CARNEGIE MELLON DATABASE GROUP

FALL 2015

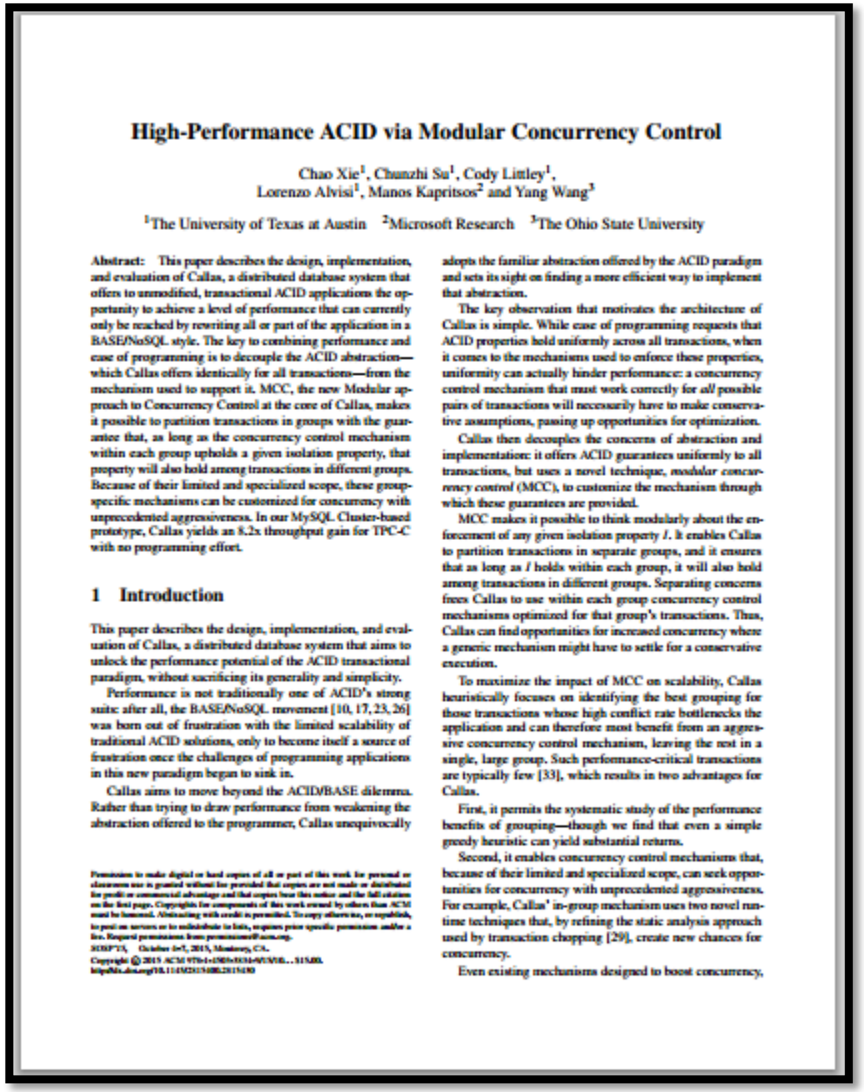
High-Performance ACID via Modular Concurrency Control

Chao Xie[1], Chunzhi Su[1], Cody Littley[1], Lorenzo Alvisi[1], Manos Kapritsos[2], Yang Wang[3]

(slides by Mrigesh)

TODAY'S READING

- Background and Motivation
- Callas' Key Contributions
- Transaction Grouping
- Experiments and Evaluation
- Conclusions and "The Future"



High-Performance ACID via Modular Concurrency Control

Chao Xie[1], Chunzhi Su[1], Cody Littley[1],
Lorenzo Alvisi[1], Manos Kapritsos[2] and Yang Wang[3]

[1]The University of Texas at Austin [2]Microsoft Research [3]The Ohio State University

Abstract: This paper describes the design, implementation, and evaluation of Callas, a distributed database system that offers to unmodified, transactional ACID applications the opportunity to achieve a level of performance that can currently only be reached by rewriting all or part of the application in a BASE/NoSQL style. The key to combining performance and ease of programming is to decouple the ACID abstraction—which Callas offers identically for all transactions—from the mechanism used to support it. MCC, the new Modular approach to Concurrency Control at the core of Callas, makes it possible to partition transactions in groups with the guarantee that, as long as the concurrency control mechanism within each group upholds a given isolation property, that property will also hold among transactions in different groups. Because of their limited and specialized scope, these group-specific mechanisms can be customized for concurrency with unprecedented aggressiveness. In our MySQL Cluster-based prototype, Callas yields an 8.2x throughput gain for TPC-C with no programming effort.

1 Introduction

This paper describes the design, implementation, and evaluation of Callas, a distributed database system that aims to unlock the performance potential of the ACID transactional paradigm, without sacrificing its generality and simplicity.

Performance is not traditionally one of ACID's strong suits: after all, the BASE/NoSQL movement [10, 17, 23, 26] was born out of frustration with the limited scalability of traditional ACID solutions, only to become itself a source of frustration once the challenges of programming applications in this new paradigm began to sink in.

