Exploiting Single-Threaded Model in Multi-Core Systems

Chang Yao[‡], Divyakant Agrawal[♯], Gang Chen[§], Qian Lin[‡]
Beng Chin Ooi[‡], Weng-Fai Wong[‡], Meihui Zhang[†]

[‡]National University of Singapore, [♯]University of California at Santa Barbara
[§]Zhejiang University [†]Singapore University of Technology and Design

ABSTRACT

The widely adopted single-threaded OLTP model assigns a single thread to each static partition of the database for processing transactions in a partition. This simplifies concurrency control while retaining parallelism. However, it suffers performance loss arising from skewed workloads as well as transactions that span multiple partitions. In this paper, we present a *dynamic single-threaded in-memory OLTP system*, called LADS, that extends the simplicity of the single-threaded model. The key innovation in LADS is the separation of dependency resolution and execution into two non-overlapping phases for batches of transactions. After the first phase of dependency resolution, the record actions of the transactions are partitioned and ordered. Each independent partition is then executed sequentially by a single thread, avoiding the need for locking. By careful mapping of the tasks to be performed to threads, LADS is able to achieve a high degree of balanced parallelism. We evaluate LADS against H-Store, a partition-based database; DORA, a data-oriented transaction processing system; and Silo, a multi-core in-memory OLTP engine. The experimental study shows that LADS achieves up to 20× higher throughput than existing systems and exhibits better robustness with various workloads.

1. INTRODUCTION

Online transaction processing (OLTP) is at the core of the Internet economy. However, while hardware has been scaling according to Moore's Law [37], the scaling of transaction processing has been lackluster. Figure 1 shows a historical plot of relative single-threaded CPU performance [1] and IBM's TPC-C performance [3]. As a reference, we can see a widening gap between hardware capability scaling and that of OLTP. As we move into the multi-core era, servers will not only have much more significant processing capabilities but also much larger memory [23, 46]. Unfortunately, the historical trend seems to cast doubt as to whether the ever-improving hardware capabilities can be efficiently translated to significantly better OLTP performance. Harizopoulos et al. [17] showed that traditional database management systems (DBMS) spent substantial amounts of time on logging (≈12%), locking/latching (≈30%) and buffer management (≈35%). In other words, even if the entire database resides in memory and transactions are short-lived, the overheads of managing concurrent transaction execution would severely limit the scalability, especially when there are heavily-contended critical sections [32].

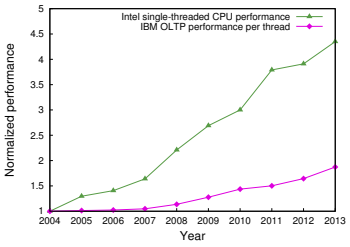


Figure 1: A historical plot illustrates an increasing trend and gap of CPU and OLTP performance.

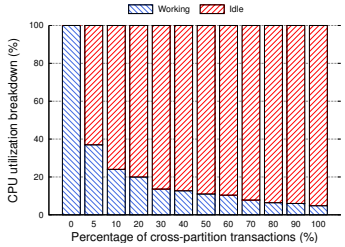


Figure 2: CPU utilization of H-Store that supports single-threaded model.

Traditional concurrency control protocols can be categorized into lock-based protocols and timestamp-based protocols. For lock-based protocols, lock thrashing and deadlock avoidance are the main challenges. For timestamp-based protocols, the main issues are the high abort rate and the need for a well coordinated timestamp allocation. Recently, the single-threaded model is widely used [20, 21, 32, 39] to improve the efficiency of in-memory database systems. In this model, each worker thread is uniquely assigned to a static partition of the database. When a transaction arrives, a worker that is responsible for its processing will obtain all the locks of the partitions to be accessed. By doing so, all the critical sections that are contended for can be resolved in advance. In a scenario where most transactions only need to access data in a single partition, the threads can work independently, achieving a high degree of parallelism. However, in practice, no matter how well the database has been partitioned, transactions that span multiple partitions cannot be totally avoided. Such transactions typically cause a substantial amount of blocking as threads tend to contend with each other to acquire all the necessary locks, resulting in more CPU idle time. Furthermore, continuously changing workloads may cause load imbalance where some worker threads experience significant blocking while others have too much work to handle. For a statically partitioned database where each partition is assigned to a thread, a skewed workload could lead to skewed CPU utilization. Consequently, it significantly reduces the efficiency of the entire system. As illustrated in Figure 2, the percentage of CPU idle time of H-Store [20] increases with the increase of cross-partition transactions. When there is no cross-partition transaction in the workload, each worker thread can process transactions independently without any blocking. When the workload contains cross-partition

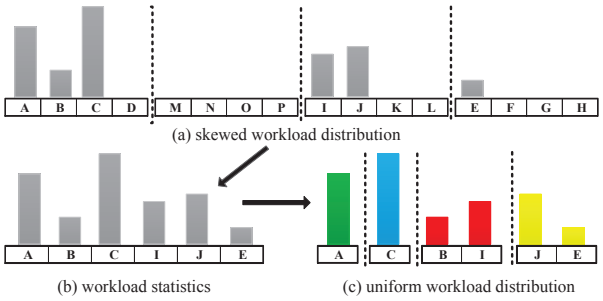


Figure 3: Dynamic partitioning based on actual workloads.

transactions, potential contention among worker threads appears with its probability rising along the increment of cross-partition transactions. To resolve contention, thread blocking is usually inevitable, which degrades the CPU utilization.

In this paper, we examine the design of multi-core in-memory OLTP systems with the goal of improving the throughput of transaction processing through better fitting of the single-threaded model onto modern multi-core hardware. In particular, we propose LADS, a dynamic single-threaded OLTP system, which separates the concurrency control from the actual transaction execution. LADS first resolves a batch of transactions into a set of *record actions*. A record action is a consecutive sequence of (atomic) operations on the same tuple within a transaction. LADS makes use of dependency graphs [45] to capture dependency relations among record actions within a batch of transactions. It then decomposes the dependency graphs into sub-graphs such that the sub-graphs are about the same size, and the number of edges across these sub-graphs is minimized. The sub-graphs are subsequently distributed to the available worker threads for execution in such a manner that the actions to be performed on the same record are assigned to the same worker. Within one sub-graph, record actions are executed in transaction order to resolve the dependencies between transactions. For record actions with dependencies (edges) between sub-graphs of different worker threads, LADS also ensures that they are executed in order so that dependencies among sub-graphs of the same transaction can be quickly resolved. Hence, transaction aborts due to out-of-order execution of conflicting record actions are completely eliminated. Further, instead of partitioning the database statically, LADS dynamically partitions the workload according to its actual distribution to achieve a higher CPU utilization. As shown in Figure 3, LADS first tracks the workload distribution during the transaction dependency resolution, and then partitions the workload dynamically to balance the workload among the available workers. In this example, LADS partitions the workload into four parts, {A},{B,I},{C} and {J,E}, and assigns them to the available workers. Meanwhile, LADS is also locality aware as it takes into account the memory hierarchy of the multi-core platform, and supports optimizations to avoid cache coherence overhead. More importantly, LADS reduces the use of centralized components with no locks/latches required during its processing, which further improves the computation efficiency.

The design of LADS is based on the principle of fast batch processing in a multi-core in-memory system which enables a latency/throughput trade-off. As with any batch processing system, latency is a valid concern. However, in practice, this is not as big an issue as it appears to be. First, in real applications, requests at the client side are always sent to the server in batches in order to reduce the network overhead [26]. Besides, transactions are processed and committed in a batched manner in systems using group commit protocols [10] to reduce disk I/O cost, and in-memory systems need to write transaction logs to disk to ensure reliability [41]. Second, data accesses in multi-core in-memory systems are very fast compared with that of traditional disk-based systems. This should reduce the overall latency. Finally, the latency due to batch processing can be optimized by tuning the batch size. In short, the latency of a well designed multi-core transaction processing can be bounded to an acceptable bound. Evaluation of our implementation of LADS shows that it achieves significantly higher throughput, and scales well.

In summary, LADS offers the following innovations:

- The representation and resolution of dependencies between atomic record actions of a batch of transactions in dependency graphs.
- The separation of contention resolution and transaction execution into two stages that allows for the decentralized and dynamic partitioning of the dependency graphs into work balanced subgraphs which can be scheduled onto independent threads such that no synchronization is required.
- Techniques to deal with issues including dependency across dependency sub-graphs, range queries, and logging within the same two-phase execution framework.

These novel ideas allow LADS to perform up to $20\times$ faster than state-of-the-art systems. More importantly, LADS can better exploit the advances in hardware, and achieve a scaling that is more in line with Moore's Law.

