Supporting Distributed, Concurrent, One-Way Constraints in User Interface Applications

Krishna A. Bharat and Scott E. Hudson
Graphics, Visualization & Usability Center
Georgia Institute of Technology, Atlanta GA 30332-0280
E-mail: {kb, hudson}@cc.gatech.edu

ABSTRACT

This paper describes *Doppler* a new, fast algorithm for supporting concurrent, one-way constraints between objects situated in multiple address spaces. Because of their declarative nature, convenience, low amortized cost, and good match to interface tasks, constraints have been used to support a variety of user-interface activities. Unfortunately, nearly all existing constraint maintenance algorithms are sequential in nature, and cannot function effectively in a concurrent or distributed setting. The Doppler algorithm overcomes these limitations. It is a highly efficient distributed and concurrent algorithm (based on an efficient sequential algorithm for incremental, lazy updates). Doppler relies solely on asynchronous message passing, and does not require shared memory, synchronized clocks, or a global synchronization mechanism. It supports a high degree of concurrency by efficiently tracking potential cause and effect relationships between reads and writes, and allowing all causally independent operations to execute in parallel. This makes it scalable, and optimizes reads and writes by minimizing their blocking time.

KEYWORDS: Constraint Maintenance, Incremental Update, Concurrency, CSCW, Distributed Asynchronous Algorithms, Causal Consistency, Partially Ordered Time.

1 INTRODUCTION

As the need for multiple sites, and rapid partnering of teams with specialized skills across and within organizations increases, distributed work groups are becoming increasingly common. Members of a group, say in a design context, will share access to a collection of objects. Each person might have exclusive access to a logical portion of the collection, as well as shared access to other portions. However, changes they make, even in their local areas, may still have wide ranging effects. When a data-value of interest is affected by a change elsewhere, it is important that all local views be updated automatically. This, in effect, is a distributed constraint system.

Propagating each update over the entire shared object-space can take a significant amount of time due to communication overheads. In order to provide perfectly synchronized views, it would be necessary to wait for the update to propagate throughout the system after each change. Since reads and writes operate on a common domain, they will need to wait for each other to complete to avoid collisions. This implicitly causes users to take turns. With even modest communication delays, this would make it difficult to achieve good interactive performance, and the approach will not scale.

This was supported in part by the National Science Foundation under grant IRI-9500942

Fortunately, in most distributed sessions, users are willing to trade synchronization for interactivity; in practice for small delays they may not even notice a lag between the individual views. While individual local views need to be totally synchronized with local activity, they can afford to be synchronized with changes at other sites in a somewhat looser sense. One is generally willing to let the changes due to other users propagate with some delay, provided logically consistent views are seen at all times. In particular the following guarantees need to be met:

- Each participant sees changes in the order they occur.
- Each participant always sees the cause of a modification before the effect.
- Whenever a colleague communicates with you, your view shows every change that theirs does.

Constraint systems have been heavily studied as user interface specification techniques, and have been used in a number of systems to support a range of different user interface implementation tasks (for example [2, 4, 5, 7, 8, 9, 10, 11, 12, 15, 18, 19]). See [21] for a full discussion of the benefits of constraints. Constraint systems are uniquely suited for supporting multiple views of shared data, since they allow the relationships between data and user interface presentation to be described declaratively. Based on these descriptions, a constraint maintenance system is typically able to update (multiple) individual views automatically in response to any and all changes in underlying data objects.

As user interface applications start to become network aware and support collaborative work, they will be increasingly be called upon to operate in a concurrent and distributed fashion. Recent UI toolkits have begun taking a multi-threaded approach (e.g. Trestle[17] and Fresco[20]). Unfortunately, the update algorithms used by existing constraint systems were designed to be sequential, and cannot typically operate in a multi-threaded (concurrent) or distributed setting. (An exception is [11], which is distributed, but does not support substantial concurrency.) As a result, presently, the substantial advantages of constraints cannot be applied to distributed applications.

The Doppler constraint propagation algorithm described in this paper is based on a slightly simplified version of the very efficient sequential algorithm described in [13]. Like its predecessor, Doppler is both incremental and lazy. In addition, the algorithm is inherently multi-threaded, and is designed to allow a very high degree of concurrency. This will allow large numbers of users to operate simultaneously, as well as make it possible to take advantage of the performance improvements offered by multi-processor machines. Finally, the algorithm can be fully distributed. It does not rely on shared or synchronized clocks, and does not require shared memory.

Section 2 describes the application programmer's view of a distributed constraint system. In Section 3 we present a very brief overview of the sequential update algorithm this work is based on. In Section 4 we describe the intuition behind the algorithm, and evolve Doppler from the base sequential algorithm by progressively increasing concurrency. Sections 5 and 6 give the details of the algorithm. In Section 7 we address performance issues and sketch a proof of optimality in the level of concurrency. Finally, in Section 8, a prototype implementation is briefly discussed, and Section 9 provides conclusions.

2 PROBLEM STATEMENT

Constraint evaluation algorithms typically operate on the basis of *dependency graphs*. In a dependency graph, *nodes* represent values in the system, while *edges* represent dependencies between the nodes. If a constraint of the form `A = B + C` appeared in the system, then there would be edges from `B` to `A` and from `C` to `A` in the dependency graph. This represents the fact that `A`'s value might need to be updated if either `B` or `C` changed value.

In our model, the dependency graph is distributed. This graph is shared by a set of threads, possibly in different address-spaces.

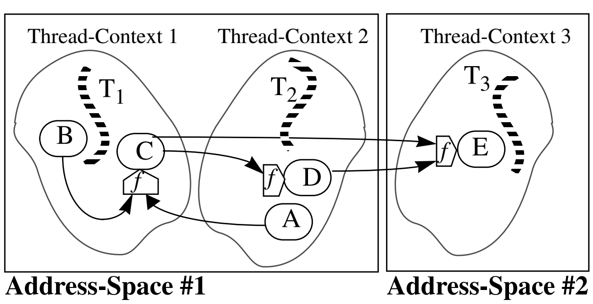

Figure 1: Distributed constraints between three threads

The set of nodes in the graph is partitioned amongst the threads so that each thread has exclusive access to a subset of the nodes, which are said to be in its *context*. There are two types of nodes: constrained and unconstrained. An *unconstrained node* (also known as a *leaf* node) may be written by the thread that owns the node. A *constrained node* (also known as an *interior* node) may never be written to. Instead it has an associated constraint that is responsible for maintaining the value of the node as a function of a set of other nodes, known as *input nodes*. In general, input nodes may reside in contexts other than that of the constrained node. Hence while a node in the distributed dependency graph is always embedded within a thread-context, an edge may run from one thread-context to another. Figure 1 shows constraints between three user- level threads in two address-spaces. Nodes C, D and E are said to be *downstream* from node A (in the direction of data-flow), and may get new values when A gets updated.