Callas aims to move beyond the ACID/BASE dilemma. Rather than trying to draw performance from weakening the abstraction offered to the programmer, Callas unequivocally adopts the familiar abstraction offered by the ACID paradigm and sets its sight on finding a more efficient way to implement that abstraction.

The key observation that motivates the architecture of Callas is simple. While ease of programming requests that ACID properties hold uniformly across all transactions, when it comes to the mechanisms used to enforce these properties, uniformity can actually hinder performance: a concurrency control mechanism that must work correctly for *all* possible pairs of transactions will necessarily have to make conservative assumptions, passing up opportunities for optimization.

Callas then decouples the concerns of abstraction and implementation: it offers ACID guarantees uniformly to all transactions, but uses a novel technique, *modular concurrency control* (MCC), to customize the mechanism through which these guarantees are provided.

MCC makes it possible to think modularly about the enforcement of any given isolation property I. It enables Callas to partition transactions in separate groups, and it ensures that as long as I holds within each group, it will also hold among transactions in different groups. Separating concerns frees Callas to use within each group concurrency control mechanisms optimized for that group's transactions. Thus, Callas can find opportunities for increased concurrency where a generic mechanism might have to settle for a conservative execution.

To maximize the impact of MCC on scalability, Callas heuristically focuses on identifying the best grouping for those transactions whose high conflict rate bottlenecks the application and can therefore most benefit from an aggressive concurrency control mechanism, leaving the rest in a single, large group. Such performance-critical transactions are typically few [33], which results in two advantages for Callas.

First, it permits the systematic study of the performance benefits of grouping—though we find that even a simple greedy heuristic can yield substantial returns.

Second, it enables concurrency control mechanisms that, because of their limited and specialized scope, can seek opportunities for concurrency with unprecedented aggressiveness. For example, Callas' in-group mechanism uses two novel run-time techniques that, by refining the static analysis approach used by transaction chopping [29], create new chances for concurrency.

Even existing mechanisms designed to boost concurrency,

SOSP'15, October 4–7, 2015, Monterey, CA.

[illegible]

THE PROBLEM

- The ACID paradigm offers an easy way to think about (and program applications involving) transactions
- However, performance is not a strong suit for ACID systems, especially when distributed (higher latency)
- This is the price of isolation: intermediate states of a transaction are hidden from other transactions

PREVIOUS SOLUTIONS

- Spanner, H-Store avoid 2PC for certain transactions
- A move towards BASE (Basic Availability, Soft-state & Eventual consistency)
  - e.g. Salt, by the same group, BASEified some transactions
- SDD-1 used statically-defined transaction classes, with fixed read/write sets
- Lynx and Sagas used SC-cycles to chop transactions

OBSERVATIONS

- Traditionally ACID guarantees are implemented uniformly to all transactions
- Conservatism guarantees correctness, but not performance
- Callas introduces “Modular Concurrency”, where a given isolation property is enforced at two-levels– within a group and (by extension) across groups

WHAT'S THE BIG IDEA?

- Modular Concurrency Control
  - Separation of concerns
  - Decouple ACID abstraction from the mechanism used to support it
  - General-purpose solution

KEY CONTRIBUTIONS

- Systematic analysis of “transaction-grouping”
- More aggressive use of traditional concurrency-boosting
- Runtime Pipelining- in-group mechanism to:
  - Allow concurrent execution of transactions based on real-time static analysis of an SC-graph
  - Guarantee atomicity while preventing Aborted Reads and avoiding enforcing rollback safety

AN ISOLATION REFRESHER

- DSG: A graph with nodes = committed transactions (T) and directed edges that indicate a scope for conflict between them
  - Read dependency: T_i installs a version x_i of object x. Tj reads x_i
  - Anti dependency: T_i reads a version x_k of object x. Tj installs next version of x.
  - Write dependency: T_i installs a version x_i of object x. Tj installs next version of x.

AN ISOLATION REFRESHER

- Circularity: The execution history contains a directed cycle
- Aborted Reads: A committed transaction T_2 reads some object modified by an aborted transaction T_1.
- Intermediate Reads: A committed transaction T_2 reads a version of an object x written by another transaction T_1 that was not T_1's final modification of x.
- Standard for serializability: Preventing these 3 states

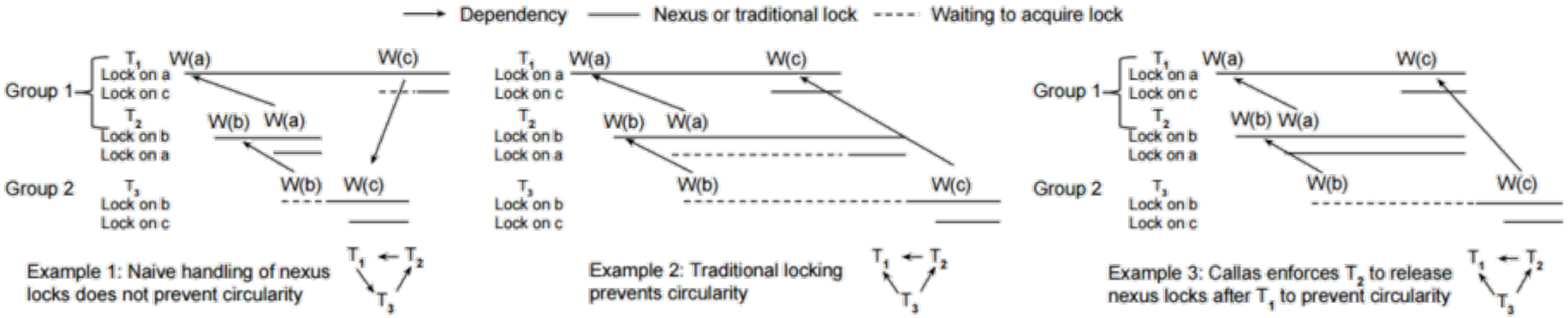
ANALYSIS OF CALLAS

- Nexus Locks
- Automated Transaction Chopping
- Runtime Pipelining
- Implementation and Evaluation