The remainder of the paper is organized as follows. In Section 2, we review the related work. A novel dynamic single-threaded transaction processing model is presented in Section 3. We describe the implementation details of LADS in Section 4. A comprehensive evaluation is presented in Section 5. Finally, the paper is concluded in Section 6.

2. RELATED WORK

Many research efforts have been devoted to improve the performance of multi-core in-memory systems. We provide a brief survey on works that are relevant to our proposal.

2.1 Systems with Traditional Concurrency Control Protocols

Systems using lock-based protocols typically require a lock manager, in which lock tables are maintained to grant and release locks. The data structure in the lock manger is typically very large and complex, which incurs both storage and processing overheads. Lightweight Intent Lock (LIL) [24] maintains a set of lightweight counters in a global lock table instead of lock queues for intent locks. Although LIL simplifies the data structure of intent locks, transactions that cannot obtain all the locks have to be blocked until a release message from the other transactions is received. In order to reduce the overhead of a global lock manager, associating the lock states with each data record has been proposed [15]. However, this technique requires each record to maintain a lock queue, and hence increases the burden of record management. By compressing all the lock states at one record into a pair of integers, Very Lightweight Locking [35] simplifies the data structure to some extent. However, it achieves this by dividing the database into disjoint partitions, which affects the performance on workloads that cannot be well partitioned.

Optimistic Concurrency Control (OCC) [25], which is a variant of timestamp-based protocol, is widely adopted. However, its

performance is sensitive to contention intensity [4, 17]. Multi-Version Concurrency Control (MVCC) [7, 30] is another protocol with which read operations do not block write operations. HyPer extends MVCC to enforce serializability [31], and BOHM [13] optimizes MVCC by avoiding all shared memory writes for read tracking.

Hekaton [11] employs lock-free data structures and OCC-based MVCC protocol to avoid applying writes until the transaction is about to commit. However, the centralized timestamp allocation may remain the bottleneck, and the read overhead may increase since each read needs to update the dependency set of the other transactions. Silo [42] is an in-memory OLTP database prototype optimized for multi-core systems. Silo supports a variant of OCC method which employs batch timestamp allocation to alleviate the performance loss. FOEDUS [23] is proposed to scale up the database and allow transactions to manipulate both DRAM and NVRAM efficiently. However, both Silo and FOEDUS do not perform nor scale well for high contention workloads.

Research efforts have also been devoted to decomposing transactions into smaller pieces to increase execution parallelism [6, 14, 38]. Homeostasis protocol [36] extracts consistency requirements from transactions via program analysis, with which transactions can be correctly executed while minimizing communications. Application level analysis could provide conditions for coordination-free execution [5]. However, compared with LADS, these works only support transaction-level conflict checking and resolution that do not expose all the available parallelism.

2.2 Systems with Single-Threaded Model

H-Store [20] is a partitioned database system, where each partition is treated as an independent database. A thread in each partition is responsible for processing the transactions, and there is no need for concurrency control within a partition. The single-threaded model provides high efficiency for single-partition transactions. However, a transaction that spans multiple partitions needs to obtain partition-level locks before processing operations, which inevitably restricts the single-threaded model's scalability and processing efficiency. HyPer [21] is another partitioned database system that also makes use of the single-threaded model.

DORA [32] is a logically partitioned database for multi-core systems. Unlike H-Store and HyPer, DORA decomposes a transaction into transaction pieces and assigns them to the threads based on where the data of interest reside. It employs a private locking mechanism to resolve conflicts during processing. Although it eliminates the centralized lock manager used in the traditional two-phase locking protocol, it may still incur some amount of overhead on the local lock management. Further, like H-Store, transactions spanning multiple partitions are potentially a performance bottleneck. PLP [33] extends DORA by physically partitioning the database based on a multi-rooted B^+-tree structure, where each partition corresponds to a subtree and is managed by one thread. It supports flexible repartitioning to reduce the costly effect of cross-partition transactions. However, it requires a centralized routing table, and frequent repartitioning also affects parallelism among transactions. In contrast, LADS employs the single-threaded model in both dependency graph construction and actual transaction execution. Execution of transactions according to the dependency graph reduces the coordination cost among the threads to an extent, which improves its efficiency and scalability.

Load balancing is typically required to redistribute the workload among partitions in single-threaded systems, and various techniques have been proposed in recent years. Horticulture [34] provides an automatic partitioning algorithm and generates partitioning strategy for a sample workload. E-Store [40] provides a two-tiered approach to handle hot and cold chunks respectively. It first distributes hot tuples throughout the cluster and then allocates cold tuples to fill the remaining space. Squall [12] introduces fine-grained live repartitioning by interleaving data migration with execution. All these techniques perform repartitioning or reconfiguration by tracking transactions' historical access patterns. However, real-time load balancing based on the analysis of historical information is difficult to materialize. They are therefore not effective for randomized workloads since frequent repartitioning incurs high overhead. In comparison, LADS processes transactions in a batched manner and supports a real-time load balancing by evenly distributing a batch of workload to the available workers, without the need to track historical access patterns.

3. TRANSACTION PROCESSING IN LADS

Traditionally, any contention that exists among transactions is resolved by means of locks or timestamps. This can lead to thread blocking or waste of computation due to transaction aborts. In LADS, each worker thread obtains a batch of transactions from the transaction queue, and resolves their dependencies at the granularity of record action (defined below) before processing them. The batch size is determined based on the number of transactions in the transaction queue and a pre-defined maximum batch size. In this manner, LADS separates contention resolution from the actual transaction execution.

LADS first constructs dependency graphs in parallel according to the transactions' timestamps and logics. Specifically, each worker constructs one dependency graph for a set of transactions and then decomposes the constructed graph into sub-graphs. During the actual transaction execution, each worker thread executes according to one sub-graph. With dependency graphs, workers can process the transactions with the guarantee of conflict serializability. As mentioned earlier, LADS exploits single-threaded model to the fullest, and consequently, single-threaded model is adopted in the dependency graph construction, dependency graph partitioning and actual transaction execution.

3.1 Dependency Graph Construction

Before constructing a dependency graph for a set of transactions, LADS has to resolve dependency relations within one single transaction. It parses the statements of a transaction and decomposes it into a set of *record actions*, where each record action only accesses a single record in the database. The formal definition of the record action is given below:

DEFINITION 1. **Record Action** *A transaction is a unit of operations performed on the database. Operations consecutively conducted on the same record within one transaction (with no intervening operation on another distinct record) constitute a single record action* α.

Given a record action α, $r(\alpha)$ denotes the record on which it acts upon, and $c(\alpha)$, $f(\alpha)$, $\rho(\alpha)$ respectively denote the transaction's timestamp, function set and parameter set.

Figure 4(a) shows an example, where the database has only one table. There are three types of transactions, namely transfer, save, and withdraw. In this example, transaction t_8 attempts to transfer \$7 from account 1003 to account 1005. LADS firstly decomposes t_8 into two record actions, $(8 : minus, 1003, 7)$ and $(8 : add, 1005, 7)$. Record action $(8 : minus, 1003, 7)$ operates on account 1003 by subtracting \$7 from its balance. Record action $(8 : add, 1005, 7)$ attempts to add \$7 to account 1005.

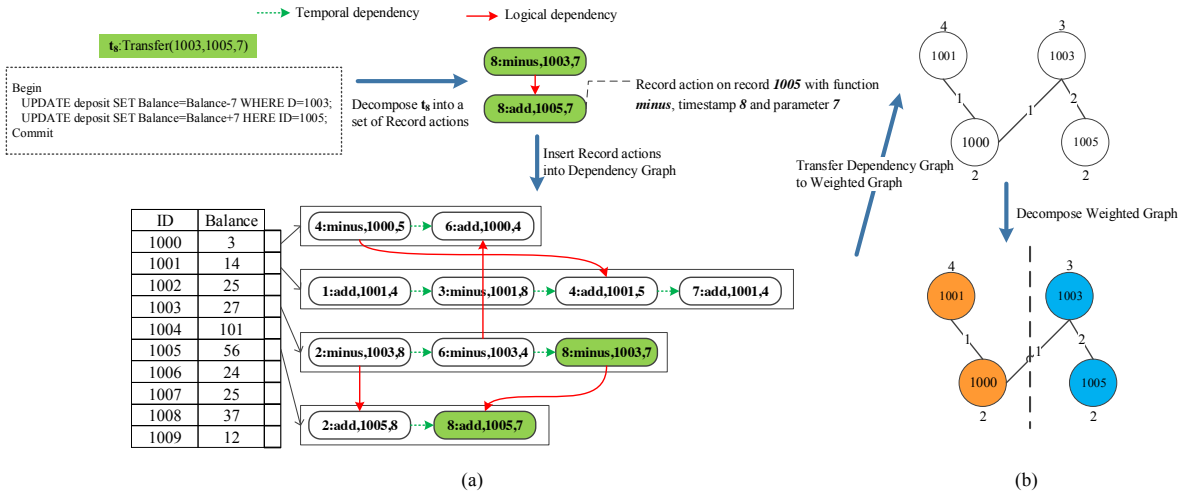


Figure 4: Example of dependency graph construction and partitioning with three types of transactions: Save, Withdraw, Transfer.

