Warranties for Faster Strong Consistency

Jed Liu *Tom Magrino* *Owen Arden* *Michael D. George* *Andrew C. Myers*
Cornell University Department of Computer Science
{liujed,tmagrino,owen,mdgeorge,andru}@cs.cornell.edu

Abstract

We present a new mechanism, warranties, to enable building distributed systems with linearizable transactions. A warranty is a time-limited assertion about one or more distributed objects. These assertions generalize optimistic concurrency control, improving throughput because clients holding warranties need not communicate to verify the warranty's assertion. Updates that might cause an active warranty to become false are delayed until the warranty expires, trading write latency for read latency. For workloads biased toward reads, warranties improve scalability and system throughput. Warranties can be expressed using language-level computations, and they integrate harmoniously into the programming model as a form of memoization. Experiments with some nontrivial programs demonstrate that warranties enable high performance despite the simple programming model.

1 Introduction

Although the trend for many systems has been to weaken consistency in order to achieve greater scalability, strong consistency is critical when lives or money are at stake. Examples include systems for medical information, banking, payment processing, and the military.

Users of weakly consistent systems may be confused by applications that appear buggy. Moreover, weak consistency can significantly complicate the job of developers who try to detect and repair inconsistencies at the application layer. Consistency failures at the bottom of a software stack can percolate up through the stack and affect higher layers in unpredictable ways, requiring defensive programming.

The need for strong consistency and a simple programming model has kept databases with ACID transactions in business. However, transactions are often considered to have poor performance, especially in a distributed setting. In this work, we introduce *warranties*, a new mechanism that improves the performance of transactions, enabling them to scale better both with the number of application clients and with the number of persistent storage nodes. Warranties help avoid the unfortunate choice between consistency and performance.

A warranty is a limited guarantee that some (potentially complex) assertion remains true regarding the state of a distributed system. The guarantee is limited in that it eventually expires. But during the term of the warranty, the application can safely use it to perform computation locally without communicating with the server that issued the warranty. Warranties are like leases [21] in that they have a duration, but differ in that they make a logical assertion rather than conferring the right to use objects.

Warranties support implementing linearizable transactions by generalizing optimistic concurrency control (OCC) [16, 32]. OCC permits aggressive caching of objects read by transactions, but requires communicating with storage servers to ensure objects are up to date. Since warranties can express guarantees that objects are up to date, communication can be reduced. Warranties are particularly effective in the common case of high read contention, where many clients want to share the same popular—yet mutable—data.

More generally, warranties can contain an assertion that the results of a language-level computation has not changed. These *computation warranties* offer a form of distributed memoization, allowing clients to share cached computation in the manner often currently done using distributed caches such as `memcached`—but with strong consistency guarantees that are currently lacking.

Overall, warranties offer a new way to ameliorate the tension between consistency and scalability in distributed applications.

The remainder of this paper is structured as follows. Section 2 discusses our system model and relevant background material. Section 3 presents the warranty abstraction in more detail, and discusses its connection to leases. Section 4 explains in more detail how optimistic transactions are implemented using warranties. The mechanisms needed for computation warranties are explored in Section 5. Our implementation using the Fabric distributed object system is described in Section 6. The evaluation in Section 7 shows that warranties significantly improve the performance of both representative benchmarks and a substantial real-world program. Related work is discussed more broadly in Section 8, and we conclude in Section 9.

2 Background and system model

We assume a distributed system in which each node serves one of two main roles: *client nodes* perform computations locally using persistent data from elsewhere, and *persistent storage nodes* (*stores*) store the persistent data. Client nodes obtain copies of persistent data from stores, perform computations, and send updates to the persistent data back to the stores. For example, the lower two tiers of the traditional three-tier web application match this description: application servers are the clients and database servers are the stores.

Our goal is a simple programming model for application programmers, offering strong consistency so they do not need to reason about inconsistent or out-of-date state. In particular, we want linearizability [25], so each committed transaction acts as though it executes atomically and in logical isolation from the rest of the system. Linearizability strengthens serializability [42, 8] to offer external consistency.

A partially successful attempt at such a programming model is the Java Persistence API (JPA) [12], which provides an object–relational mapping (ORM) that translates accesses to language-level objects into accesses to underlying database rows. JPA implementations such as Hibernate [27] and EclipseLink [15] are widely used to build web applications. However, we want to improve on both the consistency and performance of JPA.

We assume that the working set of both clients and stores fits in the node's memory. This assumption is reasonable for many applications, though not for large-scale data analytics applications, which we do not target.

In a distributed transaction system using OCC (e.g., Thor [37]) clients fetch and then cache persistent objects across transactions. Optimistic caching allows client transactions to largely avoid talking to stores until commit time, unlike with pessimistic locking. The system is faster because persistent data is replicated at the memories of potentially many client nodes. However, care must be taken to avoid inconsistency among the cached copies.

Because of its performance advantages, optimism has become increasingly popular for JPA applications, where the best performance is usually achieved through an "optimistic locking" mode that appears to provide strong consistency in some but not all implementations of JPA.[1]

To provide strong consistency, OCC logs reads and writes to objects. As part of committing the transaction, clients send the transaction log to stores involved in the transaction. The stores then check that the state of each object read matches that in the store (typically by checking version numbers), and then perform updates.

To scale up a distributed computing system of this sort, it is important to be able to add storage nodes across which persistent data and client requests can be distributed. As long as a given client transaction accesses data at just one store, and load is balanced across the stores, the system scales well: each transaction can be committed with just one round trip between the client and the accessed store.

In general, however, transactions access information located at multiple stores. For example, consider a web shopping application. A transaction that updates the user's shopping cart may still need to read information shared among many users of the system, such as details of the item purchased.

Accessing multiple stores hurts scalability. To commit such a transaction serializably, it must be known at commit time that all objects read during the transaction were up to date. A two-phase commit (2PC) is used to ensure this is the case. In the first phase (the prepare phase), each store checks that the transaction can be committed and if so, readies the updates to be committed; it then reports to the coordinator whether the transaction is serializable. If the transaction can be committed at every store, all stores are told to commit in the commit phase. Otherwise, the transaction is aborted and its effects are rolled back.

If popular, persistent data is accessed by many clients, the read contention between clients interferes with scalability. Each client committing a transaction must execute a prepare phase at the store of that data. The work done by the prepare phase consists of *write prepares* done on objects that have been updated by the transaction, and *read prepares* on objects that have been read. In both cases, the object is checked to ensure that the version used was up to date.

Read prepares can make the nodes storing popular objects into bottlenecks even when those objects are rarely updated. This is a fundamental limit on scalability of OCC, so a key benefit of warranties is addressing this performance bottleneck. An alternative strategy would be to replicate popular objects across multiple nodes, but keeping replicas in agreement is very costly.

