High-Performance ACID via Modular Concurrency Control  
(Technical Report)

Chao Xie\textsuperscript{1}, Chunzhi Su\textsuperscript{1}, Cody Littley\textsuperscript{1}, 
Lorenzo Alvisi\textsuperscript{1}, Manos Kapritsos\textsuperscript{2} and Yang Wang\textsuperscript{3}

\textsuperscript{1}The University of Texas at Austin \quad \textsuperscript{2}Microsoft Research \quad \textsuperscript{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 runtime 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, however, can benefit from a more limited scope. In particular, we find that also traditional transaction chopping, which relies on the absence of certain dependency cycles among all transactions, becomes much more incisive when the lack of circular dependency must apply only to the small number of transactions within a group.
In summary, we make the following contributions:

- We propose MCC, a new, modular approach to concurrency control. By decoupling abstraction from mechanism, MCC retains the simplicity of a uniform ACID API; by separating concerns, it lets each module customize its internal concurrency control mechanism to achieve greater concurrency without sacrificing safety.

- We introduce Runtime Pipelining, a technique that leverages execution-time information to aggressively weaken within a group the conservative requirements of the current theory of safe transaction chopping [29] and gain, as a result, unprecedented opportunities for concurrency. The key to the effectiveness of Runtime Pipelining is the flexibility offered by MCC, which makes this technique applicable within small groups of well-suited transactions.

- We present the design, implementation and evaluation of Callas, a prototype implementation of MCC within a modified MySQL Cluster distributed database. Our evaluation of Callas suggests that MCC can deliver significant performance gains to unmodified ACID applications. For example, we find that, for TPC-C, Callas achieves an 8.2x speedup over MySQL Cluster without requiring any programming effort.

The rest of the paper is organized as follows. After Section 2 discusses why an undifferentiated concurrency control mechanism is undesirable, Section 3 introduces MCC and specifies the correctness conditions that any valid instantiation of MCC must meet. Section 4 and Section 5 present the mechanisms Callas uses to ensure isolation across groups and within each group, respectively. The implementation of Callas is the topic of Section 6, while Section 7 presents the results of our experimental evaluation. Section 8 discusses related work, and Section 9 concludes the paper.

## 2 The cost of uniformity

The power of the ACID paradigm lies in its simplicity. Developers need only wrap their code in an ACID transaction, and it is guaranteed to be executed atomically, to leave the database in a consistent state, to be isolated from any other transactions, and to be durable. One of the great assets of this abstraction is that it applies uniformly to all transactions, independent of the internal logic of other transactions, thus freeing the developer from having to worry about transaction interleavings.

Current ACID databases support this uniform abstraction with an equally uniform mechanism. Notwithstanding its simplicity, when it comes to enforcing isolation this choice can become an obstacle to performance and scalability. Whether using locking or optimistic concurrency control (OCC), current ACID databases rely on one-size-fits-all—and thus fundamentally conservative—mechanisms to ensure isolation.

For example, when a transaction accesses an object (e.g., a database row), a lock-based mechanism must acquire a lock that is held until the end of the transaction, preventing all other transactions from observing intermediate states of that transaction. Perhaps surprisingly, given their name, OCC mechanisms are, in their own way, equally conservative. Although they allow transactions to speculatively execute in parallel without acquiring locks, they do not refine the criteria for determining contention, but simply delay the check: if contention is detected at commit time, they force all but one of the contending transactions to rollback.

By treating all transactions equally, one-size-fits-all mechanisms cannot take advantage of workload-specific optimizations. Isolation is uniformly enforced to prevent all other transactions from observing intermediate states. In some circumstances, however, such precautions are excessive: it is quite common to find transactions that can safely expose some of their intermediate states to some other transactions.

Consider, for example, how a lock-based mechanism would handle the simple banking application that uses the two transactions defined in Figure 1: `transfer_balance` deducts some amount from the `bal_orig` account and adds it to `bal_dest`; `sum_balance` computes the total assets across the two accounts.

When these transactions can execute concurrently, uniformly enforcing isolation requires `transfer_balance` to keep the lock on `bal_dest` until the transaction commits, to prevent `sum_balance` from computing the wrong total by observing the intermediate state where `bal_dest` has been credited but `bal_orig` not yet charged. Keeping the locks for so long, however, also prevents other instances of the `transfer_balance` transaction from executing concurrently, even though they could do so safely, since their operations commute. \(^1\) Ideally, one would like to release the lock on `bal_orig` after the amount is deducted from it, but only for other `transfer_balance` transactions; `sum_balance` should still be prevented from observing the intermediate state. Recent work on the Salt distributed database [33] shows that leveraging this insight can yield significant performance benefits. To extract them, however, Salt forgoes the simplicity of a uniform ACID paradigm and instead introduces a new abstraction, BASE transactions. Salt’s isolation mechanism offers BASE transactions the ability to make certain intermediate states visible to other BASE transactions, but invisible to ACID transactions. The cost of this flexibility, however, is added complexity for the pro-

---

\(^1\) Commutativity is only one example of the missed opportunities for greater concurrency that constitute the cost of uniformity—we discuss the performance implications for transaction chopping [29] in Section 5.
grammer. Even though it suffices to rewrite only a few [33] performance-critical ACID transactions as BASE transactions to reap substantial performance benefits, the programming effort involved is still significant and can easily introduce bugs.

In Salt, abandoning a uniform concurrency control mechanism to increase performance has led, as if by necessity, to also surrendering the benefits of a uniform ACID abstraction. In its own attempts to leverage the same insight that motivated Salt, Callas strives to stay clear of the pitfall of tightly coupling the scope of mechanism and abstraction, and comes to a fundamentally different conclusion.

3 A modular approach to isolation

Callas aims to offer to unmodified, transactional ACID applications the kind of performance that previously could only be achieved by rewriting applications in a BASE/NoSQL style. Such coding exercises are notorious for being error-prone and time consuming [30], even when backed by Object-Relational Mapping systems [12]. Callas’ goal is to do away with them completely, with only a negligible cost in performance.

The design of Callas is based on a simple proposition: that the key for combining performance and ease of programming is to decouple the ACID abstraction—which should hold identically for all transactions—from the mechanism used to support it—which should instead adapt to the unique characteristics of different transactions. The approach that we propose is rooted in three main observations.

First, no existing programming paradigm approaches the simplicity offered by ACID. Such is its superiority on this front to bring into question whether any performance benefit that a BASE alternative can deliver is actually worth the trouble [18, 30, 33].

Second, significant improvements to the performance of ACID are unlikely to come from techniques that rely on properties that must hold for all of the transactions in a given application. A case in point is transaction chopping [29], an elegant technique that can yield greater concurrency while maintaining serializability, but only if a specific property (which can be formalized as the absence of SC-cycles\footnote{We will discuss SC-cycles in more detail in Section 5.}) holds across the entire set of an application’s transactions. In practice, enforcing this property can often significantly limit opportunities for concurrency in applications that suffer from high contention.

Third, as the Salt project has recently demonstrated, the potential performance gains to be had by allowing individual transactions to export multiple granularities of isolation can be substantial [33].

The architecture of Callas leverages these observations by supporting a modular approach to regulating concurrency, realized through a novel technique we call modular concurrency control (MCC). The vision that motivates MCC is simple. Instead of relying on a single concurrency control mechanism for all transactions, MCC partitions transactions in groups and enables the flexibility to assign to each group its own private concurrency control mechanism; being charged with regulating concurrency only for the transactions within their own groups, these mechanisms can be much more aggressive while still upholding safety. Finally, MCC offers a mechanism to properly handle conflicts among transactions in different groups.

An attractive feature of this approach is its generality. First, it imposes no restrictions on the types of transactions it can handle. In particular, it does not require to predefine all transactions that will be run in the system: interactive or external transactions can always be handled by placing them in a separate group that uses a standard, conservative concurrency control mechanism. Second, although our current implementation of Callas leverages modularity only within the context of lock-based mechanisms, MCC does not, in principle, depend on whether concurrency control is implemented using locks or OCC, or on whether the targeted isolation level relies on a single version or a multiversion database—we leave a thorough exploration of the performance opportunities offered by this generality to future work.