The application programmer's interface to the local constraint system consists of:

1. Commands to *modify* the structure of the graph:
   - **`CreateLeaf(v, cbk)`**, returns a new node with initial value **`v`**. The callback procedure **`cbk`** is an optional parameter. If specified, whenever the node's value changes, **`cbk`** is called with the new value as argument. Notification callbacks are used to simulate eager evaluation semantics and to communicate with "external" entities (such as window systems).
   - **`DelLeaf(n)`**, deletes the leaf node **`n`**. This is legal only if no other node depends on **`n`**.
   - **`n.addConstraint(cfunc, nlist)`**, adds a constraint to leaf node **`n`** making it an interior node. **`cfunc`** is a function that can compute the constraint's value, and **`nlist`** is the list of input nodes that it operates on.
   - **`n.delConstraint()`**, deletes the constraint associated with interior node **`n`** making it a leaf. The last value propagated to **`n`** becomes the initial value of the leaf.
2. Methods to *read* and *write* nodes:
   - **`n.read()`**, returns the value of node **`n`** (after possibly recomputing it).
   - **`n.write(v)`**, writes the value **`v`** to (leaf) node **`n`**.

Our main focus is on speeding up reads and writes, which are global operations on the graph and directly affect performance. In the initial part of our discussion (Sections 4-5) we shall describe how reads and writes are implemented in the context of a *static* graph. In section 6, we extend it to *dynamic* graphs where the structure can change concurrently with reads and writes.

3 BASE SEQUENTIAL ALGORITHM

Doppler is a concurrent and distributed algorithm based on a simplified version of the sequential algorithm for incremental and lazy updates presented in [13] and used in the Penguims [15] and Eval/*vite* [14] systems. Specifically, the simplifications remove optimizations for early quiescence and non-strict functions (which we plan to reincorporate into a later version of Doppler).

In this simplified base algorithm, each node has:

- a *value*, to store the latest value assigned to the node,
- an *out-of-date* flag, to indicate if the value is current or out-of-date,

- an *output-list*, containing nodes that depend upon it, and
- *read* and *write* methods.

If it is a constrained node, it has in addition:

- a *constraint function*, and
- an *input-list*, containing nodes that it depends upon.

Upon a *write*, nodes that are downstream from the node that was written are invalidated by setting their out-of-date flag (also known as adding an *OOD mark*). An OOD mark represents the fact that a value may not be consistent with its defining equations, and may need to be recomputed before the value is used.

To avoid multiple updates of values that are not actually used, the algorithm takes a lazy approach, letting old values with OOD marks remain in place until they are actually requested. Because of this, the OOD mark can be thought of as a place holder for the actual, consistent value. Whenever a read of a node occurs, the OOD flag is checked. If it indicates that the node is up-to-date, then its current value is simply returned. If, however, the OOD mark is present, then the value is recomputed by requesting the values of its inputs and then executing its constraint function (this may in turn recursively request other values, etc.). We call this process of replacing an OOD mark with a consistent value, *refreshing* the value.

4 INTUITION

In this section we motivate our approach by illustrating the abundant potential for concurrency in a distributed constraint graph. In the process, we identify situations in which increasing concurrency can interfere with the read/write semantics of the program, and derive consistency requirements for correct execution with maximum concurrency.

4.1 Distributed Sequential Algorithm with Global Synchronization - "Copernicus"

A direct extension of the single-threaded algorithm would involve serializing all reads and writes, by *locking the entire dependency graph* during each operation. This requires global synchronization. All operations first contact a central lock-server, which queues up requests as they come, and grants access on a first-come-first-served basis. Given this approach, there will be no concurrency whatsoever.

4.2 Distributed Concurrent Algorithm with Global Synchronization - "Kepler"

In our first step to improve concurrency, we let reads execute concurrently with other reads, and writes execute concurrently with other writes, with *node-level locking*.

It is still not possible to execute reads and writes concurrently. This is because it is hard to predict the extent of changes caused by a write. If a write and a read executed together, the read may return an incorrect value, based on a partially invalidated graph. Hence we still need the central scheduler. The graph sees an alternating sequence of read and write phases. In a read phase successive reads execute together, and in a write phase successive writes. Assuming a fair scheduler, whenever reads and writes arrive alternately we would have the execution in Figure 2, which is effectively sequential. (In the figure, W_1, R_1,and W_2, represent a write, a read, and a write performed by threads T_1, T_2, and T_3, respectively.)

The blank portion of each operation represents the interval between the initiation of the operation and the acquisition of the lock. In this interval, the user-thread does no useful work and the user may see the interface freeze. As the number of users increase, the initial wait-time for each operation will increase as well, and the interface will progressively become sluggish. This approach clearly does not scale well.

Another disadvantage of this scheme is that it requires global synchronization. Every operation will first need to acquire a *shared global lock* before it can begin, resulting in an additional round-trip message delay (not shown).

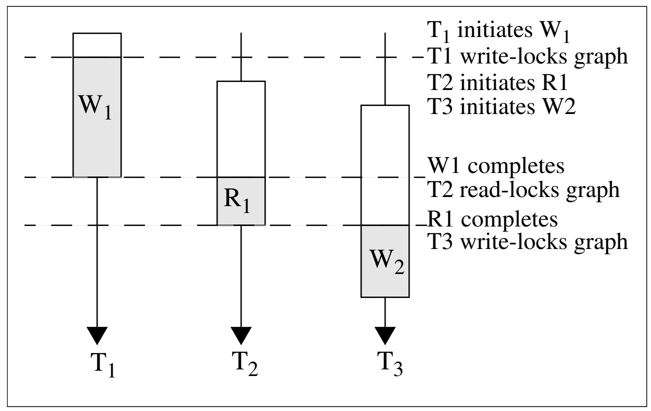

Figure 2: Alternating reads and writes are serialized

An easy improvement would be to make writes non-blocking, so that they can be propagated by background threads. This eliminates the delay during writes, but reads are still slow because they need to wait for writes to finish. The execution time of writes is large since they wait for reads. We call this forerunner of our algorithm, Kepler.

4.3 Distributed Concurrent Algorithm without Global Synchronization - "Doppler"

Ideally we would like reads and writes to execute concurrently as shown in the figure below. Also, we would like a truly asynchronous algorithm, allowing operations to execute without contacting a central synchronizing entity.

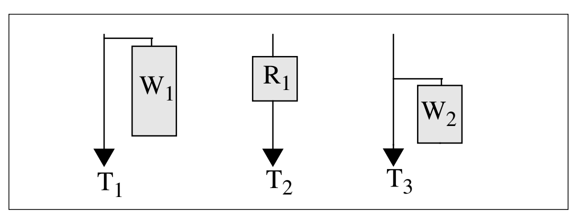

Figure 3: Ideal case with maximum concurrency

While we would like *all* reads and writes to execute concurrently as in the above figure, we must ensure at the same time that the execution respects cause-and-effect relationships between operations. Consequently *some* reads and writes will need to be delayed to preserve correctness. For example, there are situations when a certain write *must* be seen by a read for consistency, implying that the read must be ordered *after* the write. This is because a read is unable return a value that reflects (part of) a concurrently executing write, since the full effect of a write is not known until it finishes. In terms of concurrency this is the best we can do. If the operations were not ordered so, they would violate causality requirements, and threads would read inconsistent or incorrect values.

4.3.1 Causality. Causality stated simply says,

- A read must reflect every write that could potentially have caused it. In general, since we do not have specific knowledge of the computation being performed, we must assume that any write *known* to the thread at the time of the read could have caused it.
- If a read reflects a certain write, *w*, it must also reflect every write that could potentially have caused *w*. Again this means it must reflect every write that was *known* to the writing thread when it executed *w*.

