Reducing Consistency While Keeping Correctness Using SAT Solvers

Computer-Aided Program Reasoning Course Project - Dr. Samanta

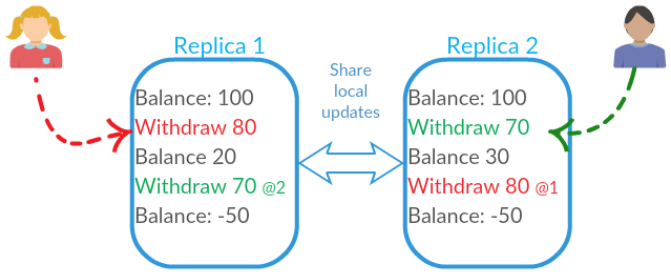
Kiarash Rahmani
rahmank@purdue.edu
https://github.com/Kiarahmani/CAPR2016

Abstract. As a result of the CAP theorem, today's distributed databases offer various levels of *weak consistency*. Application developers must pick the correct consistency level for each operation to preserve the correctness of their application. Recent works [1] successfully take this burden off the developer's shoulders. They correctly choose the weakest store offered consistency, that preserves the application-level requirements. Our work is an extension of them, in the sense that we allow users to define their own consistency requirements and our synthesis tool would implement it automatically.

Keywords: Eventual Consistency, SAT solvers, Synthesis, Weak Consistency, Quelea, Cassandra

1 Introduction

Replication is a must in the real-world web services such as those maintained by Facebook, Google or Twitter. These systems replicate data and logic across geo-distributed data centers (and also within data centers) to reduce user-perceived latency and to tolerate network partitioning and node failures.
As the running example, let's consider a replicated bank account software, that serves the clients via the host data center. In an ideal world, clients would submit request to the local replica, updates are created locally and *would become available at all other replicas immediately*. However, we know this is not the case in reality: physical network partitions are unavoidable and delay in update propagation is non-negligible. It is easy to see why this can become problematic if replicas do not synchronize operation execution:



If two users concurrently submit updates on the same replicated object (for example, in the above figure the green transaction was submitted at replica 2 before the updates from red transaction became available at replica 2), the system might become inconsistent. This is due to the *dependency of operations to the ordered history of updates available at that replica*[1]. For example, the Withdraw operation in the bank account, reads the current history of updates and decides if the money should be deducted or not.

[1] Some limited techniques can eliminate this dependency, look at [2]

Weak Consistency Levels: Above examples demonstrate the need for synchronization between replicas. However, keeping the replicated data always consistent, requires global consensus and massive communication between data stores, which reduces the overall performance. It might actually be impossible to achieve in the presence of network disconnections.
Now, one might ask, is it even necessary in all cases to guarantee **S**trong **C**onsistency for all operations? Strong Consistency guarantees that read operations always return the most recent data regardless of which node delivers the data, however, is that really required? For example, in the bank account example, we can permit the Deposit operations to perform and commit locally (with no synchronization) and the application would still be safe. In other words concurrent deposits cannot falsify the correctness of the bank account application.
In the above setting, the weakest guarantee that we must be sure of, is that *all updates become available at all replicas eventually*. This is called **E**ventual **C**onsistency and is usually offered by the underlying system. In general, any use cases in which providing an answer is more important than providing the most up-to-date answer can be served well by Eventual Consistency.
Weak Abstract Consistency levels are those that fill the gap between EC and SC guarantees. For example, **C**ausal **C**onsistency, guarantees that if an update is present in the replica, all updates that it witnessed at the time of execution -possibly originated at some other replica- are also present. In other words, two events that are *causally* related must appear in the same order at all nodes. Also, **R**ead **M**y **W**rites, guarantees that an operation always witnesses all the previous updates from its own session.

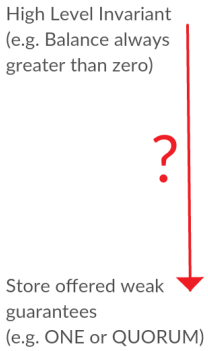
In a nutshell, at the program level, we want to be able to specify some sort of dependencies between events and expect the underlying system to preserve their order at all replicas. Consider the following two events; I *really* have to be sure that, *at all replicas* the first event is executed before the second one:

[Removes professor from friends list]

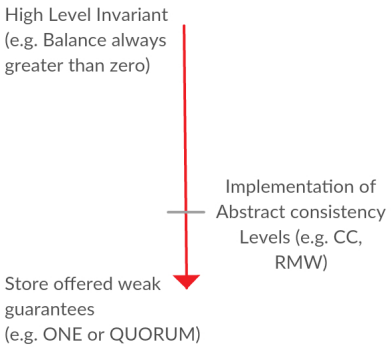
$[Posts] \rightarrow$ "My teacher is the worst, I need to drop this course!"