To succeed, this high-level plan must address two complementary concerns: performance and correctness.

The key factor for performance is to group transactions appropriately, in order for each group to best exploit opportunities for optimizations. Not all grouping choices are equally sensitive, however: optimizing grouping for transactions that run infrequently or are lightweight is less critical. Callas therefore heuristically assigns those transactions to a single group, and instead focuses on determining the most favorable grouping for the transactions that are primarily shaping the performance profile of a given application. We discuss the policy and mechanism used by Callas to group transactions in Sections 4 and 6.

Establishing correctness involves a two-step process: given any of the traditional ACID isolation guarantees, first prove that each group, separately, satisfies the guarantee; and then, under the assumption that all groups do, that the isolation guarantee is upheld globally.

The theoretical underpinnings that Callas uses to discharge these obligations are found in the general theory for expressing isolation levels introduced by Adya et al [11]. We quickly review some of the key features of their framework below.

3.1 Defining isolation

Similar to prior specifications of the ANSI SQL isolation levels, Adya et al. define isolation levels on the basis of the undesirable phenomena they proscribe. Unlike prior specifications, however, theirs applies not only to lock-based implementa-
tions, but also to optimistic and multiversion concurrency control schemes [15, 25]. Their elegant formulation refines the classic approach of leveraging a serializability graph to express whether a history is serializable: they express necessary conditions that apply to weaker notions of isolation as requirements on the structure of a new graph they define, called the Direct Serialization Graph (DSG). Each node in the DSG corresponds to a committed transaction, and each directed edge from transaction $T_i$ to $T_j$ indicates one of the following types of conflict between them:

- **Read dependency.** $T_i$ installs a version $x_i$ of an object $x$ and $T_j$ reads $x_j$.
- **Anti-dependency.** $T_i$ reads a version $x_k$ of $x$, and $T_j$ installs $x$'s next version.
- **Write dependency.** $T_i$ installs a version $x_i$ of $x$, and $T_j$ installs $x$'s next version.

Read and Anti-dependencies can be easily generalized to range reads that, rather than explicitly naming an item, apply to all items for which a given predicate (e.g., an SQL statement) holds [11].

The DSG is central to this formulation because the occurrence of some of the phenomena proscribed by a given isolation level is equivalent to the DSG exhibiting a refinement of the following condition:

- **Circularity.** The execution history contains a directed cycle.

The refinement consists of specifying which types of edges can be used to construct the cycle: the more stringent the isolation level, the larger the set of cycles to be prevented (and of phenomena to be proscribed). For example, isolation levels that forbid reading data that has not been committed (*dirty reads*) require the flow of information between any two transactions to be unidirectional—which can be achieved by proscribing DSG cycles consisting only of dependency edges [11]. Achieving serializability, however, requires ruling out also cycles that include anti-dependency edges.

Not all phenomena to be proscribed, however, correspond to cycles in the DSG. In particular, every ANSI SQL isolation level that does not allow dirty reads must also avoid the following two phenomena:

- **Aborted Reads.** A committed transaction $T_2$ reads some object (possibly via a predicate) modified by an aborted transaction $T_1$.
- **Intermediate Reads.** A committed transaction $T_2$ reads a version of an object $x$ (possibly via a predicate) written by another transaction $T_1$ that was not $T_1$’s final modification of $x$.

All popular ACID isolation levels avoid these phenomena, so, to simplify our presentation, we henceforth only consider isolation levels that do.

### 3.2 Establishing correctness

We leverage the formalization of Adya et al. to specify, for any given isolation level, the conditions that an instantiation of the Callas architecture must satisfy to guarantee correctness:

**Within each group** The concurrency control mechanism for group $G$ must prevent Aborted Reads and Intermediate Reads if $T_1$ and $T_2$ are both in $G$, and prevent Circularity (as defined for the targeted isolation level) when all transactions on the cycle are in $G$.

**Across groups** Aborted Reads and Intermediate Reads must be prevented if $T_1$ and $T_2$ are from different groups. Further, Circularity (as defined for the targeted isolation level) must be prevented if at least two transactions on the cycle are from different groups.

### 3.3 Callas at a glance

The next two sections describe the design of Callas along the two axes we have used to articulate correctness. Section 4 describes how Callas leverages a new class of locks, called *nexus locks*, to prevent Circularity and proscribe Aborted and Intermediate reads across groups.

Section 5 introduces a new in-group concurrency control mechanism, called *Runtime Pipelining*, designed to leverage the modularity of Callas: since it regulates concurrency for only a small number of transactions, it can afford to apply aggressive optimizations. Runtime Pipelining owes much of its performance—as well as its name—to its integration of static analysis with novel run-time checks that guarantee safety while increasing opportunities for concurrency.

### 4 Enforcing isolation across groups

The design of the mechanism Callas uses to guarantee inter-group isolation is driven by several considerations. Foremost, of course, is safety: the mechanism should enforce the correctness conditions identified in Section 3.2. Not far behind, however, are performance and liveness. First, we would like the inter-group mechanism to disrupt as little as possible the ability of the group-specific mechanisms to extract concurrency from the transactions they regulate. Second, we would like to guarantee fairness: the eagerness of exploiting performance opportunities within a group should not cause transactions from a less fortunate group to starve.

Callas meets these requirements using a simple lock-based approach. This choice is pragmatic: although there is nothing in Callas’ architecture that would prevent the use of OCC, the MySQL Cluster distributed database we modify to implement Callas does not support it.\(^3\) Indeed, any reasonable

\(^3\)MySQL Cluster supports only two versions of each object. While this feature guarantees that reads never block, it falls short of full multiversion concurrency control (MVCC).
implementation of the inter-group mechanism that meets the
above requirement of minimal disruption will do.

At the core of Callas' inter-group mechanism are nexus
locks, a new type of lock whose role is to regulate conflicts be-
tween transactions that belong to different groups while leave-
ing transactions within each group relatively unconstrained.
Nexus locks in Callas are ubiquitous: any transaction, be-
fore being allowed to perform a read or write operation on
a database row, must acquire the corresponding nexus lock.
This demand may seem to run contrary to the requirement
of making the inter-group mechanism inconspicuous to the
concurrency control mechanisms specific to each group. The
key to resolving this apparent tension lies in the flexibility
of nexus locks. When two transactions in different groups
try to acquire a nexus lock on a row, the lock functions as an
enforcer: unless both transactions are reading the row, one
of them will have to wait until the other releases the lock.
If the transactions belong to the same group, however, the
nexus lock imposes no such constraints: both transactions
can acquire the nexus lock simultaneously.

Forcing transactions to acquire nexus locks on the rows
they access prevents Aborted Reads and Intermediate Reads
from occurring across groups. If two transactions from differ-
et groups access the same row and one of them is a write,
only the first will acquire the row’s nexus lock, while the
other will not be able to acquire the lock until the earlier
transaction completes. It is thus impossible for the later trans-
action to read aborted or intermediate states from the earlier
one.

Simply acquiring nexus locks, however, is not sufficient
to prevent Circularities. Consider the first example of Figure 2:
it focuses on write dependencies, since write dependency
cycles are forbidden by all ANSI isolation levels. Assume, in
the spirit of MCC, that the concurrency control mechanism
of Group 1 guarantees no dependency cycles between
T1 and T2. Although nexus locks prevent dependency cycles
between T1 and T3 (and similarly between T2 and T3), a dependency
cycle spanning T1, T2, and T3 can still form.

We extend the enforcement power of nexus locks by refining
the way in which traditional locking prevents Circularities.
With traditional locking (Example 2 of Figure 2), the “de-
pend” relation between transactions is tied to the “com-
pletes before” relation: if T2 depends on T1, then T2 must wait
for T1 to release its lock at the end of its execution, ensuring
that T2 will not start until T1 completes. Since “completes
before”, unlike “depends on”, is inherently acyclic, by tying
the two relations traditional locking guarantees that “depends
on” will be acyclic too.