3 The warranty abstraction

A warranty is a time-limited assertion about the state of the system: it is guaranteed to remain true for some fixed period of time. Warranties improve scalability for two reasons: first, because they reduce or eliminate the work needed for read prepares; second, more generally, they enable the distributed caching of computations and enforce a more semantic notion of consistency.

Because warranties make guarantees about the state of the system, they allow transactions to be committed without preparing reads against the objects covered by

---

[1] The term "optimistic locking" is misleading; locking occurs only during transaction commit. The JPA 2 specification appears to guarantee that objects *written* by a transaction are up to date—but, unfortunately, not the objects *read* unless explicitly locked. Implementations differ in interpretation.

warranties. When all reads to a store involved in a transaction are covered by warranties, that store need not be contacted. Consequently, two-phase commit can be reduced to a one-phase commit in which the prepare and commit phases are consolidated, or even to a zero-phase commit in which no store need be contacted. The result is significantly improved performance and scalability.

In this section, we give a more detailed overview of how warranties work.

- Simple *state warranties* generalize OCC (§3.1) and also, to some extent, leases (§3.2).
- Updates to the system are prevented from invalidating warranties (§3.3), with implications for performance (§3.4).
- Warranty assertions can be expressive, enabling distributed caching of computed results (§3.5).
- Warranties are requested by clients (§3.6) and generated on demand by stores (§3.7).
- Warranties are distributed throughout the system to clients that need them (§3.9).
- The term of warranties can be set automatically, based on run-time measurements (§3.8).

3.1 State warranties

The simplest form of warranty is a *state warranty*, an assertion that the concrete state of an object has a particular value. A warranty is guaranteed to be true (*active*) during the warranty's *term*. At the end of its term, the warranty *expires* and is no longer guaranteed to be true.

For example, a state warranty for an object representing a bank account might be ⟨`assert = {name = "John Doe", bal = $20,345}, exp = 1364412767.1`⟩. Here, the field `assert` specifies the state of the object, and the field `exp` is the time that the warranty expires.

A warranty is issued by a store, and times appearing in the warranties are measured by the clock of the store that issued the warranty. We assume that clocks at nodes are loosely synchronized; well-known methods exist to accomplish this [40].

If a warranty expires before the transaction commits, the warranty may continue to be *valid*, meaning that the assertion it contains is still true even though clients cannot rely on its remaining true. Clients can, however, still use the warranty optimistically and check at commit time that the warranty remains valid.

As can be seen, state warranties generalize optimistic concurrency control. Ordinary OCC equates to always receiving a zero-length warranty for the state of the object read, and using that expired warranty optimistically.

3.2 Warranties vs. leases

Leases [21] have been used in many systems (e.g., [51, 2]) to improve performance. Warranties exploit the key insight of leases that time-limited guarantees increase scalability by reducing coordination overhead. As defined originally by Gray and Cheriton, leases confer time-limited *rights* to access objects in certain ways, and must be held by clients in order to perform the corresponding access. Conversely, warranties are time-limited *assertions* about what is *true* in the distributed system, and are not, therefore, held by any particular set of nodes. Unlike with leases, an expired warranty may be used to access an object optimistically. Gray does sketch in his dissertation [20] how read leases might be integrated into an optimistic transaction processing system, but we are not aware of any detailed design or implementation.

Leases and warranties do partly overlap. Since *read leases* on objects effectively prevent modifying object state, they must enforce assertions regarding the state of that data. Therefore, state warranties can be viewed as read leases that are given to many clients and that cannot be relinquished by those clients.

However, we see a fundamental difference between these two perspectives. The value of the warranty (assertion) perspective is that state warranties naturally generalize to expressive assertions over state—in particular, warranties that specify the results of application-defined computations over the state of potentially many objects.

3.3 Defending warranties

Transactions may try to perform updates that affect objects on which active warranties have been issued. Updates cannot invalidate active warranties without potentially violating transactional isolation for clients using those warranties. Therefore, stores must defend warranties against invalidating updates, a process that has no analogue in OCC.

A warranty can be defended against an invalidating update transaction in two ways: the transaction can either be rejected or delayed. If rejected, the transaction will abort and the client must retry it. If delayed, the updating transaction waits until it can be safely serialized. Rejecting the transaction does not solve the underlying problem of warranty invalidation, so delaying is typically the better strategy if the goal is to commit the update. To prevent write starvation, the store stops issuing new warranties until after the commit. The update also shortens the term of subsequent warranties.

3.4 Performance tradeoffs

Using warranties improves read performance for objects on which warranties are issued, but delays writes to these objects. Such a tradeoff appears to be an unavoidable with strong consistency. For example, in conventional database systems that use pessimistic locking to enforce consistency, readers are guaranteed to observe consistent states, but update transactions must wait until all read transactions have completed and released their locks. With many simultaneous readers, writers can be significantly delayed. Thus, warranties occupy a middle ground between optimism and pessimism, using time as a way to reduce the coordination overhead incurred with locking.

The key to good performance, then, is to issue warranties that are long enough to allow readers to avoid revalidation but not so long that they block writers more than they otherwise would be blocked.

For applications where it is crucial to have both high write throughput and high read throughput to the same object, replication is essential, and the cost of keeping object replicas in sync makes strong consistency infeasible. However, if weak consistency is acceptable, there is a simple workaround: implement replication by explicitly maintaining the state in multiple objects. Writes can go to one or more persistent objects that are read infrequently, and only by a process that periodically copies them (possibly after reconciliation of divergent states) to a frequently read object on which warranties can be issued. This is a much easier programming task than starting from weak consistency and trying to implement strong consistency where it is needed. The only challenging part is reconciliation of divergent replicas, which is typically needed in weakly consistent systems in any case (e.g., [50, 47, 14]).

3.5 Computation warranties

Warranty assertions are not limited to specifying the concrete state of persistent objects. In general, a warranty assertion is an expression in a language that can describe a computation that operates on persistent objects and that can be evaluated at the store. SQL is one query language that fits this description, but in this work, we integrate assertions more tightly with the programming language. Computation warranties provide guarantees about computations described in terms of method calls.

In current distributed applications, it is common to use a distributed cache such as `memcached` [18] to share data and computation across many nodes. For example, web application servers can cache the text of commonly used web pages or content to be included in web pages. Computation warranties can be used to cache such computed results without abandoning strong consistency.

Example: top N items. Many web applications display the top-ranked N items among some large set (such as advertisements, product choices, search results, poll candidates, or game ladder rankings).

Although the importance of having consistent rankings may vary across applications, there are at least some cases in which the right ranking is important and may have monetary or social impact. Election outcomes matter, product rankings can have a large impact on how money is spent, and game players care about ladder rankings. But at present there is no easy and efficient way to ensure that cached computation results are up to date.

To cache the results of such a computation, we might define a computation `top(n, i, j)`, which returns the set `s` of the `n` top-ranked items whose indices in an array of items lie between `i` and `j`. A warranty of the form `s = top(n, 0, num_items)` then allows clients to share the computation of the top-ranked items within the range.