NEXUS LOCKS

- Core of Callas' concurrency control mechanism
- New type of lock
  - Regulates conflicts between transactions from different groups
  - Places no constraint on transactions within same group
- In some cases, the release of a lock can be delayed

CROSS-GROUP ISOLATION



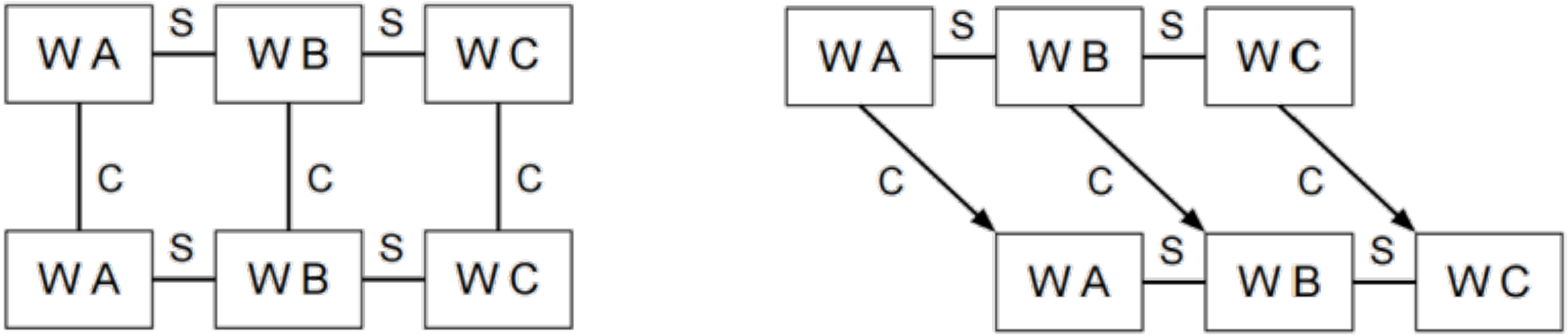
INTRA-GROUP ISOLATION

- Secret Sauce that enables Callas' performance gains
- Effective optimization within groups requires:
  - Appropriate grouping techniques that maximize the potential for concurrency
  - Identifying mechanisms to increase concurrency within a group

TRANSACTION CHOPPING

- Break transactions into constituent sub-transactions, which can be interleaved
- Analysis using an SC-graph:
  - Vertices = Candidate sub-transactions
  - S-edges = Undirected links within the same transaction
  - C-edges = Connected links between different transactions accessing the same object
- Need to ensure: rollback safety and prevention of SC-cycles

RUNTIME PIPELINING



(a) SC-cycle analysis cannot chop (b) Runtime Pipelining

- Operations within a transaction piece are only allowed to access read-write tables of the same rank.
- For any pair of pieces p1 and p2 of a given transaction that access read-write tables, if p1 is executed before p2, then p1 must access tables of smaller rank than p2.

RUNTIME PIPELINING IN PRACTICE

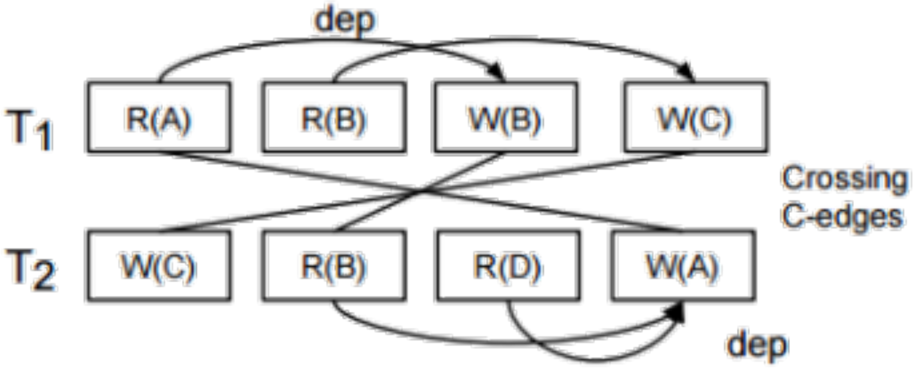
STEP ONE

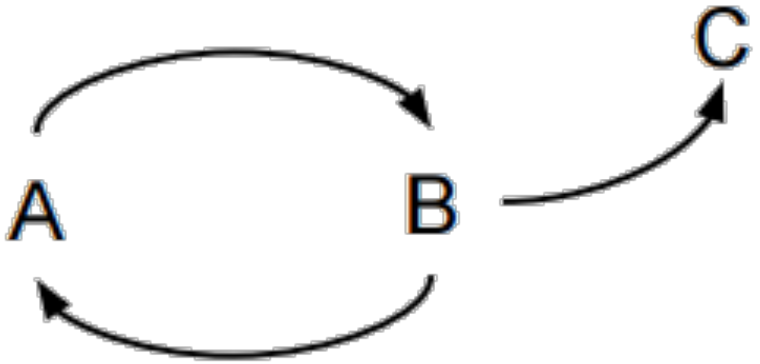
Input: transactions with dependency information → Build table dependency graph G → Sort graph vertices and rank tables → In each transaction, place operations that access tables of the same rank in the same piece