When we speak of knowledge in the context of threads, we use the term in its broadest possible sense. If a thread perceives some consequence of a write, albeit in an indirect manner, it is said to have knowledge of the write. The best we can do is get an over-estimate of the knowledge that threads have of writes in the system. To do this, we conservatively track information flow in the system. A thread obviously knows about every write it has initiated. When a thread communicates with another, we shall assume that it could have passed on information about all the writes it knows about.

Figure 4 shows two examples of *read-write* causality. In case i (the degenerate case), the thread initiates the write W_1 and then executes the read R_1. The thread would naturally expect R_1 to reflect W_1. Hence the read is made to block.

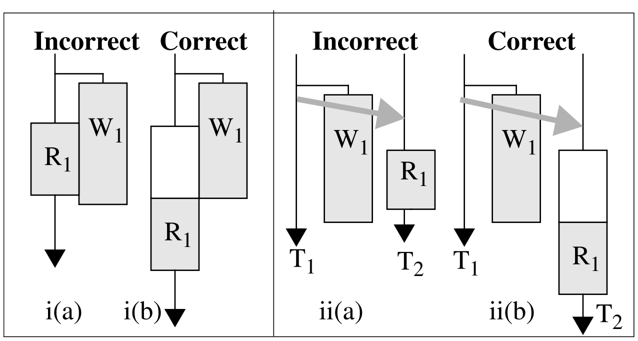

Figure 4: Reads that must follow causally earlier writes

In case ii, the read is executed after T_1 sends a message (or executes a remote procedure call) to T_2, denoted by the gray arrow. This communication constitutes a causal link, since it could have conveyed information about W_1 that prompted T_2 to execute R_1. Hence, R_1 must wait for W_1 to finish, and then compute its value.

Figure 5 shows an example of *write-write* causality. As before the gray arrow represents a message between threads. The write W_2 may causally depend on W_1 because a message travelled from T_1 to T_2 in the interval between the writes. In a more general case there could be a sequence of messages from T_1 to T_2, through a number of other threads, which could causally influence the execution of W_2.

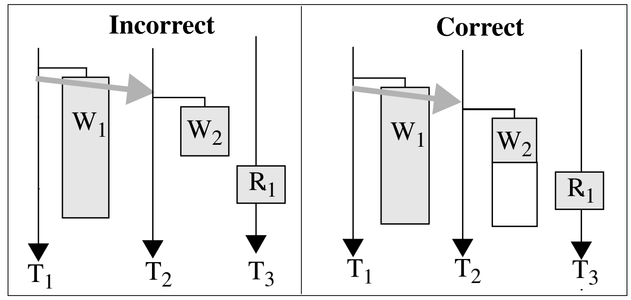

Figure 5: A situation in which a write needs to wait for a causally preceding write

This causal link does not prevent W_2 from executing concurrently with W_1 but it does prevent it from being seen before W_1. If it completes early (as in the left half), then the read R_1 could potentially reflect W_2 without reflecting W_1. This would cause the thread T_3 to see the effect without seeing the cause, which would be inconsistent. Hence in a correct execution the commit point of W_2 will be delayed until W_1 finishes and R_1 will reflect neither. A degenerate but very common case of causality is when both writes are executed by the same thread. The second write could be based on the assumption that the first had occurred.

4.3.2 Causal Consistency. Doppler delivers causal consistency. Causal consistency is based on Lamport's concept of potential causality [16]. It has been widely studied in distributed systems (e.g. causal broadcasts [3] and causal memory [1]), and is accepted as a good combination of concurrency and correctness. It allows more concurrency than conventional sequential consistency, wherein all operations are ordered linearly (often in an ad hoc manner). At the same time cause and effect relationships are respected.

Based on the discussion in Section 4.3.1, we can define causal consistency in the context of a distributed dependency graph as follows:

- Reads reflect all causally preceding writes.
- If a read reflects a write W, then it also reflects all writes that causally precede W.

Formally, the "causally precedes" relation on the operations, read, write, message-send and message-receive is the transitive closure of the relation '$\rightarrow$' in which:

- **op1** $\rightarrow$ **op2** if they are executed by the same thread, and **op1** was executed before **op2** in program order.
- **op1** $\rightarrow$ **op2** if **op1** is a message-send and **op2** is the corresponding message-receive.

4.3.3 *Causally Consistent Execution.*

Since reads cannot reflect concurrently executing writes, causal consistency places the following restrictions on the execution of reads and writes:

Requirement 1: A read must wait for all causally preceding writes to complete before it can begin.

Requirement 2: A write must wait for all causally preceding writes to complete before it can complete.

Requirement 3: A read must reflect all causally preceding writes.

Requirement 4: If a read sees the effect of a write W, it must also see the effect of every write that causally precedes W.

These requirements may seem stringent at first but they are not. In fact these requirements are met by the Kepler algorithm as well. Although causality between operations in the same thread is common, there is typically no causal dependence between concurrent operations in separate threads. In the latter case, Doppler allows reads and writes to execute simultaneously whereas Kepler does not. Hence Doppler represents a significant improvement in *inter-thread concurrency* over Kepler. Also, since there is no global synchronization in the algorithm and all communication is asynchronous, operation execution times are significantly reduced.

Although we have assumed till now that the dependency graph's structure is static, the above discussion can apply to dynamically changing graphs as well. For the purposes of consistency maintenance, a change to the graph structure can be regarded as generalized case of a write (see Section 6).

4.3.4 *Partially-Ordered Time.*

In a distributed system, due to the absence of shared or synchronized clocks, it is hard for any executing thread to figure out the order in which distributed operations were initiated in real-time. The *only* case in which it is possible to accurately infer that one operation began before another is when the first causally precedes the second (using program semantics).

In the absence of shared or synchronized clocks, if two operations are not causally related then,

- There is no way to tell which was initiated first in real-time.
- The two operations must have been initiated by separate threads, since all operations initiated by a given thread are causally ordered.
- They can be ordered arbitrarily as their effect on the graph is commutative. This is because writes to the same node are always causally related, and writes to distinct nodes are commutative.

For a causally consistent execution, we need to keep track of causal relations between operations. To do this efficiently, we map the initiation-time of each operation to a point in *partially-ordered time* (specifically a representation of time ordered by the "causally precedes" relation defined in Section 4.3.2). Whereas points in linear time are scalars, points in partially-ordered time may be conveniently represented as vectors. Each thread keeps an estimate of partially-ordered time in a vector called *GlobalClock*. If there are n threads in the system, partially-ordered time can advance along n dimensions, and hence GlobalClock is a vector of size n. GlobalClock[i] will represent a count of significant events initiated by thread i. When an operation is initiated, it is tagged with the current value of GlobalClock, also known as the operation's *timestamp*.

The following relations on vectors: $<$, $\leq$, $=$, $\geq$ and $>$, are applicable to partially-ordered time-vectors as well. For example $T_1 \leq T_2$ means that every element of T_1 is less than or equal to the corresponding element in T_2. $T_1 < T_2$ is similar but requires that at least one element in T_1 be strictly lesser than its peer. $T_1 = T_2$ applies when all elements are equal.