The reason why the `top` function has arguments `i` and `j` is to permit `top` to be implemented recursively and efficiently using results from subranges, on which further warranties are issued. We discuss later in more detail how this approach allows computation warranties to be updated and recomputed efficiently.

Example: airplane seats. Checking whether airplane flights have open seats offers a second example of a computation that can be worth caching. Because the client-side viewer may be sorting lists of perhaps hundreds of potential flights, flights are viewed much more often than their seating is updated. Scalability of the system would be hurt by read prepares.

Efficient searching over suitable flights can be supported by issuing warranties guaranteeing that at least a certain number of seats of a specified type are available; for a suitable constant number of seats n large enough to make the purchase, a warranty of this form works:

$$\texttt{flight.seats\_available(type)} \geq n$$

This warranty helps searching efficiently over the set of flights on which a ticket might be purchased. It does not help with the actual update when a ticket is purchased on a flight. In this case, it becomes necessary to find and update the actual number of seats available. However, this update can be done quickly as long as the update does not invalidate the warranty.

Like state warranties, computation warranties can be used optimistically even if they expire during the transaction. In this case, the dependencies of the computation described in the warranty must be checked at commit time to ensure that the warranty's assertion remains true, just as objects whose state warranties expire before commit time must be checked. A warranty that is revalidated in this fashion can then be issued as a new warranty.

Like active state warranties, active computation warranties must be defended against invalidation by updates. This mechanism is discussed in Section 5.2.

3.6 Programming with warranties

As clients compute, they request warranties as needed. State warranties are requested automatically when objects are newly fetched by a computation. Computation warranties can also be generated in a natural way, relying on simple program annotations.

Computation warranties explicitly take the form of logical assertions, so they could be requested by using a template for the desired logical assertion. In the airline seat reservation example above, a query of the form `flight.seats_available(type)` $\geq$? could be used to find all available warranties matching the query, and at the same time fill in the "?" with the actual value n found in the warranty. In the case where multiple warranties match, a warranty might be chosen whose duration and value of n are "best" according to application-specific criteria.

We pursue a more transparent way to integrate warranty queries into the language, via memoized function calls. For example, we can define a memoized method with the signature `memoized boolean seats_lb(type,` n`)` that returns whether there are at least n seats of the desired type still available on the flight. The keyword `memoized` indicates that its result is to be memoized and warranties are to be issued on its result. To use these warranties, client code uses the memoized method as if it were an ordinary method, as in the following code:

```
for (Flight f : flights)
  if (f.seats_lb(aisle, seats_needed))
    display_flights.add(f);
```

When client code performs a call to a memoized method, the client automatically checks to see if a warranty for the assertion ? = `seats_lb(type, n)` has either been received already or can be obtained. If so, the result of the method call is taken directly from the warranty. If no warranty can be found for the method call, the client executes the method directly.

With appropriate language support, the implementation of such a memoized method is also straightforward:

```
memoized boolean seats_lb(Seat t, int n) {
    return seats_available(t) >= n;
}
```

A language that correctly supports transparent OCC already automatically logs the reads and writes performed on objects; this logging already computes the dependencies of computation warranties.

3.7 Generating warranties

Warranties are issued by stores, because stores must know about warranties in order to defend them against updates that might invalidate them. However, for scalability, it is important to avoid giving the store extra load. Therefore, it only makes sense to generate warranties for some objects and computations: those that are used much more frequently than they are invalidated.

For state warranties, the store already has enough information to decide when to generate a warranty for an object, because it sees both when the object is updated and when it is necessary to check that the version of the object read by a client is up to date. State warranties improve performance by removing the need to do version checks on read objects, but at the cost of delaying updates that would invalidate active warranties. This tradeoff makes sense if the version checks are sufficiently more numerous than the updates.

For computation warranties, the store may be able to infer what warranties are needed from client requests, but it makes more sense to have the client do the computational work. Recall that clients that fail to find a suitable warranty compute the warranty assertion themselves. If the assertion is true, it is the basis of a potential warranty that is stored in the client's local cache and reused as needed during the same transaction. As part of committing the transaction, the client sends such potential warranties to the store, which may issue these warranties, both back to this client and to other clients. The decision whether to issue a warranty properly depends on whether issuing the warranty is expected to be profitable.

3.8 Setting warranty terms

Depending on how warranty terms are set, warranties can either improve or hurt performance. However, it is usually possible to automatically and adaptively set warranty terms to achieve a performance increase.

Warranties improve performance by avoiding read prepares for objects, reducing the load on stores and on the network. If *all* read and write prepares to a particular store can be avoided, warranties eliminate the need even to coordinate with that store.

Warranties can hurt performance primarily by delaying writes to objects. The longer a warranty term is, the longer the write is delayed. If warranty terms are set too long, writers may experience unacceptable delays. A good rule of thumb is that we would like writers to be delayed no more than they would be by read locks in a system using pessimistic locks.

Excessively long warranties may also allow readers to starve writers, although starvation is mitigated because new warranties are not issued while writers are blocked waiting for a warranty to expire. Note that with pure OCC, writers can block readers by causing all read prepares to fail [43]; thus, warranties shift the balance of power away from writers and toward readers, addressing a fundamental problem with OCC.

To find the right balance between the good and bad effects of warranties, we take a dynamic, adaptive approach. Warranty terms are automatically and individually set by stores that store the relevant objects. Fortunately, stores observe enough to estimate whether war-

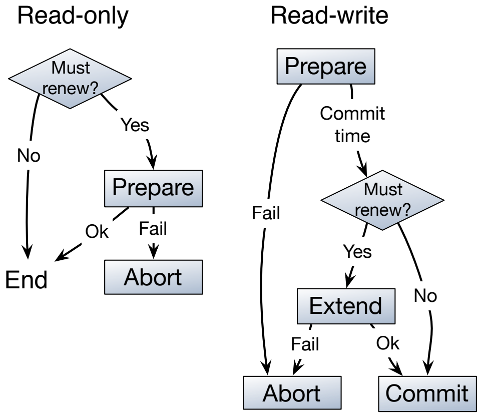

Figure 1: Warranty commit protocol for read-only and read-write transactions.

ranty terms are likely to be profitable. Stores see both read prepares and write prepares. If the object receives many read prepares and few or no write prepares, a state warranty on that object is likely to be profitable. A similar observation applies to computation warranties.

To determine whether to issue a warranty for an object, and its warranty term L in the case where a warranty is issued, the system plugs measurements of object usage into a simple system model. The system measures the rate W of writes to each object, and when there is no warranty issued on the object, it also measures the rate R of reads to the object. Both rates are estimated using an exponentially weighted moving average (EWMA) [28] of the intervals between reads and writes. We modify EWMA to exponentially decay historical read-prepare data during warranty periods, when read prepares cannot be observed. Empirically, this modification improves the accuracy of rate estimation. To lower the overhead of monitoring, unpopular objects are flagged and given lower-cost monitoring as long as they remain unpopular.