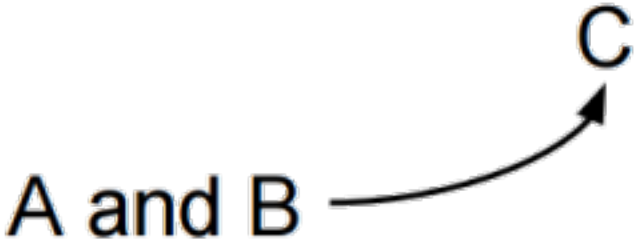
STEP TWO

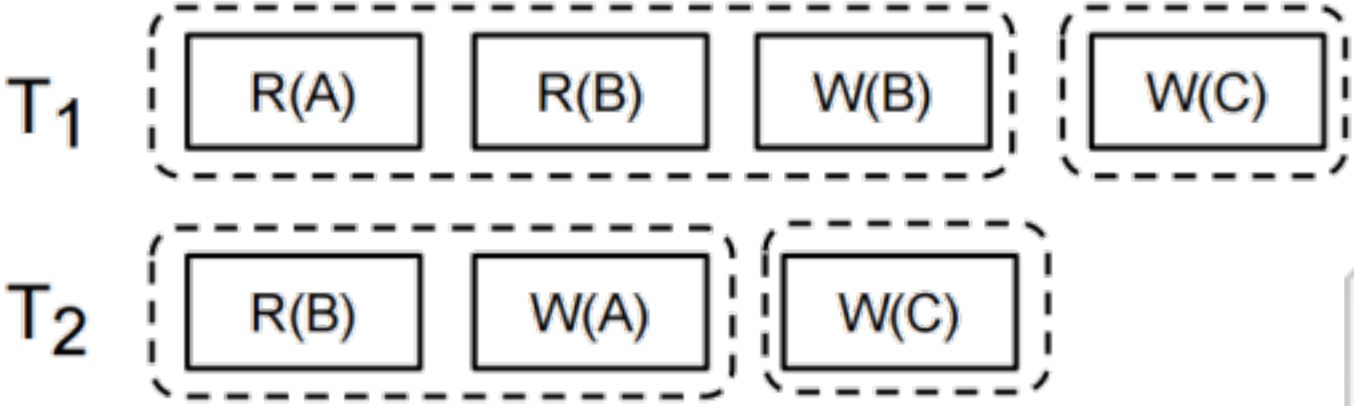
Build operation dependency graph → For each transaction, sort vertices to order pieces → Order operations within each piece according to their original order

CALLAS IN PRACTICE (STEP ONE)