Next, we define two types of dependency relations on record actions: *logical dependency* $\succ_{\text{logic}}$ and *temporal dependency* $\succ_{\text{temporal}}$.

DEFINITION 2. **Logical Dependency** *Record action α_i logically depends on α_j, denoted as*

$$\alpha_i \succ_{logic} \alpha_j$$

if and only if both α_i and α_j belong to the same transaction, i.e., $c(\alpha_i) = c(\alpha_j)$, and α_i must be executed after *α_j.*

From the above definition, we can see that $\succ_{\text{logic}}$ determines the logical execution order of record actions within one transaction. In the previous example, record action $(8 : add, 1005, 7)$ is logically dependent on $(8 : minus, 1003, 7)$, since transaction t_8 has to ensure that the balance in account 1003 is sufficient. Apart from the logical dependency relation, we also need to resolve the contention of record actions from different transactions, which is defined by the temporal dependency relation $\succ_{\text{temporal}}$.

DEFINITION 3. **Temporal Dependency** *A temporal dependency exists between record actions α_i and α_j, denoted as*

$$\alpha_i \succ_{temporal} \alpha_j$$

if and only if α_i and α_j belong to different transactions with $c(\alpha_i) > c(\alpha_j)$, and $r(\alpha_i) = r(\alpha_j)$

As shown in Figure 4(a), record actions $(8 : minus, 1003, 7)$ and $(8 : add, 1005, 7)$ temporally depend on $(6 : minus, 1003, 4)$ and $(2 : add, 1005, 8)$, respectively.

Now, we define the *dependency graph*, where a vertex represents a record action and an edge represents a dependency relation between two record actions. The formal definition is as follows:

DEFINITION 4. **Dependency Graph** *Given a set of transactions $\mathcal{T} = \{t_1, t_2, \cdots, t_n\}$, and the associated sets of record actions $\varphi_{t_1}, \varphi_{t_2}, \cdots, \varphi_{t_n}$, the dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a directed graph and consists of*

- $\mathcal{V} = \varphi_{t_1} \cup \varphi_{t_2} \cup \cdots \cup \varphi_{t_n}$, *and*
- $\mathcal{E} = \{(\alpha_i, \alpha_j) | (\alpha_j \succ_{logical} \alpha_i \text{ or } \alpha_j \succ_{temporal} \alpha_i), \alpha_i \in \varphi_{t_i}, \alpha_j \in \varphi_{t_j}\}$

In particular, it is not efficient to analyze every record action α on record $r(\alpha)$ as we add α into dependency graph $\mathcal{G}$. Furthermore, explicitly recording all the temporal dependency edges between a pair of record actions may result in many edges. Therefore, during dependency graph construction, we maintain the *latest record action* $L(k)$ for each record k that has been accessed in $\mathcal{G}$.

By decomposing a transaction into record actions, logical dependency relations are naturally resolved. When record action α is inserted into the dependency graph, an edge from $L(r(\alpha))$ to α is created, since $\alpha \succ_{\text{temporal}} L(r(\alpha))$. We maintain an action queue ϕ_k for each record k. Given a record action α, it should be appended to the end of $\phi_{r(\alpha)}$. The record actions in a queue should satisfy either $\succ_{\text{temporal}}$ or $\succ_{\text{logical}}$. Record actions in different queues may only have $\succ_{\text{logical}}$. Note that $\succ_{\text{temporal}}$ never exists between record actions in different queues according to its definition. The dependency graph construction algorithm for a set of transactions $\mathcal{T}$ is given in Algorithm 1.

Algorithm 1: Dependency graph construction

Input: transaction set $\mathcal{T}$
Output: dependency graph $\mathcal{G}$ for $\mathcal{T}$

```
Initialize an empty graph G
foreach t ∈ T do
    φ_t ← {α_1, α_2, ..., α_m} decomposed from t
    for i ← 1 to m do                    /* for temporal dependency */
        G.AddVertex(α_i)
        if L(r(α_i)) exists then G.AddEdge(L(r(α_i)), α_i)
        L(r(α_i)) ← α_i
    for i ← 1 to m − 1 do                /* for logical dependency */
        for j ← i + 1 to m do
            if α_i ≻_logic α_j then G.AddEdge(α_j, α_i)
return G
```

During the dependency graph construction, each worker thread maintains a constructor to resolve dependency relations among a batch of transactions, which is essentially a consecutive number of transactions from its transaction queue, and build the dependency graph accordingly. When the system is saturated, the batch size is equal to the pre-defined maximum batch size. Otherwise, after finishing one round of batch processing, the worker will check

the transaction queue. If the number of transactions waiting in the transaction queue is less than the pre-defined maximum batch size, all of them will be processed as a batch. The batch size in LADS changes dynamically to adapt to workloads of different request rates.

Each worker is responsible for the construction of one dependency graph. To better exploit the parallelism in the CPU, several graphs can be constructed in parallel by different worker threads. Each thread can construct a dependency graph asynchronously since dependency graph construction for a batch of transactions is a completely independent process.

3.2 Dependency Graph Partitioning

Each worker thread then decomposes each constructed dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ into sub-graphs for distribution over all the available worker threads. For optimal performance, it is necessary to ensure that, as far as possible, each worker thread gets the same amount of work. Furthermore, execution of a record action in one worker thread may result in sending of information to its dependent record actions in other worker threads.

The objective of the decomposition is to get a partition Ψ of $\mathcal{V}$ with n elements. That is $\Psi = \{\mathcal{V}_1, \mathcal{V}_2, \cdots, \mathcal{V}_n\}$, where

$$\mathcal{V}_1 \cup \cdots \cup \mathcal{V}_n = \mathcal{V}$$
$$\mathcal{V}_i \cap \mathcal{V}_j = \emptyset, \forall i \neq j$$

such that the partitions are of about the same size with minimal number of cross-partition edges.

We note that the graph decomposition problem is isomorphic to the classical uniform graph partitioning problem [22]. To solve the problem, we only need to minimize the size of edge cut among the partitions.

$$\sum_{i \neq j} | \otimes_{ij} |$$

where $\otimes_{ij} = \{(\alpha_p, \alpha_q) | \alpha_p \in \mathcal{V}_i \; and \; \alpha_q \in \mathcal{V}_j\}$ with constraint that

$$\forall i, |\mathcal{V}_i| \leq (1+\epsilon)\frac{|\mathcal{V}|}{n}$$

As all the record actions in ϕ_k are to be performed on the same record k, it is better to group them to the same partition. We define a new weighted graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$.

$$\mathbb{V} = \{\phi_i | \phi_i \neq \emptyset\}$$

$$\mathbb{E} = \{(\phi_i, \phi_j) | \alpha_p \in \phi_i, \alpha_q \in \phi_j \, (\alpha_p, \alpha_q) \in \mathcal{E} \; or \; (\alpha_q, \alpha_p) \in \mathcal{E}\}$$

A vertex in the weighted graph $\mathbb{G}$ is a record action queue, whose weight equals to the number of record actions in the queue. The weight of one edge equals to the number of logical dependency relations between the two queues. We use function w to calculate the weight. Compared with the initial dependency graph $\mathcal{G}$, the weighed graphs $\mathbb{G}$ has much fewer vertices and edges. This simplifies the complexity of partitioning. In Figure 4(b), the weighted graph only contains four vertices and three edges. The graph decomposition problem is to minimize the following function:

$$\sum_{i \neq j} w(\otimes_{ij})$$

where $\otimes_{ij} = \{(p, q) | p \in \mathbb{V}_i, q \in \mathbb{V}_j\}$ with the constraint that

$$\forall i, w(\mathbb{V}_i) \leq (1+\epsilon)\frac{w(\mathbb{V})}{n}$$

We adopt a greedy algorithm to accelerate the dependency graph partitioning. LADS first evenly partitions the weighted graph according to key range and extracts edge cuts based on the partitioning result. Since LADS is designed for a multi-core environment such as NUMA [28], we also take data locality into consideration for performance purposes. Correspondingly, data within the same NUMA node are given a higher probability for them to be in the same partition. It then invokes a repartitioning process to minimize the weight of the edge cuts. The basic idea is summarized in Algorithm 2. To illustrate the idea, consider the earlier example and Figure 4(b), where the weighted graph is decomposed into two partitions, $\{1000, 1001\}$ and $\{1003, 1005\}$. During transaction execution, worker 1 is responsible for the record actions on accounts 1000 and 1001, and worker 2 is responsible for the record actions on accounts 1003 and 1005.