If we now go back to the first example of Figure 2, what
got wrong there is clear: Circularities can arise because nexus
locks tie “depends on” to “completes before” only for trans-
actions that belong to different groups. Although T2 depends
on T1, since they are both in the same group T2 is allowed to
start before T1 completes. Were nexus locks to do otherwise,
however, and delay T2, they would curb concurrency within
Group 1.

To solve this puzzle, Callas refines the condition used
by traditional locking to avoid circularity. Rather than tying
“depends on” to “completes before”, Callas binds it to the
weaker (and yet provably sufficient [34]) “released before” to release
and enforces the following rule:

**Nexus Lock Release Order** If transaction T2 depends on
transaction T1, and they are from the same group, then T2
cannot release its nexus locks until T1 does.

The third example of Figure 2 illustrates how this rule,
which is implied by the stronger “completes before”, prevents
dependency cycles without hampering concurrency.

We prove its correctness as following:

**Definition 1.** We use Ti → Tj to denote Tj depends on Ti.

**Lemma 1.** If Ti → T2 → ... → Tm, then Tm cannot release its
nexus lock until Ti does.

**Proof.** First, we prove that for any Ti → Tj, Tj cannot re-
lease its nexus lock until Ti does. Case 1: Ti and Tj are from
the same group. In this case, Nexus Lock Release Order en-
forces that Tj cannot release its nexus lock until Ti does. Case
2: Ti and Tj are from different groups. In this case, since Tj
cannot acquire the nexus lock until Ti releases it, of course
Tj cannot release the lock until Ti does. Then one can easily
prove Lemma 1 by induction.

**Theorem 1.** Nexus locks prevent dependency cycles spanning
multiple groups.
Proof. We prove by contradiction. Assume there exists a dependency cycle spanning multiple groups \( T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \). Since it spans multiple groups, there must exist \( T_{j-1} \) and \( T_j \) such that \( T_{j-1} \rightarrow T_j \) and they are from different groups. This means \( T_{j-1} \) must release the nexus lock before \( T_j \) can acquire it to finish execution (Fact 1: \( t_{release}(T_{j-1}) < t_{finish}(T_j) \)). On the other hand, the assumption that \( T_j \rightarrow T_{j+1} \rightarrow \ldots T_{j-1} \) means \( T_{j-1} \) cannot release its its nexus lock until \( T_j \) does (Lemma 1), and \( T_j \) only releases its nexus locks after it finishes (Fact 2: \( t_{release}(T_{j-1}) > t_{finish}(T_j) \)). Facts 1 and 2 contradict with each other and thus it is impossible to form a dependency cycle spanning multiple groups.

To ensure that every transaction’s nexus locks are eventually released, Callas makes the simple choice of maintaining a FIFO queue for each nexus lock. Note that, thanks to MCC, the release of nexus locks is completely decoupled from the act of committing the transaction that holds those locks, which Callas leaves to the concurrency control mechanism of the group to which the transaction belongs. As soon as a transaction commits, any resource that the transaction held to control concurrency within its group can be released, even as the transaction may hold onto its nexus locks in order to release them in the correct order.

Nexus locks, ACID locks, and latency An unobtrusive inter-group mechanism is essential to achieving the potential for greater performance of MCC. A key feature of nexus locks is that any latency overhead they introduce, when compared with ACID locks, is due solely to their implementation, and not inherent to their semantics. Indeed, ignoring implementation overheads, if \( T_2 \) wants to acquire and then release a nexus lock held by \( T_1 \), it can always do so no later than if the lock had been ACID. The reason is simple: if \( T_1 \) and \( T_2 \) are from different groups, then a nexus lock behaves exactly like an ACID lock; if \( T_1 \) and \( T_2 \) are from the same group, then \( T_2 \) is always allowed to acquire a nexus lock while \( T_1 \) still holds it, while instead access to ACID locks is exclusive unless both \( T_1 \) and \( T_2 \) seek a read lock. The Nexus Lock Release Order rule can delay the release of \( T_2 \)'s locks if \( T_2 \) depends on \( T_1 \), but never more than if the locks had been ACID—in which case, \( T_2 \) would not even be allowed to acquire the locks until \( T_1 \) released them.

Of course, implementation overheads cannot in practice be ignored. However, we find them to be low in most cases (§7.4), in particular when compared with the substantial performance gains nexus locks enable by making it possible to safely deploy the kind of aggressive in-group concurrency control mechanisms we are discussing next.

5 Enforcing isolation within groups

While the inter-group mechanism’s main goal is to do no harm, the key to unlocking the performance potential of MCC is in the group-specific concurrency control mechanisms that it enables. Fulfilling that potential involves two steps: grouping transactions appropriately, and identifying mechanisms that can yield greater concurrency within each group, while maintaining safety.

The first of these steps appears hard to complete, as the number of possible groupings to consider is exponential. In practice, our experience building Callas is significantly more encouraging. As we already pointed out, the transactions that shape the performance of an application tend to be few [33] and we found that even just one or two simple specialized mechanisms can produce significant performance gains (§7): with such small numbers, systematically exploring all interesting groupings becomes a tractable problem (§6).

The additional concurrency called for by the second step demands transactions to expose more intermediate states. This could be done, for instance, by weakening their isolation properties [33], but to do so within a group would violate our requirement to offer all transactions the same ACID abstraction.

Transaction chopping (and its limitations) An attractive alternative is to turn, as several recent systems have done [27, 35], to an elegant theory that increases concurrency by chopping transactions—but in a way guaranteed to maintain serializability [29].

To prevent Aborted Reads and guarantee Atomicity, the theory requires transactions to be rollback-safe, meaning that any rollback statement must lie in the first subtransaction produced by a valid chopping. Since for serializability the absence of Circularity implies no Intermediate Reads, the theory focuses on preventing the former. It uses static analysis to construct an SC-graph, whose vertices are candidate transaction pieces, and whose edges, which are undirected, are of two kinds: S-edges connect the pieces within a transaction; C-edges connect pieces of different transactions that access the same object, when at least one of the accesses is a write. The theory shows that if a candidate chopping gives rise to an SC-cycle, then Circularity might arise during an execution. Hence, a candidate chopping of a set of transactions is considered safe (i.e., guarantees serializability) if (i) it is rollback-safe and (ii) it contains no SC-cycles.

Unfortunately, in practice these two conditions tend to produce choppings too conservative to result in much additional concurrency. To satisfy rollback safety, the first piece of each transaction must be large enough to include all rollback statements, limiting the opportunity for new interleavings.4

Relying on SC-cycles for safety has even more significant performance implications. Applications typically contain so many dependency cycles among their transactions that the only safe choppings, if any, are very coarse. One might expect

\[^4\text{This problem could be solved by asking application developers to rewrite their transactions to explicitly account for rollbacks at the application level. Our goal, however, is to achieve high performance with no additional programming effort.}\]
grouping to help here, since it restricts the requirement of being free of SC-cycles only to the transactions within each group—and it does (§7), but only to a limited extent. We find that SC-cycles tend to arise quite commonly among the very performance-critical transactions that, if they could be more finely chopped, would most benefit the application’s performance. An extreme but quite common case of this phenomenon occurs when a performance-critical transaction cannot be aggressively chopped because multiple instances of it may conflict with each other if executing concurrently.

Consider, for example, the new_order transaction in TPC-C. In first approximation, it roughly follows the access pattern of Figure 3(a): first, it inserts rows into the order table, then it inserts rows into the item table, and finally it updates the order_line table. As Figure 3(a) shows, SC-cycle analysis would conclude that it is not possible to split this transaction into subtransactions, as any two instances of the new_order transaction have three dependency edges (C-edges) between them.