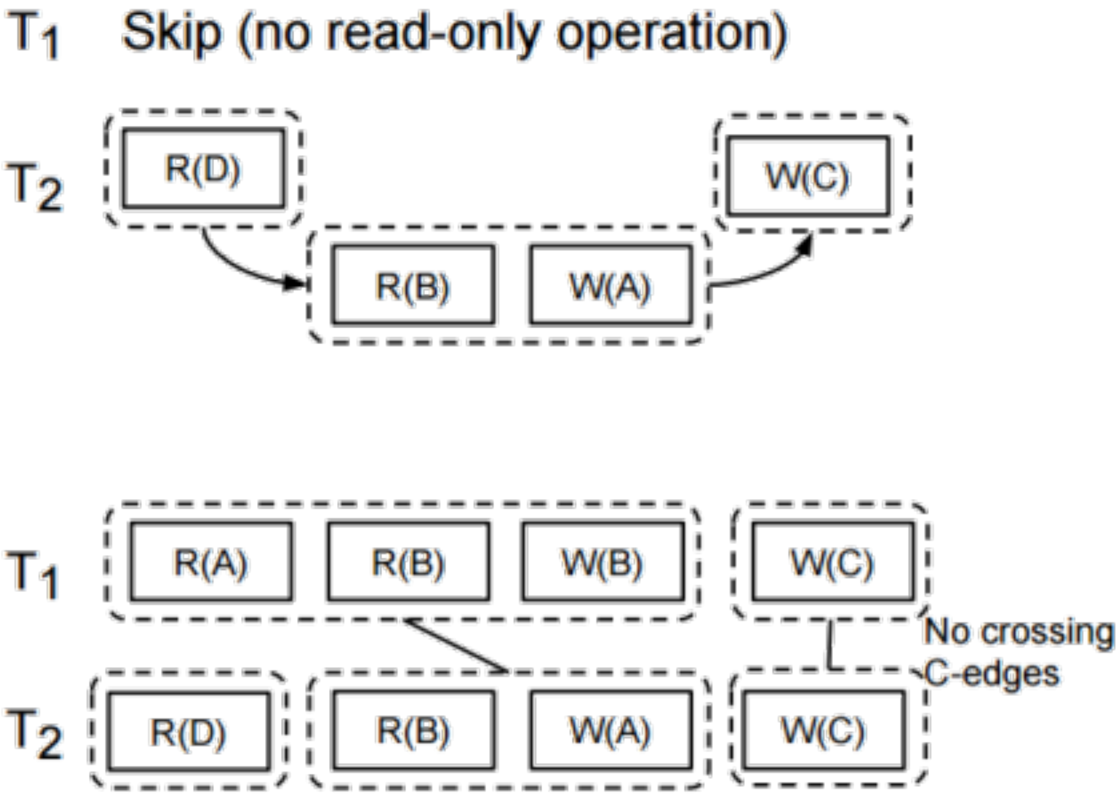








CALLAS IN PRACTICE (STEP TWO)



EVALUATION : GOALS

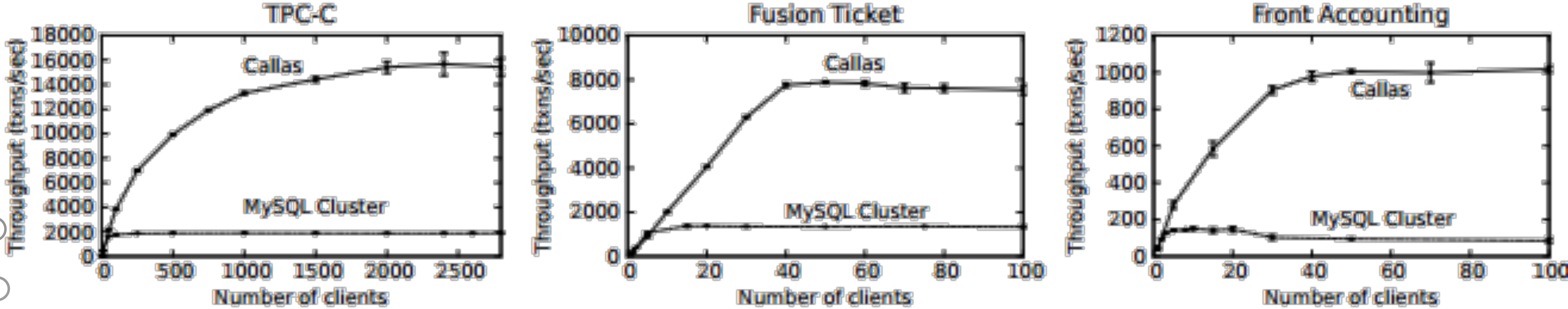
- Compare performance improvement over equivalent ACID
- Compare performance improvement of each optimization
- Impact of different parameters / settings on performance
- Overhead of Nexus Locks

EVALUATION : TESTBED

- Three applications:
  - TPC-C
  - Fusion Ticket
  - Front Accounting
- Experimental setup:
  - MySQL Cluster
  - 10 database partitions
  - 3-way replicated
  - System saturated with load

- On Dell PowerEdge R320 machines
  - Xeon E5-2450 processor, 16 GB of memory, four 7200 RPM SATA disks, and 1 Gb Ethernet