These relations have the same meaning in partially-ordered time as in linear time. For example if e_1.timestamp $\leq$ e_2.timestamp then we can infer that the event e_1 happened before or at the same time as event e_2. When none of the above relations apply, the time-vectors are said to be incomparable, and represent points in time which may be treated as concurrent.

We will employ the following convention when comparing operations: We will use *before* (or *earlier*) to represent the $\leq$ relation on timestamps, and *strictly before* (or *strictly earlier*) to represent the $<$ relation; similarly *after* (or *later*) to represent $\geq$, and *strictly after* (or *strictly later*) for $>$.

We will often need to compute the *earliest* point in time *after* a set of time-vectors. For this purpose we define the operator T_{max}. If S is a set of time-vectors, $T_{max}(S)$ is the unique smallest vector such that for every vector t in S, $t \leq T_{max}(S)$. T_{max} is easy to compute. It is simply the element-wise maximum of all the vectors in S.

The only condition we impose on the kind of partially-ordered time defined by GlobalClock is that it respects real-time order and reflects all causal dependencies. Given this assumption we can redefine the requirements on the basis of timestamps:

Requirement I: A read must wait for all *earlier* writes to complete before it can begin.

Requirement II: A write must wait for all *strictly earlier* writes to complete before it can finish.

Requirements 3 and 4 may be combined.

Requirement III: A read must reflect all *earlier* writes and none other.

If each operation needs to first contact a central scheduler (causing partially-ordered time to become linear), the above requirements will give us the same execution as in Kepler.

4.3.5 *Measuring Partially-Ordered Time.*

We now show how GlobalClock can be maintained in a manner that best suits our needs. Threads cooperatively maintain GlobalClock using a *gossip* technique.

To begin with , all elements of GlobalClock are set to 0. Every operation that a thread initiates causally succeeds all its previous operations. Hence every time thread i initiates a write, it increments GlobalClock[i] and uses the resulting value of GlobalClock as the timestamp. Reads are not significant events and do not advance partially-ordered time, because no action causally depends upon a read. The existing value of GlobalClock is used to timestamp the read.

If a thread i sends a message to thread j, then all operations that i initiated before sending the message should be ordered *before* any operations initiated by j after receiving the message. Hence thread i will send GlobalClock_i along with the message. On receipt j's constraint system will advance its GlobalClock so that it is ordered *after* GlobalClock_i. It sets its GlobalClock to $T_{max}(\{\text{GlobalClock}_i, \text{GlobalClock}_j\})$.

Thus far only causal dependencies have been represented. In practice, we would also like to compensate for external cause-and-effect relationships as well. These are due to alternate channels of communication between users. If user1 performed an action and then calls user2 over the phone, causing user2 to perform a second action, there is an external causal link between the events that the system is not aware of.

To compensate for this (maintain correctness) and to stay close to real time in general (improve responsiveness), whenever two operations are not causally related but are separated by an appreciable interval in real time, the algorithm will attempt to order them accordingly in partially-ordered time. For this purpose, each thread maintains a running estimate of the set of completed writes in the system. When a local write finishes it is added to the estimate. Also, constraint systems synchronize periodically by transmitting their estimates to their peers using *sync* messages. When a sync message is received, the constraint system merges the peer's estimate into its own.

Each thread's constraint system ensures that its operations are *after* all known completed writes by ensuring that GlobalClock $\geq$ T_{max}({timestamps of known completed writes}) at all times. Typically GlobalClock exceeds this quantity because it will also reflect some writes in progress; specifically causally preceding writes in execution.

Since $T_{max}(\text{set1} \cup \text{set2}) = T_{max}(\{T_{max}(\text{set1}), T_{max}(\text{set2})\})$, the set of known completed writes need not be maintained explicitly. Instead, the vector *CompletedWrites* is used to keep track of the value, T_{max}({timestamps of known completed writes}). When a write W completes, CompletedWrites is set to T_{max}({CompletedWrites, W.timestamp}).

When a message is received bearing a peer's estimate, $\text{CompletedWrites}_{peer}$, the local constraint system sets:

- CompletedWrites to the value,
  $T_{max}(\{\text{CompletedWrites}, \text{CompletedWrites}_{peer}\})$
- GlobalClock to the value,
  $T_{max}(\{\text{GlobalClock}, \text{CompletedWrites}_{peer}\})$

User-level (causal) messages carry CompletedWrites as well.

With the above, we have carefully ensured that the following invariants always hold for a given thread:

- GlobalClock is T_{max}({causally preceding writes} $\cup$ {all known completed writes}), and
- CompletedWrites is
  T_{max}({all known completed writes}).

The figure below illustrates how a thread can keep track of various sets of writes using vectors.

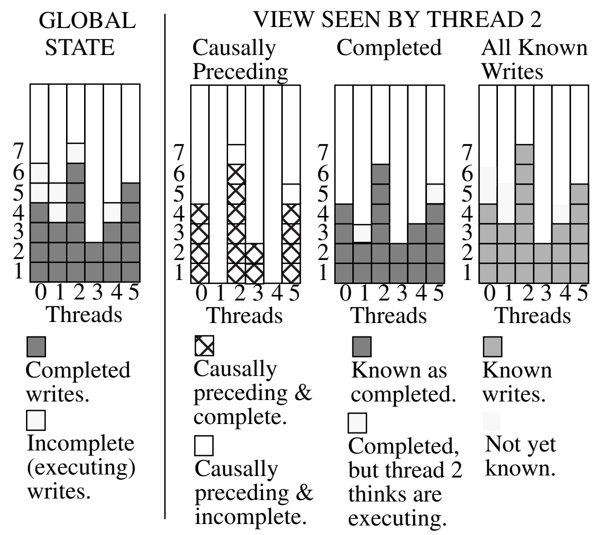

Figure 6: A single thread's estimate of causally preceding, completed and known writes

Note that every causally preceding write is necessarily a known write. This makes sense because causality is defined in terms of knowledge, and a thread's behavior cannot be causally influenced by a write it does not know. The upper boundary of any region gives the T_{max} value of the set it represents. Hence, GlobalClock is the upper boundary of {Causally Preceding} $\cup$ {Known Completed}, which is (4, 2, 7, 2, 3, 5). CompletedWrites is the upper boundary of {Known Completed}, which is (4, 1, 6, 2, 3, 4).

By definition, GlobalClock measures the thread's current time and can be used to partition time into *earlier*, *concurrent* and *later* regions. All *earlier* writes will have timestamps $\leq$ GlobalClock. The three requirements for consistency can now be rewritten thus:

Requirement I: A read R must wait for all writes with timestamps $\leq$ R.timestamp to finish, before beginning.

Requirement II: A write W must allow all writes with timestamps $<$ W.timestamp to complete, before it can finish.

Requirement III: A read R must return a value based solely on writes with timestamps $\leq$ R.timestamp.

These requirements dictate precisely how reads and writes to should execute.

Consider a read initiated by thread 2, given the state in Figure 6. The timestamp of the read would be the value of GlobalClock, namely (4, 2, 7, 2, 3, 5). By Requirement III, this vector precisely identifies the set of writes that will be reflected in the read -- namely 4 writes by thread 0, 2 writes by thread 1 etc. Since this vector is larger than the CompletedWrites vector we know that some causally preceding writes have yet to complete. Hence the read will block (by Requirement I). As these writes complete sync and causal messages will propagate information to thread 2, and the CompletedWrites vector will advance until it crosses the read timestamp. At this point the read can execute. The more frequent the sync messages, the earlier it will be unblocked. The timestamp of the read will be used to interpret the data in the graph, and return a value based on the writes allowed by its timestamp and none other.