Algorithm 2: Graph partitioning

Input: weighted graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$
Output: a set of sub-graphs $\{\mathbb{G}_1, \mathbb{G}_2, \ldots, \mathbb{G}_n\}$

Partition $\mathbb{G}$ into a set of sub-graphs $\{\mathbb{G}_1, \mathbb{G}_2, \ldots, \mathbb{G}_n\}$ with about the same weight according to the key range

$\mathbb{E}_{cross} = \bigcup_{p \neq q} \otimes_{pq}$ /* edge cut among the partitions */
foreach $e_{ij} \in \mathbb{E}_{cross}$ **do**
  benefit[e_{ij}] = $w(\mathbb{E}_{cross})$ - $w(\mathbb{E}_{cross}^{i \to j})$
end
max_itrs = maximum iteration number
for $l \leftarrow 1$ **to** max_itrs **do**
  `Sort` (benefit)
  e_{ij} = benefit.`EdgeWithMaxBenefit` ()
  $\mathbb{G}_p$ = `GetSubgraph` (v_i)
  $\mathbb{G}_q$ = `GetSubgraph` (v_j)
  if $\forall k, w(V_k) \leq (1+\epsilon)\frac{w(V)}{n}$ & benefit[e_{ij}] >0 **then**
    $\mathbb{G}_p$.`RemoveVertex` (v_i)
    $\mathbb{G}_q$.`AddVertex` (v_i)
    `Update` (benefit)
  else
    Break
  end
end
return $\{\mathbb{G}_1, \mathbb{G}_2, \ldots, \mathbb{G}_n\}$

3.3 Transaction Execution

3.3.1 Transaction Execution Based on Dependency Graphs

A global synchronization operation is enforced to ensure that all the worker threads will complete the graph construction and partitioning phase before entering the transaction execution phase simultaneously. Given a set of partitions $\Psi = \{\mathcal{V}_1, \mathcal{V}_2, \cdots, \mathcal{V}_n\}$, LADS distributes it to the available workers to perform the actual transaction execution in parallel. Each worker i executes the record actions of $\mathcal{V}_i$ in a greedy manner as summarized in Algorithm 3. Initially, the worker selects record actions with no in-edges in the sub-graph and inserts them into an executable set that it maintains. Any record action in the executable set will not conflict with any other remaining record actions and can be executed correctly. A worker thread then iteratively selects record actions from the executable set to execute. After one record action is executed, it will be removed from the sub-graph together with its out-edges from the original dependency graph $\mathcal{G}$. Those record actions without in-edges in the residual graph should then be inserted into the executable set. Figure 5 illustrates an example of

the above procedure. Initially, there is one executable record action $(2 : minus, 1003, 8)$ in sub-graph 2. When it is executed, record actions $(6 : minus, 1003, 4)$ and $(2 : add, 1005, 8)$ are inserted into the executable set. The process is repeated until all the record actions in $\mathcal{V}_i$ are fully executed.

Algorithm 3: Dependency graph execution

Input: dependency graph $\mathcal{G}_i$=($\mathcal{V}_i$,$\mathcal{E}_i$)

$\mathcal{R} = \emptyset$ /* initialize executable set */

while $\mathcal{R} \neq \emptyset$ & $\mathcal{G}_i$.Size ()>0 **do**

  $\mathcal{R}$.AddVertex ($\mathcal{G}_i$.RootVertex ())

  $\alpha_q = \mathcal{R}$.Pop ()

  α_q.Execute ()

  $\mathcal{G}_i$.RemoveOutEdge (α_q)

  $\mathcal{G}_i$.RemoveVertex (α_q)

LADS evenly partitions the workload for the transaction execution phase, thus addressing the problem of workload skew. In addition, LADS reorders operations before transaction execution so that operations on the same record can be executed within a short time interval, thereby improving cache hit rate.

As there are more than one dependency graph available during the execution, it is possible that there exist conflicts among dependency graphs. We resolve such conflicts by processing the conflicting dependency graphs sequentially. After all the record actions of a graph are fully processed, the corresponding transactions commit in a group manner.

3.3.2 Handling Transaction Aborts

Since LADS resolves all the conflicts among transactions before transaction execution, transaction abort caused by the conflicts is eliminated. However, transaction aborts due to field constraints of the database are still possible. For instance, the remaining balance in Figure 4(a) cannot be negative. This can lead to cascading aborts even in LADS. To avoid the overhead of cascading aborts, we add a condition-variable-check function as the first record action in each transaction. Other record actions in the transaction logically depend on this first record action. If the first record action aborts, it sends disable messages to the rest of the record actions in the same transaction. As a consequence, no cascading abort is possible during the transaction execution. For example, in Figure 5, record action $(4 : minus, 1000, 5)$ will abort, since the balance left in account 1000 is not sufficient. Before aborting, it will send a disable message to record action $(4 : add, 1001, 5)$, which will not be executed later on.

3.3.3 Handling Range Queries

LADS also supports queries that access a range of keys in the database. The keys accessed by such a range query may change during the actual transaction execution, and violates serializability. LADS deals with this challenge by organizing the record action of range query at a coarser granularity. In this case, LADS treats the entire transaction as a single record action that needs to update the entire column, table, or partition, as the case may require. So all the record actions of the subsequent transactions to be performed on that column (table or partition) must depend on this composite record action. In the worst case where all the transactions perform range queries, LADS would degrade to a partitioned database system. As range query is usually rare in OLTP applications, this should not be a problem in practice.

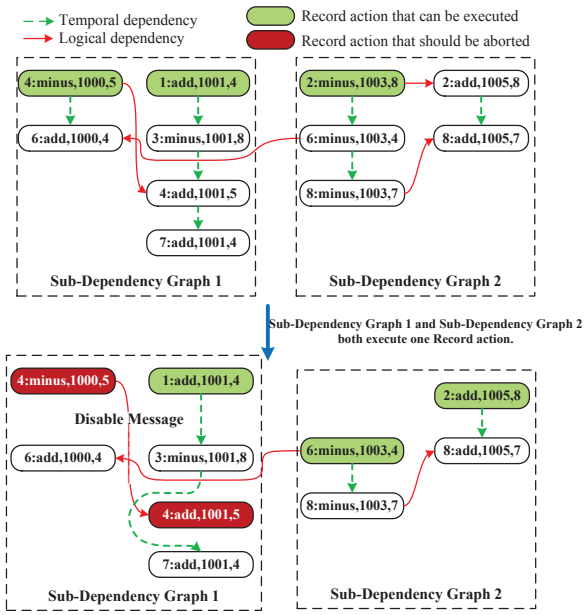


Figure 5: Dependency graph execution.

3.4 Correctness

We now prove that the dependency graph $\mathcal{G}$ constructed by LADS guarantees conflict serializability. The dependency graph $\mathcal{G}$ works as a schedule η of transaction set $\mathcal{T}$. We can prove that η is conflict-serializable. According to *conflict serializability theorem* [44], we only need to show that the *conflict graph* $G(\eta)$ constructed based on η is acyclic.

DEFINITION 5. **Conflict Graph** *Given the dependency graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, let η be its schedule. The conflict graph $G(\eta) = (V, E)$ of η is defined by*

$$V = \mathcal{T}$$

$$(t_i, t_j) \in E \iff (i \neq j) \; and \; \exists \alpha_i, \alpha_j \in \mathcal{V}, \alpha_j \succ \alpha_i$$

According to our previous definitions of dependency relations, the conflict relation in the conflict graph $G(\eta)$ should be either $\succ_{\text{temporal}}$ or $\succ_{\text{logic}}$.

First, let us consider $\succ_{\text{temporal}}$ in $G(\eta)$. If there is a directed edge from α_i to α_j, then $i < j$. Now if $G(\eta)$ is *cyclic*, then we can always find a cycle with edges, $(\alpha_{i_0}, \alpha_{i_1})$, $(\alpha_{i_1}, \alpha_{i_2})$, $\cdots$, $(\alpha_{i_{v-1}}, \alpha_{i_v})$, $(\alpha_{i_v}, \alpha_{i_0})$, where $i_0 < i_1 < \cdots < i_{v-1} < i_v$ and $i_v < i_0$. Obviously, this violates the initial condition, namely $i < j$. In other words, if we only consider $\succ_{\text{temporal}}$, $G(\eta)$ must be acyclic.