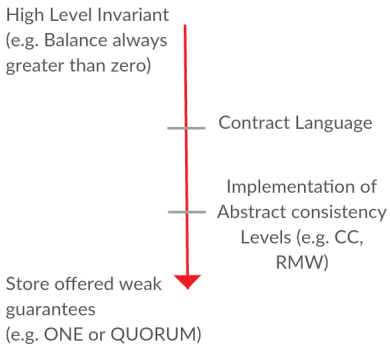
2 Problem Definition

Real world data-stores, do not offer any of the conceptually defined weak guarantees such as RMW or CC. They only let the users work with low-level notions, such as TWO (A write must be written on at least two node). As we can see there is a large conceptual gap here: the application writer only cares about the high-level invariant of the program (non-negative balance, for instance), but has to pick store-offered, low-level guarantees for each operation.



As explained earlier this gap is filled by conceptual guarantees like CC, that offer one level of abstraction for the application writers: now users can pick one of these more abstract guarantees that have been previously implemented on top of the store. We call them more abstract, because they consider updates and relationships among them, as opposed to low-level inter replica agreements.





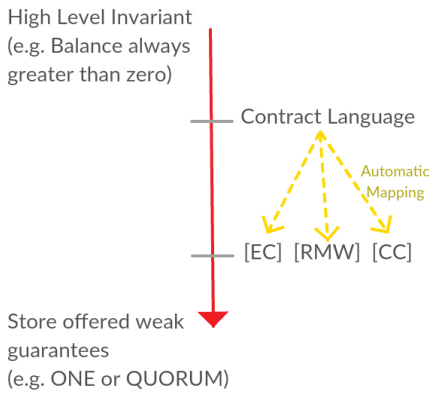
Recent developments, offer another level of abstraction: using a language of **contracts**, user can specify the *application level requirements* for each operations, and operations are automatically mapped to the correct available consistency implementation. For example following contract is mapped to CC:

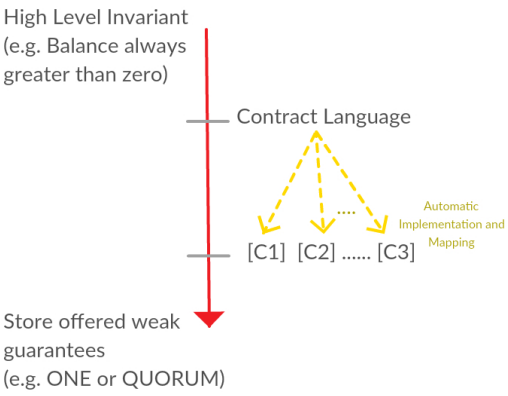
$$\forall(a : deposit)(b : withdraw).((vis(a, b)) \wedge (vis(b, \eta)) \Rightarrow (vis(a, \eta)))$$

It simply says that if the current operation η witnesses an update b, and b has witnessed another update a, it must be the case that η also witnesses a.

This obviously makes the developer's job easier, since he or she does not have to deal with the low-level consistency guarantees anymore.
However, there are still some problems remained...

Current Solution and Proposed Extension Current system, based on the given contract, *maps* each operation to some *pre-implemented* consistency level. Since the mapped level must preserve the contract's requirements, the system tends to be conservative: If $L1$ does not preserve the contract for the operation *op* but $L2$ does, *op* must be mapped to $L2$. However, $L2$ might be too strong for *op* and we lose performance because of unnecessary synchronization.
What if we could first analyze the code and the contracts and automatically synthesis the consistency levels that offer the exact guarantees required for each operation?