Enter MCC These limitations motivate us to explore how to leverage the modularity of MCC to move beyond the opportunities for concurrency offered by the current theory of safe transaction chopping. To that end, Callas introduces Runtime Pipelining, a new in-group mechanism whose aggressive approach to concurrency control proves particularly effective within small groups. Runtime Pipelining relies on two new techniques: it leverages at execution time a refinement of the static analysis approach used by traditional transaction chopping to allow concurrency when SC-cycles would prevent it; and it prevents Aborted Reads and guarantees atomicity while avoiding, whenever possible, the performance downsides of enforcing rollback safety.

Similar to transaction chains [35], Runtime Pipelining assumes that the tables (though not necessarily the rows) accessed by each transaction are known prior to execution. The scope of this assumption, however, is much weaker than in transaction chains, since it applies only to the transactions in the target group. In practice, this assumption needs only to hold for the few transactions that, being performance critical, can most take advantage of a more aggressive in-group concurrency control mechanism.

5.1 Runtime Pipelining

Shasha et al. prove [29] that their static analysis technique produces the finest transaction chopping guaranteed to be safe: any more refined chopping has the potential to create Circularity and violate serializability. This sobering fact, however, does not imply that renouncing any further concurrency need be the price of safety. The key insight behind Runtime Pipelining is that, rather than preemptively inhibiting the possibility of Circularity, it may be feasible in some circumstances to allow for that possibility, relying instead on run-time techniques to prevent it from becoming an actuality.

Figure 3(a) illustrates the opportunity that Runtime Pipelining targets. Note how, as long as one can ensure that, during the execution, the top transaction accesses each table before the bottom one does, all C-edges acquire the same direction: all cycles are broken, and the transactions can be safely chopped in three pieces (Figure 3(b)). This finer chopping enables a form of pipelining: while the top transaction accesses the item table, the bottom one can concurrently access the order table and so forth.

The example suggests a way forward to safely extract greater concurrency from transaction chopping: rather than eliminating all SC-cycles, allow, intuitively, those where C-edges do not cross, since they can be neutralized at run time by controlling the order of execution of conflicting transaction pieces. To carry out this plan, we use a combination of static analysis and run-time mechanisms.

A new static analysis algorithm What prevents C-edges to cross and makes it safe to chop more aggressively in the example of Figure 3(b) is that both transactions access read-write tables in the same order. Generalizing from that example, assume that there exists a total ranking of each of the read-write tables accessed by the transactions in a group. Then, the goal of our new static analysis algorithm is to produce choppings that satisfy the following two golden rules.

GR1: Operations within a transaction piece are only allowed to access read-write tables of the same rank (read-only tables, which by definition have no rank, can be also accessed).

GR2: For any pair of pieces $p_1$ and $p_2$ of a given transaction that access read-write tables, if $p_1$ is executed before $p_2$, then $p_1$ must access tables of smaller rank than $p_2$ (as in GR1, read-only tables can be also accessed).

Then, by construction, the only C-edges that remain are those that connect pieces of different transactions that access tables of the same rank.

The two-step algorithm that achieves this goal and an example that illustrates its unfolding are shown in Figure 4. The first step totally ranks the read-write tables accessed by any of the transactions, and, within each transaction, groups together in a single piece all operations that access tables of the same rank. At the end of this step, the relative order of execution

---

4If a table is read-only, accessing it does not create C-edges
of the operations that, in each transaction, access read-write tables, is set. The second step then determines the execution order of the read-only operations of each transaction.

**Step 1: Ranking read-write tables.** Under the aggressive assumption that all operations in a transaction can be safely reordered, there is no constraint on the rank of read-write tables, and finding the finest chopping that does not have crossing C-edges is simple: we can assign a unique rank to each read-write table, sort operations in each transaction according to their original order in the transaction that access the same table. One can easily prove this chopping satisfies our two golden rules.

In practice, however, there often exist data or control dependencies that compels the ordering of operations within a transaction and constrain the ranking of tables. In Figure 4, for example, R(A) must happen before W(B) in T₁, which forces \( \text{rank}(A) \leq \text{rank}(B) \) (GR2), while for T₂, R(B) must happen before W(A), implying \( \text{rank}(B) \leq \text{rank}(A) \). This means we must assign the same rank to tables A and B and merge all operations that touch them into a single piece.

Concretely, we achieve this result with the help of a **table-dependency graph**. The graph’s nodes are read-write tables: we consider operations that access read-only tables in the next step. To add edges to the graph, we proceed as follows. For every transaction in the group, if there exists a data or control dependency between two operations \( \text{op}_1 \) and \( \text{op}_2 \) of the transaction, we add a directed edge between the tables they access (Figure 4, Step 1.a), indicating \( \text{rank}(	ext{Table}_{\text{op}_1}) \leq \text{rank}(	ext{Table}_{\text{op}_2}) \); we then assign the same rank to all the tables in the same strongly connected component, and assign ranks to all read-write tables according to their topological order in the resulting graph (Figure 4, Step 1.b). Within each transaction, operations that access tables with the same rank are merged into a single piece (Figure 4, Step 1.c).

**Step 2: Ordering operations that access read-only tables.** Transactions that access read-only tables contain operations that have not yet been ordered. To do so, we create a new graph for each transaction \( T \), adding a vertex for each of the ranked pieces of \( T \) produced by Step 1 (such vertices acquire the rank of their corresponding piece) and a vertex, with no assigned rank, for each operation of \( T \) that accesses
read-only tables. To encode the outcome of Step 1, we create a path that connects ranked vertices, from the least to the highest ranked; in addition, we add a directed edge between two vertices if there exists a data or control flow dependency between them (Figure 4, Step 2.a). Next, we proceed as we did in Step 1.b of Figure 4: we evolve the graph so that each strongly connected component is represented as a single new vertex, joining in parallel the corresponding transaction pieces, and topologically sort the resulting graph to obtain the definitive order of execution of the pieces that comprise each transaction. Within each piece, operations are executed in the order in which they appeared in their transaction, prior to its chopping (Figure 4, Step 2.b).

Next we formally prove that the previous static analysis algorithm can generate a valid chopping that achieves our two golden rules.

**Lemma 2.** Step 2 does not change the chopping generated in Step 1. Formally, for \( op_1 \) and \( op_2 \) that access read-write tables, 1) if \( op_1 \) and \( op_2 \) are placed in a single piece in Step 1, they will also be placed in a single piece in Step 2; 2) if \( op_1 \) and \( op_2 \) are placed in different pieces in Step 1, they will also be placed in different pieces in Step 2; 3) if \( op_1 \) is placed before \( op_2 \) in Step 1, \( op_1 \) will also be placed before \( op_2 \) in Step 2.

**Proof.** 1) is true because Step 2 never breaks any pieces created by Step 1. 3) is true because in Step 2.a, when the algorithm constructs the graph, it adds a path from lowest to highest ranked piece, ensuring that the order calculated by Step 1 is respected in Step 2.

We prove 2) by contradiction. Assume in Step 1, \( op_1 \) is placed in piece \( p_1 \), \( op_2 \) is placed in piece \( p_2 \), and \( p_1 \) is placed before \( p_2 \); in Step 2, \( op_1 \) and \( op_2 \) are placed in the same piece. This means in the graph generated by Step 2.a, \( p_1 \) and \( p_2 \) are in the same strongly connected component, and thus there must be a path from \( p_2 \) to \( p_1 \). This means some operation \( (op_1') \) in \( p_1 \) must depend on some operation \( (op_2') \) in \( p_2 \). In this case, in the graph generated by Step 1.a, there must be an edge from the table accessed by \( op_1' \) to the table accessed by \( op_2' \) and thus it is impossible for Step 1.b to place \( p_1 \) before \( p_2 \) and this contradicts our assumption.

**Lemma 3.** The result chopping is valid. Formally, for any two operations \( op_1 \) and \( op_2 \) in a transaction, if there is a data or control dependency from \( op_1 \) to \( op_2 \), then in the result chopping, \( op_1 \) appears before \( op_2 \).