Next, we consider $\succ_{\text{logic}}$. Based on its definition, $\succ_{\text{logic}}$ will not lead to an edge in $G(\eta)$ because $\succ_{\text{logic}}$ only exists between two record actions within the same transaction. So $G(\eta)$ is still acyclic. We can conclude that $G(\eta)$ must be acyclic and thus η is a conflict-serializable schedule.

4. IMPLEMENTATION

In this section, we present the architecture of LADS. As a relational database engine, LADS organizes data into tables. Each row with a unique key is called a record, which is stored in a segment of allocated memory.

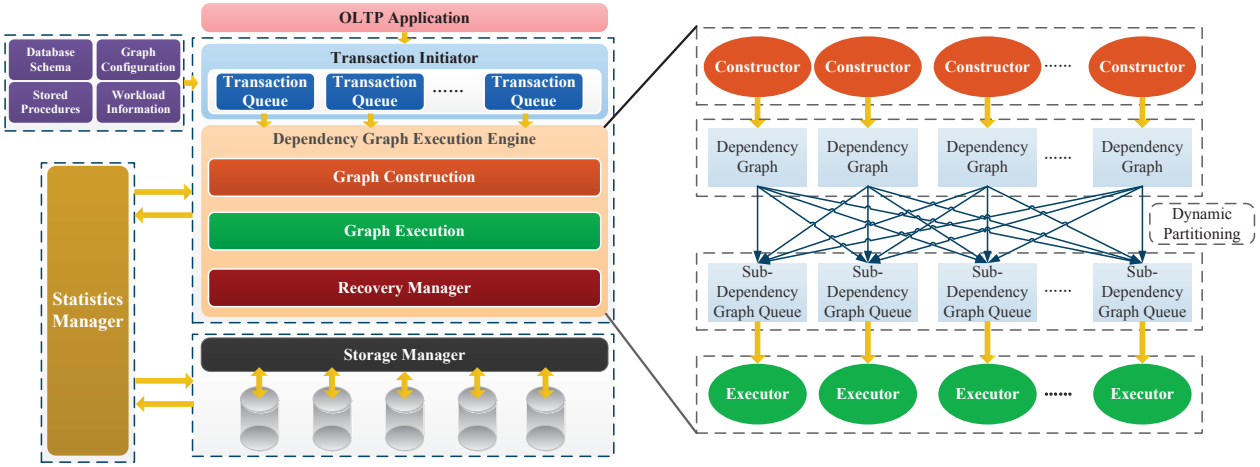


Figure 6: LADS system architecture.

4.1 System Architecture

The architecture of LADS consists of four components as shown in Figure 6.

The **Transaction Initiator** maintains a set of transaction request queues that are handled by different worker threads. Typically, arriving transactions are not processed by the system immediately. Rather, they will wait in a transaction queue. In some applications, transaction requests may have different priorities [8, 29]. LADS can adjust the priority of each queue (worker thread) accordingly.

The **Dependency Graph Execution Engine** is responsible for constructing dependency graphs and conducting the transaction execution in batches. To improve efficiency, worker threads construct multiple dependency graphs in parallel. LADS processes the graphs based on their priorities. The number of parallel threads is equal to the number of transaction queues, and the threads alternate between graph construction and graph execution phase, much like a simple Bulk Synchronous Parallel execution model [43].

The **Storage Manager** is designed to manage the data in the database. It interacts with the execution engine to retrieve, insert, update and delete data. Both the B^+-tree index and hash index are supported. LADS guarantees the serializability and conflict-free read/write operations.

The **Statistics Manager** collects runtime statistical information such as real-time throughput and latency. It also interacts with the other components to adjust the system configuration dynamically. For example, since LADS processes transactions in batches, the batch size affects both the throughput and latency. A larger batch size should result in a higher throughput, while a smaller batch size provides a shorter response time. The maximum batch size can be adjusted accordingly based on the statistics and the requirements.

4.2 Implementation Details

4.2.1 *Timestamp Assignment*

When a transaction arrives, LADS assigns a timestamp to the transaction and inserts it into the transaction queue. Compared with other timestamp-based protocols, the timestamp assignment in LADS is thread-private and does not rely on any centralized resources. LADS only makes use of timestamps to resolve temporal dependency for transactions in the same transaction queue. However, it is possible that conflicts also exist between transactions that are in different transaction queues. As shown in Figure 6, LADS constructs several graphs in parallel. It will then execute them one after another. By doing this, LADS resolves the conflicts between dependency graphs without using the timestamps.

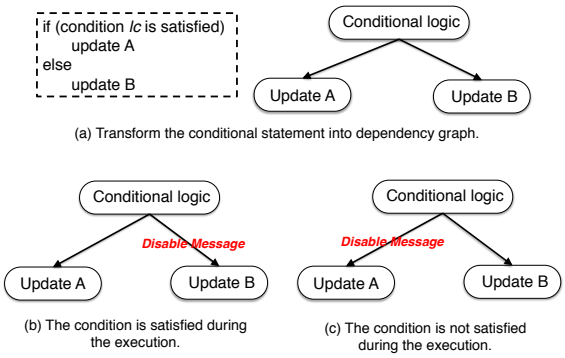


Figure 7: A conditional statement example.

4.2.2 *Data Layout*

LADS maintains a row-based database, and each record contains the following information:

- **The record data.** To reduce cache coherence overhead, LADS employs cache line alignment when it stores the record data.
- **The pointer to a record action queue.** LADS adds a field in each record to store the pointer that refers to its record action queue.

As discussed in Section 3.3, after all the record actions of one graph have been processed, the transactions will commit at the same time. The record data is only modified by the last update. In the example shown in Figure 5, four record actions operate on account 1001. However, only the update performed by record action $(7 : add, 1001, 4)$ modifies the record in place.

4.2.3 *Transaction Decomposition*

To build the dependency graph, LADS first parses the statements of each transaction, and then transforms them into a set of record actions. Although OLTP transactions are typically short-lived, they may contain some complex statements such as conditional statements.

LADS handles transactions with conditional logic by adding extra record actions. In Figure 7, the transaction updates record A if it satisfies the condition lc. Otherwise, it updates record B. Figure 7(a) shows how this conditional logic is represented in the dependency graph. During actual transaction execution, the record action that handles the conditional logic should send messages to disable record actions in the false branch, as shown in Figure 7(b) and Figure 7(c).

4.2.4 Transaction Logs and Checkpointing

Being an in-memory data engine, LADS stores all the data in main memory. Yet, for reliability, LADS flushes transaction logs into disks for recovery purposes. LADS makes use of command logging [27]. Each record action in the dependency graph is associated with a log record consisting of the record key, function set, parameters, and dependency information. No real data is recorded in the log files, and hence the logging overhead and size of the logs are reduced. During recovery, we only need to replay those log records to reconstruct the dependency graphs and then execute the reconstructed graph. Instead of generating log records for a single transaction, LADS can construct log records for transactions in a batched manner. Writing all those log records as a batch fully utilizes the disk I/O bandwidth, thereby improving the system's overall performance. Furthermore, each worker maintains a separate log buffer to eliminate contention during the processing.

In order to recover the database within a bounded time, LADS also performs periodic checkpointing. It maintains several checkpointing threads. The entire memory is divided up into sections and each checkpointing thread is responsible for one such section. Even as the checkpointing threads are working, transactions continue to execute. However, those commits are not reflected in the checkpointing. This means our checkpointing is not a consistent snapshot of the database, and it needs to combine with the logging. To recover from a failure, LADS has to reload the latest checkpoint, and replay the transaction log records from that time point onwards. It then reprocesses the committed transactions.

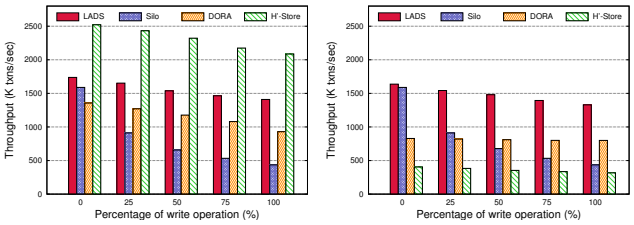
5. EXPERIMENTS

In this section, we shall evaluate LADS against H-Store [20], a partition-based database that adopts the single-threaded model; DORA [32], a data-oriented transaction processing system; and Silo [42], a multi-core in-memory OLTP engine with optimistic concurrency control protocol. The original H-Store is mainly implemented in Java. For a fairer comparison, we reimplemented H-Store in C++, and also reduced some of its functionalities, such as network communication. For clarity, we call our implementation H'-Store. Similarly, since DORA is originally implemented on a disk-based storage manager called Shore-MT [19], we extended DORA so that it makes use of the same storage manager as LADS to maintain the entire database in memory.