If thread 2 were to initiate a write given the situation in the figure, it would increment its local clock (GlobalClock[2]) creating the timestamp, (4, 2, 8, 2, 3, 5). The write will then invalidate the downstream portion of the graph. In order to complete, it must wait for all causally preceding writes to complete. These have timestamps $<$ (4, 2, 8, 2, 3, 5). When information flow, due to sync and causal messages, causes CompletedWrites to advance to a point where it is $\geq$ (4, 2, 7, 2, 3, 5), we know that Requirement II is satisfied, and the write can complete.

Next, we present the algorithm in sufficient detail for implementation.

5 THE DOPPLER ALGORITHM

In Doppler, each user-level thread has an associated "local" constraint system. This consists of:

i) a *context*, which embeds local nodes and contains state information,

ii) the *API* -- a set of procedures and methods to provide access to the functionality, and

iii) an active system thread to periodically exchange sync messages with peer constraint systems

For ease of explanation we assume that remote method invocation can be done transparently. This was the case in the environment we used for our implementation, Obliq [6]. In other environments an explicit RPC mechanism may be necessary.

5.1 Data Structures

Each *context* is an object containing the following fields:

- *index*, an ordinal value to uniquely identify the thread
- *mutex*, a lock to provide mutual exclusion (can only be accessed locally)
- *GlobalClock*, a time-vector (as explained in 4.3.4)
- *CompletedWrites*, a time-vector (as in Section 4.3.4)

Each *node* is an object with the same fields as the base algorithm node, but instead of the *value* and *out-of-date* fields, it has a field called *parset. Parset* is a partially ordered set of tuples of the form <val, timestamp>, wherein *val* is either a normal value or the special constant, OOD (standing for out of date). Parset entries are partially ordered by *timestamp*.

In addition each node has:

- *context*, a link to the local context
- *mutex*, a lock for mutual exclusion.

5.2 Executing Writes and Reads

In Copernicus and Kepler, no writes were in progress when a read was being executed. Hence the operations proceeded as in the sequential algorithm. Here we have a graph that changes as the read happens. To cope with concurrency we modify the execution of the base algorithm writes and reads.

5.2.1 Writes. A **write** invalidates the downstream portion of the graph as in the base algorithm. However it no longer merges its OOD marks with those of previous writes, because: (i) timestamps are often incomparable, and (ii) even if they are comparable, there may be an intervening, concurrent read. Instead it adds a new OOD mark to the node, along with its timestamp. This is done by adding the tuple <OOD, timestamp> to parset. The write then waits for all *strictly earlier* writes to finish (as per requirement II), and then completes. This causes the node to acquire a set of OOD marks, tagged by write initiation time. Subsequent reads turn some OOD marks into values. The entries of parset are partially ordered by timestamp.

Figure 7 illustrates a sequence of writes by two threads.

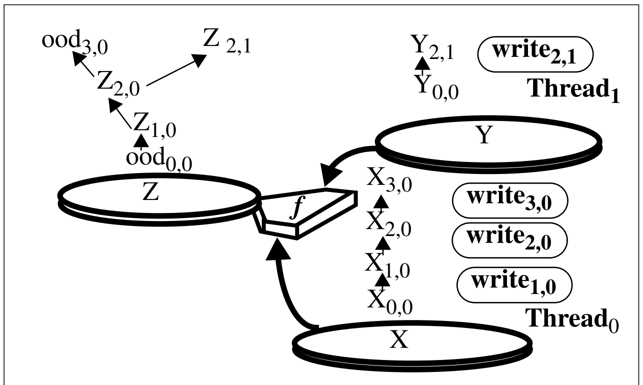

Figure 7: Concurrent writes by two threads

$Thread_0$ writes to X, which lies in its context, and similarly $Thread_1$ writes to Y. Initially GlobalClock is (0,0) in all contexts (we ignore the entries corresponding to other threads). The parsets of leaf nodes X and Y have the values $X_{0,0}$ and $Y_{0,0}$ respectively, and Z is initially out-of-date and contains the OOD mark $ood_{0,0}$. $Thread_0$ writes the values $X_{1,0}$, $X_{2,0}$ and $X_{3,0}$ to X in succession. In each case GlobalClock[0] is incremented and the GlobalClock value is used to timestamp the write. The writes propagate the OOD marks $ood_{1,0}$, $ood_{2,0}$ and $ood_{3,0}$ respectively to Z, and complete in the order of initiation (since they are causally ordered). The OOD marks $ood_{1,0}$ and

$ood_{2,0}$ are refreshed to yield the values $Z_{1,0}$ and $Z_{2,0}$ by reads (not shown). $Thread_1$'s GlobalClock advances to (2,0), either due to a causal communication or because of a sync message (not shown). It then writes $Y_{2,1}$ to Y which propagates the OOD mark $ood_{2,1}$ to Z. This is refreshed by a read (not shown) to yield the value $Z_{2,1}$.

The code below indicates how user level writes are performed.

```
method write(n, v)
    let nc = n.context;
    (* Get earlier writes info and a timestamp *)
    lock nc.mutex do
      (* GlobalClock is Tmax({earlier writes})*)
      let earlier_writes = nc.GlobalClock;
      let tstamp = increment(nc.GlobalClock);
    end; (* lock is released *)
    (* start background thread to do write *)
    thread_fork(writeAt(n, nc, earlier_writes,
          tstamp, v));
end;
```

The background write, `writeAt`, calls `markOODAt` to invalidate downstream nodes, and waits for earlier writes to complete before terminating.

```
proc writeAt(n, nc, earlier_writes, tstamp, v)
    (* add value and invalidate dependent nodes *)
    lock n.mutex do
      append(n.parset, <v , tstamp>);
      let dependents = copy(n.outlist);
    end; (* lock is released *)
    foreach nd in dependents do
      nd.markOODAt(tstamp);
    end;
    (* Wait for earlier writes to finish *)
    await(nc.CompletedWrites ≥ earlier_writes);
    lock nc.mutex do
      nc.CompletedWrites :=
        Tmax(nc.CompletedWrites, tstamp);
    end; (* lock is released *)
end;
```

The method, `markOODAt`, adds `<OOD, tstamp>` to the node's parset, and recursively invalidates nodes in `outputlist`.

```
method markOODAt(n, tstamp)
    var already_OOD = true;
    lock n.mutex do
      (* Does this timestamp already exist? *)
      if <_ , tstamp> ∉ n.parset then
        append(n.parset, <OOD, tstamp>);
        let dependents = copy(n.outputlist);
        already_OOD := false;
      end;
    end; (* lock is released *)
    if not already_OOD then
      foreach nd in dependents do
        nd.markOODAt(tstamp);
      end;
    end;
end;
```

5.2.2 Reads. A **read** first waits for all *earlier* writes to complete (to satisfy requirement I). Then it decides on the set of writes it will base its value on. By requirement III this set should include all *earlier* writes and none other. Hence all writes with timestamps $\leq$ the read's timestamp are relevant. Let the read-timestamp be T. The user-level `read` method calls a system-level `readAt` method to compute the value at T.