To ensure that the expected number of writes delayed by a warranty is bounded by a constant $k_1 < 1$ that controls the tradeoff between read and write transactions. The warranty term is set to k_1/W with a maximum warranty L_{max} used to bound write delays. Our goal is that warranties are profitable: they should remove load from the store, improving scalability. A warranty eliminates roughly RL read prepares over its term L, but adds the cost of issuing the warranty and some added cost for each write that occurs during the term. The savings of issuing a warranty is positive if each write to an object is observed by at least k_2 reads for some value k_2, giving us a condition $RL \geq k_2$ that must be satisfied in order to issue a warranty. The value for constant k_2 can be derived analytically using measurements of the various costs, or set empirically to optimize performance.

This way to set terms for state warranties also works for computation warranties, with the following interpretation: uses of a computation warranty are "reads" and updates to its dependencies are "writes".

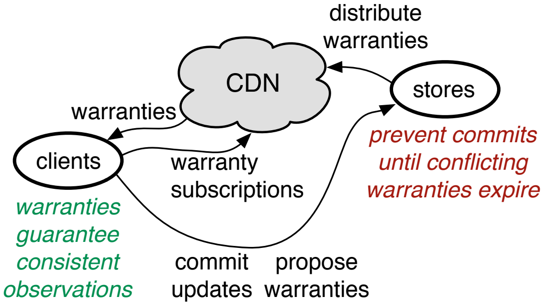

Figure 2: Warranty distribution architecture.

The tension between write latency and read throughput can also be eased by using warranty refresh in addition to a maximum warranty term. The term L is computed as above, but warranties are issued to clients with a shorter term corresponding to the maximum acceptable update latency. The issuing store proactively refreshes each such warranty when it is about to expire, so the warranty stays valid at clients throughout its term.

3.9 Distributing warranties

Warranties can be used regardless of how they get to clients and can be shared among any number of clients. Therefore, a variety of mechanisms can be used to distribute warranties to clients.

One option for warranty distribution is to have clients directly query stores for warranties, but this makes the system less scalable by increasing load on stores. As shown in Figure 2, Stores will be less loaded if warranties are distributed via a content distribution network (CDN) that clients query to find warranties.

Going a step further, applications can *subscribe* to warranties that match a given pattern, as shown in Figure 2. Stores automatically *refresh* warranties with later expiration times before the old warranties expire, by pushing these extended warranties either directly to clients or into the CDN. Warranty refresh makes it feasible to satisfy client requests with shorter warranty terms, consequently reducing write latency.

This strategy for achieving high availability and high durability differs from that used in many current distributed storage systems, which use replication to achieve high availability, low latency, and durability. Those three goals are handled separately here. Distributing warranties through a CDN makes data objects highly available with low latency, without damaging consistency. Because the authoritative copies of objects are located at stores, a write to an object requires a round-trip to its store; the latency this introduces is ameliorated by the

	Stores		Phases:	
Stores	written	Unexpired?	Warranties	OCC
1+	0	Y	0	1
1+	0	N	1	1
1	1	Y/N	1	1
2+	1	Y	1	2
2+	1	N	2	2
2+	2+	Y	2	2
2+	2+	N	3	2

Table 1: Warranties require fewer phases than traditional OCC in some cases (highlighted).

support for relatively large transactions, in which communication with stores tends to happen at the end of transactions rather than throughout.

To achieve high durability, stores should be implemented using replication, so that each "store" mentioned in this paper is actually a set of replicas. Since wide-area replication of stores implementing strong consistency will have poor performance, we assume store replicas are connected with low latency.

4 Transactions and warranties

Warranties improve the performance of OCC by reducing the work needed during the prepare phase and by allowing phases to be eliminated entirely.

4.1 The warranty commit protocol

When a transaction completes, the client performs a modified two-phase commit, illustrated in Figure 1 for both read-only and read-write transactions. In the prepare phase, the client sends the write set of the transaction (if any), along with any warranties in the read set whose term has expired. If all warranties in the read set can be renewed, the transaction may commit. Since outstanding warranties may cause the updates to be delayed, the store responds with a *commit time* indicating when the commit may be applied successfully.

When the client receives a commit time from all stores, it checks to ensure the terms of the warranties it holds exceed the maximum commit time. If not, it attempts to renew these warranties beyond the commit time in an additional *extend* phase. If active warranties are obtained for all dependencies, the client sends the commit message, and the stores commit the updates at the specified time.

4.2 Avoiding protocol phases

While a two-phase commit is required in the general case, performance can be improved by eliminating or combining phases when possible. For read-only transactions, the commit phase is superfluous, and clients executing transactions that involve only one store can combine the prepare and commit phases into one round-trip. The optimizations to 2PC that warranties make possible are summarized in Table 1.

The read-only (rows 1–2) and single-store optimizations (row 3) are available with or without warranties. However, unexpired warranties enable eliminating additional phases, shown by the two rows highlighted in gray.

Row 1 shows that read-only transactions whose read set is covered by unexpired warranties may commit without communicating with stores—a zero-phase commit. This optimization matters because for read-biased workloads, most transactions will be read-only.

Row 4 shows that transactions that read from multiple stores but write to only one store may commit in a single phase if their read set is fully warrantied. This single-phase optimization pays off if objects are stored in such a way that writes are localized to a single store. For example, if a user's information is located on a single store, transactions that update only that information will be able to exploit this optimization.

While warranties usually help performance, they do not strictly reduce the number of phases required to commit a transaction. Transactions performing updates to popular data may have their commits delayed. Since the commit time may exceed the expiration time of warranties used in the transaction, the additional *extend* phase may be required to renew these warranties beyond the delayed commit time, as shown in the final row.

5 Computation warranties

A computation warranty is a guarantee until time t of the truth of a logical formula ϕ, where ϕ can mention computational results such as the results of method calls. We focus here on the special case of warranties generated by memoized function calls, where ϕ has the form $o.f(\vec{x}) =?$ for some object o on which method f is invoked using arguments $\vec{x}$, producing a value to be obtained from the warranty. Note that the value returned by f need not be a primitive value. In the general case, it may be a data structure built from both new objects constructed by the method call and preexisting objects.

Our goal is that warranties do not complicate programmer reasoning about correctness and consistency. Therefore, when f is a memoized method, a computation of the form $v = o.f(\vec{x})$ occurring in a committed transaction should behave identically whether or not a warranty is used to obtain its value. This principle has several implications for how computation warranties work. It means that only some computations make sense as computation warranties, and that updates must be prevented from invalidating active warranties.

5.1 Memoizable computations

To ensure that using a computation warranty is equivalent to evaluating it directly, we impose three restrictions.