5.1 Experimental Setup

All the experiments are conducted on a multi-core server equipped with four Intel Xeon 2.2 GHz processors that have six cores each. This gives us a total of 24 cores. The server has 64 GB of DRAM. The four processors are connected to form a NUMA architecture. Each core has a private 32 KB L1 cache, a 256 KB L2 cache and supports two hyper-threads. Each six-core processor has a 12 MB L3 cache shared by its cores.

Two popular OLTP benchmarks, namely YCSB [9] and TPC-C [2], are used in the evaluations. First, the YCSB benchmark contains various combinations of read/write operations. Originally, each YCSB transaction only reads/writes a single data record, which usually generates few conflicts during execution. To simulate the situation with high contention of data access in the evaluations, we extend the YCSB benchmark such that each transaction reads or writes 20 records. Moreover, we shrank the record size from the standard 1000 bytes to 100 bytes. This makes a fairer comparison with Silo which requires allocating memory space for every version of a recored, and shrinking the record size mitigates the issue of Silo's memory allocation being its performance bottleneck [42]. Second, the TPC-C benchmark simulates a complete order-entry environment whose transaction scenario is more complex than that of YCSB. Five types of transactions (New-Order, Payment, Delivery, Order-Status and Stock-Level) are generated in the TPC-C benchmark.



(a) Percentage of corss-partition transaction is 0%

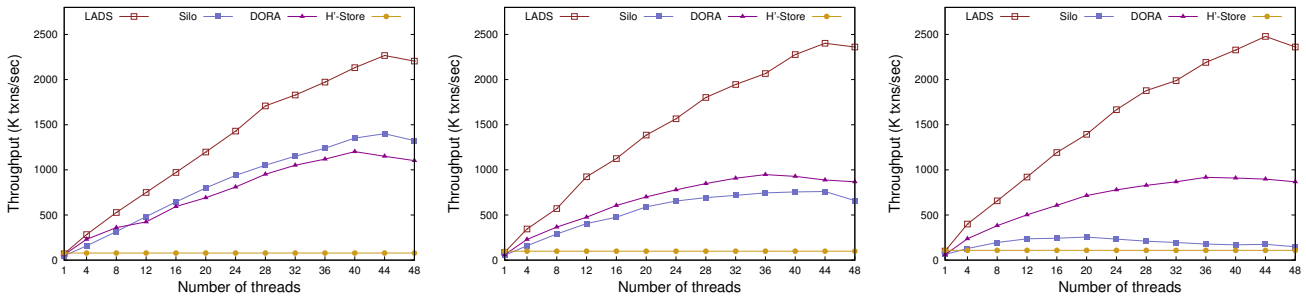
(b) Percentage of corss-partition transaction is 10%

Figure 8: Effects of write operations on the YCSB benchmark. The number of threads is fixed as 24 and Zipfian θ=0.8.

Table 1: Parameter values for evaluations

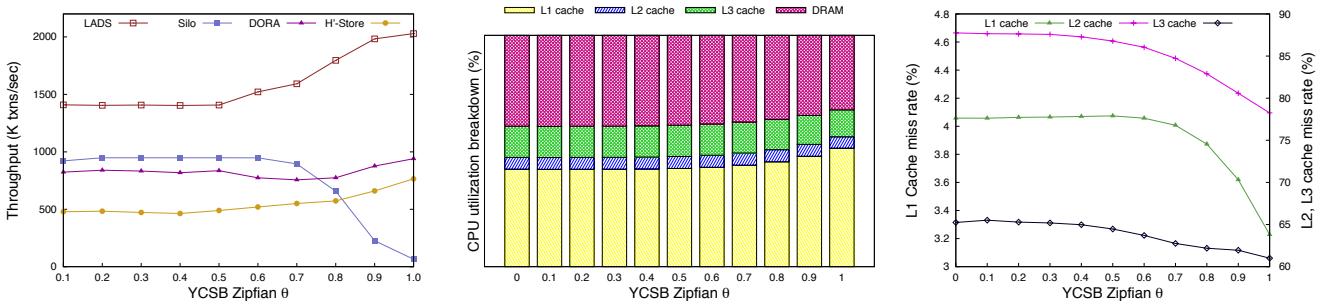
Parameter	Values
Number of threads	4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48
YCSB Zipfian θ	0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
Percentage of YCSB writes	0, 0.25, 0.5, 0.75, 1
Number of TPC-C warehouse	4, 8, 24
Percentage of cross-partition transaction	0, 0.1, 0.2, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9, 1

In a multi-core environment, contention among threads seriously affects system performance. There are three factors that typically dominate the intensity of contention. The first is the percentage of write operations in the workload. While reads on the same record can always be operated in parallel, writes on the same record must be performed sequentially. Hence, more write operations usually lead to higher contention. The second is the skewness of data access. In practice, OLTP applications tend to access certain data more frequently. For example, in an online shopping scenario, popular items are accessed more frequently than others. YCSB simulates the skewness of data access through a Zipfian distribution [16] in which the parameter θ determines the level of skewness. For a given number of working threads, a larger θ results in more contentions. In the evaluations, by setting θ to 0.5, 0.8 and 0.9, 10% of the records in the database are accessed by about 35%, 60% and 75% of all transactions, respectively. The third factor is the number of concurrent threads. A larger amount of parallel transactions usually leads to more contention. Apart from the three factors above, for partitioned database systems with the single-threaded model, the percentage of cross-partition transactions is another important factor. A cross-partition transaction may block the threads that work on the partitions where the transaction needs to access, and thus affect system performance. In the following experiments, we study the performance of LADS with respect to all these factors using the two benchmarks. The parameters are listed in Table 1, where each underlined value indicates the default setting.



(a) Workloads of low contention (θ=0.5) (b) Workloads of medium contention (θ=0.8) (c) Workloads of high contention (θ=0.9)

Figure 9: Effects of skewed workload on the YCSB benchmark.



(a) Effects of data access skew (b) Cache efficiency of LADS (c) Cache miss rate variation in LADS

Figure 10: Effects of data access skew and cache efficiency.

5.2 Impact of Write Operations

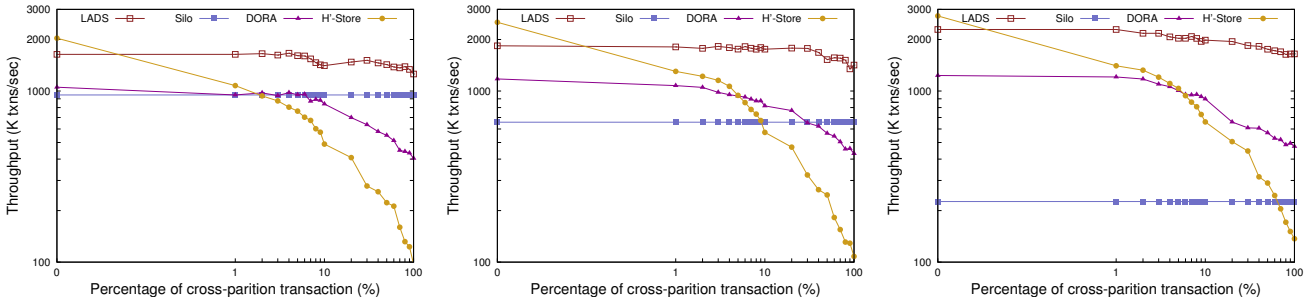
We first evaluate how the performance of the four systems is affected by write operations by running the YCSB benchmark. Figure 8(a) shows the performance variation when the Zipfian θ is set to 0.8 and no cross-partition transactions are involved. All the four systems perform better on workload with more read operations, since write operations are usually more expensive. The performance of Silo drops significantly as the percentage of write operations increases in the workload. This is because, with increasing amount of write conflicts, Silo has to spend more time in resolving contention upon transaction commits. Furthermore, Figure 8(b) shows the results when cross-partition transactions are involved. Unlike LADS, the performance of H'-Store and DORA drops significantly, and it becomes resilient to the variation of percentage of write operations. This is expected since thread blocking caused by cross-partition transaction is expensive and it has more impact than the effects of write operations.