As in the case of a base algorithm read, `readAt` needs to first check if the value at T is already present in the node's parset, possibly with an *earlier* timestamp. Here is how we check if an applicable value, T_{appl}, exists:

> Note that $T_{appl} \leq T$ ... (i)
>
> Let *earlier-set* be the set of timestamps corresponding to the writes that have invalidated the node *before* T. Every such write would have an entry in parset. Let T_{after} represent the *earliest* point in time *after* all the timestamps in earlier-set. We compute T_{after} as T_{max}(earlier-set). If there exists an applicable value in the set it must be timestamped *after* T_{after}, in order to reflect all the writes in earlier-set.
>
> Hence $T_{appl} \geq T_{max}$(earlier-set) ... (ii)
>
> However no time-vector in the *earlier-set* can be *after* T_{after}, by definition. Given conditions (i) and (ii), it is clear that only a value at T_{max}(earlier-set) will suffice.

Hence, if an applicable value exists it must necessarily have the timestamp T_{after}. If `readAt` finds such a value it returns it.

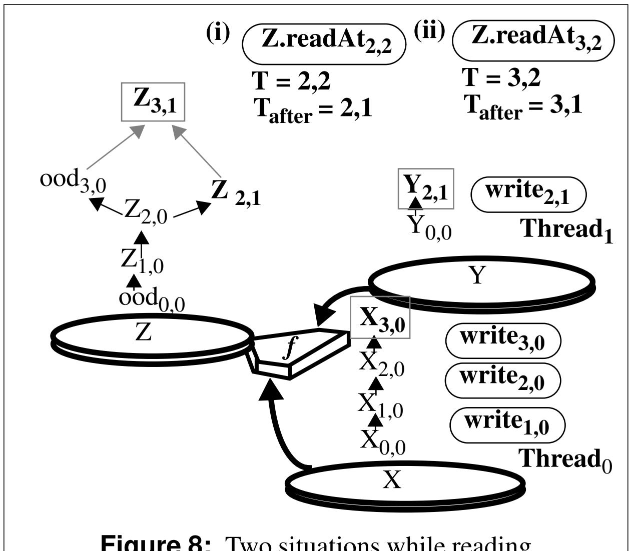

Figure 8: Two situations while reading

This is illustrated in case (i) in Figure 8. When `readAt` is invoked with the argument (2,2), it first computes the timestamps in its earlier-set. This set should contain all writes *before* (2,2), namely {(0,0), (1,0), (2,1)}. T_{after} is computed as (2,1). There is a value at T_{after}, namely $Z_{2,1}$. This is returned.

When a value at T_{after} is not available, `readAt` reads all the input nodes. In each case it invokes the `readAt` method at T_{after}. The values thus computed are used as arguments to the constraint function. A new value is computed at T_{after}. This is added to the parset, and replaces an OOD mark if present.

Observe that returning a value with a timestamp *strictly less* than T_{after} is not legal. This is shown in case (ii) in Figure 8. The earlier-set of $readAt_{3,2}$ is {(0,0), (1,0), (2,1), (3,0)}. T_{after}, the earliest point in time *after* the writes in earlier-set is (3,1). No value exists at (3,1). The value at (2,1) is not legal because it does not reflect the write at (3,0). Hence the value at (3,1) needs to be explicitly computed. Recursively reading the input nodes, X and Y, at (3,1) yields $X_{3,0}$ and $Y_{2,1}$ respectively. These are used to compute the value of Z at (3,1): $Z_{3,1} = f(X_{3,0}, Y_{2,1})$. $Z_{3,1}$ is added to Z's parset, and returned as the result.

```
method read(n)
    let nc = n.context;
    lock nc.mutex do
      let tstamp = nc.GlobalClock;
    end; (* lock is released *)
    await(nc.CompletedWrites ≥ tstamp);
    return(n.readAt(tstamp));
end;

method readAt(n, tstamp)
    var retval, valpresent = false;
    lock n.mutex do
      let TMAX = Tmax({ t | <_ , t> ∈ n.parset
          AND t <= tstamp });
      if ∃ v, (< v, TMAX > ∈ n.parset AND
          v ≠ OOD) then
          retval := v;
          valpresent := true;
      else
        let inputs = copy(n.inputlist);
      end;
    end; (* lock is released *)
    if not valpresent then
      var args = [];
      for i in first(inputs) to last(inputs) do
        append(args, inlist[i].readAt(TMAX));
      end;
      retval := n.constraint-func(args);
      lock n.mutex do
        (* This replaces an existing entry at *)
        (* TMAX if there is one, else appends *)
        addvalue(n.parset, <retval, TMAX> );
      end; (* lock is released *)
    end;
    return(retval);
end;
```

5.3 Causal and Sync Messages

All user-level communication between threads is accompanied by *causal* messages. Round-trip communication (as in a remote procedure call) will be accompanied by two causal messages -- one in each direction. These messages carry GlobalClock and CompletedWrites vectors to ensure that the constraint system at the receiving end is aware of all causally preceding writes.

At the receiving end, the constraint system sets GlobalClock to T_{max}(GlobalClock, $\text{GlobalClock}_{peer}$), and CompletedWrites to T_{max}(CompletedWrites, $\text{CompletedWrites}_{peer}$).

Each thread periodically sends *sync* messages to its peers. Sync messages carry only the CompletedWrites vector. When a sync is received, GlobalClock is set to T_{max}(GlobalClock, $\text{CompletedWrites}_{peer}$), and CompletedWrites to T_{max}(CompletedWrites, $\text{CompletedWrites}_{peer}$).

The pattern and frequency of sync messages will depend on the application. One option is to broadcast sync messages to all peers. This increases network traffic and is wasteful, since sync messages are transitive and the same information will be conveyed along many redundant routes. Instead a round-robin scheme could be employed.

The number of sync messages used to convey information about a completed write to a certain peer will determine the write propagation time, and hence the lag between the two threads. In user interface applications, we would like minimize the lag between: (i) actively cooperating threads, and (ii) threads that serve the same user. Such threads could be made to broadcast (or multicast) sync messages to each other, while communicating with the rest of the threads in a round-robin fashion. Whenever shared memory is available, e.g. in the case of threads within the same application, communication time can be greatly reduced.

5.4 Garbage Collection

If the above algorithm were implemented as described, efficiency would rapidly decline as writes are executed, due to the increase in the size of node parsets. The read and write methods take time linear in the size of the parset, at each node that is examined. This also increases the amount of storage required. Luckily, values in parsets rapidly become stale and can be periodically garbage collected. To facilitate this, constraint systems maintain *MinReadTime*, a conservative estimate of the earliest active read in the system.

To keep track of MinReadTime, the threads use a gossip scheme to maintain a conservative estimate the timestamp of the earliest pending/potential read in each of the threads. This is stored as a vector of read timestamps, indexed by thread number, called *EarliestRead*. When thread i executes a read it will set its EarliestRead[i] to the read-timestamp. If there is no read active, the entry is set to GlobalClock, to account for a potential read in the future. The EarliestRead vector is sent with all communication (namely causal and sync messages). When a peer's EarliestRead vector is received, the thread sets its EarliestRead[k] to T_{max} (EarliestRead[k], EarliestRead$_{peer}$[k]), for all k.