**Proof.** First note that since there is a dependency from \( op_1 \) to \( op_2 \), \( op_1 \) must appear before \( op_2 \) in the original transaction. Then we prove by cases:

Case 1: both \( op_1 \) and \( op_2 \) access read-write tables. In this case, Step 1.a places a directed edge from the table \( T_1 \) accessed by \( op_1 \) to the table \( T_2 \) accessed by \( op_2 \) and orders them based on the topological order of the graph. There are two possible subcases:

Case 1.1. \( T_1 \) and \( T_2 \) are not in the same strongly connected component. Step 1.b gives \( T_1 \) a lower rank than \( T_2 \) in this case and thus Step 1.c places \( op_1 \) before \( op_2 \). Step 2 will not change this result (Lemma 2) and thus finally \( op_1 \) will be placed before \( op_2 \).

Case 1.2. \( T_1 \) and \( T_2 \) are in the same strongly connected component. Step 1.b gives the same rank to \( T_1 \) and \( T_2 \) in this case and thus Step 1.c places \( op_1 \) and \( op_2 \) in the same piece. Step 2 will not change this result (Lemma 2) and Step 2.b orders \( op_1 \) and \( op_2 \) according to their original order in the transaction and thus places \( op_1 \) before \( op_2 \).

Case 2: at least one of \( op_1 \) and \( op_2 \) accesses a read-only table. In this case, Step 2.a places a directed edge from \( op_1 \) to \( op_2 \) and orders them based on the topological order of the graph. There are also two possible subcases:

Case 2.1: if \( op_1 \) and \( op_2 \) are not in the same strongly connected component, Step 2.a places \( op_1 \) before \( op_2 \).

Case 2.2: if \( op_1 \) and \( op_2 \) are in the same strongly connected component, Step 2.a places them in the same piece and Step 2.b orders them according to their original order in the transaction and thus places \( op_1 \) before \( op_2 \).

**Lemma 4.** The resulting chopping satisfies GR1: operations within a transaction piece are only allowed to access read-write tables of the same rank.

**Proof.** By construction, Step 1.c merges operations that access tables with the same rank into a piece. Therefore, the property holds at the end of Step 1. Step 2 does not change the chopping for operations that access read-write tables (Lemma 2). Therefore, the property still holds.

**Lemma 5.** The resulting chopping satisfies GR2: for any pair of pieces \( p_1 \) and \( p_2 \) of a given transaction that access read-write tables, if \( p_1 \) is executed before \( p_2 \), then \( p_1 \) must access tables of smaller rank than \( p_2 \).

**Proof.** By construction, Step 1.c sorts operations based on the rankings of tables they access. Therefore, the property holds at the end of Step 1. Step 2 does not change the chopping for operations that access read-write tables (Lemma 2). Therefore, the property still holds.

**Theorem 2.** The static analysis algorithm generates a valid chopping that satisfies GR1 and GR2.

**Proof.** Combining Lemmas 3, 4, and 5, we can prove this theorem.

**Enforcing safety at run time** Once static analysis produces choppings that satisfy our golden rules, neutralizing the remaining SC-cycles at run time is easy. Consider a piece of transaction \( T \) that accesses a table that involves a C-edge. If
in so doing \( T_i \) becomes (anti-)dependent on some uncommitted transaction \( T_j \) that has already accessed that table, then that C-edge and every subsequent C-edge between \( T_i \) and \( T_j \) become (logically) directed: thenceforth, \( T_i \) cannot commit until \( T_j \) does, and every piece of \( T_i \) that accesses tables with ranking \( r \) must wait until \( T_j \) either has executed a piece that accesses tables with ranking at least \( r \), or commits.

In practice, Runtime Pipelining is even more aggressive in pursuing opportunities for concurrency. It only declares a dependency between \( T_i \) and \( T_j \) if they access the same row at run time (this is easy for \( T_i \) to verify in Callas by checking if \( T_j \) has acquired a nexus lock on the row). If not, Runtime Pipelining imposes no restrictions on execution ordering.

Although our discussion has focused on enforcing serializability, Runtime Pipelining can be easily applied to other notions of isolation by simply weakening the conditions under which it declares a dependency. For example, were Runtime Pipelining tuned to enforce read committed isolation, anti-dependencies would not trigger ordered execution.

We prove its correctness as following:

**Theorem 3.** *Runtime Pipelining prevents Intermediate Reads.*

**Proof.** If a transaction writes to the same table more than once, static analysis algorithm puts those operations into a single piece, thus it is impossible for another piece to observe states made by the earlier writes.

**Theorem 4.** *Runtime Pipelining prevents dependency cycles.*

**Proof.** We prove by contradiction. Suppose there exists a dependency cycle \( T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \).

By construction, only pieces that access tables with the same rank can generate a dependency edge. Suppose \( r \) is the maximum rank of tables that are involved in any dependency edges in the cycle. We define \( p_i \) as the first piece of \( T_i \) that accesses a table with a rank not smaller than \( r \), or the commit operation if such a piece does not exist. We define \( t_i \) as the completion time of \( p_i \).

First, we prove that any transaction \( T_k \) must have executed \( p_k \) when the cycle forms and \( t_{k-1} < t_k \) if \( T_{k-1} \rightarrow T_k \). Based on our assumption, there must exist at least one transaction \( T_i \) that has already executed a piece that accesses a table with rank \( r \). Then let’s consider \( T_{i-1} \). There are two possible cases:

**Case 1:** The dependency edge \( T_{i-1} \rightarrow T_i \) occurs on a table with rank smaller than \( r \). In this case, \( T_i \) has already depended on \( T_{i-1} \) and thus Runtime Pipelining ensures that \( T_i \) cannot execute \( p_i \) until \( T_{i-1} \) completes \( p_{i-1} \). Therefore, \( T_{i-1} \) must have executed \( p_{i-1} \) and \( t_{i-1} < t_i \).

**Case 2:** The dependency edge \( T_{i-1} \rightarrow T_i \) occurs on a table with rank \( r \). In this case, \( p_{i-1} \) and \( p_i \) must both access tables with rank \( r \). Since the dependency edge already occurs, \( T_{i-1} \) must have already finished \( p_{i-1} \) before \( T_i \) executes \( p_i \). Therefore, \( T_{i-1} \) must have executed \( p_{i-1} \) and \( t_{i-1} < t_i \).

By induction, any transaction \( T_k \) must have executed \( p_k \) and \( t_{k-1} < t_k \).

Then by induction, we can get \( t_1 < t_2 < \ldots < t_n < t_1 \), which is impossible. Therefore, our assumption—a dependency cycle can form—is wrong.

**Beyond rollback safety** Runtime Pipelining takes an equally aggressive approach when it comes to avoiding the Aborted Reads and Atomicity violations that chopping introduces. Rather than settling for either the loss of concurrency or programming effort that rollback safety may cause, Runtime Pipelining adopts an optimistic approach: it allows a transaction \( T_1 \) to read uncommitted states from \( T_2 \), but it does not allow \( T_1 \) to commit until \( T_2 \) commits. If \( T_2 \) is rolled back, then \( T_1 \) must also roll back.

While optimism pays off in finer chopping, no programming effort, and greater performance in groups when aborts and rollbacks are rare, it raises the possibility of performance loss in the presence of cascading rollbacks.

To avoid this danger, Runtime Pipelining takes two steps. First, it leverages MCC to prevent rollbacks from propagating outside of a group. Thus, misplaced optimism only affects performance in groups that are guilty of it. Second, it dynamically responds to an unexpected incidence of rollbacks by becoming increasingly more conservative. When the rollback rate crosses a threshold, Runtime Pipelining goes temporarily back to enforcing rollback safety; since we expect high rollback rates to be infrequent, however, it periodically tries to revert to its original optimistic approach.

The option of enforcing rollback safety on demand allows Runtime Pipelining to enjoy the full benefits of optimism when the rollback rate is reasonably low and avoid long-term damage from misplaced optimism.

Further, to ensure liveness in the face of rollbacks, Runtime Pipelining limits the depth of dependency chains composed of uncommitted transactions, and prevents a transaction that has been rolled back from performing uncommitted reads on retry.