First, computation warranties must be deterministic: given equivalent initial state, they must compute equivalent results. Therefore, computations using a source of nondeterminism, such as input devices or the system clock, do not generate computation warranties.

Second, we prevent memoization of any computation that has observable side effects. Side effects are considered to be observable only when they change the state of objects that existed before the beginning of the memoized computation.

Importantly, this definition of "observable" means that memoized computations are allowed to create and initialize new objects as long as they do not modify preexisting ones. For example, the top-N example from Section 3.5 computes a new object representing a set of items, and it may be convenient to create the object by appending items sequentially to the new set. Warranties on this kind of side-effecting computation are permitted. Enforcing this definition of the absence of side effects is straightforward in a system that already logs which objects are read and written by transactions.

Third, a memoized function call reads from some set of objects, so updates to those objects may change its result, and may occur even during the same transaction that performed the function call. At commit time, the transaction's write set is intersected with the read set of each potential warranty. If the intersection is nonempty, the potential warranty is invalidated.

5.2 Defending computation warranties

Once a computation warranty is requested by a worker and issued by a store, the store must ensure that the value of the call stays unchanged until the warranty expires.

Revalidation A conservative way to defend warranties against updates would be to delay all transactions that update objects used by the warranty. This approach is clearly safe because of the determinism of the warranty computation, but it would prevent too many transactions from performing updates, hurting write availability. Instead, we attempt to *revalidate* affected warranties when each update arrives. The store reruns the warranty computation and checks whether the result is equivalent to the result stored in the warranty.

For primitive values and references to pre-existing objects (not created by the warranty computation), the result must be unchanged. Otherwise, two results are considered equivalent if they are semantically equal per the `equals()` method, which operates as in Java.

Warranty dependencies In general, a warranty computation uses and thus depends on other warranties, whether state warranties or general computation warranties. For example, if the method `top` is implemented recursively (see Figure 3), the warranty for a call to `top` depends on warranties for its recursive calls. The dependencies between warranties form a tree in which computation warranties higher in the tree depend on warranties lower down, and the leaves are state warranties.

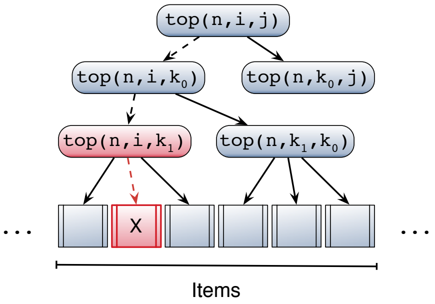

Figure 3: An update to X causes a semantic warranty to be invalidated, but the updated value for the re-evaluated method does not invalidate other warranties.

Any warranty that has not expired must be defended against updates that could invalidate it. Defense is easy when the term of a warranty is contained within (a subset of) the terms of all warranties it depends on, including state warranties on all direct references to objects, because the validity of the higher-level warranty is implied by the defense of the lower-level warranties.

In general, however, a warranty can have a longer term than some of its dependencies. Updates to those dependencies must be prevented if they invalidate the warranty, even if they are expired warranties. Conversely, it is possible to allow updates to warranty dependencies that do not invalidate the warranty. The implication is that it is often feasible to give higher-level warranties longer terms than one might expect given the rate of updates to their dependencies.

For example, consider the recursive call tree for the method `top(n, i, j)` shown in Figure 3. If the request to see the top n items among the entire set is very popular, we would like to issue relatively long computation warranties for that result. Fortunately, updates to items (shown at the leaves of the call tree) that change their ranking might invalidate some of the warranties in the tree, but most updates will affect only a small part of the tree. Assuming that lower levels of the tree have short warranties, most updates need not be delayed much.

5.3 Reusing computation warranty values

In the case where the warranty computation created new objects, it may be crucial for correctness of the computation that the objects returned by the warranty are distinct from any existing objects. This desired semantics is achieved when using a warranty computation result by making a copy of all objects newly created during the warranty computation. These objects are explicitly identified in the warranty.

Computation warranties are used whenever available to the client, to avoid performing the full computation. If the client is holding an expired warranty, or obtains an expired warranty from the CDN, it can use that expired warranty optimistically. At commit time, the expired warranty is revalidated during the prepare phase, exactly like a read prepare.

5.4 Creating computation warranties

Whenever code at a client makes a call to a memoized method, the client searches for a matching computation warranty. If the client is not already holding such warranty, it may search using a CDN, if available, or request the warranty directly from the appropriate store.

If the client cannot find an existing computation warranty, it performs the warranty computation itself. It starts a new transaction and executes the method call. As the call is evaluated, the transaction's log keeps track of all reads, writes, and object creations performed by the call. When the call is completed, the result is recorded and the log is checked to verify that the call does not violate any of the restrictions outlined above. If the warranty is still valid, the call, value, and transaction log are gathered to form a complete warranty proposal.

At commit time, if the warranty proposal has not already been invalidated by an update to its read set, the proposal is sent to the store. The store looks at the request and, using the same mechanism as for state warranties, sets a warranty term. For state warranties, terms are set individually for each object, but here the warranty identity is defined by the entire set of arguments to the memoized method. Finally, the computation warranty is issued to the requesting client and the store begins to defend the new warranty or warranties proposed by the client.

6 Implementation

To evaluate the warranty mechanism, we extended the Fabric secure distributed object system [38]. Fabric provides a high-level programming model that, like the Java Persistence API, presents persistent data to the programmer as language-level objects. Language-level objects may be both persistent and distributed. It implements linearizability using OCC.

Fabric also has many security-related features—notably, information flow control—designed to support secure distributed computation and also secure mobile code [5]. The dynamic security enforcement mechanisms of Fabric were not turned off for our evaluation, but they are not germane to this paper.

We extended the Fabric system and language to implement the mechanisms described in this paper. Our extended version of Fabric supports both state warranties and computation warranties. Computation warranties were supported by extending the Fabric language with memoized methods. Client (worker) nodes were extended to use warranties during computation and to evaluate and request computation warranties as needed. The Fabric dissemination layer, a CDN, was extended to distribute warranties and to support warranty subscriptions. Fabric workers and stores were extended to implement the new transaction commit protocols, and stores were extended to defend and revalidate warranties.

The previously released version of Fabric (0.2.1) contains roughly 44,000 lines of (non-blank, non-comment) code, including the Fabric compiler and the run-time systems for worker node, store nodes, and dissemination nodes, written in either Java or the Fabric intermediate language. In total, about 6,900 lines of code were added or modified across these various system components to implement warranties.

Fabric ships objects from stores to worker nodes in object groups rather than as individual objects. State warranties are implemented by attaching individual warranties to each object in the group.

Some features of the warranties design have not been implemented; most of these features are expected to improve performance further. The single-store optimization of the commit protocol has been implemented for base Fabric, but rows 3–5 of Table 1 have not been implemented for warranties. The warranty refresh mechanism is also not yet implemented.