5.3 Impact of Workload Skewness

Next, we study the impact on system robustness with skewed workloads. The skewness may exist among partitions and/or within a single partition.

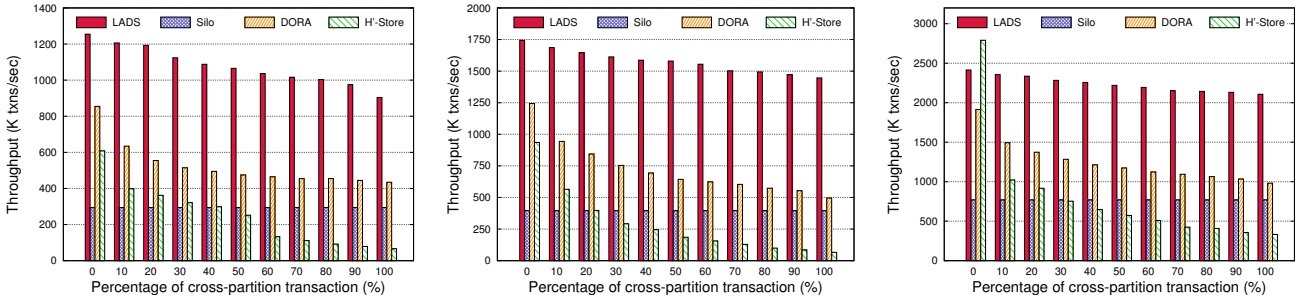
To simulate the skewness among partitions, we apply all the workloads on a single partition. Figure 9(a) shows the throughput of the four systems running the YCSB benchmark. As can be seen, increasing the number of threads has no effect on the throughput of H'-Store, because only one thread can work on the partition. On the other hand, DORA, Silo and LADS scale well with the increased number of threads. DORA maintains logical partitions and the periodic adjustment of the logical partitions reduces the effects of workload skewness. For both Silo and LADS, the whole database is shared by all available threads that can work on the same partition simultaneously. With more contentions in the workloads, Figure 9(b) and Figure 9(c) show that the throughput of Silo decreases significantly. This results from the increased transaction aborts caused by the conflicting data operations. Although both DORA and LADS are more resilient to variations in contention, the throughput of DORA increases much slower than that of LADS, especially for workloads of higher contention. DORA uses locks to resolve conflicts among transactions. Furthermore, to avoid deadlocks, each thread in DORA latches the incoming action queues on all accessed logical partitions before dispatching actions of a transaction phase. This restricts parallelism. In contrast, LADS partitions and executes the workloads according to the dependency graphs computed. Deadlocks and transaction aborts due to conflict are totally eliminated. Moreover, the whole processing logic of LADS is lock/latch-free and does not rely on any centralized component. Thus, LADS copes better with the workload skewness among the partitions under different degrees of contention.

Apart from the workloads skewness among partitions, skewness within a single partition is also common. The skewness of the YCSB workloads is controlled by the parameter θ of the Zipfian distribution. Figure 10(a) shows the impact of the Zipfian θ on the performance of all the four systems when the number of threads is fixed at 24. The database is evenly divided into 24 partitions according to the key range. When θ is small, data accesses are more likely to be uniformly distributed. As θ increases, the data access distribution becomes more skewed, resulting in higher contention. The throughput of Silo drops with the increase of θ, since there are more transaction aborts due to conflict. The throughput of DORA slightly decreases at the beginning and then increases at a later stage. On one hand, higher skewness causes more locking/latching, which restricts the parallelism during the transaction dispatching and thus degrades the throughput. On the other hand, higher skewness benefits DORA's cache efficiency and as a consequence, leads to a throughput increase. The performance of H'-Store and LADS increases with the Zipfian θ. This is because some data items are more frequently accessed, which improves the cache efficiency. To further examine the cache efficiency of LADS, Figure 10(b) shows the CPU utilization breakdown of LADS in visiting L1, L2, L3



(a) Workloads of low contention (θ=0.5) (b) Workloads of medium contention (θ=0.8) (c) Workloads of high contention (θ=0.9)

Figure 11: Effects of cross-partition transactions on the YCSB benchmark.



(a) 4 warehouses (b) 8 warehouses (c) 24 warehouses

Figure 12: Effects of cross-partition transactions on the TPC-C benchmark.

caches and DRAM. We count their accesses and calculate the results according to statistical empirical values as suggested in [18]. The results show that LADS experiences more cache accesses for workloads of higher skewness. Furthermore, Figure 10(c) shows the cache miss rate with respect to the skewness variation. When θ is increased from 0 to 1, the L1, L2 cache miss rate decreases by 1% and 6%, respectively. LADS executes conflicting operations successively during the transaction execution, thereby reducing the cache miss rate. Compared with the L1 and L2 cache miss rates, the L3 cache miss rate decreases slightly more because cores on the same die share the same L3 cache. The above results confirm that LADS exhibits good robustness when faced with skewed single-partition workloads.

5.4 Impact of Cross-Partition Transactions

We next evaluate the impact of cross-partition transactions on performance in the four systems, where the number of available worker threads is fixed to be 24. Figure 11 and Figure 12 show the throughput of running the YCSB benchmark and TPC-C benchmark with different levels of contention as we increase the percentage of cross-partition transactions. For the YCSB benchmark, the database is evenly divided into 24 partitions according to the key range. For the TPC-C benchmark, the database is statically partitioned by warehouse, such that each partition contains all the tables in one warehouse. The number of partitions equals to the number of warehouses.

As shown by the results, H'-Store is sensitive to the percentage of cross-partition transactions. Even a small amount of cross-partition transactions (as little as 1%) can cause a significant drop in its throughput, since a cross-partition transaction blocks threads on all the partitions that it accesses during its execution. Similarly, the throughput of DORA also drops significantly with respect to the increase of the percentage of cross-partition transactions. When a transaction arrives, the thread in DORA latches all the relevant incoming queues of its accessed logical partitions before dispatching it. Furthermore, when a transaction commits, it needs to notify all the partitions involved to release the related locks. This restricts the parallelism and tends to generate a performance bottleneck. In contrast, Silo and LADS are robust to the variation of the percentage of cross-partition transactions, since data are shared across all the worker threads. LADS works like a shared-memory database and eliminates thread blocking during the dependency graph construction. It then partitions and executes the workloads according to the constructed dependency graph like a partitioned database. In short, the above results confirm that LADS could handle cross-partition transactions efficiently.

5.5 Scalability and Durability

In the following set of experiments, we evaluate the scalability of LADS and the cost of ensuring durability.

The results in Figure 9 already show that LADS scales well on the YCSB benchmark with different levels of contention. We now evaluate the scalability using the TPC-C benchmark. Figure 13 shows the results with 24 warehouses and 10% of the transactions involving more than one partition. All the four systems exhibit good scalability when the number of available workers is less than 24. However, all four systems fail to scale when the number of warehouses is less than the number of worker threads, although LADS still shows better scalability than the other three. In TPC-C, every *Payment* transaction updates the same field in the warehouse table. These updates cannot be executed in parallel, even if idle threads are available. As the number of workers increases, H'-Store, DORA and Silo obtain little performance gain. LADS also does not scale smoothly but still maintains an upward trend, albeit at a slower rate, since LADS can execute record actions on other tables in parallel.

LADS provides durability by writing logs to disk along transaction processing, which is done by separate threads. Each logging thread maintains a log buffer and flushes the contents to individual log file before a transaction commits. In Figure 14, the throughput

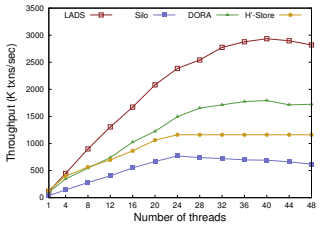


Figure 13: Scalability on the TPC-C benchmark with 24 warehouses.

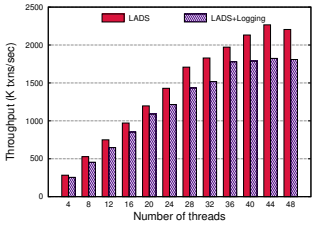


Figure 14: Effects of logging on the YCSB benchmark.

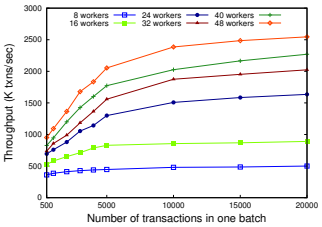


Figure 15: Effects of batch size on throughput.