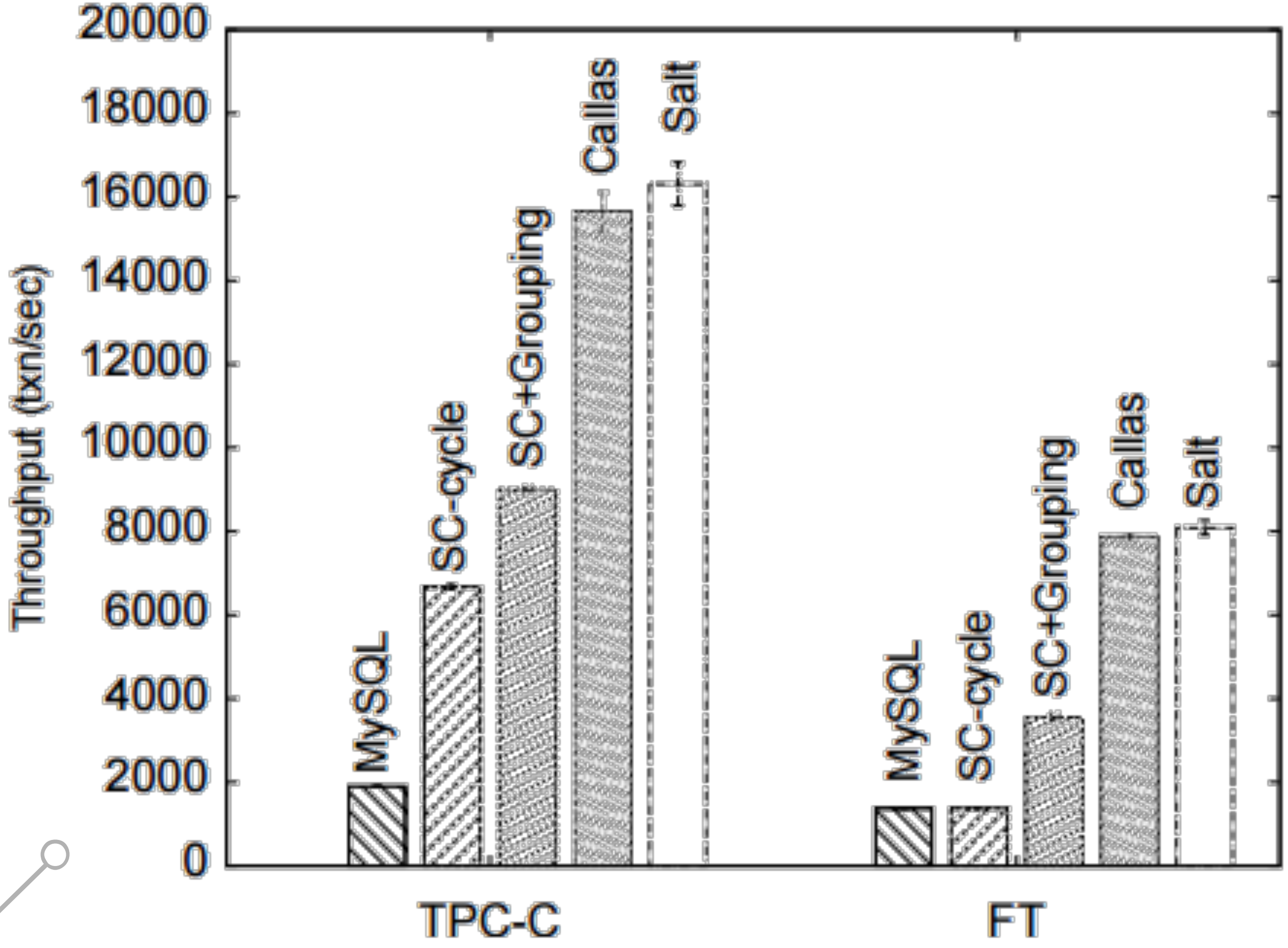
EVALUATION : CALLAS V/S UNMODIFIED



- Callas performs:
  - 8.2x better on TPC-C
  - 6.7x better on Front Accounting
  - 5.7x better on Fusion Ticket

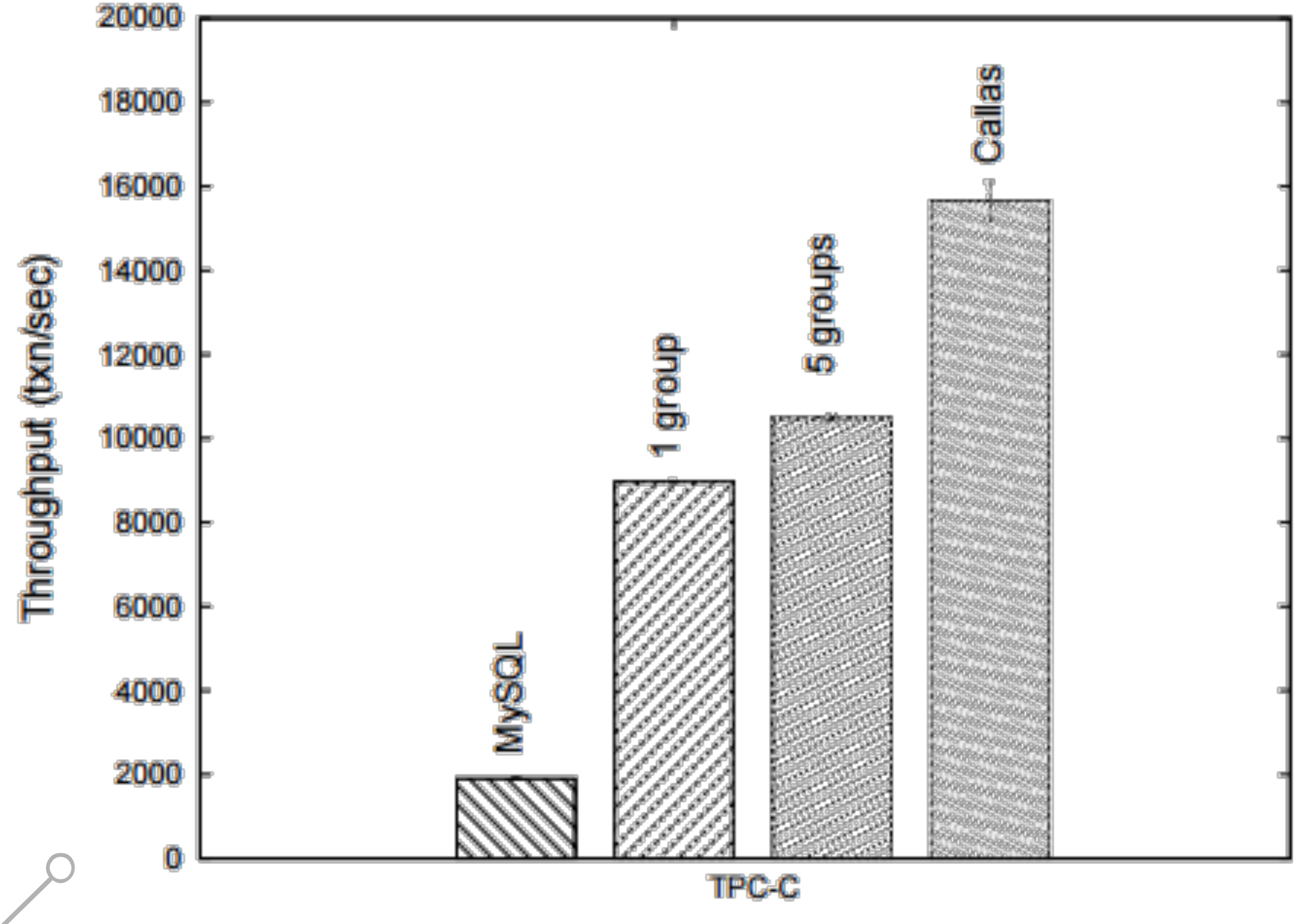
Latency(ms)	MySQL Quantile		Callas Quantile	
	50th	99th	50th	99th
new_order (TPC-C)	26	51	28	50.5
checkout (FT)	12	25.3	12	25
delivery (FA)	36.3	69	36.6	66