3 Algorithm

Well-Formed Contracts: Our first observation is that we can limit the structure of the contracts, without losing any generality. All interesting contracts can in fact, be written in the following form:

$$\forall a, \Phi_{pre} \Rightarrow vis(a, \eta)$$

which specifies that, all updates that satisfy the pre-condition Φ_{pre} must be visible to the current operation η. This intuitively makes sense, since developer is only interested in specifying what updates must be witnessed by the operation that is being executed.

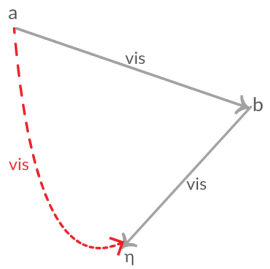
Cache Synthesis: Our idea is to first analyze the contracts and create multiple caches on top of the store, each of which is guaranteed to show the specified behavior by the contract. For example if the contract of an operation is: $\forall a, so(a, \eta) \Rightarrow vis(a, \eta)$, a cache is created for this contract that makes sure that the operation is executed only if $so^{-1}(\eta)$ is present in the cache (can be done by blocking the execution, until the required updates become available). For this project I am focusing on the important problem of figuring out the maximal set of events that the current operation can see, without falsifying user-defined correctness.

3.1 First Steps

We observed that contracts can be divided into two major groups and we deal with each separately:

Wait-Free Contracts: are those that never require blocking the operation: the cache can pick some subset of the available update and make only them visible to the current operation. For example, consider the following contract and its associated graph:

$$\forall a.vis(a, b) \wedge vis(b, \eta) \Rightarrow vis(a, \eta)$$

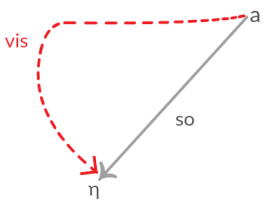


The contract specifies that if an effect b is visible to the current operation, all updates a that were visible to b, must also be visible to the current operation. However, this does not mean the current operation must be blocked until all such updates are received: we can filter those bs that does not satisfy this property and then execute the current operation. In other words, we by filtering some present updates at the replica, the current operation can see a *safe environment* based on its contract.

The important property that our filter must have, is called MAX-VIS and is defined as follows: *if the cache filters out some updates and returns S as the safe environment, there cannot be an update u in the store, that $S \cup \{u\}$ is safe.*

Waiting Contracts: Similarly, there are some contracts, that when implemented in the cache, waiting for some updates would be unavoidable:

$$\forall a.so(a, \eta) \Rightarrow vis(a, \eta)$$



This specifies that every previous update in the same session with the current operation, must be visible to this operation. This means that the current operation must be blocked until those updates become available at the local replica. Similar to the previous section, we define MIN-WAIT property for the blocking, as follows:
if cache blocks the current operation, until a set s of updates are arrived, there cannot be a smaller set s', that $R \cup \{s'\}$ is safe (R is the effects available at the replica).

3.2 Contract Analysis

By analyzing the associated graph for each contract, we realized that for all the wait-free contracts, the pre-condition part (gray section in the figures) ends with a *vis* edge pointing to the current operation. This intuitively makes sense, because a *vis* edge brings flexibility to the system: we can manipulate the pre-condition of the contract, by modifying the set of effects that are visible to the current operation. On the other hand, Waiting contracts always have an *so* edge pointing to the current operation: *so* is the session order relation, which cannot be modified by the system; as a result pre-condition cannot be changed and wait is unavoidable.

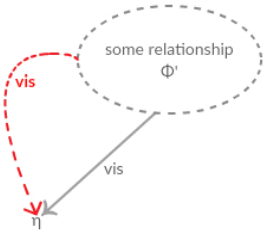


Fig. 1: Wait-Free Contracts

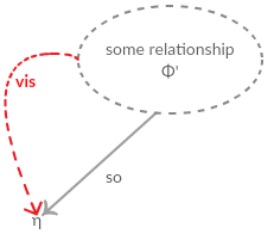
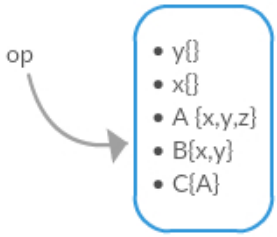


Fig. 2: Waiting Contracts

As mentioned previously, for wait-free contracts we need to find out what subset of available effects must be made visible to the current operation, and for the waiting contracts the operation must be blocked until the minimal set of required effects become available. Our synthesis is nothing but a shim layer on top of the actual database that figures out the visibility set for the current operation or possibly blocks it until all dependencies are available. Thus the main theoretical question we need to answer is how to find out the maximal *visibility* and minimal *wait-for* sets.