MinReadTime is computed as T_{min} of all the elements in EarliestRead. T_{min} is analogous to T_{max}. The operator computes the latest time-vector which is *before* all the elements in its argument set.

In the beginning of the `readAt` and `markOODAt` methods, we add code to check if the size of the parset has exceeded a certain threshold. If so the `garbageCollect` method is called.

The `garbageCollect` method deletes all parset entries less than MinReadTime, and leaves behind exactly one entry at MinReadTime to handle future reads. It computes this entry by doing the same calculation that a read would do. It computes T_{max} of entries less than MinReadTime, and sees if there is an entry with that timestamp. If there is one it is used; otherwise <OOD, MinReadTime> is used.

```
(* System Level Garbage Collection Method *)
(* Complexity = O( SIZE(n.parset) ) *)
method garbageCollect(n)
    let nc = n.context;
    lock nc.mutex do
        let MRT = nc.MinReadTime;
    end; (* lock is released *)

    lock n.mutex do
      let prior= {t| <_ , t> ∈ n.parset &
                 t ≤ MRT};
      let Tafter = Tmax(prior);
      if ∃<v, Tafter> in prior then
        n.parset := n.parset - prior ∪
                   {<v, MRT>};
      else
        n.parset := n.parset - prior ∪
                  {<OOD, MRT>};
      end;
    end; (* lock is released *)
end;
```

5.5 Notification Callbacks

A notification callback is invoked when a write changes the value of the node it is attached to. Specifically, it is invoked by the constraint system of the thread that owns the node when: (i) the write in question is known to have completed, and (ii) the value is seen to have changed. To facilitate this, writes check to see if the node they are invalidating (or writing to) has an associated notification callback. If there is one, the write adds the node to a list of pending notifications, called *PendingNotifies* (in the node's context), along with the timestamp of the write. Whenever the CompletedWrites vector is updated, the constraint system checks to see if CompletedWrites is *after* any of the timestamps in PendingNotifies. If so, the entry is dequeued, the node is read (at GlobalClock), and the new value is compared with the value computed during the last notification. If there is a change, the callback is executed. Since reads by the same thread are totally ordered, the previous value is well defined, and conventional semantics apply.

6 DYNAMIC GRAPHS

Now we extend the algorithm described in Section 5, to dynamic graphs. These are graphs whose structure changes concurrently with the reads and writes in the system.

The operations, **`CreateLeaf(v, cbk)`** and **`DelLeaf(n),`** are trivial to implement because they are local.

6.1 Adding a constraint

The operation **`n.addConstraint(cfunc, nlist)`** adds a constraint to leaf node **`n`** making it an interior node. The non-blocking **`addConstraint`** operation resembles a write, and is executed thus:

1. The context **`nc`** is locked while **`nc.GlobalClock`** is incremented and a new timestamp, **`tstamp`**, is obtained as in the case of a write. The lock is released. The rest of the operation is executed by a background thread.
2. The method **`n.markOODAt(tstamp)`** is invoked.
3. The thread waits for all *strictly earlier* writes to complete.
4. The node **`n`** is locked and **`cfunc`** and **`nlist`** are stored as the constraint and input-list respectively. The **`n.markOODAt`** method is configured so that only marks *after* **`tstamp`** are propagated. The lock on **`n`** is released. The last step prevents the spreading of marks at times which are *concurrent* with **`tstamp`** via this route once input links are added. Timestamps *before* tstamp will not occur because all *earlier* writes have completed. Timestamps *after* **`tstamp`** may be circulating in the system but these are taken care of in step 5. These timestamps are caused by the transmission of **`GlobalClock`** on causal messages.
5. An edge is added from each node, **`nd`**, in **`nlist`** to **`n`**. This is done by locking **`nd`**, adding **`n`** to its output-list, and releasing the lock before considering the next node. In the process any timestamps in **`nd`**'s parset, which are *after* **`tstamp`**, are noted. For each such timestamp, **`ts`**, **`n.markOODAt(ts)`** is executed. This makes it appear as if these edges were created atomically at **`tstamp`**, although they were added later.
6. The context **`nc`** is locked and **`nc.CompletedWrites`** is set to **`Tmax(nc.CompletedWrites, tstamp)`** signifying that the operation is over. The context is unlocked. Note that until this step is complete, the node is treated as a leaf, node when it is read. There would have been no reads *after* **`tstamp`** during this period, because any such read would have needed to wait for this operation to complete before beginning. Hence there is no inconsistency.

6.2 Deleting a constraint

The **`n.delConstraint`** operation deletes the constraint attached to the interior node **`n,`** making it a leaf. The non-blocking **`delConstraint`** operation resembles a write:

1. The context **`nc`** is locked. **`nc.GlobalClock`** is incremented and a new timestamp, **`tstamp`**, is obtained.
2. The node **`n`** is locked and the **`n.markOODAt`** method is

configured so that only marks *before* `tstamp` are propagated. Both locks are released. A background thread is forked to execute the rest of the operation.

3. The thread waits for all *strictly earlier* writes to complete.
4. Each node, `nd`, in `n.inputlist` is considered in turn. `nd` is locked and `n` is removed from `nd`'s output-list. Then the lock is released and the next node is considered. Note that the nodes are still in `n.inputlist` since some reads *before* `tstamp` may still need to be satisfied. With respect to all other reads, it behaves like a leaf node.
5. The node `n` is locked. All OOD marks *after* `tstamp` are deleted from the `n.parset`. The node is unlocked.
6. The context `nc` is locked and `nc.CompletedWrites` is set to `Tmax(nc.CompletedWrites, tstamp)` signifying that the operation is over. The context is unlocked.

At this point `n` still has references to its input nodes, because there may be some OOD marks which are *before* `tstamp`. Since the input nodes no longer have references to `n`, no more OOD marks will appear. If necessary, the input-list can be deleted when the last OOD mark is refreshed (or garbage collected).

7 ANALYSIS

7.1 Storage

The number of threads in the system may not be known a priori. Fortunately we do not need arbitrarily large timestamp vectors. Elements corresponding to future threads will have entries set to zero. These do not need to be maintained and vectors can be expanded as needed. Since the number of unique timestamps inside a context is quite small, they can be stored in a hash table and referenced indirectly. This simplifies garbage collection.

The number of useful entries in a node's parset is bounded as well. It equals the number of writes initiated since MinReadTime. Typically this is bounded by the number of concurrent reads and writes active in the system. The only case in which this bound is exceeded is when a read takes a long time to complete, allowing other threads to perform multiple writes while it holds MinReadTime down.

7.2 Concurrency

Doppler provides provably optimal *concurrency* for any distributed constraint maintenance algorithm that delivers causal consistency. This is because:

- Writes execute concurrently with all other operations.
- Reads execute concurrently with all operations except causally preceding writes, which they need to wait for. *This is unavoidable in any causal implementation.*

In the absence of inter-thread communication the only writes that an operation waits for are previous writes in the same thread. This gives us full parallelism at the operation level. Each thread executes operations in virtual isolation, without blocking for other threads. When there is inter-thread communication, threads need to synchronize. Consequently subsequent reads may block for causally preceding writes.