EVALUATION : INDIVIDUAL OPTIMIZATIONS



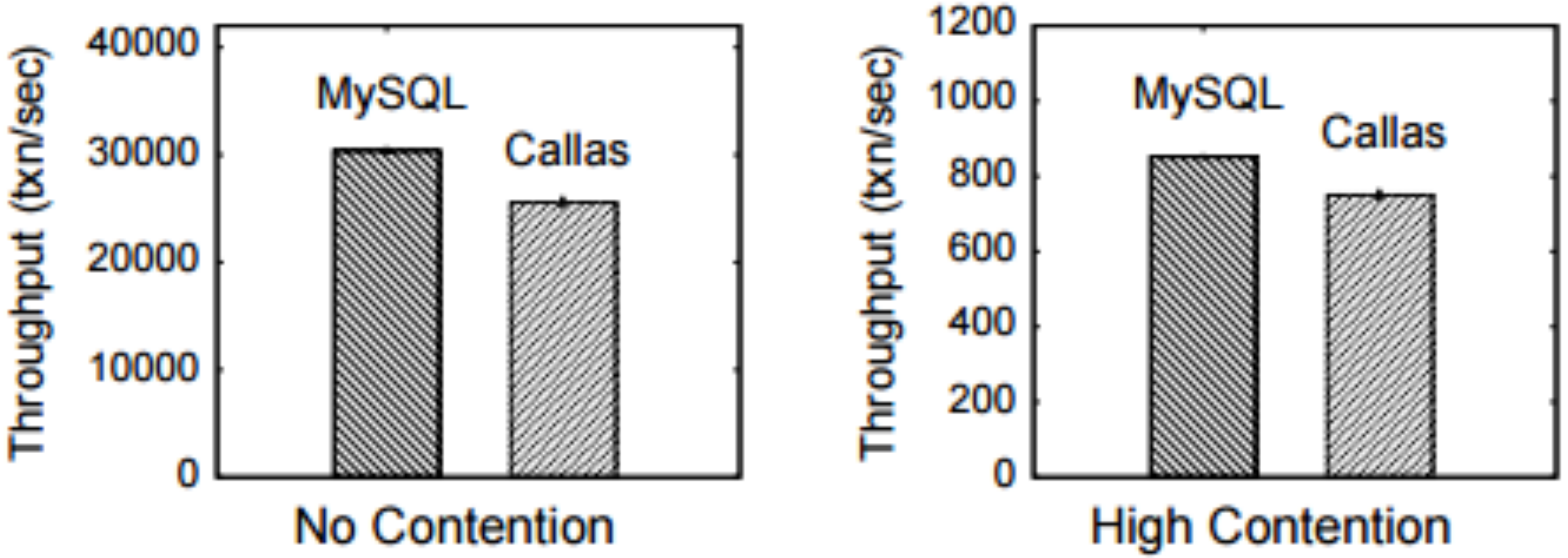
- Optimizations are App-dependent
- In TPC-C, simple choppings in the same group improve performance; FT benefits more from creating multiple groups
- In both cases, heuristically creating groups has a significant impact