3.3 From Contract to SAT Formula - An Example

Let's start with a concrete example. Assume we have the following local setting: 5 effects are available at the replica whose visibility sets are recorded. For example, effect A at the time of execution has witnessed three effects: $\{x, y, z\}$. Now an operation *op* is submitted to the replica whose consistency guarantee is given by the following contract: $(vis(a, b)) \wedge (vis(b, \eta)) \Rightarrow (vis(a, \eta))$. Since this is a wait-free contract, there is no need to block *op*. However, the shim must automatically figure out what subset of $\{x, y, A, B, C\}$ is safe to be shown to *op*.



For this example, we can see that the maximal visibility set for op is $\{x, y, B\}$; because if we include A in the visibility set, that means $vis(A, \eta)$ holds, and since $vis(z, A)$ also holds, based on op's contract, $vis(z, \eta)$ must also hold, which is not possible at the moment, because z is not available at the replica yet. Moreover, since A is not in the visibility set, with a similar reasoning C also cannot be included.

We successfully, translated this problem into a Max-SAT problem as follows: Find an assignment for each variable that satisfies the following hard constraints and maximizes the number of soft constraints. All variables that are assigned true must be included in the visibility set.

$$\begin{array}{c|c} \textbf{Hard} & \textbf{Soft} \\ \begin{array}{c} z : (\neg z) \\ x : (True) \\ y : (True) \\ A : (\neg A \vee x) \wedge (\neg A \vee y) \wedge (\neg A \vee z) \\ B : (\neg B \vee x) \wedge (\neg B \vee y) \\ C : (\neg C \vee A) \end{array} & (A) \wedge (B) \wedge (C) \wedge (x) \wedge (y) \end{array}$$

The soft constraints clearly represent the presence of the effects in the visibility set, and the hard constraints are built from this particular example's contract. Each line represents constraints on one of the effects: z is not available yet, so it cannot be True, x and y have no constraint because their original visibility sets were empty. For the rest, each clause represents one visibility relation that according to the contract must be satisfied. For example, the only clause for effect C, clearly states that either C must not be true, or A must also be true (= present).

3.4 SAT Formula Generator

Here we explain an algorithm to craft a Max-SAT instance similar to the previous example, given a contract. For the sake of simplicity, here we explain the algorithm for a simpler and more restricted type of contracts. We only consider contracts in which each effect appears only in at most two relations - i.e. no multi-edge node in the associated graph. The algorithm can easily be extended to support multi-edge graphs also -it just requires more book-keeping.

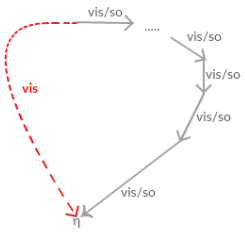


Fig. 3: Simplified Contracts

Our algorithm is a graph traversal procedure that starts from η and goes up until reaches the last node before the red vis edge. Starting from the base node, at each iteration, it covers one edge and computes the set for the next node and repeats for all edges. Later, it computes the hard and soft constraints based on these sets: the soft constraints are the conjunction of all present effects at the replica and assigned once, but hard constraints are modified at each iteration.

```
Input: list of edges [E_1, E_2, .... , E_k]; set A of available effects
Output: Mapping from A to sets of hard clauses
-------------------------------------------------------------------------
H[1] = A;
for (i in [1...k-1])
  if (E_i.type() == vis)
    H[i+1] = union of x.visibility() for all effects x in H[i]

  else if(E_i.type() == so)
      H[i+1] = union of x.sess_precedents() for all effects x in H[i]

//Now list H contains all effects at each node that satisfy the relation to
    the base effect at that node specified by the subgraph.

for (i in [1...k-1])
  if (E_i.type() == vis){
    for effect e in H[i]
      if (e is present in the replica)
        for all a in e.visibility()
          add clause (not e \/ a) to the hard constraints of e
      else
        add (not e)
  }
  else if(E_i.type() == so)
    for effect e in H[i]
      if (e is present in the replica)
        for all a in e.session_order()
          add clause (not e \/ a) to the hard constraints of e
      else
        add (not e)
```

Listing 1.1: Algorithm

We manually computed the Max-SAT formula for a number of a number of contracts and the algorithm sketch above seems to work fine. In a nutshell, it just traverses the graph upward, and at each step figures out all effects that satisfy the relation given by the traversed subgraph to the base effect η. When covering an edge of type r, for each of the effects e present at the set associated with the current node, it finds out all effects a that satisfy relation r with e, and adds a clause to the hard constraints of e. The clause is of the form $(\neg e \vee a)$ which obviously states that if e is going to be present (i.e is true), a must also be present.