To simplify the work needed to defend computation warranties, the current implementation only generates warranties for computations that involve objects from a single store. Also, our implementation does not use the dissemination layer to distribute computation warranties.

7 Evaluation

We evaluated warranties against existing OCC mechanisms, and other transactional mechanisms, primarily using three programs. First, we used the multiuser OO7 benchmark [13]. Second, we used versions of Cornell's deployed Course Management System [10] (CMS) to examine how warranties perform with real systems under real-world workloads. Both of these programs were ported to Fabric in prior work [38]. Third, we developed a new benchmark that simulates a component of a social network in which users have subscribers.

7.1 Multiuser OO7 benchmark

The OO7 benchmark was originally designed to model a range of applications typically run using object-oriented databases. The database consists of several modules, which are tree-based data structures in which each leaf of the tree contains a randomly connected graph of 20 objects. In our experiments we used the "SMALL" sized database. Each OO7 transaction performs 10 random traversals on either the *shared* module or a *private* module specific to each client. When the traversal reaches a leaf of the tree, it performs either a read or a write action. These are relatively heavyweight transactions compared to many current benchmarks; each transaction reads about 460 persistent objects and modifies up to 200 of them. By comparison, if implemented in a straightforward way with a key-value store, each transaction would perform hundreds of get and put operations. Transactions in the commonly used TPC-C benchmark are also roughly an order of magnitude smaller [52], and in the YCSB benchmarks [54], smaller still.

Because OO7 transactions are relatively large, and because of the data's tree structure, OO7 stresses a database's ability to handle read and write contention. However, since updates only occur at the leaves of the tree, writes are uniformly distributed in the OO7 specification. To better model updates to popular objects, we modified traversals to make read operations at the leaves of the tree exhibit a power-law distribution with $\alpha = 0.7$ [11]. Writes to private objects are also made power-law distributed, but remain uniformly distributed for public objects.

7.2 Course Management System

The CS Course Management System [10] (CMS) is a 54k-line Java web application used by the Cornell computer science department to manage course assignments and grading. The production version of the application uses a conventional SQL database; when viewed through the JPA, the persistent data forms an object graph not dissimilar to that of OO7. We modified this application to run on Fabric. To evaluate computation warranties, we memoized a frequently used method that filters the list of courses on an overview page.

We obtained a trace from Cornell's production CMS server from three weeks in 2013, a period that encompassed multiple submission deadlines for several courses. To drive our performance evaluation, we took 10 common action types from the trace. Each transaction in the trace is a complete user request including generation of an HTML web page, so most request types access many objects. Using JMeter [30] as a workload generator, we sampled the traces, transforming query parameters as necessary to map to objects in our test database with a custom JMeter plugin.

7.3 Top-subscribers benchmark

The third benchmark program simulates a relatively expensive analytics component of a social network in which users have subscribers. The analytics component computes the set of 5 users with the largest number of subscribers, using the memoized top-N function described in Section 3.5. The number of subscribers per user is again determined by a power-law distribution with $\alpha = 0.7$. The workload consists of a mix of two operations: 98% compute the list of top subscribers, corresponding to viewing the home page of the service; 2% are updates that randomly either subscribe or unsubscribe some randomly chosen user. This example explores the effectiveness of computation warranties for caching expensive computed results.

7.4 Comparing with Hibernate/HSQLDB

To provide a credible baseline for performance comparisons, we also ported our implementation of CMS to the Java Persistence API (JPA) [12]. We ran these implementations with the widely used Hibernate implementation of JPA 2, running on top of HyperSQL (HSQLDB), a popular in-memory database in `READ COMMITTED` mode. For brevity, we refer to Hibernate/HSQLDB as *JPA*. For JPA, we present results only for a single database instance. Even in this single-store setting, and even with Hibernate running in its optimistic locking mode, which does not enforce serializability, Fabric significantly outperforms JPA in all of our experiments. (Note that JPA in optimistic locking mode is in turn known to outperform JPA with pessimistic locking, on read-biased workloads [49, 17]). This performance comparison aims to show that Fabric is a good baseline for evaluating the performance of transactional workloads: its performance is competitive with other storage frameworks offering a transactional language-level abstraction.

7.5 Experimental setup

Our experiments use a semi-open system model. An open system model is usually considered more realistic [48] and a more appropriate way to evaluate system scalability. Worker nodes execute transactions at exponentially distributed intervals at a specified *average request rate*. Consequently, each worker is usually running many transactions in parallel. Overall system throughput is the total of throughput from all workers. To find the maximum throughput, we increase the average request rate until the target throughput cannot be achieved.

The experiments are run on a Eucalyptus cluster. Each store runs on a virtual machine with a dual core processor and 8 GB of memory. Worker machines are virtual machines with 4 cores and 16 GB of memory. The physical processors are 2.9 GHz Intel Xeon E5-2690 processors.

The parameters k_1 and k_2 (Section 3.8) are set to 0.5 and 2.0, respectively; the maximum warranty term was 10 s. Performance is not very sensitive to k_1 and k_2.

7.6 Results

We evaluated scalability using the OO7 benchmark with different numbers of stores. A "shared store" was reserved for the assembly hierarchies of all modules. The component parts of the modules were distributed evenly across the remaining stores. Only shared composite parts

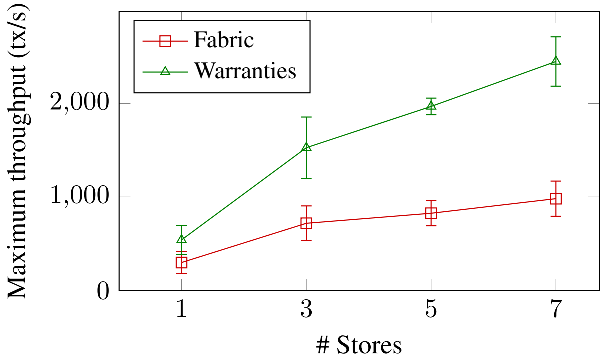

Figure 4: OO7 maximum throughput on a 2%-write workload as the number of stores increases. Warranties allow throughput to scale up with more stores.

were placed on the shared store. Results presented are the average of three runs.

Figure 4 shows maximum throughput in total transactions committed per second by 36 workers, as the number of stores increases. Error bars show the standard deviation of the measurements. As expected, adding stores has little effect on maximum throughput in base Fabric because the shared store is a bottleneck. Warranties greatly reduce load on the shared store allowing us to add roughly 400 tx/s per additional store. Note that the plot only counts committed transactions; the percentage of aborted transactions for Fabric at maximum throughput ranges from 2% to 6% as the number of stores increases from 3 to 7; with warranties, from 4% up to 15%.

Table 2 reports on the performance of the CMS application in various configurations. The first three rows of Table 2 show that Fabric, without or without warranties, delivers more than an order of magnitude performance improvement over JPA. Although the JPA implementation enforces weaker consistency, Fabric's more precise object invalidation helps performance as contention increases. Warranties help improve performance further, even in a single-store configuration.