## 6 Implementation

The current prototype of Callas is built upon the MySQL Cluster distributed database [6]. To implement Runtime Pipelining, we detect conflicts at the MySQL Cluster locking module and notify the transaction coordination module to enforce ordering between subtransactions, when necessary; to ensure isolation across groups, we modify the locking module of MySQL Cluster to support nexus locks and enforce their release order. Relying on MySQL Cluster, however, means that the current prototype of Callas must use the *read-committed* isolation level, the only one that MySQL Cluster supports.

To combine performance with simplicity, we developed tools that automate the process of grouping transactions and chopping them into subtransactions.
6.1 Automated chopping

The automated-chopping tool closely follows the Runtime Pipelining algorithm (§5) to statically analyze the transaction code and add markers to indicate the subtransaction boundaries to the runtime system, but introduces three additional optimizations: (i) it performs static analysis over columns rather than tables to produce finer choppings; (ii) it removes unnecessary C-edges; and (iii) it identifies better performing subtransaction orderings. We discuss the two latter optimizations in greater detail below.

Removing redundant C-edges Since commutative operations can be executed in any order without violating isolation, our tool, like Lynx [35], removes C-edges between them.

Additionally, it searches for instances of runtime uniqueness, where multiple transaction instances modify the same table, but each is guaranteed to operate on a different row. For example, in TPC-C, the new_order transaction acquires a unique order ID by incrementing a nextOrderID object, and then proceeds to modify the corresponding row. Such opportunities are identifiable by searching for “monotonic” objects, i.e., objects, such as counters, that all transactions modify monotonically before using them as a key in a query. Runtime uniqueness is yet another example of an optimization whose effectiveness can be magnified by the modularity of MCC, since runtime uniqueness is less likely to hold in large groups of transactions. In TPC-C, for example, uniqueness does not hold globally, as other transactions (e.g., delivery) do not use nextOrderID and may therefore access the same row as new_order.

Identifying more performant orderings Given a transaction T, any topological order of the pieces of T produced by Step 2.b of the algorithm in Figure 4 yields a safe way to execute T. We then have some freedom in choosing the order in which T’s pieces should execute. We leverage this freedom by having the larger pieces—classified heuristically by the number of queries they contain—execute as early as possible. The rationale behind this optimization is that Runtime Pipelining only enforces ordering between transactions once a dependency manifests at run time. By executing large subtransactions early, we decrease the chance that they will be subject to ordering, thus increasing parallelism.

6.2 Automated grouping

The goal of our grouping tool is to identify groups of transactions that contend heavily with each other. The user need only provide her workload of choice; the tool analyzes the performance of the workload using various groupings and returns the grouping that yields the best performance. Our current tool does not explore all possible groupings, but rather uses heuristics to identify groupings that are more likely to increase concurrency. Our evaluation suggests that this heuristic approach is enough to provide significant performance benefits (e.g., 8.2x speedup for TPC-C).

The tool works in iterations. In each iteration it runs the workload and creates a profile for this iteration’s performance measurements. Based on these measurements, it tries to identify the most prominent source of contention and suggests a grouping that could alleviate it. It then runs our chopping tool on this grouping, and proceeds to measure the performance of this new configuration in the next iteration. This process terminates if an iteration does not yield any performance improvement.

To identify sources of contention, we use as a hint the latency of individual operations. As the load on the system grows, the latency of highly contending operations tends to increase disproportionately. The corresponding transactions are then our primary candidates for optimization. If there are only few such transactions, our tool enumerates all possible groupings; otherwise, it focuses on those that hold locks on contended items for long intervals.

7 Evaluation

The goal of Callas is to provide unmodified database applications with the level of performance that was previously only achievable by manually modifying all or part of the application code. To assess whether Callas achieves this goal, we evaluate the performance of Callas using various applications and workloads. In particular, our evaluation answers the following questions:

- What is the performance gain of Callas over a traditional ACID database? (§7.1)
- How does the performance of Callas compare against that
of other approaches that aim to improve database throughput? (§7.1)

- How do various optimizations, groupings, and workload parameters affect the performance of Callas? (§7.2, §7.3, §7.5)
- What is the overhead of nexus locks? (§7.4)
- As the rate of rollbacks changes, how effective is it to optimistically renounce rollback safety to extract performance? (§7.6)

We answer these questions by measuring the performance of Callas using microbenchmarks and three applications: TPC-C [19], Fusion Ticket [3], and Front Accounting [2]. TPC-C is a database benchmark that models online transaction processing. It contains three highly-contending read-write transactions and two read-only transactions.

Fusion Ticket is an open source software solution for online ticketing and advanced sales. To perform a fair comparison with Salt [33], we run the same workload used in that paper, which includes several transactions critical to the performance and functionality of an online shop.

Front Accounting is an open source accounting and Enterprise Resource Planning (ERP) program. It allows a company to manage its sales, purchases, and stock levels. Our workload includes 17 transactions that simulate the workload of a retail company: the company purchases goods from suppliers at a low price and sells them to customers at a higher price. It includes five read-write transactions: create-order, payment, delivery, pay-supplier, and stock-adjustment, and 12 read-only transactions to query order information.

**Experimental setup** In TPC-C, we populate ten warehouses, and assign each warehouse to a separate partition. Our Fusion Ticket setup mirrors that of Salt: there is one event and two categories of tickets, with 10,000 seats in each category. Finally, in Front Accounting, we configure the retail company to operate on 100 different types of goods, and on average to make a bulk purchase for every 1,000 sale orders.

For all experiments, we use ten database partitions, each of which is three-way replicated. All our throughput numbers were calculated while the system is saturated.

Our experiments are carried out on Dell PowerEdge R320 machines in CloudLab [1]. Each machine is equipped with a Xeon E5-2450 processor, 16 GB of memory, four 7200 RPM SATA disks, and 1 Gb Ethernet.

### 7.1 Callas’ performance

Our first set of experiments uses TPC-C, Fusion Ticket, and Front Accounting to compare the throughput of Callas to that of MySQL Cluster—the system from which Callas descends.

As shown in Figure 5, the performance of Callas on the TPC-C benchmark is about 8.2x higher than that of the original MySQL Cluster.

Callas’ performance improvement is partly due to the ability of the automated grouping tool to identify highly contending transactions and group them accordingly. In this case, the tool placed the new_order and payment transactions in one group; the delivery transaction in a second group; and the remaining transactions in a third group. This grouping reflects the contention pattern of these transactions: new_order and payment contend heavily for the warehouse and district tables, but Runtime Pipelining is effective in allowing them to release their locks early, after acquiring a unique ID. Moreover, although both new_order and payment update the district table, they update different columns, which allows our static analysis to remove the C-edge between them. This makes them ideal candidates for belonging to the same group: they contend for the same row lock, but do not have any C-edges between them, and therefore can be grouped together without introducing any SC-cycles. Callas uses Runtime Pipelining to chop transactions in the first two groups, while the third group is left unoptimized, as it does not contain performance-critical transactions.

Figure 6 shows the performance of Callas and MySQL Cluster for the Fusion Ticket application. Callas outperforms MySQL Cluster by a factor of 5.7x. For this application, our tool generates two groups: the first contains the create-order and payment transactions, and the second contains the remaining transactions and is left unoptimized.

As shown in Figure 7, when running the Front Accounting application, Callas outperforms MySQL Cluster by a factor of 6.7x. For this application, our tool generated four groups:

<table>
<thead>
<tr>
<th></th>
<th>Latency(ms)</th>
<th>MySQL</th>
<th>Callas</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quantile</td>
<td>50th</td>
<td>99th</td>
</tr>
<tr>
<td>new_order (TPC-C)</td>
<td>26</td>
<td>51</td>
<td>28</td>
</tr>
<tr>
<td>checkout (FT)</td>
<td>12</td>
<td>25.3</td>
<td>12</td>
</tr>
<tr>
<td>delivery (FA)</td>
<td>36.3</td>
<td>69</td>
<td>36.6</td>
</tr>
</tbody>
</table>

Table 1: Latency under low throughput.