Now, we have a reduction algorithm from finding max-vis set to max-SAT. Of course, it needs to be generalized to support all sorts of graphs and then implemented. The synthesizer must take a contract and return a function F. Function F is a black-box in our tool that magically (actually, after calling the SAT solver) tells us what subset of the available effects at the replica must be made visible to the current operation.

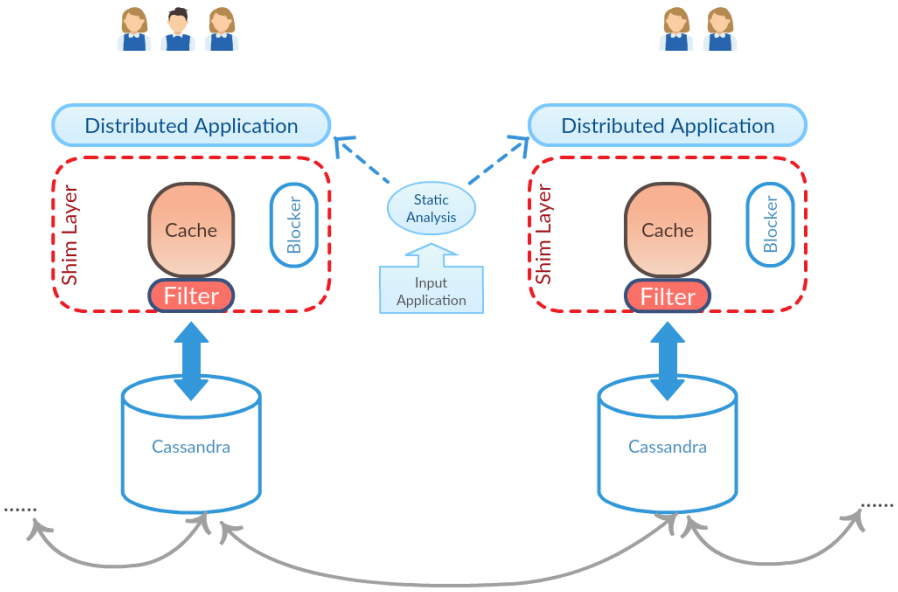
4 Implementation

In this section, we will discuss the details of Quelea [1], a tool developed at Purdue university in 2014, and then introduce modifications done by us, to implement the new algorithm.

4.1 Quelea

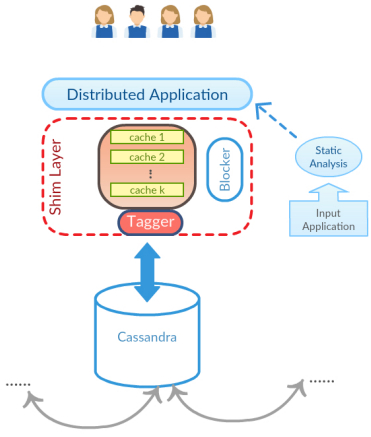
Quelea is developed as a Haskell program that runs next to each database (Cassandra) node. It also provides users with the language and frameworks to write their own applications and define

their requirements. The following figure shows the outline of the structure, we will use it later to explain our new implementation.



The original Quelea, as it is depicted above, includes two separate parts: Static analyzer (light blue section) and the run-time consistency manager (red dashed part). The analyzer first makes use of Z3 to map each operation to one of the consistency levels (EC, CC, SC), based on the user's requirements. The resultant application (with each operation mapped to a consistency level) is then executed on top of the run-time shim layer. The shim layer maintains a causally consistent cache for CC operations and implements a global lock for SC ones. As explained in the first chapter, this system only implements one level of weak consistency (CC) between EC and SC. The filter in the shim is hardcoded to only filter out those updates (from other nodes) whose *causal dependencies* are not yet arrived. In other words, the cache always contains a causal cut of the dependencies graph. The blocker's job is to halt the request, until all dependencies are arrived - it checks for dependencies after the next cache refresh.