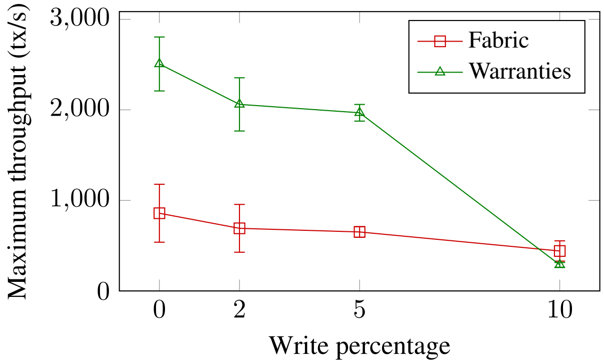

Figure 5: Effect of write percentage on OO7 maximum throughput on 3 stores with 24 workers.

System	Stores	Tput (tx/s)	Latency (ms)
JPA	1	72 ± 12	211 ± 44
Fabric	1	3032 ± 144	143 ± 120
Warranties	1	4142 ± 112	27 ± 27
Comp. Warranties	1	4088 ± 189	114 ± 30
Fabric	3	4090 ± 454	311 ± 175
Warranties	3	5886 ± 124	35 ± 4

Table 2: CMS throughput and latency on various systems. Both are averaged over 10 s at max throughput.

To evaluate how the system scales for a more realistic workload, we also ran CMS with 3 stores using Fabric and Warranties. Two stores each held data for multiple courses, while the third store contained metadata. As Table 2 shows, Warranties scale better than Fabric with the additional stores.

Increases in throughput would be less compelling if they came at the cost of high latency. Table 2 also reports the latency measured with the CMS workload on the various systems. Fabric has similar latency with or without warranties. Because CMS was not designed with computation warranties in mind, the functions we designated to be memoized turn out not to have a significant impact on performance. They are relatively cheap to evaluate on cached objects, and the bookkeeping for computation warranties adds no noticeable overhead.

Figure 5 shows how the performance of warranties is affected by the fraction of update transactions. Four different workload mixes were measured, each having a 94:6 shared-to-private traversal ratio and a 1:10 shared-to-private write ratio. When more than 10% of the transactions are updates, the cost of maintaining and issuing warranties in the current implementation is too high to obtain a performance improvement. The latencies at some of these throughputs are higher than Fabric's, but still relatively low. At 2% and 5% writes, the latency of warranties is about 400 ms higher than Fabric's but nearly the same as Fabric's at 0% and 10% writes.

Warranties can result in delaying transactions that are attempting to write to an object that has a warranty. We call this *write delay*. For all of the runs depicted in Figure 5, the median write delay is 0 ms. However, some fraction of transactions are forced to wait until one or more warranties expire. The more read-biased the transaction, the more frequently this happens. In the 2%-write workload, 70% of read-write transactions see no write delay. In the 10%-write workload, 82% see no write delay. Among those that encounter write delay, the delay is roughly uniformly distributed from 0 up to the max warranty length.

	Tput	Median Latency	95th pct Write Delay
Fabric	17 ± 5	568 ± 500	N/A
Warranties	26 ± 7	1239 ± 644	623 ± 387
Comp. Warranties	343 ± 14	12 ± 3	16 ± 5

Table 3: Top-N benchmark: maximum throughput (tx/s), latency (ms), and 95th percentile write delay (ms).

7.7 Computation warranties

To further evaluate the impact of computation warranties, we ran the top-N benchmark with Fabric, state warranties, and with computation warranties. Because the performance of the recursive top-N strategy on Fabric and on state warranties was very poor, we used an alternate implementation that performed better on those configurations. Table 3 shows the average across three runs of the maximum throughput and the corresponding latency achieved in the system without any operations failing to commit during a 15 minute period. Computation warranties improve throughput by more than an order of magnitude. Since the computation warranty is on the value of the top 5 accounts rather than on each individual value used in computing the result, writes are not delayed as heavily as they are when using only state warranties.

8 Related work

Many mechanisms for enforcing concurrency control have been proposed in the literature: locks, timestamps, versions, logs, leases, and many others [33, 22, 34, 46, 7, 21]. Broadly speaking, these can be divided into optimistic and pessimistic mechanisms. The monograph by Bernstein, Hadzilacos, and Goodman provides a broad overview from the perspective of databases [8]. Warranties are an optimistic technique, allowing clients to concurrently operate on shared data.

Haerder [24] divides mechanisms for validating optimistic transactions into "forward" and "backward" techniques. Backward validation is a better choice for the distributed setting [3], so Fabric uses backward validation: transactions are aborted in the prepare phase if any object in the read set has been modified.

Traditionally, most systems adopted *serializability* or *linearizability* as the gold standard of strong consistency [42, 8, 25]. But many recent systems have sacrificed serializability in pursuit of scalable performance. Vogels [53] discusses this trend and surveys various formal notions of *eventual consistency*. Much prior work aims to provide a consistency guarantee that is weaker than serializability; for example, causal consistency (e.g., [44, 39]) and *probabilistically-bounded staleness* [6]. Because this paper is about strong consistency, we do not discuss this prior work in depth.

Leveraging application-level information to guide implementations of transactions was proposed by Lamport [33] and explored in Garcia-Molina's work on *semantic types* [19], as well as recent work on *transactional boosting* [26] and *coarse-grained transactions* [31]. Unlike warranties, these systems use mechanisms based on commuting operations. A related approach is *red–blue consistency* [36], in which red operations must be performed in the same order at each node but blue operations may be reordered.

Like warranties, Sinfonia [4] aims to reduce client–server round trips without hurting consistency. It does this through *mini-transactions*, in which a more general computation is piggybacked onto the prepare phase. This optimization is orthogonal to warranties.

Warranties borrow from leases [21] the idea of using expiring guarantees, though important differences are discussed in Section 3.2. In fact, the idea of expiring state guarantees occurs prior to leases in Lampson's global directory service [35]. We are not aware of any existing system that combines optimistic transactions with leases or lease-like mechanisms, against which we could meaningfully compare performance.

A generalization of leases, *promises* [23, 29] is a middleware layer that allows clients to specify resource requirements via logical formulas. A resource manager considers constraints across many clients and issues time-limited guarantees about resource availability. Scalability of promises does not seem to have been evaluated.

The tracking of dependencies between computation warranties, and the incremental updates of those warranties while avoiding unnecessary invalidation, is close to the update propagation technique used in self-adjusting computation [1], realized in a distributed setting. Incremental update of computed results has also been done in the setting of MapReduce [9].

