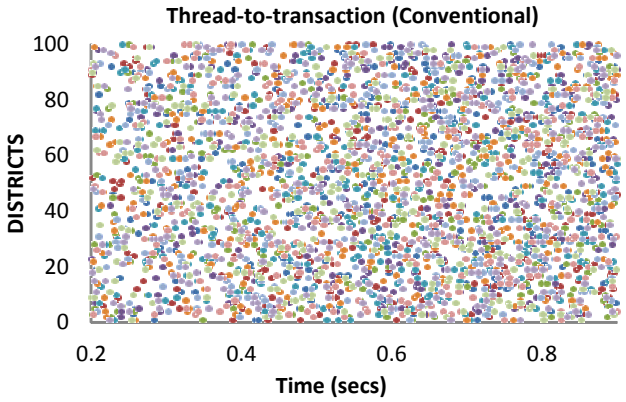

Figure 10a. *Trace of the record accesses by the threads of a conventional system; data accesses are uncoordinated and complex.*

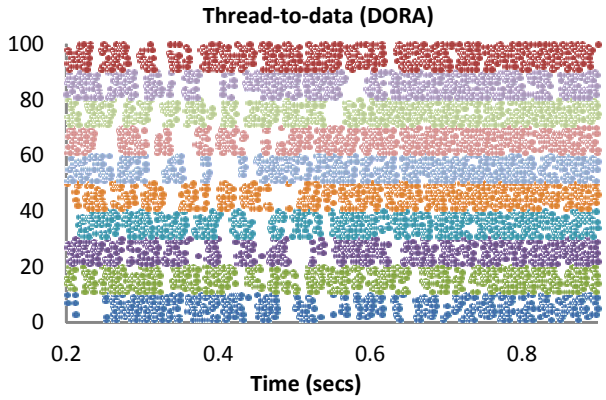

Figure 10b. *Trace of the record accesses by the threads of a DORA system; data accesses are coordinated and show regularity.*

exploited in order to improve both the I/O, as well as, the microarchitectural behavior of the OLTP.

In particular, the I/O executed during conventional OLTP is random and low performing.[1] The DORA executors can buffer the I/O requests and issue them in batches since those I/Os are expected to target pages that are physically close to each other, improving the I/O behavior.

Furthermore, the main characteristic of the micro-architectural behavior of conventional OLTP systems is the very large volume of shared read-modify accesses by multiple processing cores [A1]. Accesses which, unfortunately, are also highly unpredictable [A6]. Due to the two aforementioned reasons, emerging hardware technologies such as reactive distributed on-chip caches (e.g., [A4][A1]) and/or the most advanced hardware prefetchers (e.g., [A7]) fail to significantly improve the performance of conventional OLTP. Since DORA's design is based on that the majority of the accesses to a specific data region are coming by a specific thread, we expect a "friendlier" behavior which can realize the full potential of the latest hardware developments by providing more private and predictable memory accesses.

As future work, we plan to explore the potential of the DORA design in those two fronts.

A.4 Intra-transaction parallelism with aborts

DORA is designed around intra-transaction parallelism. The low-latency and high-bandwidth inter-core communication in modern multicores allows the execution of the DORA transactions to flow from one thread to the other with minimal overhead, as each transaction accesses different parts of the database. One challenge with intra-transaction parallelism are transactions with non-negligible abort rates. For example, the TM1 benchmark is unusual in that a large fraction of transactions (~25%) fail due to invalid inputs. In such workloads, DORA may end up executing actions from already-aborted transactions.

There are two execution strategies DORA can follow for such intra-parallel transactions with high abort rates. The first execution strategy, is to continue to execute such transactions in parallel and to check frequently for aborts. The second is to serialize the execution. That is, even though there is opportunity for actions from such transactions to proceed in parallel, DORA can be pessimistic and execute them serially. This execution strategy ensures that if an action aborts there is no work wasted by the execution of any other parallel action.

[1] As a proof, the performance of conventional OLTP systems is significantly improved with the usage of Flash-based storage technologies which exhibit high random access bandwidth [A5].

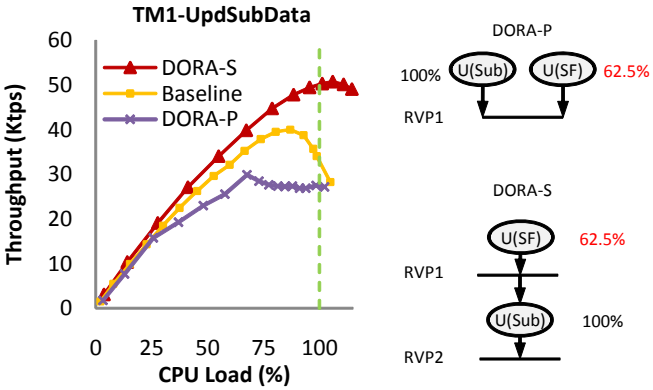

Figure 11. *Performance on TM1-UpdateSubscriberData, a transaction with high abort rate.*

Figure 11 compares the throughput of the Baseline system and the two variations of DORA when an increasing number of clients submit repeatedly *UpdateSubscriberData* transactions from the TM1 benchmark. This transaction, whose parallel and serial transaction flow graphs are depicted on the right side of the figure, consists of two independent actions. One action attempts to update a Subscriber and always succeeds. The other action attempts to update a corresponding SpecialFacility entry and it succeeds only 62.5% of the time, failing the rest of the time due to wrong input.

We plot the throughput of both two execution strategies for DORA. The parallel execution is labeled DORA-P, whereas the serial execution, which first attempts to update the SpecialFacility and only if that succeeds it tries to update the Subscriber, is labeled DORA-S. As we can see, the parallel plan is a bad choice for this workload. DORA-P achieves less performance than even the Baseline, whereas DORA-S scales as expected.

The DORA resource manager monitors the abort rates of entire transactions and individual actions in each executor. When the abort rates are high, DORA switches to serial execution plans by inserting empty rendezvous points between actions of the same phase. Still, it remains a challenge to apply optimizations specific for DORA transactions.

APPENDIX REFERENCES

[A1] B. M. Beckmann, and D. A. Wood. “Managing Wire Delay in Large Chip-Multiprocessor Caches.” In Proc. MICRO, 2004.

[A2] C. Curino, Y. Zhang, E. Zones, and S. Madden. “Schism: a Workload-Driven Approach to Database Replication and Partitioning.” In Proc. VLDB, 2010.

[A3] J. Gray, and A. Reuter. “Transaction Processing: Concepts and Techniques.” Morgan Kaufmann, 1993.

[A4] N. Hardavellas, M. Ferdman, B. Falsafi and A. Ailamaki. “Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches.” In Proc. ISCA, 2009.

[A5] S.-W. Lee, et al. “A Case for Flash Memory SSD in Enterprise Database Applications.” In Proc. SIGMOD, 2008.

[A6] S. Somogyi, et al. “Memory Coherence Activity Prediction in Commercial Workloads.” In Proc. WMPI, 2004.

[A7] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. “Spatio-Temporal Memory Streaming.” In Proc. ISCA, 2009.

[A8] VoltDB. “Developing VoltDB Applications.” At: http://voltdb.com/content/webinar-recording-developing-voltdb-apps