4.2 Modifications



As explained earlier, we want to extend Quelea to support finer grained consistency levels. To do so, instead of hard coding a single CC cache in the shim, we use a tagging system to implement multiple number of *logical caches*.

The static analyzer, similar to Quelea, first labels each operation with a consistency level, but unlike Quelea here we are not limited to only 3 labels; in fact, we can have as may consistency levels as the number of operations. Our system also synthesizes a shim layer for the given application (as opposed to Quelea, where shim layers are hard-coded the same for all applications). The synthesized shim layer, maintains multiple number of logical caches that are guaranteed to preserve a specific consistency requirements. Each operation is executed at its own cache, which allows us to handle operations

with different consistency requirements in the same shim. The following figure shows this idea. When the cache is being refreshed, the tagger's job is to mark new updates with a set of tags. If an update is tagged with a certain consistency level, it is included in that level's associated logical cache. Obviously, an effect eventually ends up at all caches, but it might take some time.

It's the tagger's job to make sure that only effects can have a tag, whose dependencies also have that tag - Note that here we mean more general types of dependencies, not just closure of hb relation (i.e. causal dependency). Here, dependencies are arbitrary relations defined by the application writer. Now it is obvious why the same parts in the old and the new systems are called *filter* and *tagger* respectively: the old system had to take care of effects going into only one logical cache by filtering them out, which can be thought of as a special case of tagging with only one consistency level.

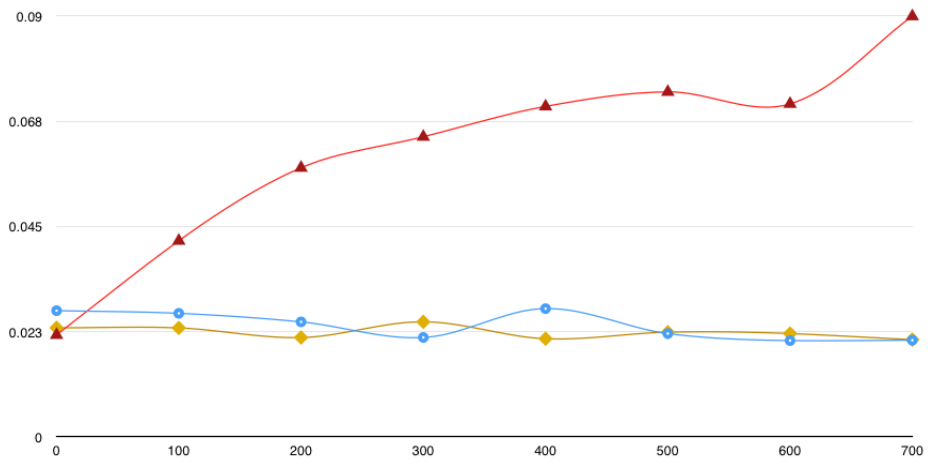
4.3 Using Z3 and Haskell Bindings

Our tool is based on the algorithm explained earlier, which transforms the question of finding the maximal set of consistent updates, to a Max-SAT formula. Instead of creating the consistent subset at the arrival time of each operation, we created the aforementioned shim layer, that periodically refreshes the logical caches and determines the maximal set of effects that can have a tag (of course, this is done for all tags). The tool might end up in operations seeing slightly out-dated states of the cache, however solving a max-SAT problem at each operation execution is not practical, specially when the cache size becomes large.

There are multiple Z3 bindings available for Haskell, however we found SBV package suitable for all SMT based verifications, including the max-SAT problem. It has a very interesting high level interface that allows programmers to write symbolic programs and assert their constraints and receive models and even proofs. It can also work with different SMT solvers at the same time and return the best result as the output.

4.4 Empirical Results

In order to support our proposal, that finer grained consistency levels will allow users to pick the right levels of synchronizations which would result into performance gain, we created a pathological -yet realistic- environment and compared the performance of CC implementation and RMW. We launched 3 local Cassandra nodes and enforced artificial delays between messages passed among them. As the graph below shows, CC operations (red line) would be delayed longer when the communication is slower, but RMW (blue line) remains almost constant and behaves as fast as EC operations (yellow line). The details of this experiment can be found at this blog post.