The TxCache system [45] provides a simple abstraction for sharing cached results of functions operating over persistent data from a single storage node in a distributed system. As with the Fabric implementation of computation warranties, functions may be marked for memoization. TxCache does not ensure that memoized calls have no side effects, so memoized calls may not behave like real calls. Compared to Fabric, TxCache provides a weaker consistency guarantee, transactional consistency, requiring that all transactions operate over data that is consistent with a prior snapshot of the system.

Escrow transactions [41] have some similarities to computation warranties. They generalize transactions by allowing commit when a predicate over state is satisfied. Certain updates (incrementing and decrementing values) may take place even when other transactions may be updating the same values, as long as the predicate still holds. Compared to computation warranties, escrow transactions support very limited predicates over state, and their goal is different: to permit updates rather than to allow the result of a computation to be widely reused.

9 Conclusions

Strong consistency tends to be associated with the very real performance problems of pessimistic locking. While optimistic concurrency control mechanisms deliver higher performance for typical workloads, read prepares on popular objects are still a performance bottleneck. Warranties generalize OCC in a way that reduces the read-prepare bottleneck. Warranties address this bottleneck by allowing stores to distribute warranties on popular objects, effectively replicating their state throughout the system. Warranties can delay update transactions, but our results suggest that the delay is acceptable. Effectively, warranties generalize OCC in a way that adjusts the balance of power between readers and writers, substantially increasing overall performance. Computation warranties improve performance further by supporting `memcached`-like reuse of computations—but without losing strong consistency.

Acknowledgments

We would especially like to thank Robert Soulé for help setting up experiments and Nate Foster for good suggestions. Chin Isradisaikul also had good ideas for presentation, and we thank our shepherd Yuan Yu. We thank Hakim Weatherspoon for the use of Fractus cloud infrastructure provided by an AFOSR DURIP award, grant FA2386-12-1-3008.

This project was funded partly by the Office of Naval Research (grant N00014-13-1-0089), by MURI grant FA9550-12-1-0400, by a grant from the National Science Foundation (CCF-0964409), and by an NDSEG Fellowship. This paper does not necessarily reflect the views of any of these sponsors.

References

[1] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *35th ACM Symp. on Principles of Programming Languages (POPL)*, pages 309–322, 2008.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, December 2002.

[3] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA, May 1995.

[4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *21st ACM Symp. on Operating System Principles (SOSP)*, pages 159–174, October 2007.

[5] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *IEEE Symp. on Security and Privacy*, pages 191–205, May 2012.

[6] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 5(8):776–787, April 2012.

[7] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM CSUR*, 13(2):185–221, 1981.

[8] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987. Available at `http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx`.

[9] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: MapReduce for incremental computations. In *ACM Symp. Cloud Computing*, October 2011.

[10] Chavdar Botev et al. Supporting workflow in a course management system. In *36th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 262–266, February 2005.

[11] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.

[12] Heiko Böck. *Java Persistence API*. Springer, 2011.

[13] Michael Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *9th ACM SIGPLAN Conf. on Object-Oriented Programming,*

Systems, Languages and Applications (OOPSLA), pages 414–426, 1994.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[15] EclipseLink. `http://www.eclipse.org/-eclipselink`.

[16] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. of the ACM*, 19(11):624–633, November 1976. Also published as IBM RJ1487, December, 1974.

[17] Gavin King et al. Hibernate developer guide. Hibernate Community Documentation. `http://docs.jboss.org/hibernate/orm/4.0/-devguide/en-US/html/ch05.html`.

[18] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, August 2004.

[19] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS*, 8(2):186–213, June 1983.

[20] Cary G. Gray. *Performance and Fault-Tolerance in a Cache for Distributed File Service*. PhD thesis, Stanford University, December 1990.

[21] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symp. on Operating System Principles (SOSP)*, SOSP '89, pages 202–210, 1989.

[22] Jim N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems, an Advanced Course*, volume 60 of *LNCS*, pages 393–481. Springer-Verlag, 1978.

[23] Paul Greenfield, Alan Fekete, Julian Jang, Dean Kuo, and Surya Nepal. Isolation support for service-based applications: A position paper. In *Proc. 3rd CIDR*, pages 314–323, 2007.

[24] T. Haerder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, June 1984.

[25] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. Technical Report CMU-CS-88-120, Carnegie Mellon University, Pittsburgh, Pa., 1988.

[26] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proc. 13th PPoPP*, pages 207–216, February 2008.

[27] Hibernate. `http://www.hibernate.org`.

[28] J. Stuart Hunter. The exponentially weighted moving average. *Journal of Quality Technology*, 18:203–210, 1986.

[29] J. Jang, A. Fekete, and P. Greenfield. Delivering promises for web services applications. In *Proc. 5th ICWS*, pages 599–606, July 2007.

[30] JMeter. `http://jmeter.apache.org`.

[31] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. Coarse-grained transactions. In *Proc. 37th POPL*, pages 19–30, January 2010.

[32] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. on Database Systems*, 6(2):213–226, June 1981.

[33] L. Lamport. Towards a theory of correctness for multi-user data base systems. Report CA-7610-0712, Mass. Computer Associates, Wakefield, MA, October 1976.

[34] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[35] Butler W. Lampson. Designing a global name service. In *Proc. 5th PODC*, pages 1–10, August 1986.

[36] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. 10th OSDI*, October 2012.

[37] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *ACM SIGMOD International Conference on Management of Data*, pages 318–329, June 1996.

[38] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, October 2009.

[39] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symp. on Operating System Principles (SOSP)*, 2011.

[40] D. L. Mills. Network time protocol (version 3) specification, implementation and analysis. Network Working Report RFC 1305, March 1992.

[41] P. O'Neil. The escrow transactional method. *ACM Trans. on Database Systems*, 11(4):405–430, December 1986.

[42] Christos H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 26(4):631–653, October 1979.

[43] Peter Peinl and Andreas Reuter. Empirical comparison of database concurrency control schemes. In *9th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 97–108, 1983.

[44] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *17th ACM Symp. on Operating System Principles (SOSP)*, St. Malo, France, October 1997.

[45] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2010.

[46] David P. Reed. Naming and synchronization in a decentralized computer system. Technical Report MIT-LCS-TR-205, Massachusetts Institute of Technology, 1978.

[47] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM CSUR*, 37(1):42–81, March 2005.

[48] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *Proc. 3rd Conf. on Networked Systems Design & Implementation (NSDI)*, pages 18–31, Berkeley, CA, USA, 2006. USENIX Association.

[49] ObjectDB Software. ObjectDB 2.3 developer's guide. `http://www.objectdb.com/java/jpa/persistence/lock`.

[50] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Mike J. Spreitzer. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th ACM Symp. on Operating System Principles (SOSP)*, pages 172–183, December 1995.

[51] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. In *16th ACM Symp. on Operating System Principles (SOSP)*, pages 224–237, 1997.

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

[53] Werner Vogels. Eventually consistent. *CACM*, 52(1):40–44, January 2009.

[54] Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.