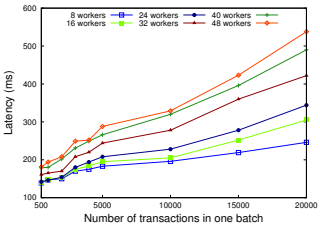
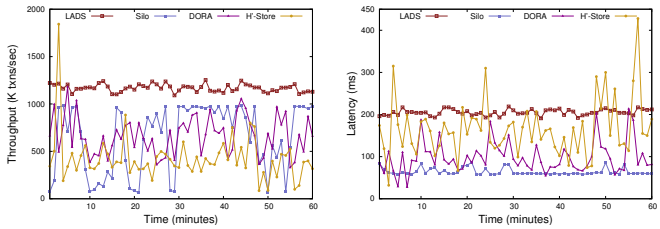
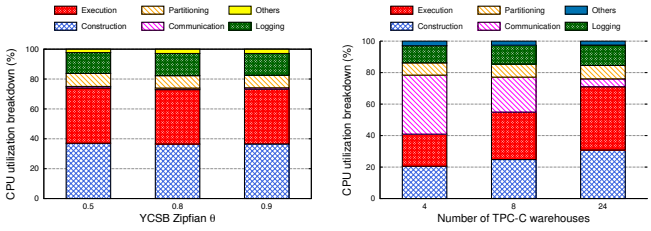


Figure 16: Effects of batch size on latency.



(a) Throughput (b) Latency

Figure 17: A one-hour dynamic workload simulation on the YCSB benchmark.



(a) Cost analysis on YCSB (b) Cost analysis on TPC-C

Figure 18: Cost analysis of LADS. The number of threads is fixed to 24 and the percentage of cross-partition transactions is 10%.

of LADS with logging decreases by 15% compared with that without logging. Although logging incurs additional overheads, it does not affect the scalability of LADS.

5.6 Impact of Batch Size in LADS

In this section, we evaluate the impact of batch size on the performance in LADS. The batch size is constrained by the number of transactions in the transaction queue and a pre-defined maximum batch size. Figure 15 and Figure 16 show the impacts of the batch size on throughput and latency, respectively. When the number of threads is fixed, the throughput first increases with the batch size before plateauing when the computation resources are fully stretched. With more worker threads, LADS always needs a larger batch size to fully exploit their computation potential. As with throughput, the average latency also increases with the batch size. In LADS, threads process the constructed dependency graphs sequentially during transaction execution, thereby increasing the waiting time of the later graphs and thus leads to a higher latency. A trade-off has to be made between the throughput and latency by tuning the batch size. In particular, LADS dynamically adjusts the maximum batch size based on the statistics information and the user-defined requirements.

5.7 Performance on Dynamic Workloads

In practice, workloads are always changing continuously. Without prior knowledge of any future workloads, it is hard for any system to adapt in real time. Hence, robustness to dynamic workloads is an important factor. To evaluate such robustness, we conduct experiments to simulate real-world workloads on the YCSB benchmark by generating transactions with random percentages of cross-partition transactions and contention intensities. In a one-hour simulation, we change the workload parameters every minute, and keep all systems running at full capacity. The average percentage of cross-partition transactions is 10% and the average Zipfian θ is 0.7. For LADS, we set the default size of the transaction queue to 5000 which is equal to LADS' pre-defined maximum batch size.

In Figure 17(a), the throughput of LADS fluctuates within a narrow range. In contrast, the throughputs of H'-Store, DORA and Silo show severe fluctuations because they are sensitive either to the percentage of cross-partition transactions or the intensity of contention. Figure 17(b) shows the average latency with respect to the dynamic workloads. The average latency of H'-Store and DORA also fluctuates severely. For H'-Store and DORA, a cross-partition transaction blocks threads on the partitions that are accessed. Consequently, it increases the waiting time of transactions in the transaction queue and leads to a higher latency. Compared with H'-Store and DORA, the average latency of LADS and Silo shows little variations for workloads of different settings. Since LADS processes transactions in a batched manner, its average latency is affected by both transaction complexity and batch size. The average latency of LADS is higher than that of Silo, since LADS commits a batch of transactions at one time. Overall, the results above confirm that LADS exhibits superior robustness to dynamic workloads.

5.8 Micro-Benchmarking

Finally, we study the performance breakdown in LADS. Figure 18 shows the percentages of CPU utilizations contributed by the individual components of LADS.

As shown in Figure 18(a), for the YCSB benchmark, the cost of each component is almost the same for different contention intensities. Most of the CPUs cycles (about 75%) are spent in resolving and executing transactions. Such effective CPU utilization on the actual transaction processing contributes to the better performance in LADS. The cost of resolving dependency relations within one transaction depends on the complexity of each transaction. Thus, the cost of dependency graph construction remains nearly the same with respect to different contentions. Moreover, by evenly partitioning the workloads in the granularity of records, LADS reduces the effects of thread starving to a certain extent. Thus, the communication cost, including synchronization cost, takes up only a small fraction of the total cost during execution. Hence, the cost of transaction execution stays nearly constant with different contention settings.

In addition, the result for the TPC-C benchmark is shown in Figure 18(b). As can be seen, the communication cost increases when there are fewer warehouses. When the number of warehouses

changes from 4 to 24, the percentage of communication cost drops from 37.5% to 5%. There are two reasons that lead to this change. First, the contention in the TPC-C benchmark is limited to a small set of records. This internal characteristic increases the complexity of obtaining balanced partitions and constrains its execution parallelism. Second, TPC-C transactions have more logical dependency relations, and most of them are also on the same set of records, causing an increase in communication cost.

6. CONCLUSION

In this paper, we proposed LADS, a dynamic single-threaded OLTP system. LADS extends the simplicity of the single-threaded model while overcoming the latter's robustness issue under different kinds of workloads. LADS resolves conflicts among transactions by constructing dependency graphs for which there are no aborts that may arise due to conflicts during the transaction execution. It distributes incoming workloads to available workers in a balanced manner to achieve higher parallelism and efficiency. LADS also leverages modern hardware features. Our extensive experimental study shows that LADS can achieve up to 20× higher throughput than three state-of-the-art systems.

7. REFERENCES

[1] Single-threaded performance of desktop cpus. http://www.cpu-world.com/benchmarks/desktop_CPUs_single.html.

[2] TPC-C. http://www.tpc.org/tpcc/.

[3] TPC-C historical results. http://www.tpc.org/information/historical.asp.

[4] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *TODS*, 1987.

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

[6] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis. Concurrency control for step-decomposed transactions. *Information Systems*, 1999.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987.

[8] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. *University of Wisconsin-Madison* 1989.

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

[10] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. *Implementation techniques for main memory database systems*. ACM, 1984.

[11] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. *SIGMOD*. ACM, 2013.

[12] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. *SIGMOD*, 2015.

[13] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *VLDB*, 2015.

[14] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *TODS*, 1983.

[15] V. Gottemukkala and T. J. Lehman. Locking and latching in a memory-resident database system. *VLDB*, 1992.

[16] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Record*. ACM, 1994.

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

[18] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 2000.

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

[20] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-Store: a high-performance, distributed main memory transaction processing system. *VLDB*, 2008.

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

[22] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 1970.

[23] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. *SIGMOD*, 2015.

[24] H. Kimura, G. Graefe, and H. A. Kuno. Efficient locking techniques for databases on modern hardware. *ADMS@ VLDB*, 2012.

[25] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *TODS*, 1981.

[26] R. H. Louie, Y. Li, and B. Vucetic. Practical physical layer network coding for two-way relay channels: performance analysis and comparison. *Wireless Communications*, 2010.

[27] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. *ICDE*, 2014.

[28] N. Manchanda and K. Anand. Non-uniform memory access (numa). *New York University*, 2010.

[29] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for oltp and transactional web applications. *ICDE*, 2004.

[30] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. *SIGMOD Record*, 1992.

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

[32] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *VLDB*, 2010.

[33] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: page latch-free shared-everything oltp. *VLDB*, 2011.

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

[35] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *VLDB*, 2012.

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

[37] R. R. Schaller. Moore's law: past, present and future. *Spectrum*, 1997.

[38] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *TODS*, 1995.

[39] T. Subasu and J. Alonso. Database engines on multicores, why parallelize when you can distribute. In *EuroSys*, 2011.

[40] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. *VLDB*, 2014.

[41] K.-L. Tan, Q. Cai, B. C. Ooi, W.-F. Wong, C. Yao, and H. Zhang. In-memory databases–challenges and opportunities.

[42] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *SOSP*, 2013.

[43] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 1990.

[44] G. Weikum and G. Vossen. Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. *Elsevier*, 2001.

[45] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, sql or c++. *HPTS*, 1997.

[46] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *VLDB*, 2014.