![Fig. 8: Effect of different techniques](image_url)
transactions $create\text{-}order$, $delivery$, and $payment$ are each placed in their own groups and use Runtime Pipelining for chopping, while the rest of the transactions are placed in a fourth, unoptimized, group.

**Comparison with other techniques** The next experiment compares the performance of Callas for TPC-C and Fusion Ticket with that of other techniques. We first consider applying the SC-cycle static analysis of traditional transaction chopping to the entire application (i.e., without grouping). This step boosts the performance of TPC-C to 3.5x of the MySQL baseline, but does not help Fusion Ticket at all, since traditional transaction chopping cannot safely chop the performance-critical transaction of Fusion Ticket. Next, we combine traditional transaction chopping with our grouping mechanism: we split transactions into groups and use SC-cycle analysis to chop transactions within each group. This approach further improves the throughput of TPC-C by 35%, and raises the throughput of Fusion Ticket to 2.6x of the baseline. Callas, using Runtime Pipelining instead of standard SC-cycle analysis, achieves a further 74% and 120% throughput boost, respectively. Remarkably, the performance of Callas is within 5% of that of Salt [33]. We find it encouraging that, despite staying true to the ACID paradigm, Callas can achieve performance similar to approaches that require manual modification of the application code.

**Latency** Table 1 presents the request latency for our three applications, when the system is under low load. In all cases, the latency of Callas is similar to that of the unmodified MySQL Cluster.

### 7.2 Performance impact of various optimizations

Figure 9 breaks down the contribution of each optimization to the performance of Callas (using the grouping produced by our heuristic algorithm) for TPC-C and Fusion Ticket.

The effectiveness of the different optimizations is application-dependent. In Fusion Ticket, Runtime Pipelining alone, even naively applied at the granularity of tables, is enough to achieve almost all of Callas’ performance improvement. Not so in TPC-C, where Callas gets a significant performance boost from performing static analysis at the column—rather than the table—level by identifying several columns that are accessed in read-only mode (even as the table they belong to is accessed in read-write mode). This allows Callas to remove conflict edges between transactions and achieve finer-grained chopping. Leveraging commutativity yields only a minor performance improvement in TPC-C, because it only applies to a few individual statements. Runtime uniqueness instead provides another big boost in the performance of TPC-C by removing several critical conflict edges in the $new\_order$ transaction, leading to finer-grained chopping.

Note that, although it does not explicitly appear in Figure 9, MCC is essential to Callas’ performance gains, because many of its optimizations would simply not be applicable without MCC. Runtime uniqueness, for example, could not be leveraged in TPC-C, since it does not hold for all TPC-C’s transactions; and Runtime Pipelining itself would prove virtually ineffective if applied across the *entire* set of Fusion Ticket’s often-complex transactions.

### 7.3 Performance impact of different groupings

To demonstrate the importance of grouping transactions appropriately, we measure Callas’ throughput when running TPC-C using different transaction groupings. TPC-C has five transactions: three are read-write transactions and two are read-only. We first compare our heuristic grouping to two naive groupings. The first puts all transactions in a single group, while the second puts each of the five transactions in a separate group. Our heuristic grouping—the result of running the heuristic algorithm of Section 6.2—consists of three groups: $new\_order$ and $payment$ are in one group, $delivery$ is in a second, and the two read-only transactions in a third.

---

6These were the applications used to evaluate Salt [33].
Throughput (txns/sec)

100000
10000
100000
10000
10000
1000
1000
100
0.0001
0.001
0.01
0.1
1

Execution frequency ratio of T1

MySQL Cluster
Callas

Fig. 11: Effect of execution frequency on performance.

Throughput (txn/sec)

100000
10000
10000
100000
0.0001
0.001
0.01
0.1
1

Contention probability across groups (1/N)

MySQL Cluster
Callas

Fig. 12: Overhead of nexus locks

7.4 Overhead of nexus locks

Two factors contribute to the overhead of nexus locks: the cost of maintaining an additional lock and that of correctly enforcing the Nexus Lock Release Order rule (§4). The latter cost is only incurred when transactions conflict, while the former is always present. To measure separately their effect on throughput, we designed two microbenchmarks, one with no contention and one with high contention.

Fig. 13: Effect of contention probability across groups.

To separate sufficiently the performance of Callas and that of MySQL Cluster, in order for us to study the effects of inter-group contention on the former, both our microbenchmarks use a sequence of five non-conflicting operations.

As shown in Figure 12, in the no-contention experiment, MySQL Cluster outperforms Callas by about 19%. Our profiling shows the bottleneck lies in the additional demands on the CPU to acquire, maintain, and release nexus locks. In the high-contention experiment, the throughput of MySQL Cluster is about 13.6% higher than that of Callas.

The CPU overhead of nexus locks is of course still there, but it becomes relatively less prominent because contention increases the execution time of transactions. Instead, the additional message exchanges Callas requires to enforce the Nexus Lock Release Order rule become the dominant factor.

7.5 Effect of contention rate on performance

The design of Callas focuses on optimizing in-group contention while being conservative about contention between transactions in different groups. While our experience with real applications suggests that it is typically possible to partition transactions, so that inter-group contention is minimized, we would like to understand how robust the performance of Callas is to increased levels of inter-group contention. We design two microbenchmarks, each exploring a different factor of inter-group contention: execution frequency and contention rate. Both microbenchmarks start by executing operations that cause conflicts (across groups or inside a group) and end with a sequence of operations that cause no conflicts. Unlike MySQL Cluster, Callas can chop contending and non-contending operations in separate pieces and release contending locks early: hence, the longer the sequence of non-conflicting operations at the end of a transaction, the greater the performance benefits that Callas can bring.

To get a sense of how the grouping produced by our algorithm compares to an optimal grouping, we iterated over all the possible grouping strategies: we found that, at least for TPC-C, no other grouping achieved a higher throughput.
The first microbenchmark explores the performance repercussions of having two frequently executing transactions in different groups. The microbenchmark includes two types of transactions, $T_1$ and $T_2$. Each transaction contains six (1 + 5) operations: the first operation updates one row, randomly chosen out of ten rows, thus fixing the contention rate between $T_1$ and $T_2$ at 10%. The remaining five operations update non-conflicting rows (i.e., rows that are private to each transaction instance). We place $T_1$ and $T_2$ in separate groups that use Runtime Pipelining for chopping, and we tune the relative execution frequency of these two transactions.

The second microbenchmark explores the effect of inter-group contention rate on the performance of Callas. This microbenchmark is similar to the first: we use two types of transactions, $T_1$ and $T_2$, each in its own group, only this time they have the same execution frequency. Each transaction contains seven (2 + 5) operations: the first operation updates one row at random, chosen out of $N$ rows, where $N$ is a parameter that controls the inter-group contention rate. The second operation modifies a random row, chosen out of ten rows, from a table private to each transaction type, thus introducing a 10% in-group contention rate. The remaining five operations update non-conflicting rows (private to each transaction instance).

Figures 11 and 13 show the results for these two microbenchmarks. Both experiments show the same trend: when the inter-group contention is low, the performance of Callas far exceeds that of a traditional ACID database. For example, when the execution frequency ratio of $T_1$ to $T_2$ is 100:1 (Figure 11), the throughput of Callas is 20.2x that of MySQL Cluster. Similarly, when $T_1$ and $T_2$ have a low conflict rate of 0.01% (Figure 13), the throughput of Callas is 16.6x that of MySQL.

As the inter-group contention rate increases — because both transactions run frequently or contend heavily — the benefit of Callas decreases. This is to be expected, as Callas is effectively attempting to regulate heavy contention using traditional locking.