Operation execution times are increased by the need to do O(no. of threads) book-keeping at every node. It is expected that this cost will be small compared to the cost of inter-thread/inter-address-space communication and constraint evaluation.

Writes execute in the background and do not cause user threads to block. Reads do. In Kepler, a read needs to wait for all previous writes. As the number of threads increases, the performance deteriorates. In Doppler, the delay faced by a read depends not on the number of threads, but on the amount of causality in the system (due to inter-thread communication). Hence the algorithm scales well.

Concurrency improves performance by reducing the execution times of reads and writes. Both operations need to block until they are notified of the completion of causally preceding writes. In the case of causally preceding writes by the same thread, the waiting operation is notified the moment the write completes. The delay suffered by a read due to a causally preceding write by another thread will depend on the speed of the sync mechanism. Note that we can always do better than in Kepler, by broadcasting sync messages whenever a write completes, thus reducing the write propagation time to a single message transmission. In Kepler (and in any synchronous implementation) it would take at least two messages to transfer the lock from a writing thread to a blocked read, for instance.

For a small number of threads, the performance should be about the same in both cases. However, as the number of threads increases, there will be increased contention for the graph in the case of synchronous algorithms like Kepler. Threads will usually block for operations in the previous phase to complete. In Doppler, concurrent operations can execute without waiting for each other, unless they are causally related. Hence performance will not degrade as rapidly.

7.3 Transactions

In Kepler and in all previous lazy algorithms, the execution time of writes was reduced by having writes stop when they encountered an OOD mark by a previous write. In Doppler, this is not typically possible because there could be an intervening read. However, when a set of writes need to be executed together, it is efficient to execute them with a shared timestamp, as a *write transaction*. This prevents them from invalidating the same nodes multiple times. The last write to finish updates CompletedWrites.

A set of reads with a common timestamp forms a *read transaction*. The operation waits for all causally preceding writes to complete and then issues multiple reads, possibly in parallel. Since they share a common timestamp waiting time is amortized over all the reads. After the reads complete, the EarliestRead entry is updated, ending the transaction. The reads in a read transaction will return consistent values, corresponding to a point in partially-ordered time. This is useful when we wish to take a consistent snapshot of a portion of the graph, say for checkpointing or debugging.

8 IMPLEMENTATION

The Doppler algorithm was implemented and tested in Obliq[6], a multi-threaded and distributed scripting language with transparent access to remote objects. Our implementation takes about 1000 lines of Obliq code. We have also built a few test applications to observe the impact of various sync protocols on perceived user performance. In the future we hope to extend the algorithm to multi-way constraints.

9 CONCLUSION

This paper has described Doppler, a distributed and concurrent algorithm for update of one-way constraints which opens the use of constraint systems to a wide range of distributed and collaborative applications. Doppler provides a very high degree of concurrency and works in a completely asynchronous manner without the need for shared memory, shared or synchronized clocks, or centralized locking. These properties will allow the algorithm to scale to large numbers of users as well as take advantage of concurrent processing resources.

We gratefully acknowledge the contribution of Mustaque Ahamad in the development of this algorithm.

10 REFERENCES

[1] Ahamad, M., Neiger, G., Burns. J.E., Hutto, P.H., and Kohli, P., "Causal Memory: Definitions, Implementations and Programming", To appear in *ACM Transactions on Distributed Computing Systems*.

[2] Barford, L. A., and Vander Zanden, B. T., "Attribute Grammars in Constraint-Based Graphics Systems", *Software - Practice and Experience*, 19:4, 1989, pp. 309-328.

[3] Birman,K., Schiper, A., and Stephenson, P., "Lightweight Causal and Atomic Group Multicast", *ACM Transactions on Computing Systems*, 9:3, 1991, pp. 272-314.

[4] Borning, A., "The Programming Language Aspects of ThingLab: A Constraint-Oriented Simulation Laboratory", *ACM Transactions on Programming Languages and Systems*, 3:4, Oct. 1981, pp. 353-387.

[5] Borning, A., and Duisberg, R., "Constraint-Based Tools for Building User Interfaces", *ACM Transactions on Graphics*, 5:4, Oct. 1986, pp. 345-374.

[6] Cardelli, L., "A Language with Distributed Scope", *Proc. of the 22nd ACM Symposium on Principles of Programming Languages*, Jan 1995, pp. 286-297.

[7] Freeman-Benson, B. N., Maloney, J., and Borning. A. "An Incremental Constraint Solver", *Communications of the ACM*, 33:1, Jan. 1990, pp. 54-63.

[8] Freeman-Benson, B. N., "Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming", *Proceedings of OOPSLA/ECOOP '90*, 1990, pp. 77-88.

[9] Gleicher, M., "A Graphics Toolkit Based on Differential Constraints", *Proceedings of UIST '93*, Atlanta, GA, Nov. 1993, pp. 109-120.

[10] Henry, T. R., Hudson, S. E., "Using Active Data in a UIMS", *Proceedings of UIST '88*, Banff, Alberta, Canada, Oct. 1988, pp. 167-178.

[11] Hill, R. D., "The Rendezvous Constraint Maintenance System", *Proceedings of UIST '93*, Atlanta, GA, Nov. 1993, pp. 225-234.

[12] Hudson, S. E., and King, R., "Semantic Feedback in the Higgens UIMS", *IEEE Transactions on Software Engineering*, 14:6, Aug. 1988, pp. 1188-1206.

[13] Hudson, S. E., "Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update", *ACM Transactions on Programming Languages and Systems*, 13:3, July 1991, pp. 315-341.

[14] Hudson, S. E., "A System for Efficient and Flexible One-Way Constraint Evaluation in C++", Georgia Institute of Technology - Graphics, Visualization, and Usability Center Technical Report GIT-GVU 93-15, April 1993.

[15] Hudson, S. E., "User Interface Specification Using an Enhanced Spreadsheet Model", *ACM Transactions on Graphics*, 13:3, July 1994, pp. 209-239.

[16] Lamport, L., "Time, Clocks and the Ordering of Events", *Communications of the ACM*, 21;7, July 1978, pp. 558-565.

[17] Manasse, M.S., and Nelson, G., "Trestle Reference Manual", *Digital Systems Research Center - Technical Report 68*, Palo Alto, CA, Dec 1991.

[18] Myers, B. A., Giuse, D. A., Dannenberg, R. B., Vander Zanden, B., Kosbie, D.S., Pervin, E., Mickish, A., and Marchal, P., "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces", *IEEE Computer*, 23:11, Nov. 1990, pp. 71-85.

[19] Takahashi, S., Matsuoka, S. Yonezawa, A., and Kamada, T., "A general framework for bi-directional translation between abstract and pictorial data", *Proceedings of UIST'91*, Hilton Head, South Carolina, Nov 1991.

[20] Tang, S. H., and Linton, M. A., "Blending Structured Graphics and Layout", *Proceedings of UIST '94*, Marina del Rey, CA, Nov. 1994, pp. 167-174.

[21] Vander Zanden, B., Myers, B.A., Giuse, D., Szekely, P.,"Integrating Pointer Variables into One-Way Constraint Models", *ACM Transactions on Computer Human Interaction*, 1:1, June 1994.