EVALUATION : DIFFERENT GROUPINGS



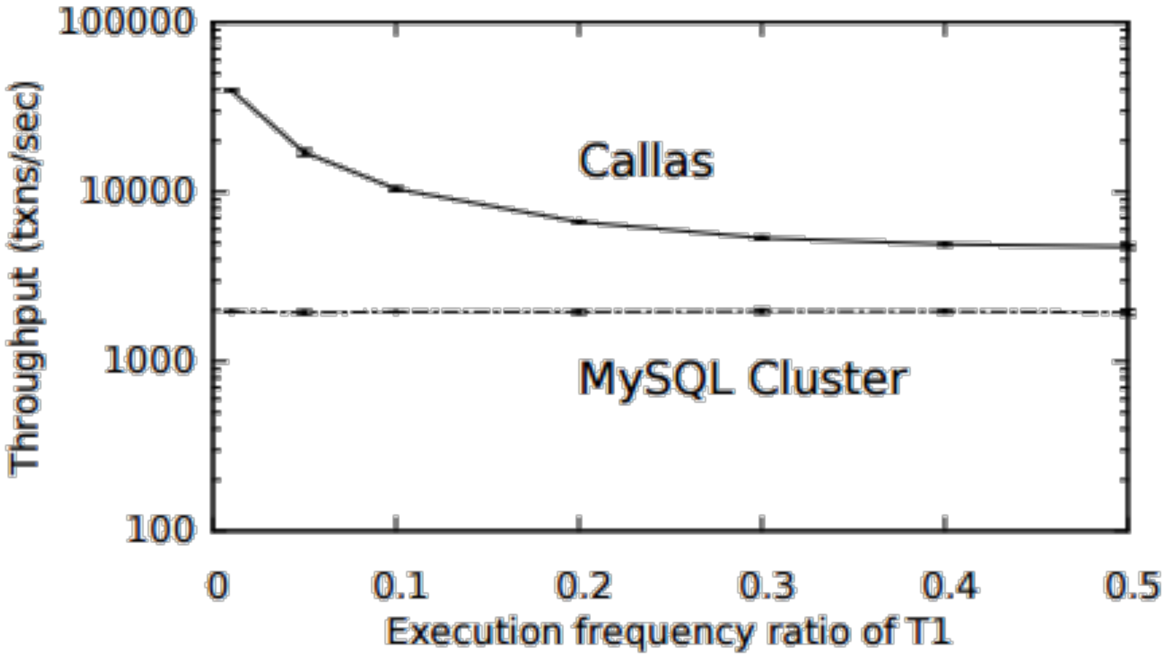
- Simple runtime pipelining on a single group has a significant improvement in performance
- Naively breaking into 5 groups (one transaction per group) has a slight further increase
- Intelligent grouping to place frequently conflicting transactions in the same group has a 50% gain on TPC-C

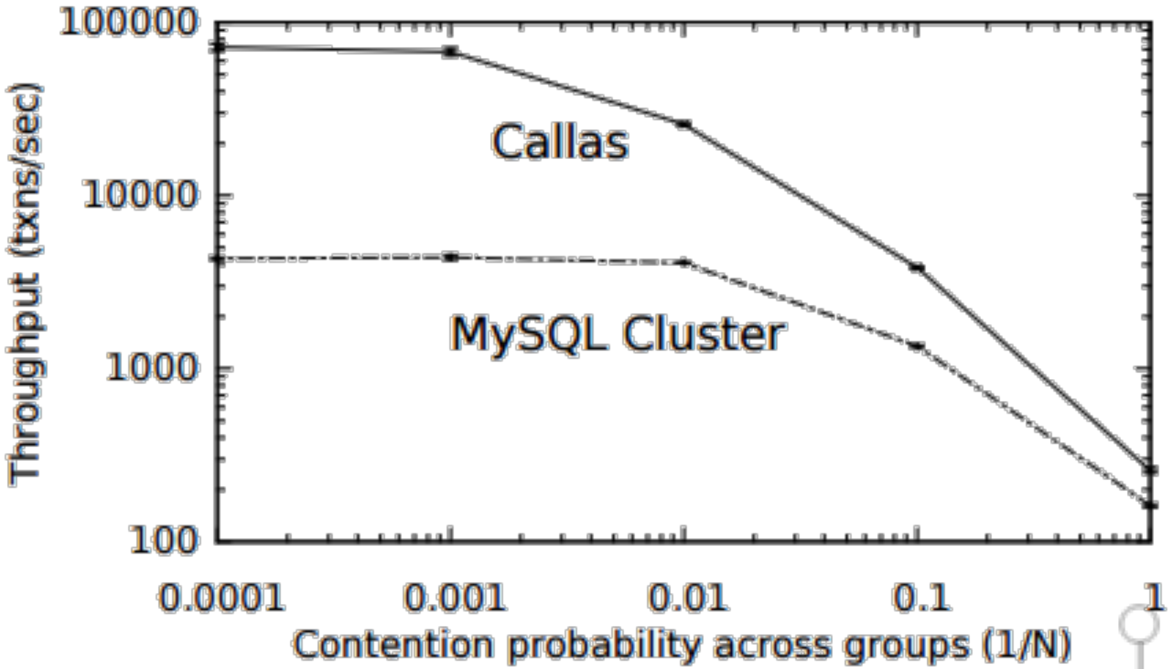
EVALUATION : NEXUS LOCKS



- In a micro-benchmark designed to neutralize Callas' benefits, MySQL performs 19% better when no contention, and 13% better when contention is high
- Bottleneck is the increased CPU overhead of maintaining Nexus locks
- In high-contention, Callas' message passing to enforce ordering in the bottleneck

EVALUATION : CONTENTION RATE





- When inter-group contention is low, Callas >> MySQL Cluster
- Contention rate increases (due to more frequent transactions or more likely contention decreases this differential)
- Even in the worst case, Callas performs twice as well as the ACID MySQL Cluster

CONCLUSIONS AND FUTURE

- Callas relies on sound system design principles, and leverages smaller groups to improve performance
- MySQL Cluster based prototype of Callas exhibits significant throughput gains
- Need to verify performance gains on different systems (e.g. using OCC, MVCC) with different backends

FINI