Note, however, that even when the inter-group contention is as high as the in-group contention, the performance benefit of Callas is still substantial. In Figure 11, even when $T_1$ and $T_2$ are executed with the same frequency, Callas’ throughput is more than twice that of MySQL’s; in Figure 13, when $N=1$ (i.e., contention is at 100%), Callas achieves a 60% throughput gain. The reason behind this performance increase is that, even when the workload is uniform (e.g., both transactions have the same frequency), this does not mean that a $T_1$ always followed by a $T_2$ (and vice versa). As long as two $T_1$s (or $T_2$s) are executed consecutively, Runtime Pipelining can optimize their execution. Interestingly, increasing in-group concurrency implicitly increases inter-group concurrency as well, since transactions hold their locks for shorter times.

### 7.6 Beyond rollback safety

Our final set of experiments measure the performance of Runtime Pipelining’s adaptive approach for preventing Aborted Reads and Atomicity violations.

We design a microbenchmark that can trigger cascading rollbacks in a controlled manner. It uses three tables—$A$, $B$, and $C$—with ten rows each, and contains one transaction with 11 operations. The first operation picks a number $i$ at random between 1 and 10, and checks a condition on the $i^{th}$ row in table $A$ and the second operation updates that row if the check succeeds. The third and fourth operation do the same for the $i^{th}$ row of table $B$; and the fifth and sixth do the same for the $i^{th}$ row of table $C$. The last five operations update non-conflicting rows (private to that transaction instance).

Runtime Pipelining splits this transaction into the following eight subtransactions: $\langle 1, 2 \rangle \langle 3, 4 \rangle \langle 5, 6 \rangle \langle 7 \rangle \langle 8 \rangle \langle 9 \rangle \langle 10 \rangle \langle 11 \rangle$, the first subtransaction containing the first two operations, etc. The check of the fifth operation has a probability to fail and cause a rollback, triggering a cascading rollback if other transactions already depend on this transaction.

In our first experiment, we tune the probability of the third subtransaction triggering a rollback. As shown in Figure 14, the throughput of both Callas and MySQL Cluster decreases as the rollback rate increases. The throughput of Callas is always higher than that of MySQL Cluster, but the improvement decreases from 2.9x to 60%. When the rollback rate is
low, Callas can execute all eight subtransactions in a pipeline, but when the rollback probability increases, Callas’ adaptive control mechanism falls back to “safe mode” by merging the first three subtransactions—thus placing the rollback statement in the first subtransaction. Even in safe mode, however, Callas can still parallelize the execution of the last five subtransactions. In Figure 14, the switch to safe mode happens when the rollback rate is higher than 7%. For reference, we also measured the throughput of Callas with safe mode always on, and with safe mode always off. In the former case we lose parallelism when the rollback rate is low, whereas in the latter we incur significant overhead when the rollback rate is high. Thanks to Runtime Pipelining’s adaptive mechanism, Callas comes close to the best of both worlds.

In practice, since real applications have low rollback rates most of the time, we expect safe mode to be triggered only infrequently; the rest of the time Callas would still be aggressively optimizing transactions.

Figure 15 takes a closer look at the performance of Callas under stress. We fix the rollback rate to a high value (50%) and increase the load of the system until we reach saturation. We observe that under low load, it is not critical for Runtime Pipelining to adaptively fall back to safe mode; in fact, an always-aggressive version of Callas performs slightly better. As the load—and, hence, parallelism—increases, however, the adaptivity of Runtime Pipelining prevents cascading rollbacks from causing a performance collapse, while allowing Callas to continue leveraging some parallelism.

8 Related work

The ACID paradigm has been adopted by most academic and commercial databases [4–9, 28], mainly because of the simplicity of its abstraction. Frustrated by its performance, however, researchers have tried several approaches to improve the performance of ACID databases.

Optimizing certain transaction types Read-only transactions and partition-local transactions have been a popular target for optimization [18, 20, 30, 31]. For example, Spanner [18] can avoid the two-phase commit (2PC) protocol for read-only transactions, assuming clocks are well synchronized. Similarly, H-Store [31] avoids 2PC for partition-local transactions.

Optimizing under certain conditions Another approach is to optimize transactions when certain conditions hold for the workload. For example, Sagas [24] lets developers chop long-running transactions into pieces when such chopping does not affect the application semantics. Transaction chopping [29] and Lynx [35] use SC-cycles to identify transactions eligible for chopping (§5).

Like Callas, Lynx observes that executing transaction pieces in a well-defined order can avoid conflicts: its origin ordering technique ensures that if two transactions \( T_1 \) and \( T_2 \) start on the same server, and \( T_1 \) starts before \( T_2 \), then, to guarantee safety, \( T_1 \) pessimistically executes before \( T_2 \) at every server where they both execute. However, since it is hard in practice to anticipate the specific servers where user transactions will execute, origin ordering can only prevent conflicts among the predictable internal transactions used for updating secondary indexes and joint tables. In contrast, Callas’ Runtime Pipelining is widely applicable, since it relies on information (the order in which transactions access tables) that can be easily established through static analysis, and only enforces ordering if it detects an actual conflict at run time, leaving significantly greater opportunities for concurrency.

Rococo [27] relaxes Lynx’s eligibility condition by reordering transaction operations and applying additional runtime mechanisms. Calvin [32] avoids using 2PC by predetermining an execution schedule for transactions, but again under the assumption that the system can predict which server a transaction will access when it is executed.

In general, such optimizations have the potential to yield significant performance improvements, but the assumptions on which they rely are usually hard to satisfy in real applications. MCC can help increase the applicability of these techniques by requiring those assumptions to only hold within each group, rather than globally.

To reduce the overhead of determining whether two transactions may conflict at run time, SDD-1 [16] introduces the notion of transaction classes, which bear an intriguing but ultimately passing similarity to Callas’ transaction groups. In SDD-1, each transaction class is defined statically by the database administrator, and it is formally identified by a logical read and write set. A transaction \( T \) fits in any class whose read and write set are a superset of the corresponding sets for \( T \): the class to which a specific instance of \( T \) is actually assigned is not decided until run time. SDD-1 simplifies concurrency control by first using static analysis to identify conflicts within classes (rather than transactions), and by then leveraging the observation that transactions that are assigned at run time to different classes can conflict only if their classes conflict. In Callas, transaction groups are instead the key mechanism that enables the separation of concerns that is at the core of MCC. By delimiting the scope of each in-group concurrency control mechanism, they allow them to aggressively seek opportunities for greater concurrency.

Weakening the abstraction or reducing its scope A third line of work tries to increase performance by either weakening the ACID abstraction, or by providing its guarantees for a subset of transactions. For example, ElasTraS [21], MegaStore [13], G-Store [22], and Microsoft’s Cloud SQL Server [14] only provide ACID transactions within a single partition or key group. For requests that touch multiple partitions, these systems rely on the developers to ensure correctness, which is tedious and sometimes undesirable [18, 30]. Salt [33] alleviates this problem somewhat by requiring the
developers to optimize only a few performance-critical transactions and guaranteeing that the other transactions don’t see their intermediate states. Callas, instead, provides the ACID abstraction uniformly to all transactions, without requiring any effort from the developers.

9 Conclusions

Separating concerns and decoupling abstraction from mechanism are basic tenets of sound system design—and for good reasons. We confirm their benefits yet again, by applying them to the long-standing problem of improving the performance of ACID applications. Our initial experience is encouraging: the flexibility of the modular concurrency control architecture at the core of Callas allows the applications we have tested, to obtain, unmodified, the kind of performance previously achievable only by manually rewriting all or part of the applications’ code.

Acknowledgements

We had a dream shepherd in Petros Maniatis: demanding, able to see both the forest and the tiniest leaf, and always ready to help. This paper has benefitted from his prodding, encouragement, and concrete suggestions in countless ways. We are grateful to the anonymous reviewers for their insightful comments and to Natacha Crooks, Marco Serafini, Spyros Blanas, and Xiaodong Zhang for feedback on early drafts of this paper. This work would simply not have been possible without the patience and support of the amazing CloudLab team [1] throughout our experimental evaluation. This material is based in part upon work supported by a Google Faculty Research Award and by the National Science Foundation under Grant Number CNS-1409555.

References