We have also completed a simple version of the compiler, that can handle simplified contracts of the form of figuure 3. It takes the current state (tags) of the cache, a new update effect, and a sequence of *vis* and *so* relations (which represents the input contract) and returns the new state of the cache. The new cache, includes the new effect with refreshed tags for all other effects (After adding this new effect, some other pre-existing effects can have more tags, since their dependencies are now present).

We gave Monotonic Writes contract [3] as the input to the compiler. The MW guarantee requires all effects in the session-order with any effect visible to the current operation, to be visible as well. The test results for the bank account application are collected in a table, the last column of which, shows the average operation latencies for different number of effects.

Number of Effects	test1	test2	Avg
10	0.0220	0.0226	0.0223
20	0.02721	0.0275	0.027355
50	0.0345	0.0339	0.0342
100	0.0420	0.0408	0.0414
200	0.0559	TO	0.0559
500	TO	TO	⚠
1000	TO	TO	⚠

Table 1: Latency vs Cache Size

The system was unable to produce any results for cache sizes larger than 200, however the behavior was manually checked and found to be correct. We find this promising, since we now have a correct working system and can proceed to engineering issues. Two ideas to get there are the followings:

Creating Contexts: Our initial implementation uses a naive approach, since it discards all the tags previously given, and re-tags all effects from the scratch at each cache refresh. We need to change this in the next versions, since we know that the set of tags is a grow-only set, i.e. once a tag is given to an effect (which means the required dependencies are present) it can't be taken back. We will re-engineer the system to allow preservation of a context at each step and we expect noticeable performance increase b y doing so.

Garbage Collection: We are also planning to implement a garbage collection procedure to maintain a maximum cache size. The procedure requires semantical help from the programmer: for example, in the bank account application, programmer must also indicate the *summarization* method, which replaces a large number of effects (withdraw or deposit) with a single deposit of proper amount. We are confident that the large cache size problem can be fixed this way, and will make the system more competitive to the current ones.

5 Conclusion

In this project, we are generalizing the notion of *dependency*, used in distributed systems. Current systems only consider causal consistency, which guarantees the presence of all updates that causally happened before the current operation, at the time of its execution. We have successfully found examples (shopping cart application) that have weaker requirements, but are mapped to causal consistency, since that is the only level available between eventual and strong consistency.

We implemented the idea in a tool named Quelea by extending its ability of maintaining a causally consistent cache into a multi-cache setting. We also coded a simple version of the synthesizer that given a set of consistency requirements -called contracts- creates caches that each maintain the required level of consistency at all time. The contracts are written in a language provided to the application developer. *We have shown that the language is expressive enough to write all known famous consistency requirements in.*

The current implementation, successfully presents the correctness of our ideas, however lacks the performance requirements of real-world scenarios. Part of this is due to the limits set by the SMT solvers. We believe that since the problem we are dealing with, is always of the exact same form, we might actually be able to come up with tools that would be faster than transforming our problem into a Max-SAT and feeding it to a general purpose solver.

Researchers are actively working in the area of eventually consistent datastores, and there are works done on verification and performance of weakly consistent systems. However, as far as our knowledge goes the idea of synthesizing finer grained consistency levels has never been discussed before and we are planning to continue our work into a conference paper in the next few months. The implementation can be accessed at: https://github.com/Kiarahmani/Quelea

References

1. KC Sivaramakrishnan, G. Kaki, S. Jagannathan: Declarative Programming over Eventually Consistent Data Stores. PLDI '15.
2. M. Shapiro, N. Preguia, C. Baquero, M. Zawirski : A comprehensive study of Convergent and Commutative Replicated Data Types. RR-7506, INRIA. 2011, pp.50.
3. D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, B. Welchi : Session Guarantees for Weakly Consistent Replicated Data
4. S. Burckhardt et. al: Replicated Data Types: Specification, Verification, Optimality. POPL'14.
5. S. Gilbert, N. Lynch: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33(2):51-59, June 2002.
6. M. Lesani, C. Bell, A. Chlipala: Chapar: Certified Causally Consistent Distributed Key-Value Stores. POPL'16.
7. G.DeCandia et.al. : Dynamo: Amazon's Highly Available Key-value Store. SOSP'07.